Assignment 1 - EN2550

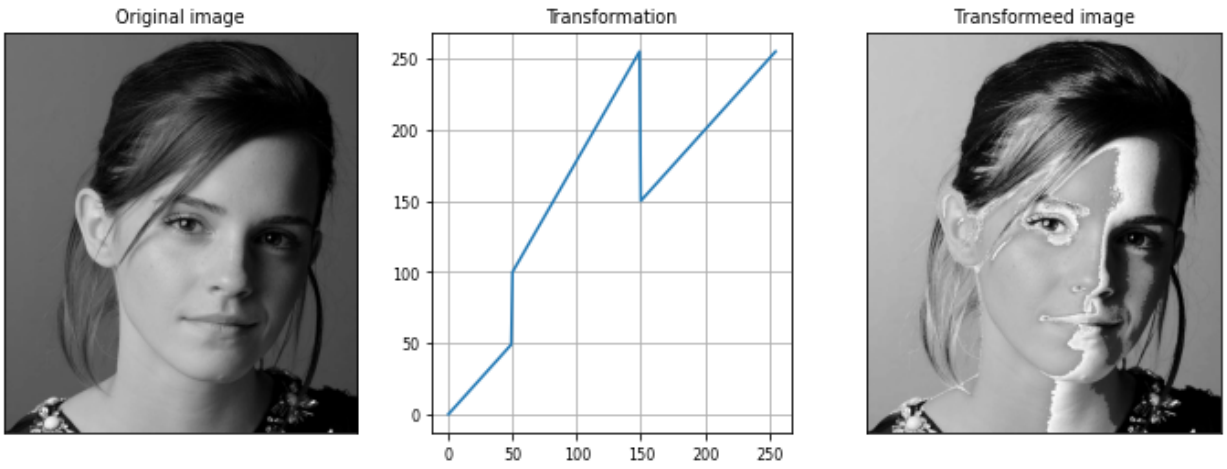
190456K

Question 1

- Created the transformation by horizontally stacking three numpy arrays

```
t1 = np.arange(0, 50, 1)
t2 = np.linspace(100, 255, 100)
t3 = np.arange(150, 256, 1)
t = np.hstack((t1, t2, t3))
```

- Then used the `cv2.LUT` (Look up table) function to transform the image

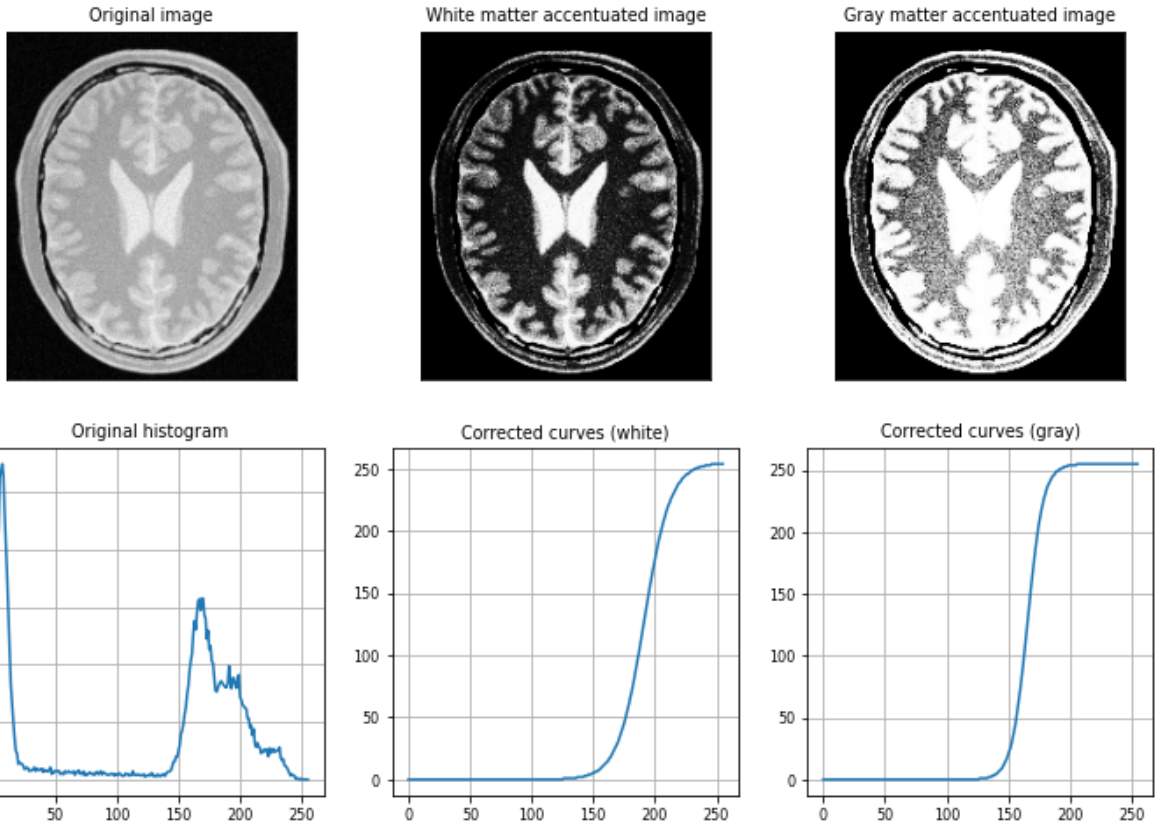


Question 2

- Used a transformed `tanh` function to do accentuate the required parts

```
def tanh_f(x):
        num = np.exp(x) - np.exp(-x)
        den = np.exp(x) + np.exp(-x)
        return num/den

def accentuating_transformation(center, width):
        center = center
        width = width
        x = np.linspace(center-width, center+width, 256)
        tanh = tanh_f(x)
        max_val = np.max([np.max(tanh), np.abs(np.min(tanh))])
        scaled_f = tanh/max_val*127.5
        shifted_f = scaled_f + 127.5
        quantized_f = np.round(shifted_f).astype(np.uint8)
        return quantized_f
```

- By applying these transformations, got the following parts enhanced



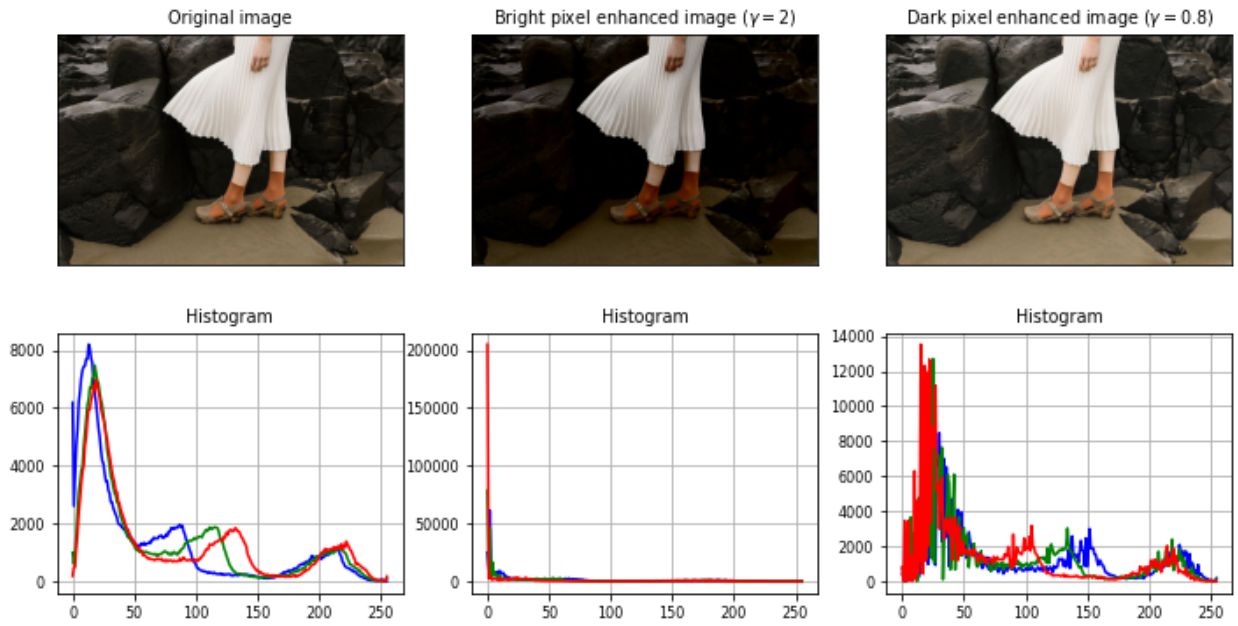
Question 3

- The L channel is what determines the luminosity of the image (also known as the Luma channel in graphic designing)
- Hence, by tweaking the Luma channel, the pixel values will be mapped to different ranges based on the **overall brightness** relevant to that pixel
- Used the following gamma transforms

```
gamma_light = 2
t_light = np.array([(p/255)**gamma_light*255 for p in range(256)]).astype(np.uint8)
gamma_dark = 0.8
t_dark = np.array([(p/255)**gamma_dark*255 for p in range(256)]).astype(np.uint8)
```

- Then converted the image to the LAB color space and applied the above transformations to the Luma channel

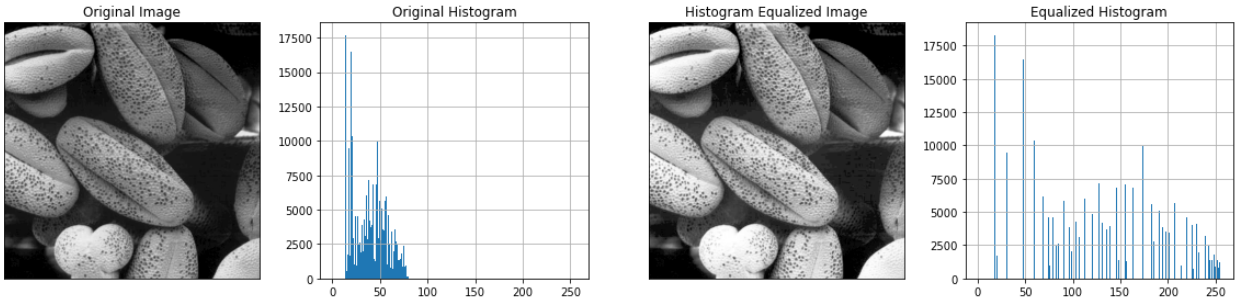
```
LAB_light_img = cv.cvtColor(inp_img, cv.COLOR_BGR2LAB)
LAB_light_img[:,:,0] = cv.LUT(LAB_light_img[:,:,0], t_light)
```



Question 4

- Used the following function

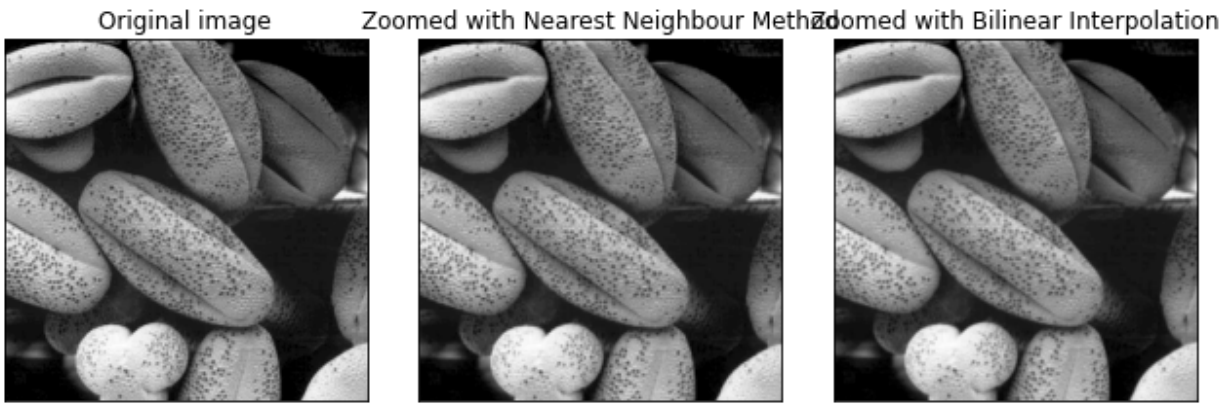
```
def getHistBalImg(img):
        '''
        Takes any input image channel from the user and
        return the channel by equalizing the histogram
        '''
        L = 256
        MN = img.size
        return_img = np.zeros(img.shape, dtype='uint8')
        intensity_bins = np.zeros(L)
        for i in range(L):
                intensity_bins[i] = img[img==i].size
        cum_intensity_bins = intensity_bins.cumsum()
        scaled_cum_intensity_bins = (L-1)/MN*cum_intensity_bins
        return_img = scaled_cum_intensity_bins[img]
        return return_img
```



Question 5

- Used the following function which handles both Nearest Neighbour Method and Bilinear Interpolation Method, and all worked fine

```
def zoomImg(inp_img, scale, zoom_type):
    '''
    zoom_type may take the two values 0 and 1
        0: Will zoom with nearest neighbour method
        1: Will zoom with bilinear interpolation
    '''
    rows = inp_img.shape[0]*scale
    cols = inp_img.shape[1]*scale
       out_img = np.zeros((rows, cols))
    if (zoom_type == 0):
        for i in range(rows):
            for j in range(cols):
                out_img[i][j] =
inp_img[np.round(i/scale).astype(int)-1][np.round(j/scale).astype(int)-1]
    else:
        # p1      p2
        # x1   x  x2
        #
        # p3      p4
        for i in range(rows):
            for j in range(cols):
                p1 = inp_img[int(i/scale)][int(j/scale)]

                p2 = inp_img[int(i/scale)][int(j/scale)]
                if int((i/scale)+1) < rows/scale:
                    p2 = inp_img[int(i/scale)+1][int(j/scale)]

                p3 = inp_img[int(i/scale)][int(j/scale)]
                if int((j/scale)+1) < cols/scale:
                    p3 = inp_img[int(i/scale)][int(j/scale)+1]

                p4 = inp_img[int(i/scale)][int(j/scale)]
                if  int((j/scale)+1) < cols/scale and int((i/scale)+1) < rows/scale:
                    p4 = inp_img[int(i/scale)+1][int(j/scale)+1]

                x1 = p1*(int(j/scale)+1-j/scale)+p3*(j/scale-int(j/scale))
                x2 = p2*(int(j/scale)+1-j/scale)+p4*(j/scale-int(j/scale))
                x = x2*(i/scale - int(i/scale))+x1*(int(i/scale)+1-i/scale)

                out_img[i][j] = x
        out_img = np.round(out_img).astype(int)
    return out_img
```



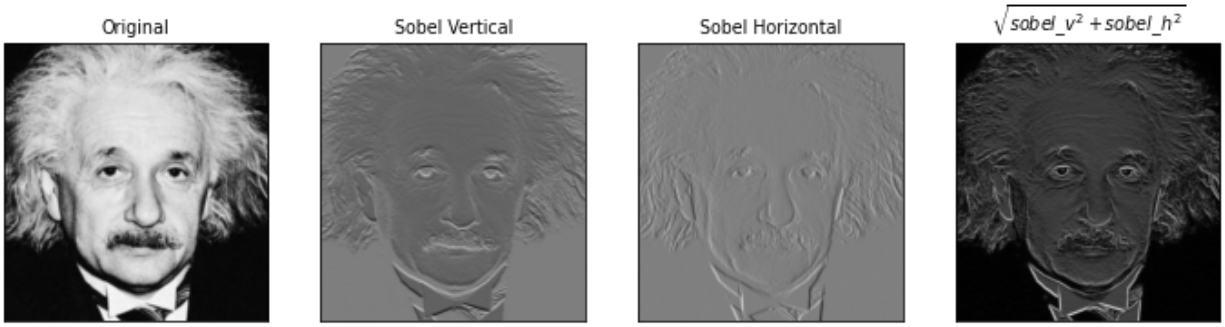
Question 6

(a)

- The casual method,

```
sobel_v_kernel = np.array([[-1, -2, -2], [0, 0, 0], [1, 2, 1]], dtype=np.float32)
img_sobel_v = cv.filter2D(img, -1, sobel_v_kernel)
sobel_h_kernel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)
img_sobel_h = cv.filter2D(img, -1, sobel_h_kernel)
img_grad = np.sqrt(img_sobel_v**2 + img_sobel_h**2)
```

- Resulted following



(b)

- Defined the following function, which resulted in the same

```
def sobelFilt(inp_img, v_h):
    '''
    v_h can take two values
        0: vertical sobel kernel
        1: horizontal sobel kernel
    '''
```

```
    sobel_kernel = np.zeros((3,3))
    if v_h == 0:
        sobel_kernel = np.array([[-1, -2, -2], [0, 0, 0], [1, 2, 1]],
dtype=np.float32)
    elif v_h == 1:
        sobel_kernel = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]],
dtype=np.float32)
    else:
        raise Exception("Invalid v_h parameter")
    arr = inp_img
    zero_row = np.zeros(arr.shape[1]+2)
    zero_col = np.zeros((arr.shape[0], 1))
    zero_padded = np.vstack((zero_row, np.hstack((zero_col, arr, zero_col)),
zero_row))
    out_img = np.zeros(inp_img.shape)
    for i in range(inp_img.shape[0]):
        for j in range(inp_img.shape[1]):
            out_img[i][j] = np.sum(np.multiply(zero_padded[i:i+3,
j:j+3],sobel_kernel))
    return out_img
```

(c)

- Defined the following function which reduced the computational time considerably. Which also resulted in the same

```
def zeroPad(inp_arr):
    arr = inp_arr
    zero_row = np.zeros(arr.shape[1]+2)
    zero_col = np.zeros((arr.shape[0], 1))
    zero_padded = np.vstack((zero_row, np.hstack((zero_col, arr, zero_col)),
zero_row))
    return zero_padded

def sobelFiltOptim(inp_img, v_h):
    '''
    v_h can take two values
        0: vertical sobel kernel
        1: horizontal sobel kernel
    '''
    sobel_1 = np.zeros((3,1))
    sobel_2 = np.zeros((1,3))
    if v_h == 0:
        sobel_1 = np.array((1,0,-1)).reshape((3,1))
        sobel_2 = np.array((1,2,1)).reshape((1,3))
    elif v_h == 1:
```

```
        sobel_1 = np.array((1,2,1)).reshape((3,1))
        sobel_2 = np.array((1,0,-1)).reshape((1,3))
    else:
        raise Exception("Invalid v_h parameter")
    zero_padded = zeroPad(inp_img)
    out_img = np.zeros(inp_img.shape)
    for i in range(inp_img.shape[0]):
        for j in range(inp_img.shape[1]):
            out_img[i][j] = np.sum(np.multiply(zero_padded[i, j:j+3],sobel_1))
    zero_padded = zeroPad(out_img)
    for i in range(inp_img.shape[0]):
        for j in range(inp_img.shape[1]):
            out_img[i][j] = np.sum(np.multiply(zero_padded[i:i+3, j],sobel_2))
    return out_img
```

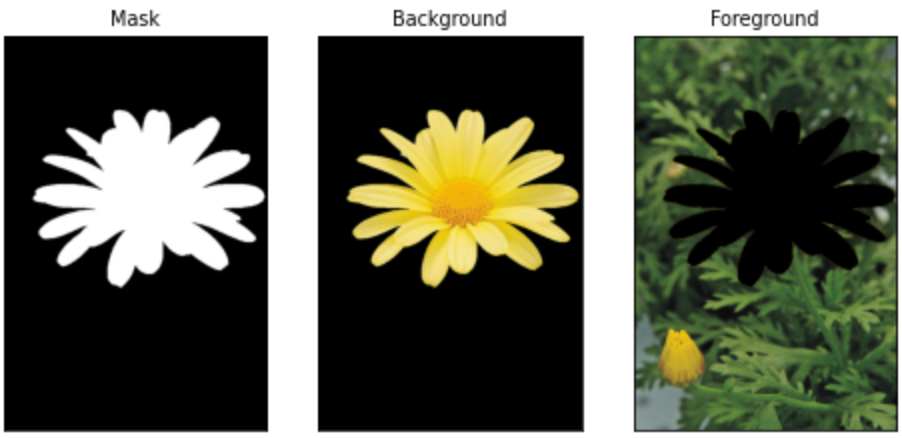
Question 7

(a)

- Used the following code

```
    mask = np.zeros(inp_img.shape[:2],np.uint8)
    bgdModel = np.zeros((1, 65), np.float64)
    fgdModel = np.zeros((1, 65), np.float64)
    rect = (56, 154, 500, 384)
    cv.grabCut(inp_img, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)
    mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8') # foreground
    mask3 = np.zeros(mask2.shape) # background
    mask3[mask2 == 0] = 1
    inp_img = cv.cvtColor(inp_img, cv.COLOR_BGR2RGB)
    img_foreground = inp_img*mask2[:,:,np.newaxis].astype(np.uint8)
    img_backgound = inp_img*mask3[:,:,np.newaxis].astype(np.uint8)
```

- 5 iterations we enough to get a decent output



(b)

- Blurred the background, applied the background mask again, added to the foreground

```
blured_background = cv.GaussianBlur(img_backgound, (15,15), 10)
enhanced_img = blured_background*mask3[:,:,np.newaxis].astype(np.uint8)
enhanced_img += img_foreground
```

- Yielded the following

(c)

- Because when blurring, the black pixels in the background gets also added to the neighboring pixels (in this case, the boundary pixels of the background and foreground)