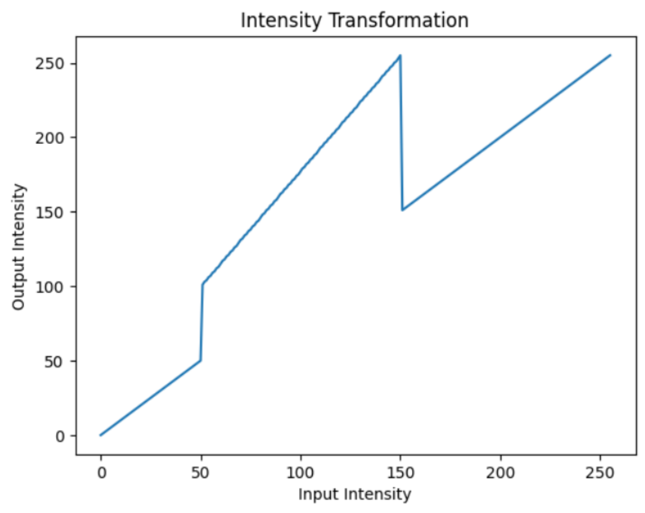
EN3160 – Assignment 01

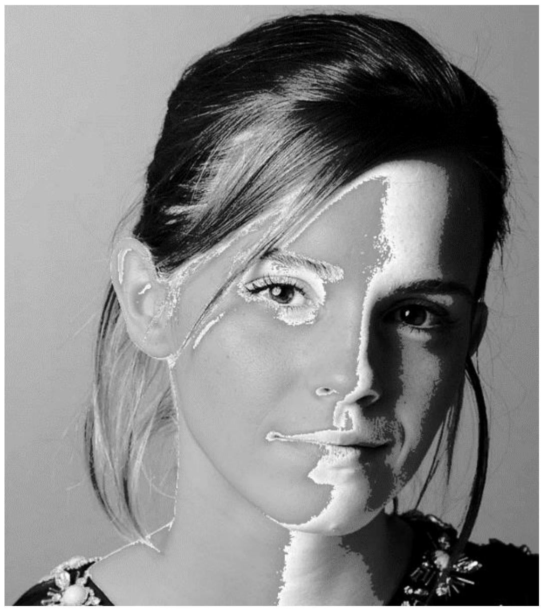
Gunawardena M.N **200201V**

<u>Question 1</u>

```
x1 = np.arange(51)
x2 = np.arange(51,151)
x3 = np.arange(151,256)

# defining the intensity transformations as piece wise functions
y1 = x1
y2 = 1.55 * x2 + 22.5
y3 = x3

transform = np.hstack((y1,y2,y3)).astype('uint8')

img = cv.imread('emma.jpg',0)
img_transformed = cv.LUT(img,transform)
```

The transform array is indexed from 0 to 255 (possible input intensity values) with each index carrying the respective output intensity value. The LUT function will look up the transform array for each pixel value of the input image, mapping it to the respective output intensity value.



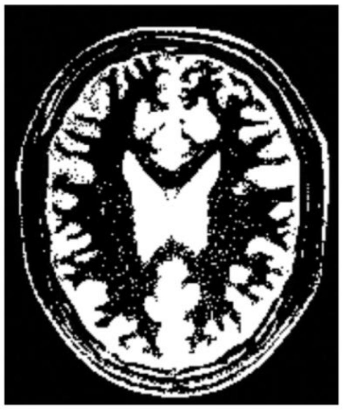


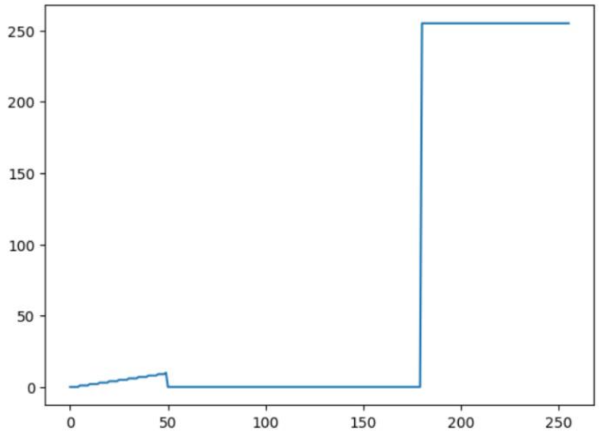
The pixels with intensity values between 50-150 have been transformed to higher intensity values. These pixels result in the whiter regions in the resulting image compared to the original image.

Question 2

To accentuate white matter, the pixels corresponding to brighter areas are transformed to an intensity value of 255 (white), i.e., made more brighter, and pixels corresponding to dark areas are mapped to low intensity values (mostly 0). By doing so brighter areas are accentuated.

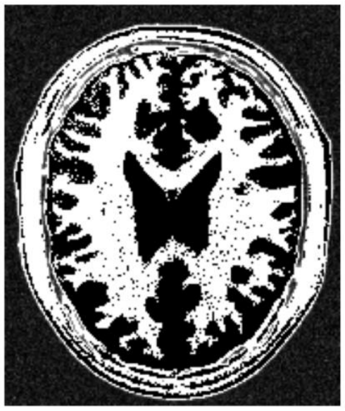
```
y1 = np.linspace(150,0,50).astype('uint8')
y2 = np.zeros(130).astype('uint8')
y3 = (np.ones(76)*255).astype('uint8')

transform = np.hstack([y1,y2,y3])
img_t2 = cv.LUT(img, transform)
```

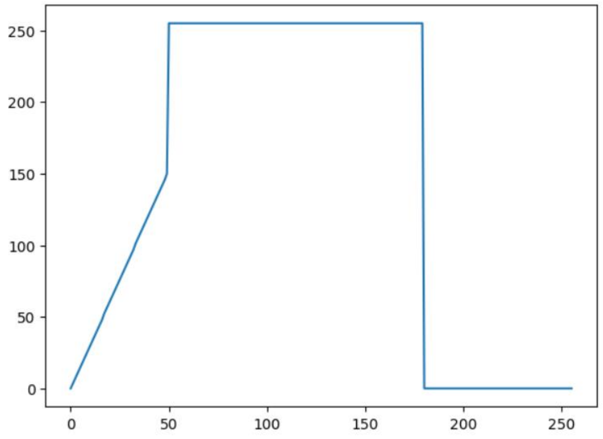




To accentuate gray matter, darker areas are made brighter (transformed mostly to 255) and the brighter areas are made darker (mapped to 0).

```
y1=np.linspace(0,150,50).astype(np.uint8)
y2=(np.ones(130)*255).astype('uint8')
y3=np.zeros(76).astype('uint8')
transform1 = np.hstack([y1,y2,y3])
img_t1=cv.LUT(img, transform1)
```

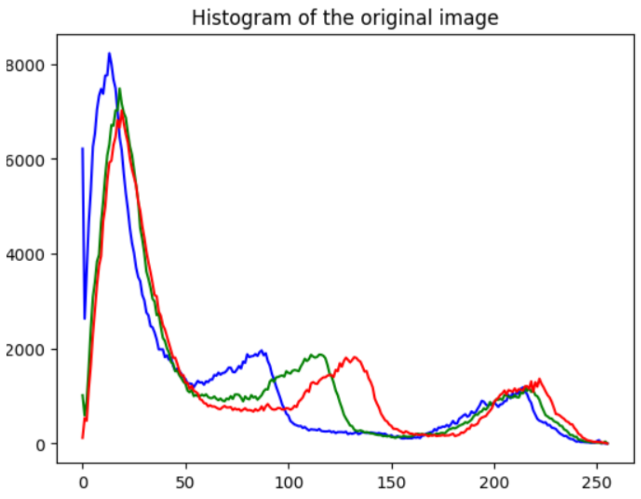


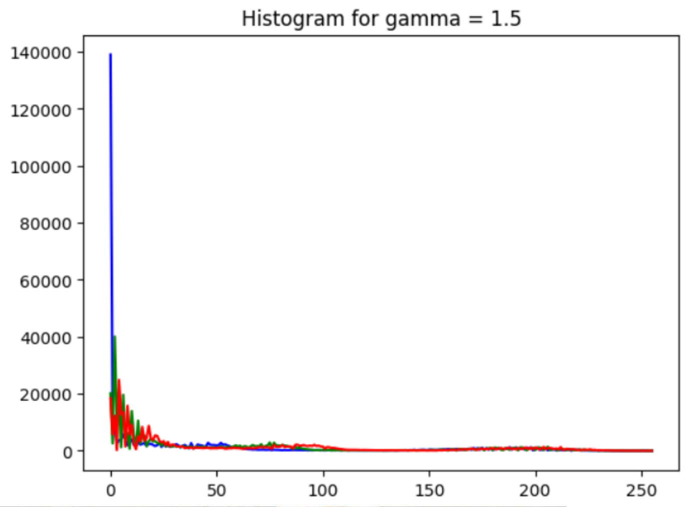


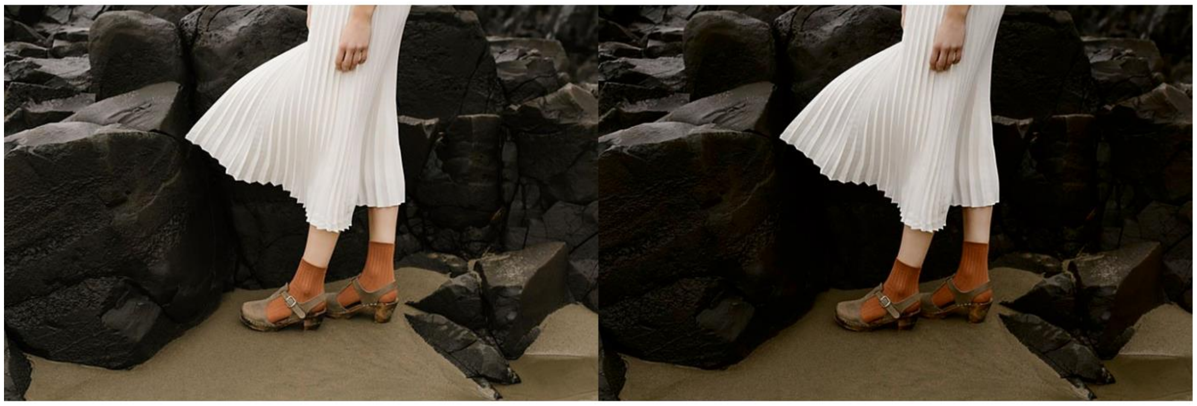
Question 3

```
#converting image to lab color space
lab_img = cv.cvtColor(img, cv.COLOR_BGR2Lab)
#cv2_imshow(lab_img)

#applying gamma correction to the L plane
for gamma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,1,2,5]:
  # for in_pixel in lab_img[:,:,0]:
  #   out_pixel = in_pixel ** gamma
  L, a, b = cv.split(lab_img)

  L_corrected = np.power(L / 255.0, gamma) * 255.0
  L_corrected = np.clip(L_corrected, 0, 255).astype(np.uint8)

  # Merge the corrected L channel with the original a and b channels
  lab_corrected = cv.merge([L_corrected, a, b])
  img_new = cv.cvtColor(lab_corrected,cv.COLOR_Lab2BGR)
```

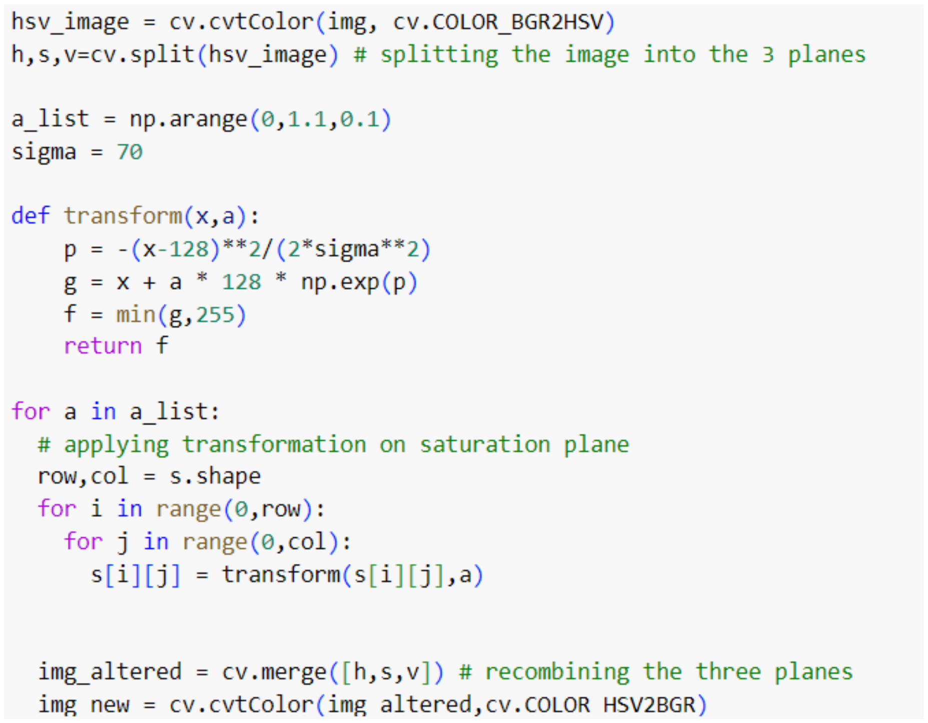
From the histogram of the original image, it can be seen that a majority of pixels have low intensity values. When gamma correction is applied, as gamma increases from 0.1, the brightness of the resulting image reduces and consequently the histogram starts shifting to the left. For gamma values greater than 1, the resulting image becomes more darker and the number of pixels with low intensity values increases as can be seen from the respective histograms. For the best visual appearance, a gamma value between 1 and 2 is optimal. Given below is the result for gamma =1.5.



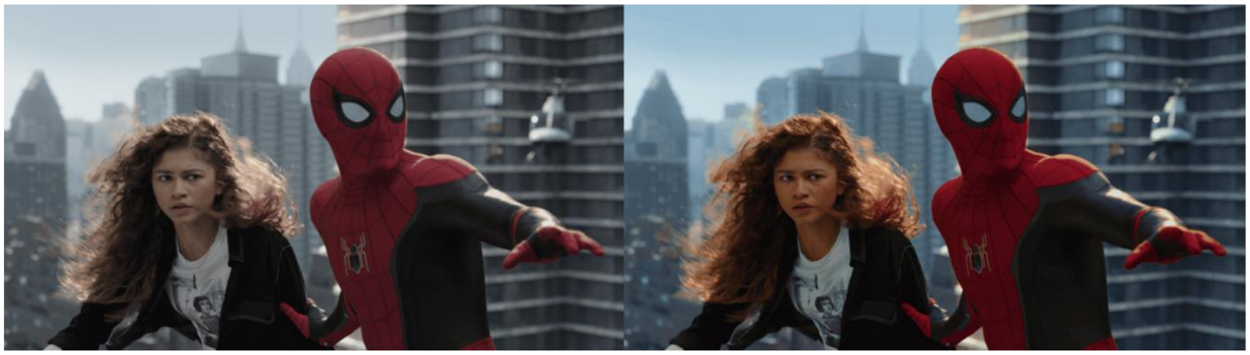


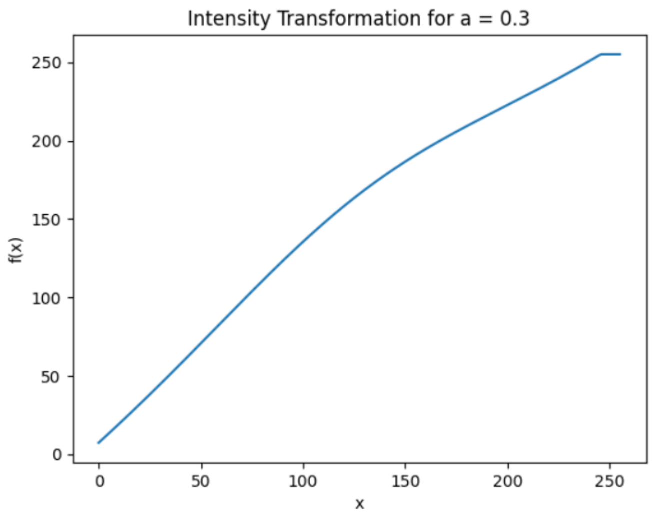


Question 4



```
hsv_image = cv.cvtColor(img, cv.COLOR_BGR2HSV)
h,s,v=cv.split(hsv_image) # splitting the image into the 3 planes

a_list = np.arange(0,1.1,0.1)
sigma = 70

def transform(x,a):
    p = -(x-128)**2/(2*sigma**2)
    g = x + a * 128 * np.exp(p)
    f = min(g,255)
    return f

for a in a_list:
  # applying transformation on saturation plane
  row,col = s.shape
  for i in range(0,row):
    for j in range(0,col):
      s[i][j] = transform(s[i][j],a)


  img_altered = cv.merge([h,s,v]) # recombining the three planes
  img new = cv.cvtColor(img altered,cv.COLOR HSV2BGR)
```



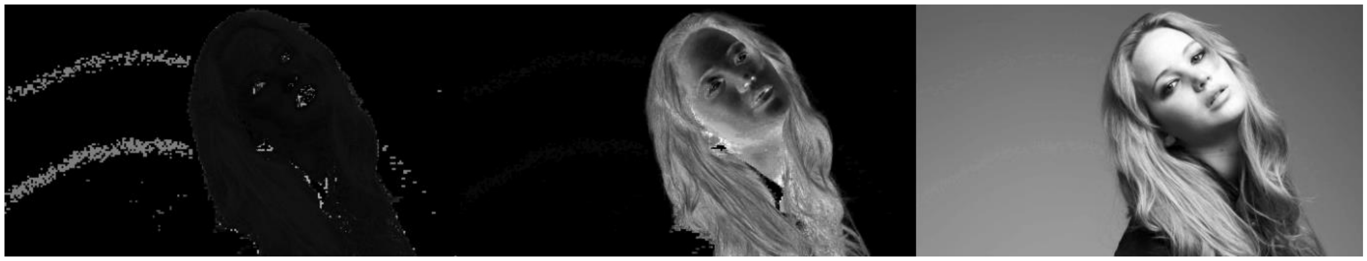


Here, we open the image using HSV format and we manipulate the saturation plane by applying an intensity trasnformation to it. This is applied for different values of "a" and the most pleasing visual output was obtained for a = 0.3. Here, the transform function will take an individual pixel as input and apply the transformation resulting in the desired output pixel.

Question 6

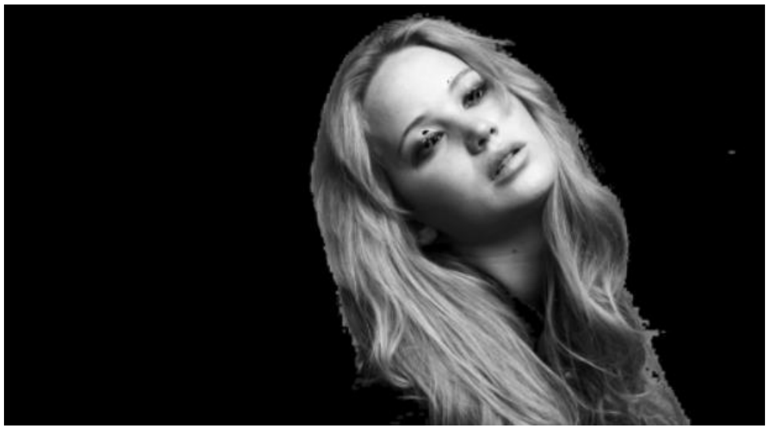
```
img = cv.imread('jeniffer.jpg')
img_hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
h,s,v = cv.split(img_hsv)
planes = np.hstack([h,s,v])
cv2_imshow(planes)
```

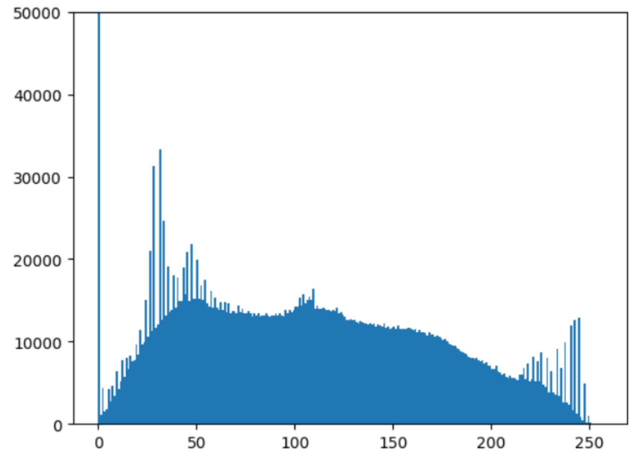
```
img=cv.imread("jeniffer.jpg",0)
t, mask = cv.threshold(s, 11, 1, cv.THRESH_BINARY)
img_fore=cv.bitwise_and(img,img,mask=mask)
```

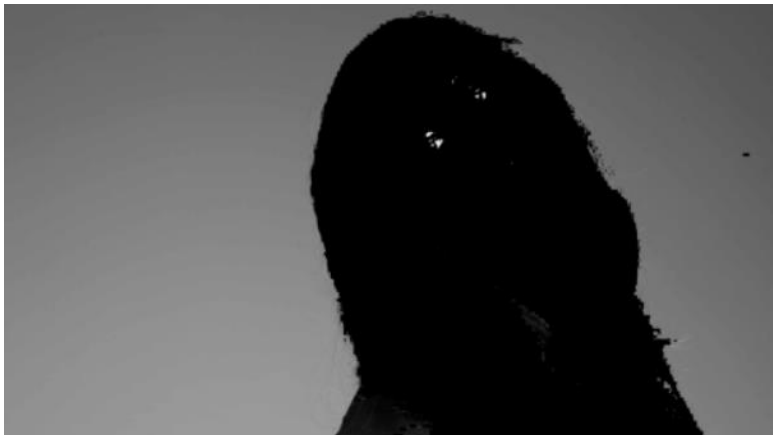


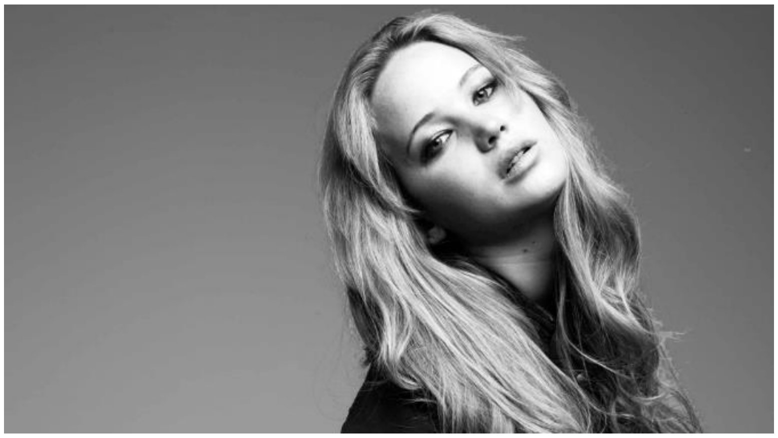
```
fore_equ = cv.equalizeHist(img_fore)
img_back=cv.bitwise_and(img,img,mask=1 - mask)
result = cv.add(img_back,fore_equ)
```

The threshold function is used to create a binary mask where, for intensity values of the input image greater than 11, output will be white(1) and for values lesser, pixels will be black(0). Using the bitwise and operation with this binary mask, the foreground's extracted. Similarly, by using the complement of this same mask, the background can be extracted.









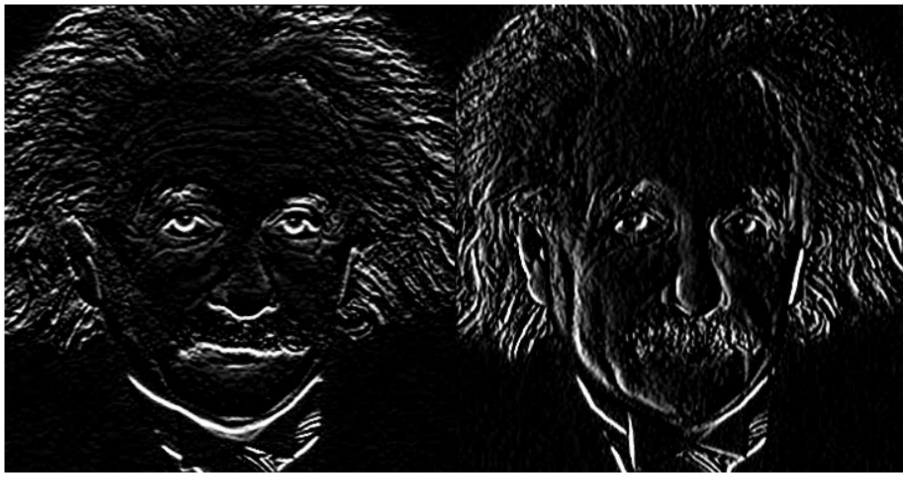
Question 7

a)

```
sobel_vert = np.array([[-1,-2,-1],[0,0,0],[1,2,1]], dtype = 'float')
sobel_hori = np.array([[-1,0,1],[-2,0,2],[-1,0,1]], dtype = 'float')
gradient_y = cv.filter2D(img,-1,sobel_vert)
gradient_x = cv.filter2D(img,-1,sobel_hori)
```

b)

```
def sobelfilter(img,kernel):
  result = np.zeros(img.shape)
  r_d , c_d = int((kernel.shape[0]-1)/2) , int((kernel.shape[1]-1)/2)
  #padded_img = cv.copyMakeBorder(img, r_d, r_d, c_d, c_d, cv.BORDER_CONSTANT, value=(0, 0, 0))
  rows,cols =img.shape
  for r in range(r_d,rows-r_d):
    for c in range(c_d,cols-c_d):
      result[r][c] = np.dot(img[r-r_d:r+r_d+1 , c-c_d:c+c_d+1].flatten(),kernel.flatten())
  return result
```



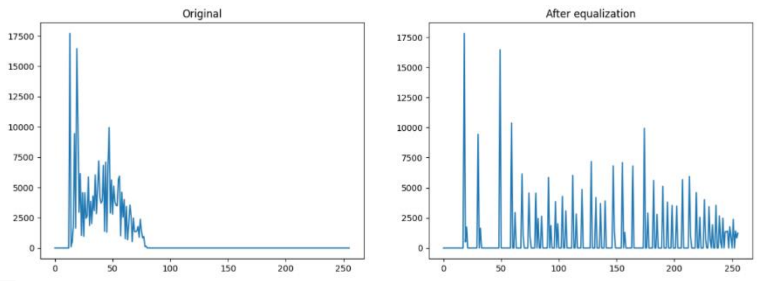
```
arr1 = np.array([[1],[2],[1]])
arr2=np.array([[1,0,-1]])

result = sobelfilter(image,arr1)
result = sobelfilter(result,arr2)
```

Since spatial filtering is the convolution of the image with the kernel, and convolution is an associative operation, we can get the same result through c).

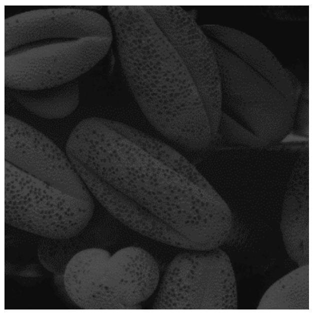
Question 5

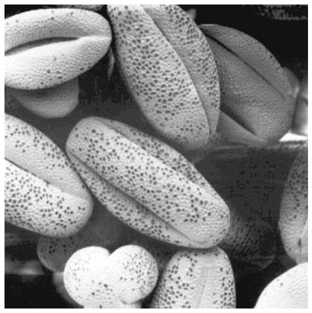
```
def histogram_equalization(image):
    image_t = image.copy()
    cols = image.shape[1]
    rows = image.shape[0]
    hist, bins = np.histogram(image.flatten(), 256, [0,256])
    transform = []
    cumsum = 0
    for i in range(len(hist)):
        cumsum+= hist[i]
        transform.append(np.round(cumsum/(rows*cols)*255))
    for i in range(rows):
      for j in range(cols):
        image_t[i][j] = transform[image[i][j]]
    transform = np.array(transform)
    return image_t
```

This spreads the intensity values in a more uniform manner compared to the original.

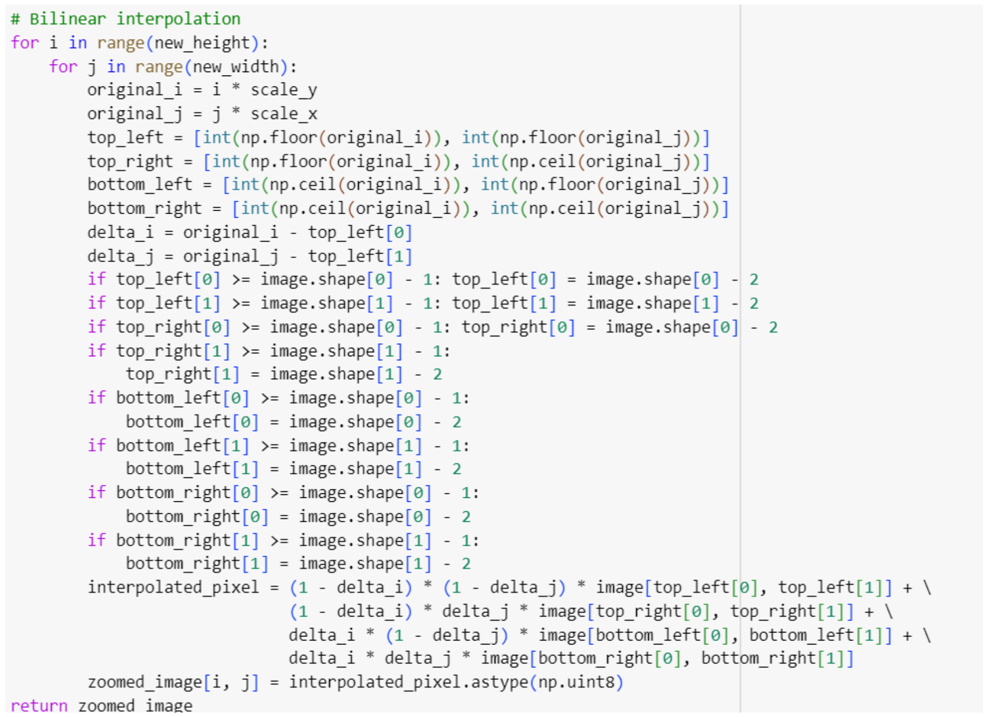


Here we calculate the cumulative distribution of the histogram, then normalize it and finally apply it as a transformation to the original image.



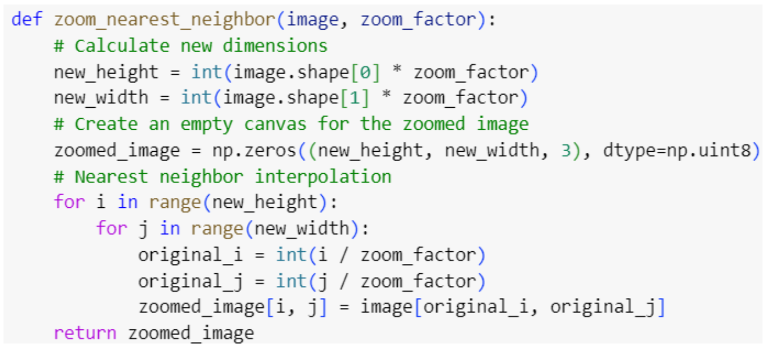


Question 8



```
# Bilinear interpolation
for i in range(new_height):
    for j in range(new_width):
        original_i = i * scale_y
        original_j = j * scale_x
        top_left = [int(np.floor(original_i)), int(np.floor(original_j))]
        top_right = [int(np.floor(original_i)), int(np.ceil(original_j))]
        bottom_left = [int(np.ceil(original_i)), int(np.floor(original_j))]
        bottom_right = [int(np.ceil(original_i)), int(np.ceil(original_j))]
        delta_i = original_i - top_left[0]
        delta_j = original_j - top_left[1]
        if top_left[0] >= image.shape[0] - 1: top_left[0] = image.shape[0] - 2
        if top_left[1] >= image.shape[1] - 1: top_left[1] = image.shape[1] - 2
        if top_right[0] >= image.shape[0] - 1: top_right[0] = image.shape[0] - 2
        if top_right[1] >= image.shape[1] - 1:
            top_right[1] = image.shape[1] - 2
        if bottom_left[0] >= image.shape[0] - 1:
            bottom_left[0] = image.shape[0] - 2
        if bottom_left[1] >= image.shape[1] - 1:
            bottom_left[1] = image.shape[1] - 2
        if bottom_right[0] >= image.shape[0] - 1:
            bottom_right[0] = image.shape[0] - 2
        if bottom_right[1] >= image.shape[1] - 1:
            bottom_right[1] = image.shape[1] - 2
        interpolated_pixel = (1 - delta_i) * (1 - delta_j) * image[top_left[0], top_left[1]] + \
                             (1 - delta_i) * delta_j * image[top_right[0], top_right[1]] + \
                             delta_i * (1 - delta_j) * image[bottom_left[0], bottom_left[1]] + \
                             delta_i * delta_j * image[bottom_right[0], bottom_right[1]]
        zoomed_image[i, j] = interpolated_pixel.astype(np.uint8)
return zoomed_image
```

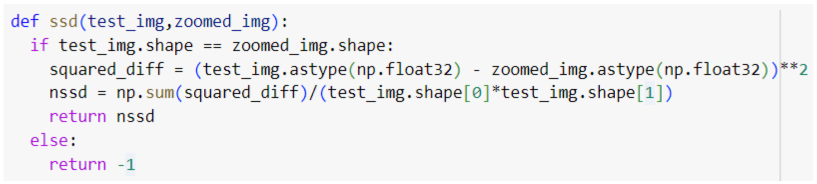
The pixel values of the resized image are calculated as the weighted average of the 4 nearest neighboring pixels of the original image.



```
def zoom_nearest_neighbor(image, zoom_factor):
    # Calculate new dimensions
    new_height = int(image.shape[0] * zoom_factor)
    new_width = int(image.shape[1] * zoom_factor)
    # Create an empty canvas for the zoomed image
    zoomed_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)
    # Nearest neighbor interpolation
    for i in range(new_height):
        for j in range(new_width):
            original_i = int(i / zoom_factor)
            original_j = int(j / zoom_factor)
            zoomed_image[i, j] = image[original_i, original_j]
    return zoomed_image
```

Nearest neighbor determines the pixel values of the resized image by the pixel values of the nearest neighbors of the original image.

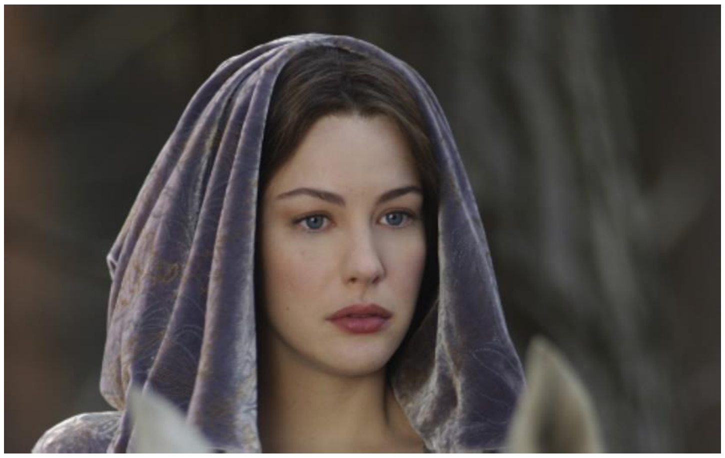
The images zoomed using nearest neighbor was more pixelated compared to the image zoomed using bilinear interpolation.



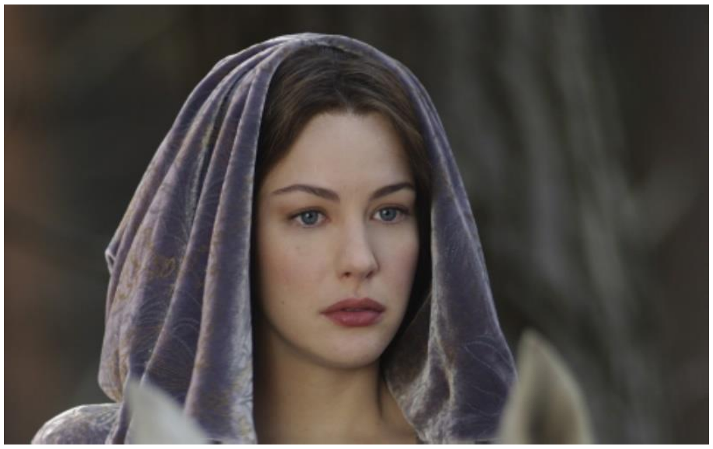
```
def ssd(test_img,zoomed_img):
  if test_img.shape == zoomed_img.shape:
    squared_diff = (test_img.astype(np.float32) - zoomed_img.astype(np.float32))**2
    nssd = np.sum(squared_diff)/(test_img.shape[0]*test_img.shape[1])
    return nssd
  else:
    return -1
```

Using the normalized sums of square differences, we can determine the quality of the zoomed image compared to that of the test image.

Nearest neighbor

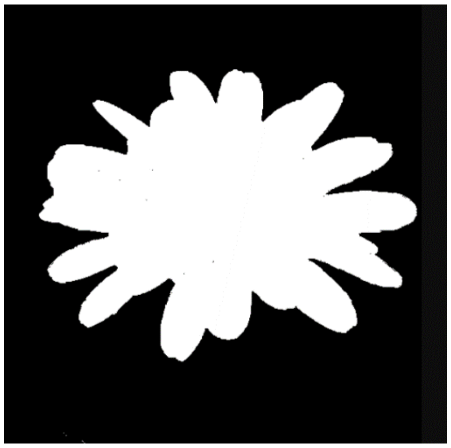
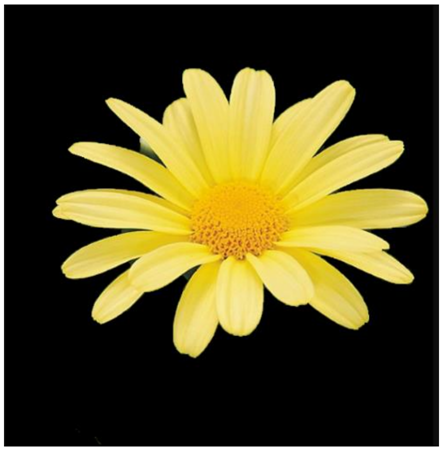
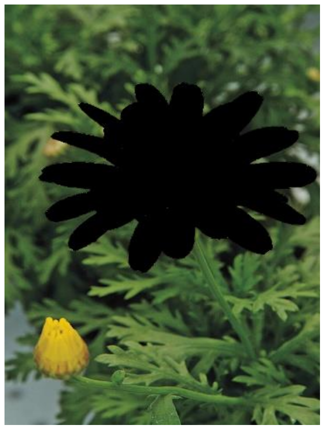


Bilinear interpolation



Question 9

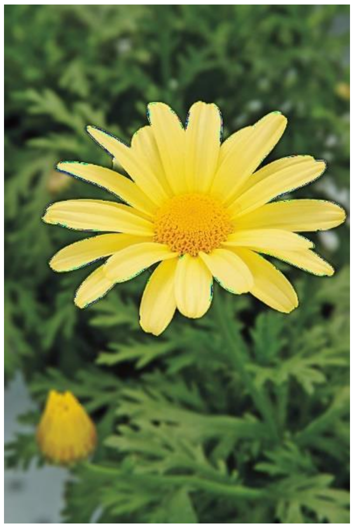
```
mask = np.zeros(img.shape[:2], dtype = 'uint8')
rect = (40,150,560,500)

fgModel = np.zeros((1, 65), dtype="float")
bgModel = np.zeros((1, 65), dtype="float")

(mask, bgModel, fgModel) = cv.grabCut(img, mask, rect, bgModel,fgModel, iterCount=2, mode=cv.GC_INIT_WITH_RECT)

fore_mask = (mask == cv.GC_PR_FGD).astype("uint8") * 255
outmask_fore = np.where((mask == cv.GC_BGD) | (mask == cv.GC_PR_BGD),0,1)
outmask_fore = (outmask_fore*255).astype(np.uint8)
foreground_img = cv.bitwise_and(img,img,mask=outmask_fore)

back_mask = (mask == cv.GC_PR_BGD).astype("uint8") * 255
outmask_back = np.where((mask == cv.GC_FGD) | (mask == cv.GC_PR_FGD),0,1)
outmask_back = (outmask_back*255).astype(np.uint8)
background_img = cv.bitwise_and(img,img,mask=outmask_back)
```

After defining the rectangle that contains the flower, the grabCut function is used to obtain the mask for the foreground and the background. Then they are extracted using *bitwise and* operation between the mask and the original image.

```
result = np.clip(np.add(foreground_img, cv.GaussianBlur(background_img, (9,9) , 4)) ,0,255)
```

Gaussian blur is added to the background and then merged with the foreground image to give an image with a blurred background.



When the background is blurred, the edges of the flower get blurred too. Due to the convolution, the value of these pixels in the edges can increase above 0. Hence, when the foreground and the background are added together, the edge pixels of the flower will be darker than the yellow color.

GitHub link : https://github.com/mnavindug/Machine-Vision-and-Image-Processing