DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN3160 - Image Processing and Machine Vision

Assignment 01

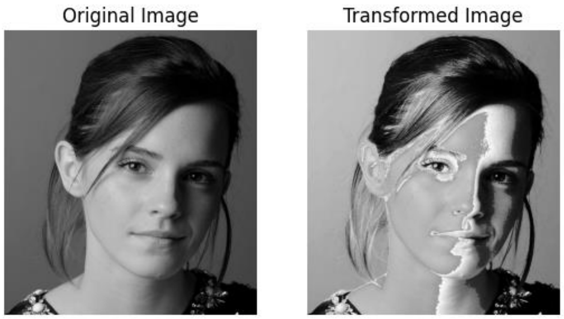
Intensity Transformations and Neighborhood Filtering

PUSHPAKUMARA H.M.R.M.

200488E

September 01, 2023

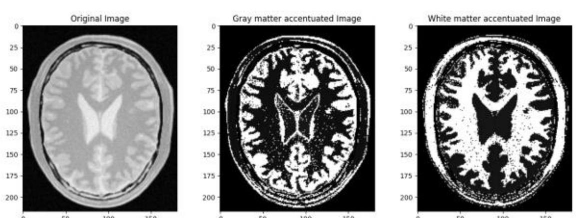
Question 1



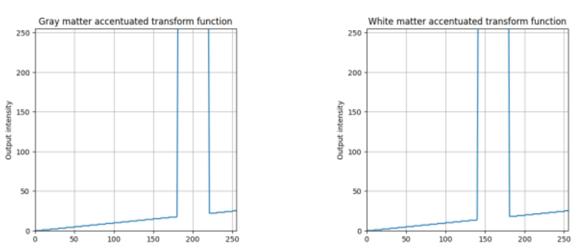
The transformation has enhanced the input intensities between 50 and 150, making input intensities near 150 appear white in the output image.

```python
c=np.array([(50, 100) , (150, 255)])
t1 = np.linspace(0, c[0,0], c[0,0]+1 - 0).astype('uint8')
print(len(t1))
t2 = np.linspace(c[0,1] , c[1,1], c[1,0] - c[0,0]).astype('uint8')
print(len(t2))
t3 = np.linspace(c[1,0], 255, 255 - c[1,0]).astype('uint8')
print(len(t3))

transform = np.concatenate((t1,t2), axis = 0).astype('uint8')
transform = np.concatenate((transform,t3), axis = 0).astype('uint8')
print(len(transform))
```

Question 2



The transformations I have used to enhance white matter and gray matter are as follows.



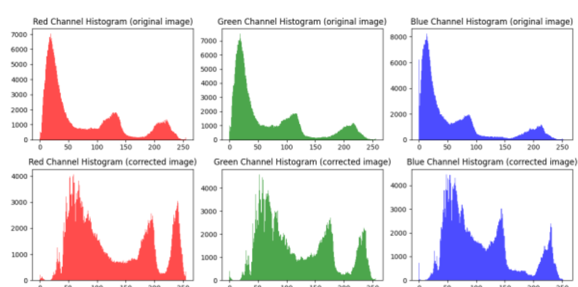
I found that the pixel intensities of the gray matter are between 180 and 220 in the input image and pixel intensities of the white matter in between 140 and 180 in the input image. Therefore, to enhance gray matter I increase pixel intensities between 180 – 220 to 255, and other intensities are lower by a factor of 10. To enhance white matter, I did the same for 140 – 180 input intensities.

```python
# Gray matter accentuated transform function
n = 10
enhance_down = 180
enhance_up = 220
t1 = np.linspace(0, enhance_down/n, enhance_down+1 - 0).astype('uint8')
print(len(t1))
t2 = np.linspace(255 , 255, enhance_up - enhance_down).astype('uint8')
print(len(t2))
t3 = np.linspace(enhance_up/n, 255/n, 255 - enhance_up).astype('uint8')
print(len(t3))

transform_g = np.concatenate((t1,t2), axis = 0).astype('uint8')
transform_g = np.concatenate((transform_g,t3), axis = 0).astype('uint8')
print(len(transform_g))


# White matter accentuated transform function
n = 10
enhance_down = 140
enhance_up = 180
t1_w = np.linspace(0, enhance_down/n, enhance_down+1 - 0).astype('uint8')
print(len(t1))
t2_w = np.linspace(255 , 255, enhance_up - enhance_down).astype('uint8')
print(len(t2))
t3_w = np.linspace(enhance_up/n, 255/n, 255 - enhance_up).astype('uint8')
print(len(t3))

transform_w = np.concatenate((t1_w,t2_w), axis = 0).astype('uint8')
transform_w = np.concatenate((transform_w,t3_w), axis = 0).astype('uint8')
print(len(transform_w))
```

Question 3

Here the Gamma value that I used is 2.2.

The histograms of the image before and after the gamma correction are,



Here we can see the brightness of the image has increased and dark areas have enhanced.

```
# Convert the image to the L*a*b* color space
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2Lab)

# Extract the L* component
L_channel = lab_image[:,:,0]

# Specify the gamma value 2.2
gamma = 2.2

# Apply gamma correction
L_corrected = np.power(L_channel / 255.0, 1.0 / gamma) * 255.0
L_corrected = np.clip(L_corrected, 0, 255).astype(np.uint8)

# Replace the original L* component with the corrected one
lab_image[:,:,0] = L_corrected

# Convert back to the BGR color space
output_image = cv2.cvtColor(lab_image, cv2.COLOR_Lab2BGR)
```

Question 4

(a) Original image with hue, saturation, and value planes.

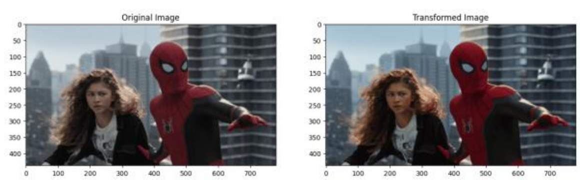
(b) After applying the given transformation to the saturation plane for the different values of **a** between 0 and 1.

(c) By the above observation, I choose **a = 0.4**. Here the original saturation channel and transformed saturation channel with a = 0.4.

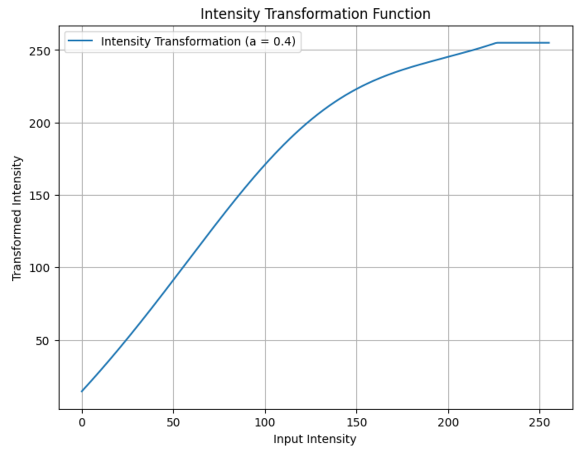
(d)

```
def intensity_transformation(x,a):
    return min((x+(a*128)*np.exp(-((x-128)**2)/(2*(70**2)))),255)

# Read the image
img = cv.imread("spider.png", cv.IMREAD_COLOR)

# Display the original image
plt.figure(figsize=(15,15))

plt.subplot(1,2,1)
plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
plt.title("Original Image")

# Convert the imagr to HSV color space
hsv_img = cv.cvtColor(img, cv.COLOR_BGR2HSV)

# Extract the saturation channel
saturation_channel = hsv_img[:, :, 1]

# Form above observation I select the parameter a = 0.4
a = 0.4

# Apply the intensity transformation function to the saturation channel
transformed_saturation = np.vectorize(intensity_transformation)(saturation_channel, a)

# Replace the saturation channel with the transformed saturation channel
hsv_img[:, :, 1] = transformed_saturation

# Convert the image back to BGR color space
transformed_img = cv.cvtColor(hsv_img, cv.COLOR_HSV2BGR)
```

(e) After all, the original image and the vibrance-enhanced image are as follows.

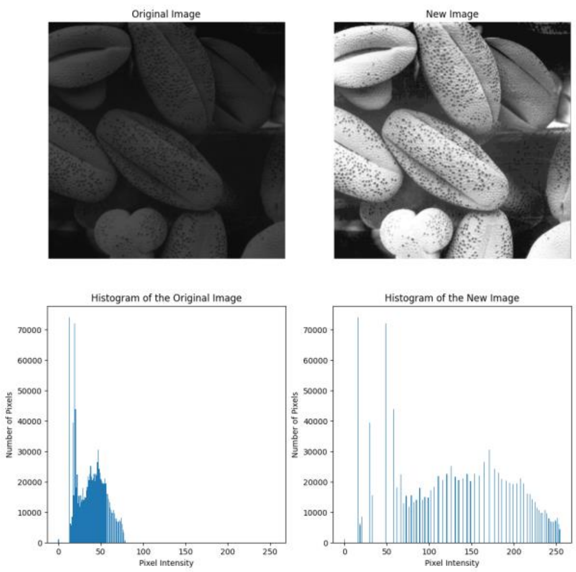


The intensity transformation is,



When we compare the input image and the output image, we can see the colors are more pleasantly visible in the output image than in the input image.

Question 5

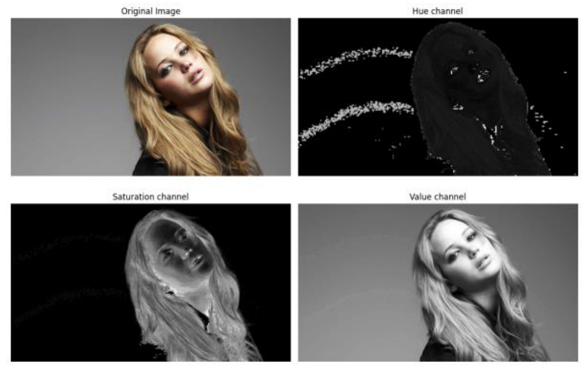


When compared with the original image histogram, the transformed image histogram has a more spread-out histogram (of course this is because of the histogram equalization) in high intensities. Therefore, the transformed image is lighter than the original image and we can see all features more clearly in the transformed image.

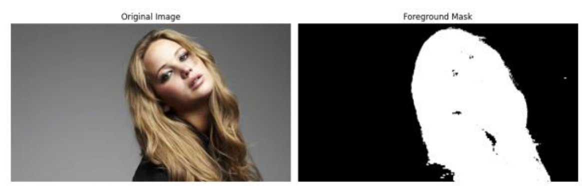
```python
# Pixel intensity array
pixel_intensity = np.zeros(256)

for pixel in range(0, 256):
    # Count the number of pixels with the same intensity
    count = np.count_nonzero(img == pixel)
    # Add the number of pixels to the pixel intensity array
    pixel_intensity[pixel] = count

# Calculate the probability of each pixel intensity
probability = pixel_intensity / image_size

# Calculate the cumulative probability
cumulative_probability = np.cumsum(probability)

# Calculate the new pixel intensity
new_pixel_intensity = np.round(cumulative_probability * 255).astype(np.uint8)

# Create a new image with the new pixel intensity
new_img = np.zeros(img.shape, dtype=np.uint8)
for i in range(256):
    new_img[img == i] = new_pixel_intensity[i]
```

Question 6

(a) Original image and hue, saturation, and value planes in grayscale.

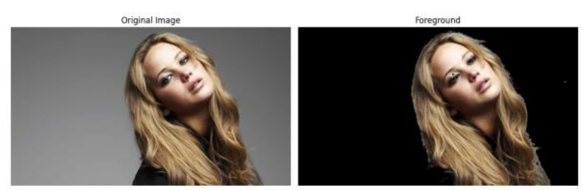


(b) When we observe the above planes, we can see that the saturation channel has a clear difference between background and foreground. Therefore, I selected the **saturation** channel to extract the foreground mask.

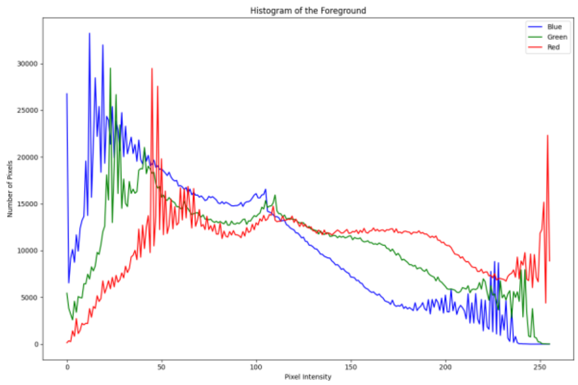


To obtain the foreground mask, I used 11 as the threshold value.

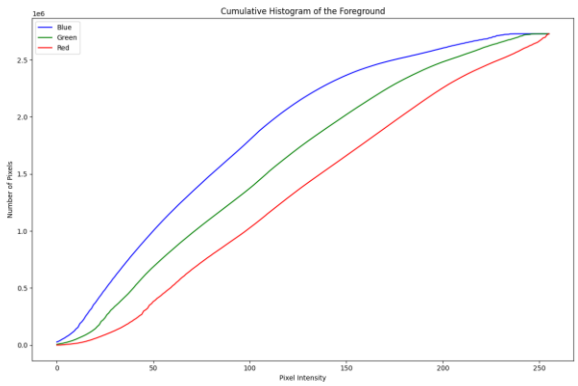
(c) Obtained foreground by using **cv.bitwise_and** is,



The histogram for the 3 channels of the foreground is,

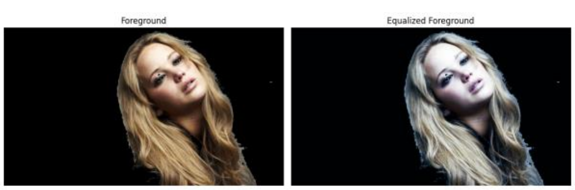


(d) Cumulative sum of the histogram

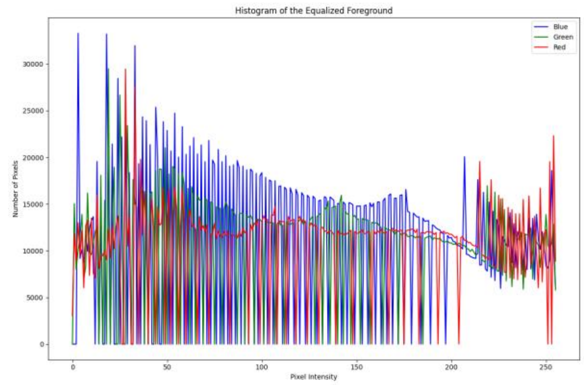


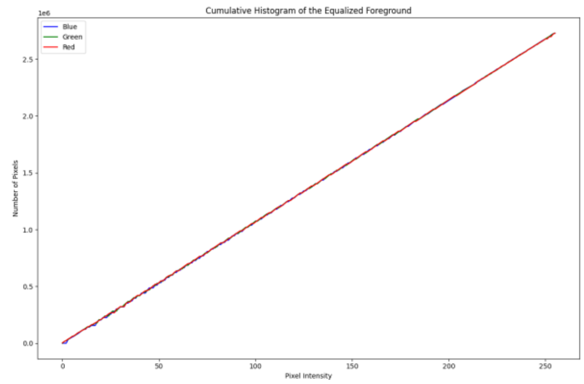

```
# Convert the image into 3 channels
hsv_img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2HSV)

# Split the image into 3 channels
hue_channel = hsv_img_orig[:, :, 0]
saturation_channel = hsv_img_orig[:, :, 1]
value_channel = hsv_img_orig[:, :, 2]

# Tresholf the saturation channel to extract the foreground mas
threshold = 11
_, foreground_mask = cv.threshold(saturation_channel, threshold, 255, cv.THRESH_BINARY)

# Obtain the foreground using cv.bitwise_and,compute the histogram and display
foreground = cv.bitwise_and(img_orig, img_orig, mask=foreground_mask)

# Convert foreground to grayscale for histogram computation
# foreground = cv.cvtColor(foreground, cv.COLOR_BGR2GRAY)

# Compute the histogram of the foreground
foreground_hist_blue = cv.calcHist([foreground], [0], foreground_mask, [256], [0, 256])
foreground_hist_green = cv.calcHist([foreground], [1], foreground_mask, [256], [0, 256])
foreground_hist_red = cv.calcHist([foreground], [2], foreground_mask, [256], [0, 256])

# Obtain the cumulative sum of the histogram
cumulative_hist_blue = np.cumsum(foreground_hist_blue)
cumulative_hist_green = np.cumsum(foreground_hist_green)
cumulative_hist_red = np.cumsum(foreground_hist_red)
```

(e) Histogram-equalized foreground

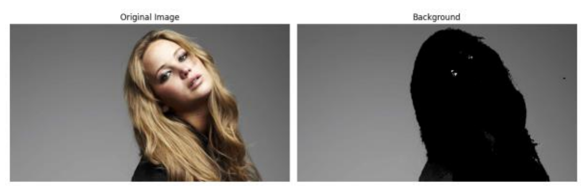


After the histogram equalization, the histogram and the cumulative sum of the histogram are as follows.

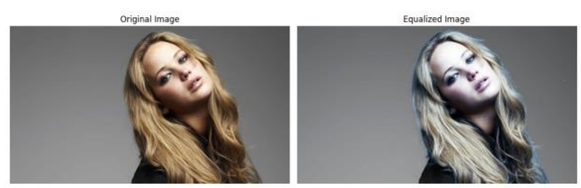




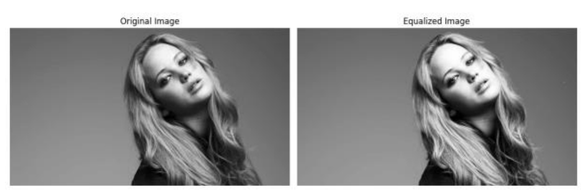
(f) Extracted background.



The result after adding the histogram equalized foreground with the extracted background is,



This is the equalized image and the original image in grayscale.



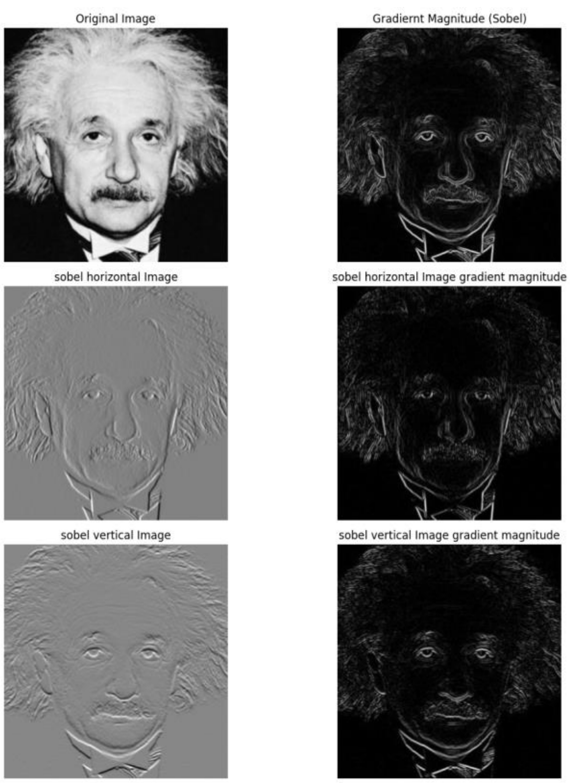
When we compare the original image and the foreground histogram equalized image, we can see the colors in the foreground in the equalized image have reduced. It means that the original image colors are more biased towards the light intensities.

But when we go to the grayscale, we can see that the image brightness seems to be increased and the image has pleasantly enhanced.

```
# Histogram equalization = Cumulative probability * 255
transformed_hist_blue = np.round(cumulative_hist_blue * 255 / cumulative_hist_blue[-1]).astype(np.uint8)
transformed_hist_green = np.round(cumulative_hist_green * 255 / cumulative_hist_green[-1]).astype(np.uint8)
transformed_hist_red = np.round(cumulative_hist_red * 255 / cumulative_hist_red[-1]).astype(np.uint8)

# Create a new image with the new pixel intensity
eqz_foreground = np.zeros(img_orig.shape, dtype=np.uint8)
for i in range(256):
    eqz_foreground[(foreground[:, :, 0] == i), 0] = transformed_hist_blue[i]
    eqz_foreground[(foreground[:, :, 1] == i), 1] = transformed_hist_green[i]
    eqz_foreground[(foreground[:, :, 2] == i), 2] = transformed_hist_red[i]

# Extract the background from the original image
background = cv.bitwise_and(img_orig, img_orig, mask=cv.bitwise_not(foreground_mask))

# Combine the equalized foreground and the background
eqz_img = cv.add(eqz_foreground, background)
```

Question 7

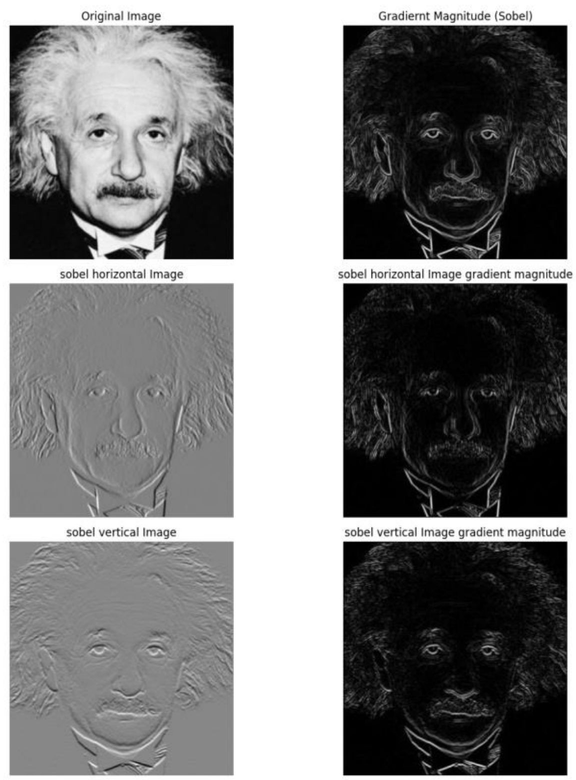
(a) Filter with Sobel operator by using the existing **filter2D** function.



Here I have filtered the image by using the Sobel horizontal and Sobel vertical operators and plotted the gradient magnitude for each Sobel horizontal and vertical filter output. Also, I have plotted the resultant gradient magnitude by using both Sobel vertical and horizontal outputs.

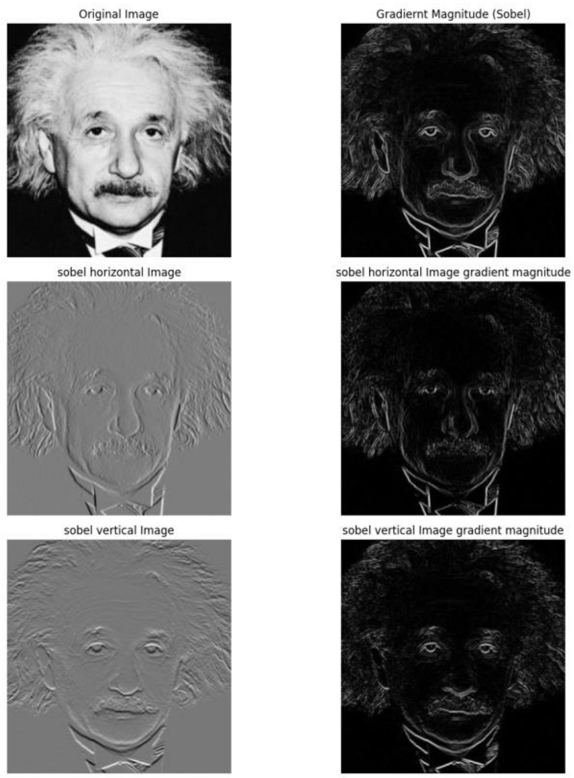
```
# Apply Sobel filter using filter2D
sobel_x = cv.filter2D(img, cv.CV_64F, np.array([[-1,0,1], [-2,0,2], [-1,0,1]]))
sobel_y = cv.filter2D(img, cv.CV_64F, np.array([[-1,-2,-1], [0,0,0], [1,2,1]]))

# Compute the gradient magnitude
gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)
gradient_magnitude_x = np.sqrt(sobel_x**2)
gradient_magnitude_y = np.sqrt(sobel_y**2)
```

(b) The code that I wrote to do Sobel filtering is an **iterative function** and the function outputs are as follows.




```
# Apply sobel filter to the image in x and y direction iteratively
sobel_x_img = np.zeros(img.shape, dtype='float')
sobel_y_img = np.zeros(img.shape, dtype='float')

for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        sobel_x_img[i,j] = np.sum(np.multiply(img[i-1:i+2, j-1:j+2], sobel_x))
        sobel_y_img[i,j] = np.sum(np.multiply(img[i-1:i+2, j-1:j+2], sobel_y))

# Compute the gradient magnitude
gradient_magnitude = np.sqrt(sobel_x_img**2 + sobel_y_img**2)
gradient_magnitude_x = np.sqrt(sobel_x_img**2)
gradient_magnitude_y = np.sqrt(sobel_y_img**2)
```

(c) Then the obtained results by using matrix multiplication property are as follows,




```
vertical_x_result = cv.filter2D(img, cv.CV_64F, sobel_x_vertical)
sobel_filtered_x = cv.filter2D(vertical_x_result, cv.CV_64F, sobel_x_horizontal)

# Apply Sobel filter for y direction using filter2D
sobel_y_vertical = np.array([[1], [0], [-1]], dtype='float')
sobel_y_horizontal = np.array([[1, 2, 1]], dtype='float')

horizontal_y_result = cv.filter2D(img, cv.CV_64F, sobel_y_horizontal)
sobel_filtered_y = cv.filter2D(horizontal_y_result, cv.CV_64F, sobel_y_vertical)

# Compute the gradient magnitude
gradient_magnitude = np.sqrt(sobel_filtered_x**2 + sobel_filtered_y**2)
gradient_magnitude_x = np.sqrt(sobel_filtered_x**2)
gradient_magnitude_y = np.sqrt(sobel_filtered_y**2)
```

Question 8

```
# Define the zoom factor
zoom_factor = 4

# Define the new image size
new_img_size = (img.shape[0] * zoom_factor, img.shape[1] * zoom_factor, img.shape[2])
new_img_height = new_img_size[0]
new_img_width = new_img_size[1]

# Create a new image with the new size
new_img = np.zeros(new_img_size, dtype=np.uint8)

# Zoom the image using nearest neighbour interpolation
new_image_nni = zoom_nearest_neighbour(img,new_img,new_img_height,new_img_width,zoom_factor)

# Zoom the image using bilinear interpolation
new_image_bli = zoom_bilinear_interpolation(img,new_img,new_img_height,new_img_width,zoom_factor)
```

Nearest-neighbor interpolation

```
# Function to zoom image by using nearest neighbour interpolation
def zoom_nearest_neighbour(img,new_img,new_img_height,new_img_width,zoom_factor):
    for i in range(new_img_height):
        for j in range(new_img_width):
            new_img[i,j] = img[int(i/zoom_factor),int(j/zoom_factor)]
    return new_img
```

SSD for im01 $=$ 39.240651041666666

SSD for im02 $=$ 16.20416898148148

SSD for im04 $=$ 81.65426066583076

Bilinear interpolation

```
def zoom_bilinear_interpolation(img,new_img,new_img_height,new_img_width,zoom_factor):
    for i in range(new_img_height):
        for j in range(new_img_width):
            x = int(i/zoom_factor)
            y = int(j/zoom_factor)
            a = i/zoom_factor - x
            b = j/zoom_factor - y
            if x == img.shape[0]-1 or y == img.shape[1]-1:
                new_img[i,j] = img[x,y]
            else:
                new_img[i,j] = (1-a)*(1-b)*img[x,y] + (1-a)*b*img[x,y+1] + a*(1-b)*img[x+1,y] + a*b*img[x+1,y+1]
    return new_img
```

SSD for im01 $=$ 39.240651041666666

SSD for im02 $=$ 16.20416898148148

SSD for im04 $=$ 81.65426066583076

```
# compute the sum of squared difference (SSD) between the original and zoomed images
ssd_nni = np.sum((original_image - new_image_nni)**2)
ssd_bli = np.sum((original_image - new_image_bli)**2)

# Normalize the SSD by the total number of pixels
num_pixels = original_image.shape[0] * original_image.shape[1] * original_image.shape[2]  # Total number of pixels
ssd_nni_normalized = ssd_nni / num_pixels
ssd_bli_normalized = ssd_bli / num_pixels
```

Question 9

(a) Original image, final segmentation mask, foreground image, background image.

```
# Use grabCut to segment the image and Show the final segmentation mask, foreground image, and background image
mask = np.zeros(img.shape[:2], np.uint8)
bgdModel = np.zeros((1,65), np.float64)
fgdModel = np.zeros((1,65), np.float64)
rect = (50, 100, 500, 450)

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

mask2 = np.where((mask==2)|(mask==0), 0, 1).astype('uint8')

# Segment the foreground and background
img_fg = img * mask2[:,:,np.newaxis]
img_bg = img * (1 - mask2[:,:,np.newaxis])
# img_bg = img - img_fg
```

(b) To blur the background, I added a Gaussian noise. The original background and blurred background are,

After that, I added the blurred background to the foreground. The original image and the enhanced image are as follows,


```
# Apply gaussian blur to the background image
img_bg_blur = cv.GaussianBlur(img_bg, (21, 21), 0)

# Combine the foreground and blurred background
img_final = img_fg + img_bg_blur
```

(c) To Segment the image to the foreground and background I used the GrabCut algorithm. It is a popular algorithm for image segmentation. It works by estimating the pixels that belong to the foreground and background based on an initial rectangle. However, the algorithm may not be perfect, especially around the edges of the object.

And I use the Gaussian filter to blur the background. Gaussian blur is a smoothing filter that averages the values of neighboring pixels. When applied to the background of an image, it will blur both the true background and the pixels that were incorrectly classified as background. This is because the blur does not distinguish between the two types of pixels.

Because of these reasons, there is quite dark in the enhanced image background just beyond the edge of the flower.