EN3160 – Image Processing and Machine Vision

Assignment 01 - Intensity Transformations and Neighborhood Filtering

Name: Alahakoon U M Y B

Index No: 210027C

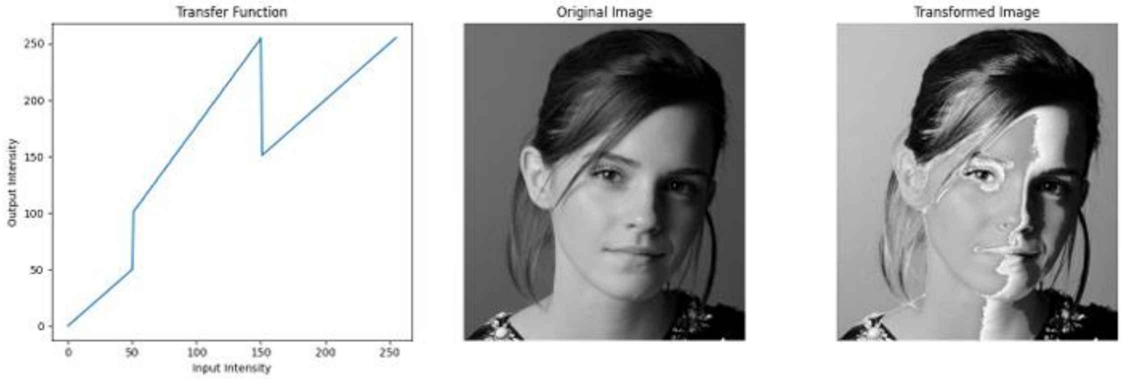
Date: 30/09/2024

GitHub Link: YasiruAlahakoon/-Intensity-Transformations-and-Neighborhood-Filtering (github.com)

Question 01

```
discontinuities = np.array([(50, 50), (50, 100), (150, 255), (150,
150), (255, 255)])

transform = np.concatenate([
    np.linspace(0, discontinuities[0, 1], discontinuities[0, 0] +
1).astype('uint8'),
    np.linspace(discontinuities[0, 1], discontinuities[1, 1],
discontinuities[1, 0] - discontinuities[0, 0] + 1)
[1:].astype('uint8'),
    np.linspace(discontinuities[1, 1], discontinuities[2, 1],
discontinuities[2, 0] - discontinuities[1, 0] + 1)
[1:].astype('uint8'),
    np.linspace(discontinuities[2, 1], discontinuities[3, 1],
discontinuities[3, 0] - discontinuities[2, 0] + 1)
[1:].astype('uint8'),
    np.linspace(discontinuities[3, 1], discontinuities[4, 1],
discontinuities[4, 0] - discontinuities[3, 0] + 1)[1:].astype('uint8')
])

Emma_transformed = cv.LUT(Emma, transform)
```

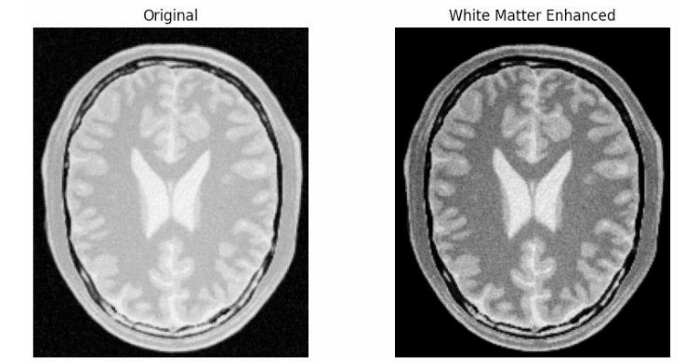
- The intensity transformation graph shows that input intensity values between *50 and 150* have been increased, resulting in a brighter appearance of the left side of the face in the transformed image.



Question 02

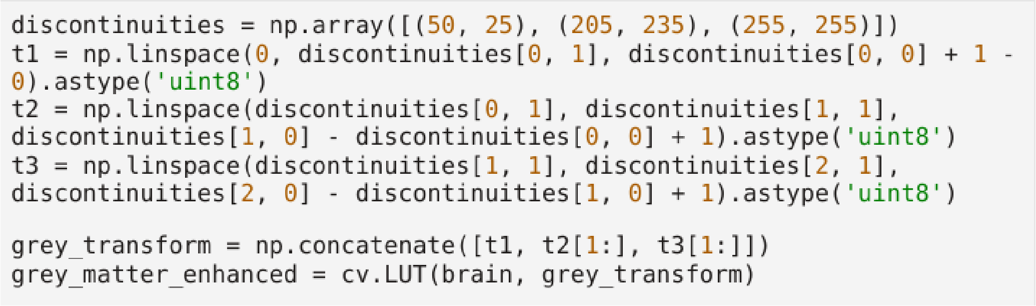
A) White matter Transformation

```
discontinuities = np.array([(125, 25), (255, 255)])
t1 = np.linspace(0, discontinuities[0, 1], discontinuities[0, 0] + 1 -
0).astype('uint8')
t2 = np.linspace(discontinuities[0, 1], discontinuities[1, 1],
discontinuities[1, 0] - discontinuities[0, 0] + 1).astype('uint8')
white_transform = np.concatenate([t1, t2[1:]])

whitematter_enhanced = cv.LUT(brain, white_transform)
```



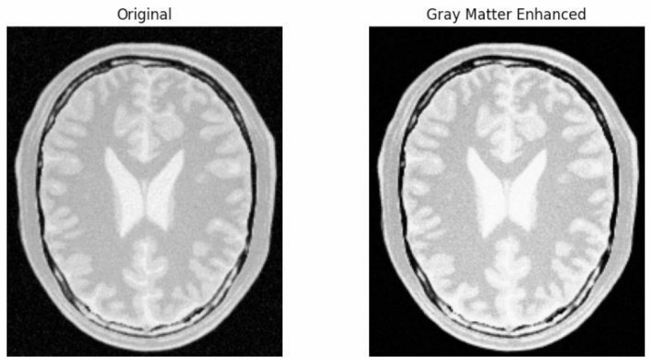
Explanation: The *white matter* is brighter than the surrounding areas, so increasing contrast in these bright regions can bring out more details.

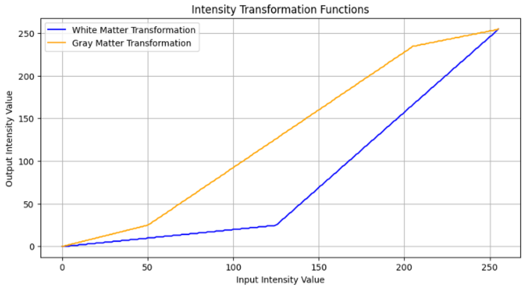
B) Grey matter Transformation



```
discontinuities = np.array([(50, 25), (205, 235), (255, 255)])
t1 = np.linspace(0, discontinuities[0, 1], discontinuities[0, 0] + 1 -
0).astype('uint8')
t2 = np.linspace(discontinuities[0, 1], discontinuities[1, 1],
discontinuities[1, 0] - discontinuities[0, 0] + 1).astype('uint8')
t3 = np.linspace(discontinuities[1, 1], discontinuities[2, 1],
discontinuities[2, 0] - discontinuities[1, 0] + 1).astype('uint8')

grey_transform = np.concatenate([t1, t2[1:], t3[1:]])
grey_matter_enhanced = cv.LUT(brain, grey_transform)
```

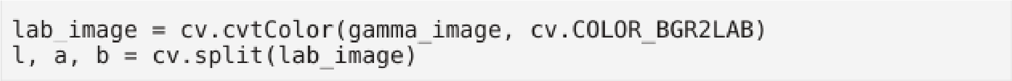
Explanation: *The grey matter* is darker than the white matter, so enhancing contrast in the medium intensity areas will reveal more details.



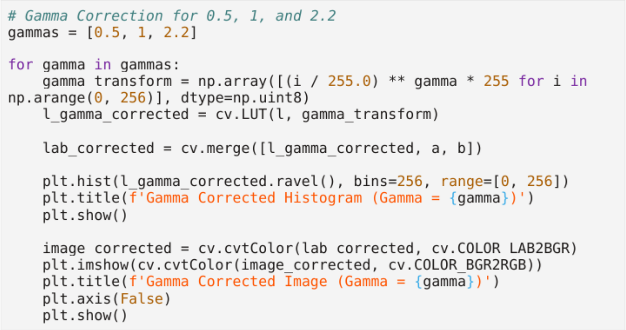


Question 03

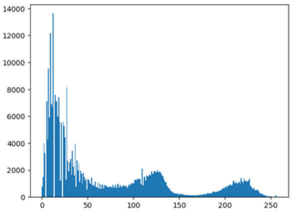
- **For a Gamma Value equal to 1**: gamma correction curve is linear, the image stays the same, and its histogram keeps its original shape.
- **For a Gamma Value less than 1(0.5)**: This results in gamma compression, which darkens the image and lowers the contrast. The histogram shifts to the left, with more values clustering toward the darker end.
- **For a Gamma Value greater than 1(2.2)**: This leads to gamma expansion, which brightens the image and enhances the contrast. The histogram shifts to the right, with more values concentrating toward the lighter end

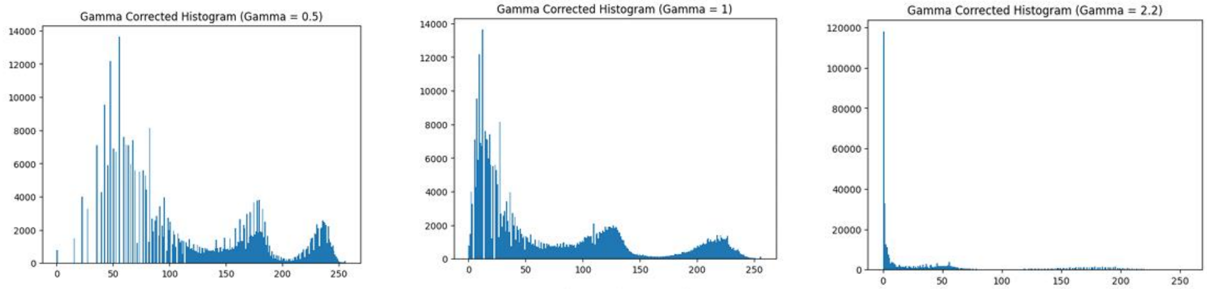


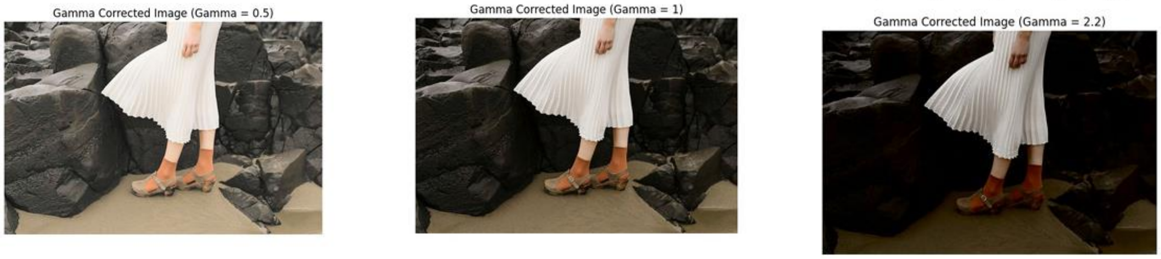
```
lab_image = cv.cvtColor(gamma_image, cv.COLOR_BGR2LAB)
l, a, b = cv.split(lab_image)
```



```
# Gamma Correction for 0.5, 1, and 2.2
gammas = [0.5, 1, 2.2]

for gamma in gammas:
    gamma_transform = np.array([(i / 255.0) ** gamma * 255 for i in
np.arange(0, 256)], dtype=np.uint8)
    l_gamma_corrected = cv.LUT(l, gamma_transform)

    lab_corrected = cv.merge([l_gamma_corrected, a, b])

    plt.hist(l_gamma_corrected.ravel(), bins=256, range=[0, 256])
    plt.title(f'Gamma Corrected Histogram (Gamma = {gamma})')
    plt.show()

    image_corrected = cv.cvtColor(lab_corrected, cv.COLOR_LAB2BGR)
    plt.imshow(cv.cvtColor(image_corrected, cv.COLOR_BGR2RGB))
    plt.title(f'Gamma Corrected Image (Gamma = {gamma})')
    plt.axis(False)
    plt.show()
```

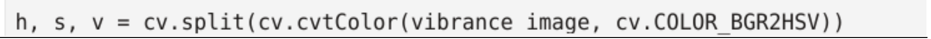




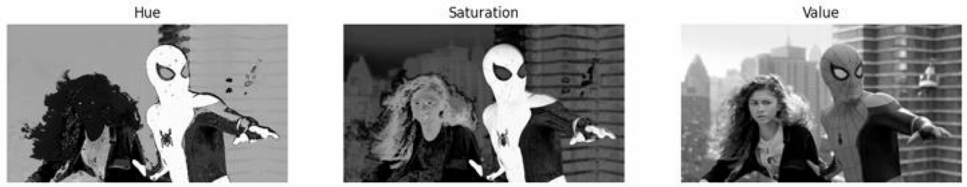

Gamma Corrected Image (Gamma = 0.5)
Gamma Corrected Image (Gamma = 1)
Gamma Corrected Image (Gamma = 2.2)

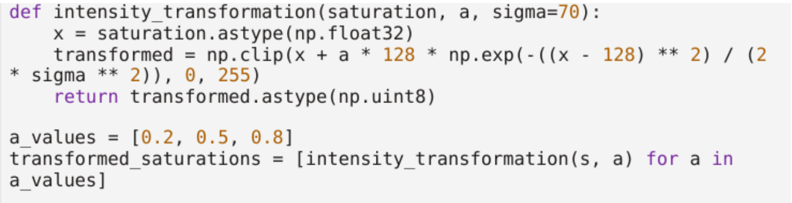
Question 04

A)



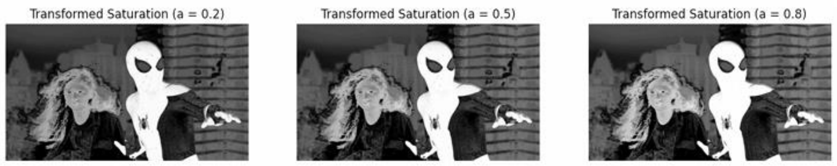
```
h, s, v = cv.split(cv.cvtColor(vibrance_image, cv.COLOR_BGR2HSV))
```

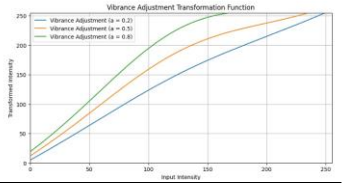




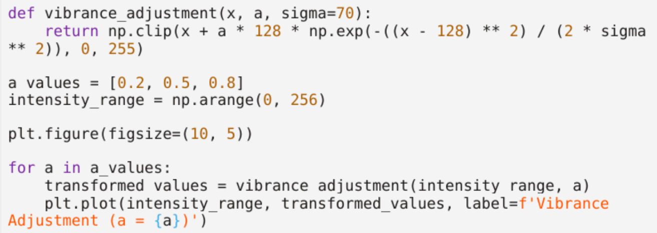
```
def intensity_transformation(saturation, a, sigma=70):
    x = saturation.astype(np.float32)
    transformed = np.clip(x + a * 128 * np.exp(-((x - 128) ** 2) / (2
* sigma ** 2)), 0, 255)
    return transformed.astype(np.uint8)

a_values = [0.2, 0.5, 0.8]
transformed_saturations = [intensity_transformation(s, a) for a in
a_values]
```

B)


Transformed Saturation (a = 0.2)
Transformed Saturation (a = 0.5)
Transformed Saturation (a = 0.8)



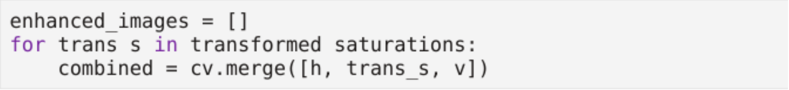
C)



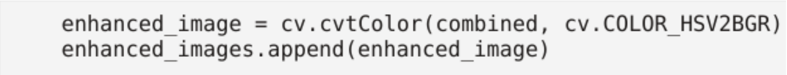
```python
def vibrance_adjustment(x, a, sigma=70):
    return np.clip(x + a * 128 * np.exp(-((x - 128) ** 2) / (2 * sigma
** 2)), 0, 255)

a values = [0.2, 0.5, 0.8]
intensity_range = np.arange(0, 256)

plt.figure(figsize=(10, 5))

for a in a_values:
    transformed values = vibrance adjustment(intensity range, a)
    plt.plot(intensity_range, transformed_values, label=f'Vibrance
Adjustment (a = {a})')
```

D)

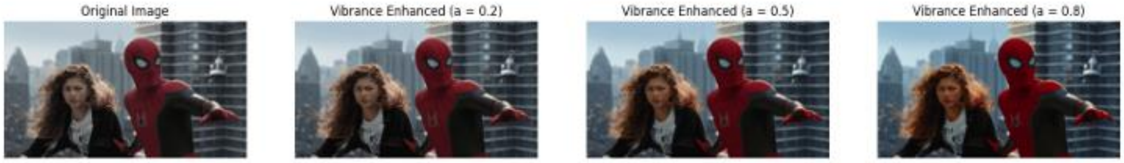
Explanation: Different values of a were tested in a loop. For small **a** value, the vibrance increased only slightly, while a value of ***a = 0.8*** led to a much higher vibrance. Based on the observed images, ***a = 0.5*** seemed like a good balance, as it boosted vibrance without making the image look overly saturated, which happens with larger **a** value.



```python
enhanced_images = []
for trans s in transformed saturations:
    combined = cv.merge([h, trans_s, v])
```

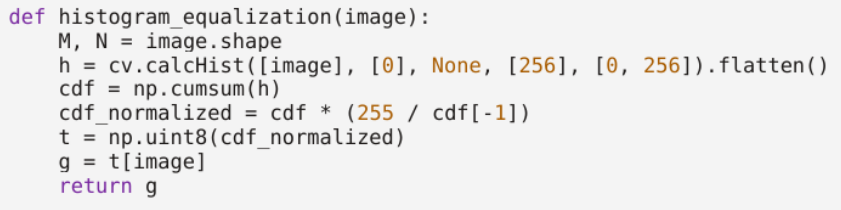


```python
    enhanced_image = cv.cvtColor(combined, cv.COLOR_HSV2BGR)
    enhanced_images.append(enhanced_image)
```


Original Image
Vibrance Enhanced (a = 0.2)
Vibrance Enhanced (a = 0.5)
Vibrance Enhanced (a = 0.8)

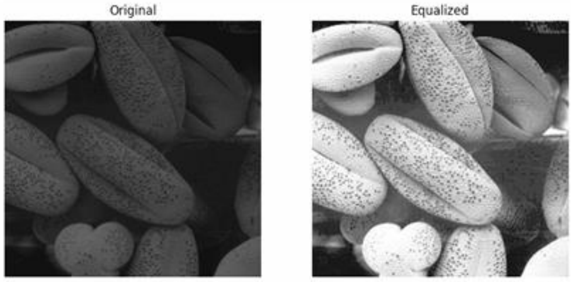
E)

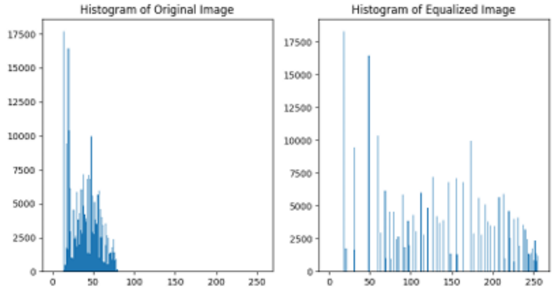
Question 05



```python
def histogram_equalization(image):
    M, N = image.shape
    h = cv.calcHist([image], [0], None, [256], [0, 256]).flatten()
    cdf = np.cumsum(h)
    cdf_normalized = cdf * (255 / cdf[-1])
    t = np.uint8(cdf_normalized)
    g = t[image]
    return g
```

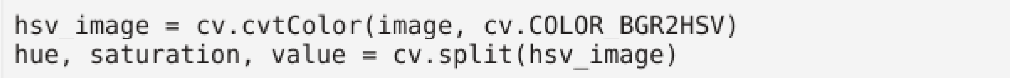
Explanation: The original histogram values are spread across the full intensity range, and these adjusted intensities are used to create a more balanced or equalized image.



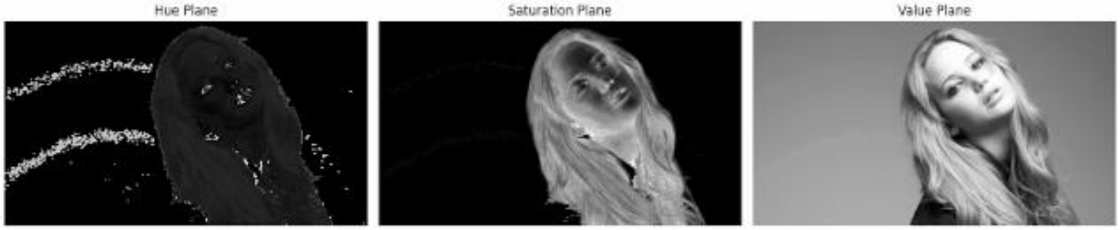


Question 06

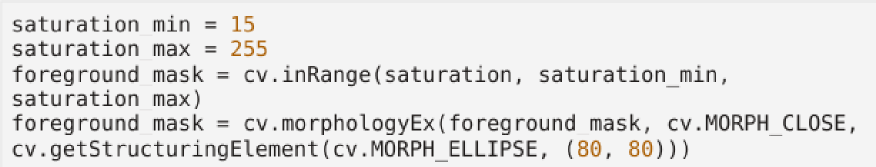
A)



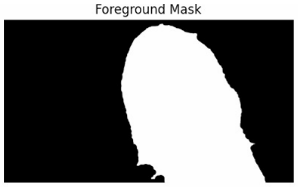
```
hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)
hue, saturation, value = cv.split(hsv_image)
```



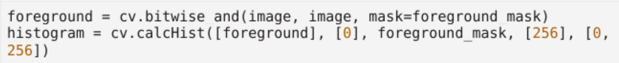
B)



```
saturation_min = 15
saturation_max = 255
foreground_mask = cv.inRange(saturation, saturation_min,
saturation_max)
foreground_mask = cv.morphologyEx(foreground_mask, cv.MORPH_CLOSE,
cv.getStructuringElement(cv.MORPH_ELLIPSE, (80, 80)))
```

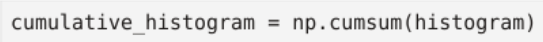


C)

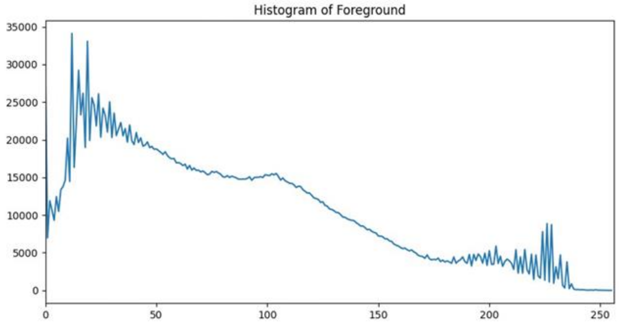


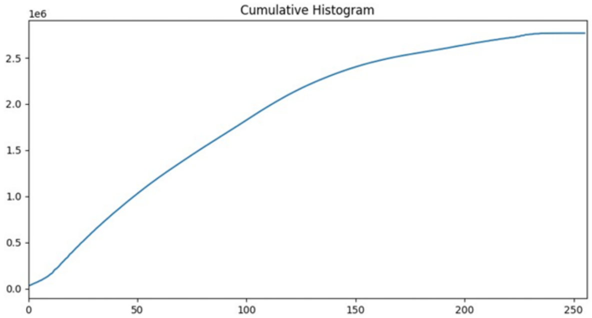
```
foreground = cv.bitwise_and(image, image, mask=foreground_mask)
histogram = cv.calcHist([foreground], [0], foreground_mask, [256], [0,
256])
```

D)

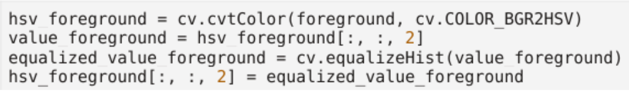


```
cumulative_histogram = np.cumsum(histogram)
```



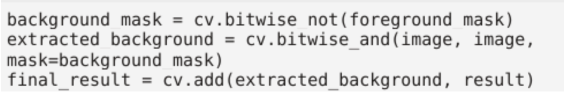


E)

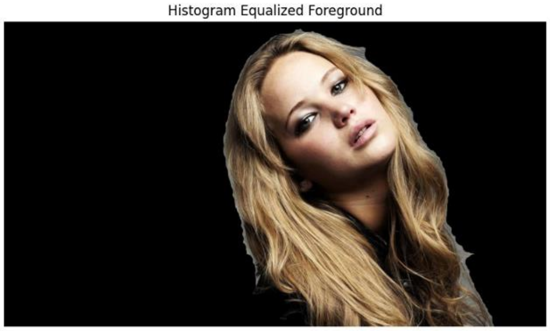


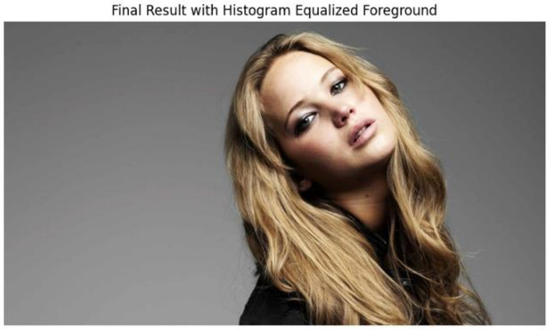
```
hsv_foreground = cv.cvtColor(foreground, cv.COLOR_BGR2HSV)
value_foreground = hsv_foreground[:, :, 2]
equalized_value_foreground = cv.equalizeHist(value_foreground)
hsv_foreground[:, :, 2] = equalized_value_foreground
```

F)



```
background_mask = cv.bitwise_not(foreground_mask)
extracted_background = cv.bitwise_and(image, image,
mask=background_mask)
final_result = cv.add(extracted_background, result)
```

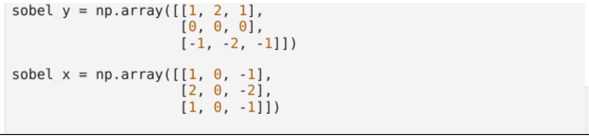




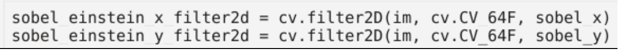
- The foreground object is much more saturated than the background, so the saturation plane is used for thresholding, with fine-tuning needed for exact values.

Question 07

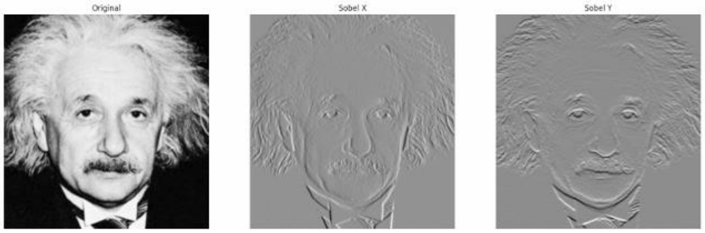
A)



```python
sobel_y = np.array([[1, 2, 1],
                    [0, 0, 0],
                    [-1, -2, -1]])

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

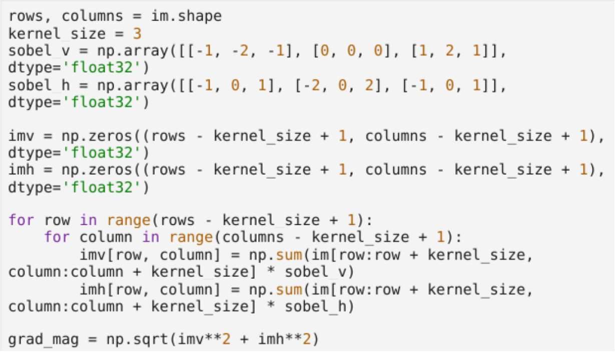


```python
sobel_einstein_x_filter2d = cv.filter2D(im, cv.CV_64F, sobel_x)
sobel_einstein_y_filter2d = cv.filter2D(im, cv.CV_64F, sobel_y)
```

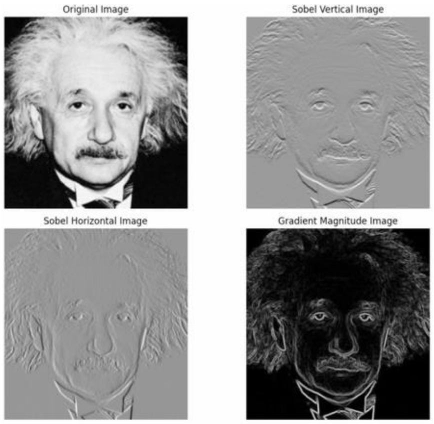


Explanation: The Sobel X filter highlights vertical edges by detecting horizontal intensity changes, while the Sobel Y filter uses horizontal edges by capturing vertical intensity changes. Similar results were obtained, Using custom created filter.

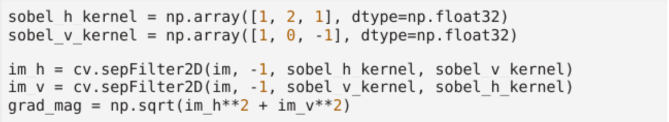
B)



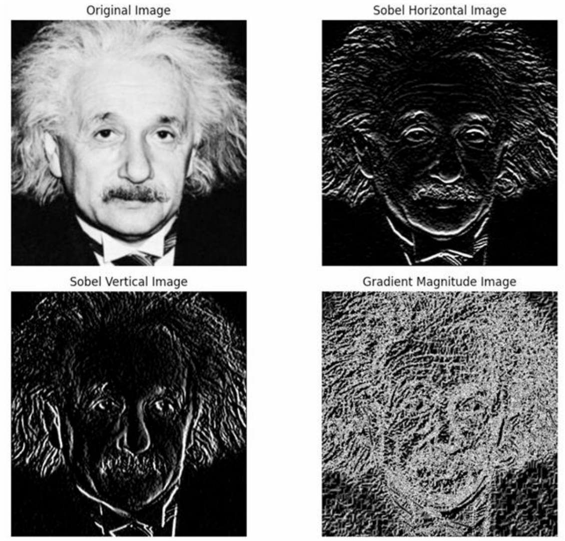
```python
rows, columns = im.shape
kernel_size = 3
sobel_v = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]],
dtype='float32')
sobel_h = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]],
dtype='float32')

imv = np.zeros((rows - kernel_size + 1, columns - kernel_size + 1),
dtype='float32')
imh = np.zeros((rows - kernel_size + 1, columns - kernel_size + 1),
dtype='float32')

for row in range(rows - kernel_size + 1):
    for column in range(columns - kernel_size + 1):
        imv[row, column] = np.sum(im[row:row + kernel_size,
column:column + kernel_size] * sobel_v)
        imh[row, column] = np.sum(im[row:row + kernel_size,
column:column + kernel_size] * sobel_h)

grad_mag = np.sqrt(imv**2 + imh**2)
```



C)

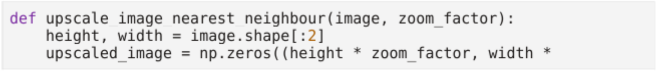


```python
sobel_h_kernel = np.array([1, 2, 1], dtype=np.float32)
sobel_v_kernel = np.array([1, 0, -1], dtype=np.float32)

im_h = cv.sepFilter2D(im, -1, sobel_h_kernel, sobel_v_kernel)
im_v = cv.sepFilter2D(im, -1, sobel_v_kernel, sobel_h_kernel)
grad_mag = np.sqrt(im_h**2 + im_v**2)
```

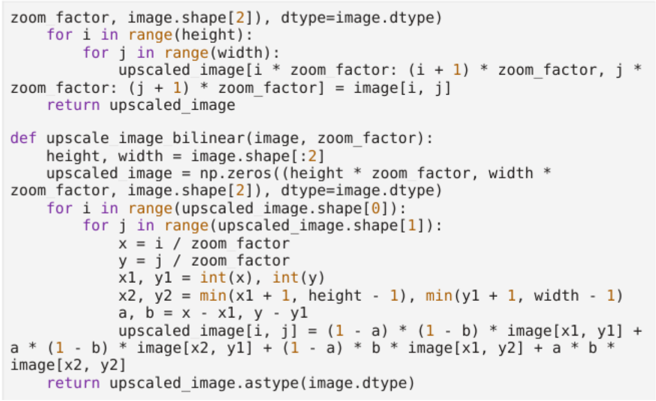


- The vertical and horizontal gradients are combined to calculate the overall gradient at each pixel.

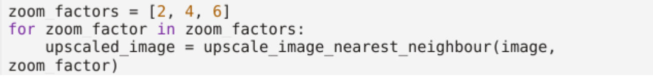
Question 08



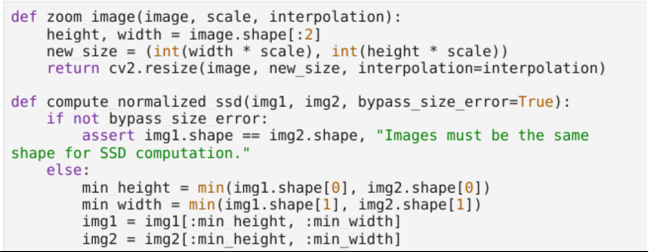
```
def upscale_image_nearest_neighbour(image, zoom_factor):
    height, width = image.shape[:2]
    upscaled_image = np.zeros((height * zoom_factor, width *
```



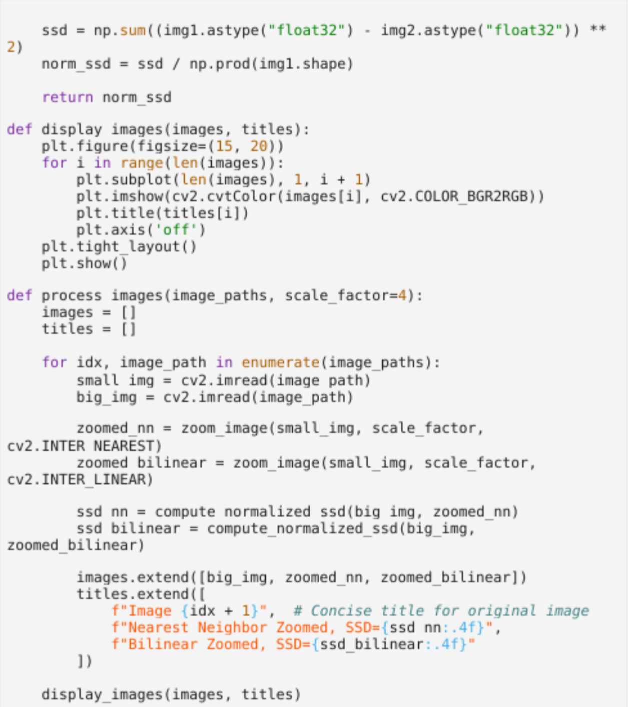
```
zoom_factor, image.shape[2]), dtype=image.dtype)
    for i in range(height):
        for j in range(width):
            upscaled_image[i * zoom_factor: (i + 1) * zoom_factor, j *
zoom_factor: (j + 1) * zoom_factor] = image[i, j]
    return upscaled_image

def upscale_image_bilinear(image, zoom_factor):
    height, width = image.shape[:2]
    upscaled_image = np.zeros((height * zoom_factor, width *
zoom_factor, image.shape[2]), dtype=image.dtype)
    for i in range(upscaled_image.shape[0]):
        for j in range(upscaled_image.shape[1]):
            x = i / zoom_factor
            y = j / zoom_factor
            x1, y1 = int(x), int(y)
            x2, y2 = min(x1 + 1, height - 1), min(y1 + 1, width - 1)
            a, b = x - x1, y - y1
            upscaled_image[i, j] = (1 - a) * (1 - b) * image[x1, y1] +
a * (1 - b) * image[x2, y1] + (1 - a) * b * image[x1, y2] + a * b *
image[x2, y2]
    return upscaled_image.astype(image.dtype)
```



```
zoom_factors = [2, 4, 6]
for zoom_factor in zoom_factors:
    upscaled_image = upscale_image_nearest_neighbour(image,
zoom_factor)
```



```
def zoom_image(image, scale, interpolation):
    height, width = image.shape[:2]
    new_size = (int(width * scale), int(height * scale))
    return cv2.resize(image, new_size, interpolation=interpolation)

def compute_normalized_ssd(img1, img2, bypass_size_error=True):
    if not bypass_size_error:
        assert img1.shape == img2.shape, "Images must be the same
shape for SSD computation."
    else:
        min_height = min(img1.shape[0], img2.shape[0])
        min_width = min(img1.shape[1], img2.shape[1])
        img1 = img1[:min_height, :min_width]
        img2 = img2[:min_height, :min_width]
```



```
    ssd = np.sum((img1.astype("float32") - img2.astype("float32")) **
2)
    norm_ssd = ssd / np.prod(img1.shape)

    return norm_ssd

def display_images(images, titles):
    plt.figure(figsize=(15, 20))
    for i in range(len(images)):
        plt.subplot(len(images), 1, i + 1)
        plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB))
        plt.title(titles[i])
        plt.axis('off')
    plt.tight_layout()
    plt.show()

def process_images(image_paths, scale_factor=4):
    images = []
    titles = []

    for idx, image_path in enumerate(image_paths):
        small_img = cv2.imread(image_path)
        big_img = cv2.imread(image_path)

        zoomed_nn = zoom_image(small_img, scale_factor,
cv2.INTER_NEAREST)
        zoomed_bilinear = zoom_image(small_img, scale_factor,
cv2.INTER_LINEAR)

        ssd_nn = compute_normalized_ssd(big_img, zoomed_nn)
        ssd_bilinear = compute_normalized_ssd(big_img,
zoomed_bilinear)

        images.extend([big_img, zoomed_nn, zoomed_bilinear])
        titles.extend([
            f"Image {idx + 1}",  # Concise title for original image
            f"Nearest Neighbor Zoomed, SSD={ssd_nn:.4f}",
            f"Bilinear Zoomed, SSD={ssd_bilinear:.4f}"
        ])

    display_images(images, titles)
```

Original Image
Upscaled Image (Nearest Neighbour) Factor: 2
Original Image
Upscaled Image (Bilinear) Factor: 2
Original Image
Upscaled Image (Nearest Neighbour) Factor: 4
Original Image
Upscaled Image (Bilinear) Factor: 4

Original Image
Upscaled Image (Nearest Neighbour) Factor: 6
Original Image
Upscaled Image (Bilinear) Factor: 6

Explanation: Bilinear interpolation produces better results than nearest neighbor interpolation across all images. Nearest neighbor can lead to blocky artifacts in the zoomed image because it only uses the closest pixel value without considering surrounding pixels. However, bilinear interpolation provides smoother transitions but may introduce some blurring and artifacts since it assumes a linear change between pixel values. When consider SSD values we can't see much difference between both methods.

```
process_images(image_paths)
```

Here are calculated SSD values:

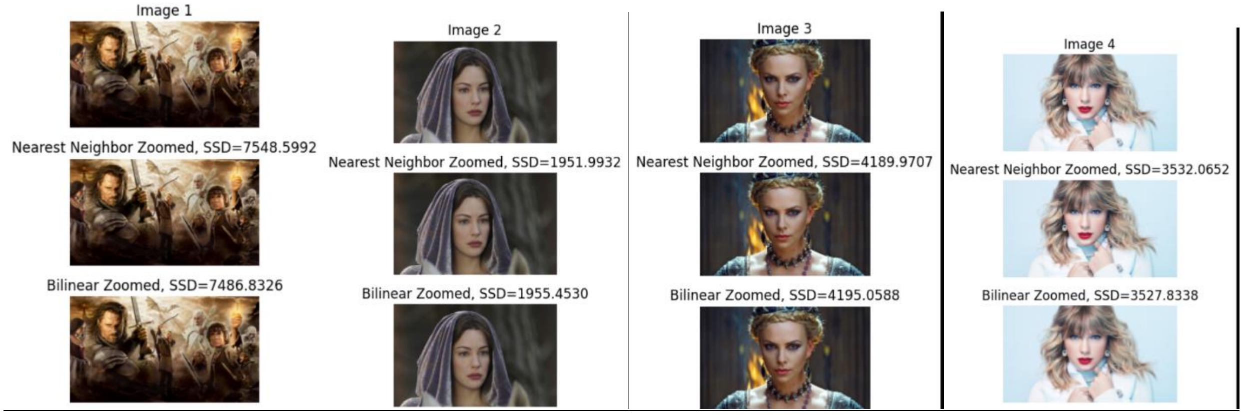


Image 1

Nearest Neighbor Zoomed, SSD=7548.5992

Bilinear Zoomed, SSD=7486.8326

Image 2

Nearest Neighbor Zoomed, SSD=1951.9932

Bilinear Zoomed, SSD=1955.4530

Image 3

Nearest Neighbor Zoomed, SSD=4189.9707

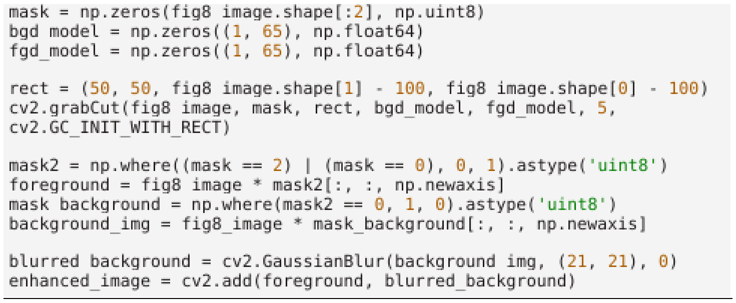
Bilinear Zoomed, SSD=4195.0588

Image 4

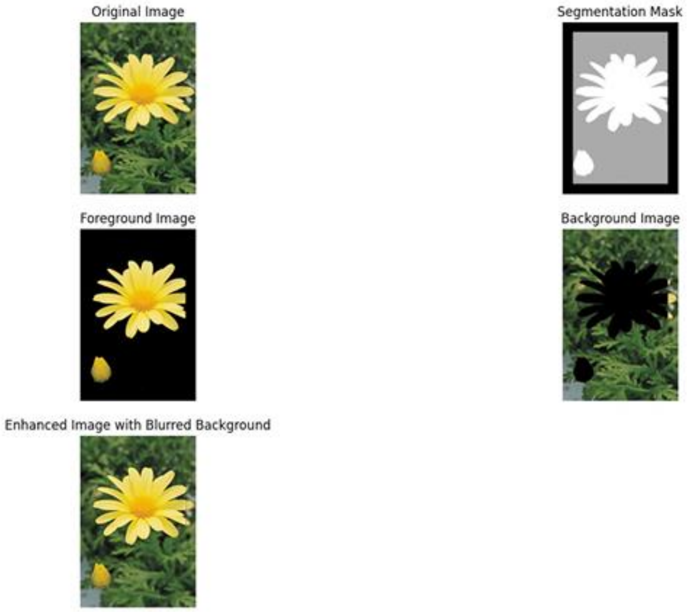
Nearest Neighbor Zoomed, SSD=3532.0652

Bilinear Zoomed, SSD=3527.8338

Question 09



```
mask = np.zeros(fig8_image.shape[:2], np.uint8)
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)

rect = (50, 50, fig8_image.shape[1] - 100, fig8_image.shape[0] - 100)
cv2.grabCut(fig8_image, mask, rect, bgd_model, fgd_model, 5,
cv2.GC_INIT_WITH_RECT)

mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
foreground = fig8_image * mask2[:, :, np.newaxis]
mask_background = np.where(mask2 == 0, 1, 0).astype('uint8')
background_img = fig8_image * mask_background[:, :, np.newaxis]

blurred_background = cv2.GaussianBlur(background_img, (21, 21), 0)
enhanced_image = cv2.add(foreground, blurred_background)
```



Original Image

Segmentation Mask

Foreground Image

Background Image

Enhanced Image with Blurred Background

Explanation: During Grab Cut segmentation, two masks are created: one for the foreground (the flower) and one for the background. Sometimes, pixels near the boundary might be incorrectly classified as background. When blurring is applied, it affects these boundary pixels, mixing them with the true background and making the edge appear darker. These pixels are set to zero in the mask, which results in dark pixels when the background is blurred and combined with the foreground.

****