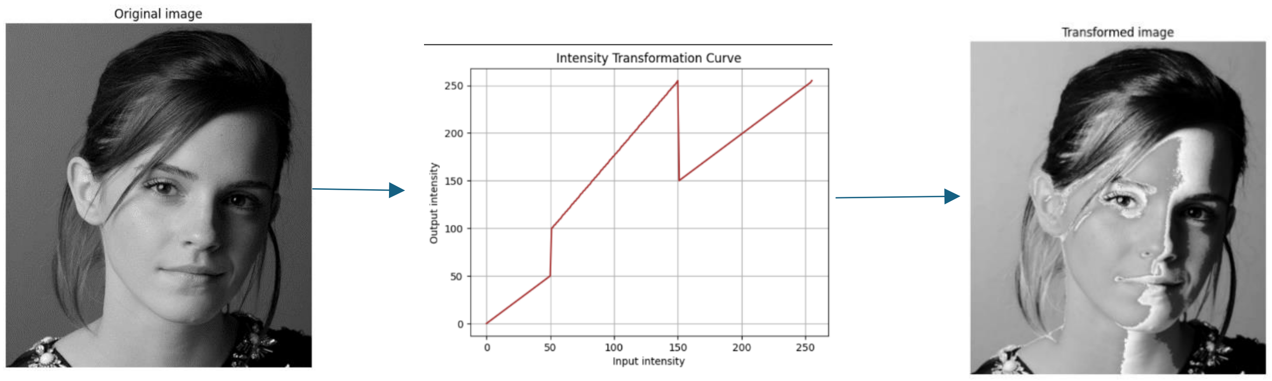
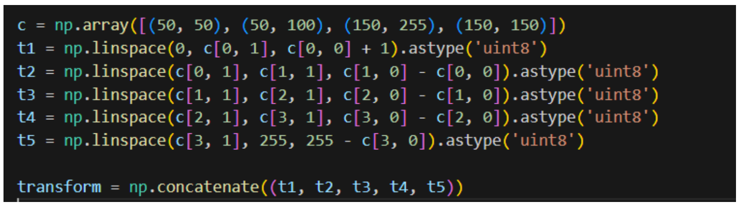
Fernando W.W.R.N.S. – 210169L

GITHUB LINK : https://github.com/rnsfernando/EN3160-Image_Processing_and_Machine_Vision

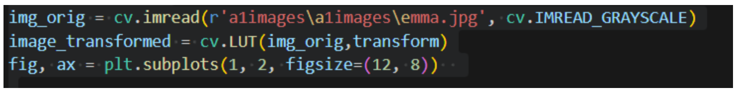
Question 1





```
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150)])
t1 = np.linspace(0, c[0, 1], c[0, 0] + 1).astype('uint8')
t2 = np.linspace(c[0, 1], c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1], c[2, 1], c[2, 0] - c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1], c[3, 1], c[3, 0] - c[2, 0]).astype('uint8')
t5 = np.linspace(c[3, 1], 255, 255 - c[3, 0]).astype('uint8')

transform = np.concatenate((t1, t2, t3, t4, t5))
```

Defining the transformation

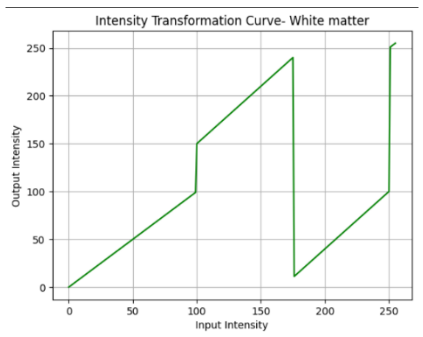


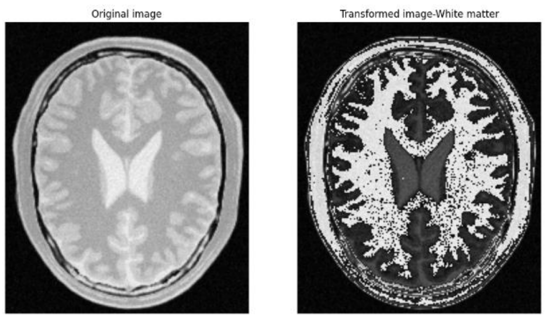
```
img_orig = cv.imread(r'a1images\a1images\emma.jpg', cv.IMREAD_GRAYSCALE)
image_transformed = cv.LUT(img_orig,transform)
fig, ax = plt.subplots(1, 2, figsize=(12, 8))
```

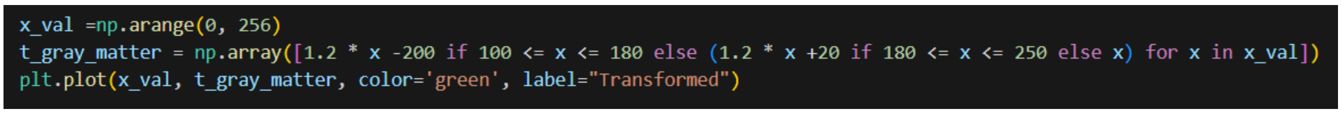
Applying the transformation

Question 2 - Intensity Transformation for Accentuating White and Gray Matter in Brain Proton Density Images

White matter mask:

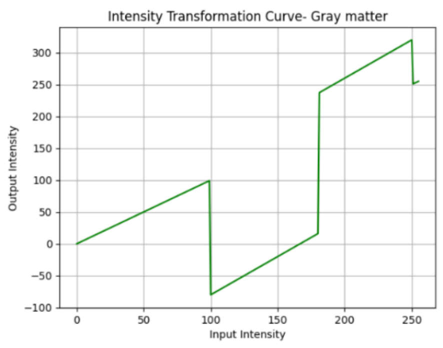


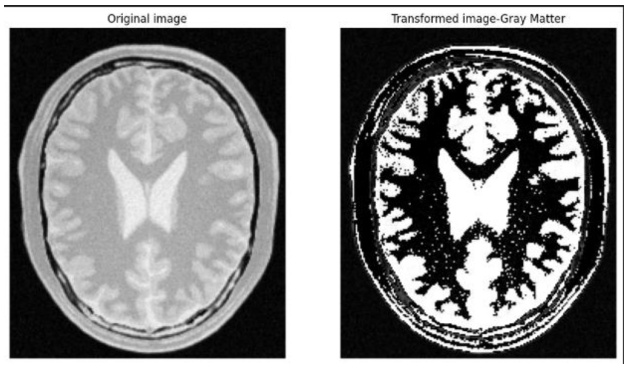


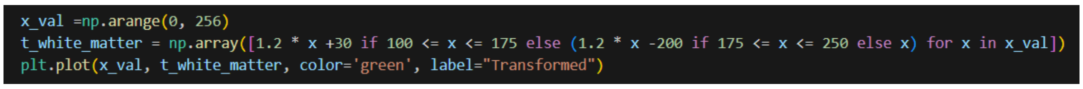


```
x_val =np.arange(0, 256)
t_gray_matter = np.array([1.2 * x -200 if 100 <= x <= 180 else (1.2 * x +20 if 180 <= x <= 250 else x) for x in x_val])
plt.plot(x_val, t_gray_matter, color='green', label="Transformed")
```

Gray matter mask:







```
x_val =np.arange(0, 256)
t_white_matter = np.array([1.2 * x +30 if 100 <= x <= 175 else (1.2 * x -200 if 175 <= x <= 250 else x) for x in x_val])
plt.plot(x_val, t_white_matter, color='green', label="Transformed")
```

- White Matter Accentuated Image:

Input intensities in the range [100, 175] were increased, while higher intensities (up to 250) were scaled differently. This transformation highlighted the white matter by increasing the contrast in these regions.

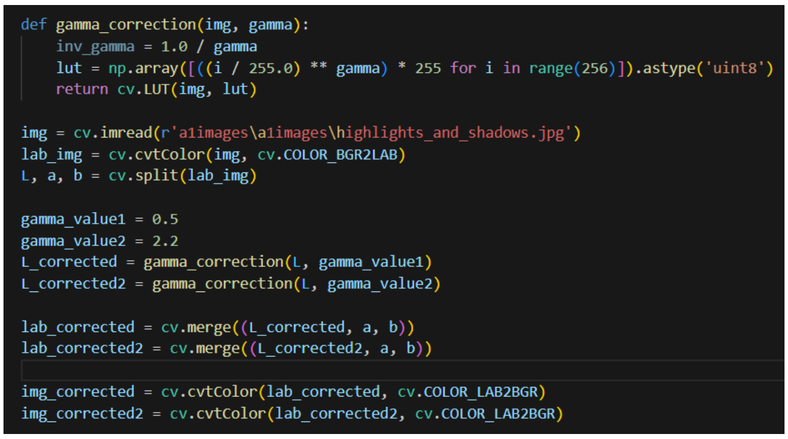
- Gray Matter Accentuated Image:

Input intensities in the range [100, 180] were adjusted to enhance gray matter, resulting in an image where gray matter structures are more pronounced.

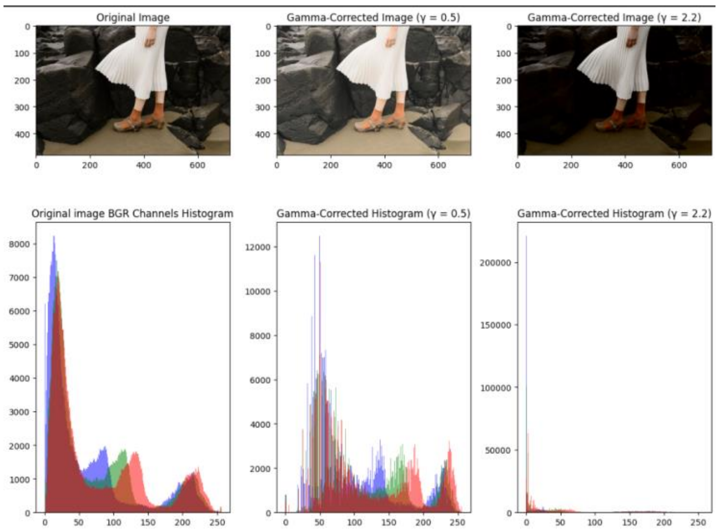
The transformations successfully highlighted the respective brain tissues, making white and gray matter regions more distinguishable in the images. The transformation functions were carefully designed to manipulate specific intensity ranges corresponding to each matter type.

Question 3 - Gamma Correction and Histogram Analysis

This section details the application of gamma correction to the luminance (L) channel of an image in the L*a*b* color space to enhance brightness and contrast. Two gamma values were used: γ=0.5 and γ=2.2. The image processing involved converting the image from BGR to L*a*b* color space, followed by separating the L, a, and b channels. The L channel underwent gamma correction using two values to enhance brightness and contrast. Finally, the corrected L channel was merged with the original a and b channels and converted back to BGR for display.



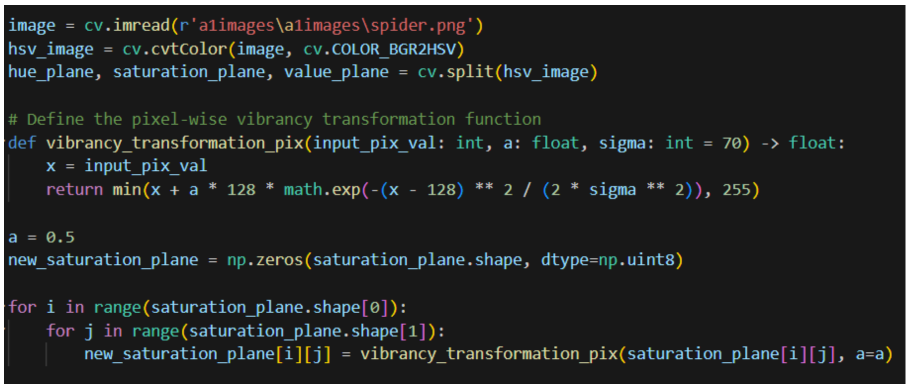
```
def gamma_correction(img, gamma):
    inv_gamma = 1.0 / gamma
    lut = np.array([((i / 255.0) ** gamma) * 255 for i in range(256)]).astype('uint8')
    return cv.LUT(img, lut)

img = cv.imread(r'a1images\a1images\highlights_and_shadows.jpg')
lab_img = cv.cvtColor(img, cv.COLOR_BGR2LAB)
L, a, b = cv.split(lab_img)

gamma_value1 = 0.5
gamma_value2 = 2.2
L_corrected = gamma_correction(L, gamma_value1)
L_corrected2 = gamma_correction(L, gamma_value2)

lab_corrected = cv.merge((L_corrected, a, b))
lab_corrected2 = cv.merge((L_corrected2, a, b))

img_corrected = cv.cvtColor(lab_corrected, cv.COLOR_LAB2BGR)
img_corrected2 = cv.cvtColor(lab_corrected2, cv.COLOR_LAB2BGR)
```



Gamma correction has a big effect on how bright and clear an image looks. For γ=0.5, the correction makes the image brighter, with most pixel values concentrated at the higher end. On the other hand, γ=2.2 increases contrast, spreading the pixel values more evenly and making details in the darker areas easier to see. Comparing the histograms of the original and gamma-corrected images shows these differences clearly: the histogram for γ=0.5 has more bright pixel values, while γ=2.2spreads the values more in darker pixel values. This shows how different gamma values can greatly change how an image looks.

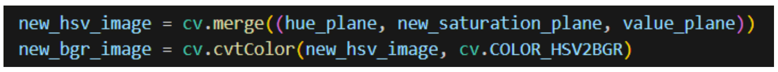
Question 4 - Vibrance Enhancement

Image Splitting, Intensity Transformation Function, Applying the Transformation:



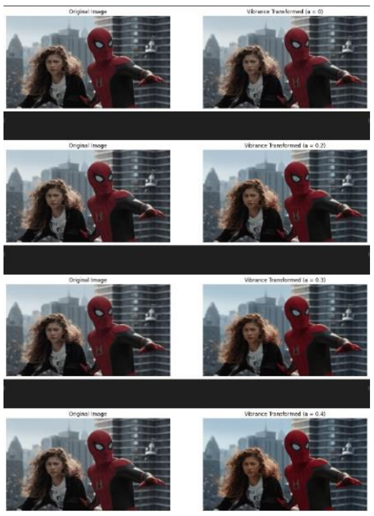
```python
image = cv.imread(r'a1images\a1images\spider.png')
hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)
hue_plane, saturation_plane, value_plane = cv.split(hsv_image)

# Define the pixel-wise vibrancy transformation function
def vibrancy_transformation_pix(input_pix_val: int, a: float, sigma: int = 70) -> float:
    x = input_pix_val
    return min(x + a * 128 * math.exp(-(x - 128) ** 2 / (2 * sigma ** 2)), 255)

a = 0.5
new_saturation_plane = np.zeros(saturation_plane.shape, dtype=np.uint8)

for i in range(saturation_plane.shape[0]):
    for j in range(saturation_plane.shape[1]):
        new_saturation_plane[i][j] = vibrancy_transformation_pix(saturation_plane[i][j], a=a)
```

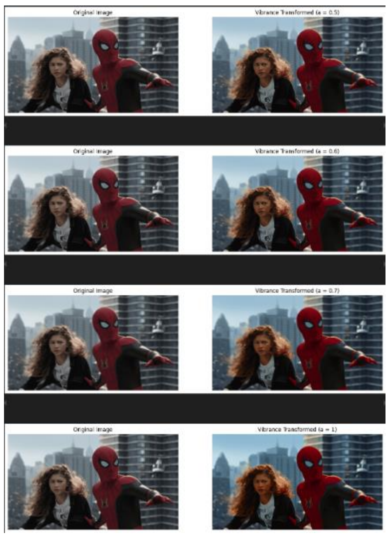
Merge and convert:



```python
new_hsv_image = cv.merge((hue_plane, new_saturation_plane, value_plane))
new_bgr_image = cv.cvtColor(new_hsv_image, cv.COLOR_HSV2BGR)
```

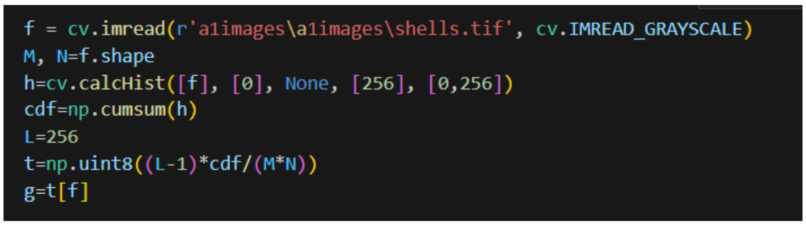
Results:

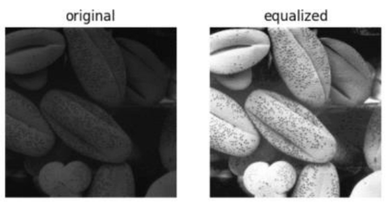


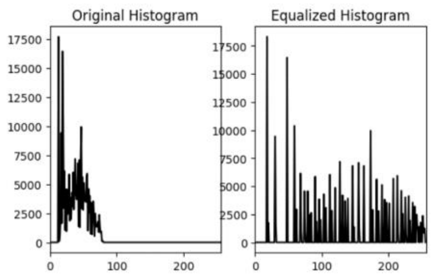


The results show how changing the value of ‘a’ affects the vibrance of the image. When ‘a’ is low, the increase in saturation is small and gentle. But when ‘a’ is high, the effect is much stronger. This comparison highlights the importance of finding the right balance in enhancing vibrance so that the image doesn’t become oversaturated, which can make the colors look unnatural. So, for this image the suitable ‘a’ value is 0.4 out of tested a values.

Question 5 - Histogram Equalization


```python
f = cv.imread(r'a1images\a1images\shells.tif', cv.IMREAD_GRAYSCALE)
M, N=f.shape
h=cv.calcHist([f], [0], None, [256], [0,256])
cdf=np.cumsum(h)
L=256
t=np.uint8((L-1)*cdf/(M*N))
g=t[f]
```





Comparison of Results:

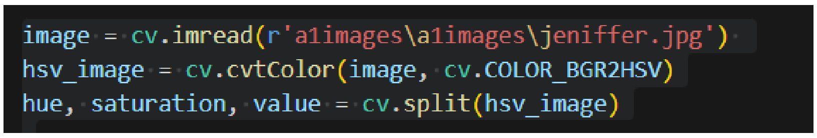
The original image has a narrow range of brightness levels, making it look less clear. After using histogram equalization, the new image shows a wider range of brightness values. This change improves the contrast and makes more details visible.

Interpretation of Results:

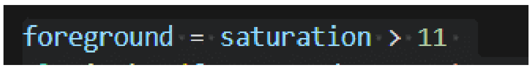
Histogram equalization spreads the brightness levels across the full range, which enhances the overall contrast of the image. The original histogram has peaks at certain brightness levels, meaning many pixels have similar brightness. In contrast, the equalized histogram is more even, which helps show finer details in the image.

Question 6 – Producing an image with a histogram equalized foreground.

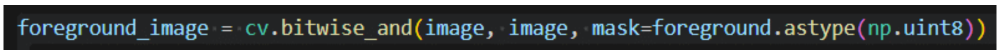
Load image and split into HSV planes:


```python
image = cv.imread(r'a1images\a1images\jeniffer.jpg')
hsv_image = cv.cvtColor(image, cv.COLOR_BGR2HSV)
hue, saturation, value = cv.split(hsv_image)
```

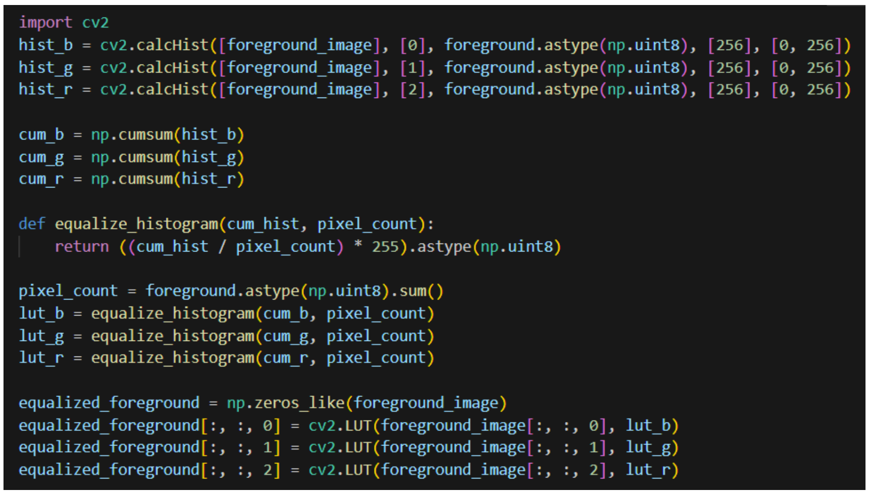
Create Foreground Mask:


```python
foreground = saturation > 11
```

Extract foreground:

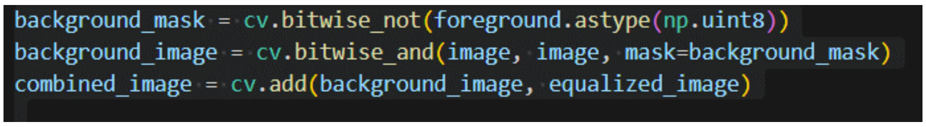

```python
foreground_image = cv.bitwise_and(image, image, mask=foreground.astype(np.uint8))
```

Cumulative histogram calculation and histogram equalization:

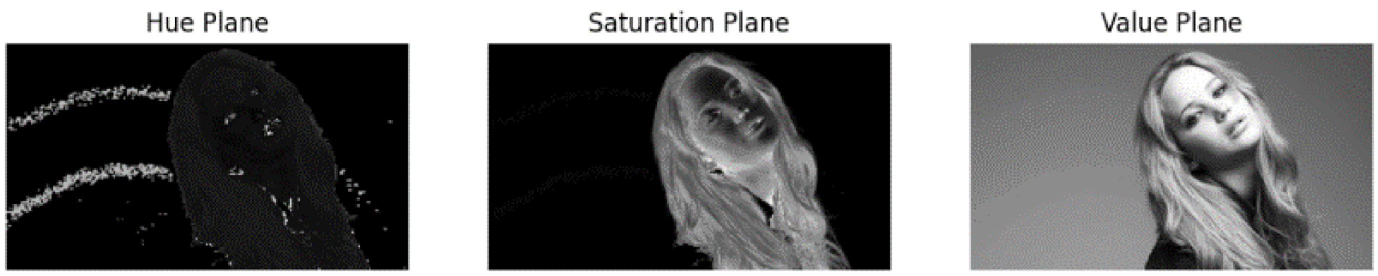


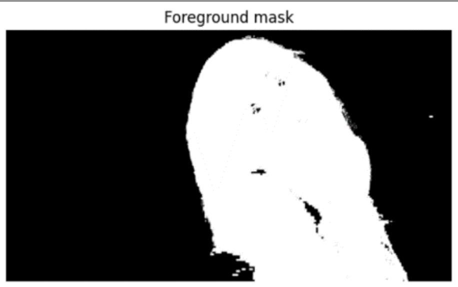
```
import cv2
hist_b = cv2.calcHist([foreground_image], [0], foreground.astype(np.uint8), [256], [0, 256])
hist_g = cv2.calcHist([foreground_image], [1], foreground.astype(np.uint8), [256], [0, 256])
hist_r = cv2.calcHist([foreground_image], [2], foreground.astype(np.uint8), [256], [0, 256])

cum_b = np.cumsum(hist_b)
cum_g = np.cumsum(hist_g)
cum_r = np.cumsum(hist_r)

def equalize_histogram(cum_hist, pixel_count):
    return ((cum_hist / pixel_count) * 255).astype(np.uint8)

pixel_count = foreground.astype(np.uint8).sum()
lut_b = equalize_histogram(cum_b, pixel_count)
lut_g = equalize_histogram(cum_g, pixel_count)
lut_r = equalize_histogram(cum_r, pixel_count)

equalized_foreground = np.zeros_like(foreground_image)
equalized_foreground[:, :, 0] = cv2.LUT(foreground_image[:, :, 0], lut_b)
equalized_foreground[:, :, 1] = cv2.LUT(foreground_image[:, :, 1], lut_g)
equalized_foreground[:, :, 2] = cv2.LUT(foreground_image[:, :, 2], lut_r)
```

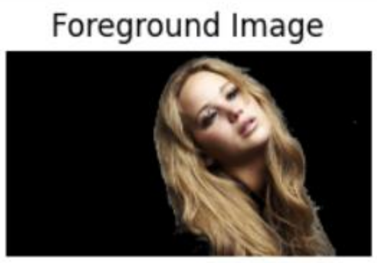
Combine foreground and background:

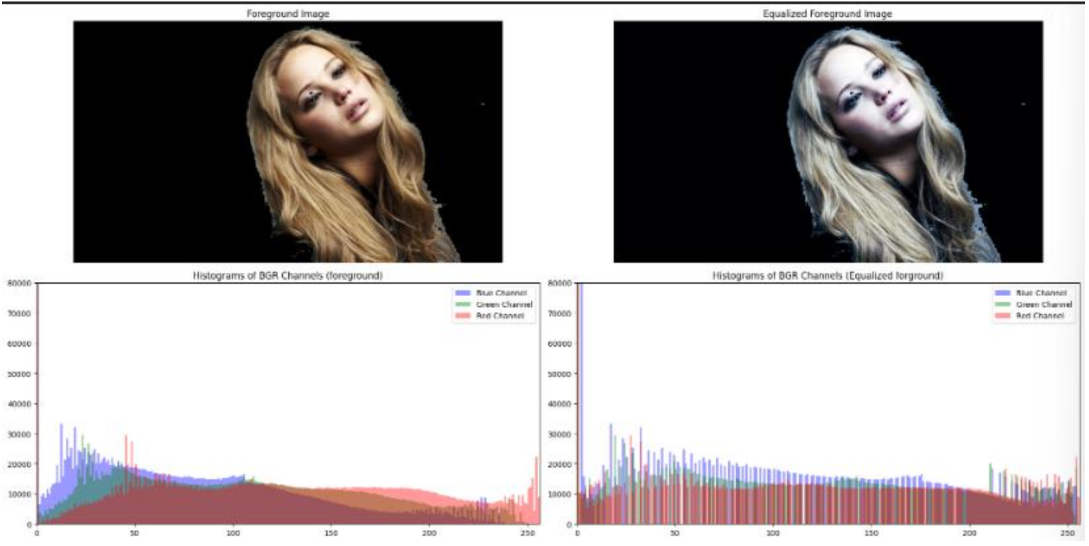


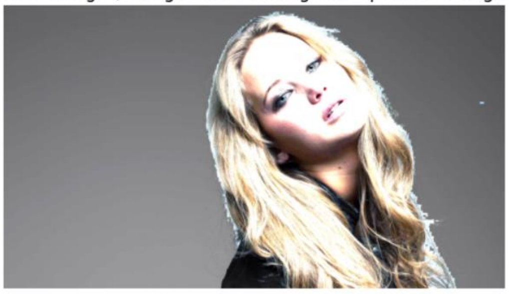
```
background_mask = cv.bitwise_not(foreground.astype(np.uint8))
background_image = cv.bitwise_and(image, image, mask=background_mask)
combined_image = cv.add(background_image, equalized_image)
```





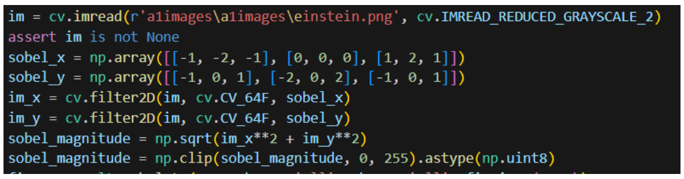




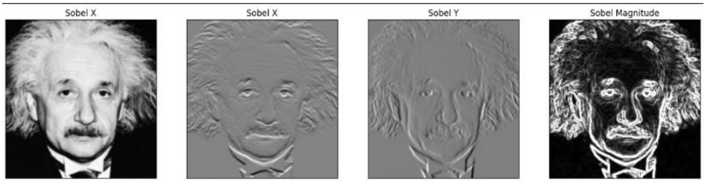

Combined Image (Background + Histogram Equalized Foreground)

Question 7 – Sobel operator

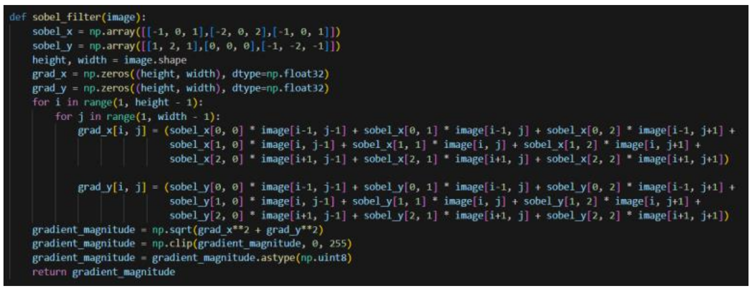
Using filter2D:



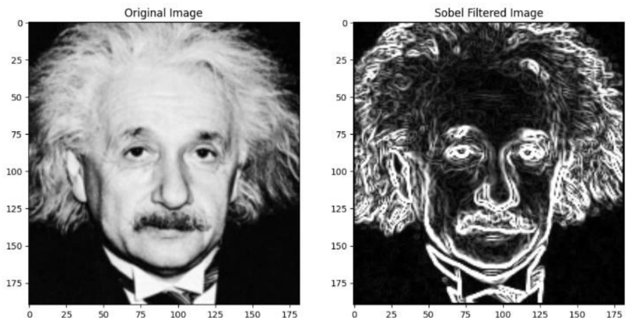
```
im = cv.imread(r'a1images\a1images\einstein.png', cv.IMREAD_REDUCED_GRAYSCALE_2)
assert im is not None
sobel_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
sobel_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
im_x = cv.filter2D(im, cv.CV_64F, sobel_x)
im_y = cv.filter2D(im, cv.CV_64F, sobel_y)
sobel_magnitude = np.sqrt(im_x**2 + im_y**2)
sobel_magnitude = np.clip(sobel_magnitude, 0, 255).astype(np.uint8)
```



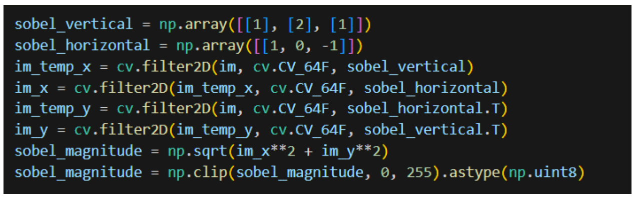
Using own code:



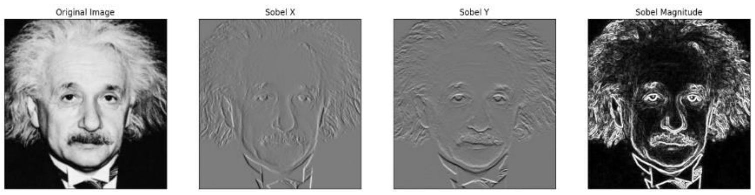
```
def sobel_filter(image):
    sobel_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
    sobel_y = np.array([[1, 2, 1],[0, 0, 0],[-1, -2, -1]])
    height, width = image.shape
    grad_x = np.zeros((height, width), dtype=np.float32)
    grad_y = np.zeros((height, width), dtype=np.float32)
    for i in range(1, height - 1):
        for j in range(1, width - 1):
            grad_x[i, j] = (sobel_x[0, 0] * image[i-1, j-1] + sobel_x[0, 1] * image[i-1, j] + sobel_x[0, 2] * image[i-1, j+1] +
                            sobel_x[1, 0] * image[i, j-1] + sobel_x[1, 1] * image[i, j] + sobel_x[1, 2] * image[i, j+1] +
                            sobel_x[2, 0] * image[i+1, j-1] + sobel_x[2, 1] * image[i+1, j] + sobel_x[2, 2] * image[i+1, j+1])

            grad_y[i, j] = (sobel_y[0, 0] * image[i-1, j-1] + sobel_y[0, 1] * image[i-1, j] + sobel_y[0, 2] * image[i-1, j+1] +
                            sobel_y[1, 0] * image[i, j-1] + sobel_y[1, 1] * image[i, j] + sobel_y[1, 2] * image[i, j+1] +
                            sobel_y[2, 0] * image[i+1, j-1] + sobel_y[2, 1] * image[i+1, j] + sobel_y[2, 2] * image[i+1, j+1])
    gradient_magnitude = np.sqrt(grad_x**2 + grad_y**2)
    gradient_magnitude = np.clip(gradient_magnitude, 0, 255)
    gradient_magnitude = gradient_magnitude.astype(np.uint8)
    return gradient_magnitude
```



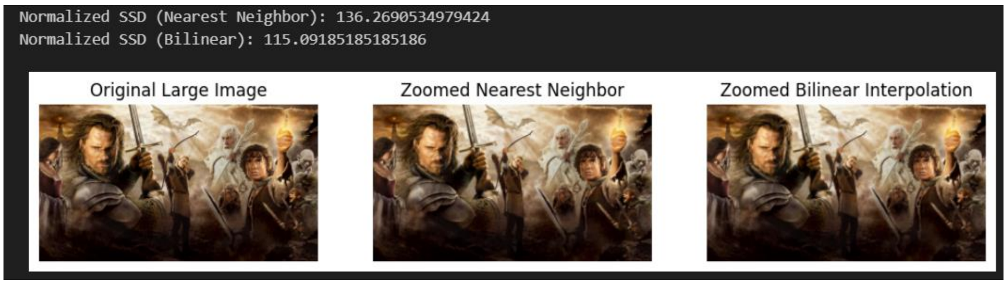
Using the property:



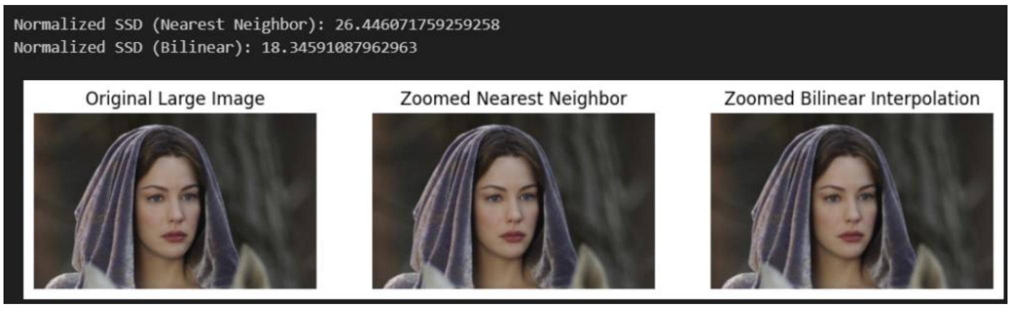
```
sobel_vertical = np.array([[1], [2], [1]])
sobel_horizontal = np.array([[1, 0, -1]])
im_temp_x = cv.filter2D(im, cv.CV_64F, sobel_vertical)
im_x = cv.filter2D(im_temp_x, cv.CV_64F, sobel_horizontal)
im_temp_y = cv.filter2D(im, cv.CV_64F, sobel_horizontal.T)
im_y = cv.filter2D(im_temp_y, cv.CV_64F, sobel_vertical.T)
sobel_magnitude = np.sqrt(im_x**2 + im_y**2)
sobel_magnitude = np.clip(sobel_magnitude, 0, 255).astype(np.uint8)
```



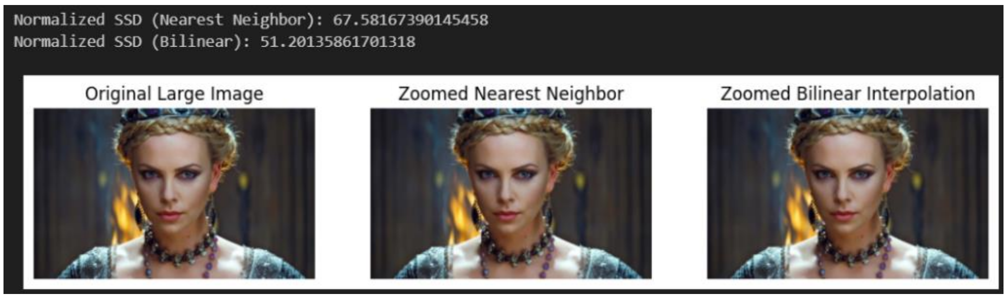
Question 8 – zoom images



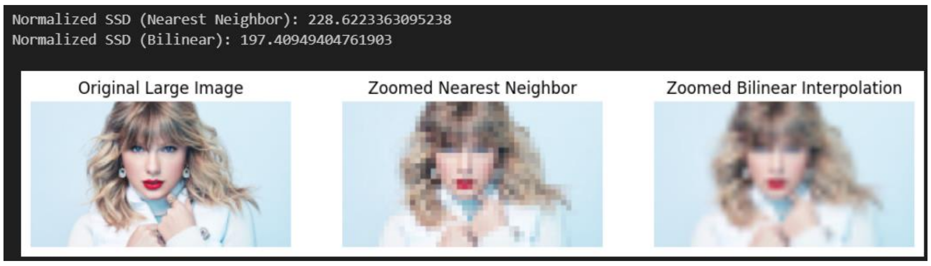
```
Normalized SSD (Nearest Neighbor): 136.2690534979424
Normalized SSD (Bilinear): 115.09185185185186
```



```
Normalized SSD (Nearest Neighbor): 26.446071759259258
Normalized SSD (Bilinear): 18.34591087962963
```



```
Normalized SSD (Nearest Neighbor): 67.58167390145458
Normalized SSD (Bilinear): 51.20135861701318
```

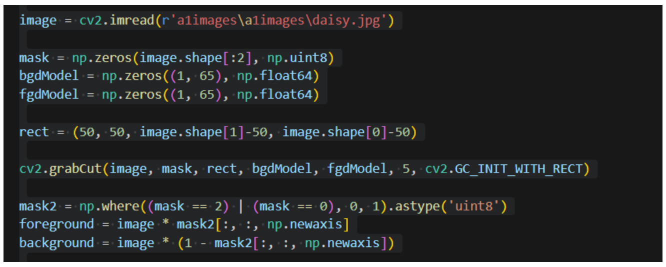


```
Normalized SSD (Nearest Neighbor): 228.6223363095238
Normalized SSD (Bilinear): 197.40949404761903
```

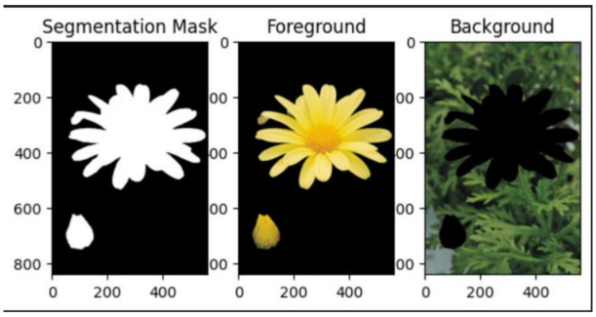
The results indicate that bilinear interpolation consistently produces smaller normalized SSD values than nearest-neighbor interpolation. This suggests that bilinear interpolation provides a closer approximation to the original image, resulting in better visual quality and less distortion after scaling.
Nearest-Neighbor Interpolation - Produces sharper edges but may introduce pixelation, leading to higher SSD values.
Bilinear Interpolation - Provides smoother transitions between pixels and generates a more visually accurate result with a lower SSD but may appear softer than nearest-neighbor interpolation.

Question 9 – Image Segmentation and Enhancement Using grabCut Algorithm

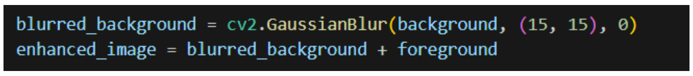
foreground extraction using grabCut:



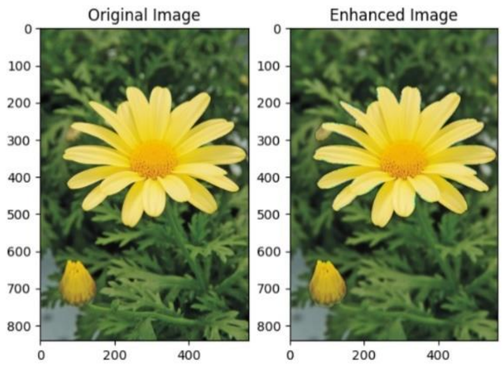
```python
image = cv2.imread(r'a1images\a1images\daisy.jpg')

mask = np.zeros(image.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

rect = (50, 50, image.shape[1]-50, image.shape[0]-50)

cv2.grabCut(image, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)

mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
foreground = image * mask2[:, :, np.newaxis]
background = image * (1 - mask2[:, :, np.newaxis])
```



Blurring the Background:



```python
blurred_background = cv2.GaussianBlur(background, (15, 15), 0)
enhanced_image = blurred_background + foreground
```



The dark edges around the flower in the enhanced image happen because of the blur effect. When the background is blurred, the computer averages the colors of nearby pixels. This includes the pixels at the edge of the flower, which are darker. As a result, the area just beyond the flower looks darker since the dark edges mix with the background during the blur.

If the background has shadows or areas with different brightness, the blur can make those differences stand out more. To avoid this, we can improve the separation between the flower and the background before applying the blur, which would make the edge cleaner.