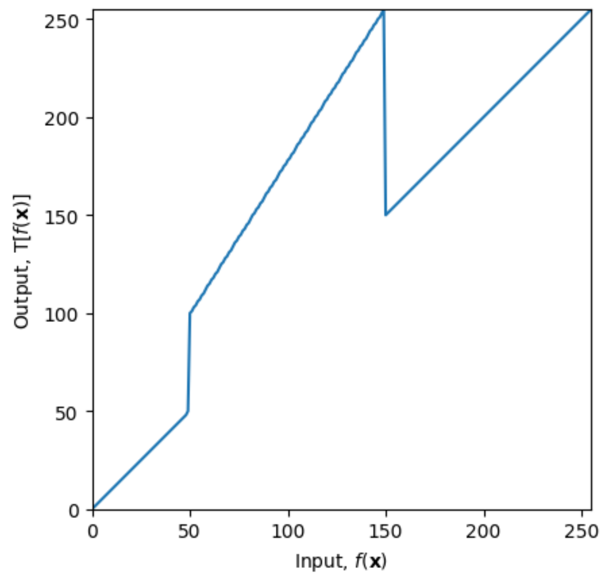
200500L

Question 1

```
t1 = np.linspace(0, 50, 50  - 0).astype('uint8')
print(len(t1))
t2 = np.linspace(100, 255, 150 - 50).astype('uint8')
print(len(t2))
t3 = np.linspace(150 , 255, 256 - 150).astype('uint8')
print(len(t3))
transform = np.concatenate((t1, t2), axis=0).astype('uint8')
transform = np.concatenate((transform, t3), axis=0).astype('uint8')
```



original

transformed

After applying the transformation, the pixel intensities originally falling within the range of 50 to 150 have been enhanced. Consequently, we can observe that these pixels, which initially had intensity values between 50 and 150, have become brighter.

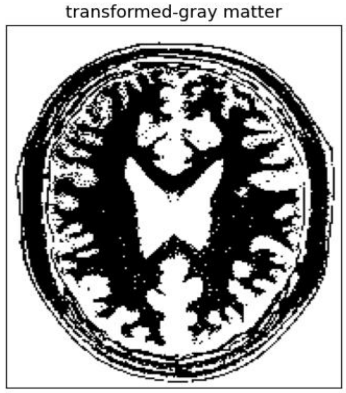
Question 2

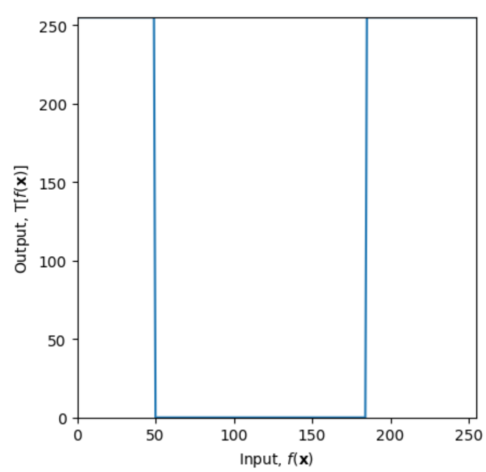
```
#for gray filtering
t1 = np.linspace(255, 255, 50  - 0).astype('uint8')
print(len(t1))
t2 = np.linspace(0, 0, 185 - 50).astype('uint8')
print(len(t2))
t3 = np.linspace(255 , 255, 256 - 185).astype('uint8')
print(len(t3))
```

```
#for wite filtering
tt1 = np.linspace(0, 0, 50  - 0).astype('uint8')
print(len(tt1))
tt2 = np.linspace(0, 0, 183 - 50).astype('uint8')
print(len(tt2))
tt3 = np.linspace(255 , 255, 256 - 183).astype('uint8')
print(len(tt3))
```

original

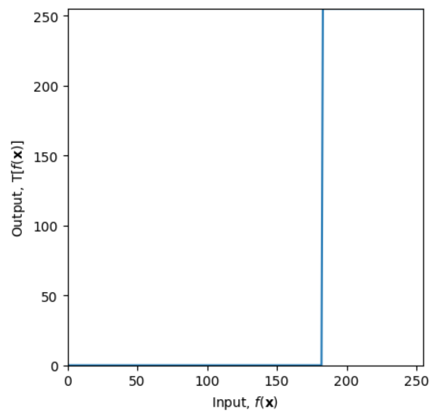
transformed-gray matter

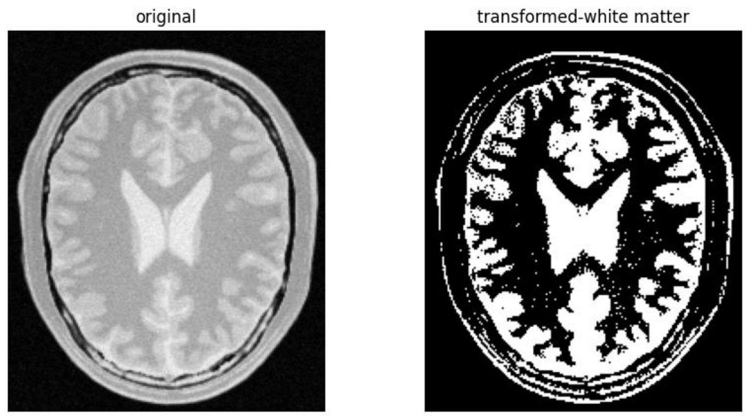




Git hub - [Jupyter note book]

200500L





intensity 185 used as the threshold for the gray and white diffrence

Question 3

```
gamma = 2.15
print('gamma = ', gamma)
table = np.array([(i/255.0)**(1.0/gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')
img_gamma = cv.LUT(img_orig, table)
```

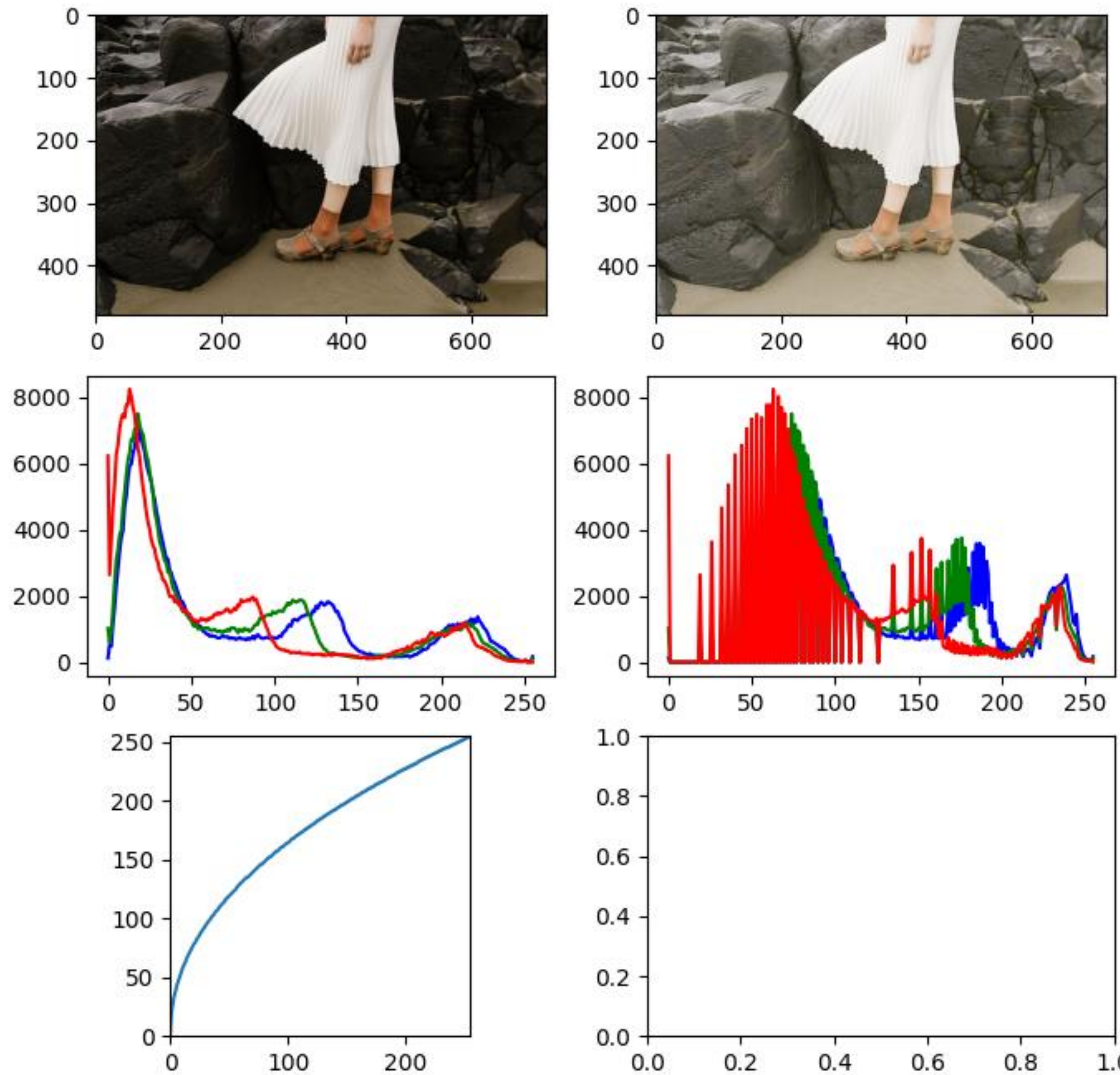
```
color = ('b', 'g', 'r')
for i, c in enumerate(color):
    hist_orig = cv.calcHist([img_orig], [i], None, [256], [0,256])
    axarr[1,0].plot(hist_orig, color = c)
    hist_gamma = cv.calcHist([img_gamma], [i], None, [256], [0,256])
    axarr[1,1].plot(hist_gamma, color = c)
```

gamma = 2.15

Gamma correction, expressed as g = f^(γ), adjusts image intensity. The original image (f) has many dark pixels, obscuring details and edges. Gamma correction expands the bright pixel range, making them brighter, and narrows the dark pixel range, making them darker. This boosts overall brightness and reveals hidden features.

Git hub - [Jupyter note book]()

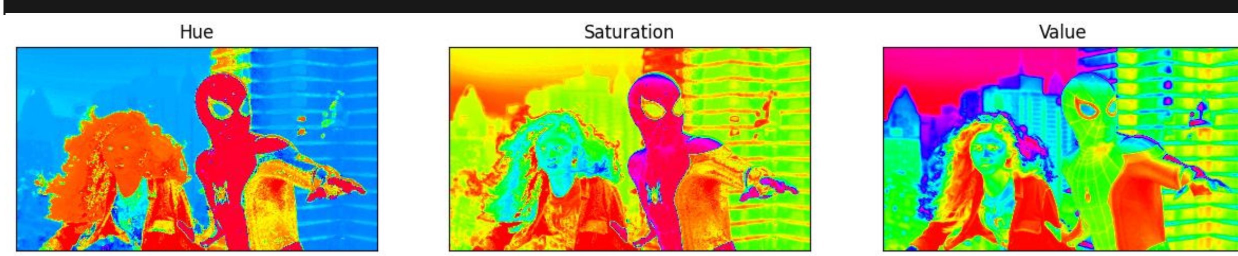
200500L



Question 4

```python
# Split the HSV image into individual planes
hue_plane = hsv_image[:, :, 0]
saturation_plane = hsv_image[:, :, 1]
value_plane = hsv_image[:, :, 2]
```

```python
a=0.8
sig=70
transform_values = [min(255, i + a * (128) * np.exp(-((i - 128) ** 2) / ((2 * sig )** 2))) for i in range(256)]
```



Git hub - [Jupyter note book]()

200500L

original

transformed

Split image into hue, saturation, and value planes. Transform only saturation, brightening bright pixels. Merge planes for vibrant, colorful image.

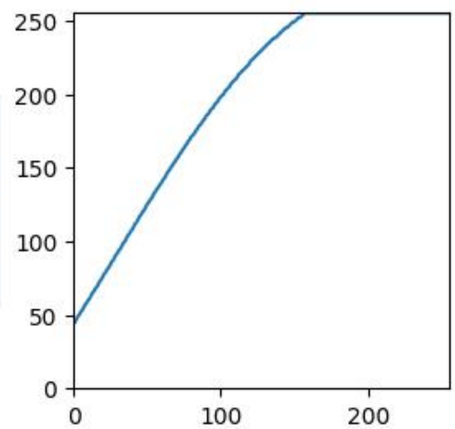
transformed

```
a =  0.8
```

Chosen value for a = 0.8

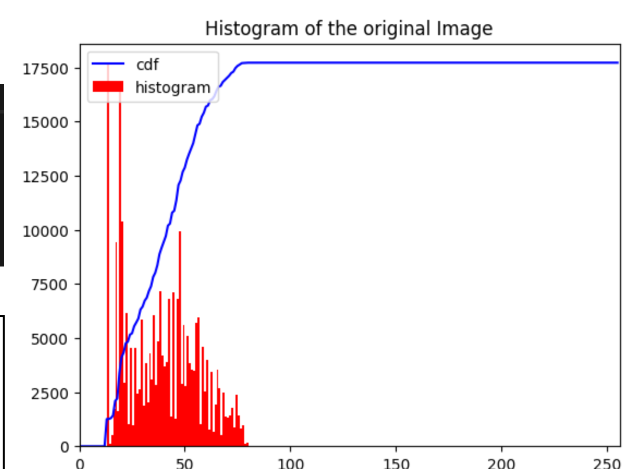
original

transformed



Quection5

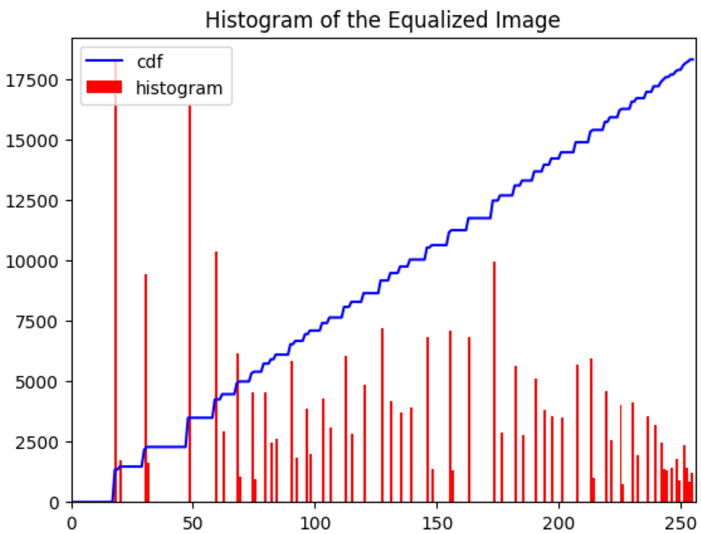
```
def custom_histogram_equalization(image_gray):
    hist , bins = np.histogram(image_gray.ravel(),256,[0,256])
    M,N = image_gray.shape
    Transformation = (255*((hist.cumsum())/(M*N))).astype(np.uint8)
    equalized_image = Transformation[image_gray]
```



Histogram equalization enhances images. The original histogram is dark and concentrated. Equalization spreads it uniformly. The image becomes brighter and vibrant, making it more appealing.

Git hub - [Jupyter note book]

200500L



Original Grayscale Image

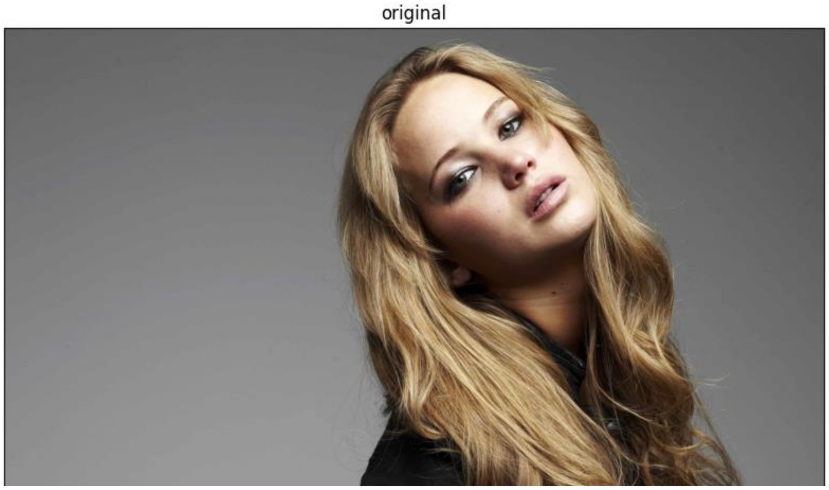
Equalized Image

Question 6

```
threshold_value = 25
_, foreground_mask = cv.threshold(saturation_plane, threshold_value, 255, cv2.THRESH_BINARY)
# _ is used as a variable name to essentially "discard" or ignore a value that is returned by a function but is not intended
foreground = cv.bitwise_and(saturation_plane, saturation_plane, mask=foreground_mask)
hist_foreground, _ = np.histogram(foreground.flatten(), bins=256, range=[0, 256])
cum = hist_foreground.cumsum()
equ_foreground = cv.equalizeHist(foreground)
hist_equ_foreground, _ = np.histogram(equ_foreground.flatten(), bins=256, range=[0, 256])
```

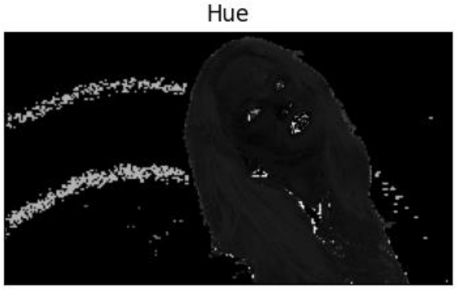
```
murged = cv.merge((hue_plane, equ_foreground, value_plane))
murged = cv.cvtColor(murged, cv.COLOR_HSV2RGB)
fig,ax = plt.subplots(nrows=1, ncols=2, figsize=(10,5))
```

original

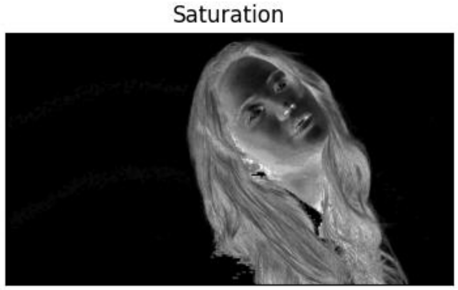


Thresholding selects a color plane with clear background-foreground differences, often saturation. It uses a chosen threshold to create a mask, separating foreground from background using bitwise 'and' operation.

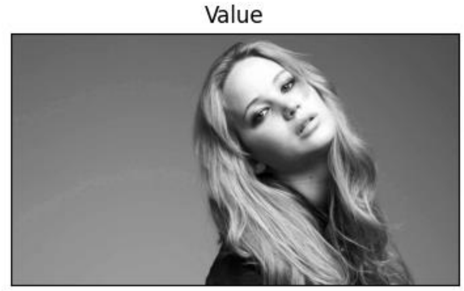
Hue



Saturation

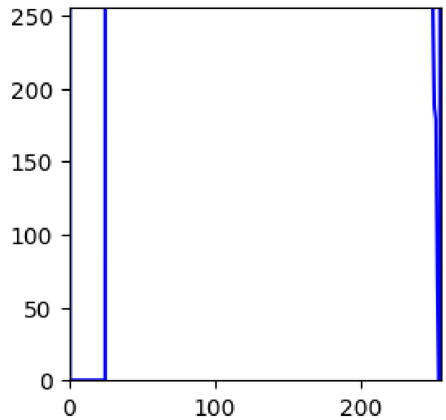


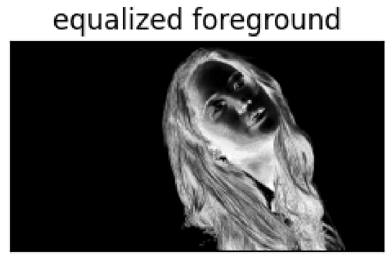
Value



200500L

```
saturation plane suitable for create a mask
```





Question 7

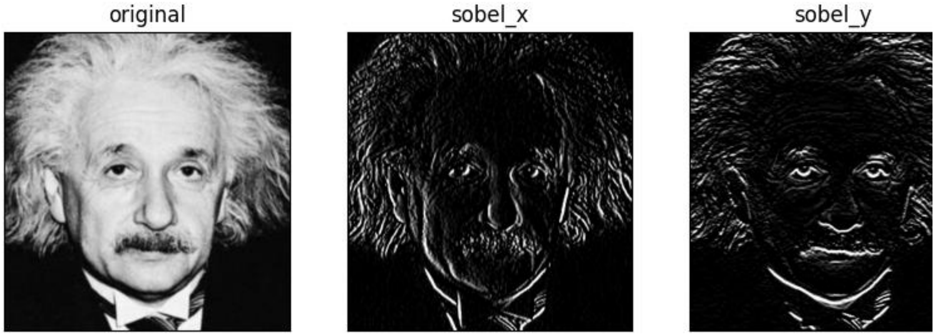
```python
# Apply the Sobel filter using filter2D
sobel_x = cv2.filter2D(image, -1, sobel_kernel_x)
sobel_y = cv2.filter2D(image, -1, sobel_kernel_y)
```

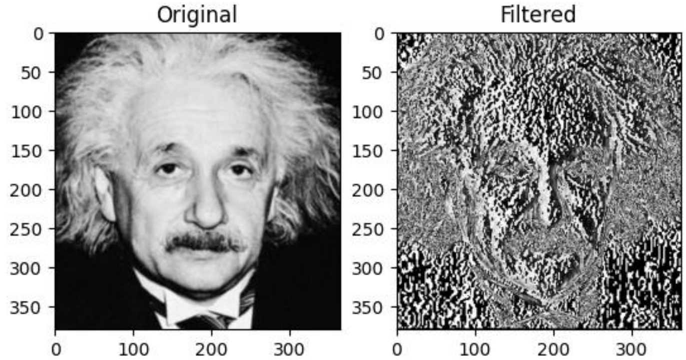
```python
# Create Sobel kernels for horizontal and vertical edges
sobel_kernel_x = np.array([[-1, 0, 1],
                           [-2, 0, 2],
                           [-1, 0, 1]])

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

```python
# Using the property
def generate_kernel(matrix_A, matrix_B):
    kernel = np.dot(matrix_A, matrix_B)
    kernel = kernel / np.sum(kernel)
    return kernel

def filter_with_kernel(image, kernel):
    h, w = image.shape
    image_float = cv.normalize(image.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)
    result = np.zeros(image.shape, 'float')

    k_hh, k_hw = divmod(kernel.shape[0], 2), divmod(kernel.shape[1], 2)

    # Pad the image with border reflections
    image_padded = cv.copyMakeBorder(image_float, k_hh[0], k_hh[0], k_hw[0], k_hw[1], cv.BORDER_REFLECT)

    # Perform convolution using vectorized operations
    for m in range(k_hh[0], h - k_hh[0]):
        for n in range(k_hw[0], w - k_hw[0]):
            region = image_padded[m - k_hh[0]:m + k_hh[0] + 1, n - k_hw[0]:n + k_hw[0] + 1]
            result[m, n] = np.sum(region * kernel)

    return result
```


```python
def filter(image, kernel):
    assert kernel.shape[0]%2 == 1 and kernel.shape[1]%2 == 1
    k_hh, k_hw = math.floor(kernel.shape[0]/2), math.floor(kernel.shape[1]/2)
    h, w = image.shape
    image_float = cv.normalize(image.astype('float'), None, 0.0, 1.0, cv.NORM_MINMAX)
    result = np.zeros(image.shape, 'float')

    for m in range(k_hh, h - k_hh):
        for n in range(k_hw, w - k_hw):
            result[m,n] = np.dot(image_float[m-k_hh:m + k_hh + 1, n - k_hw : n + k_hw + 1].flatten(), kernel.flatten())
    return result
```

Sobel filters find edges: horizontal detects vertical edges, while vertical detects horizontal edges.



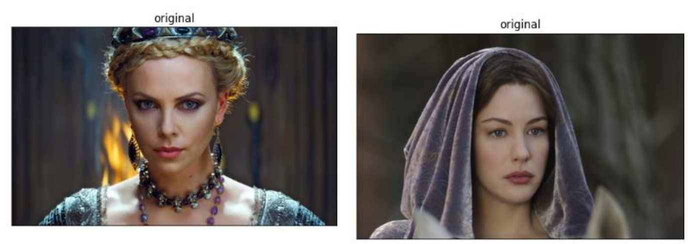


question 8

```python
def resize_image_nearest_neighbor(image, n):
    old_height, old_width, _ = image.shape

    # Calculate the new dimensions
    new_height = old_height * n
    new_width = old_width * n

    # Create a new empty image with the desired dimensions
    new_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)

    for i in range(new_height):
        for j in range(new_width):
            x = int(j / n)
            y = int(i / n)
            new_image[i, j, :] = image[y, x, :]

    return new_image
```

```python
def resize_bilinear(image, n):
    old_height, old_width, _ = image.shape
    # Calculate the new dimensions
    new_height = int(old_height * n)
    new_width = int(old_width * n)
    # Create a new empty image with the desired dimensions
    new_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)
    for i in range(new_height):
        for j in range(new_width):
            # Calculate the corresponding coordinates in the original image
            x = j / n
            y = i / n
            # Calculate the four nearest neighbors
            x1, y1 = int(np.floor(x)), int(np.floor(y))
            x2, y2 = min(x1 + 1, old_width - 1), min(y1 + 1, old_height - 1)
            # Bilinear interpolation
            dx = x - x1
            dy = y - y1
            top_left = image[y1, x1]
            top_right = image[y1, x2]
            bottom_left = image[y2, x1]
            bottom_right = image[y2, x2]
            new_pixel = (1 - dx) * (1 - dy) * top_left + dx * (1 - dy) * top_right + \
                        (1 - dx) * dy * bottom_left + dx * dy * bottom_right

            new_image[i, j, :] = new_pixel.astype(np.uint8)

    return new_image
```

```python
def normalized_ssd(image1, image2):
    assert image1.shape[-1] == image2.shape[-1], "Both images should have the same
    # Get the dimensions of both images
    h1, w1 = image1.shape[:2]
    h2, w2 = image2.shape[:2]
    # Determine the size of the overlapping region
    h_overlap = min(h1, h2)
    w_overlap = min(w1, w2)
    image1_cropped = image1[:h_overlap, :w_overlap]
    image2_cropped = image2[:h_overlap, :w_overlap]
    # Calculate the sum of squared differences
    ssd = np.sum((image1_cropped - image2_cropped) ** 2)
    normalized_ssd = ssd / (h_overlap * w_overlap * image1.shape[-1])
    return normalized_ssd
```

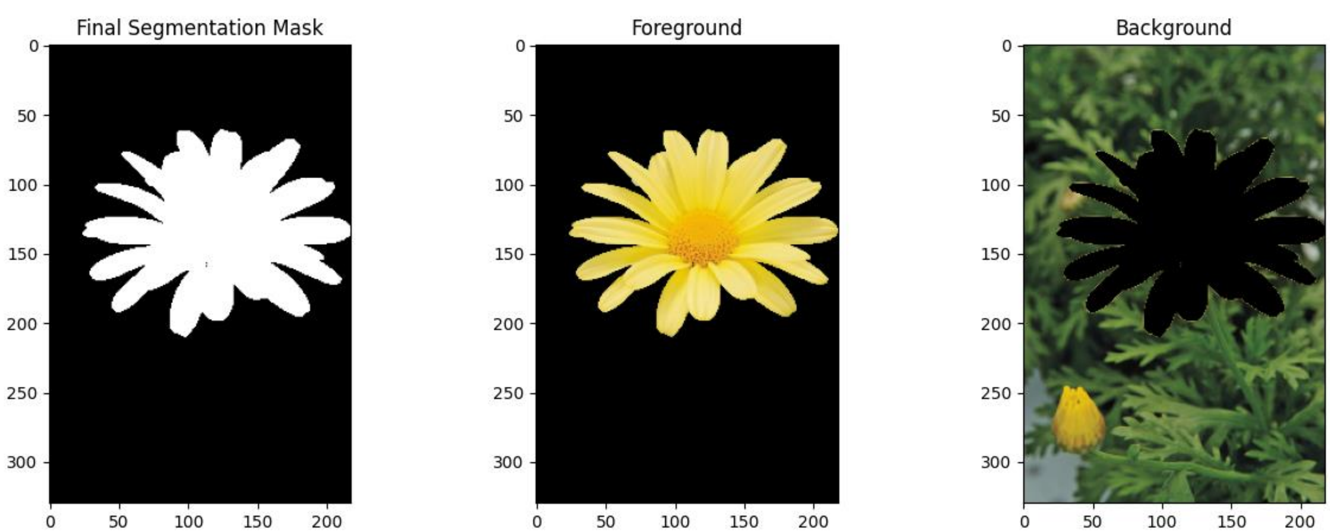


Choose above 2 pictures and scaled

```
SSD between original and scaled image using nearest neighbor:  17.171342909907853
SSD between original and scaled image using bilinear interpolation:  22.376757387099232
```

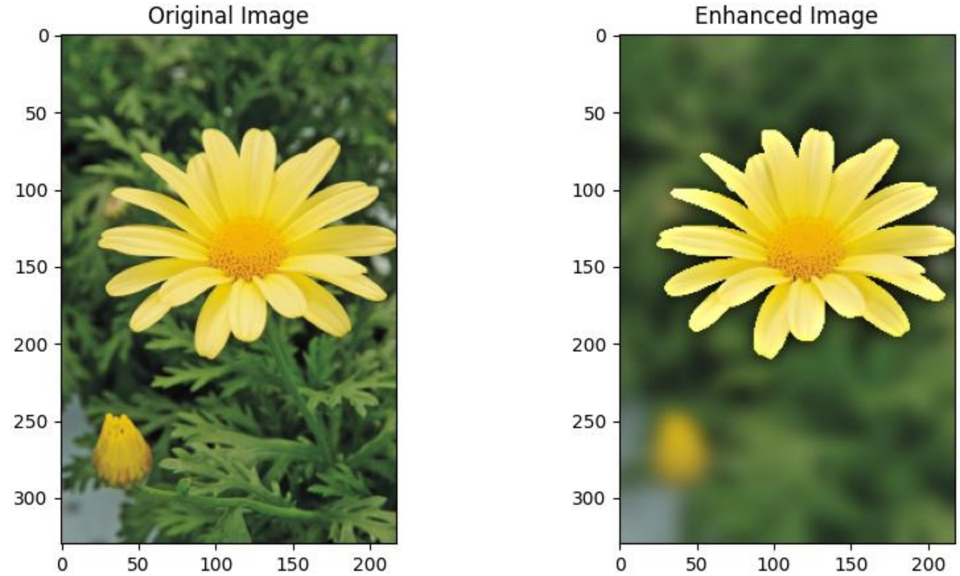
```
SSD between original and scaled image using nearest neighbor:  11.902013310185184
SSD between original and scaled image using bilinear interpolation:  16.21177662037037
```

Question 9



```python
# Define the rectangle for initial object approximation (x, y, width, height)
initial_rect = (10, 50, 420, 220)
```

```python
# Initialize the mask, background model, and foreground model
mask = np.zeros(image.shape[:2], dtype="uint8")
background_model = np.zeros((1, 65), dtype="float")
foreground_model = np.zeros((1, 65), dtype="float")
# Apply GrabCut with rectangle initialization
(cv_mask, background_model, foreground_model) = cv.grabCut(
    image, mask, initial_rect, background_model, foreground_model, 18, cv.GC_INIT_WITH_RECT
)
# Create a binary mask from the GrabCut result
segmentation_mask = np.where((cv_mask == 2) | (cv_mask == 0), 0, 1).astype('uint8')
# Run GrabCut with mask initialization
cv.grabCut(image, segmentation_mask, None, background_model, foreground_model, 18, cv.GC_INIT_WITH_MASK)
# Create foreground and background images based on the final mask
foreground_image = cv.bitwise_and(image, image, mask=segmentation_mask)
background_image = image - foreground_image
```



GrabCut segments the foreground of an image. We start with a hint, a rectangle around the region of interest. It automatically detects the foreground. Then, we apply a Gaussian filter to blur the background, merging it with the original image.

C.

GrabCut is an iterative segmentation algorithm that starts with an initial guess and refines it over multiple iterations. It may misclassify pixels near object boundaries or beyond them if the initialization or mask is not accurate. These misclassifications can lead to artifacts like darkening around the object's edges in the final segmented image. To improve results, precise initialization and manual correction of the mask may be necessary in some cases.

Git hub - [Jupyter note book]