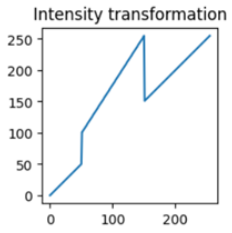
Assignment1- Intensity Transformations and Neighborhood Filtering

200148M-Dombawala.D.P.C. L GitHub - 200148M

Question-1

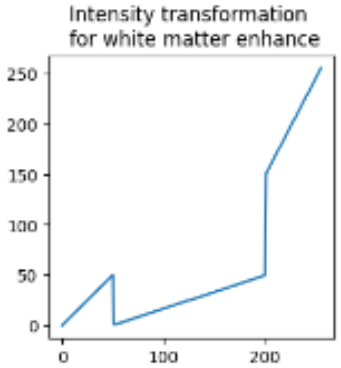
```
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150)])
t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')
t2 = np.linspace(c[0, 1] + 1 , c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1] + 1 , c[2, 1], c[2, 0] - c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1] + 1 , c[3, 1], c[3, 0] - c[2, 0]).astype('uint8')
t5 = np.linspace(c[3, 1] + 1 , 255, 255 - c[3, 0]).astype('uint8')
transform = np.concatenate((t1,t2,t3,t4,t5), axis=0).astype('uint8')
img_orig = cv.imread('emma.jpg', cv.IMREAD_GRAYSCALE)
image_transformed = cv.LUT(img_orig, transform)
```

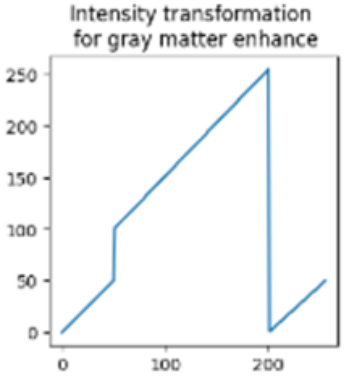


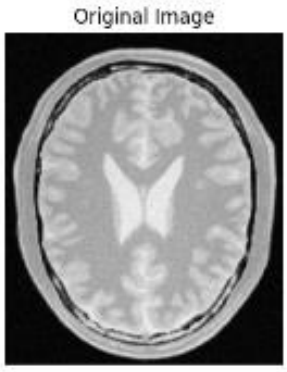
This intensity transformation maintains the original intensity values for the darkest (0 to 50) and brightest (151 to 255) pixels, leaving them unchanged. However, it shifts the intermediate intensity values to higher levels, making shadowed areas appear lighter while keeping darker tones like hair, pupils, and clothing, as well as brighter pixels like skin, unaffected.

Question -2(a), 2(b)

```
c = np.array([(50, 50), (50, 0), (200, 50), (200, 150)])
t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')
t2 = np.linspace(c[0, 1] + 1 , c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1] + 1 , c[2, 1], c[2, 0] - c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1] + 1 , c[3, 1], c[3, 0] - c[2, 0]).astype('uint8')
t5 = np.linspace(c[3, 1] + 1 , 255, 255 - c[3, 0]).astype('uint8')
s = np.array([(50, 50), (50, 100), (200, 255), (200, 0)])
s1 = np.linspace(0, s[0, 1], s[0, 0] + 1 - 0).astype('uint8')
s2 = np.linspace(s[0, 1] + 1 , s[1, 1], s[1, 0] - s[0, 0]).astype('uint8')
s3 = np.linspace(s[1, 1] + 1 , s[2, 1], s[2, 0] - s[1, 0]).astype('uint8')
s4 = np.linspace(s[2, 1] + 1 , s[3, 1], s[3, 0] - s[2, 0]).astype('uint8')
s5 = np.linspace(s[3, 1] + 1 , 50, 255 - s[3, 0]).astype('uint8')
transform1 = np.concatenate((t1, t2,t3,t4,t5), axis=0).astype('uint8')
transform2 = np.concatenate((s1, s2,s3,s4,s5), axis=0).astype('uint8')
g = cv.LUT(img_orig,transform1)
h = cv.LUT(img_orig,transform2)
```

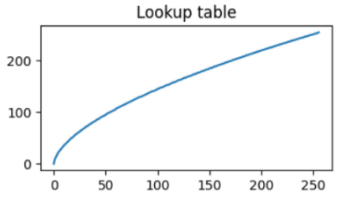


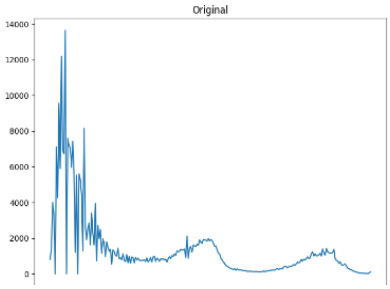


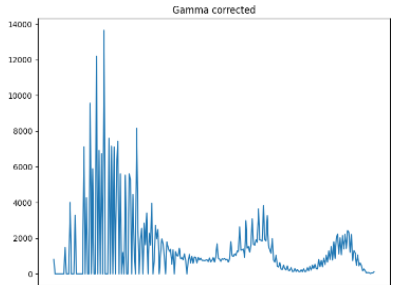


Question-3

```
lab = cv.cvtColor(f,cv.COLOR_BGR2Lab)
L, a, b = cv.split(lab)
gamma = 0.6
t = np.array([(p/255.0)**(gamma)*255.0 for p in range (0,256)]).astype('uint8')
g = cv.LUT(L, t)
lab_orig = cv.cvtColor(f,cv.COLOR_BGR2Lab)
lab[:,:,0] = g
img = cv.cvtColor(lab, cv.COLOR_Lab2RGB).astype('uint8')
hist_original = cv.calcHist([lab_orig],[0],None,[256],[0,256])
hist_gamma = cv.calcHist([lab],[0],None,[256],[0,256])
```







Gamma > 1: Darkens the image, enhancing dark details.

Gamma < 1: Brightens the image, highlighting dark areas.

Here, γ value of 0.6 expands the range of dark pixels, enhancing visibility in darker areas. This transformation is evident in the histograms, illustrating how the original image's limited dark pixel range extends in the corrected image. As a result, the original image's darker regions become more discernible, leading to improved image clarity.

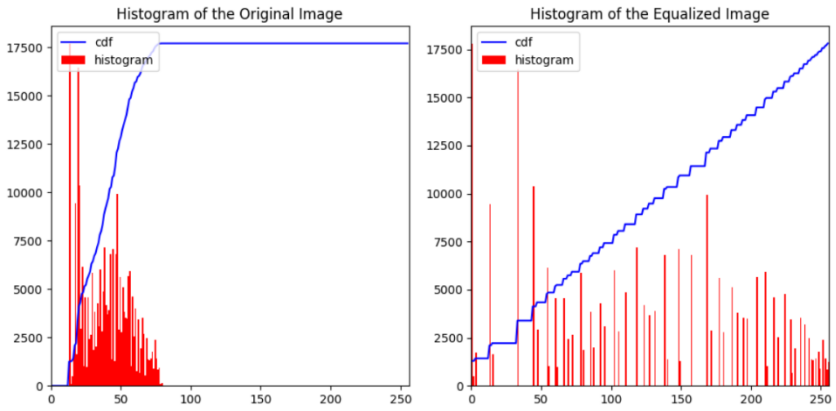
Question-4

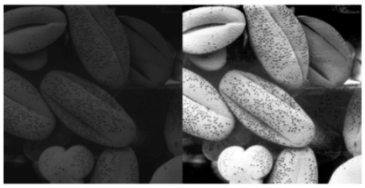
```
hue_plane, saturation_plane, value_plane = hsv_image[:,:,0], hsv_image[:,:,1],
hsv_image[:,:,2]
a, sigma = 0.7, 70
x = np.arange(256)
y = np.minimum(x + a * 128 * np.exp(-(x - 128)**2 / (2 * sigma**2)), 255)
transformed_saturation_plane = y[saturation_plane]
transformed_saturation_plane = np.clip(transformed_saturation_plane, 0, 255)
hue_plane, transformed_saturation_plane, value_plane = hue_plane.astype(np.uint8),
transformed_saturation_plane.astype(np.uint8), value_plane.astype(np.uint8)
vibrance_enhanced_hsv_image = np.stack([hue_plane, transformed_saturation_plane,
value_plane], axis=-1)
vibrance_enhanced_image = cv2.cvtColor(vibrance_enhanced_hsv_image, cv2.COLOR_HSV2BGR)
```

1. **Increase "a":** If you want to enhance vibrance more aggressively, increase the value of "a." For example, changing **a = 0.7** to **a = 1.0** would result in a stronger color enhancement effect. You'll see a more pronounced boost in color saturation.
2. **Decrease "a":** To apply a more subtle vibrance enhancement, decrease the value of "a." For instance, changing **a = 0.7** to **a = 0.5** would produce a milder effect, preserving some of the original colors while still increasing vibrance.

Question-5

```
img = cv.imread('shells.jpg', cv.IMREAD_GRAYSCALE)
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
img = cv.imread('shells.jpg', cv.IMREAD_GRAYSCALE)
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
hist, bins = np.histogram(img.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
equ = cv.equalizeHist(img)
hist, bins = np.histogram(equ.ravel(), 256, [0, 256])
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max() / cdf.max()
```

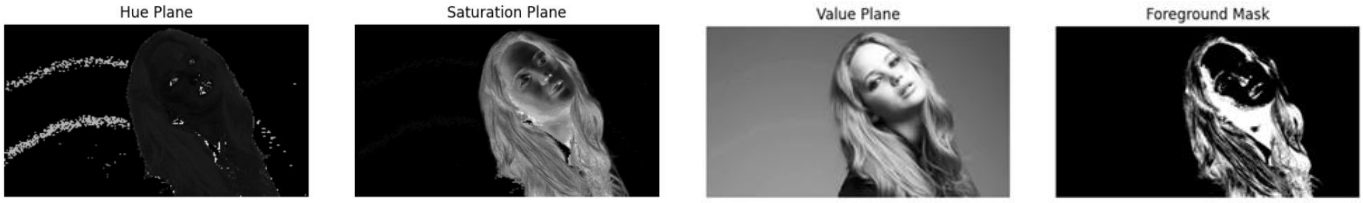




Histogram equalization is employed to enhance the contrast of a grayscale image. Initially, the code computes the histogram and Cumulative Distribution Function (CDF) of the original image's pixel intensities. Histogram equalization is then applied using OpenCV's equalizeHist function, which redistributes pixel values across the entire available range, improving contrast. The code plots both the original and equalized histograms and CDFs for visual comparison. The equalized histogram exhibits a more even distribution of pixel values. By visualizing the original and equalized images side by side, the code effectively illustrates the significant contrast enhancement and detail improvement achieved through histogram equalization.

Question-6

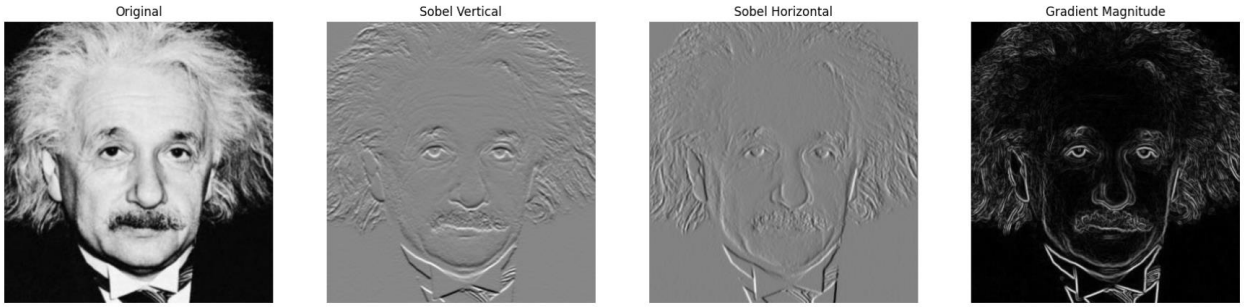
```
image = cv2.imread('jeniffer.jpg')
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
fig, axes = plt.subplots(2, 4, figsize=(16, 8))
threshold_plane = hsv[:, :, 1]
_, mask = cv2.threshold(threshold_plane, 120, 255, cv2.THRESH_BINARY)
foreground = cv2.bitwise_and(image, image, mask=mask)
hist = cv2.calcHist([foreground], [0], None, [256], [0, 256])
cumulative = np.cumsum(hist)
cumulative_normalized = cumulative / cumulative[-1]
equalized_foreground = np.interp(foreground[:, :, 0], np.arange(256),
cumulative_normalized).astype(np.uint8)
background_mask = cv2.bitwise_not(mask)
background = cv2.bitwise_and(image, image, mask=background_mask)
result_image = cv2.add(background, cv2.cvtColor(equalized_foreground,
cv2.COLOR_GRAY2BGR))
```



Here, we enhance the contrast of a person's face, "Jennifer," while keeping the background unchanged. It converts the image to the HSV color space and extracts the face by thresholding the saturation plane, which highlights the face's color. Histogram equalization is then applied to the extracted face, enhancing contrast and detail. The background is isolated using a mask, preserving its original appearance. The final image combines the enhanced face with the unaltered background. This selective contrast enhancement makes Jennifer's features more pronounced and visually appealing. The code also displays the original image, the result with the improved face, and a histogram of the face's pixel intensities for visual comparison.

Question-7(a)

```
f = cv.imread(r'einstein.png', cv.IMREAD_GRAYSCALE).astype(np.float32)
assert f is not None
sobel_v = np.array([[-1,-2,-1],[0,0,0],[1,2,1]], dtype='float')
f_x = cv.filter2D(f, -1, sobel_v)
sobel_h = np.array([[-1,0,1],[-2,0,2],[-1,0,1]], dtype='float')
f_y = cv.filter2D(f, -1, sobel_h)
grad = np.sqrt(f_x**2 + f_y**2)
```



Sobel kernels are used to detect an image's vertical and horizontal edges. Here the kernels are custom made to accentuate the vertical and horizontal lines. Then these made kernels are 2D convoluted with the original painting using the filter2D. Finally, the vertical and horizontal edges are combined in one image to make a clear image with all the edges.

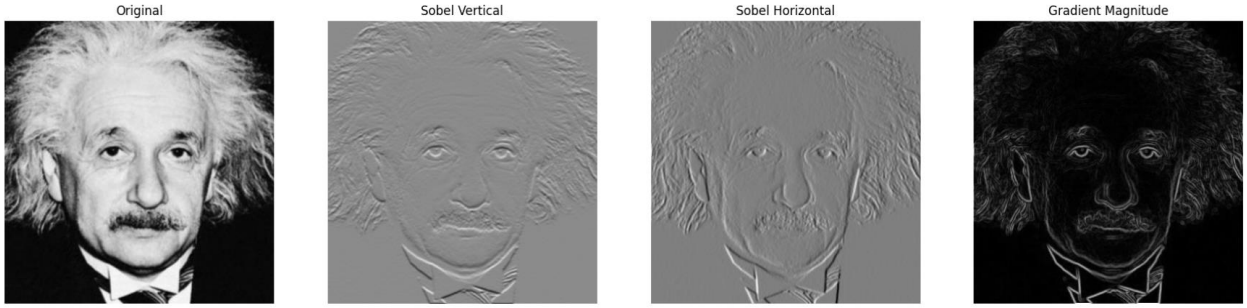
Question-7(b)

```
def convolution2d(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1
        x = x - m + 1
        new_image = np.zeros((y,x))
        for i in range(y):
            for j in range(x):
                new_image[i][j] = np.sum(image[i:i+m, j:j+m]*kernel)
    return new_image

f = cv.imread(r'einstein.png', cv.IMREAD_GRAYSCALE).astype(np.float32)
rows, cols = f.shape[0] , f.shape[1]
sobel_v = np.array([[-1,-2,-1],[0,0,0],[1,2,1]], dtype='float')
sobel_h = np.array([[-1,0,1],[-2,0,2],[-1,0,1]], dtype='float')
grad = np.sqrt(f_x**2 + f_y**2)
padded = np.zeros((rows+2,cols+2))
for i in range(rows):
    for j in range(cols):
        padded[i+1,j+1] = f[i,j]
f_x = convolution2d(padded, sobel_v)
f_y = convolution2d(padded, sobel_h)
grad = np.sqrt(f_x**2 + f_y**2)
```

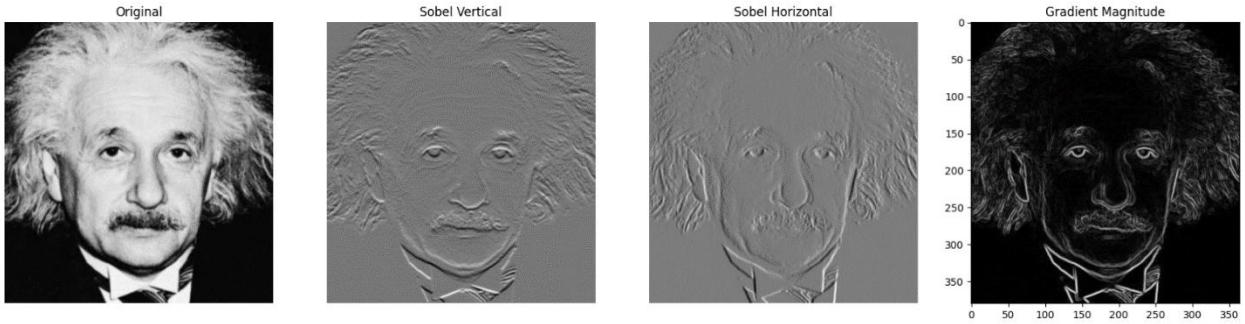
The difference in this part of the code is that instead of using built-in functions or libraries for image padding, the code manually adds a border of zeros around the image before applying the Sobel edge detection filters. This manual padding ensures that the convolution operation can be performed

consistently across all pixels in the image, including those at the edges, without encountering any issues related to boundary effects or missing data.



Question-7(c)

```
f = cv.imread(r'einstein.png', cv.IMREAD_GRAYSCALE).astype(np.float32)
assert f is not None
sobel_h1 = np.array([[1],[2],[1]], dtype='float')
sobel_h2 = np.array([[1,0,-1]],dtype='float')
sobel_v1 = np.array([[-1],[0],[1]], dtype='float')
sobel_v2 = np.array([[-1,-2,-1]],dtype='float')
f_y1 = cv.filter2D(f, -1, sobel_h1)
f_y2 = cv.filter2D(f_y1, -1, sobel_h2)
f_x1 = cv.filter2D(f, -1, sobel_v1)
f_x2 = cv.filter2D(f_x1, -1, sobel_v2)
grad = np.sqrt(f_x2**2 + f_y2**2)
```



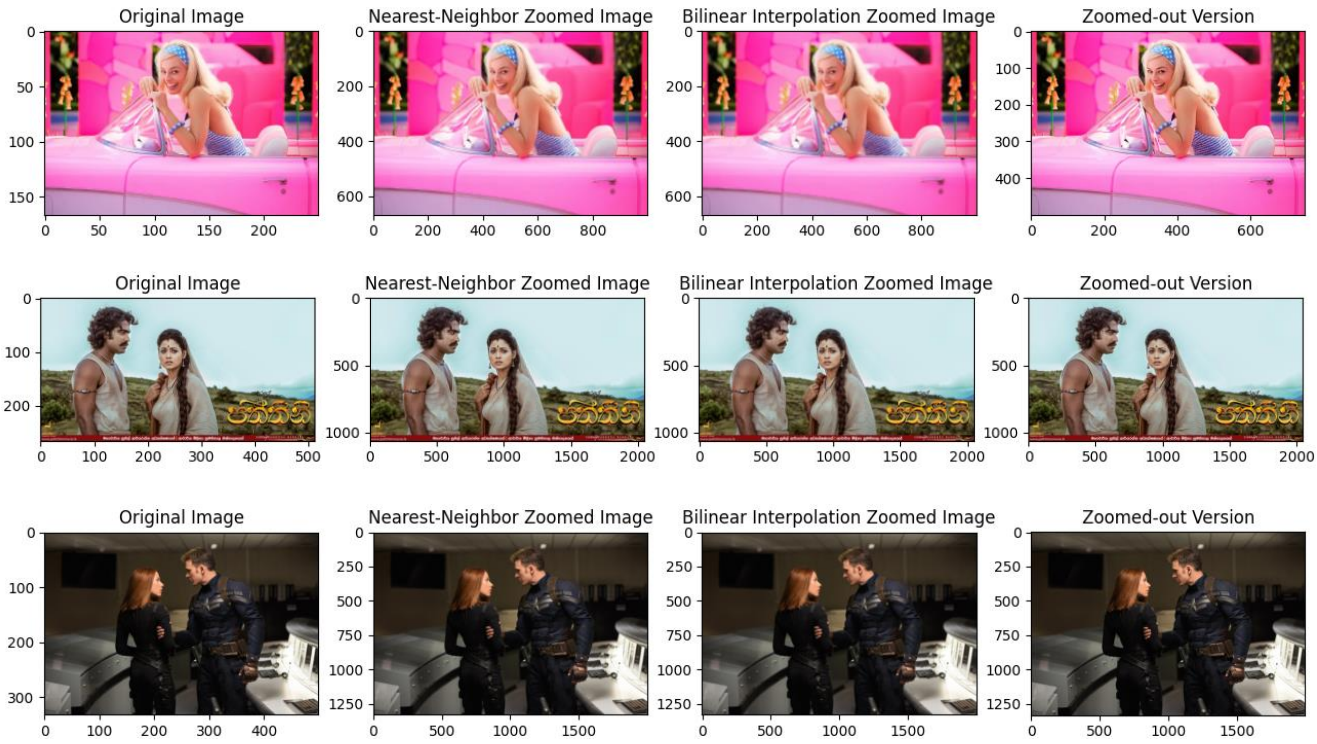
The order of performing a convolution does not affect the output since the convolution is an associative process. By carefully using this property we can reduce the computation complexity in some instances. Here you can see the output is the same as before.

Question-8

```
originals = ["im01small.png", "im02small.png", "im03small.png", "im04small.png", "im05small.png",
"im06small.png", "im07small.png", "im08small.png", "im09small.png", "im10small.png",
"im11small.png"]
zoomed_outs = ["im01.png", "im02.png", "im03.png", "im04.png", "im05.png", "im06.png",
"im07.png", "im08.png", "im09.png", "im10.png", "im11.png"]

for i in range(11):
    img = cv.imread(originals[i])
    img1 = cv.imread(zoomed_outs[i])
    img_resized = cv.resize(img, None, fx=4, fy=4, interpolation=cv.INTER_NEAREST)
    bilinear_img = cv.resize(img, img_resized.shape[1::-1], interpolation=cv.INTER_LINEAR)
    ssd_near = np.sum((img_resized - img_resized) ** 2)  # Nearest-neighbor SSD is zero
    ssd_bilinear = np.sum((img_resized - bilinear_img) ** 2)
    max_possible_ssd = np.sum((img_resized - np.zeros_like(img_resized)) ** 2)
```

```
normalized_ssd_near = ssd_near / max_possible_ssd
normalized_ssd_bilinear = ssd_bilinear / max_possible_ssd
```



1.Nearest-Neighbor Interpolation:

- Nearest-neighbor interpolation is a simple method that replicates the nearest pixel's color when scaling up an image.
- It tends to produce blocky, pixelated results with sharp edges.
- It doesn't introduce any new color values between existing pixels.

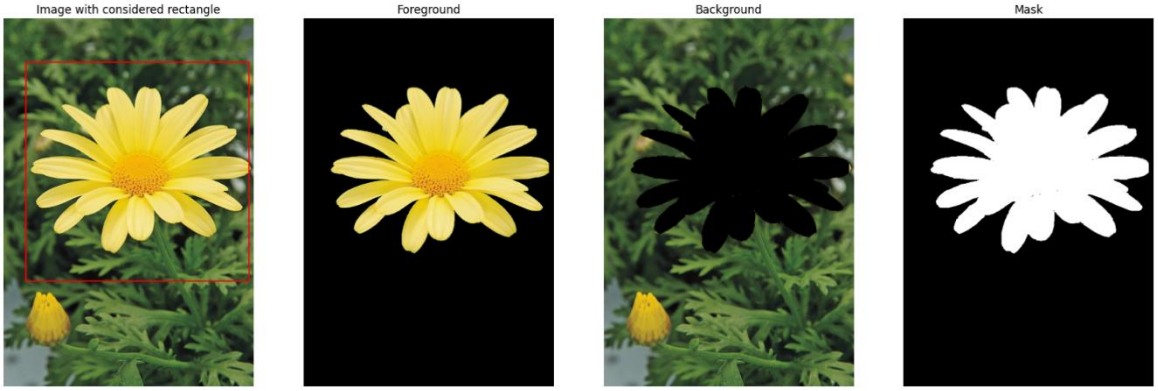
2. Bilinear Interpolation:

- Bilinear interpolation is a more advanced method that calculates the color value at a new pixel by considering the weighted average of nearby pixels.
- It produces smoother results with gradual transitions between colors.
- It can introduce new color values between existing pixels, resulting in a more visually pleasing upscale.

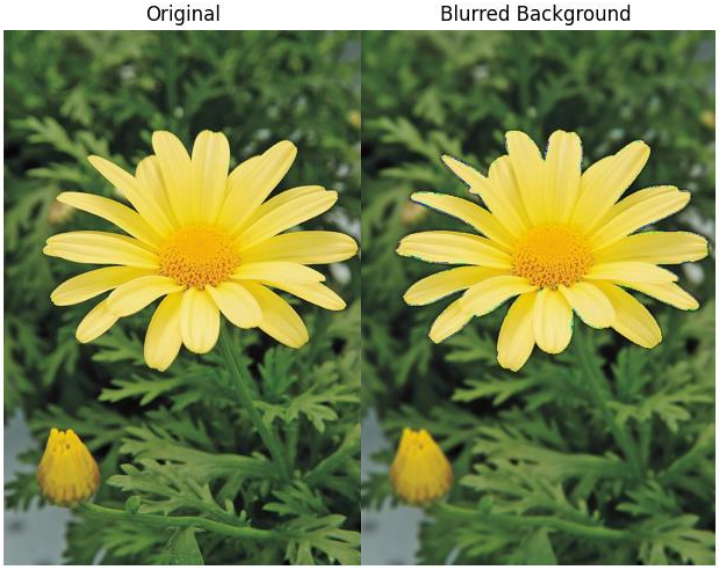
In summary, if you want a simple and quick upscale with a blocky appearance, you can use nearest-neighbor interpolation. If you want smoother, more visually appealing results with gradual color transitions, bilinear interpolation is a better choice.

Question-8(a),8(b)

```
im = cv2.imread('daisy.jpg')
orig = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
mask = np.zeros(im.shape[:2], np.uint8)
bgdModel, fgdModel = np.zeros((1, 65), np.float64), np.zeros((1, 65), np.float64)
rect = (50, 100, 500, 500)
cv2.grabCut(im, mask, rect, bgdModel, fgdModel, 3, cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
FG, BG = im * mask2[:, :, np.newaxis], im - (im * mask2[:, :, np.newaxis])
cv2.rectangle(orig, (50, 100), (550, 600), (255, 0, 0), 2)
```




```
sigma = 4
BG_gauss = cv.GaussianBlur(BG,(7,7),sigma)
blurred_im= np.add(FG,BG_gauss)
```



GrabCut is an image segmentation method that employs graph cuts to automatically delineate a specific object from its background in an image. Instead of manual selection, it relies on a user-provided pixel value range indicating the object. This information guides GrabCut in effectively isolating the object from the background, making it a valuable tool for tasks like object extraction and image manipulation, where accurate separation is required without laborious manual efforts.

Question-8(c)

In the GrabCut algorithm we're employing, it tends to detect edges in regions where there are no clear boundaries. This can lead to situations where some pixels that should be considered part of the background are mistakenly identified as part of the foreground. Consequently, when we attempt to enhance the foreground, these incorrectly labeled pixels can cause the surrounding areas, including those beyond the actual object edges, to appear darker or exhibit unintended artifacts.