EN3160 Assignment 1 on Intensity Transformations and Neighborhood Filtering

Sahan Abeyrathna
210005H
Github link: https://github.com/Sahanmin/Intensity-Transformations-and-Neighborhood-Filtering

Question 1

```
# Define the range for input and output intensities
input_range_start, input_range_end = 50, 150
output_range_start, output_range_end = 100, 255

# Create the transformation array
transformation = np.arange(256, dtype=np.uint8)
transformation[input_range_start: input_range_end+1] =
np.linspace(output_range_start, output_range_end, input_range_end- input_range_start + 1, dtype=np.uint8)

# Plot the transformation curve
plt.figure(figsize=(5, 5))
plt.plot(transformation)
plt.title("Applied Transform")
plt.xlabel("Input Intensity")
plt.ylabel("Output Intensity")
```

Figure:Transform generation

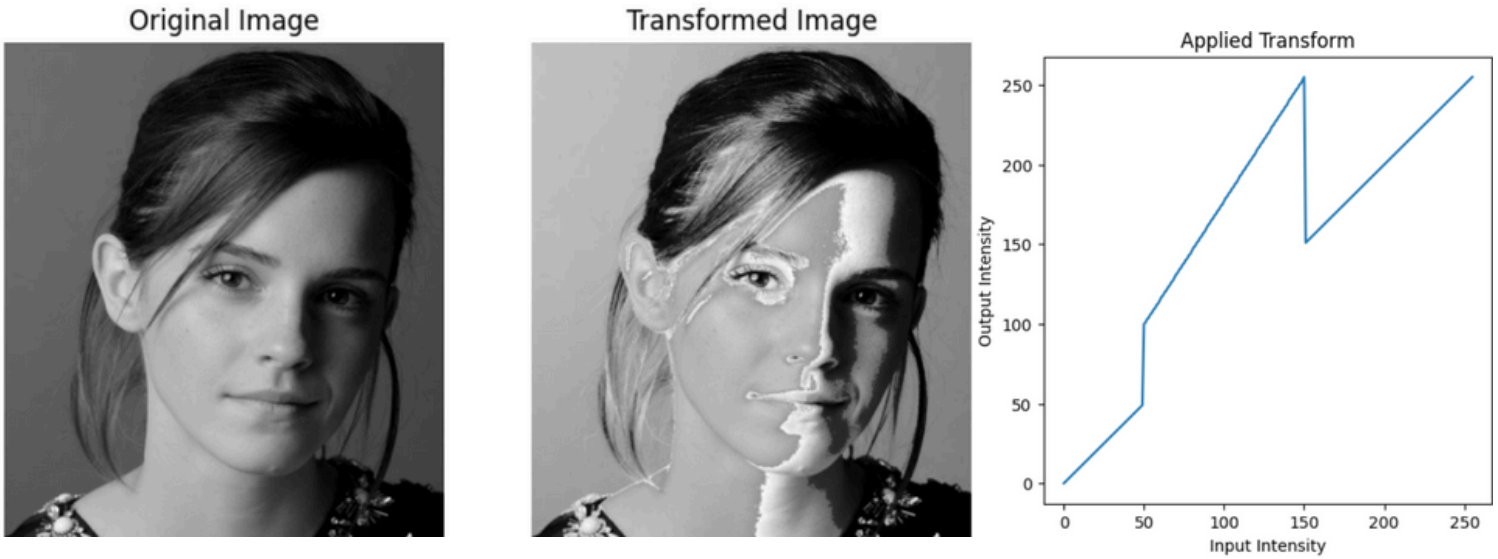


Figure:Transformation

- The hair color appears unchanged because pixel values in the [0,50] range are mapped similarly, while mid range values (50-150) are exaggerated, causing white patches on the right side of the face. The left, already lighter, shows little change.

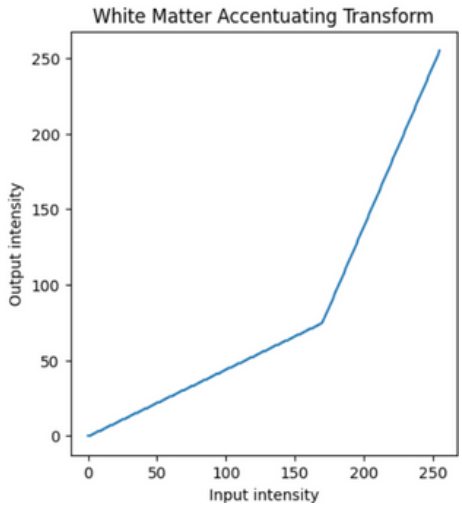
Question 2

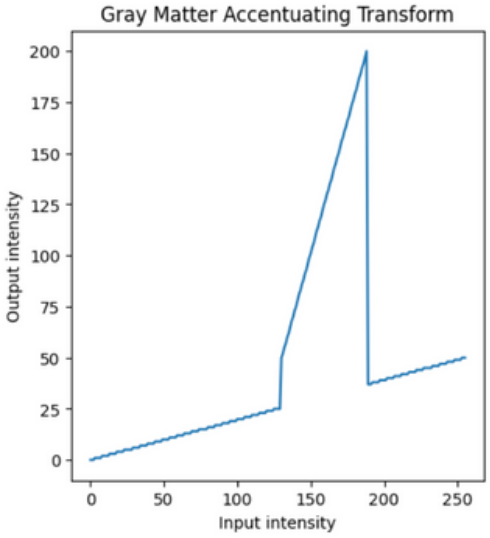
```
# White matter transformation
midpoint_x = 170
midpoint_y = 75

white_transform = np.arange(256, dtype=np.uint8)
white_transform[:midpoint_x + 1] = np.linspace(0, midpoint_y, midpoint_x + 1, dtype=np.uint8)
white_transform[midpoint_x:] = np.linspace(midpoint_y, 255, 256 - midpoint_x, dtype=np.uint8)
```

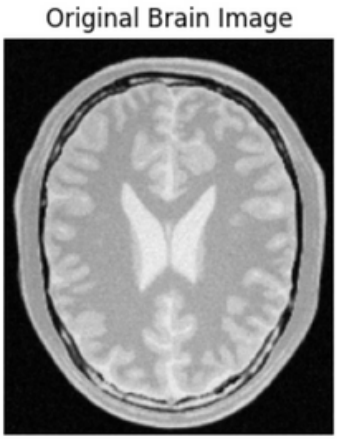
Figure:White mattarTransformation

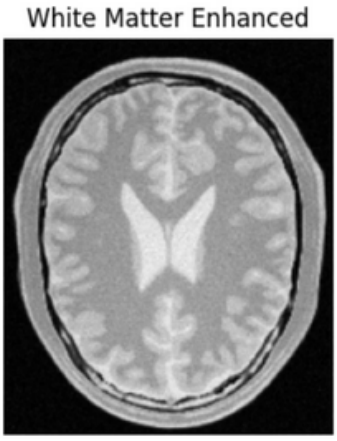
```
# Gray matter transformation
low_x, high_x = 130, 188
low_y, high_y = 50, 200

grey_transform = np.linspace(0, low_y, 256)
grey_transform = np.round(grey_transform).astype(np.uint8)
grey_transform[low_x:high_x + 1] = np.linspace(low_y, high_y, high_x + 1 - low_x, dtype=np.uint8)
```

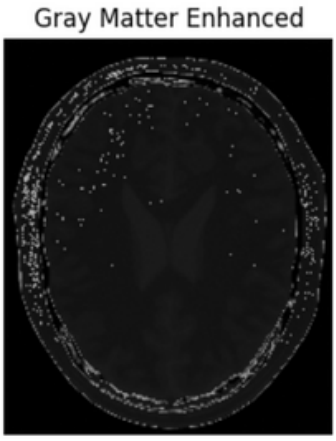
Figure:Grey mattar Transformation











- For the white matter transformation, intensities below the midpoint (170) are mapped between 0 and 75, and those above 170 are stretched to 255. As a result, the white matter regions in the image appear brighter and more pronounced, making the white structures stand out clearly.In the gray matter transformation, intensities between 130 and 188 are mapped from 50 to 200, enhancing contrast in the gray matter. This causes the gray regions to become more distinct and noticeable in the image. The output images reflect these changes: the white matter enhanced image shows bright, well defined white matter structures, while the gray matter enhanced image highlights darker regions with increased contrast around the gray matter areas.

Question 3

```
gamma = 0.7
gamma_transform = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')

img3 = cv.imread( "/content/highlights_and_shadows.jpg", cv.IMREAD_COLOR)
img3_lab = cv.cvtColor(img3, cv.COLOR_BGR2LAB)  # Convert to LAB color space
# In the LAB colour space, the L plane encodes brightness only
img3_lab[:, :, 0] = gamma_transform[img3_lab[:, :, 0]]  # Apply transform only to L plane
```

Figure: Gamma correlation

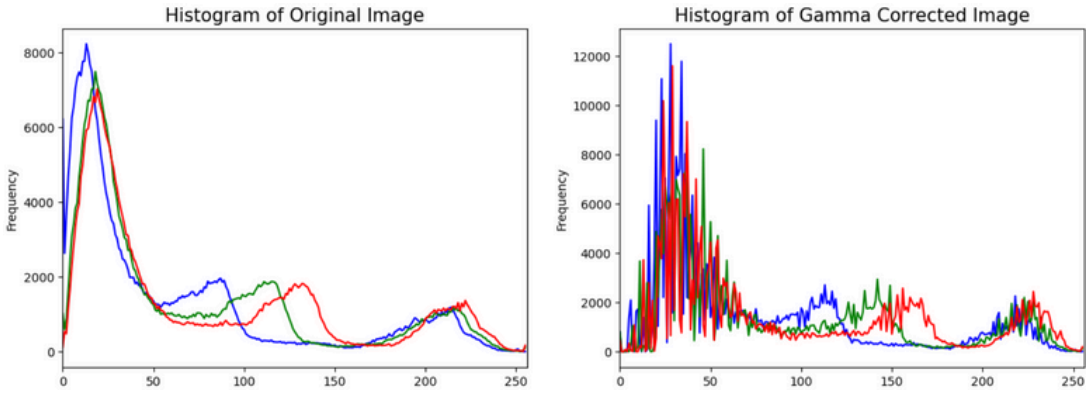


Figure: Histograms

Gamma Value = 1: The image remains unchanged with a linear gamma curve, preserving the original histogram shape.
Gamma Value < 1: The image darkens with reduced contrast, shifting the histogram leftward toward darker values.
Gamma Value > 1: The image brightens and contrast increases, shifting the histogram rightward toward lighter values.

Question 4

a)

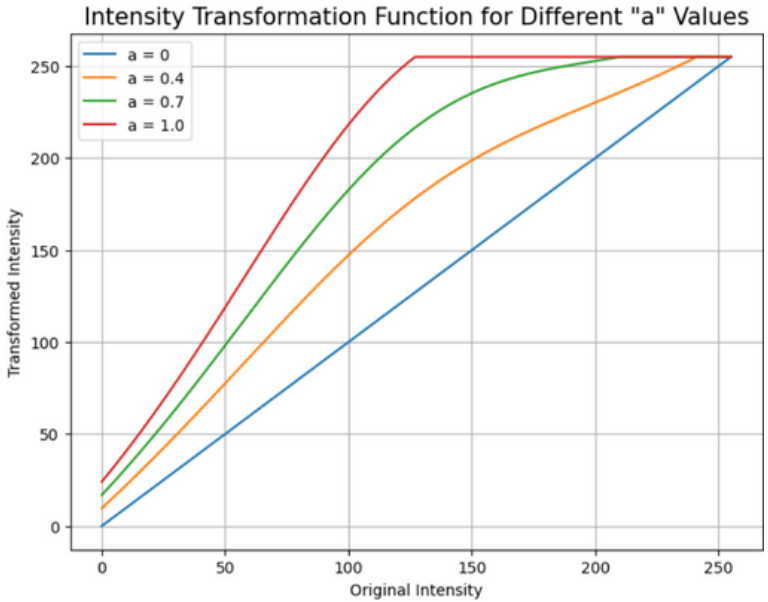
```
# Load the image in BGR color space
image_bgr = cv.imread('/content/spider.png', cv.IMREAD_COLOR)

# Convert the BGR image to HSV color space
image_hsv = cv.cvtColor(image_bgr, cv.COLOR_BGR2HSV)

# Extract the saturation (S) plane
h, saturation_plane, v = cv.split(image_hsv)
```

b)

```
# Define the intensity transformation function
def intensity_transform(x, a, sigma=70):
    f_x = np.clip(x+a*128*np.exp(-(x-128)**2/(2*sigma**2)), 0, 255)
    return f_x

# Apply the intensity transform function
transforemed_saturation_plane = intensity_transform(saturation_plane, 0.4
```

c)

```
image_copy = image_hsv.copy()
# recombine the planes
image_copy[:, :, 1] = transforemed_saturation_plane

# Convert the HSV image back to BGR color space
original_image = cv.cvtColor(image_hsv, cv.COLOR_HSV2RGB)
transformed_image = cv.cvtColor(image_copy, cv.COLOR_HSV2RGB)
```

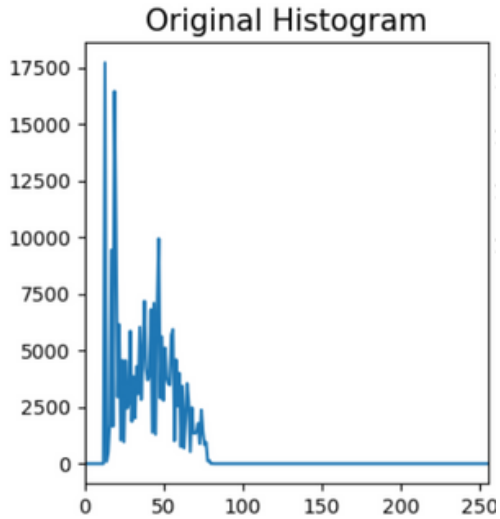


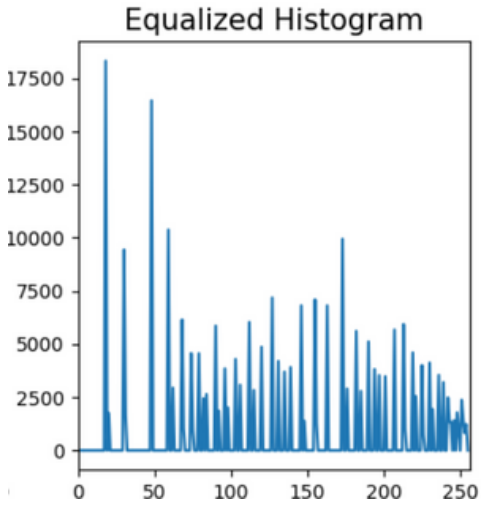
- This adjusts the saturation plane, enhancing color vibrancy and contrast, especially in low saturation areas. While it boosts saturation, it doesn't clearly highlight image details. Fine tuning the 'a' parameter is needed for the best results.

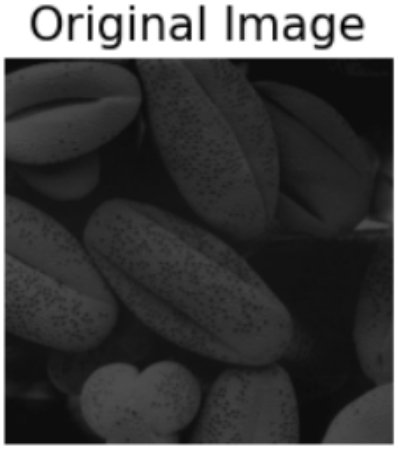
Question 5

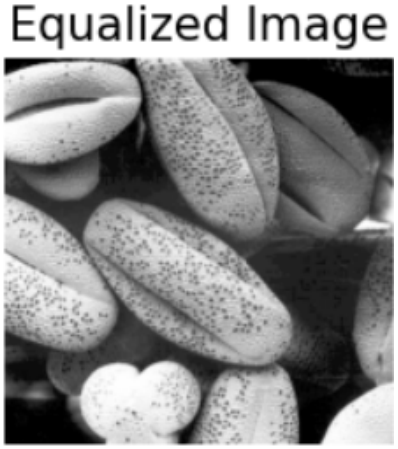
```python
def histogram_equalization(im):
    img = cv.imread(im,cv.IMREAD_GRAYSCALE)

    # Calculate the histogram of the image
    histogram = np.zeros(256, dtype=int)
    for pixel_value in img.flat:
        histogram[pixel_value] += 1

    # Calculate the cumulative distribution function (CDF)
    cdf = np.zeros(256, dtype=int)
    cdf[0] = histogram[0]
    for i in range(1, 256):
        cdf[i] = cdf[i - 1] + histogram[i]

    # Perform histogram equalization
    num_pixels = img.size
    equalized_image = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            pixel_value = img[i, j]
            equalized_pixel = int((cdf[pixel_value] / num_pixels) * 255)
            equalized_image[i, j] = equalized_pixel

    # Calculate the histogram of the equalized image
    histogram_equalized = np.zeros(256, dtype=int)
    for pixel_value in equalized_image.flat:
        histogram_equalized[pixel_value] += 1
```

- The code computes histograms and performs equalization by mapping original values across the full range, creating a more balanced intensity distribution. As a result, the darker image becomes brighter with a wider dynamic range.





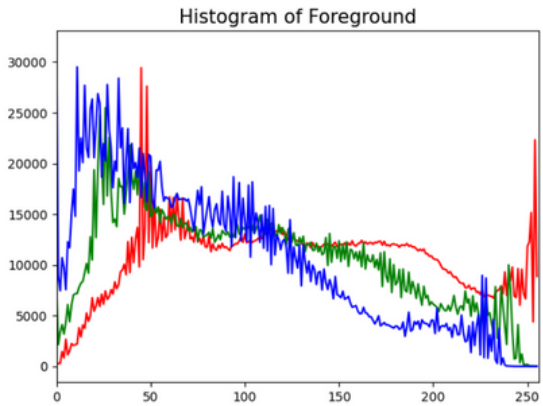


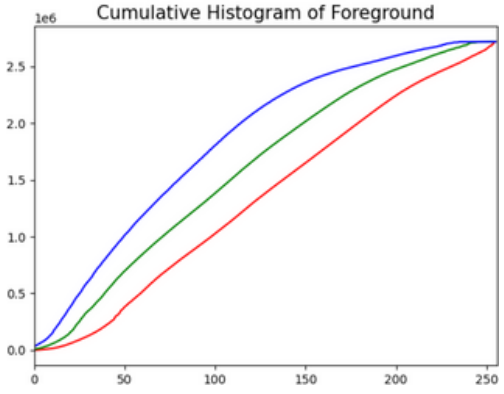


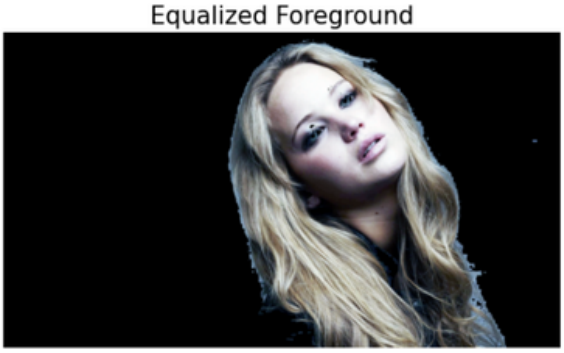
Question 6

a)


```
# Create a copy of the foreground image for equalization
equalized_foreground = foreground_rgb.copy()
channels = ('r', 'g', 'b')
foreground_pixels = mask.sum() // 255

# Loop through each color channel and calculate histograms
for i, channel in enumerate(channels):
    # Compute the histogram considering only the foreground using the mask
    hist = cv.calcHist([foreground_rgb], [i], mask, [256], [0, 256])
    ax[0].plot(hist, color=channel)
    ax[0].set_xlim([0, 256])

    # Compute the cumulative sum of the histogram
    cumulative_hist = np.cumsum(hist)
    ax[1].plot(cumulative_hist, color=channel)
    ax[1].set_xlim([0, 256])

    # Apply histogram equalization by transforming intensity values
    transformation = cumulative_hist * 255 / cumulative_hist[-1]
    equalized_foreground[:, :, i] = transformation[foreground_rgb[:, :, i]]

# Reapply the mask after equalization to remove the background
equalized_foreground = np.bitwise_and(equalized_foreground, mask_3d)

# Set plot titles
ax[0].set_title("Histogram of Foreground")
ax[1].set_title("Cumulative Histogram of Foreground")
```





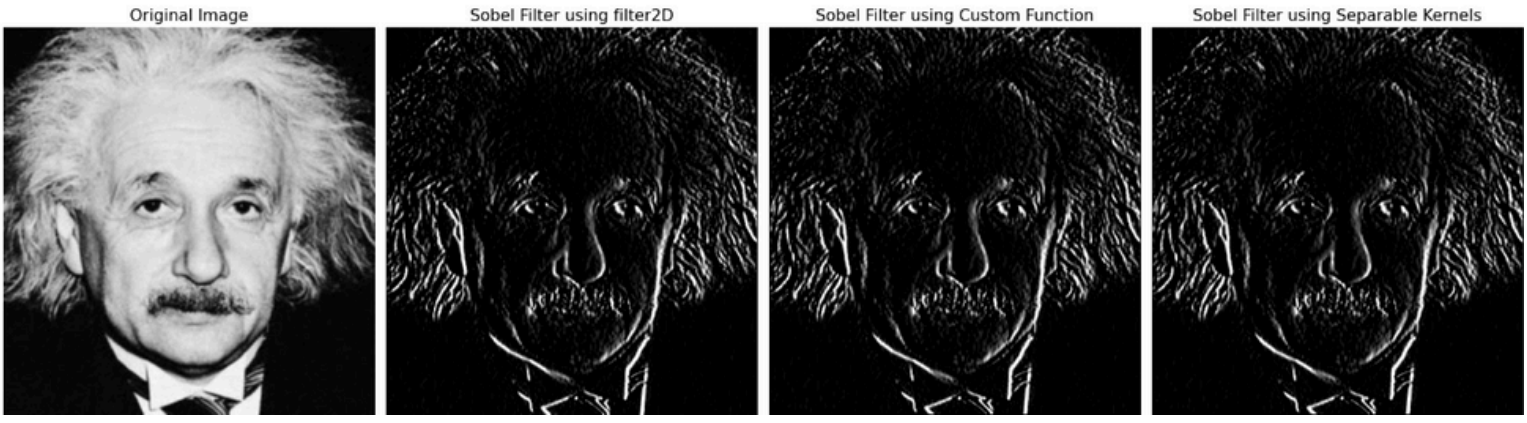


- The foreground object is more saturated than the background, so the saturation plane is used for thresholding. Lighter pixels are selected with "cv.inRange", but darker foreground details like eyes are missed.

Question 7

```python
# Custom function to apply a filter to an image
def apply_filter(image, kernel):
    # Ensure the kernel dimensions are odd
    assert kernel.shape[0] % 2 == 1 and kernel.shape[1] % 2 == 1

    # Calculate kernel half-heights and half-widths
    k_hh, k_hw = kernel.shape[0] // 2, kernel.shape[1] // 2
    h, w = image.shape

    # Normalize the image to the range [0, 1]
    image_float = cv.normalize(image.astype('float'), None, 0, 1, cv.NORM_MINMAX)

    # Initialize an empty result array
    result = np.zeros(image.shape, dtype='float')

    # Apply the filter by looping over the image
    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())

    # Rescale the result to [0, 255]
    result = result * 255
    result = np.clip(result, 0, 255).astype(np.uint8)  # Ensure values are within [0, 255]
    return result
```

```python
# Define a function to filter a normalized image without rounding
def apply_filter_step(image, kernel):
    # Ensure the kernel dimensions are odd
    assert kernel.shape[0] % 2 == 1 and kernel.shape[1] % 2 == 1

    # Calculate kernel half-heights and half-widths
    k_hh, k_hw = kernel.shape[0] // 2, kernel.shape[1] // 2
    h, w = image.shape

    # Initialize an empty result array
    result = np.zeros(image.shape, dtype='float')

    # Apply the filter in a single step
    for m in range(k_hh, h - k_hh):
        for n in range(k_hw, w - k_hw):
            result[m, n] = np.dot(image[m - k_hh: m + k_hh + 1, n - k_hw: n + k_hw + 1].flatten(), kernel.flatten())

    return result
```



- The final images show slight differences in edge detection based on how the Sobel filter was applied. The 'filter2D' method produces the cleanest and sharpest edge detection due to OpenCV's optimized convolution process. The 'custom function' also detects edges well, but the edges may appear slightly less sharp because the manual convolution process might introduce small differences in precision. In the 'separable convolution method', edges are also detected well, but the result may look a bit smoother since the convolution is split into two steps, slightly affecting sharpness.

```python
# Function to interpolate pixel values for a color image
def interpolate(image, indices, method):
    if method == 'nn':  # Nearest neighbor interpolation
        indices[0] = np.minimum(np.round(indices[0]), image.shape[0] - 1)
        indices[1] = np.minimum(np.round(indices[1]), image.shape[1] - 1)
        indices = indices.astype(np.uint64)
        return image[indices[0], indices[1]]

    elif method == 'bi':  # Bilinear interpolation
        floors = np.floor(indices).astype(np.uint64)
        ceils = floors + 1

        # Ensure ceiling indices don't go out of bounds
        ceils_limited = [np.minimum(ceils[0], image.shape[0] - 1), np.minimum(ceils[1], image.shape[1] - 1)]

        # Extract the four corner pixels
        p1 = image[floors[0], floors[1]]
        p2 = image[floors[0], ceils_limited[1]]
        p3 = image[ceils_limited[0], floors[1]]
        p4 = image[ceils_limited[0], ceils_limited[1]]

        # Repeat indices for each color channel
        indices = np.repeat(indices[:, :, :, None], 3, axis=3)
        ceils = np.repeat(ceils[:, :, :, None], 3, axis=3)
        floors = np.repeat(floors[:, :, :, None], 3, axis=3)

        # Interpolate horizontally between pixels
        m1 = p1 * (ceils[1] - indices[1]) + p2 * (indices[1] - floors[1])
        m2 = p3 * (ceils[1] - indices[1]) + p4 * (indices[1] - floors[1])
```

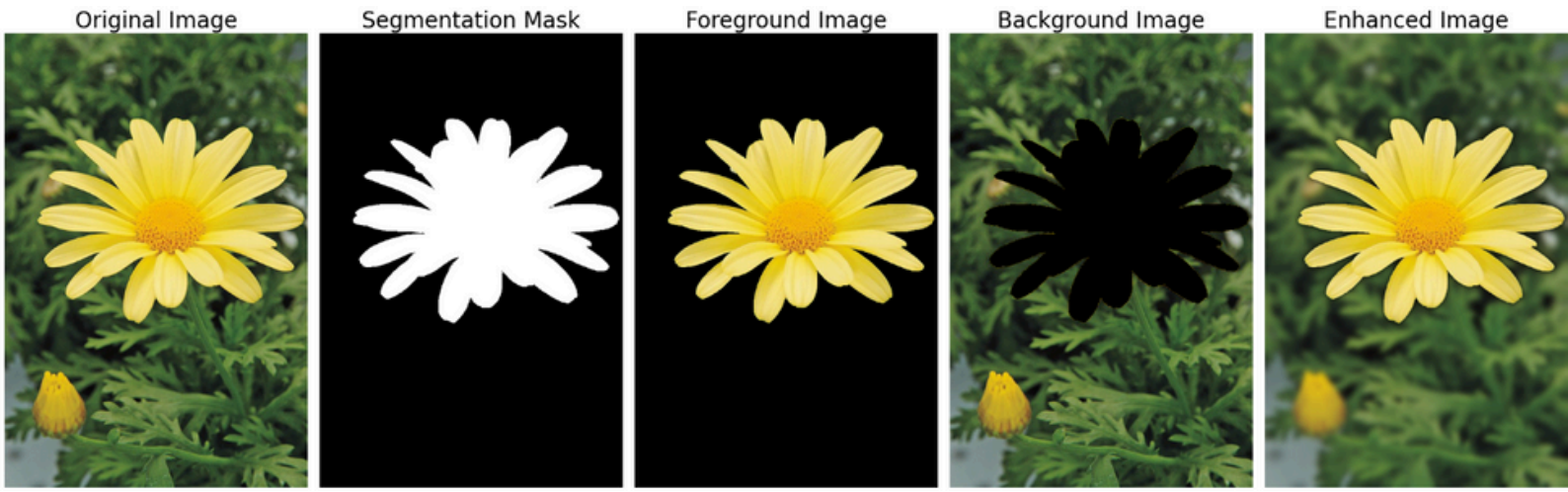
```python
# Function to zoom an image by a scaling factor
def zoom(image, factor, interpolation='nn'):
    h, w, _ = image.shape
    zoom_h, zoom_w = round(h * factor), round(w * factor)

    # Initialize an empty zoomed image
    zoomed_image = np.zeros((zoom_h, zoom_w, 3)).astype(np.uint8)

    # Compute the corresponding indices in the original image
    zoomed_indices = np.indices((zoom_h, zoom_w)) / factor

    # Apply the interpolation method to fill the zoomed image
    zoomed_image = interpolate(image, zoomed_indices, interpolation)

    return zoomed_image

# Function to compute the normalized sum of squared differences (SSD) between two images
def normalized_ssd(image1, image2):
    ssd = np.sum((image1 - image2) ** 2)
    return ssd / (image1.size * 255 * 255)
```

- Bilinear interpolation delivers better results than the nearest neighbor method, which often creates blocky artifacts due to its lack of smooth transitions between pixels. Bilinear interpolation is smoother but can still cause some blurring since it assumes a linear transition. Before calculating the normalized SSD, error handling ensures the generated image and original are the same size, resizing the generated image if necessary using `cv.resize`.

Question 9

```
# Create an initial mask
mask = np.zeros(img9.shape[:2], np.uint8)
mask[150:550, 50:600] = cv.GC_PR_FGD  # Mark the inner region as probable foreground
mask[300:410, 220:380] = cv.GC_FGD    # Mark the flower center as definite foreground to prevent holes

# Initialize background and foreground models
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Define the rectangle for grabCut
rect = (50, 150, 612 - 50, 600 - 150)
iters = 5  # Number of iterations for grabCut

# Apply grabCut using the initial mask
cv.grabCut(img9, mask, rect, bgdModel, fgdModel, iters, cv.GC_INIT_WITH_MASK)

# Set pixels that are background or probable background to 0, others to 1
foreground_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
background_mask = 1 - foreground_mask  # Background mask as the inverse

# Extract the foreground and background from the image
foreground_img9 = img9 * foreground_mask[:, :, np.newaxis]
background_img9 = img9 * background_mask[:, :, np.newaxis]
```

```
# Image with blurred background
kernel = 51
sigma = 5
blurred_background = cv.GaussianBlur(background_img9, (kernel, kernel), sigma)
blurred_background = blurred_background * background_mask[:, :, np.newaxis]

final_img9 = blurred_background + foreground_img9
```



- The dark pixels around the flower's edge in the enhanced image likely result from the GrabCut segmentation. Background pixels near the flower may have been inaccurately classified, and blurring extends into these areas, blending them with the true background. This makes the edge appear darker because misclassified pixels get mixed with the background during the blurring process.