

Name: M.M.H.H.B Gallella

Index No.: 210174X

Submission Date: 1st October 2024

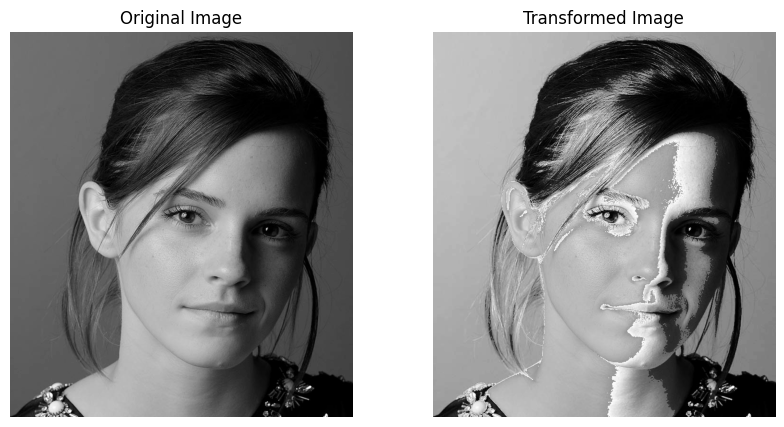
GitHub: https://github.com/HasithaGallella

Assignment Codes on: [Google Colab](https://colab.research.google.com/drive/1ZRDT6V-cFY3Rg0EKDHYxZ1GvyFXls8Az?usp=sharing)

**EN3160 – Image Processing and Machine Vision**

Assignment 1 - Intensity Transformations and Neighborhood Filtering

**Question 1 – Implementing an intensity transformation on Emma Watson**



# Read the image

image = cv2.imread('./a1images/emma.jpg', cv2.IMREAD\_GRAYSCALE) # PC

# Define the intensity transformation function

def intensity\_transformation(input\_intensity):

    # Assuming a piecewise linear function for intensity transformation

    if input\_intensity < 50:

        return input\_intensity

    elif input\_intensity < 150:

        return (1.55\*input\_intensity + 22.5)

    else:

        return input\_intensity

# Vectorize the function to apply to the image

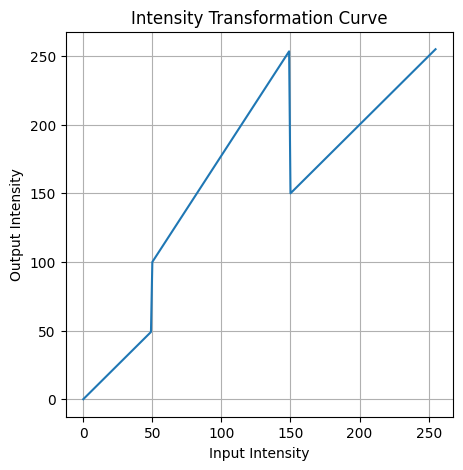
vectorized\_transformation = np.vectorize(intensity\_transformation)

# Apply the transformation to the image

transformed\_image = vectorized\_transformation(image).astype(np.uint8)

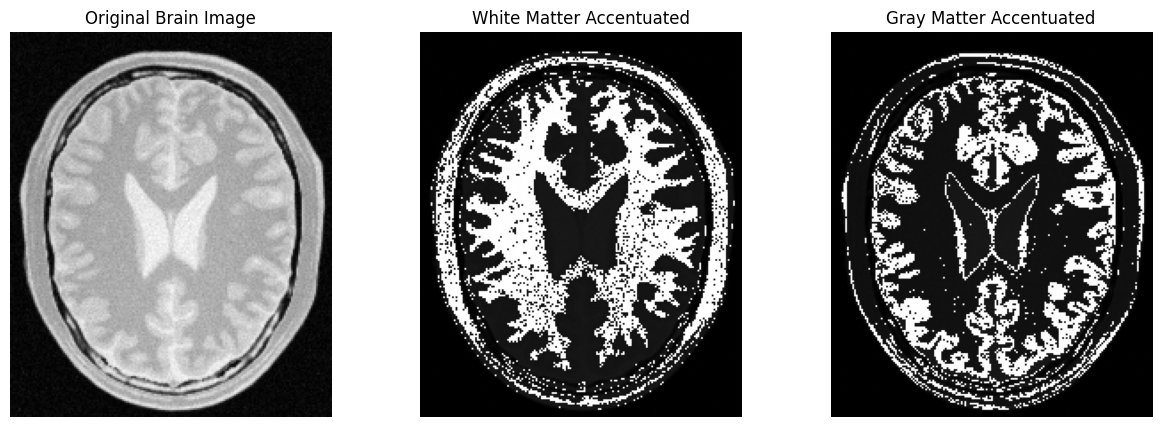
# Plot the original and transformed images

plt….



**Discussion:** Pixel values in the [0,50] and [150,255] ranges are mapped similarly, resulting in little change in her hair color, which falls in the lower range. Mid-range values (50-150) are exaggerated, causing white patches on the right side of the face, while the left side, being in the 150+ range, shows minimal change.

**Question 2 – Intensity transformation on brain proton density image**



A graph with a line graph

Description automatically generated with medium confidence

# Define intensity transformation functions for white matter and gray matter

def accentuate\_white\_matter(intensity):

    # Custom function to enhance white matter

    if intensity < 150:

        return intensity\*0.1

    elif intensity >= 150 and intensity <= 175:

        return (255 - 25 + intensity\*0.1)

    else:

        return intensity\*0.1

def accentuate\_gray\_matter(intensity):

    # Custom function to enhance gray matter

    if intensity < 150:

        return intensity\*0.1

    elif intensity >= 185 and intensity <= 210:

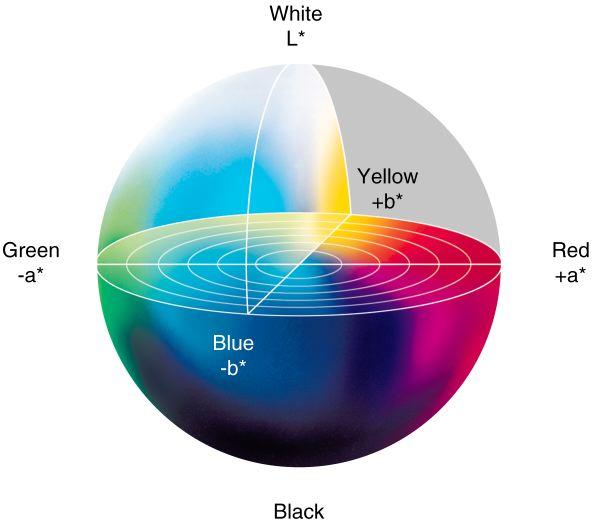
        return (255 - 25 + intensity\*0.1)

    else:

        return intensity\*0.1

**Discussion:** The Intensities measured for white matter and gray matter using my special function **get\_pixel\_intensity(event, x, y, flags, param).** Then next 2 functions accentuate white matter; range 150 to 175 and accentuate gray matter; range 185 to 210 intensities will be exaggerated as we did in the Question 1. This will allow us to view clearly the regions of white and gray matter in the brain.

**Question 3 – Gamma correction to the L plane in the L∗a∗b∗ color space**



# Load the image and convert BGR to LAB color space

img3 = cv.imread("images/highlights\_and\_shadows.jpg", cv.IMREAD\_COLOR)

img3\_lab = cv.cvtColor(img3, cv.COLOR\_BGR2LAB)

# Set the gamma value for correction

gamma = 0.6

# Create a gamma transformation lookup table

gamma\_transform = np.array([(i / 255.0) \*\* gamma \* 255.0 for i in np.arange(256)]).astype('uint8')

# Apply gamma correction to the L plane (brightness channel)

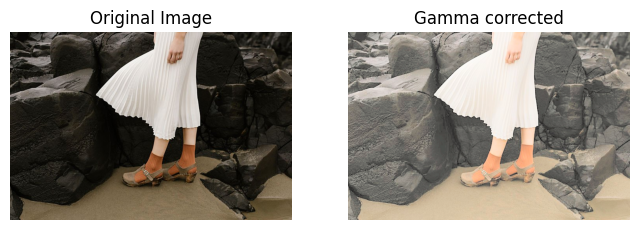
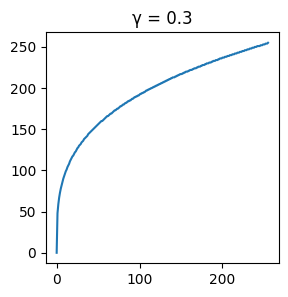
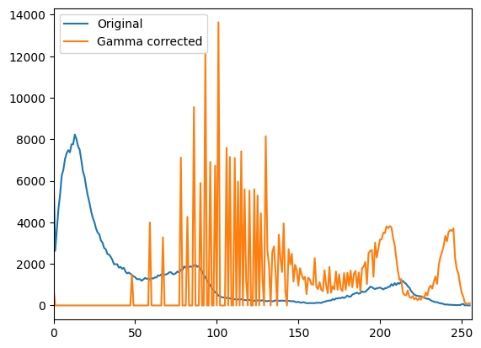
img3\_lab[:, :, 0] = gamma\_transform[img3\_lab[:, :, 0]]

# Convert back to BGR for display

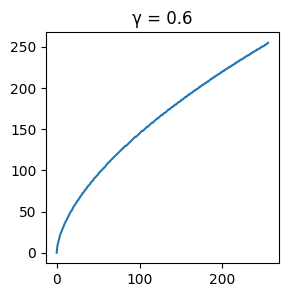
gamma\_corrected\_img = cv.cvtColor(img3\_lab, cv.COLOR\_LAB2BGR)

# Display the original and gamma corrected images

plt…

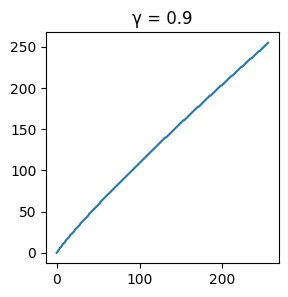


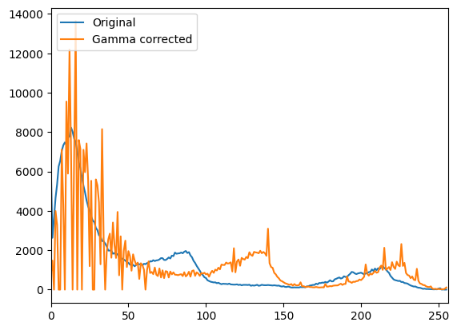
Gamma correction with different values; **γ =** **0.3,0.6,0.9**

****

**A graph of a number of blue and orange lines

Description automatically generated**

****

**A collage of rocks

Description automatically generated**

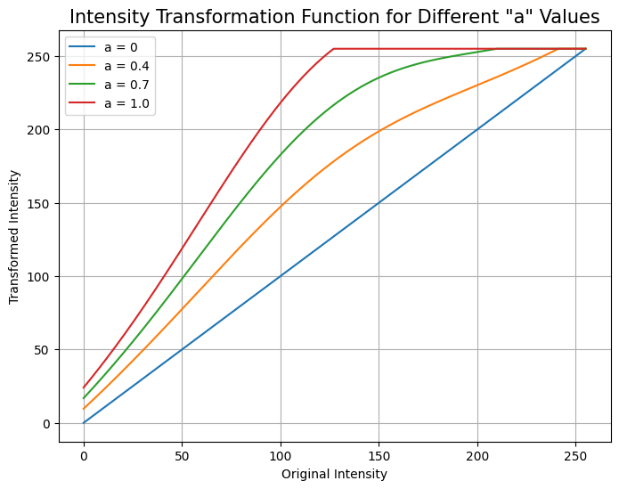
**Discussion:** Gamma correction affects the brightness and contrast of an image depending on the value of gamma. When the gamma value is set to 1, the gamma correction curve is linear, meaning the image remains unchanged and the histogram retains its original distribution. When the gamma value is less than 1 (for example, 0.2), gamma compression occurs, which darkens the image and shifts the histogram towards the left, concentrating more pixel values in the darker regions. Conversely, when the gamma value is greater than 1 (for example, 2.0), gamma expansion occurs, brightening the image and shifting the histogram to the right, thereby increasing the concentration of pixel values in the lighter regions. After testing various gamma values, it was found that a gamma value of 0.6 provided the best result for this image, enhancing the contrast while maintaining a natural look.

**For gamma = 0.6 :** Histograms for all 3 color planes before and after applying the correction

**A close-up of a graph

Description automatically generated**

**Question 4 – Intensity transformation to the saturation plane for increase the vibrance**

****

# Transformation function for vibrance enhancement

a = 0.3 # Adjusted value of 'a' for a natural vibrance

sigma = 70

def f(x):

return np.minimum(255, x + (a \* 128) \* np.exp(-(x - 128)\*\*2 / (2 \* sigma\*\*2)))

# Load the image and convert to HSV color space

img4 = cv.imread("images/spider.png", cv.IMREAD\_COLOR)

img4\_hsv = cv.cvtColor(img4, cv.COLOR\_BGR2HSV)

# (a) Apply transformation only to the saturation plane

img4\_hsv[:, :, 1] = f(img4\_hsv[:, :, 1])

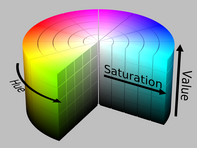
# (d) Recombine the three planes

vibrance\_enhanced\_img = cv.cvtColor(img4\_hsv, cv.COLOR\_HSV2BGR)

# (e) Display the original and enhanced images side by side

plt.figure….

****



**Discussion:**The saturation plane is modified by applying an intensity transformation, increasing the vibrancy of the image. Areas with low saturation become more vivid, enhancing color contrast. Fine-tuning the parameter a is essential; when a approaches 1, the image can appear overly vibrant. A **value of a = 0.4** provided a natural vibrance level, keeping the image visually pleasing without oversaturation.

**Question 5 – Histogram equalization**

**A close-up of a graph

Description automatically generated**

**Discussion:**

This function computes the histogram and performs histogram equalization by mapping original pixel intensities to the full dynamic range using the cumulative distribution function (CDF). As a result, the darker regions in the image are brightened, providing a more balanced intensity distribution and enhancing image contrast.

# Function for histogram equalization

def histogram \_equalize(image):

total = image.size # Total number of pixels

hist, \_ = np.histogram(image.ravel(), 256, [0, 256]) # Calculate histogram

cdf = hist.cumsum() # Cumulative distribution function (CDF)

cdf\_normalized = (cdf \* 255 / total).astype(np.uint8) # Normalize CDF to [0, 255]

equalized\_image = cdf\_normalized[image] # Map original image pixel values using the CDF

return equalized\_image

# Load the grayscale image

img = cv.imread("image.png", cv.IMREAD\_GRAYSCALE)

# Apply custom histogram equalization

equalized\_img = histogram \_equalize(img)

**Question 6 – Histogram equalizing the foreground of an image**

# (a) Convert image to HSV and split into hue, saturation, and value channels

hsv\_image = cv.cvtColor(image, cv.COLOR\_BGR2HSV)

hue, saturation, value = cv.split(hsv\_image)

****

# (b) Threshold the saturation plane to extract the foreground mask

saturation\_min, saturation\_max = 15, 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))) # Apply morphological closing to clean up the mask

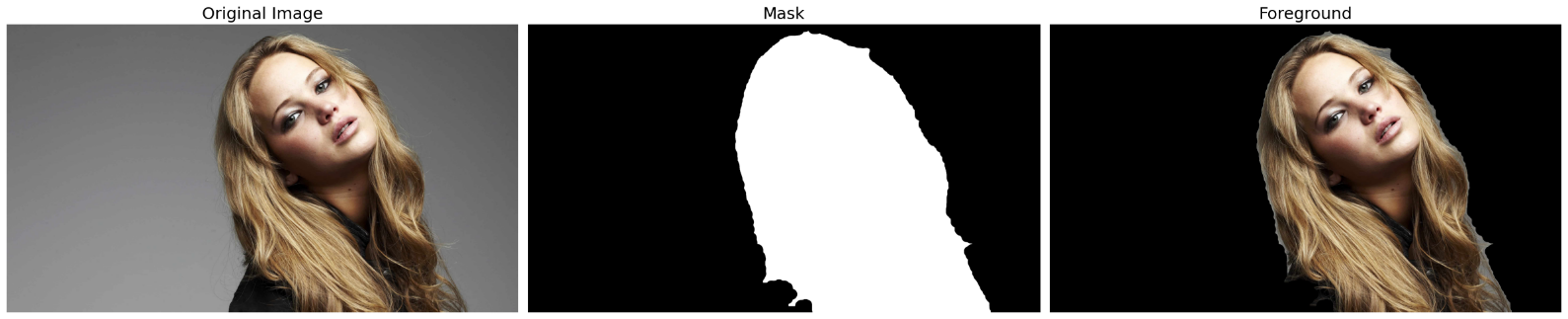
# (c) Obtain the foreground using the mask and compute its histogram

foreground = cv.bitwise\_and(image, image, mask=foreground\_mask)

histogram = cv.calcHist([foreground], [0], foreground\_mask, [256], [0, 256])

# (d) Compute the cumulative histogram (CDF)

cumulative\_histogram = np.cumsum(histogram)

****

# (e) Apply histogram equalization to the foreground's value plane

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) Extract the background and combine it with the equalized foreground

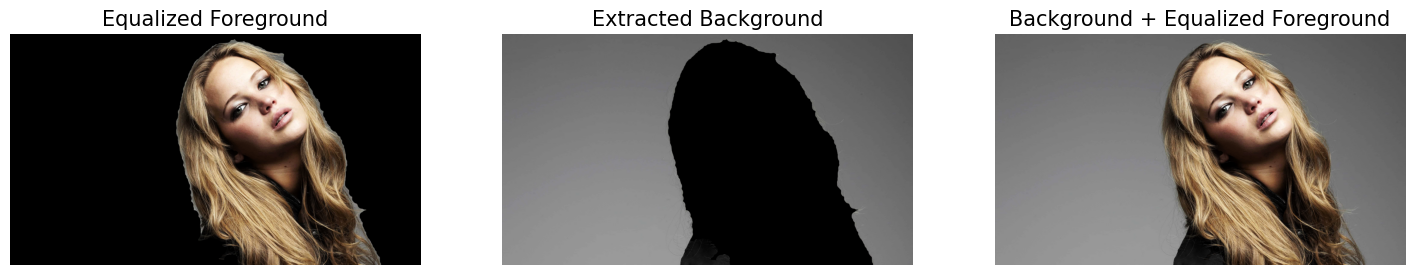
background\_mask = cv.bitwise\_not(foreground\_mask)

extracted\_background = cv.bitwise\_and(image, image, mask=background\_mask)

result = cv.add(extracted\_background, cv.cvtColor(hsv\_foreground, cv.COLOR\_HSV2BGR))

****

**Discussion:** The saturation plane was chosen for thresholding since the foreground is more saturated than the background. After thresholding with cv.inRange(), morphological operations were used to clean the mask and capture darker foreground details like the eyes. Histogram equalization was applied to enhance contrast in the value plane, and the foreground was combined with the original background to produce the final result.

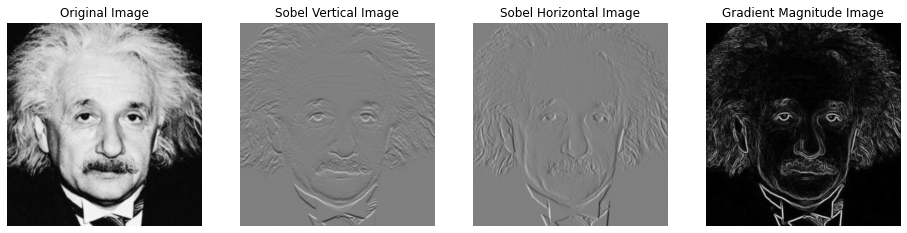
****

**Question 7 – Sobel filtering the photo of Albert Einstein**

**(a) Using filter2D**:This step applies the vertical and horizontal Sobel filters using cv.filter2D() with predefined Sobel kernels. The gradient magnitude is computed using both horizontal and vertical gradients.

**(b) Manual Application**:A manual convolution is performed by sliding a 3x3 Sobel kernel over the image and calculating the sum of pixel intensities for both vertical and horizontal directions.

**(c) Using sepFilter2D**:The separable Sobel filter is applied using cv.sepFilter2D() which decomposes the Sobel filter into two 1D filters and applies them along the rows and columns.

****

# (a) Using filter2D to apply Sobel filter

sobel\_v = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype='float32') # Vertical Sobel kernel

sobel\_h = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype='float32') # Horizontal Sobel kernel

imv = cv.filter2D(img, -1, sobel\_v) # Apply vertical Sobel

imh = cv.filter2D(img, -1, sobel\_h) # Apply horizontal Sobel

grad\_mag = np.sqrt(imv\*\*2 + imh\*\*2) # Compute gradient magnitude

# (b) Manually applying the Sobel filter

rows, cols = img.shape

kernel\_size = 3

imv\_manual = np.zeros((rows - kernel\_size + 1, cols - kernel\_size + 1), dtype='float32')

imh\_manual = np.zeros((rows - kernel\_size + 1, cols - kernel\_size + 1), dtype='float32')

for row in range(rows - kernel\_size + 1):

for col in range(cols - kernel\_size + 1):

imv\_manual[row, col] = np.sum(img[row:row + kernel\_size, col:col + kernel\_size] \* sobel\_v)

imh\_manual[row, col] = np.sum(img[row:row + kernel\_size, col:col + kernel\_size] \* sobel\_h)

grad\_mag\_manual = np.sqrt(imv\_manual\*\*2 + imh\_manual\*\*2) # Compute gradient magnitude

**Discussion:**

All methods yield similar results. The Sobel vertical filter emphasizes vertical edges, detecting intensity changes from top to bottom, while the horizontal filter emphasizes horizontal edges. Combining both gradients provides the overall gradient magnitude at each pixel. The output image dimensions are reduced by the kernel size during convolution.

# (c) Using sepFilter2D for Sobel filtering

sobel\_h\_kernel = np.array([1, 2, 1], dtype=np.float32)

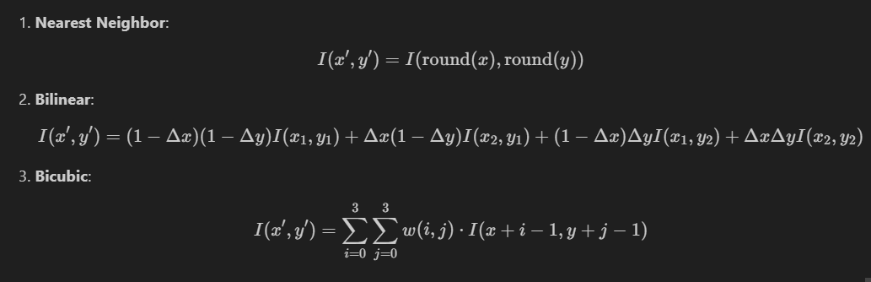
sobel\_v\_kernel = np.array([1, 0, -1], dtype=np.float32)

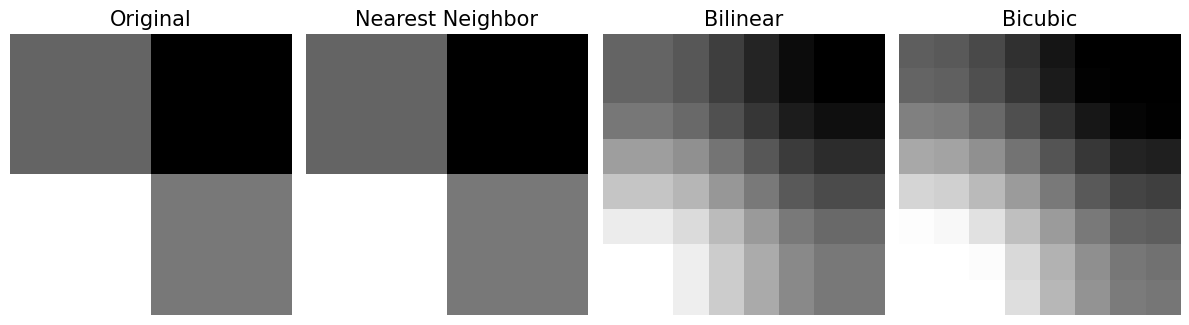
im1 = cv.sepFilter2D(img, -1, sobel\_h\_kernel, sobel\_v\_kernel) # Separable Sobel filtering

im2 = cv.sepFilter2D(img, -1, sobel\_v\_kernel, sobel\_h\_kernel) # Separable Sobel filtering

grad\_mag\_sep = np.sqrt(im1\*\*2 + im2\*\*2) # Compute gradient magnitude

**Question 8 –** **Zooming an image with nearest-neighbor and bilinear interpolation**

****

****

# Zoom function to handle nearest-neighbor and bilinear interpolation

def zoom\_image(image, scale, method):

rows, cols = int(image.shape[0] \* scale), int(image.shape[1] \* scale) # New dimensions

zoomed = np.zeros((rows, cols), dtype=image.dtype) # Initialize zoomed image

# (a) Nearest-neighbor interpolation

if method == 'nearest-neighbour':

for i in range(rows):

for j in range(cols):

original\_i = min(int(round(i / scale)), image.shape[0] - 1)

original\_j = min(int(round(j / scale)), image.shape[1] - 1)

zoomed[i, j] = image[original\_i, original\_j]

# (b) Bilinear interpolation

elif method == 'bilinear':

for i in range(rows):

for j in range(cols):

x, y = i / scale, j / scale

x1, y1 = int(np.floor(x)), int(np.floor(y))

x2, y2 = min(int(np.ceil(x)), image.shape[0] - 1), min(int(np.ceil(y)), image.shape[1] - 1)

dx, dy = x - x1, y - y1

zoomed[i, j] = (

image[x1, y1] \* (1 - dx) \* (1 - dy) +

image[x1, y2] \* (1 - dx) \* dy +

image[x2, y1] \* dx \* (1 - dy) +

image[x2, y2] \* dx \* dy

)

return zoomed

# Compute normalized sum of squared differences (SSD)

def compute\_ssd(image1, image2):

image2\_resized = cv.resize(image2, (image1.shape[1], image1.shape[0])) # Ensure same size

return np.sum((image1 - image2\_resized) \*\* 2) / np.prod(image1.shape)

# Example usage with image and scale factor

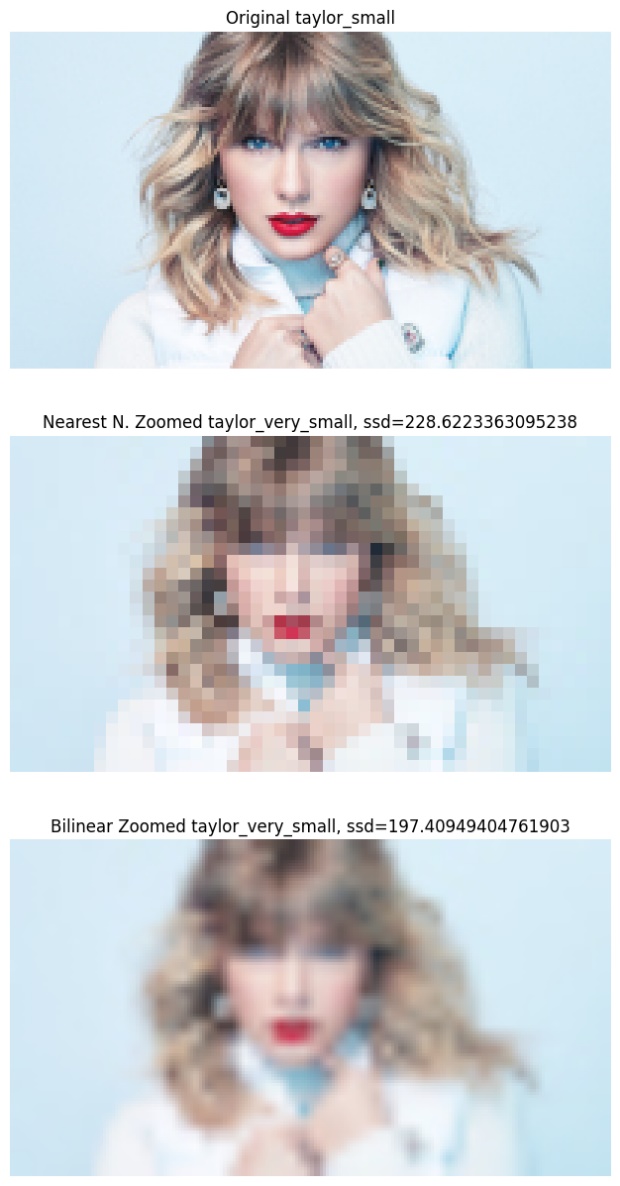
scale\_factor = 4

zoomed\_nn = zoom\_image(img, scale\_factor, method='nearest-neighbour')

zoomed\_bilinear = zoom\_image(img, scale\_factor, method='bilinear')

ssd\_nn = compute\_ssd(zoomed\_nn, original\_large\_image) # SSD for nearest-neighbor

ssd\_bilinear = compute\_ssd(zoomed\_bilinear, original\_large\_image) # SSD for bilinear

****

**Code:**

**1]** Nearest-Neighbor Interpolation: Each pixel in the zoomed image is assigned the value of the nearest pixel from the original image.

2] Bilinear Interpolation: Uses linear interpolation in both x and y directions, resulting in smoother but slightly blurred images.

3] SSD Calculation: The compute\_ssd() function computes the normalized sum of squared differences between the zoomed image and the original large image after resizing.

**Discussion:** Bilinear interpolation offers smoother results compared to nearest-neighbor, which can cause blocky artifacts. However, some blurring may still occur. Normalized SSD helps compare the resized image with the original, but results may still show noise and artifacts despite size adjustments.

**Question 9 – Segmentation of a Yellow Daisy**

# Initialize mask and models for GrabCut

mask = np.zeros(img.shape[:2], np.uint8)

bg\_model = np.zeros((1, 65), np.float64)

fg\_model = np.zeros((1, 65), np.float64)

rect = (40, 10, 505, 505)

# Apply GrabCut for initial segmentation

cv.grabCut(img, mask, rect, bg\_model, fg\_model, 5, cv.GC\_INIT\_WITH\_RECT)

# Extract foreground mask and foreground image

mask1 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

foreground = img \* mask1[:, :, np.newaxis]

# Extract background mask and background image

mask2 = np.where((mask == 3) | (mask == 1), 0, 1).astype('uint8')

background = img \* mask2[:, :, np.newaxis]

# Apply Gaussian blur to the background and combine with the foreground

blurred\_img = foreground + cv.GaussianBlur(background, (15, 15), 0)



**Dark Edge Explanation**: The dark edge in the enhanced image occurs because background pixels near the boundary, especially those inaccurately segmented, mix with black pixels (masked) during Gaussian blurring, creating a dark halo around the flower.