Department of Electronic and Telecommunications Engineering

EN3160 Image Processing and Machine Vision

Intensity Transformations and Neighborhood Filtering

Assignment 1

Eashan S.G.S (220148G)

August 10, 2025

GitHub Repository

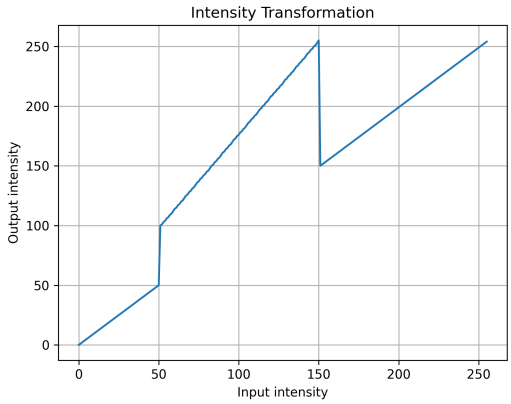
Link to GitHub repository: `https://github.com/sahas-eashan/EN3160-Assignment-1`

1 Question 1

The given intensity transformation can be realized using piecewise linear segments. The transformation enhances mid-intensity pixels while keeping low and high intensity pixels relatively unchanged.

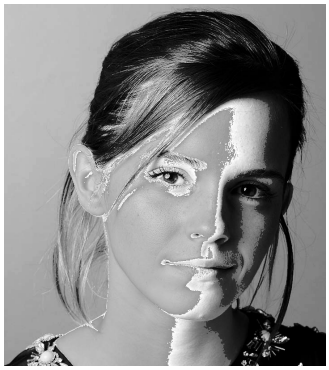
Listing 1: Intensity transformation implementation

```python
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

# Define piecewise transformation function
c = np.array([(50, 50), (150, 255)], dtype=np.uint8)
t1 = np.linspace(0, 50, 51, dtype=np.uint8)
t2 = np.linspace(100, 255, 150 - 50, dtype=np.uint8)
t3 = np.linspace(150, 255, 256 - 150, dtype=np.uint8)

# Concatenate all segments to create the transformation array
lut = np.concatenate((t1, t2, t3), axis=0)
lut = lut[:256]
```



(a) Intensity transformation curve

(b) Original image



(c) transformed image

Figure 1: Intensity transformation results for Question 1

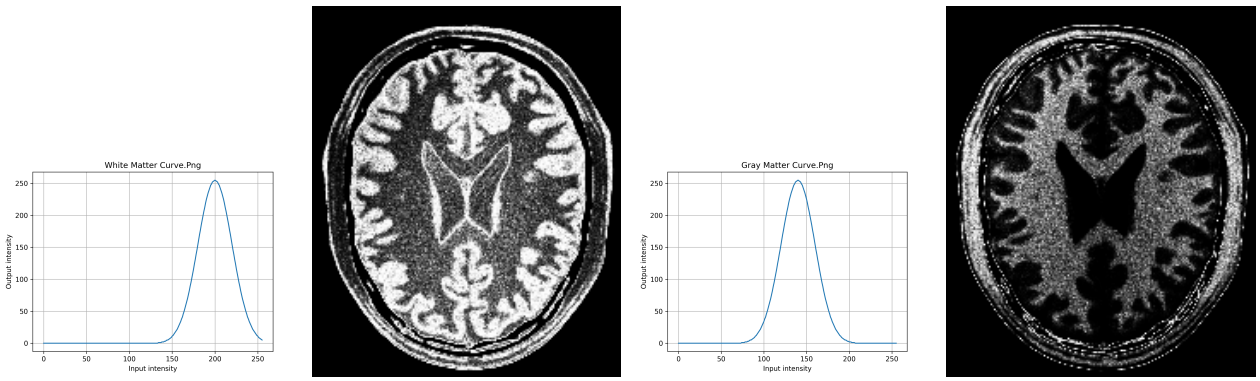
Interpretation: The transformation creates jump discontinuities that result in high contrast regions. Mid-intensity pixels are enhanced while preserving the extreme intensity values.

2 Question 2

To accentuate white and gray matter in the brain MRI, I used Gaussian pulse transformations centered at different intensity values to highlight specific tissue types.

Listing 2: Brain tissue enhancement

```python
# White matter enhancement (centered around intensity 150)
mu_white, sigma_white = 200, 20
lut_white = (
    (255 * np.exp(-((x - mu_white) ** 2) / (2 * sigma_white**2)))
    .clip(0, 255)
    .astype(np.uint8)
)

# Gray matter enhancement (centered around intensity 200)
mu_gray, sigma_gray = 140, 20
lut_gray = (
    (255 * np.exp(-((x - mu_gray) ** 2) / (2 * sigma_gray**2)))
    .clip(0, 255)
    .astype(np.uint8)
)
```



(a) White matter curve (b) White matter enhanced (c) Gray matter curve (d) Gray matter enhanced

Figure 2: Brain tissue enhancement results

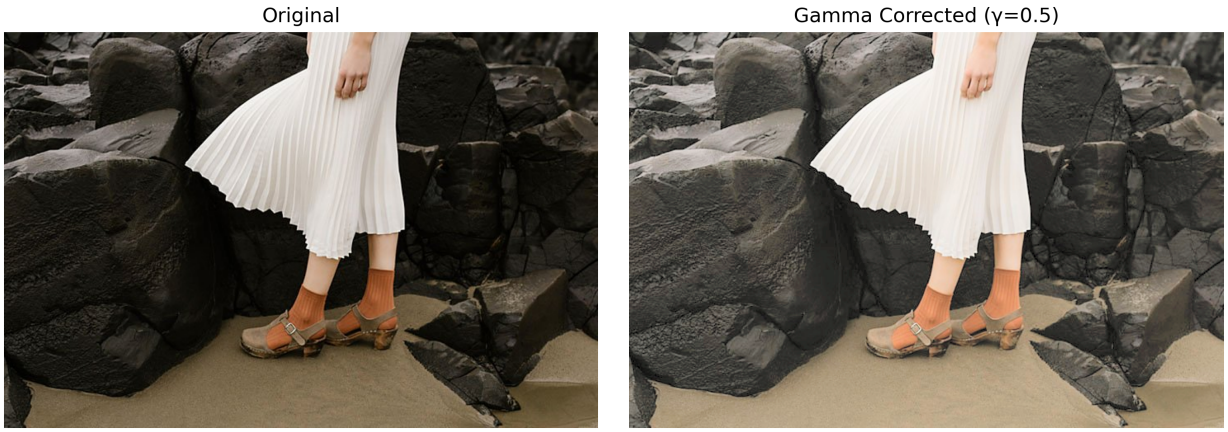
Interpretation: The Gaussian pulse transformations provide smooth contrast enhancement for specific intensity ranges, effectively highlighting white and gray matter tissues.

3 Question 3

Gamma correction was applied to the L* channel of the L*a*b* color space to adjust image lightness and improve shadow details.

Listing 3: Gamma correction implementation

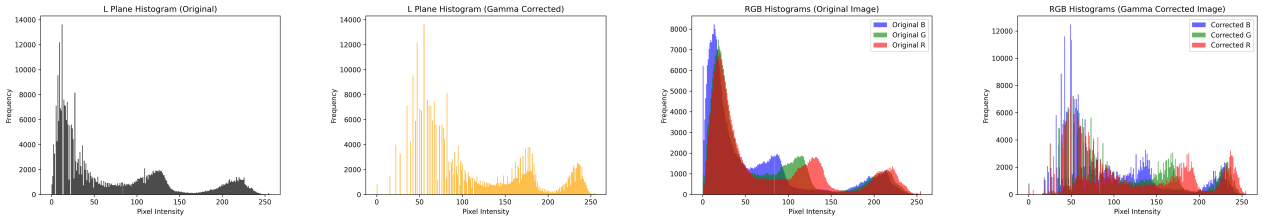
```
# Convert to L*a*b* color space
lab = cv.cvtColor(img, cv.COLOR_BGR2Lab)
L, a, b = cv.split(lab)

# Apply gamma correction (gamma = 0.5)
gamma = 0.5
L_corrected = np.clip((L / 255.0) ** gamma * 255, 0, 255).astype(np.uint8)

# Merge channels back
lab_corrected = cv.merge((L_corrected, a, b))
img_corrected = cv.cvtColor(lab_corrected, cv.COLOR_Lab2BGR)
```



(a) Original image (b) Gamma corrected image

Figure 3: Gamma correction results (gamma = 0.5)

Interpretation: Gamma correction with gamma = 0.5 lightens the shadows while preserving highlights, improving overall image visibility.



(a) L* channel histogram (original) (b) L* channel histogram (corrected) (c) RGB histogram (original) (d) RGB histogram (corrected)

Figure 4: Histograms of L* and RGB channels (original vs corrected)

4 Question 4

The vibrance enhancement was achieved by applying a specific transformation to the saturation channel in HSV color space.

Listing 4: Vibrance enhancement

```
# Convert to HSV and split channels
hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
H, S, V = cv.split(hsv)

# Define vibrance transformation
a = 0.65
sigma = 70.0
x = np.arange(256, dtype=np.float32)
f = x + a * 128.0 * np.exp(-((x - 128.0) ** 2) / (2.0 * sigma**2))
lut = np.clip(f, 0, 255).astype(np.uint8)

# Apply to saturation channel
S_enh = cv.LUT(S, lut)
hsv_enh = cv.merge((H, S_enh, V))
img_enh = cv.cvtColor(hsv_enh, cv.COLOR_HSV2BGR)
```

Figure 5: HSV channel separation

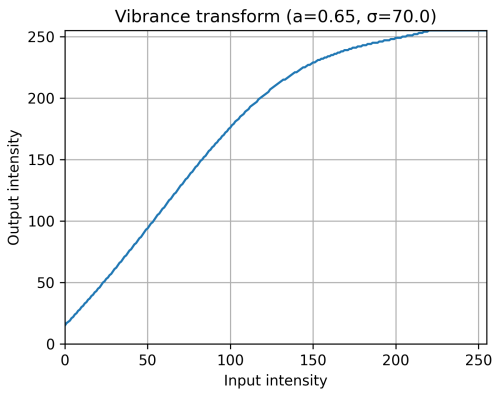


Figure 6: Vibrance transformation

(a) Original image (b) Enhanced result (a = 0.65)

Figure 7: Vibrance enhancement results

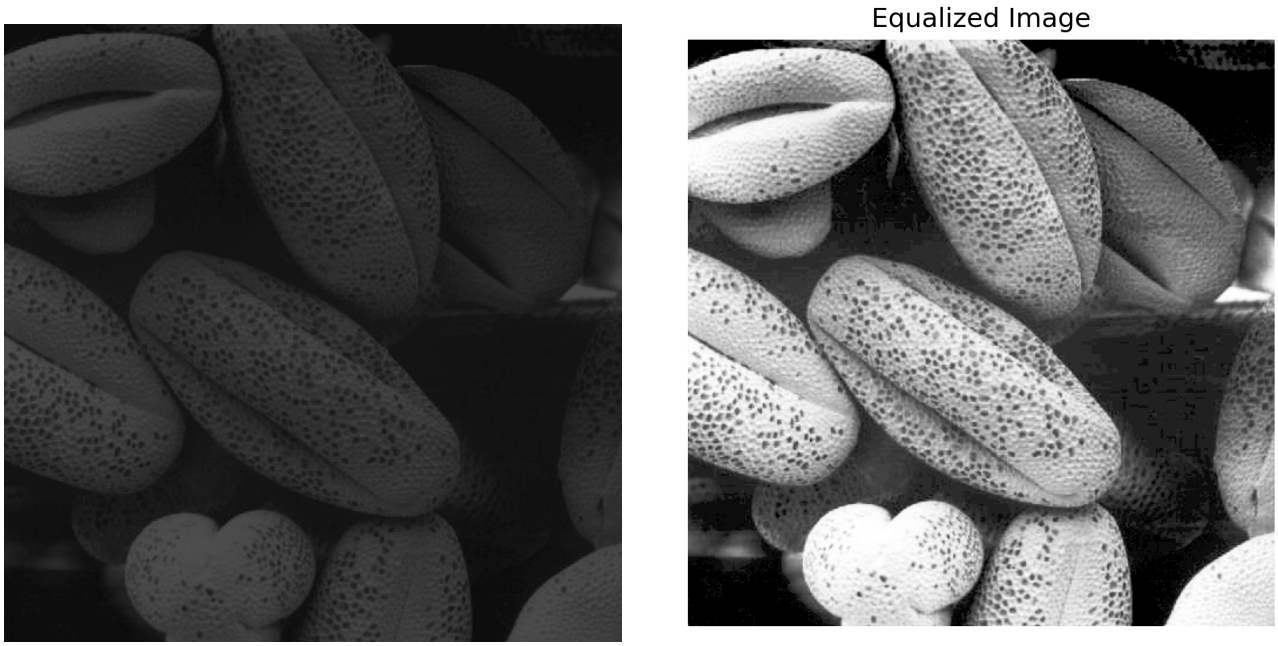
Interpretation: The transformation selectively enhances colorful regions while preserving grayscale areas, effectively increasing image vibrance.

5 Question 5

Custom histogram equalization was implemented to spread the intensity distribution uniformly across the full dynamic range.

Listing 5: Custom histogram equalization function

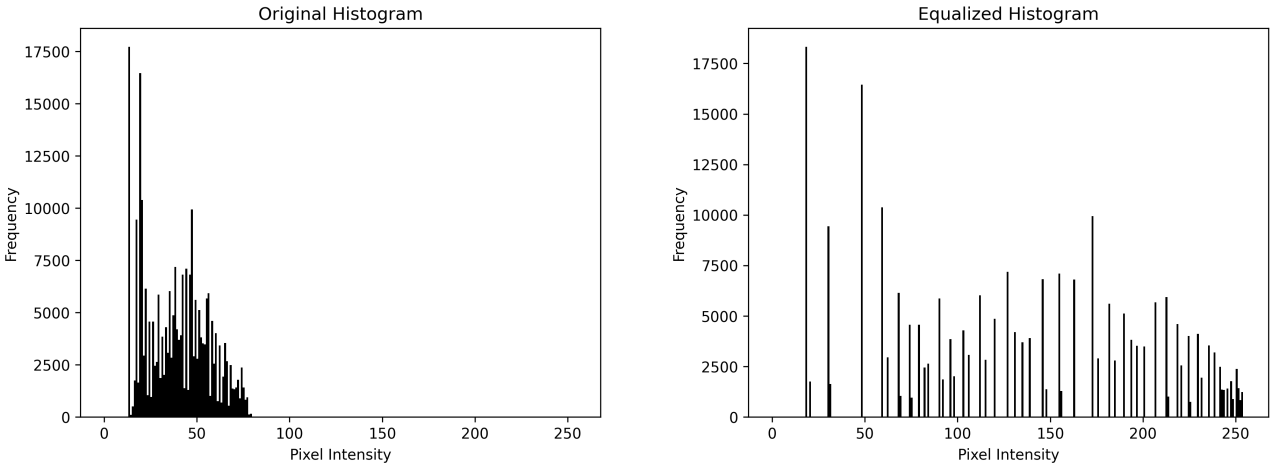
```
def my_hist_equalization(img):
    L = 256
    M, N = img.shape
    hist = cv.calcHist([img], [0], None, [L], [0, L])
    cdf = hist.cumsum()

    t = np.array([(L - 1) / (M * N) * cdf[K] for K in range(L)]).astype("uint8")
    return t[img]
```



(a) Original image (b) Equalized image

Figure 8: Histogram equalization results

Figure 10: HSV planes



(a) Original histogram

(b) Equalized histogram

Figure 9: Histogram comparison

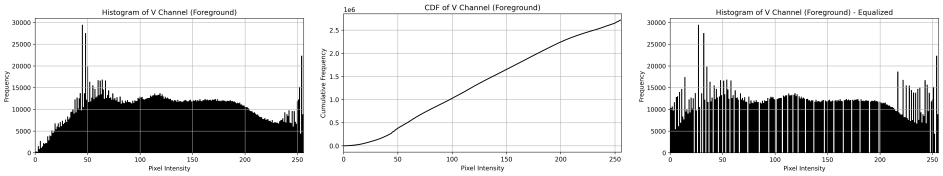
Interpretation: The equalization process spreads the histogram more uniformly, improving overall contrast and detail visibility.

6 Question 6

Selective histogram equalization was applied only to the foreground region using mask-based processing. The saturation plane was used to create a mask that isolates the foreground.

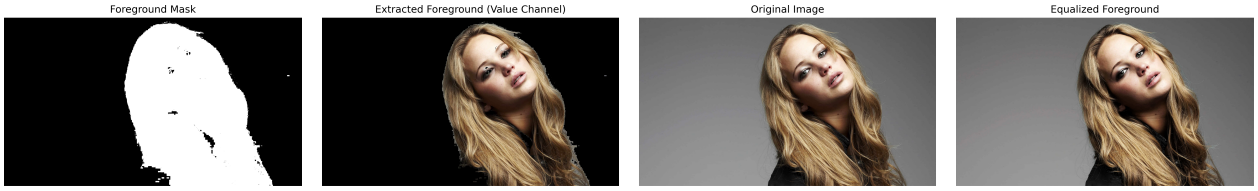
Listing 6: Foreground histogram equalization

```
# Convert to HSV and split channels
img_hsv = cv.cvtColor(img_bgr, cv.COLOR_BGR2HSV)
h, s, v = cv.split(img_hsv)

# Create mask using Saturation channel
_, mask = cv.threshold(s, 12, 255, cv.THRESH_BINARY)

# Extract foreground
foreground = cv.bitwise_and(img_bgr, img_bgr, mask=mask)

foreground_hsv = cv.cvtColor(foreground, cv.COLOR_BGR2HSV)
H_fg, S_fg, V_fg = cv.split(foreground_hsv)

hist = cv.calcHist([V_fg], [0], mask, [256], [0, 256])
x_positions = np.arange(len(hist))

cdf = hist.cumsum()

# Equalize only the foreground
v_eq = my_hist_equalization(V_fg, cdf)
hist_eq = cv.calcHist([v_eq], [0], mask, [256], [0, 256])
x_positions_eq = np.arange(len(hist_eq))

# Merge back channels
hsv_eq = cv.merge([H_fg, S_fg, v_eq])
modified_fg = cv.cvtColor(hsv_eq, cv.COLOR_HSV2BGR)

background = cv.bitwise_and(img_bgr, img_bgr, mask=cv.bitwise_not(mask))
final_img_bgr = cv.add(cv.cvtColor(background, cv.COLOR_BGR2RGB), modified_fg)
```



(a) original histogram (b) Original CDF (c) equalized histogram

Figure 11: Foreground histogram equalization process



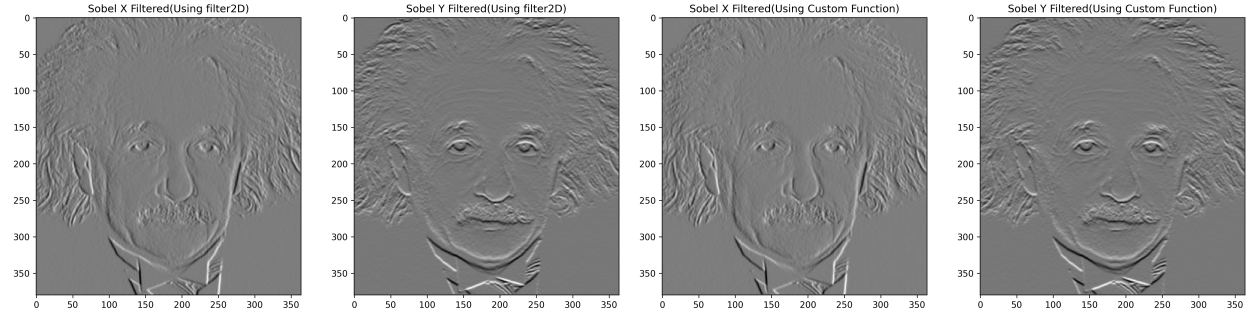
(a) Foreground mask (b) Extracted foreground (c) Original image (d) Final result

Figure 12: Foreground histogram equalization process

Interpretation: Selective equalization enhances foreground details while preserving the background, resulting in improved subject contrast.

7 Question 7

Sobel filtering was implemented using three different approaches: OpenCV's filter2D, custom 2D convolution, and separable convolution.



(a) OpenCV filter2D result (b) Custom implementation

Figure 13: Sobel filtering comparison

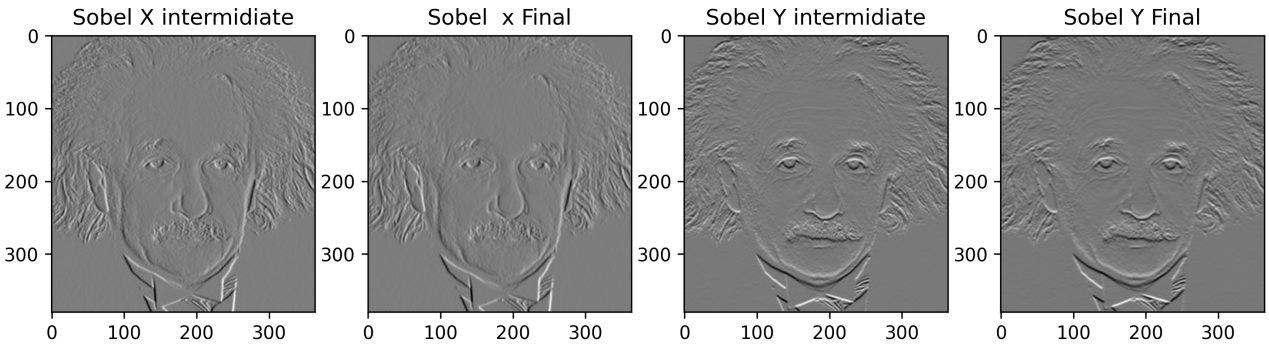


Figure 14: Separable convolution

Listing 7: Custom Sobel filtering implementation

```python
def sobel_filter(img, filter):
    """Apply Sobel filter to an image."""
    rows, cols = img.shape
    filtered_img = np.zeros_like(img, dtype=np.float64)

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            region = img[i - 1 : i + 2, j - 1 : j + 2]
            filtered_img[i, j] = np.sum(region * filter)

    return filtered_img

# Sobel kernels
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]])
# Separable kernels
sobel_x_vertical = np.array([[1], [2], [1]])
sobel_x_horizontal = np.array([[1, 0, -1]])

sobel_y_vertical = np.array([[1], [0], [-1]])
sobel_y_horizontal = np.array([[1, 2, 1]])

x1 = cv.filter2D(einstein, cv.CV_64F, sobel_x_horizontal)
x2 = cv.filter2D(x1, cv.CV_64F, sobel_x_vertical)

y1 = cv.filter2D(einstein, cv.CV_64F, sobel_y_vertical)
y2 = cv.filter2D(y1, cv.CV_64F, sobel_y_horizontal)
```

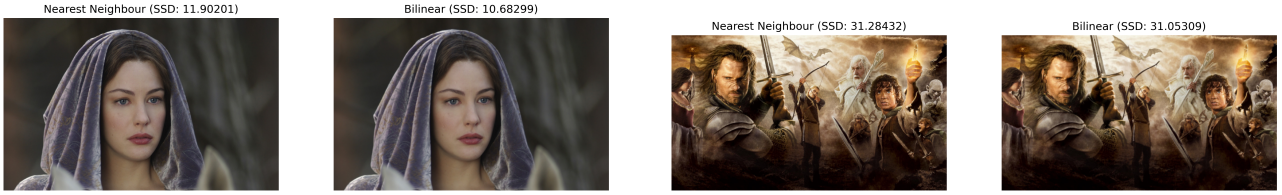
Interpretation: All three approaches produce similar edge detection results, with separable convolution being computationally more efficient.

8 Question 8

Image zooming was implemented using both nearest neighbor and bilinear interpolation methods.

Listing 8: Image zooming implementation

```python
# zoom the image with given scale (s)
def zoom_cv(img, s: float, method: str = "nearest"):
    interp = cv.INTER_NEAREST if method == "nearest" else cv.INTER_LINEAR  # bilinear
    h, w = img.shape[:2]
    out = cv.resize(img, None, fx=s, fy=s, interpolation=interp)
    return out

# Calculate normalized SSD for comparison
def normalized_ssd(A, B, max_val=255.0):
    if A.shape == B.shape:
        ssd = np.sum((A - B) ** 2)
        return float(ssd / A.size)
    else:
        print("A and B must have the same shape")
        return None
```



(a) Nearest neighbor comparison (b) Bilinear comparison

Figure 15: Image zooming results with SSD values

Interpretation: Bilinear interpolation produces smoother and more visually pleasing results compared to nearest neighbor. The SSD value is slightly lower for bilinear than for nearest neighbor, indicating a closer match to the reference image.

9 Question 9

GrabCut algorithm was used for foreground-background segmentation followed by selective background blurring.

Listing 9: GrabCut segmentation and background blur

```
# GrabCut segmentation
mask = np.zeros(img.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
rect = (50, 100, 550, 550)

cv.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)

# Convert mask to binary foreground/background
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype("uint8")
foreground = img * mask2[:, :, np.newaxis]
background = img * (1 - mask2[:, :, np.newaxis])

# Blur background
blurred_bg = cv.GaussianBlur(background, (51, 51), 0)
enhanced_img = cv.add(foreground, blurred_bg)

# Convert mask to 3-channel for visualization
mask_vis = cv.cvtColor(mask2 * 255, cv.COLOR_GRAY2BGR)
```

Figure 16: GrabCut segmentation and background blur results

Interpretation: The background appears dark near the flower edges because the Gaussian blur averages neighboring pixels, and pixels replaced by zero (from foreground extraction) contribute to this darkening effect at the boundaries.

Reason for dark background at flower edges: When applying Gaussian blur to the background, the kernel averages surrounding pixels. Near the flower edges, some of these pixels have been set to zero during foreground extraction, causing the averaged values to be darker than the original background pixels.