Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN3160 - Image Processing and Machine Vision
Assignment 1
Intensity Transformations and Neighborhood Filtering

NAME: HATHURUSINGHA HAR
INDEX NO: 220221B

Date of Submission: 09/07/2025

GitHub Repository
`https://github.com/Achi-456/CV-Assignment-1.git`

1 Question 1 - Piecewise Linear Intensity Transformation

```
t1 = np.linspace(0, 50, 51).astype(np.uint8)
t2 = np.linspace(100, 255, 100).astype(np.uint8)
t3 = np.linspace(150, 255, 105).astype(np.uint8)
lut = np.concatenate((t1, t2, t3), axis=0).astype(np.uint8)
img_transformed = cv2.LUT(img_orig, lut)
```

Listing 1: Piecewise Linear Intensity Transformation

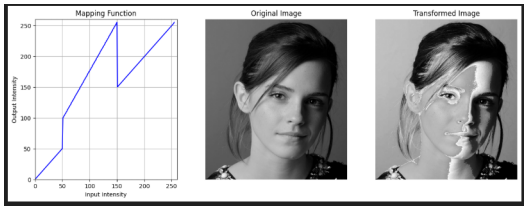


Figure 1: Original and transformed images

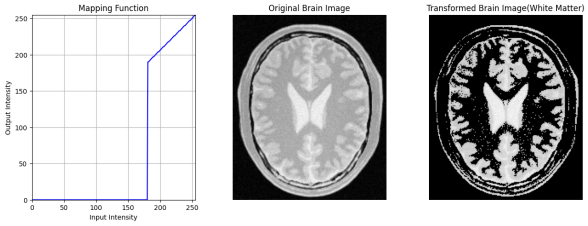
The implementation creates a piecewise linear transformation with three segments using np.linspace() - t1 (0-50→0-50), t2 (51-150→100-255), and t3 (151-255→150-255) - concatenated into a LUT and applied via cv2.LUT(). Results: The transformation function shows three distinct slopes: flat initial, steep middle, and moderate final segments. The transformed image exhibits enhanced mid-tone contrast with a steep t2 slope brightening facial features, while preserving shadow details in dark regions and compressing highlights to prevent overexposure, making it effective for images with poor mid-tone contrast.

2 Question 2 - Brain Tissue Enhancement

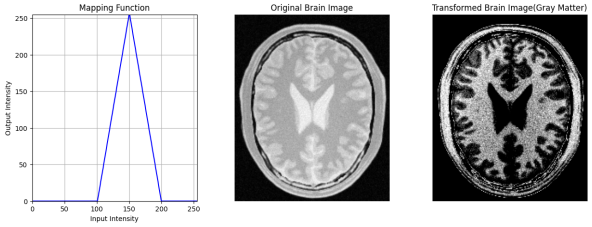
Using LUT approach to accentuate brain tissue regions through piecewise linear transformations for white matter and gray matter enhancement.

```
# White Matter Enhancement
t1 = np.linspace(0, 0, 181).astype(np.uint8)
t2 = np.linspace(190, 255, 75).astype(np.uint8)
wm_lut = np.concatenate((t1, t2), axis=0).astype(np.uint8)
img_brain_transformed_WM = cv.LUT(img_brain, wm_lut)

# Gray Matter Enhancement
t1_gm = np.linspace(0, 0, 100).astype(np.uint8)
t2_gm = np.linspace(0, 255, 50).astype(np.uint8)
t3_gm = np.linspace(255, 0, 106).astype(np.uint8)
gm_lut = np.concatenate((t1_gm, t2_gm, t3_gm), axis=0).astype(np.uint8)
img_brain_transformed_GM = cv.LUT(img_brain, gm_lut)
```

Listing 2: Brain Tissue Enhancement Code



(a) White matter transformation with mapping function and results



(b) Gray matter transformation with mapping function and results

Figure 2: Brain tissue enhancement results

The LUT approach enhances brain tissue using piecewise linear transformations. For white matter, intensities 0-181 are mapped with a 0.0 slope, and 181-255 with 1.0, accentuating white matter. For gray matter, 0-100 uses a 0.0 slope, and 100-255 uses 1.0, highlighting gray matter, as coded in C.

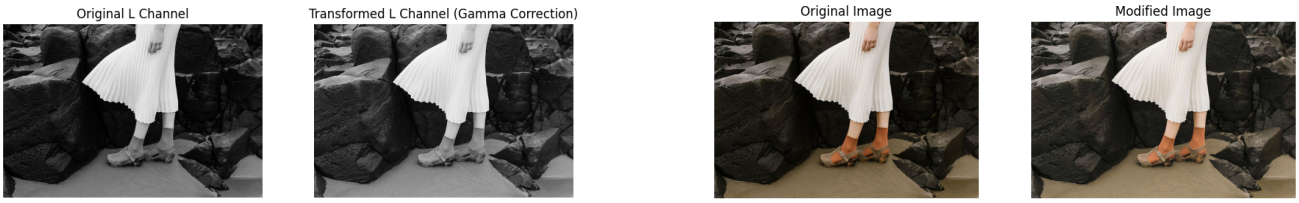
Results in Figure 2 show effective enhancement. White matter transformation clearly boosts white matter visibility with a steep intensity rise above 181. Gray matter transformation similarly improves gray matter contrast above 100, both outperforming the original MRI.

3 Question 3 - Gamma Correction in L*a*b* Space

```
chnl_img = cv2.cvtColor(org_img, cv2.COLOR_BGR2Lab)
L_channel, a_channel, b_channel = cv2.split(chnl_img)
gamma = 0.75
table = np.array([((i / 255.0) ** gamma) * 255 for i in np.arange(0, 256)]).astype("uint8")
L_channel_transformed = cv2.LUT(L_channel, table)
modified_lab_img = cv2.merge((L_channel_transformed, a_channel, b_channel))
modified_rgb_img = cv2.cvtColor(modified_lab_img, cv2.COLOR_Lab2RGB)
```

Listing 3: Gamma Correction on L Channel

The image is converted from BGR to RGB, then to L*a*b* space. Gamma correction is applied to the L channel to enhance brightness and contrast. An initial value of $\gamma = 0.5$ was tested, and several values were tried to balance highlight enhancement with shadow detail, with $= 0.75$ chosen as the optimal setting. The corrected L channel is then recombined with the original a and b channels and converted back to RGB for display.



(a) L channels comparison (b) Original vs transformed

Figure 3: Gamma correction results ($\gamma = 0.75$)

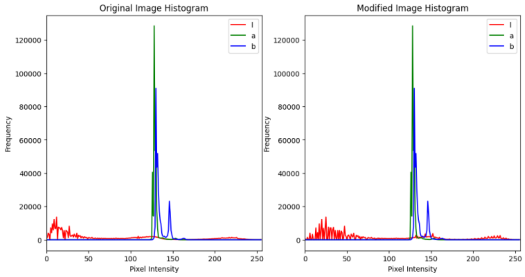


Figure 4: Histogram comparison showing enhanced L channel distribution

In the original image histogram, the L-channel shows a broad distribution, while the transformed histogram indicates a shift with enhanced intensity peaks, reflecting the correction's effect. The a and b channels remain unchanged, maintaining their original distributions in both histograms, as seen in the green and blue lines. This selective adjustment of the L-channel improves brightness and contrast in the modified image, as evident in the comparison between the original and transformed images

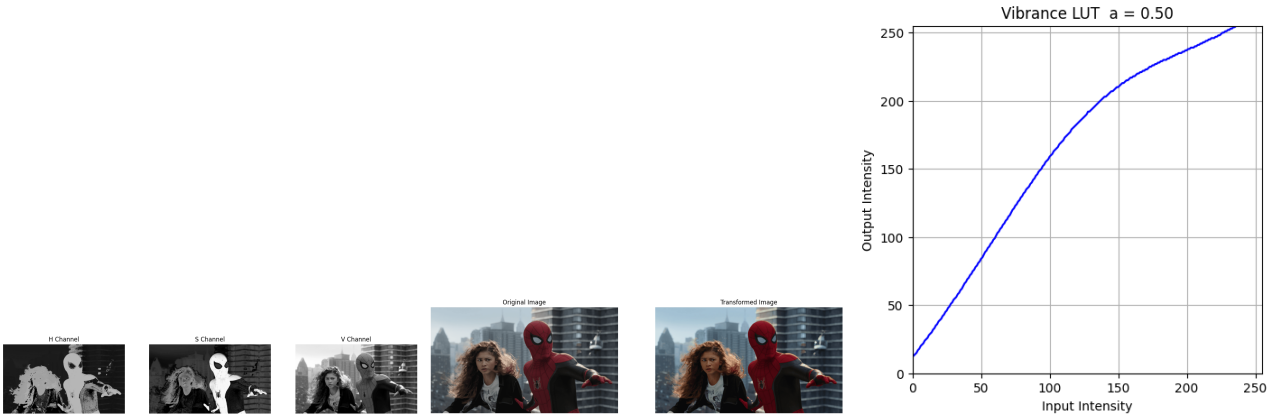
4 Question 4 - Vibrance Enhancement

Vibrance enhancement using intensity transformation in HSV space: $f(x) = \min(x + a \times 128 \times e^{-(x-128)^2/(2\sigma^2)}, 255)$ where $a = 0.5$ and $\sigma = 70$.

```python
def vibrance_transform(image, a, sigma=70):
    hsv_img = cv.cvtColor(cv.cvtColor(image, cv.COLOR_BGR2RGB), cv.COLOR_BGR2HSV)
    h_channel, s_channel, v_channel = cv.split(hsv_img)

    lut = np.array([int(min(x + (a * 128) * np.exp((-(x - 128) ** 2) / (2 * (sigma ** 2))),
        255))
                    for x in range(256)], dtype=np.uint8)

    s_channel_transformed = cv.LUT(s_channel, lut)
    hsv_img_transformed = cv.merge((h_channel, s_channel_transformed, v_channel))
    return cv.cvtColor(hsv_img_transformed, cv.COLOR_HSV2BGR)
```

Listing 4: Vibrance Transformation

The image is first read in BGR format and converted to RGB, then transformed into the HSV color space using OpenCV. The HSV channels (hue, saturation, and value) are split, and the vibrance transformation is applied to the saturation channel using a lookup table (LUT) generated from the given function. The transformed saturation channel is merged with the original hue and value channels, and the result is converted back to BGR. Various values of a from 0 to 1 in steps of 0.1 were tested, with a = 0.5 selected for a visually pleasing output.



(a) HSV channels (b) Original vs enhanced (c) Vibrance LUT

Figure 5: Vibrance enhancement results

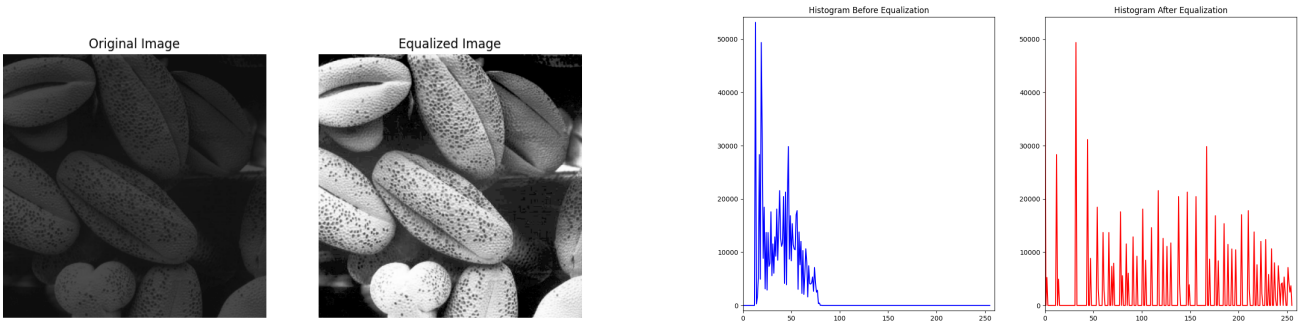
The transformed image compares the original image with the transformed image using a = 0.5. The transformed image exhibits enhanced vibrance, with more vivid colors in the costumes and background, improving the overall visual appeal while maintaining the scene's integrity. The third image shows the intensity transformation plot for a = 0.5, illustrating a smooth curve that increases the output intensity around the mid-range (around 128), enhancing saturation while capping at 255, which contributes to the balanced vibrance enhancement observed.

5 Question 5 - Custom Histogram Equalization

```python
def custom_histogram_equalization(image):
    hist, bins = np.histogram(image.flatten(), bins=256, range=[0, 256])
    cdf = hist.cumsum()
    cdf_m = np.ma.masked_equal(cdf, 0)
    cdf_m = (cdf_m - cdf_m.min()) * 255 / (cdf_m.max() - cdf_m.min())
    cdf_final = np.ma.filled(cdf_m, 0).astype('uint8')
    img_equalized = cdf_final[image]
    return img_equalized
```

Listing 5: Custom Histogram Equalization

The algorithm computes the image histogram, calculates the cumulative distribution function (CDF), normalizes it to [0,255] range while handling zero values with masking, and maps original pixel values to equalized values using the CDF as a lookup table.



(a) Images comparison (b) Histogram comparison

Figure 6: Histogram equalization results

The original grayscale image shows limited contrast with obscured microscopic details due to narrow intensity range. The equalized image exhibits dramatically enhanced contrast with clearer texture and structural details. **Histogram Analysis:** The original histogram shows a concentrated peak at lower intensities indicating a dark image. After equalization, the histogram is more uniformly distributed across the full intensity range, confirming improved contrast and broader dynamic range that enhances detail visibility.

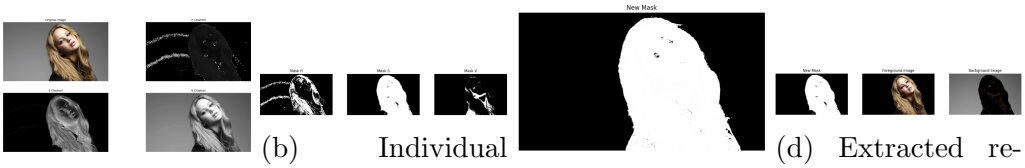
6 Question 6 - Foreground Histogram Equalization

Selective histogram equalization applied to foreground using HSV-based masking and channel-wise equalization.

```
# HSV conversion and thresholding
hsv_img = cv2.cvtColor(img_jeniffer, cv2.COLOR_BGR2HSV)
h_channel, s_channel, v_channel = cv2.split(hsv_img)

# Create combined mask (thresholds: 15, 50)
_, mask_s = cv2.threshold(s_channel, 15, 255, cv2.THRESH_BINARY)
_, mask_v = cv2.threshold(v_channel, 50, 255, cv2.THRESH_BINARY)
mask = mask_s + 255 - mask_v

# Extract and equalize foreground
foreground_img = cv.bitwise_and(img_jeniffer, img_jeniffer, mask=mask.astype(np.uint8))
background_img = cv.bitwise_and(img_jeniffer, img_jeniffer, mask=cv.bitwise_not(mask.astype(np.
    uint8)))

# Apply histogram equalization to RGB channels
for channel in range(3):
    foreground_img[:,:,channel] = custom_histogram_equalization(foreground_img[:,:,channel])

# Recombine foreground and background
final_img = cv.bitwise_or(foreground_equalized, background_img)
```

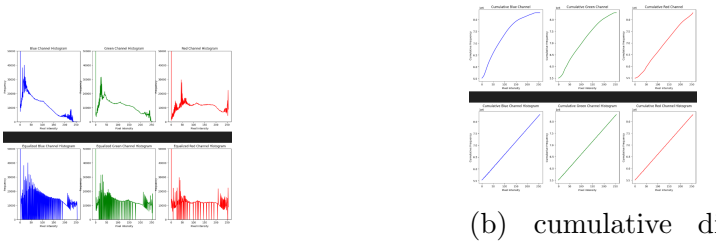
Listing 6: Foreground Histogram Equalization

The process converts the image to HSV space, creates masks using saturation and value thresholds (15, 50), combines them as $mask = mask_s + 255 - mask_v$ to isolate the foreground, applies histogram equalization to each RGB channel of the extracted foreground, and recombines with the unchanged background. **Results:** The original histograms show concentrated peaks at lower intensities. After equalization, the cumulative distribution functions become more



(a) HSV channels (b) Individual masks (c) Combined mask (d) Extracted regions

Figure 7: Foreground histogram equalization process



(a) RGB histograms before/after (b) cumulative distributions before/after

linear, and the histograms spread uniformly across the intensity range for all RGB channels, resulting in enhanced foreground contrast while preserving the background.

Figure 9: Final Result

7 Question 7 - Sobel Edge Detection

The commonoutput image displays the gradient magnitude, revealing sharp edges and contours of the subject (e.g., Einstein's face) with enhanced visibility of facial features and hairlines, resulting from the Sobel filtering process across all methods.

Figure 10: Common output: gradient magnitude showing edge detection

Method A: OpenCV filter2D

```
sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=np.float32)
sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=np.float32)
grad_x = cv.filter2D(image, cv.CV_32F, sobel_x)
grad_y = cv.filter2D(image, cv.CV_32F, sobel_y)
magnitude = np.sqrt(grad_x**2 + grad_y**2)
```

Listing 7: OpenCV Sobel Filter

This code uses OpenCV's filter2D to apply Sobel kernels for x and y gradients. It defines the standard Sobel kernels, computes gradients, calculates the magnitude using the Euclidean norm, and clips the results to the [0, 255] range for display as uint8 images.

Method B: Custom Implementation

```
def custom_convolution(image, kernel):
    img_h, img_w = image.shape
    kernel_h, kernel_w = kernel.shape
    pad_h, pad_w = kernel_h // 2, kernel_w // 2
    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    output_image = np.zeros_like(image, dtype=np.float32)

    for i in range(img_h):
        for j in range(img_w):
            region = padded_image[i:i + kernel_h, j:j + kernel_w]
            output_image[i, j] = np.sum(region * kernel)
    return output_image
```

Listing 8: Custom Convolution

This code implements a custom convolution function that applies the Sobel kernels by padding the image, sliding the kernel over each pixel,and computing the dot product.

Method C: Separable Kernels

```
# Sobel X using separable kernels
kernel_x_v = np.array([[1], [2], [1]], dtype=np.float32)
kernel_x_h = np.array([[1, 0, -1]], dtype=np.float32)
temp_x = cv.filter2D(img_einstein, cv.CV_32F, kernel_x_v)
sobel_x_sep = cv.filter2D(temp_x, cv.CV_32F, kernel_x_h)
```

Listing 9: Separable Sobel Implementation

This code leverages the separable property of the Sobel operator, decomposing it into vertical and horizontal kernels. It applies filter2D twice for each direction (x and y), first with a vertical kernel and then a horizontal kernel

8 Question 8 - Image Zooming

Two interpolation methods for image zooming with scale factor $s \in (0, 10]$.

Nearest-Neighbor Interpolation

```
def nearest_neighbor_zoom(image, scale_factor):
    img_h, img_w, img_c = image.shape
    new_h, new_w = int(img_h * scale_factor), int(img_w * scale_factor)
    new_image = np.zeros((new_h, new_w, img_c), dtype=image.dtype)

    for i in range(new_h):
        for j in range(new_w):
            orig_x = min(int(i / scale_factor), img_h - 1)
            orig_y = min(int(j / scale_factor), img_w - 1)
            new_image[i, j] = image[orig_x, orig_y]
    return new_image
```

Listing 10: Nearest-Neighbor Zoom

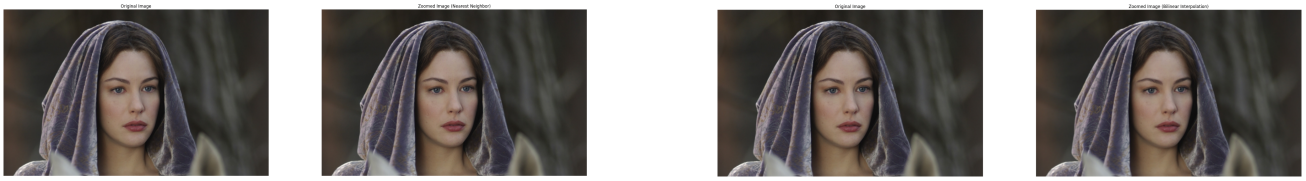
Bilinear Interpolation

```
def bilinear_zoom(image, scale_factor):
    img_h, img_w, img_c = image.shape
    new_h, new_w = int(img_h * scale_factor), int(img_w * scale_factor)
    new_image = np.zeros((new_h, new_w, img_c), dtype=image.dtype)

    x_scale = (img_w - 1) / (new_w - 1) if new_w > 1 else 0
    y_scale = (img_h - 1) / (new_h - 1) if new_h > 1 else 0

    for i in range(new_h):
```

```
        for j in range(new_w):
            orig_y, orig_x = i * y_scale, j * x_scale
            y1, x1 = int(np.floor(orig_y)), int(np.floor(orig_x))
            y2, x2 = min(y1 + 1, img_h - 1), min(x1 + 1, img_w - 1)
            dy, dx = orig_y - y1, orig_x - x1

            # Bilinear weights
            w1, w2, w3, w4 = (1-dx)*(1-dy), dx*(1-dy), (1-dx)*dy, dx*dy

            for c in range(img_c):
                new_image[i, j, c] = (w1 * image[y1, x1, c] + w2 * image[y1, x2, c] +
                                      w3 * image[y2, x1, c] + w4 * image[y2, x2, c])
    return np.clip(new_image, 0, 255).astype(image.dtype)
```

Listing 11: Bilinear Zoom

Method Differences: Nearest-neighbor uses simple pixel replication by mapping each new pixel to the closest original pixel through integer division, resulting in fast computation but blocky artifacts. Bilinear interpolation calculates fractional coordinates and performs weighted averaging of the four surrounding pixels based on distance, producing smoother results with higher computational cost.



(a) Nearest-neighbor (blocky) (b) Bilinear (smooth)

Figure 11: Zooming comparison. SSD: Nearest=182795356, Bilinear=179225113

Quantitative Comparison: The Sum of Squared Differences (SSD) analysis reveals significant quality differences: nearest-neighbor achieves SSD = 182,795,356 while bilinear interpolation achieves SSD = 179,225,113. The lower SSD value for bilinear interpolation (3.6 million difference) indicates closer resemblance to the original image in terms of pixel-wise accuracy. This quantitative improvement aligns with the visual assessment where bilinear interpolation produces smoother transitions and reduced aliasing artifacts, making it superior for applications requiring high-quality image enlargement despite increased computational complexity.

9 Question 9 - Background Blur with GrabCut

GrabCut segmentation isolates the foreground for selective background blurring.

```
# GrabCut segmentation
mask = np.zeros(img.shape[:2], np.uint8)
rect = (50, 30, img.shape[1]-50, img.shape[0]-50)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
cv.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

# Apply selective blur
foreground = img_rgb * mask2[:, :, np.newaxis]
blurred_bg = cv.GaussianBlur(img_rgb, (25, 25), 0)
enhanced_img = blurred_bg * (1 - mask2[:, :, np.newaxis]) + foreground
```

Listing 12: Background Blur with GrabCut

The code reads the image in BGR format and converts it to RGB. It initializes a GrabCut segmentation with a rectangular region around the flower, using five iterations to refine the mask, where values 0 and 2 indicate the background, and 1 and 3 indicate the foreground. The mask is binarized to separate foreground and background, which are multiplied with the original image to extract them. A Gaussian blur with a 25x25 kernel is applied to the entire image, and the enhanced image is created by combining the blurred background (where the mask is 0) with the original foreground (where the mask is 1).

Figure 12: Enhanced image with blurred background using GrabCut segmentation

The output image shows the daisy flower with sharp details in the foreground, while the background is noticeably blurred, creating a depth-of-field effect. The segmentation accurately outlines the flower, and the Gaussian blur effectively softens the background, enhancing the subject's prominence.