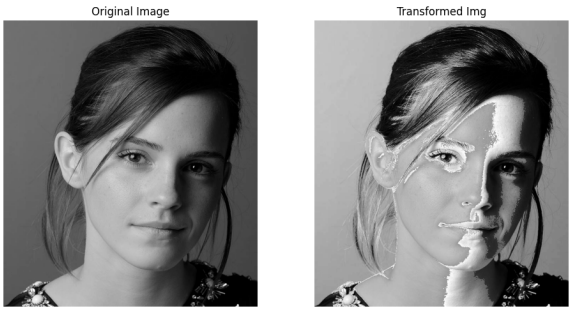
Assignment 1: Intensity Transformations and Neighborhood Filtering

Name: **T.P.N.S Peiris**
Index No.: **220454P**
Submission Date: **12th August 2025**
GitHub: Assignment 01

1.Intensity Transformation on Emma Watson

The Curve given in the question is mapped using the following equation.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.

```
def intensity_T(pix):
    if 50 <= pix <= 150:
        T_pix = 1.55 * pix + 22.5 #curve
        return T_pix
    else:
        return pix
```



2.White matter vs Grey matter

Detecting the intensities of grey matter pixels and white matter pixels

```
white_point = 110, 90
gray_point = 140, 120
```

Following image shows the representation of the points on the brain image.The intensity of the white point is 204.The intensity of the grey point is 171.

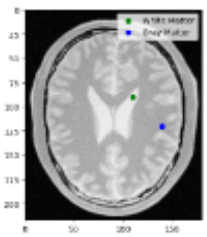


Figure 1: Intensity points of brain images

Generating separate masks

After this proper threshold values were selected to differentiate between grey matter and white matter separate masks were created.

```
# gray matter is brigher than white
gray_matter_mask = (brain_img >= 180) & ( brain_img<= 251)
transformed_image[gray_matter_mask] = 1.65 * brain_img[
    gray_matter_mask] + 25

# Apply transformation for gray matter (50 <= pixel <= 150) # white
    matter is darker than gray
white_matter_mask = (brain_img >= 140) & (brain_img <= 195)
transformed_image[white_matter_mask] = 1.40 * brain_img[
    white_matter_mask] + 26.5
```

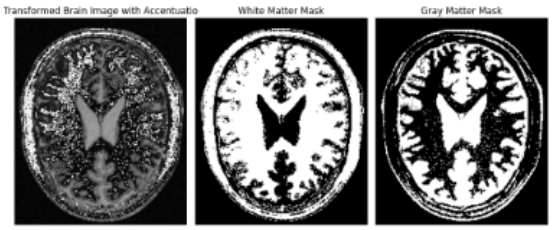


Figure 2: Accentuated image, white matter mask, gray matter mask

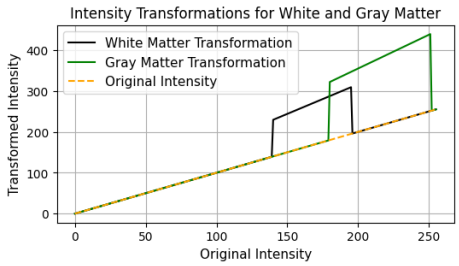


Figure 3: Intensity transformation plot

3.Gamma Correction

Gamma correction is a nonlinear operation used to encode and decode the brightness of images in a way that is more aligned with human perception. Human vision is not linear; we perceive changes in brightness differently depending on the light levels. As a result, gamma correction helps adjust the brightness of images to enhance visual quality, making them appear more natural.(Low gamma - Brightens darker images, High gamma - Darkens bright images)

```
def apply_gamma_correction(image_path, gamma):
"""
Apply gamma correction to the L* channel of an image in L*a*b*
    color space.
"""
# Load the image in RGB
image = cv2.imread(image_path)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert the image from RGB to L*a*b* color space
image_lab = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2LAB)
```

```
    # Split the LAB image into L, a, and b channels
    L, a, b = cv2.split(image_lab)

    # Apply gamma correction to the L channel
    L_normalized = L / 255.0  # Normalize L to [0, 1] range
    L_corrected = np.power(L_normalized, gamma)  # Apply gamma
        correction
    L_corrected = (L_corrected * 255).astype(np.uint8)  # Rescale to
         [0, 255]

    # Merge the corrected L channel back with the original a and b
        channels
    image_lab_corrected = cv2.merge((L_corrected, a, b))

    # Convert the corrected LAB image back to RGB color space
    corrected_image = cv2.cvtColor(image_lab_corrected, cv2.
        COLOR_LAB2RGB)

    return corrected_image, L, L_corrected
```

The following figure shows two gamma corrected images (for gamma = 0.5 and gamma = 2.2) compared with the original image. Below that is the respective histograms of the L channel.

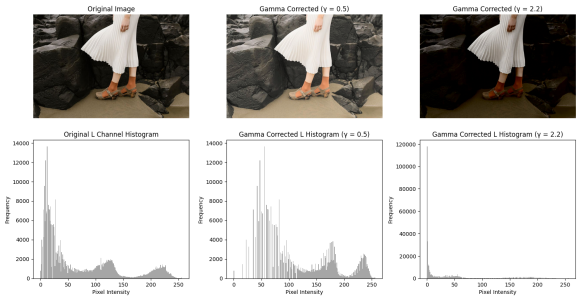


Figure 4: Gamma corrected images

4.Vibrancy

Splitting into HSV

There are different color spaces we use in when working with images.The HSV color space consist of three elements(Hue: Type of color, Saturation : Vibrancy of color, Value: The Brightness level.Let's compare the spider image and its HSV components.

Figure 5: Spider image HSV split

Applying Linear Tranformation

Let's apply this Linear transformation, f(x) = $\min\left(x + a \times 128e^{\frac{-(x-128)^2}{2\sigma^2}}, 255\right) when \sigma = 70 and a is between 0 to 1.$

```
    # Apply the intensity transformation to the sat plane
a_vals = [0.2, 0.4, 0.6, 0.8]  # Different vals of 'a'
sigma = 70

# Apply the transformation for each val of 'a'
vibrance_images = []
for a in a_vals:
    transformed_sat = vibrancify(sat, a, sigma)

    # Recombine the HSV planes back into one image
    transformed_hsv = cv2.merge([hue, transformed_sat, val])
    transformed_image_rgb = cv2.cvtColor(transformed_hsv, cv2.
        COLOR_HSV2RGB)
    vibrance_images.append(transformed_image_rgb)
```

Figure 6: Vibrancy with different a values

6.Histogram Equalization on the foreground of an image

When taking a photo it is always better to achieve a flat histogram as they are more vibrant. In this task it is tried to take the following image of and apply a histogram equalization only for the foreground.

Splitting into HSV components

```
hue, saturation, value = cv2.split(image_hsv)
```

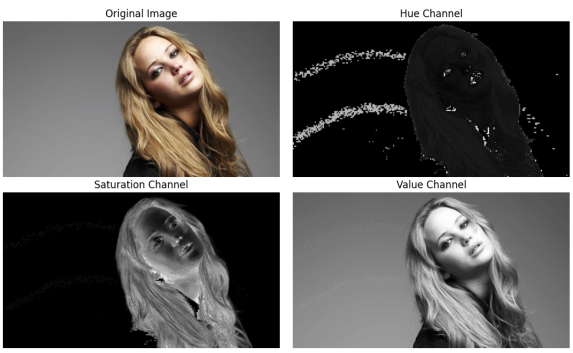


Figure 7: HSV split of image

Thresholding to extract the foreground

I used otsu's threholding method on the saturation channel of image which returns a single intensity threshold that separate pixels into two classes, foreground and background. This algorithm exhaustively searches for the threshold that minimizes the intra-class variance, defined as a weighted sum of variances of the two classes. But it didnt perform well to threshold this image. So in here this is simply a direct thresholding if image plane of

```
foreground_mask = saturation > 20
```

Then I applied a bitwise and operation on the original image with the foreground mask to extract the foreground of the image.

```
foreground = cv2.bitwise_and(image_rgb , image_rgb , mask=foreground_mask.astype(np.uint8))
```

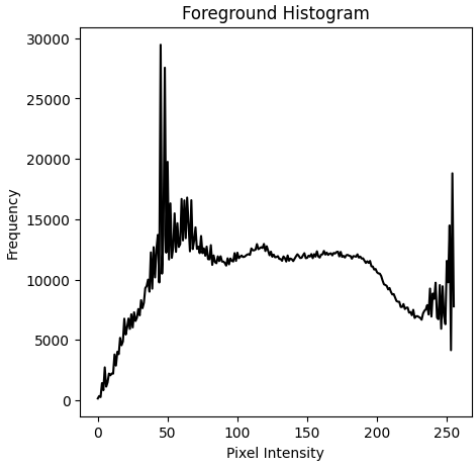


Figure 8: Foreground histogram

Cumulative sum of the histogram

```
cumsum_hist_foreground = np.cumsum(hist_foreground)
```

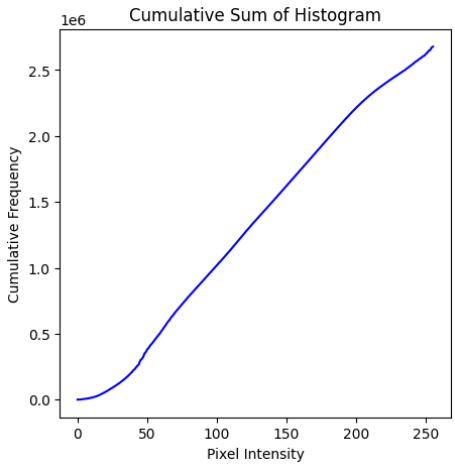


Figure 9: cumulative sum of the histogram

```
    #  Obtain cumulative sum of histograms
cum_hist_v = np.cumsum(hist_v)

pixel_count = foreground_mask.astype(np.uint8).sum()

# Look up tables for each color channel
lut_v = equalize_histogram(cum_hist_v, pixel_count)


# Apply equalization to foreground
equalized_foreground_val = np.zeros_like(foreground_val)
equalized_foreground_val = cv2.LUT(foreground_val, lut_v)



# Merge the equalized value plane with the original hue and
    saturation planes
equalized_hsv_foreground = cv2.merge((hue, sat_plane,
    equalized_foreground_val))

# Convert the image back to BGR color space
equalized_background_foreground = cv2.cvtColor(
    equalized_hsv_foreground, cv2.COLOR_HSV2BGR)


# Note: No color space conversion is needed here as both images are
    in BGR
final_equalized = cv2.add(equalized_background_foreground,
    background)
```

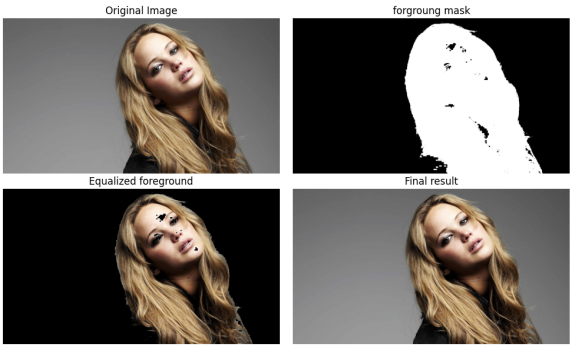


Figure 10: Equalized Foreground merged with the background

7.Filtering with Sobel Operator

Sobel filter using existing filter2D

```
    sobel_kernel_x = np.array([[-1, 0, 1],
                               [-2, 0, 2],
                               [-1, 0, 1]])

# Apply the Sobel filter using filter2D function
sobel_x = cv2.filter2D(albert_img, -1, sobel_kernel_x)
```

Here I used a separable sobel filters for the X direction and Y direction and later got the magnitude of the gradients to get the final output.

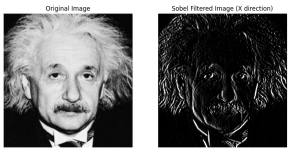
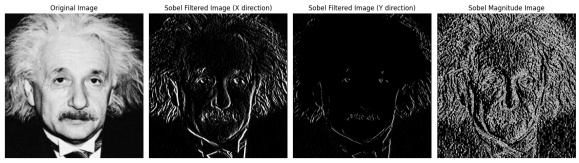


Figure 11: Sobel filter with filter2D

```
    # Apply separable Sobel filters for the X direction
sobel_x_step1 = apply_1d_filter(albert_img, sobel_1d_x, 'x')  #
    Horizontal filter
sobel_x_final = apply_1d_filter(sobel_x_step1, sobel_1d_y, 'y')  #
    Vertical filter

# Apply separable Sobel filters for the Y direction
sobel_y_step1 = apply_1d_filter(albert_img, sobel_1d_y, 'y')  #
    Vertical filter
sobel_y_final = apply_1d_filter(sobel_y_step1, sobel_1d_x, 'x')  #
    Horizontal filter

# Calculate the magnitude of the gradients
sobel_magnitude = np.sqrt(sobel_x_final ** 2 + sobel_y_final ** 2)
sobel_magnitude = np.clip(sobel_magnitude, 0, 255).astype(np.uint8)
```



8.Zooming with nearest neighbor zooming and bilinear Zooming

The results after the SSD computing are Nearest-Neighbor SSD:136.269 and Bilinear SSD: 115.0918

Figure 12: Image Zooming effects

Nearest neighbor interpolation often creates a pixelated look because it simply copies the value of the closest pixel without blending with others. In contrast, bi linear interpolation tends to produce a slight blur, as it averages the values of the four nearest pixels to create smoother transitions. If speed is a priority or maintaining exact pixel values is important, like with retro graphics, nearest neighbor is a good choice. However, if we are looking for better visual quality, especially when enlarging images, bi linear interpolation is usually preferred for its smoother, more natural look.

9.Image Enhancement

```
    # Define a rectangle around the foreground object (flower)
bounding_box = (50, 50, flower_image.shape[1] - 100, flower_image.
    shape[0] - 100)
# Apply the GrabCut algorithm to segment the image
cv2.grabCut(flower_image, segmentation_mask, bounding_box,
    background_model, foreground_model, 5, cv2.GC_INIT_WITH_RECT)

# Adjust mask so that probable background/foreground pixels are
    handled
adjusted_mask = np.where((segmentation_mask == 2) | (
    segmentation_mask == 0), 0, 1).astype('uint8')

# Extract the foreground (flower) using the adjusted mask
segmented_foreground = flower_rgb * adjusted_mask[:, :, np.newaxis]

# Extract the background by inverting the mask
segmented_background = flower_rgb * (1 - adjusted_mask[:, :, np.
    newaxis])
```

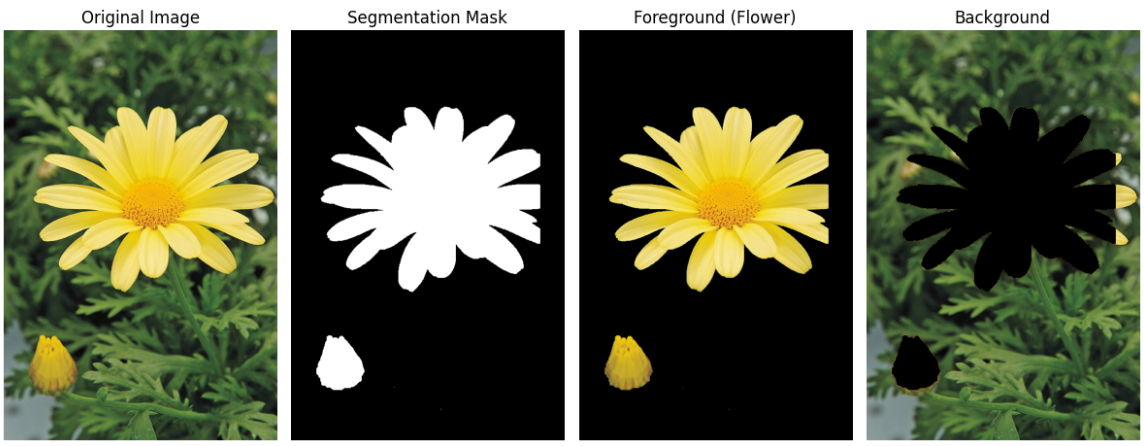


Figure 13: Grab cut image segmentations

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.