Assignment 01

Intensity Transformations and Neighborhood Filtering

submitted for

EN3160 - Image Processing and Machine Vision

Department of Electronic and Telecommunication Engineering
University of Moratuwa

Udugamasooriya P. H. J.
220658U

12 August 2025

1. The given transformation, which we denote $y = T(x)$ is exactly the identity, except for $x \in [50, 150]$, where

$$\frac{y - 100}{x - 50} = \frac{255 - 100}{150 - 50}$$
$$y = 1.55x + 22.5.$$

   Applying this transformation to the image yields the following output;

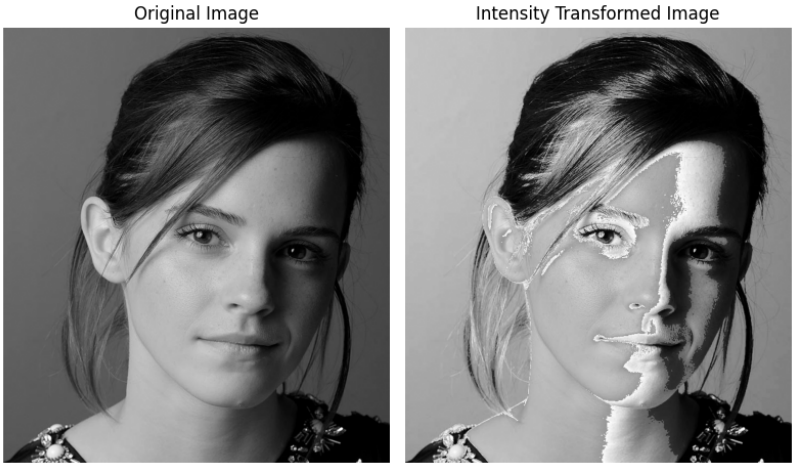

Figure 1: Question 1

2. An eyedropper tool was used to inspect the values of randomly chosen pixels from both the graymatter and whitematter regions, and observed that the graymatter is composed mostly of pixels with intensities less than around 175. To emphasize these pixels, we implement the following transformation;

$$T(x) = \begin{cases} x \ // \ 6, & x \leq 175, \\ x, & \text{otherwise}, \end{cases}$$

   where // denotes the integer-division operation.

   A plot of the intensity transformation, and the result of applying it to the original image are shown below;

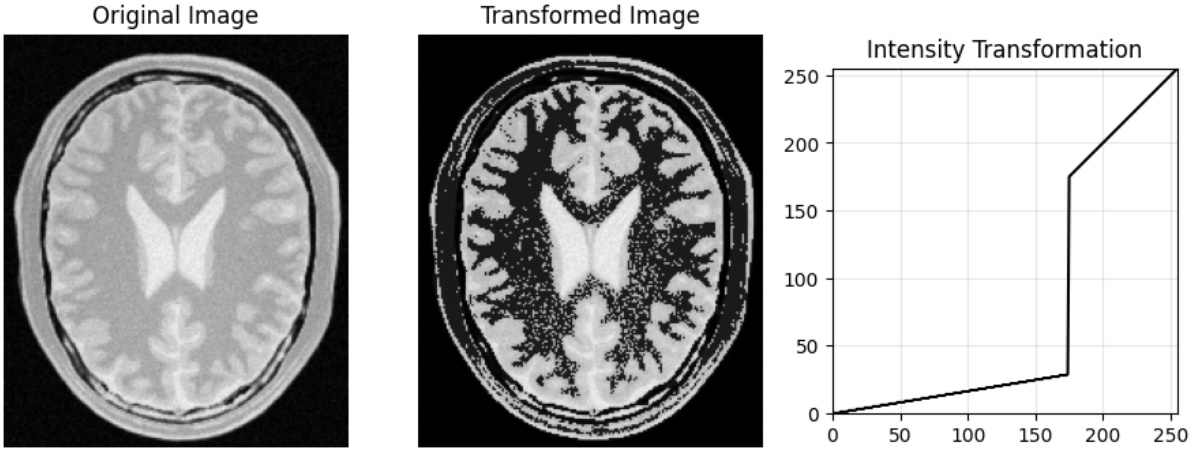

Figure 2: Question 2

analysis?

3. We apply the gamma transformation specified by

$$T(x) = 255 \cdot \left(\frac{x}{255}\right)^{\gamma}$$

to the L-channel of the image. The most aesthetically pleasing result was obtained by setting $\gamma = 0.5$. The results are given below.

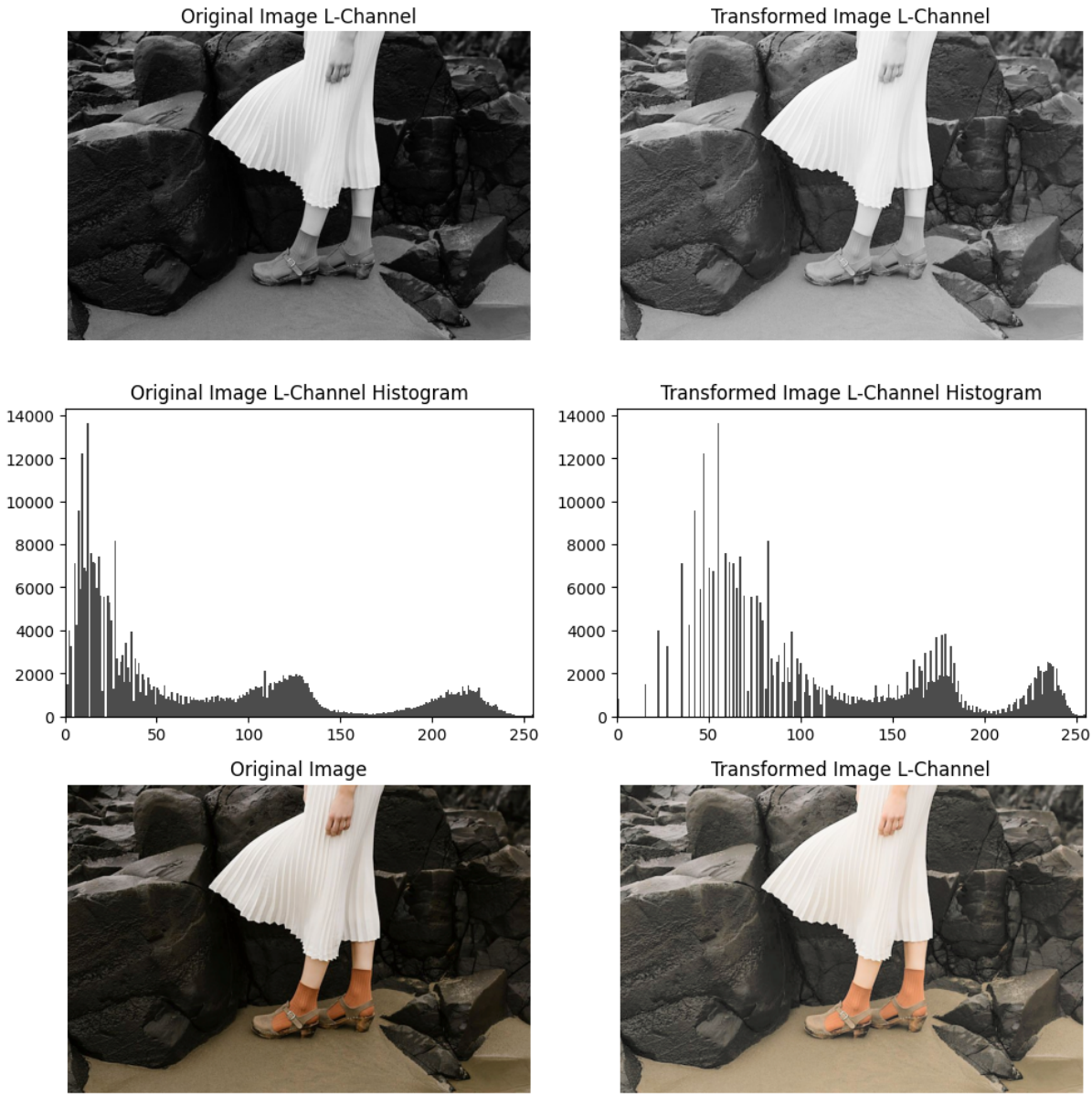

Figure 3: Question 3

4. The line

```
T = np.minimum(np.ones(256) * 255, T + a * 128 * np.exp(-((T - 128) ** 2) / (2 * sigma **
    2)))
```

implements the intensity transformation; we use `np.minimum` to pick the smaller number from one vector that is all 255's, and another populated with entries computed as per the expression given.

The most aesthetically pleasing result was obtained by setting $a = 0.65$.

The results are as follows;

Figure 4: Question 4

5. Let X be a random variable denoting the value of a pixel randomly chosen from the image. The p.m.f. of X then, is exactly the normalized histogram of the image. Our goal is to find a transformation

$$T : [0, 255] \to [0, 255],$$

such that the p.m.f. of the random variable $Y = T(X)$ is uniform, i.e., equal to $\frac{1}{255}$ over all possible values of Y. To proceed, we will assume that the T we seek is bijective.

Let $y \in [0, 255]$, i.e., let y be one of the values that Y might assume. Then, because the p.m.f. of Y must be uniform, we must have

$$\begin{aligned} \Pr(Y \leq y) &= \frac{y}{255} \\ \Pr(T(X) \leq y) &= \frac{y}{255} \\ \Pr\left(X \leq T^{-1}(y)\right) &= \frac{y}{255}. \end{aligned}$$

Suppose now that x is a value that X may take such that $T^{-1}(y) = x$, or equivalently, $y = T(x)$. In this case, $\Pr(X \leq x)$ and $\Pr\left(X \leq T^{-1}(y)\right)$ both denote the same quantity; so we can write

$$\Pr(X \leq x) = \frac{T(x)}{255}.$$

But, $\Pr(X \leq x)$ is just a running sum of the p.m.f. of X, which is known–denoting the number of pixels having intensity level i by n_i, we have

$$\frac{T(x)}{255} = \Pr(X \leq x) = \frac{1}{LM} \sum_{j=1}^{x} n_j,$$

where L and M give the dimensions of the image in pixels.

This gives the transformation we seek;

$$T(x) = \frac{255}{LM} \sum_{j=1}^{x} n_j.$$

We implement this transformation in the following lines of code;

```
def equalize_histogram(img):
    pixel_values, counts = np.unique(img, return_counts=True)
    cumulative_counts = np.cumsum(counts)

    T = np.arange(256, dtype=np.uint8)

        for i in range(len(pixel_values)):
        pixel_value = pixel_values[i]
        T[pixel_value] = np.round((cumulative_counts[i] / cumulative_counts[-1]) * 255)

    return T[img]
```

We use `np.unique` to extract the distinct pixel values present in the image, and use `np.cumsum` to form a running sum of of pixel counts from each intensity level. We construct a transformation that leaves pixels intensities absent in the image unchanged, but maps those present, to the value computed as described above.

The result of performing histogram equalization to an image is shown below.

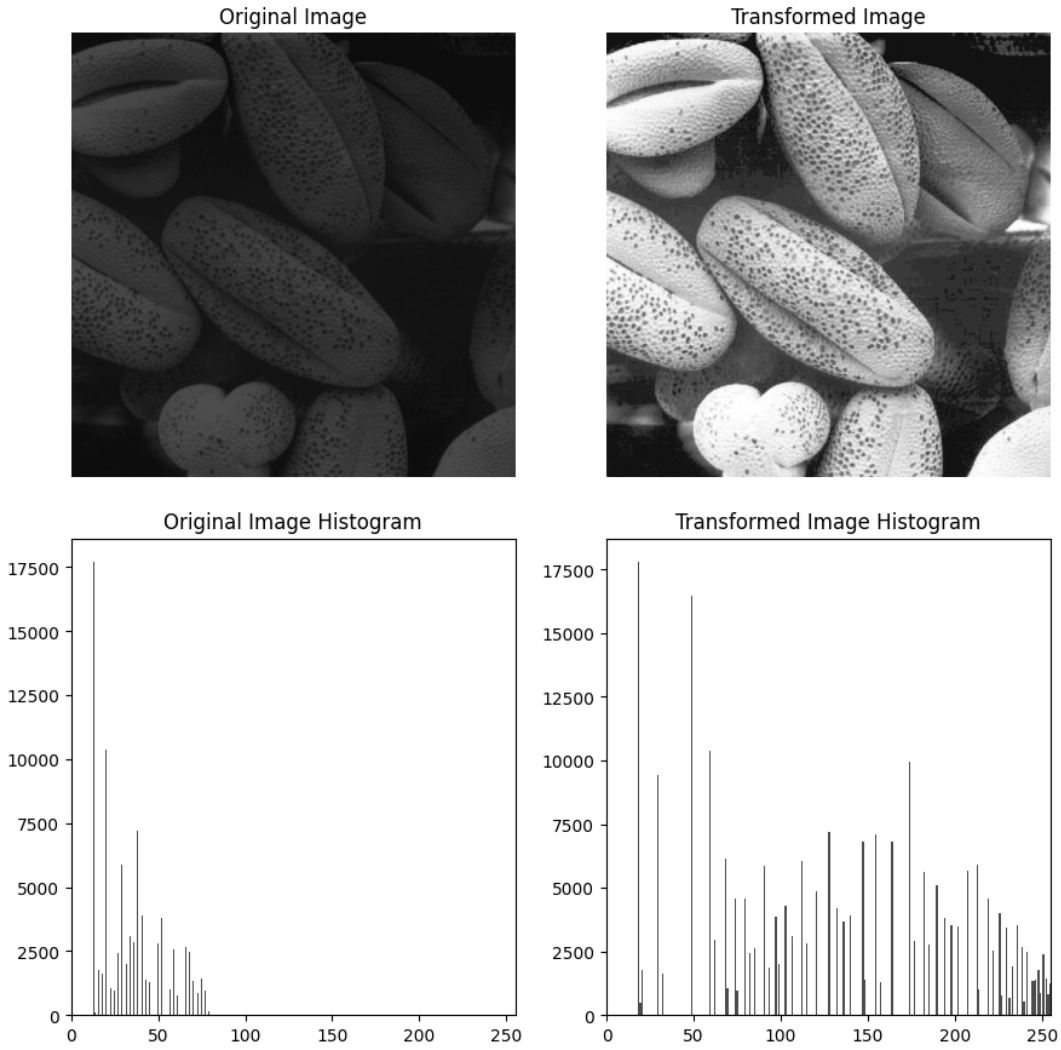

Figure 5: Question 5

6. The background of the given image is predominantly composed of light colors, so we expect the pixel values corresponding to the background in the S channel of the image to be mostly small values. Hence, we apply a threshold in the S channel using `cv2.threshold`, to yield a binary image, i.e., the foreground mask.

   We then extract the foreground from the S and V channels by performing a `cv2.bitwise_and` of each channel with the mask. A `cv2.bitwise_not` of the foreground mask yields a background mask, which we also `cv2.bitwise_and` with each channel and obtain the background from.

   We then implement the `equalize_histogram` function, which is very similar to the function implemented above, except for some slight modifications in the line

   ```
   T[pixel_value] = np.round(((cumulative_counts[i] - cumulative_counts[0]) / (
       cumulative_counts[-1] - cumulative_counts[0]) ) * 255),
   ```

   made to ignore the zero-valued pixels–because the zero-valued pixels correspond to background pixels, and the probability that a pixel of a certain value be found in the foreground must be found without considering them.

   We use this function to equalize the histograms of the foregrounds in the S and V channels, but leave the H channel unchanged, as it contains color-related information which must be preserved. Finally, we combine the histogram-equalized foregrounds from the S and V channels with the H channel of the original image, and construct the final image.

The results are as follows.

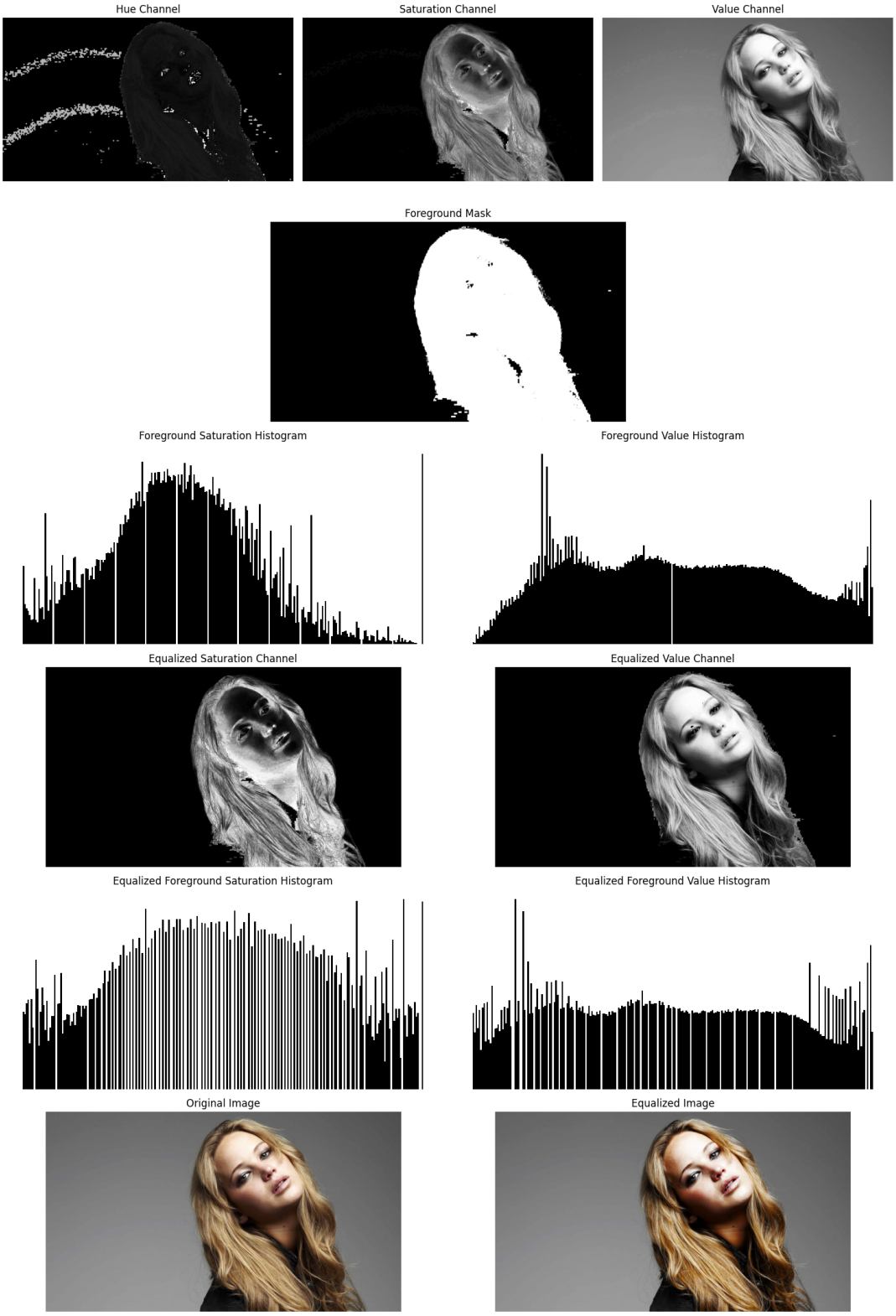

Figure 6: Question 6

7. We implement the process of applying the 2D kernel manually using for loops in the function `my_filter2D`. Another function `my_filter1D` is implemented to apply a 1D kernel to an image. The Sobel filter given is then applied to the image using both `cv2.filter2D` and `my_filter2D`.

   To see how the same result can be achieved by convolving the image with two 1D kernels in cascade, denote by $\bar{c}$ the matrix obtained by rotating the matrix c by 180° around its center. Further, denote the operation of

convolution by $*$ and correlation by $\star$.

Let a denote an image, and b and c be two 1D matrices such that we wish to apply the kernel $b * c$ to the image a. The operation of applying this kernel is $(b * c) \star a$. But, by the properties of convolution,

$$(b * c) \star a = (b * c) * \bar{a} = b * (c * \bar{a}) = b * (c \star a) = (c \star a) * b = (c \star a) \star \bar{b}.$$

8. Both forms of interpolation can be implemented in the form of a multiplication of the image on either side by two matrices L and R. The structures of these matrices for nearest-neighbor and bilinear interpolation respectively, for a scaling factor of 4, are given by

$$L_{\text{nn}} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0.5 & 0.5 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0.5 & 0.5 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}, \quad \text{and,} \quad L_{\text{bl}} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0.75 & 0.25 & 0 & \cdots & 0 \\ 0.5 & 0.5 & 0 & \cdots & 0 \\ 0.25 & 0.75 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0.75 & 0.25 & \cdots & 0 \\ 0 & 0.5 & 0.5 & \cdots & 0 \\ 0 & 0.25 & 0.75 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0.75 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix},$$

with R_{nn} and R_{bl} following a similar structure but along the other direction.

The method works because the positions of the non-zero elements have the effect of extracting the pixel in the original image corresponding to the particular row/column, and placing it in a new row/column, corresponding to the row/column that the entry belongs to. s The above algorithms were run on three images; `im01small.png`, `im02small.png`, and `taylor_very_small.jpg`, and the normalized sum of squared errors found with the actual images. The results obtained were as follows;

```
im01small.png - Normalized SSD from
Nearest-Neighbor Zoom: 39.34536040380659
Bilinear Interpolation Zoom: 39.54175524048354

im02small.png - Normalized SSD from
Nearest-Neighbor Zoom: 16.775474247685185
Bilinear Interpolation Zoom: 16.75797627314815

taylor_very_small.jpg - Normalized SSD from
Nearest-Neighbor Zoom: 49.654806547619046
Bilinear Interpolation Zoom: 50.564375
```

Based on the results, both algorithms have performed almost equally well. The smaller the initial image, the worse is the error, as seen in the case of `taylor_very_small.jpg`.

9. We use `cv2.grabCut` as specified in the documentation in the following lines of code;

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

rect = (35, 133, 521, 424)

bg_model = np.zeros((1, 65), np.float64)
fg_model = np.zeros((1, 65), np.float64)

cv2.grabCut(img, mask, rect, bg_model, fg_model, 5, cv2.GC_INIT_WITH_RECT)
```

The resulting mask is then used to extract the foreground and background, and a Gaussian blur with a kernel size of 31 and standard deviation 5 $\left(< \frac{31}{2\times 3}\right)$ is applied to the background. The blurred background is once again masked with the background mask to zero out the pixels in the foreground region, and added with the foreground to produce the final image.

The reason for the edges around for the foreground of the enhanced image to appear dark is the "bleeding" of the black pixels in the foreground region of the masked-background image due to the blurring.

The results of the exercise are as follows;

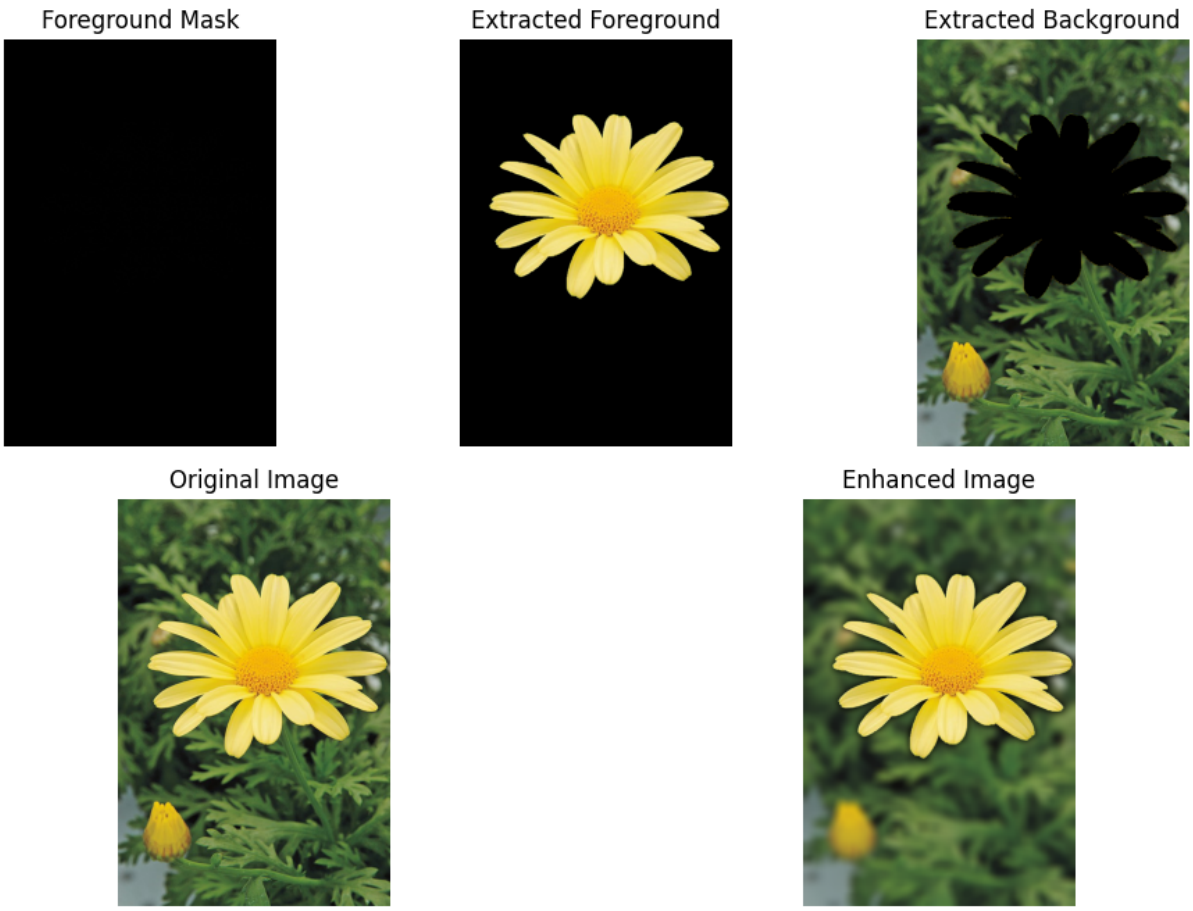

Figure 7: Question 9