EN3160: Image Processing and Machine Vision

Assignment 1 on Intensity Transformations and Neighborhood Filtering

Module:	EN3160
Date of Submission:	2024.09.30
GitHub Repository:	Warren-SJ/Image-Processing-Exercises

1. The given intensity transformation was implanted using the following code:

```
discontinuities = np.array([(50, 50), (50, 100),
(150, 255), (150, 150), (255, 255)])

# Segment-wise interpolation
t1 = np.linspace(0, discontinuities[0,1],
discontinuities[0,0] + 1 - 0).astype('uint8')
t2 = np.linspace(discontinuities[0,1], discontinuities[1,1],
discontinuities[1,0] - discontinuities[0,0] + 1).astype('uint8')
t3 = np.linspace(discontinuities[1,1], discontinuities[2,1],
discontinuities[2,0] - discontinuities[1,0] + 1).astype('uint8')
t4 = np.linspace(discontinuities[2,1], discontinuities[3,1],
discontinuities[3,0] - discontinuities[2,0] + 1).astype('uint8')
t5 = np.linspace(discontinuities[3,1], discontinuities[4,1],
discontinuities[4,0] - discontinuities[3,0] + 1).astype('uint8')

# Concatenate all the segments
transform = np.concatenate([t1, t2[1:], t3[1:], t4[1:], t5[1:]])
```

The transform was applied to the image using openCV's LUT.

```
emma_watson = cv.imread(ASSET_FOLDER + 'emma.jpg', cv.IMREAD_GRAYSCALE)
emma_watson_transformed = cv.LUT(emma_watson, transform)
```

Result is shown in figure 1

Figure 1: Result of Transformation

2. a. The white matter is brighter than the surrounding areas. Therefore, in order to enhance these areas, more detail can be brought out by increasing contrast in the brighter areas of the image.

   b. When considering the greymatter, it is darker than the white matter. Therefore, in order to enhance these areas, more detail can be brought out by increasing contrast in the medium intensity areas of the image.

   Both transformations are shown in figure 2 The results of these transformations are shown in figure 3

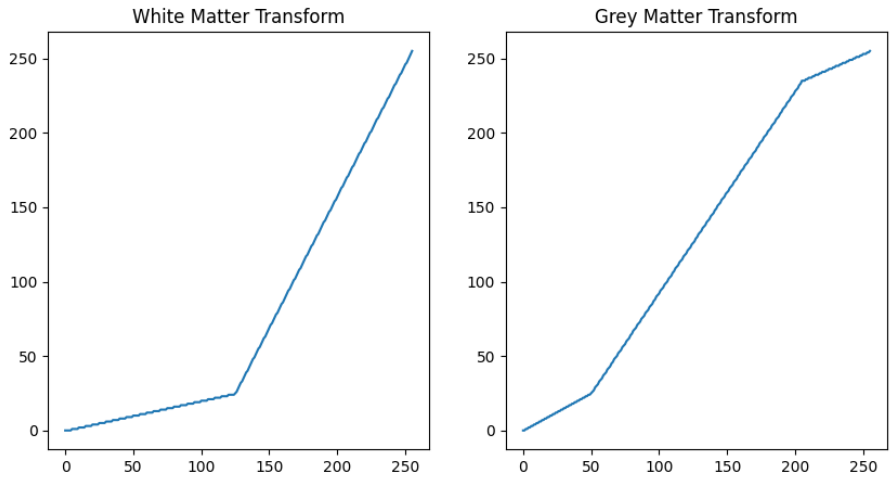

Figure 2: White matter and Gray matter transformations

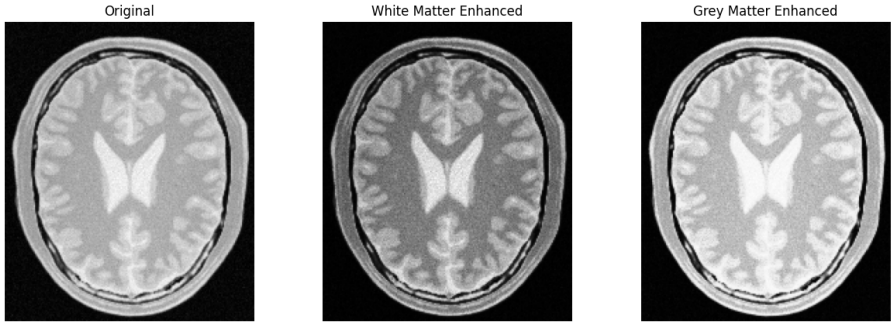

Figure 3: All 3 images

3. a. In order to obtain a suitable γ value, the histogram of the L channel will be plotted to determine a value for γ. Based on the histogram, since most values of the L channel are concentrated in the lower range, we can apply gamma correction to enhance the image such that gamma is less than 1. Upon transforming and plotting the new histogram, a γ value of **0.7** was chosen.

   b. The histograms were plotted using the following code:

```
l, a, b = cv.split(lab_image)
plt.hist(l.ravel(), bins=256, range=[0, 256])
plt.show()
```

   The old and new histograms are shown in figure 4

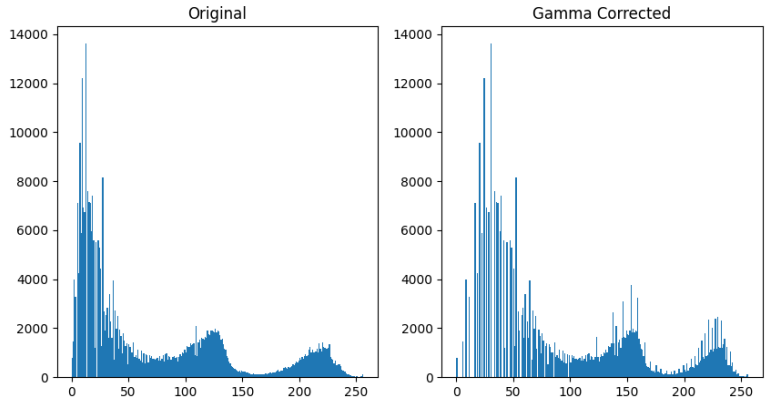

Figure 4: Old and new histograms

4. In order to enhance the vibrance of the image, the following approach as indicated by the code was performed.

   a. The image was split using

   ```
   h,s,v = cv.split(cv.cvtColor(vibrance_image, cv.COLOR_BGR2HSV))
   ```

   b. OpenCV's lookup table was used to apply the intensity transformation

   ```
   vibrance_adjustment = np.array([min(255, x + a_values[i] * 128 *
   np.exp(-(x - 128)**2 / (2 * 70**2)))
   for x in np.arange(0, 256)],dtype=np.uint8)
   s_vibrance = cv.LUT(s, vibrance_adjustment)
   ```

   c. Different values of a was tested using a loop. The transformation increases the saturation of the image and is preferred for the given image as it looks desaturated. For small a values the output has less increase in saturation while there is a large saturation when a is 1. Based on the observed images, $a = 0.4$ seems to be a good choice as it has increased the saturation while not making the image look overly saturated as is the case in larger values of a. Note that this choice is highly subjective and additionally depends on the display of the device used to view the image.

   d. This was done using

   ```
   adjusted_image = cv.merge([h, s_vibrance, v])
   ```

   e. The original image, transformation and vibrance-enhanced image are shown in figure 5

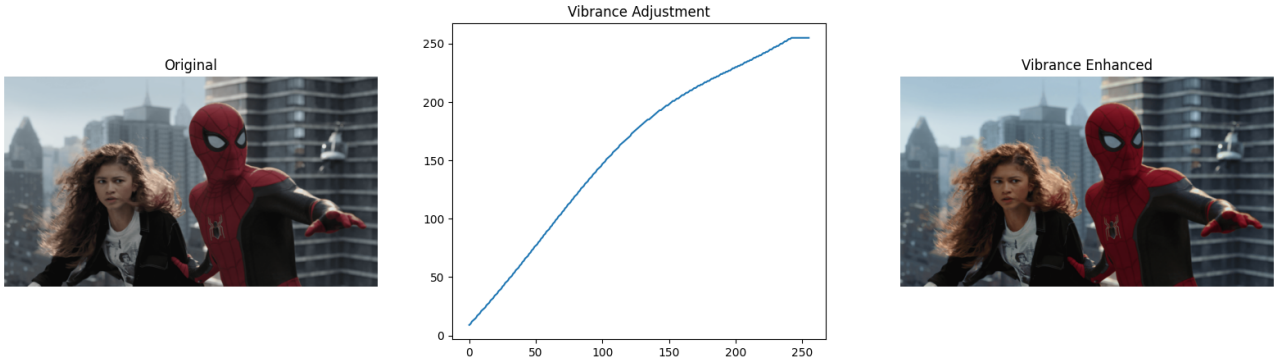

Figure 5: Original image, transformation and vibrance enhanced image

5. The following function was used to perform histogram equalization. The output is shown in figure 6

```
def histogram_equalization(image):
    M, N = image.shape
    h = cv.calcHist([image], [0], None, [256], [0, 256]).flatten()
    cdf = np.cumsum(h)
    cdf_normalized = cdf * (255 / cdf[-1])
    t = np.uint8(cdf_normalized)
    g = t[image]
    return g
```

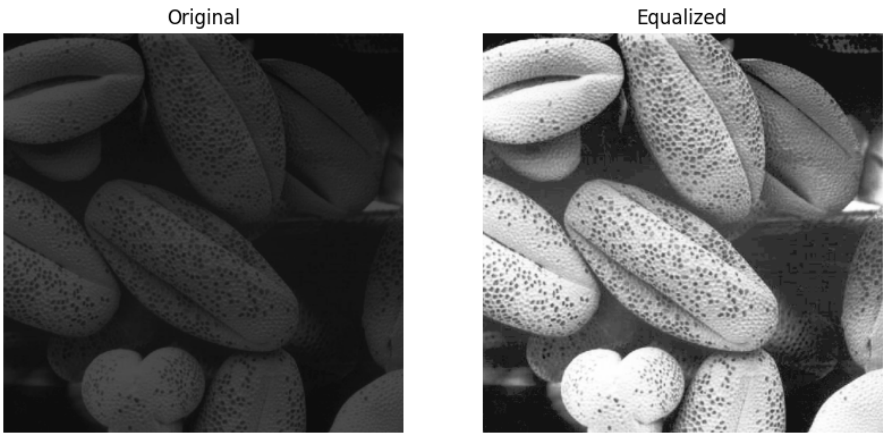

Figure 6: Histogram Equalization

6. a. The image was split into h,s and v planes using openCV's builtin *split* method which was applied on the HSV image. The images shown in figure 7 was obtained.

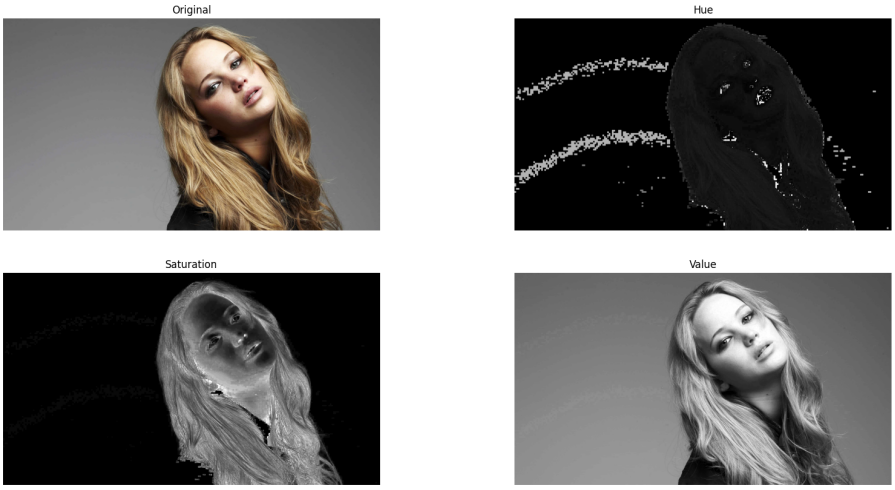

Figure 7: H,S,V Channels

b. Based on the 3 channels, the Saturation channel seems best to obtain a mask and extract the foreground.

c. A threshold was applied to get a binary mask and this was applied using the following code:

```
_, old_binary_mask = cv.threshold(s, 11, 255, cv.THRESH_BINARY)
foreground = cv.bitwise_and(jeniffer, jeniffer, mask=binary_mask)
```

However, applied just like this, parts of the dress and face are also identified as the background. It wasn't possible to tune the thresholding to get a perfect foreground separation. Therefore a morphological closing operation was done on the mask as follows:

```
binary_mask = cv.morphologyEx(old_binary_mask, cv.MORPH_CLOSE,
cv.getStructuringElement(cv.MORPH_ELLIPSE, (65, 65)))
```

The result of this is shown in figure 8

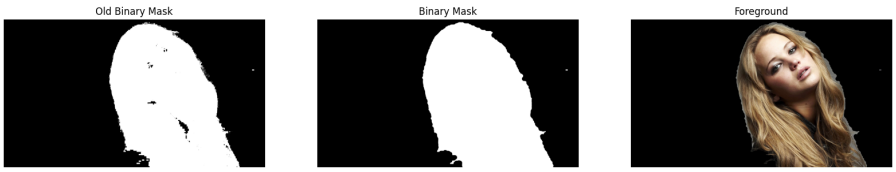

Figure 8: Result of morphological closing

d. This was done using the following code. Additionally, it was normalized.

```
original_hist=cv.calcHist([v_original],[0],None,
[256],[1,256]).flatten()
cdf = np.cumsum(original_hist)
cdf_normalized = cdf * (255 / cdf[-1])
```

e. Equalization is applied only to the foreground's v challen since intensity information is in the v-channel. This was done using the following:

```
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
equalized_image = cv.cvtColor(hsv_foreground, cv.COLOR_HSV2BGR)
output_image = jeniffer.copy()
output_image[binary_mask == 255] = equalized_image[binary_mask == 255]
```

The output is shown in figure 9. The histograms for the foreground are shown in

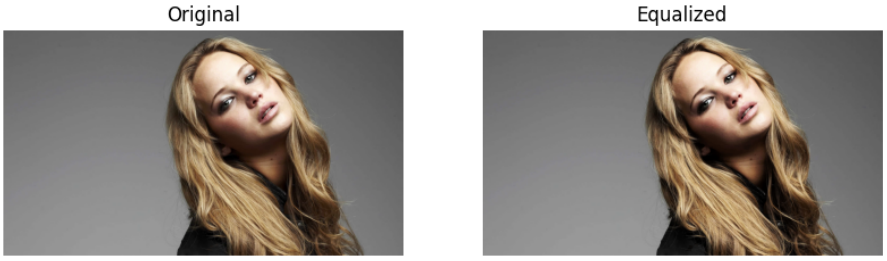

Figure 9: Original vs. Foreground Equalized Image

figure 10

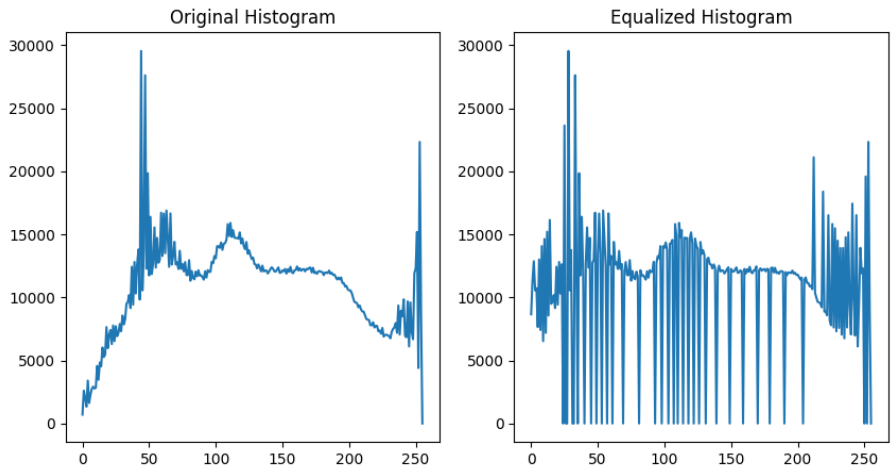

Figure 10: Original vs. Foreground Equalized Image Histograms

7. (a) The existing *filter2d* function was applied as follows:

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

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

sobel_einstein_x = cv.filter2D(einstein, cv.CV_64F, sobel_x)
sobel_einstein_y = cv.filter2D(einstein, cv.CV_64F, sobel_y)
```

(b) In order to do this, first, the image and kernel we cast to *float64* since image overflows may occur. The output ranges from -255 to 255 and therefore, it had to be clipped and normalized between $[0, 255]$

```
M, N = einstein.shape
einstein = einstein.astype(np.float64)
sobel_einstein_x = np.zeros(shape=(M, N), dtype=np.float64)
sobel_einstein_y = np.zeros(shape=(M, N), dtype=np.float64)

# Custom Sobel filtering
for i in range(1, M-1):
    for j in range(1, N-1):
        sobel_einstein_x[i,j] = einstein[i-1,j-1] - einstein[i-1,j+1]
        + 2*einstein[i,j-1] - 2*einstein[i,j+1] + einstein[i+1,j-1] -
        einstein[i+1, j+1]
        sobel_einstein_y[i,j] = einstein[i-1,j-1] + 2*einstein[i-1,j]
        + einstein[i-1,j+1] - einstein[i+1,j-1] - 2*einstein[i+1,j] -
        einstein[i+1,j+1]

sobel_einstein_x = np.clip(sobel_einstein_x, -255, 255)
sobel_einstein_y = np.clip(sobel_einstein_y, -255, 255)
sobel_einstein_x_normalized = np.uint8(255 *
(sobel_einstein_x + 255) / 510)
sobel_einstein_y_normalized = np.uint8(255 *
(sobel_einstein_y + 255) / 510)
```

(c) This is a more efficient way of applying a separable filter. A filter is separable if it's rank is 1. The following code was used to obtain the filtered image.

```
sobel_1 = np.array([1,2,1])
sobel_2 = np.array([1,0,-1])
sobel_einstein_x = cv.sepFilter2D(einstein, -1, sobel_2, sobel_1)
sobel_einstein_y = cv.sepFilter2D(einstein, -1, sobel_1, sobel_2)
```

The outputs are shown in figure 11. It is noticed that all methods produced similar results. The custom operations are slightly different from the using *filter2D* and *sepFilter2D*. This is likely because openCV's implementation applies a smoothing filter

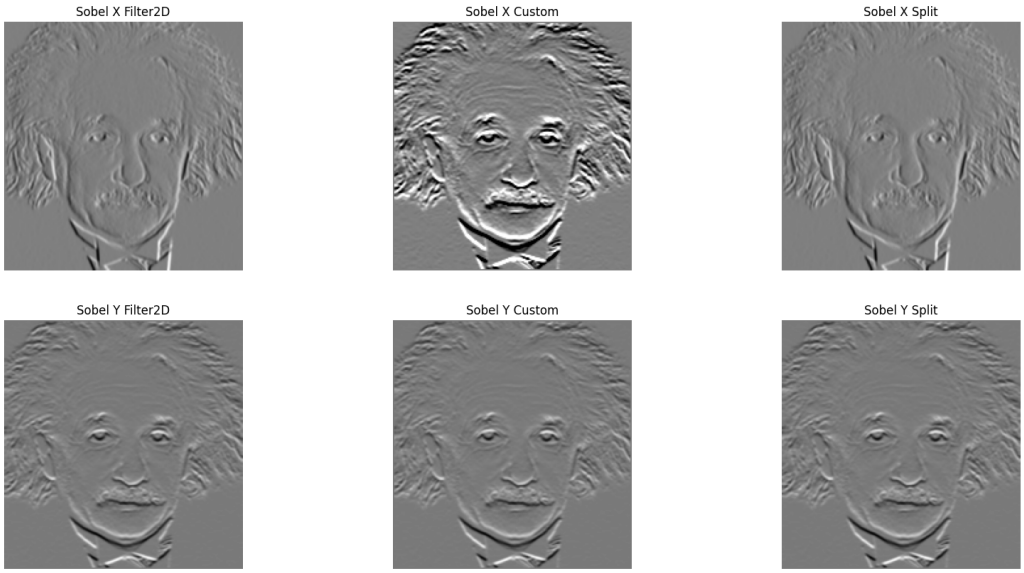

Figure 11: Comparison of Sobel Filtering

8. The functions were coded to handle both RGB and grayscale images. These functions were custom built and are included within the code. The Sum of Squared Differences for the images calculated as:

```
diff = (image1.astype(np.float64) - image2.astype(np.float64)) ** 2
```

are as follows:

A notable fact from these results is that bilinear interpolation produced a closer result

Image	Nearest Neighbour	Bilinear Interpolation
im01.png	136.27	115.09
im02.png	26.45	18.35

Table 1: SSD values

to the original high-resolution image than nearest neighbour. This is because nearest neighbour doesn't consider the surrounding pixels while bilinear interpolation considered the neighbours to produce a smoother output. The results are shown in figure 12

Figure 12: Image Upscaling

9. (a) The following code was used to generate the segmentation mask, foreground image, and background image.

```
mask = np.zeros(flower.shape[:2], np.uint8)
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)
rect = (50, 50, flower.shape[1] - 50, flower.shape[0] - 50)
cv.grabCut(flower, mask, rect, bgd_model, fgd_model, 5,
cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
foreground = flower * mask2[:, :, np.newaxis]
background = flower * (1 - mask2)[:, :, np.newaxis]
```

The mask, foreground and background are shown in figure 13

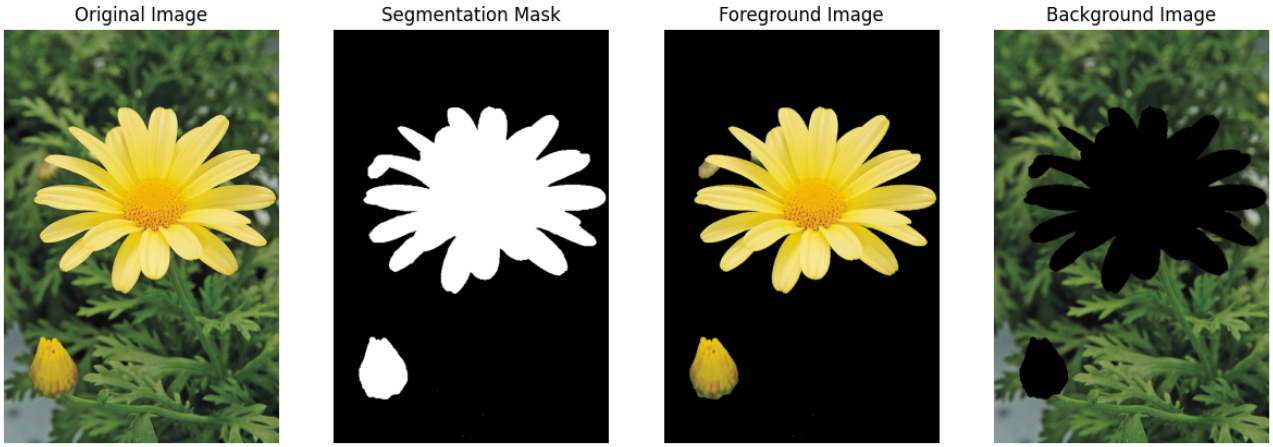

Figure 13: segmentation mask, foreground image, and background

(b) The image in figure 14 was obtained. This was done using

```
blurred_background = cv.GaussianBlur(background, (15, 15), 0)
enhanced_image = blurred_background + foreground
```

Figure 14: Background Blur

(c) When the mask is created, some pixels around the edges of the foreground object might be classified as probable background (label 2) or definite background (label 0). These pixels are set to zero in the mask, which results in dark pixels when the background is blurred and combined with the foreground.