Department of Electronic and Telecommunications Engineering

EN3160 - Image Processing and Machine Vision

Assignment 01

Intensity Transformations and Neighborhood Filtering

Rajapaksha N.N. - 210504L

GitHub profile: github.com/nadunnr
Assignment 01: nadunnr/../Assignment 01

Q1: Intensity windowing

Intensity transformation was implemented in Python and applied to the original image.

```
t1 = np.linspace(0,50,51, dtype=np.uint8)
t2 = np.linspace(100,255,150-50, dtype=np.uint8)
t3  = np.linspace(150,255, 255-150, dtype=np.uint8)
intensity_window = np.concatenate((t1,t2,t3), axis=0)
transformed_emma = cv.LUT(emma, intensity_window)
```

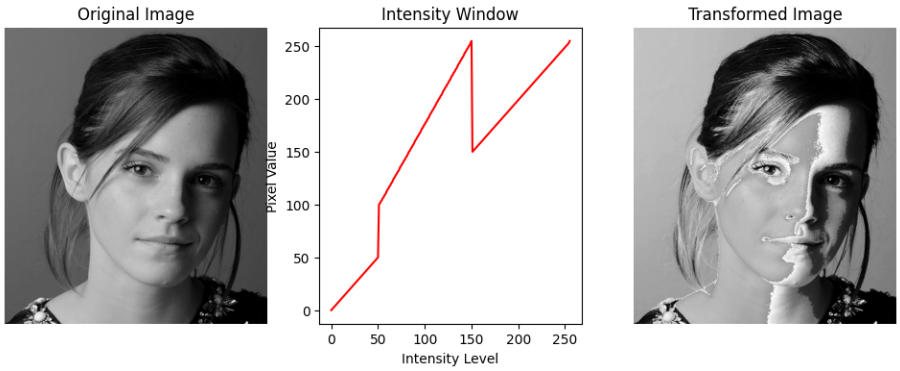


Figure 1: Intensity transformed image

Pixels with intensity levels ranging from 50 to 150 have been mapped to levels between 100 and 250. As a result, the gray areas of the image have become brighter. Due to the discontinuities in the applied intensity transformation, the output image exhibits discontinuous color regions.

Q2: Intensity transformation on brain image

Indentified the intensity values which are corresponding to white matter regions and gray matter regions. Then filtered out those regions by making other areas black (intensity value zero). Here are the Python code for extracting white matter region

```
white_th = np.array([120,175], dtype=np.uint8)
white_matter_transform = np.zeros(256, dtype=np.uint8)
white_matter_transform[white_th[0]:white_th[1]+1] = np.linspace(white_th[0],
↪  white_th[1],white_th[1]-white_th[0]+1,dtype=np.uint8)
white_matter = cv.LUT(brain_image, white_matter_transform)
```

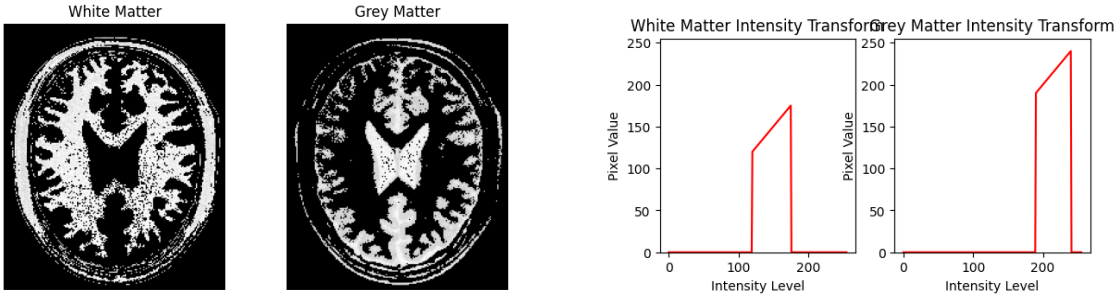


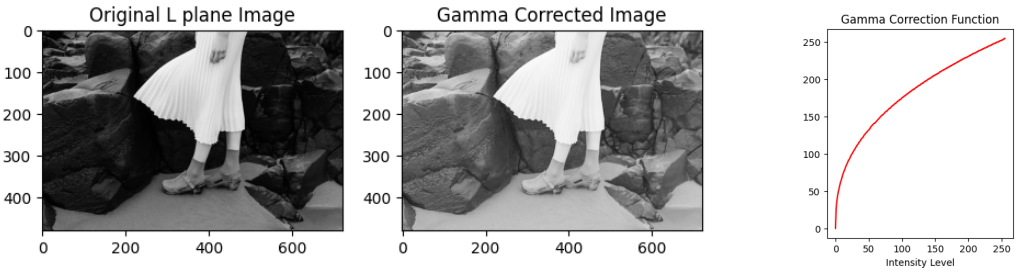
Figure 2: Intensity transformations on Brain image

Similarly, gray matter regions can be obtained. Used intensity transformation functions are shown in the above figure respectively to obtain the white matter and gray matter images.

Q3: Gamma correction

First, split the images into L, a, b color spaces. Gamma correction is applied to the L plane. $\gamma = 0.4$ selected such that both the black and white structures can be visible.

```
lab_image = cv.cvtColor(im3, cv.COLOR_BGR2LAB)
L, a, b = cv.split(lab_image)
gamma = 0.4
gamma_func = np.array([((i/255.0)**gamma)*255 for i in np.arange(256)], dtype=np.uint8)
L_gamma_corrected = cv.LUT(L, gamma_func)
```



(a) Gamma corrected image (b) transformation

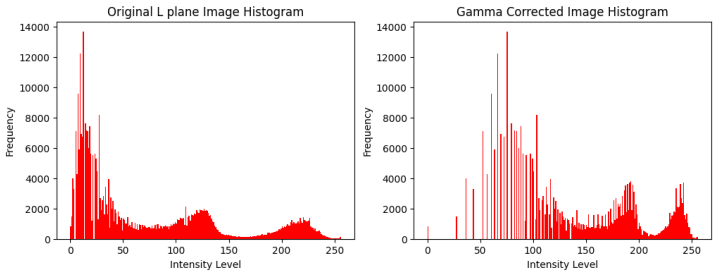


Figure 4: Histograms

From the histogram of the original image's L plane, we can observe that the distribution is left-skewed, indicating that the image is darkened. To enhance the visibility of the darker areas, we should apply gamma correction to map a narrow band of dark pixels to a wider range. This increases the detail in dark regions. For this purpose, a gamma value less than 1 is appropriate. A value of $\gamma = 0.4$ appears to produce visually pleasing results. The histogram of the output image shows that the intensities of the darker pixels have increased.

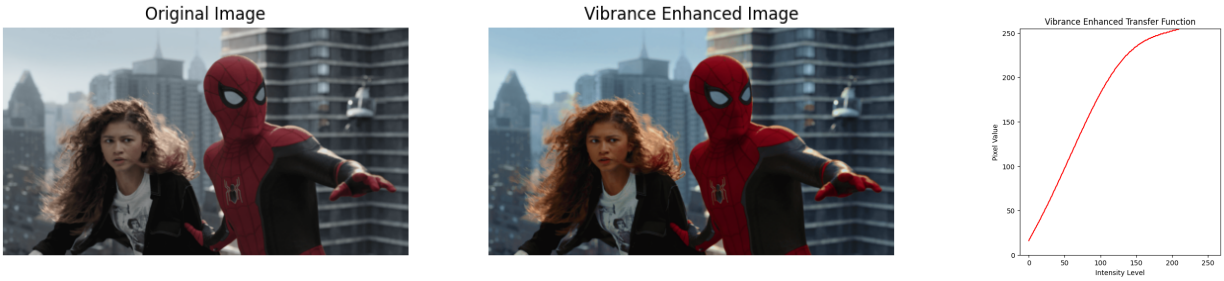
Q4: Vibrance enhancing

Split the image into hue, saturation and value planes. And applied $f(x) = \min\left(x + a \times 128e^{-\frac{(x-128)^2}{2\sigma^2}}, 255\right)$ intensity transformation to saturation plane to get a visually pleasing output using this code.

```
#splitting into hue, saturation, value planes
spider_im = cv.cvtColor(cv.imread(spider_path), cv.COLOR_BGR2HSV)
hue, saturation, value = cv.split(spider_im)

#intensity transformation
def f(x, a):
    sigma = 70
    return min(x + a*128*np.exp(-(x-128)**2/(2*sigma**2)), 255)
```

```
#select a from observation to have a visually pleasing output
a = 0.7

#applying of the mentioned intensity transformation
vibrance_enhanced = np.array([f(i, a) for i in np.arange(256)], dtype=np.uint8)
vibrance_enhanced_saturation = cv.LUT(saturation, vibrance_enhanced)

#recombine the three planes
hsv_image = cv.merge([hue, vibrance_enhanced_saturation, value])
rgb_image = cv.cvtColor(hsv_image, cv.COLOR_HSV2RGB)
```

Applying the transformation to the saturation plane boosts the color intensity, especially for mid-tone pixels, making the image more vibrant. At a = 0.7 we can get a visually pleasing output vibrant, vivid image.

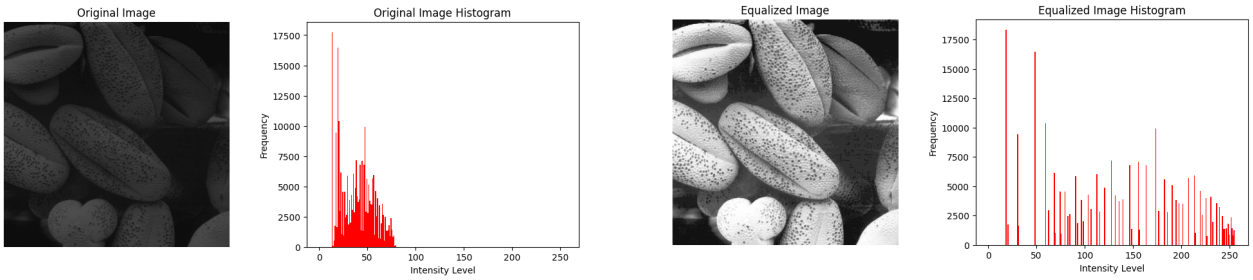


(a) Vibrance enhanced image (b) Transformation

Q5: Histogram Equalization

The following function carries out a histogram equalization for a given image.

```
def hist_equalized(image_array, L):
    M, N = image_array.shape

    # Get the histogram of the image
    histogram, bins = np.histogram(image_array.flatten(), L, [0,L])

    #Calculate cumulative density function
    cdf = histogram.cumsum()

    # Apply the histogram equalization transformation
    hist_equalized = (((L-1)/(M*N))*cdf).astype(np.uint8)
    equlized_image = cv.LUT(image_array, hist_equalized)
    return equlized_image
```



(a) Original Image (b) Histogram equalized image

The originally dark image has brightened after histogram equalization, with its histogram now spread across the full intensity range.

Q6: Histogram equalization on the image foreground

First, the image was split into hue, saturation, and value planes.

Figure 7: Jennifer image: hue, saturation, and value planes

From the above results, we can observe that the saturation plane is the suitable plane for creating the foreground mask. A threshold of intensity level 11 was selected to get the foreground mask. The image foreground was obtained by applying a foreground mask on the original image. Applied histogram equalization for the foreground. Then extracted background from the original image and the histogram-equalized foreground are combined to get the reconstructed image.

```
# Splitting into hue, saturation, value planes
hue, saturation, value = cv.split(cv.cvtColor(jennifer_im, cv.COLOR_BGR2HSV))

# Foregrounfd mask generation and foreground obtaining
sat_threshold = 11
mask = cv.threshold(saturation, sat_threshold, 255, cv.THRESH_BINARY)[1]
foreground = cv.bitwise_and(jennifer_im, jennifer_im, mask=sat_foreground_mask)

# Histogram equalization for the foreground
L = 256
histogram, bins = np.histogram(foreground.flatten(), L, [0,L])
cdf = histogram.cumsum()
M, N,c = image_array.shape
hist_equalized = (((L-1)/(M*N))*cdf).astype(np.uint8)
hist_equalized_foreground = cv.LUT(foreground, hist_equalized)

#extract background of original image
background = cv.bitwise_and(jennifer_im, jennifer_im, mask=~mask)

#reconstruted equlized image
reconstructed_image = cv.add(background, hist_equalized_foreground)

original_image_rgb = cv.cvtColor(jennifer_im, cv.COLOR_BGR2RGB)
reconstructed_image_rgb = cv.cvtColor(reconstructed_image, cv.COLOR_BGR2RGB)
```

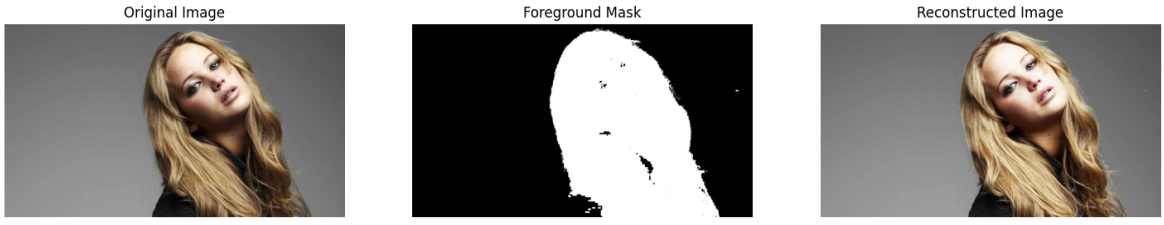


Figure 8: Foreground mask and the Reconstructed image with histogram equalized foreground

Since we only applied histogram equalization to the foreground, the reconstructed image has been enhanced. If we had equalized the entire image, including the background, the result would not have been the same.

Q7: Sobel filtering

The following Python code demonstrates the implementation applying the Sobel filter on the Einstein image 9 using given three methods.

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

# a) Using the existing filter2D to Sobel filter the image
sobel_filtered = cv.filter2D(einstein_im, cv.CV_64F, sobel_kernel)

# b) custom function for sobel filtering
def sobel_filter(image, kernel):
    M, N = image.shape
    (k,k) = kernel.shape

    filtered_image = np.zeros((M,N), dtype=np.float64)

    for i in range(1, M-k+2):
        for j in range(1, N-k+2):
            filtered_image[i,j] = np.sum(image[i-1:i+2, j-1:j+2]*kernel)

    return filtered_image

sobel_filtered_custom = sobel_filter(einstein_im, sobel_kernel)

# c) Using separable property
sobel_kernel_x = np.array([1,0,-1]).reshape(1,3)
sobel_kernel_y = np.array([1,2,1]).reshape(3,1)

sobel_filtered_x = cv.filter2D(einstein_im, cv.CV_64F, sobel_kernel_x)
sobel_filtered_separable = cv.filter2D(sobel_filtered_x, cv.CV_64F, sobel_kernel_y)
```

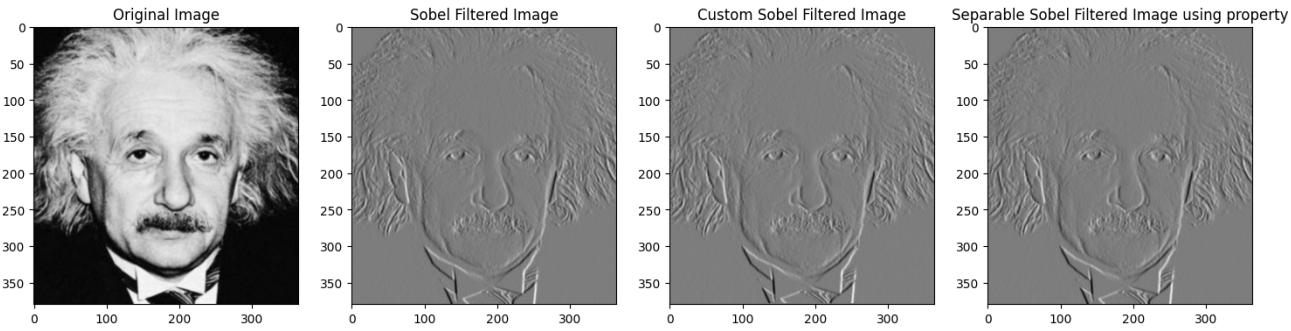


Figure 9: Applying sobel filtering

The outputs from all three methods appear visually similar. Form the results, we can see that The vertical edges of the image become prominent after applying the Sobel horizontal filter.

Q8: Zooming

Below is a function that can zoom images by a given factor $s \in (0, 10]$, supporting both nearest-neighbor and bilinear interpolation methods.

```
def zoom_image(image, zoom_factor, interpolation):
    M, N, c = image.shape
    new_M = int(M*zoom_factor)
    new_N = int(N*zoom_factor)
```

```
    if interpolation == 'nearest neighbour' or 'INTER_NEAREST':
        zoomed_image = cv.resize(image, (new_N, new_M), interpolation= cv.INTER_NEAREST)

    elif interpolation == 'bilinear' or 'INTER_LINEAR':
        zoomed_image = cv.resize(image, (new_N, new_M), interpolation=cv.INTER_LINEAR)

    else:
        try:
            zoomed_image = cv.resize(image, (new_N, new_M), interpolation=interpolation)
        except:
            raise ValueError('Invalid interpolation method')
            return None

    return zoomed_image
```

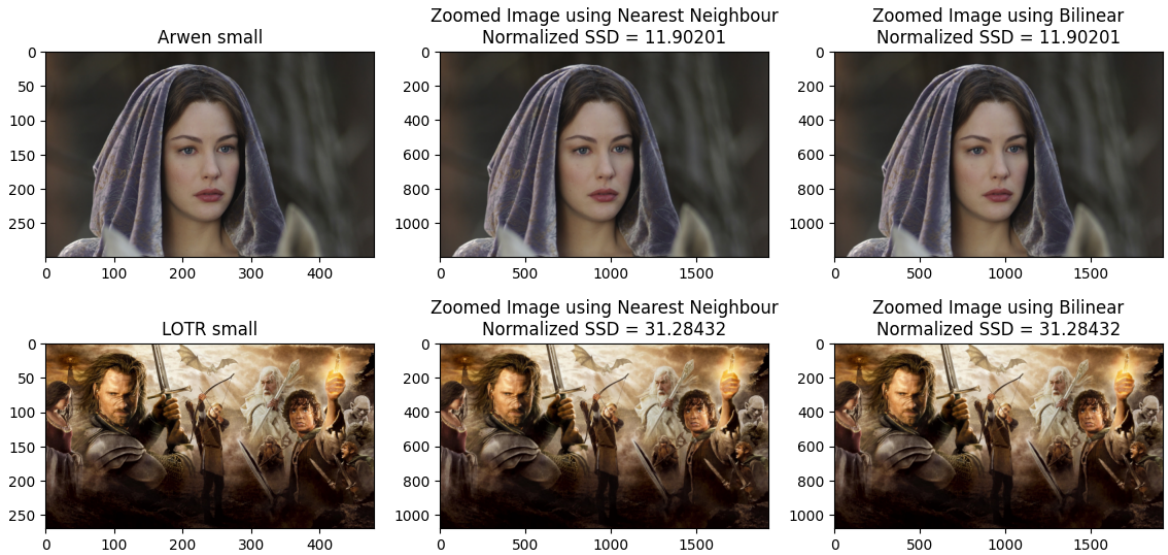


Figure 10: Zooming of images using the algorithm

The images can be zoomed by the given factor. The first image has a normalized sum of squared difference of 11.90 with the original image after zooming while the other image has a normalized sum of squared difference of 31.28. Therefore, the results are relatively good.

Q9: Segmentation using grabCut algorithm

Using the grabCut algorithm the image can be segmented.

```
daisy = cv.cvtColor(daisy, cv.COLOR_BGR2RGB)
M,N,c = daisy.shape

# Define an initial mask and rectangle
mask = np.zeros(daisy.shape[:2], np.uint8)
rect = (50, 50, daisy.shape[1] - 50, daisy.shape[0] - 50)

# Create the necessary arrays for grabCut as mentioned in the documentation
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)

# Apply grabCut algorithm
cv.grabCut(daisy, mask, rect, bgd_model, fgd_model, 5, cv.GC_INIT_WITH_RECT)
```

```
14
15 # Final mask where probable foreground and definite foreground are marked
16 final_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
17
18 # Extract the foreground and background images
19 foreground = daisy * final_mask[:, :, np.newaxis]  # Foreground
20 background = daisy * (1 - final_mask[:, :, np.newaxis])  # Background
```

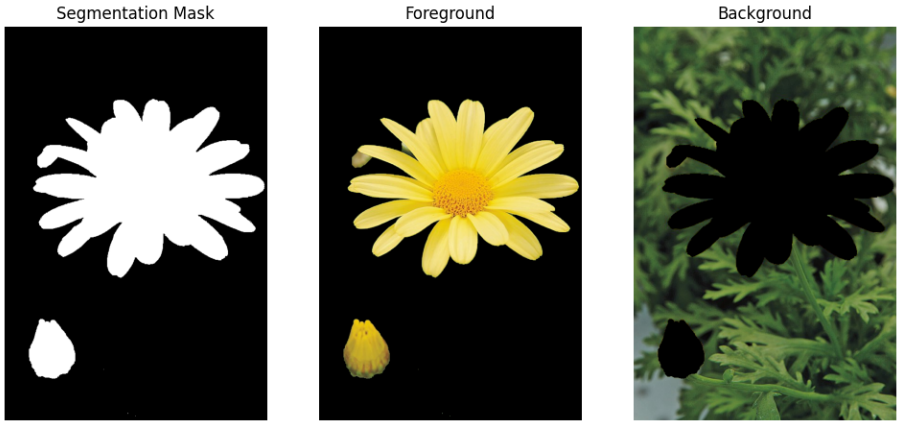


Figure 11: Segmentation using grabCut

And the background is blurred using a Gaussian kernel. Gaussian kernel smoothes the image background. Therefore, get blurred.

```
1 # apply gaussian kernel for smoothing
2 background_smoothed = cv.GaussianBlur(background, (15,15), 0)
3
4 # Combine the foreground and smoothed background
5 enhanced_image = cv.add(foreground, background_smoothed)
```

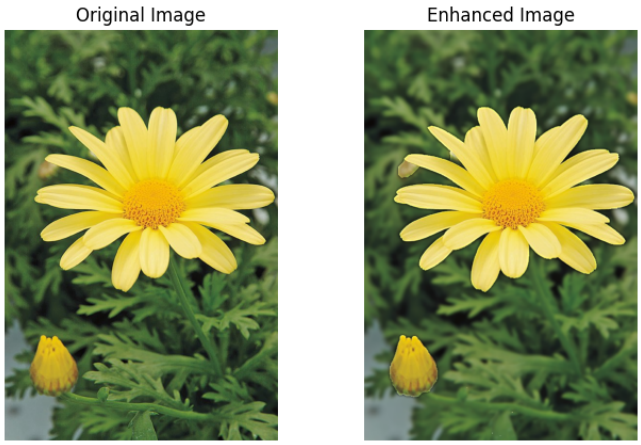


Figure 12: Enhanced image

Since the background has blurred, the daisy image has been enhanced. In the background image pixels of the foreground region has intensity value zero. When applying the Gaussian kernel over the background, the pixels of the border between the foreground and background region have become darker because Gaussian kernel is averaging the intensity values in the neighborhood of the border. Therefore, the background just beyond the edge of the flower quite dark in the enhanced image.