EN3160 – Image Processing and Machine Vision

Assignment 1 - Intensity Transformations and Neighborhood Filtering

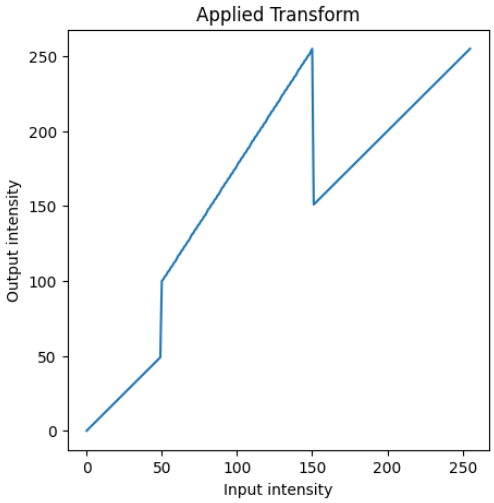
Name : H. L. Kulatunga
Index No. : 200318K
GitHub : https://github.com/helithaK/Image-processing

1)

```
c = np.array([(50,50), (50, 100), (150,255), (150, 150)])
t1 = np.linspace(0,c[0,1],c[0,0]+1-0).astype('uint8')
print(len(t1))
t2 = np.linspace(c[0,1]+1,c[1,1],c[1,0]-c[0,0]).astype('uint8')

#Implementing the piecewise function
t3 = np.linspace(c[1,1]+1,c[2,1],c[2,0]-c[1,0]).astype('uint8')
t4 = np.linspace(c[2,1]+1,c[3,0],c[2,0]-c[3,0]).astype('uint8')
t5 = np.linspace(c[2,0]+1,255,255-c[2,0]).astype('uint8')

#Concatenation
transform = np.concatenate((t1,t2),axis=0).astype('uint8')
transform = np.concatenate((transform,t3),axis=0).astype('uint8')
transform = np.concatenate((transform,t4),axis=0).astype('uint8')
transform = np.concatenate((transform,t5),axis=0).astype('uint8')
```

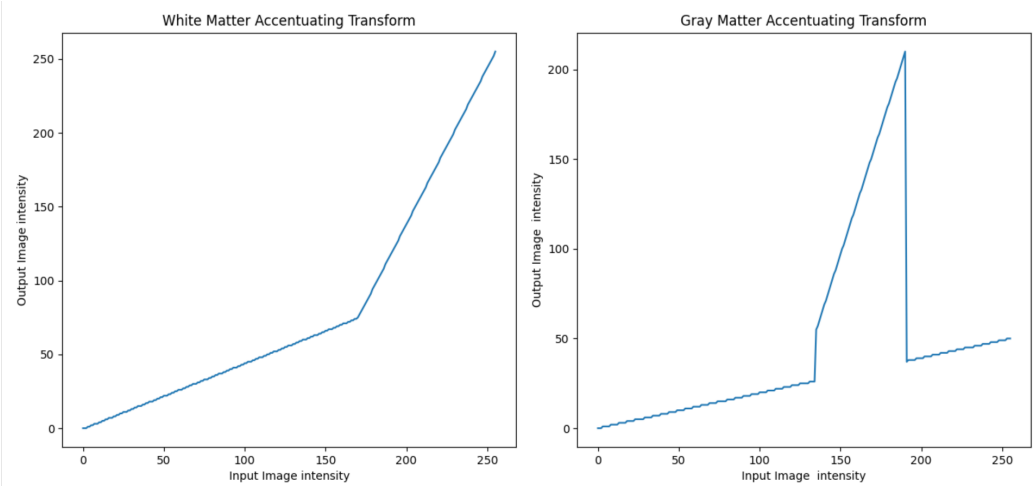


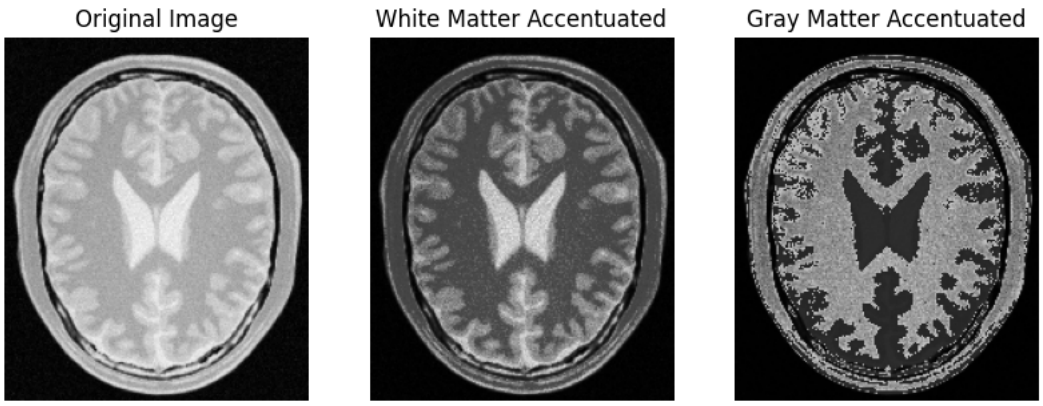
2)

```
# White matter enhancement function
def enhance_white_matter(image):
    x_mid, y_mid = 170, 75
```

```
    white_transform = np.arange(0, 256).astype(np.uint8)
    white_transform[:x_mid + 1] = np.linspace(0, y_mid, x_mid + 1, dtype=np.uint8)
    white_transform[x_mid:] = np.linspace(y_mid, 255, 256 - x_mid, dtype=np.uint8)
    return white_transform[image]

# Gray matter enhancement function
def enhance_gray_matter(image):
    x1, x2 = 135, 190
    y1, y2 = 55, 210
    grey_transform = np.linspace(0, 50, 256)
    grey_transform = np.round(grey_transform).astype(np.uint8)
    grey_transform[x1:x2 + 1] = np.linspace(y1, y2, x2 + 1 - x1, dtype=np.uint8)
    return grey_transform[image]
```



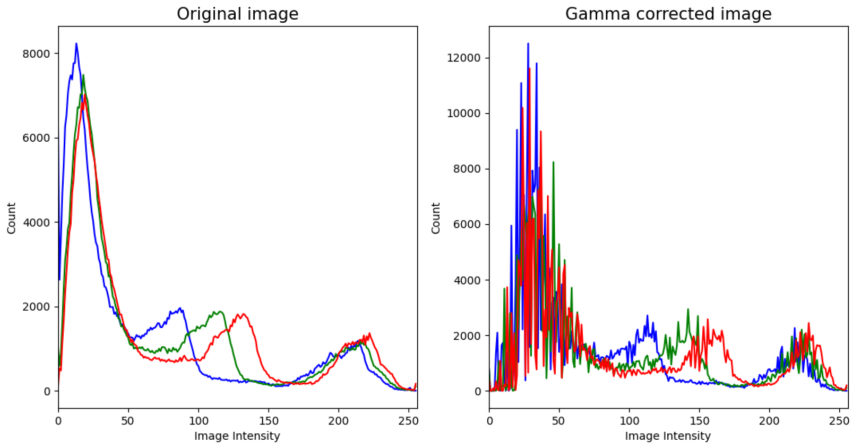


3)

```
lab_image = cv2.cvtColor(image, cv2.COLOR_BGR2Lab)
l_channel = lab_image[:,:,0]
```

```
gamma_val = 0.6  # You can change this value as needed
l_channel_corrected = np.power(l_channel / 255.0, gamma_val) * 255.0
l_channel_corrected = np.clip(l_channel_corrected, 0, 255).astype(np.uint8)
lab_image[:,:,0] = l_channel_corrected
output_image = cv2.cvtColor(lab_image, cv2.COLOR_Lab2BGR)
```

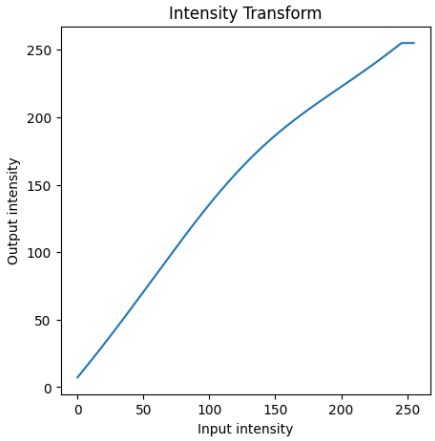

```
def plot_BGR_histogram(image):
    colors = ('b', 'g', 'r')
    for i, color in enumerate(colors):
        hist = cv2.calcHist([image], [i], None, [256], [0, 256])
        plt.plot(hist, color=color)
        plt.xlim([0, 256])
```



4)

```
# Convert the image to HSV
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
hue, saturation, value = cv2.split(hsv_image)
a = 0.4  # Increase 'a' for a stronger effect
b = 70  # Increase 'b' for a stronger effect
a = max(0, min(1, a))
transformed_saturation = np.clip(
    saturation + (a * 128) * np.exp(-((saturation - 128) ** 2) / (2 * b ** 2)), 0, 255
```

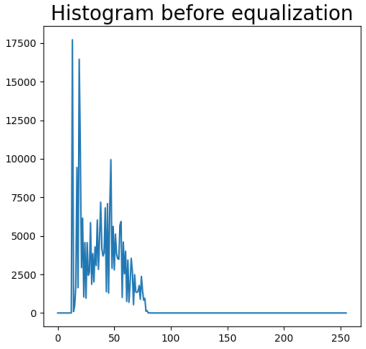
```
).astype(np.uint8)
modified_hsv_image = cv2.merge([hue, transformed_saturation, value])
modified_image = cv2.cvtColor(modified_hsv_image, cv2.COLOR_HSV2BGR)
```

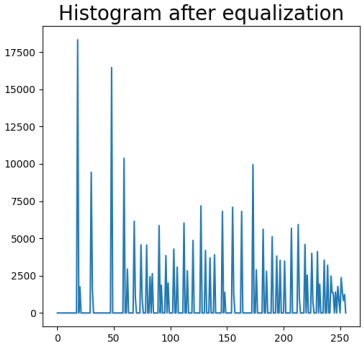


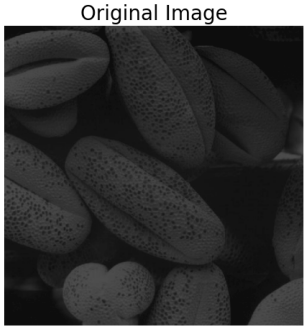
Therefore correct regulation of the gamma value will lead to a more vibrant image

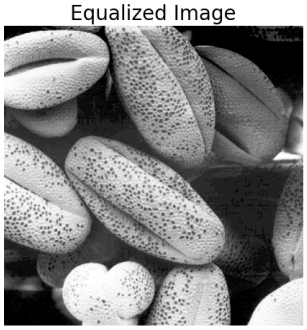
5)

```
# Function for histogram equalization
def histogram_equalization(image):
    total_pixels = image.shape[0] * image.shape[1]
    hist, bins = np.histogram(image.ravel(), 256, [0, 256])
    cdf = hist.cumsum()
    transform = (cdf * 255 / total_pixels).astype(np.uint8)
    enhanced_image = transform[image]
    return enhanced_image
```







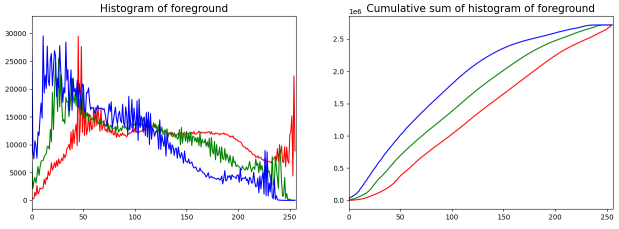


6)

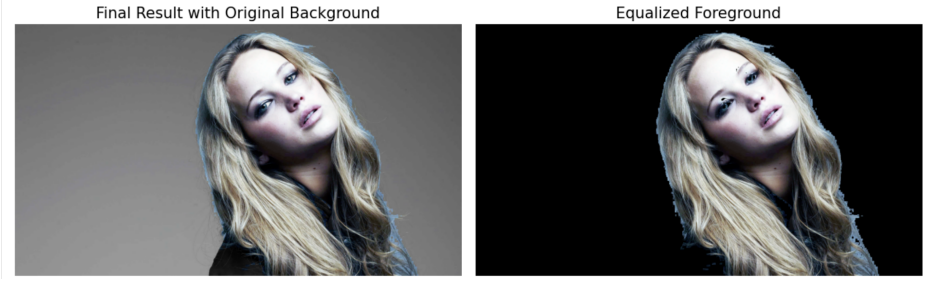
```
# Convert the input image to HSV
hsv_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2HSV)# Separate the HSV planes
hue_plane = hsv_image[:, :, 0]
saturation_plane = hsv_image[:, :, 1]
value_plane = hsv_image[:, :, 2]
```


```
# Threshold the Saturation plane to obtain a mask for the foreground
threshold = 13
foreground_mask = (saturation_plane > threshold).astype(np.uint8) * 255
foreground_mask_3d = np.repeat(foreground_mask[:, :, None], 3, axis=2)
# Extract the foreground using the mask
foreground_hsv = np.bitwise_and(hsv_image, foreground_mask_3d)
foreground_rgb = cv2.cvtColor(foreground_hsv, cv2.COLOR_HSV2RGB)
```


```
for i, channel in enumerate(channels):
    hist = cv2.calcHist([foreground_rgb], [i], foreground_mask, [256], [0, 256])
    ax[0].plot(hist, color=channel)
    ax[0].set_xlim([0, 256])
    cumulative_hist = np.cumsum(hist)
    ax[1].plot(cumulative_hist, color=channel)
    ax[1].set_xlim([0, 256])
    transformation = cumulative_hist * 255 / cumulative_hist[-1]
    equalized_foreground[:, :, i] = transformation[foreground_rgb[:, :, i]]
equalized_foreground = np.bitwise_and(equalized_foreground, foreground_mask_3d)
```



```
# Extract the background and combine it with the equalized foreground
background_mask_3d = 255 - foreground_mask_3d
background_hsv = np.bitwise_and(hsv_image, background_mask_3d)
background_rgb = cv2.cvtColor(background_hsv, cv2.COLOR_HSV2RGB)
final_result = background_rgb + equalized_foreground
```



7)

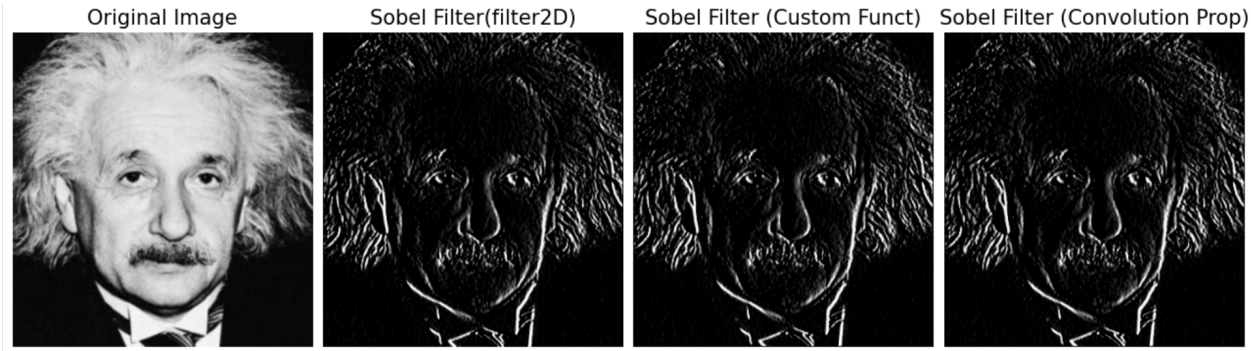
```
def custom_filter(image, kernel):
    assert kernel.shape[0] % 2 == 1 and kernel.shape[1] % 2 == 1
    k_hh, k_hw = kernel.shape[0] // 2, kernel.shape[1] // 2
    h, w = image.shape
    image_float = cv2.normalize(image.astype('float'), None, 0, 1, cv2.NORM_MINMAX)
    result = np.zeros(image.shape, 'float')

    for m in range(k_hh, h - k_hh):
        for n in range(k_hw, w - k_hw):
            result[m, n] = np.dot(image_float[m-k_hh: m+k_hh+1, n-k_hw: n+k_hw+1].flatten(), kernel.flatten())

    result = result * 255
    result = np.minimum(255, np.maximum(0, result)).astype(np.uint8)
    return result
```

```
# Define a Sobel-like vertical kernel
sobel_kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
result_a = cv2.filter2D(input_img, -1, sobel_kernel)  # Using filter2D
result_b = custom_filter(input_img, sobel_kernel)    # Using custom function
```

```python
def custom_filter_in_steps(image, kernel1, kernel2):
    image_float = cv2.normalize(image.astype('float'), None, 0, 1, cv2.NORM_MINMAX)
    result = custom_filter_step(custom_filter_step(image_float, kernel1), kernel2)
    result = result * 255
    result = np.minimum(255, np.maximum(0, result)).astype(np.uint8)
    return result
```



8)

```python
def custom_interpolate(image, indices, interpolation_method):
    if interpolation_method == 'nearest':
        indices[0] = np.minimum(np.round(indices[0]), image.shape[0] - 1)
        indices[1] = np.minimum(np.round(indices[1]), image.shape[1] - 1)
        indices = indices.astype(np.uint64)
        return image[indices[0], indices[1]]
    elif interpolation_method == 'bilinear':
        floors = np.floor(indices).astype(np.uint64)
        ceils = floors + 1
        ceils_limited = [np.minimum(ceils[0], image.shape[0] - 1), np.minimum(ceils[1], image.shape[1] - 1)]
        p1 = image[floors[0], floors[1]]
        p2 = image[floors[0], ceils_limited[1]]
        p3 = image[ceils_limited[0], floors[1]]
        p4 = image[ceils_limited[0], ceils_limited[1]]
        indices = np.repeat(indices[:, :, :, None], 3, axis=3)
        ceils = np.repeat(ceils[:, :, :, None], 3, axis=3)
        floors = np.repeat(floors[:, :, :, None], 3, axis=3)
        m1 = p1 * (ceils[1] - indices[1]) + p2 * (indices[1] - floors[1])
        m2 = p3 * (ceils[1] - indices[1]) + p4 * (indices[1] - floors[1])
        m = m1 * (ceils[0] - indices[0]) + m2 * (indices[0] - floors[0])
        return m.astype(np.uint8)

def custom_zoom(image, factor, interpolation_method='nearest'):
    height, width, _ = image.shape
    zoomed_height, zoomed_width = round(height * factor), round(width * factor)
    zoomed_image = np.zeros((zoomed_height, zoomed_width, 3)).astype(np.uint8)
    zoomed_indices = np.indices((zoomed_height, zoomed_width)) / factor
    zoomed_image = custom_interpolate(image, zoomed_indices, interpolation_method)
    return zoomed_image
```


```
C:\Users\acer\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\acer\PycharmProjects\pythonProject1\main.py
Normalized total of Squared Differences
Image 01: Nearest Neighbor Value = 0.000617, Bilinear Value = 0.000604
Image 02: Nearest Neighbor Value = 0.000258, Bilinear Value = 0.000249
Image 04: Nearest Neighbor Value = 0.001257, Bilinear Value = 0.001256
Image 05: Nearest Neighbor Value = 0.000840, Bilinear Value = 0.000826
```

So we can conclude that bilinear interpolation provided better results than nearest neighbor interpolation

9)

```
segmentation_mask = np.zeros(image.shape[:2], np.uint8)
segmentation_mask[150:550, 50:550] = cv2.GC_PR_FGD
segmentation_mask[250:410, 220:380] = cv2.GC_FGD
bg_model = np.zeros((1, 65), np.float64)
fg_model = np.zeros((1, 65), np.float64)
roi = (50, 100, 500, 500)
num_iterations = 5
cv2.grabCut(image, segmentation_mask, roi, bg_model, fg_model, num_iterations, cv2.GC_INIT_WITH_MASK)
foreground_mask = np.where((segmentation_mask == 2) | (segmentation_mask == 0), 0, 1).astype('uint8')
background_mask = 1 - foreground_mask
foreground_image = image * foreground_mask[:, :, np.newaxis]
background_image = image * background_mask[:, :, np.newaxis]
kernel_size = 51
sigma = 5
blurred_background = cv2.GaussianBlur(background_image, (kernel_size, kernel_size), sigma)
enhanced_image = blurred_background + foreground_image
```