Department of Electronic and Telecommunication Engineering University of Moratuwa

EN3160 - Image Processing and Mavhine Vision

Assignment 01

210668P Vidmal H.V.P.

1 Question 1: Intensity Transformation

```
t1 = np.linspace(0, 50, num=51).astype('uint8')
t2 = np.linspace(100, 255, num=100).astype('uint8')
t3 = np.linspace(150, 255, num=105).astype('uint8')
# Concatenate the arrays
t = np.concatenate((t1, t2, t3), axis=0).astype('uint8')
print(t.shape)
# Apply the transformation
f = t[image]
```

Discussion: This intensity transformation enhances mid-range gray values while preserving extreme light and dark areas. Its key feature is abrupt transitions in the mapping function, creating sharp contrast changes in mid-tones. These discontinuities produce a visually striking effect, increasing contrast in specific regions while maintaining the original image's overall structure.

2 Question 2: Accentuating White and Gray Matter

```
#white matter
mu = 150
sigma = 25
x = np.linspace(0, 255, 256)
t1 = 255 * np.exp(-((x - mu)*2) / (2 * sigma*2))
#gray matter
mu = 210
sigma = 25
x = np.linspace(0, 255, 256)
t2 = 255 * np.exp(-((x - mu)*2) / (2 * sigma*2))
```

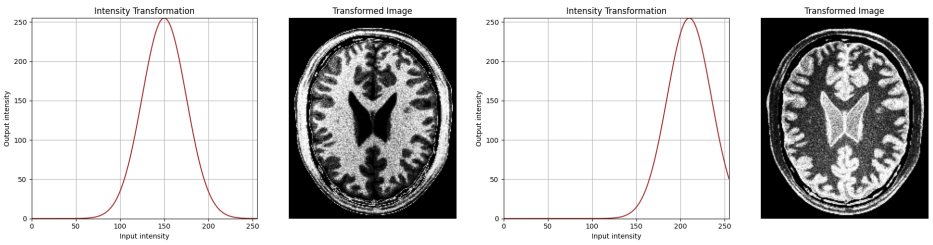


Figure 1: White Matter Figure 2: Gray Matter

Discussion The Gaussian transformations effectively enhanced specific brain tissues in the MRI. Centering at intensity 150 highlighted white matter, while centering

at 210 emphasized grey matter. These results demonstrate the method's ability to selectively enhance different anatomical features. However, the original image's inherent noise remained unaffected by the transformations, persisting in both enhanced images.

3 Question 3: Gamma Correction in L*a*b* Color Space

```
L,a,b = cv2.split(shadow_image)

gamma = 0.6
t = np.array([(i/255.0)**(gamma)*255 for i in np.arange(0, 256)
    ]).astype('uint8')
new_shadow_image = cv2.LUT(L, t)

merge_image = cv2.merge([new_shadow_image, a, b])
after_new_shadow_image= cv2.cvtColor(merge_image, cv2.
    COLOR_LAB2RGB)
```

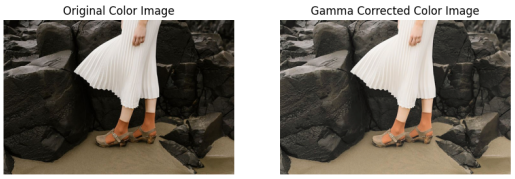


Figure 3: WOriginal and gamma corrected output

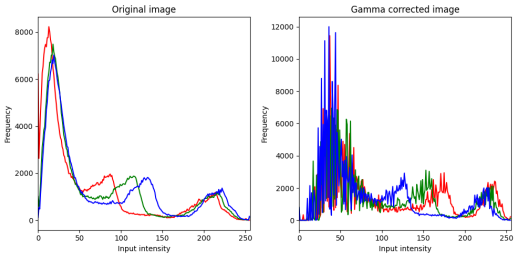


Figure 4: Histograms

Discussion Our shadow correction method uses gamma correction (0.6) on the LAB color space's lightness channel. This approach effectively brightens shadows while preserving color balance. However, it introduces quantization artifacts due to the non-linear transformation of discrete pixel values. The resulting histograms show noise-like patterns from uneven intensity redistribution. This exemplifies a common image processing trade-off: improved shadow visibility versus introduced artifacts. Potential improvements include higher bit-depth processing or adaptive techniques. Overall, the method achieves its primary goal of shadow enhancement, but with some compromises in image quality.

4 Question 4: Enhancing Vibrance

```
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_hsv = cv2.cvtColor(image_rgb, cv2.COLOR_RGB2HSV)
# Split into Hue, Saturation, and Value channels
hue, saturation, value = cv2.split(image_hsv)
def intensity_transformation(saturation, a, sigma=70):
    # Precompute constant values
    exponent = np.exp(-((saturation - 128) ** 2) / (2 * (sigma
    ** 2)))
    # Apply the intensity transformation
```

```
    transformed_saturation = np.clip(saturation + a * 128 *
    exponent, 0, 255)
    return transformed_saturation
a_value=0.6
transformed_saturation = intensity_transformation(saturation.
    astype(np.float32), a=a_value)
# Recombine the transformed saturation with the original hue and
    value
transformed_hsv = cv2.merge([hue, transformed_saturation.astype(
    np.uint8), value])
# Convert the HSV image back to RGB
transformed_image_rgb = cv2.cvtColor(transformed_hsv, cv2.
    COLOR_HSV2RGB)
```

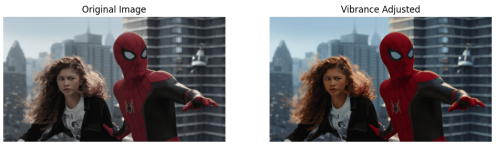


Figure 5: Vibrance Enhanced Image

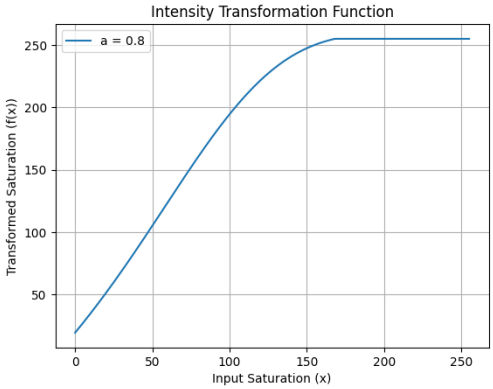


Figure 6: Intensity Transformation Function

Discussion

Our vibrance enhancement method targets the saturation plane in HSV color space. We observed that colorful areas have high saturation values, while colorless areas have low values. We applied an intensity transformation

$$f(x) = \min\left(x + a \cdot 128 \cdot e^{-\frac{(x-128)^2}{2\sigma^2}}, 255\right)$$

to the saturation channel, with $a = 0.6$ and $\sigma = 70$. This approach amplifies colorful areas while leaving colorless regions unchanged. The exponential term ensures the effect is strongest around mid-range saturation values, resulting in a natural-looking enhancement. We then recombined the transformed saturation with the original hue and value channels to produce the final vibrance-enhanced image.

5 Question 5: Histogram Equalization

```
hist, bins = np.histogram(shell_image.flatten(), 256, [0, 256])
cdf = hist.cumsum()
image_equalized = np.interp(shell_image.flatten(), np.arange(0,
    256), cdf_normalized)
image_equalized = image_equalized.reshape(shell_image.shape)
```

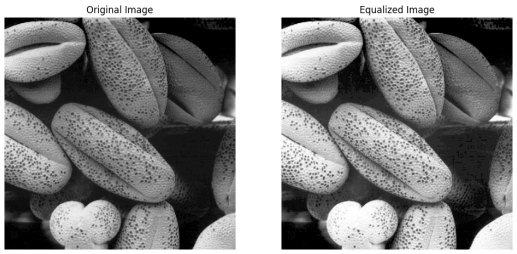


Figure 7: Output image after histogram equalization

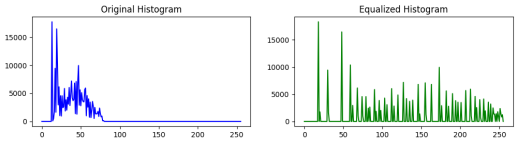
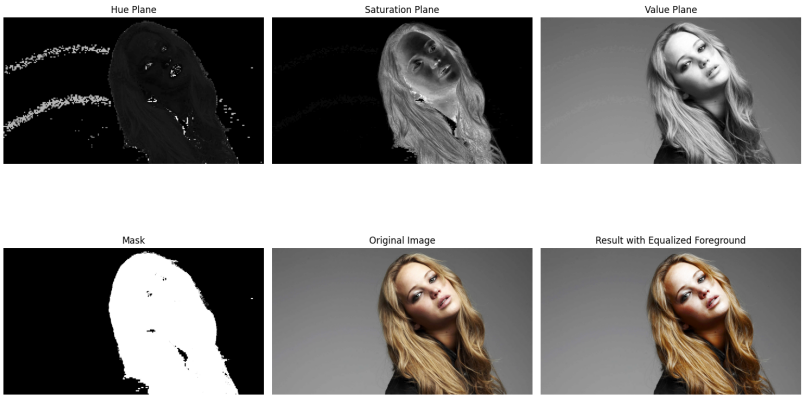


Figure 8: Histograms after equalization

Discussion: The histogram equalization applied to the shell image has markedly improved its visual quality. Figure 7 shows enhanced contrast and sharper details in the equalized image, particularly in the shells' textures. Figure 8 demonstrates the histogram transformation, with pixel intensities shifting from a concentrated lower range to a more even distribution. This redistribution unveils previously hidden details in darker areas, though it may slightly increase noise. Overall, the technique effectively enhances the image's dynamic range, resulting in a more informative and visually striking representation of the shell structures.

6 Question 6: Histogram Equalization of Foreground

```
jeniffer_image_hsv = cv2.cvtColor(jeniffer_image,cv2.
    COLOR_BGR2HSV)
hue, saturation, value = cv2.split(jeniffer_image_hsv)
ret, mask = cv2.threshold(saturation,15,255, cv2.THRESH_BINARY)
foreground = cv2.bitwise_and(saturation, saturation, mask=mask)
foreground_hist, bins = np.histogram(foreground[foreground > 0],
     256, [0, 256])
# Compute the CDF (Cumulative Distribution Function)
cdf = foreground_hist.cumsum()
cdf_normalized = (cdf-cdf.min())*255/(cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype('uint8')
foreground_equalized = cdf_normalized[foreground]
background = cv2.bitwise_and(saturation, saturation, mask=cv2.
    bitwise_not(mask))
combined_image = cv2.add(foreground_equalized, background)
hsv_equalized = cv2.merge([hue, combined_image, value])
final_image = cv2.cvtColor(hsv_equalized, cv2.COLOR_HSV2BGR)
```



Approach: This code performs selective histogram equalization on an image's foreground. It converts the image to HSV color space, creates a mask based on saturation to identify the foreground, and applies histogram equalization only to these areas. The process involves calculating the histogram of foreground pixels, deriving a normalized cumulative distribution function (CDF), and using it to remap pixel intensities. The background remains unchanged. Finally, the equalized foreground is combined with the original background and converted back to BGR color space. This method enhances contrast in the foreground while preserving the background, potentially improving the overall image quality and visual appeal.

7 Question 7: Sobel Filtering

```
def sobel_filter(image, kernel):
    [h, w] = np.shape(image) # Get rows and columns of the image
    output= np.zeros(shape=(h, w))
    for i in range(1,h-1):
        for j in range(1, w-1):
            output[i, j] = np.sum(np.multiply(kernel,image[i-1:i
    + 2, j-1:j + 2]))
    return output
# Manually apply Sobel filters
sobel_manual_x = sobel_filter(image, sobel_x)
sobel_manual_y = sobel_filter(image, sobel_y)
# Sobel filter property split into two 1D filters
kernel_x_row = np.array([[1, 0, -1]])  # 1x3
kernel_x_col = np.array([[1], [2], [1]])  # 3x1
# Apply separable Sobel filtering
sobel_separable_x = cv2.filter2D(image, -1, kernel_x_row)
sobel_separable_x = cv2.filter2D(sobel_separable_x, -1,
   kernel_x_col)
# Sobel filter in the Y direction using the same property
kernel_y_row = np.array([[1], [0], [-1]])  # 3x1
kernel_y_col = np.array([[1, 2, 1]])  # 1x3
sobel_separable_y = cv2.filter2D(image, -1, kernel_y_row)
sobel_separable_y = cv2.filter2D(sobel_separable_y, -1,
   kernel_y_col)
```

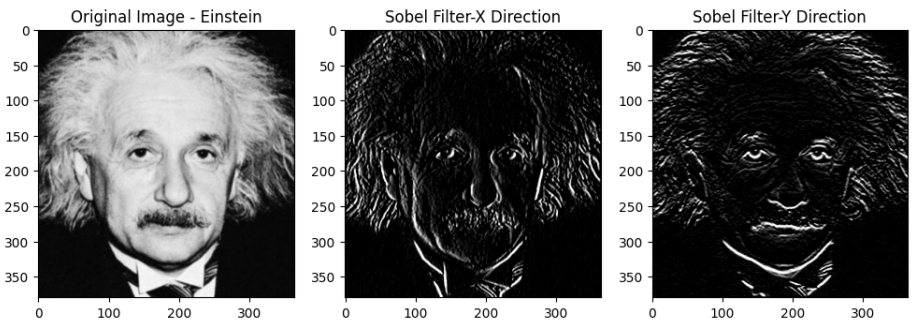


Figure 9: Existing filter2D output

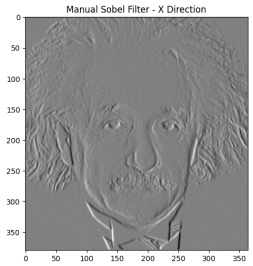


Figure 10: Custom filter Output

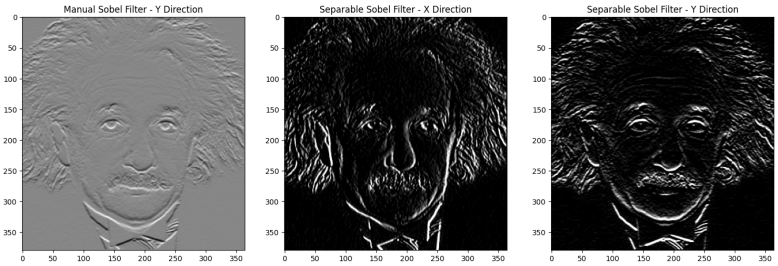
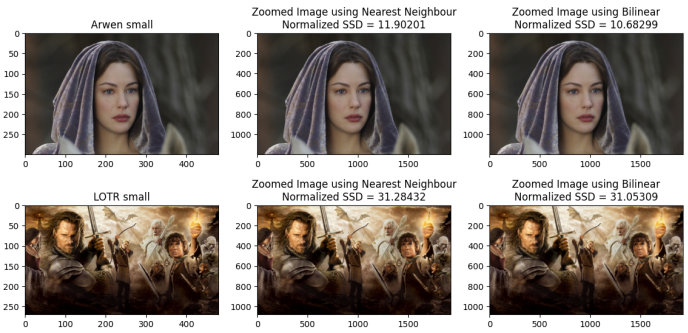


Figure 11: Output using Property

8 Question 8: Image Zooming

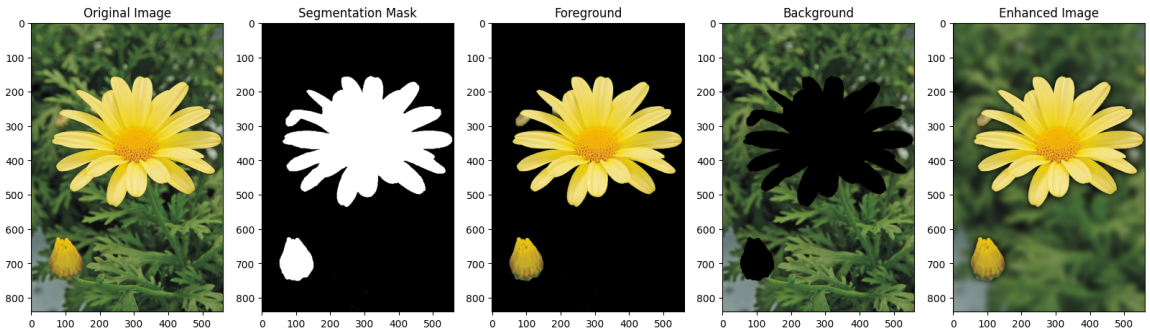
Approach: The code implements a zoom_image function that resizes images using specified zoom factors and interpolation methods. It handles nearest neighbor and bilinear interpolation, with a fallback for other OpenCV methods. An ssd function is also implemented to calculate the normalized Sum of Squared Differences between original and zoomed images, providing a metric to assess zooming quality. This approach allows for flexible image zooming and quantitative quality evaluation.

```
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 == cv2.INTER_NEAREST:
        zoomed_image = cv2.resize(image, (new_N, new_M),
    interpolation=cv2.INTER_NEAREST)
    elif interpolation == cv2.INTER_LINEAR:
        zoomed_image = cv2.resize(image, (new_N, new_M),
    interpolation=cv2.INTER_LINEAR)
    else:
        try:
            zoomed_image = cv2.resize(image, (new_N, new_M),
    interpolation=interpolation)
        except:
            raise ValueError('Invalid interpolation method')
    return zoomed_image
    # Calculate SSD
def ssd(original_image, zoomed_image):
    return np.sum(np.square(original_image - zoomed_image)) /
    original_image.size
```



9 Question 9: Image Segmentation Using GrabCut

```python
daisy_rgb = cv2.cvtColor(daisy_image, cv2.COLOR_BGR2RGB)
M,N,c = daisy_rgb .shape
# Define an initial mask and rectangle
mask = np.zeros(daisy_rgb .shape[:2], np.uint8)
rect = (50, 50, daisy_rgb .shape[1] - 50, daisy_rgb .shape[0] -
    50)
bgd_model = np.zeros((1, 65), np.float64)
fgd_model = np.zeros((1, 65), np.float64)
# Apply grabCut algorithm
cv2.grabCut(daisy_rgb , mask, rect, bgd_model, fgd_model, 5, cv2
    .GC_INIT_WITH_RECT)foreground are marked
final_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('
    uint8')
foreground = daisy_rgb  * final_mask[:, :, np.newaxis]
background = daisy_rgb  * (1 - final_mask[:, :, np.newaxis])
def apply_background_blur(image, mask, blur_strength=55):
    blurred = cv2.GaussianBlur(image, (blur_strength,
    blur_strength), 0)
    # Use the mask to combine the original foreground with the
    blurred background
    result = np.where(mask[:,:,np.newaxis] == 1, image, blurred)
    return result
# Create enhanced image with blurred background
enhanced_image = apply_background_blur(background, final_mask)
enhanced_image =cv2.add(foreground, enhanced_image)
```



(c) Reason edge of the flower quite dark in the enhanced image : The dark area around the flower's edge in the enhanced image is caused by the blurring process. When applying the Gaussian blur to the background, pixels near the foreground-background border are averaged with black pixels (zero values) from the foreground mask. This averaging creates a darkening effect just beyond the flower's outline, resulting in the observed darker region in the background close to the flower's edge.

You can find my notebook from github.