Department of Electronic and Telecommunication Engineering University of Moratuwa

EN3160 - Image Processing and Computer Vision

Assignment 1

210418C Nayanthara J.N.P.

1 Intensity Transformation

Intensity transformation is a process used in image processing to change the brightness or contrast of an image by manipulating the pixel values. Each pixel in an image has an intensity (or brightness level), and intensity transformation applies a mathematical function to these pixel values to enhance or modify the image.

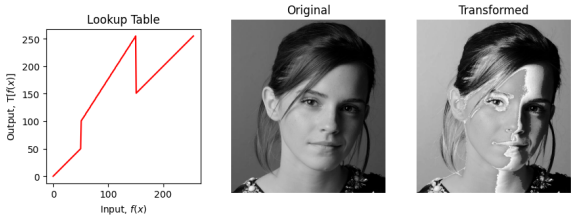
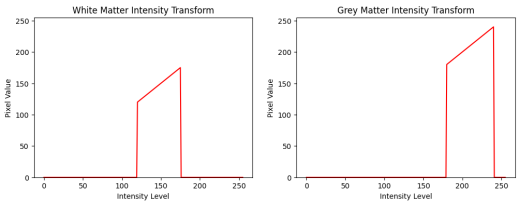


Figure 1: Intensity Transformation Outputs

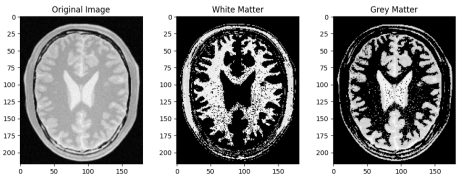
```python
c = np.array([(50, 50), (50, 100), (150, 255), (150, 150), (255, 255)])
t1 = np.linspace(0, c[0, 1], c[0, 0] + 1).astype('uint8')  # from 0 to c[0]
t2 = np.linspace(c[0, 1] + 1, c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.linspace(c[1, 1] + 1, c[2, 1], c[2, 0] - c[1, 0]).astype('uint8')
t4 = np.linspace(c[2, 1] + 1, c[3, 1], c[3, 0] - c[2, 0]).astype('uint8')
t5 = np.linspace(c[3, 1] + 1, c[4, 1], 255 - c[3, 0]).astype('uint8')

t = np.concatenate((t1, t2), axis=0).astype('uint8')
t = np.concatenate((t, t3), axis=0).astype('uint8')
t = np.concatenate((t, t4), axis=0).astype('uint8')
t = np.concatenate((t, t5), axis=0).astype('uint8')
g = cv.LUT(f, t)
```

2 Grey and White Matter of a Brain Proton

We apply intensity transformations to a brain proton density image to enhance the visibility of white and gray matter. By adjusting intensity values, we make these regions more distinguishable. The results are shown as plots, highlighting the adjustments for each area.



(a) Intensity Level Transformation

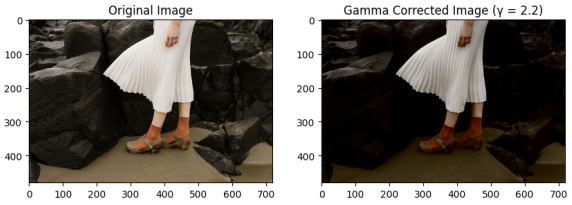


(b) Grey Matter and White Matter

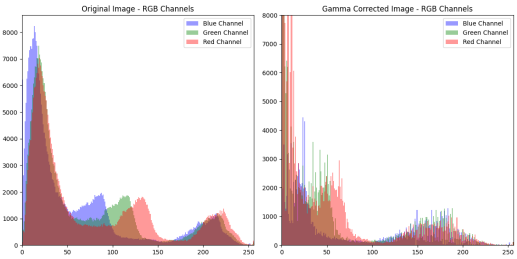
```python
white_th = np.array([120,175], dtype=np.uint8)
grey_th = np.array([180,240], dtype=np.uint8)

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)
grey_matter_transform = np.zeros(256, dtype=np.uint8)
grey_matter_transform[grey_th[0]:grey_th[1]+1] = np.linspace(grey_th[0],
    grey_th[1],grey_th[1]-grey_th[0]+1,dtype=np.uint8)

white_matter = cv2.LUT(brain_image, white_matter_transform)
grey_matter = cv2.LUT(brain_image, grey_matter_transform)
```

3 Gamma Correction

Gamma correction is an image processing technique that adjusts brightness and contrast to match human visual perception. It brightens darker areas when gamma < 1 and darkens brighter areas when gamma > 1, making the image appear more natural to the human eye.



(a) Picture with Increased Gamma



(b) Intensity Transformation

```python
# Convert the image to the L*a*b* color space
lab_image = cv.cvtColor(image, cv.COLOR_BGR2LAB)
L, a, b = cv.split(lab_image)# Split into L*, a*, b* channels
L = L/255.0 #normalize L

gamma = 2.2  # Example gamma value
L_corrected = np.array( 255*(L) ** (gamma), dtype='uint8')
lab_corrected = cv.merge((L_corrected, a, b))
```

4 Vibrancy

Vibrancy enhances dull colors while preserving bright ones, making photos more colorful and balanced without looking unnatural. It's often used to improve landscapes or portraits subtly.

4.1 Split Image into Hue, Saturation, and Value planes

Figure 4: HSV Planes

```python
# Intensity transformation function
def intensity_transform(x, a, sigma=70):
    x = np.clip(x, 0, 255)  # Ensure x stays within valid range
    transformed = x + a * 128 * np.exp(-(x - 128) ** 2 / (2 * sigma ** 2))
    return np.clip(transformed, 0, 255).astype(np.uint8)

# Apply intensity transformation to the saturation plane
a = 0.4  # Selected a value
transformed_saturation = intensity_transform(saturation, a)

# Recombine the modified saturation with the hue and value planes
transformed_hsv = cv2.merge([hue, transformed_saturation, value])
```

4.2 Original Image and Vibrance-enhanced Image

Figure 5: Visually pleasing Image at a=0.4

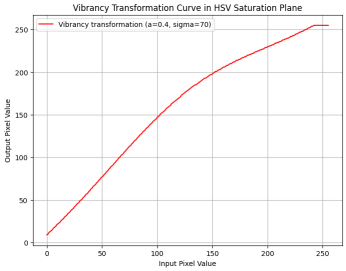
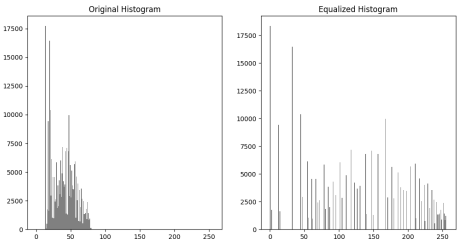


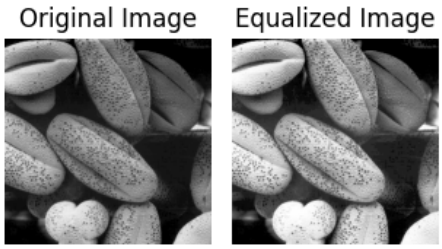
Figure 6: Vibrancy Curve

5 Histogram Equalization

Histogram equalization is an image processing technique that enhances contrast by redistributing pixel intensity values. It spreads out the most frequent intensities, improving contrast in lower-contrast areas of the image.



(a) Histograms



(b) Images

```python
def custom_histogram_equalization(img):
    hist, bins = np.histogram(img.flatten(), 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = cdf * hist.max() / cdf.max()
    cdf_m = np.ma.masked_equal(cdf, 0)  # Mask zeros
    cdf_m = (cdf_m - cdf_m.min()) * 255 / (cdf_m.max() - cdf_m.min())  # Scale
    cdf_final = np.ma.filled(cdf_m, 0).astype('uint8')  # Fill masked values with 0
    equalized_img = cdf_final[img]
    return equalized_img
```

6 Image with Histogram equalized Foreground

We apply histogram equalization to the image foreground by first splitting it into HSV planes and extracting the foreground with a binary mask. After enhancing the foreground's contrast through equalization, we merge it back with the original background.

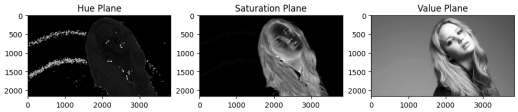
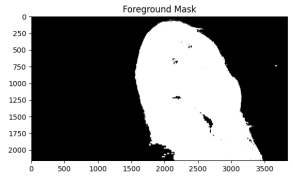


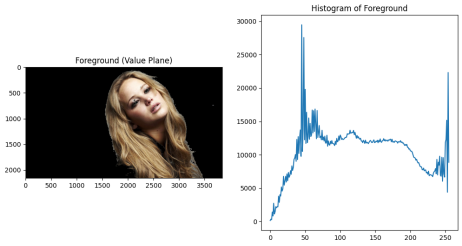
Figure 8: HSV Channels

6.1 Foreground Mask

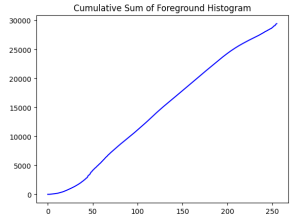
```
_, mask = cv2.threshold(saturation, 12, 255, cv2.THRESH_BINARY)# threshold=12
foreground = cv2.bitwise_and(value, value, mask=mask)

# Compute the histogram of the foreground
hist_foreground = cv2.calcHist([foreground], [0], mask, [256], [0, 256])
# Calculate the cumulative sum of the foreground histogram
cdf_foreground = np.cumsum(hist_foreground)

# Normalize the cumulative sum
cdf_normalized = cdf_foreground * hist_foreground.max() / cdf_foreground.max()
```



(a) Foreground mask threshold at 12

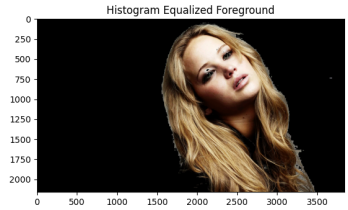


(b) Foreground mask using cv2.bitwise and histogram

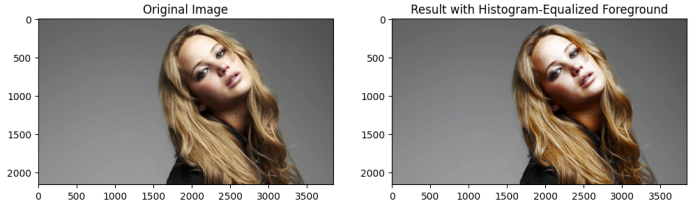


(c) Cumulative Sum Histogram

6.2 Histogram Equalization of the Foreground



(a) Histogram Equalized Foreground



(b) Extract background and combine with equalized foreground

```
cdf_m = np.ma.masked_equal(cdf_foreground, 0)
cdf_m = (cdf_m - cdf_m.min()) * 255 / (cdf_m.max() - cdf_m.min())
cdf_final = np.ma.filled(cdf_m, 0).astype('uint8')
equalized_foreground = cdf_final[foreground]

background_mask = cv2.bitwise_not(mask)
background = cv2.bitwise_and(image_rgb, image_rgb, mask=background_mask)
final_image = cv2.add(cv2.cvtColor(background,
    cv2.COLOR_BGR2RGB),equalized_foreground )
result_image = cv2.cvtColor(final_image, cv2.COLOR_BGR2RGB)
```

7 Filtering with the Sobel operator

In this task, we apply the Sobel operator to compute the gradient of an image, which helps detect edges. We first use an existing function (‘filter2D‘) to apply the Sobel filter. Then, we write our own code to perform the Sobel filtering manually. Finally, we use the separable property of the Sobel operator, breaking it into two parts (vertical and horizontal) for efficient filtering.

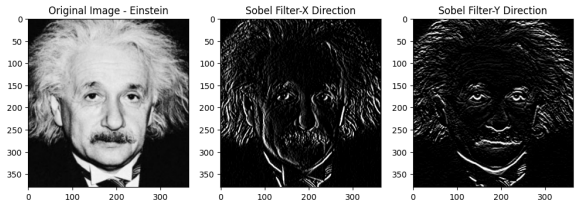
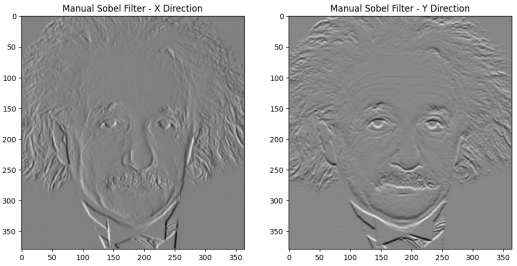
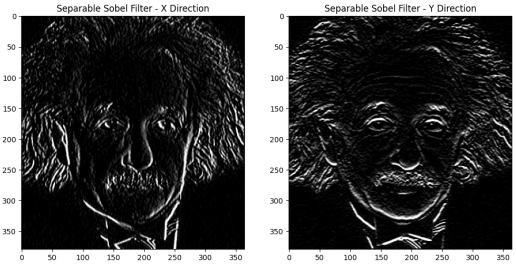


Figure 11: Sobel Filtering in X,Y direction

7.1 Manual and Seperable Sobel Filtering



(a) Manual Sobel Filtering



(b) Separable Sobel Filtering

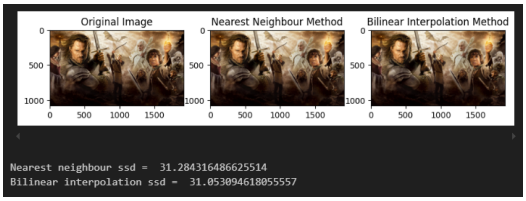
```python
#Manual Code for sobel filter
def sobel_filter(image, kernel):
    [h, w] = np.shape(image) # Get rows and columns of the image
    output= np.zeros(shape=(h, w)) # Create empty image
    for i in range(1,h-1):  # Convolve the image with the kernel
        for j in range(1, w-1):
            output[i, j] = np.sum(np.multiply(kernel,image[i-1:i + 2, j-1:j + 2]))
    return output
```

```python
#Seperable Filter
kernel_x_row = np.array([[1, 0, -1]])  # 1x3
kernel_x_col = np.array([[1], [2], [1]])  # 3x1
sobel_separable_x = cv2.filter2D(image, -1, kernel_x_row)
sobel_separable_x = cv2.filter2D(sobel_separable_x, -1, kernel_x_col)
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)
```

8 Zoom Images by a Given Factor

We use the **grabCut algorithm** to segment a flower image into foreground and background. First, we extract the flower and display the segmentation mask, foreground, and background. Then, we enhance the image by blurring the background while keeping the flower sharp. Finally, we analyze why the background near the flower edges appears darker in the enhanced image.

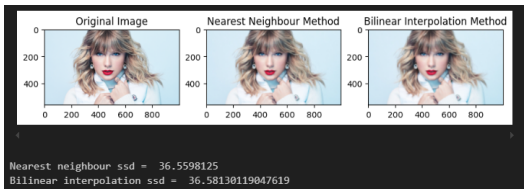
```python
if method == 'nearest':
        zoomed_image = cv2.resize(image, newsize, interpolation=cv2.INTER_NEAREST)
    elif method == 'bilinear':
        zoomed_image = cv2.resize(image, newsize, interpolation=cv2.INTER_LINEAR)

def compute_ssd(original, zoomed):
    ssd = np.sum((original - zoomed) ** 2)
    normalized_ssd = ssd / original.size
    return normalized_ssd
```



(a) Image 1

(b) Image 2

(a) Image 3



(b) Image 4

Figure 14: Merged Images

9 Flower Image with both Foreground and Background

We use the grabCut to segment a flower image into foreground and background. First, we extract the flower and display the segmentation mask, foreground, and background. Then, we enhance the image by blurring the background while keeping the flower sharp. Finally, we analyze why the background near the flower edges appears darker in the enhanced image.

9.1 Foreground,Background with Segmentation Mask

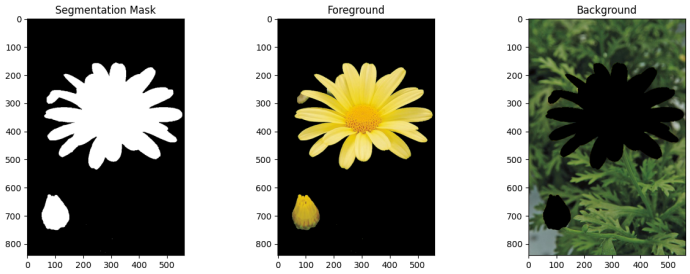


Figure 15: Foreground,Background with Segmentation Mask

```
cv2.grabCut(image, mask, rect, bg_model, fg_model, 5, cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
foreground = image_rgb * mask2[:, :, np.newaxis]#Modify mask to get the foreground
background = image_rgb * (1 - mask2[:, :, np.newaxis])# Extract the background
```

9.2 Enhanced Image with Blurred Background

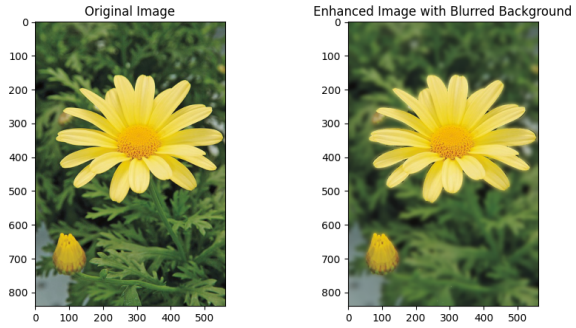


Figure 16: Enhanced Image with Blurred Background

```
blurred_background = cv2.GaussianBlur(image_rgb, (35, 35), 0)
enhanced_image = blurred_background * (1 - mask2[:, :, np.newaxis]) + foreground
```

9.3 Reasons for dark background beyond the edge of the flower

The dark background just beyond the edge of the flower in the enhanced image occurs because, during Gaussian blurring, the pixels belonging to the foreground (flower) are replaced with black (zero) in the background. As the blur averages nearby pixel values, the regions near the border mix with these black pixels, causing the areas around the edge of the flower to appear darker. This effect creates a dark halo around the object due to the sharp transition between the blurred background and the masked foreground.

You can find the assignment codes in my GitHub Repository.