Intensity Transformations and Neighborhood Filtering

EN3160 - Image Processing and Machine Vision

Balasooriya B.A.P.I.
220054N

PankajaBalasooriya/EN3160_Image_Processing_and_Machine_Vision

View Jupyter Notebook

Question 1: Basic Intensity Transformation

Piecewise intensity transformation function is constructed by linearly interpolating between the control points "c".

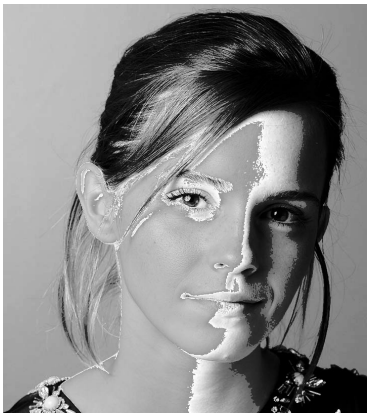
```python
c = np.array([(50, 50), (50, 100), (150, 255), (150,150)])

t1 = np.linspace(0, c[0,1], c[0,0] + 1 -0).astype('uint8')
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, 255, 255 - c[3,0]).astype('uint8')

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

img_emma_transformed = cv.LUT(img_emma, transform)
```

(a) Original Image



(b) Transformed Image

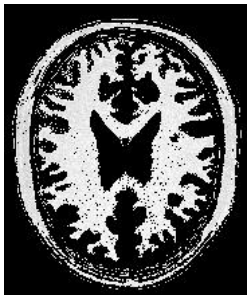
The transformation creates jump discontinuities that result in high contrast regions. Mid-intensity pixels are enhanced while preserving the extreme intensity values. This leads to improved visibility of features in the mid-range while maintaining the overall structure of the image.

Figure 1: Intensity transformation results

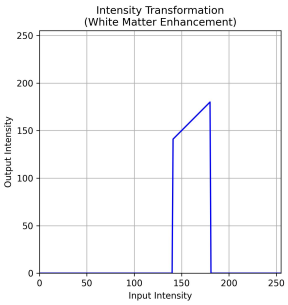
Question 2: Brain Image Enhancement

```python
# White Matter Enhancement transformation
c = np.array([(140, 0), (180, 180)])
t1 = np.zeros(c[0, 0] + 1).astype('uint8')
t2 = np.linspace(c[0, 0] + 1, c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.zeros(256 - len(t1) - len(t2)).astype('uint8')
intensity_transform_white_matter_lut = np.concatenate((t1, t2, t3), axis=0).astype('uint8')
img_brain_white_matter = cv.LUT(img_brain, intensity_transform_white_matter_lut)
```

```
# Grey Matter Enhancement transformation
c = np.array([(180, 0), (210, 210)])
t1 = np.zeros(c[0, 0] + 1).astype('uint8')
t2 = np.linspace(c[0, 0] + 1, c[1, 1], c[1, 0] - c[0, 0]).astype('uint8')
t3 = np.zeros(256 - len(t1) - len(t2)).astype('uint8')
intensity_transform_grey_matter_lut = np.concatenate((t1, t2, t3)).astype('uint8')
img_brain_grey_matter = cv.LUT(img_brain, intensity_transform_grey_matter_lut)
```

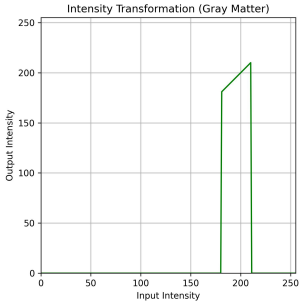


(a) White Matter Enhanced



(b) White Matter Plot

(c) Gray Matter Enhanced



(d) Gray Matter Plot

Figure 2: White and Gray matter enhancement results and transformation curves

The intensity transformations offer smooth contrast enhancement within specific intensity ranges, based on the color values of white and gray matter, effectively enhancing the visibility of these tissues.

Question 3: Gamma Correction

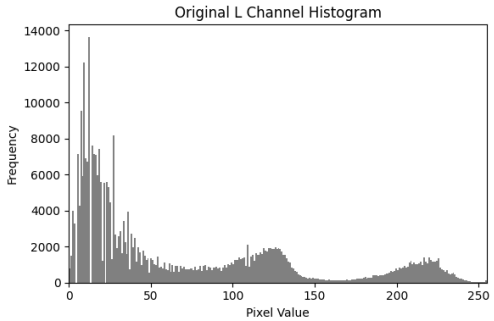
L*a*b* Color Space Conversion

```
img_lab = cv.cvtColor(img_bgr, cv.COLOR_BGR2Lab)
L, a, b = cv.split(img_lab)
```

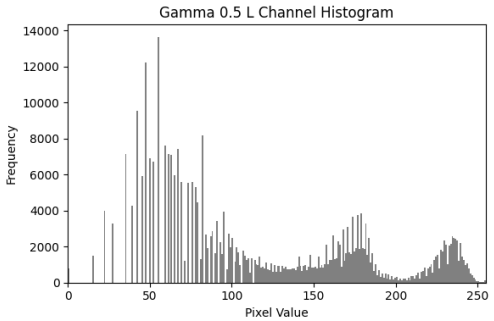
Applying γ value correction

```
L_norm = L.astype(np.float32) / 255.0
# Apply gamma correction
gamma = 0.5  # <1 brightens, >1 darkens
L_gamma = np.power(L_norm, gamma)
L_new = np.clip(L_gamma * 255.0, 0, 255).astype(np.uint8)
# Merge and convert back to BGR
img_lab_new = cv.merge((L_new, a, b))
img_bgr_new = cv.cvtColor(img_lab_new, cv.COLOR_Lab2BGR)
```

γ value is chosen as 0.5 by comparing the images by changing the values of γ, which lightens the shadows while preserving highlights, improving overall image visibility.



(a) Original L plane Histogram



(b) Gamma Corrected L plane Histogram

Question 4: Vibrance Enhancement

HSV Decomposition

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

(a) Hue

(b) Saturation

(c) Value

Figure 4: HSV plane decomposition

Vibrance Transformation

```
def transformation(x: np.ndarray, a: float, sigma: int=70) -> np.ndarray:
    x_float = x.astype(np.float64) # To stop overflow
    exponent = -1 * (x_float - 128)**2 / (2 * sigma**2)
    fx = x_float + a * 128 * np.exp(exponent)
    return np.clip(fx, 0, 255).astype(np.uint8)
```

Applying the transformation

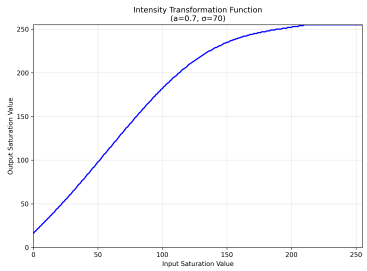
```
a_val = 0.7
sigma_val = 70
s_transformed = transformation(s, a=a_val, sigma=sigma_val)
spider_hsv_transformed = cv.merge([h, s_transformed, v])
spider_transformed_bgr = cv.cvtColor(spider_hsv_transformed, cv.COLOR_HSV2BGR)
```

A value of $a = 0.7$ was chosen to provide a balanced enhancement of image vibrance without introducing oversaturation or unnatural colors. At this level, colorful regions are noticeably intensified, improving visual appeal, while grayscale and low-saturation regions remain largely unaffected. This ensures that the transformation selectively enhances vivid areas without distorting the overall color balance, making it suitable for natural-looking vibrance enhancement.

Results Comparison

(a) Original Image

(b) Vibrance-enhanced image



(c) Intensity Transformation

Figure 5: HSV plane decomposition

Question 5: Histogram Equalization

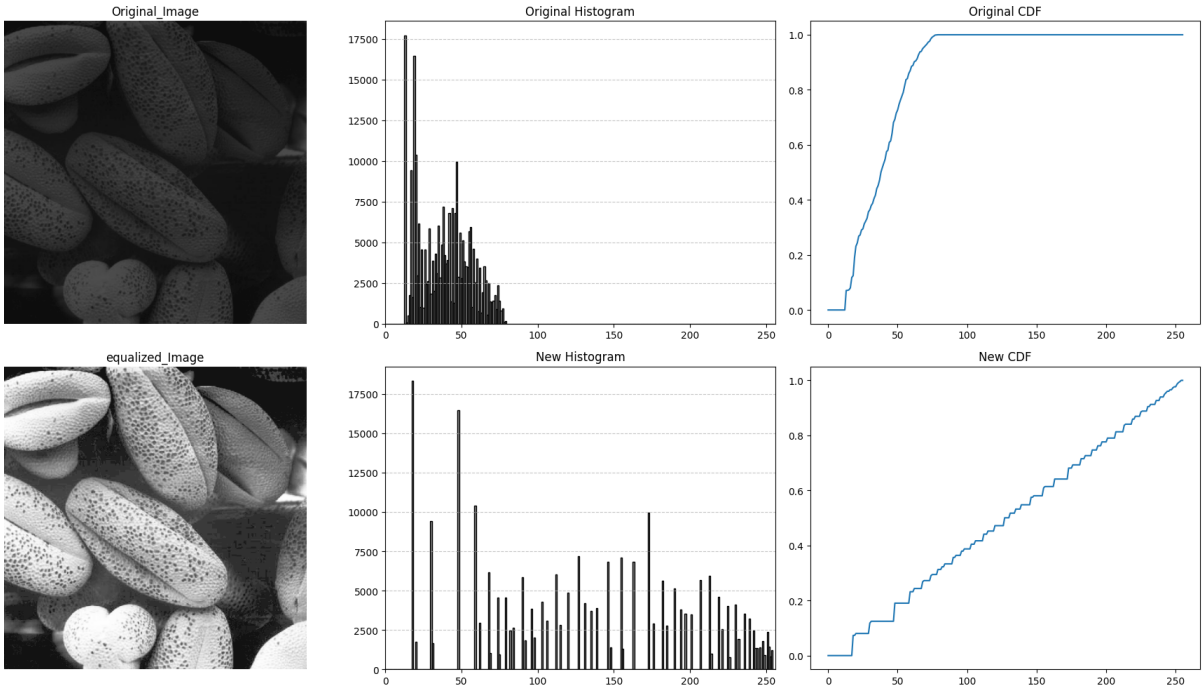
```python
def calculate_histogram(image: np.ndarray, height: int, width) -> np.ndarray:
    # Calculate histogram
    hist = np.zeros(256)
    for i in range(height):
        for j in range(width):
            hist[image[i, j]] += 1
    return hist

def calculate_cdf(image: np.ndarray, height, width):
    # Calculate cumulative distribution function (CDF)
    total_pixels = height * width
    hist = calculate_histogram(image, height, width)
    cdf = np.zeros(256)
    cdf[0] = hist[0]
    for i in range(1, 256):
        cdf[i] = cdf[i-1] + hist[i]
    # Normalize CDF
    cdf_normalized = cdf / total_pixels
    return cdf_normalized, hist

def histogram_equalization(image: np.ndarray) -> np.ndarray:
    height, width = image.shape
    total_pixels = height * width
    original_cdf, original_histogram = calculate_cdf(image, height, width)
    # Apply transformation
    equalized_image = np.zeros_like(image)
    for i in range(height):
        for j in range(width):
            equalized_image[i, j] = (255 * original_cdf[image[i, j]]).astype(np.uint8)
    return equalized_image


img = cv.imread("./a1images/shells.tif", cv.IMREAD_GRAYSCALE)
equalized_img = histogram_equalization(img, debug=True)
```

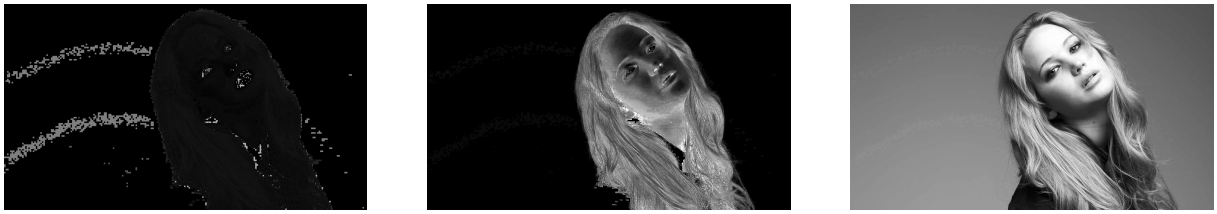
Results and Analysis



Question 6: Foreground Histogram Equalization

HSV Analysis and Masking

The Saturation channel is selected as it provides clear separation of the foreground based on color intensity, effectively isolating the subject.



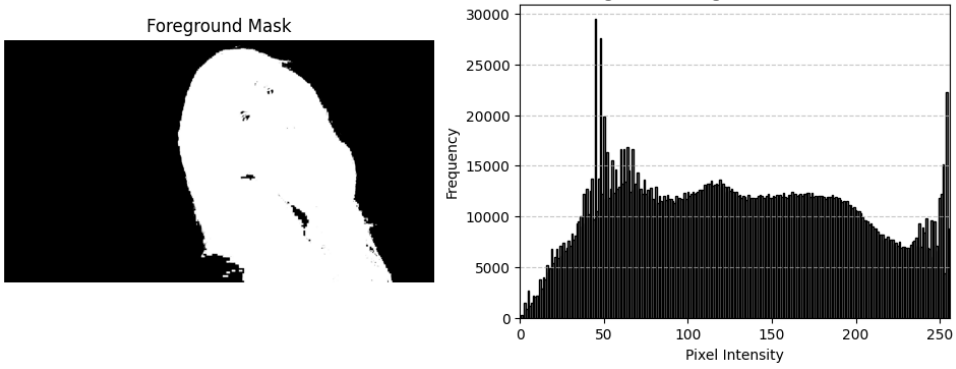
(a) Hue (b) Saturation (c) Value

Figure 6: HSV plane decomposition

Foreground Extraction

```
threshold_value = 13
_, mask = cv.threshold(s, threshold_value, 255, cv.THRESH_BINARY)

foreground = cv.bitwise_and(jeniffer_img, jeniffer_img, mask=mask)
 # Compute histogram of the value channel for foreground pixels only
foreground_v = cv.bitwise_and(v, v, mask=mask)
```



Selective Histogram Equalization

Application of histogram equalization only to the foreground.

```
cumsum = np.cumsum(hist_original)
cumsum_normalized = cumsum * 255 / cumsum[-1]   # Normalize cumulative sum

lut = np.zeros(256, dtype=np.uint8)     # Create lookup table
for i in range(256):
    lut[i] = int(cumsum_normalized[i])

# Apply histogram equalization only to foreground pixels
v_equalized = v.copy()
v_equalized[mask > 0] = lut[v[mask > 0]]

hsv_equalized = cv.merge([h, s, v_equalized])
```

Results

Selective equalization enhances foreground details while preserving the background, resulting in improved contrast and clearer distinction of the subject.

Question 7: Sobel Filtering

Method 1: Using filter2D

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

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

einstein_sobel_x = cv.filter2D(einstein_img, cv.CV_64F, sobel_x)
einstein_sobel_y = cv.filter2D(einstein_img, cv.CV_64F, sobel_y)
```

Method 2: Custom Implementation

```
def sobel_convolution(image, kernel):
    h, w = image.shape
    kh, kw = kernel.shape
    pad_h, pad_w = kh // 2, kw // 2
    padded = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    output = np.zeros_like(image, dtype=np.float64)
    for i in range(h):
        for j in range(w):
            region = padded[i:i+kh, j:j+kw]
            output[i, j] = np.sum(region * kernel)

    return output

sobel_x_manual = sobel_convolution(einstein_img, sobel_x)
sobel_y_manual = sobel_convolution(einstein_img, sobel_y)
```

Method 3: Separable Filters

```
kx = np.array([1, 0, -1], dtype=np.float32).reshape(1, 3)
ky = np.array([1, 2, 1], dtype=np.float32).reshape(3, 1)
# First filter horizontally, then vertically
temp = cv.filter2D(einstein_img, cv.CV_64F, kx)
sobel_x_sep = cv.filter2D(temp, cv.CV_64F, ky)
```

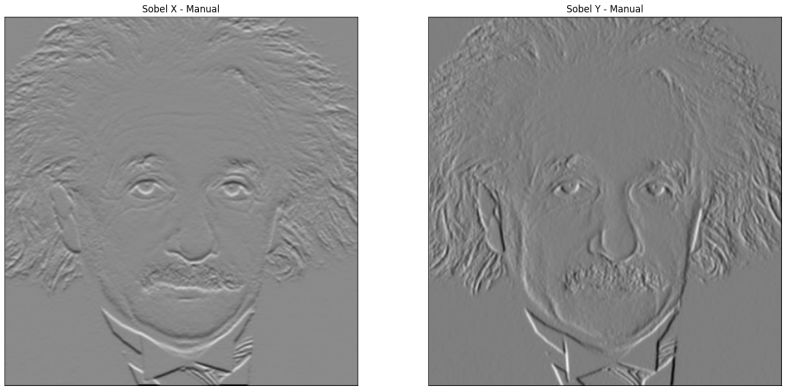


Figure 7: Sobel Filtering

Question 8: Image Zooming

0.0.1 Interpolation Methods

```python
def zoom_image(img, target_size=None, scale=None, method="nearest"):
    if method == "nearest":
        interp = cv.INTER_NEAREST
    elif method == "bilinear":
        interp = cv.INTER_LINEAR

    if target_size is not None:
        return cv.resize(img, target_size, interpolation=interp)
    elif scale is not None and (0 < scale <= 10):
        h, w = img.shape[:2]
        return cv.resize(img, (int(w * scale), int(h * scale)), interpolation=interp)

def normalized_ssd(img1, img2):
    diff = img1.astype(np.float32) - img2.astype(np.float32)
    ssd = np.sum(diff ** 2)
    norm_ssd = ssd / np.sum(img1.astype(np.float32) ** 2)
    return norm_ssd
```

Results Analysis

Bilinear interpolation produces smoother and more visually pleasing results compared to nearest neighbor. Additionally, the slightly lower SSD value for bilinear indicates a closer match to the reference image.

Question 9: Flower

```python
mask = np.zeros(img.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
rect = (50, 50, img.shape[1] - 50, img.shape[0] - 50)
# Run GrabCut
cv.grabCut(img_rgb, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_RECT)
mask2 = np.where((mask == cv.GC_FGD) | (mask == cv.GC_PR_FGD), 1, 0).astype('uint8')
foreground = img_rgb * mask2[:, :, np.newaxis]
background = img_rgb * (1 - mask2[:, :, np.newaxis])
segmentation_mask = (mask2 * 255).astype(np.uint8)
# Blur the entire image
blurred = cv.GaussianBlur(img_rgb, (25, 25), 0)
enhanced = blurred.copy()
enhanced[mask2 == 1] = img_rgb[mask2 == 1]
```

Reason for dark background at flower edges

When applying Gaussian blur to the background, the kernel averages surrounding pixels. Near the flower edges, some pixels are zeroed out from foreground extraction, causing the averaged values to be darker than the original background.