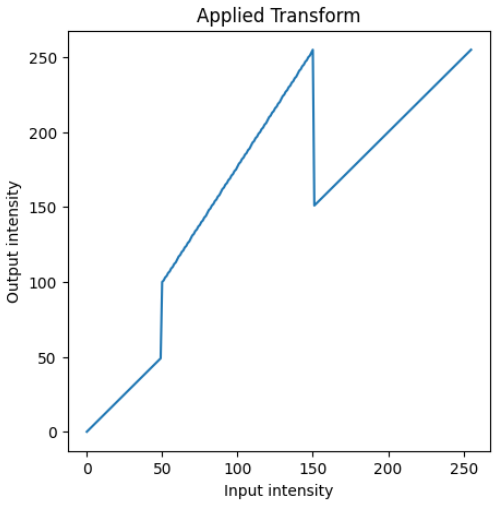
EN3160 – Image Processing and Machine Vision

Assignment 1 - Intensity Transformations and Neighborhood Filtering

Name : A. C. Pasqual
Index No. : 200445V
Submission Date : 31st August 2023
GitHub Repository: https://github.com/Anuki16/EN3160-assignment1

Question 1 – Implementing an intensity transformation

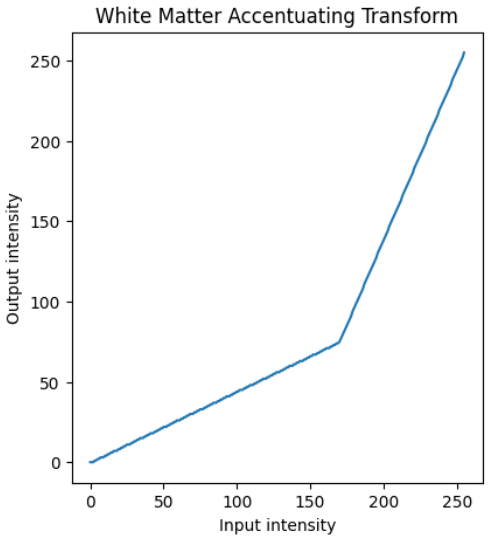
```
x_start, x_end = 50, 150
y_start, y_end = 100, 255

# Create transformation
transform = np.arange(0, 256).astype(np.uint8)
transform[x_start: x_end+1] = np.linspace(y_start, y_end, (x_end - x_start + 1), np.uint8)

# Apply transformation
transformed_img1 = transform[img1]
```

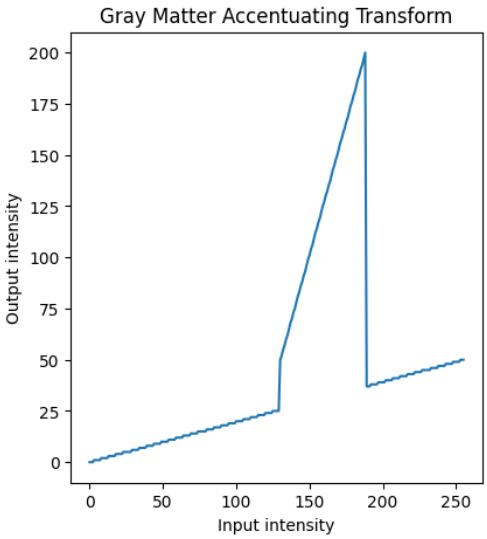


Question 2 – Enhancing parts of an image using intensity transformations

```
# White matter
x_midpoint = 170
y_midpoint = 75

white_transform = np.arange(0, 256).astype(np.uint8)
white_transform[0:x_midpoint + 1] = np.linspace(0, y_midpoint, x_midpoint + 1, np.uint8)
white_transform[x_midpoint:256] = np.linspace(y_midpoint, 255, (256 - x_midpoint), np.uint8)

# Grey matter
x1, x2 = 130, 188
y1, y2 = 50, 200

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), np.uint8)
```





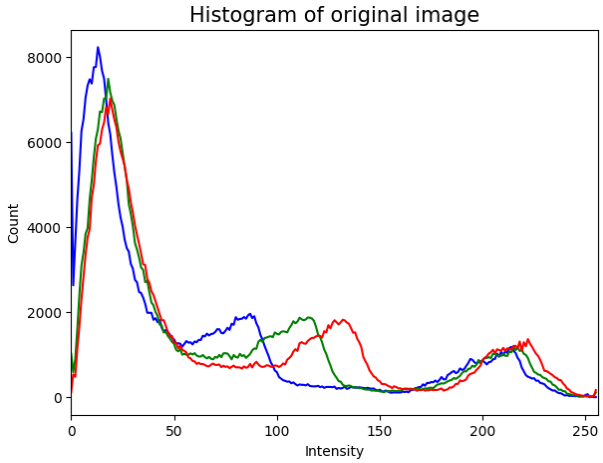
To accentuate required parts of the image, the relevant intensity ranges are mapped to a larger and brighter intensity range compared to the rest of the image.

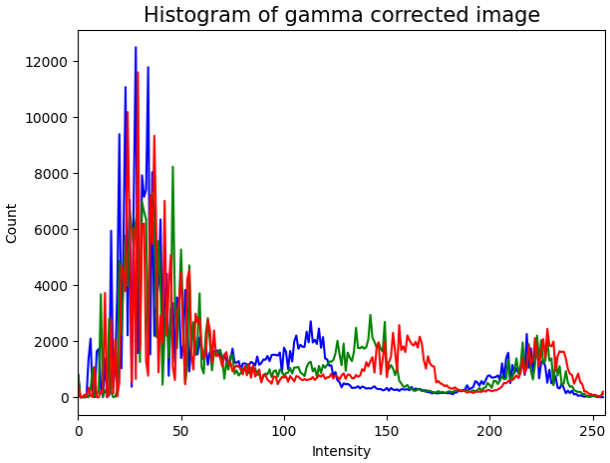
Question 3 – Applying gamma correction

```
gamma = 0.7
gamma_transform = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)])
gamma_transform = gamma_transform.astype('uint8')

img3 = cv.imread( "images/highlights_and_shadows.jpg", cv.IMREAD_COLOR)
img3_lab = cv.cvtColor(img3, cv.COLOR_BGR2LAB)  # Convert to LAB color space
# In the LAB colour space, the L plane encodes brightness only

img3_lab[:, :, 0] = gamma_transform[img3_lab[:, :, 0]]  # Apply transform only to L plane
```

γ was adjusted until the textures of the rock were clearly visible.

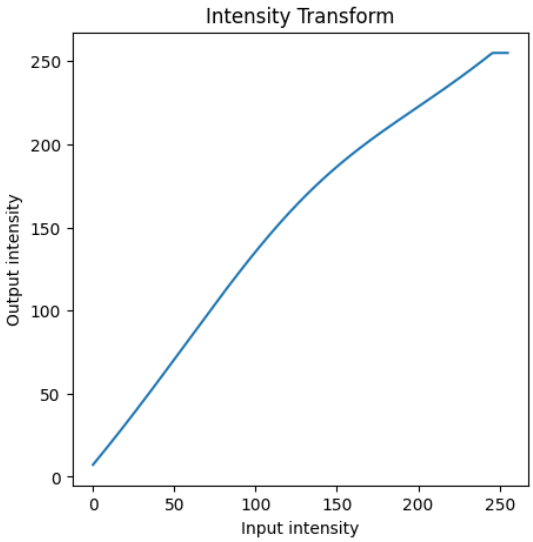

```python
# Function for computing and plotting histograms for all 3 color planes
def histBGR(img):
    # Define colors (b for blue, g for green, r for red)
    colors = ('b', 'g', 'r')

    # Loop over color channels and calculate histograms
    for i, color in enumerate(colors):
        hist = cv.calcHist([img], [i], None, [256], [0, 256])
        plt.plot(hist, color=color)
        plt.xlim([0, 256])
```





Question 4 – Vibrance enhancement

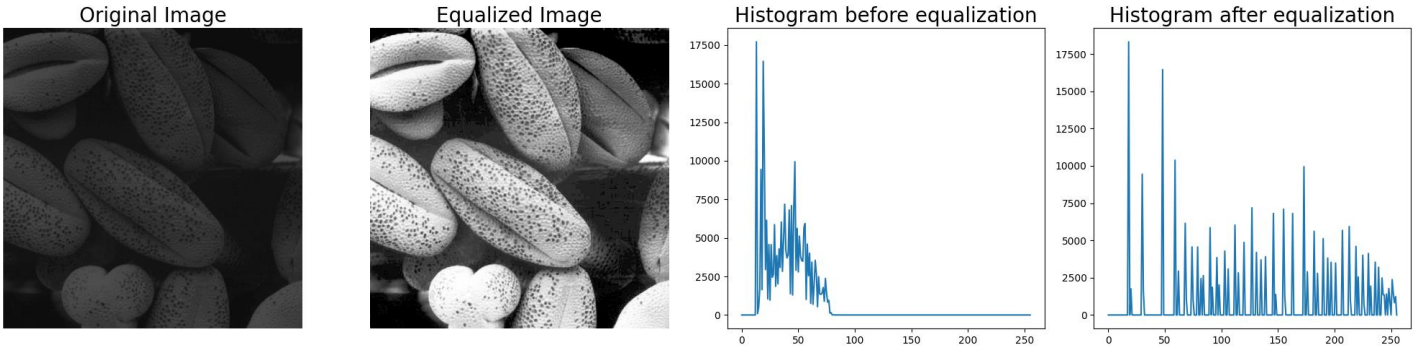
```python
a = 0.3
sigma = 70
def f(x):   # Transformation function
    return np.minimum(255, x + (a*128) * np.exp(-(x - 128)**2 / (2 * sigma**2)))

img4 = cv.imread( "images/spider.png", cv.IMREAD_COLOR)
img4_hsv = cv.cvtColor(img4, cv.COLOR_BGR2HSV)          # Convert to HSV planes
img4_hsv[:, :, 1] = f(img4_hsv[:, :, 1])    # Apply transformation only to saturation plane
```



When a gets closer to 1, the image becomes excessively vibrant. a = 0.3 gave a suitable vibrance level for the image to remain natural.

Question 5 – Histogram equalization

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

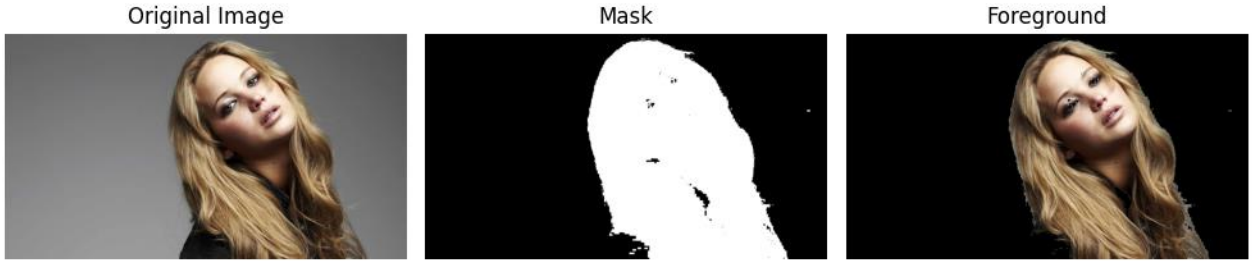


Question 6 – Histogram equalizing the foreground of an image

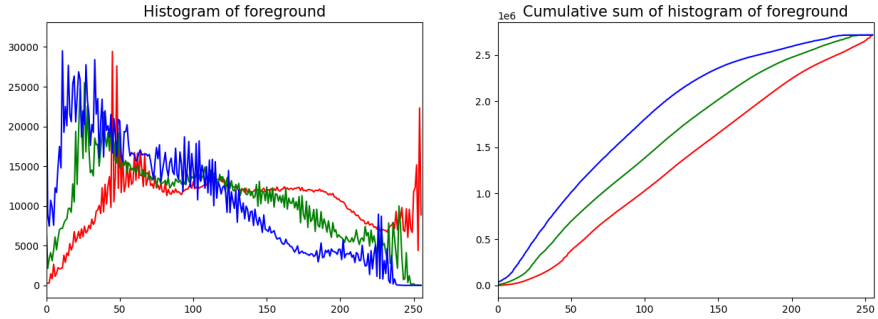
```
# Separating into HSV planes
img6_hsv = cv.cvtColor(img6, cv.COLOR_BGR2HSV)
hue_plane = img6_hsv[:, :, 0]
saturation_plane = img6_hsv[:, :, 1]
value_plane = img6_hsv[:, :, 2]
```

In the saturation plane, the background is clearly darker than the foreground. Therefore, this plane was selected for extracting the foreground using a threshold.

```
threshold = 12  # This was adjusted until the separation was satisfactory
mask = (saturation_plane > threshold).astype(np.uint8) * 255
mask_3d = np.repeat(mask[:, :, None], 3, axis=2)

foreground_hsv = np.bitwise_and(img6_hsv, mask_3d)  # obtaining the foreground
foreground_rgb = cv.cvtColor(foreground_hsv, cv.COLOR_HSV2RGB)   # Convert back to RGB
```




```
# Loop over color channels and calculate and plot histograms
for i, color in enumerate(colors):
    # Consider only foreground by giving mask as an argument
    hist = cv.calcHist([foreground_rgb], [i], mask, [256], [0, 256])
    ax[0].plot(hist, color=color)
    cumulative = np.cumsum(hist)
    ax[1].plot(cumulative, color=color)

    transform = cumulative * 255 / cumulative[-1]
    equalized_foreground[:, :, i] = transform[foreground_rgb[:, :, i]]

# Remove background again after equalization
equalized_foreground = np.bitwise_and(equalized_foreground, mask_3d)
```



```
background_mask_3d = 255 - mask_3d
background_hsv = np.bitwise_and(img6_hsv, background_mask_3d)   # Extract background
background_rgb = cv.cvtColor(background_hsv, cv.COLOR_HSV2RGB)
final_image = background_rgb + equalized_foreground     # Add with foreground
```

Equalized Foreground

Final result with original background

Question 7 – Sobel vertical filtering

```
# Custom function for filtering
def 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 = cv.normalize(image.astype('float'), None, 0, 1, cv.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   # Undo normalization
    result = np.minimum(255, np.maximum(0, result)).astype(np.uint8) # Limit between 0 and 255
    return result
```

```
# Sobel vertical kernel
kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
img7_a = cv.filter2D(img7, -1, kernel)  # Using filter2D
img7_b = filter(img7, kernel)   # Using custom function
```

```
kernel1 = np.array([1, 2, 1]).reshape((3, 1))
kernel2 = np.array([1, 0, -1]).reshape((1, 3))

def filter_in_steps(image, kernel1, kernel2): # Using convolution property
    image_float = cv.normalize(image.astype('float'), None, 0, 1, cv.NORM_MINMAX)
    result = filter_step(filter_step(image_float, kernel1), kernel2)
    result = result * 255
    result = np.minimum(255, np.maximum(0, result)).astype(np.uint8) # Limit between 0 and 255
    return result
```

Question 8 – Zooming an image

```
def interpolate(image, indices, type):
    if type == 'nn':  # Nearest neighbor interpolation
        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 type == 'bi':  # Bilinear interpolation
        floors = np.floor(indices).astype(np.uint64)
        ceils = floors + 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]]

        # Repeat indices for the 3 color planes
        indices = np.repeat(indices[:, :, :, None], 3, axis=3)
        ceils = np.repeat(ceils[:, :, :, None], 3, axis=3)
        floors = np.repeat(floors[:, :, :, None], 3, axis=3)

        # Find the horizontal midpoints
        m1 = p1 * (ceils[1] - indices[1]) + p2 * (indices[1] - floors[1])
        m2 = p3 * (ceils[1] - indices[1]) + p4 * (indices[1] - floors[1])
        # Find the vertical midpoint of horizontal midpoints
        m = m1 * (ceils[0] - indices[0]) + m2 * (indices[0] - floors[0])
        return m.astype(np.uint8)

def zoom(image, factor, interpolation = 'nn'):
    h, w, _ = image.shape
    zoom_h, zoom_w = round(h * factor), round(w * factor)   # New dimensions
    zoomed_image = np.zeros((zoom_h, zoom_w, 3)).astype(np.uint8)

    zoomed_indices = np.indices((zoom_h, zoom_w)) / factor
    zoomed_image = interpolate(image, zoomed_indices, interpolation)
    return zoomed_image
```

As seen in the following images, bilinear interpolation provided better results than nearest neighbor interpolation, but both zoomed images are significantly less sharp than the original image.


```
Normalized sum of squared differences
Image 01: Nearest neighbors = 0.0006168664775379207,    Bilinear = 0.0006037221557710472
Image 02: Nearest neighbors = 0.00025825407028279724,   Bilinear = 0.00024931605721446165
Image 04: Nearest neighbors = 0.0012573230016836275,    Bilinear = 0.00125580531543641
Image 05: Nearest neighbors = 0.0008397238286200464,    Bilinear = 0.0008259538420942087
```

Question 9 - Segmentation

```
mask = np.zeros(img9.shape[:2], np.uint8)   # Initial mask
mask[150:550, 50:600] = cv.GC_PR_FGD    # Give inner region as probably foreground
mask[300:410, 220:380] = cv.GC_FGD      # Give flower center as foreground to avoid holes
bgdModel = np.zeros((1, 65),np.float64)
fgdModel = np.zeros((1, 65),np.float64)
rect = (50, 150, (612 - 50), (600 - 150))

cv.grabCut(img9, mask, rect, bgdModel, fgdModel, 5, cv.GC_INIT_WITH_MASK)
# Select pixels that are background or probably background as 0 and others as 1
foreground_mask = np.where((mask==2)|(mask==0), 0, 1).astype('uint8')
background_mask = 1 - foreground_mask

foreground_img9 = img9 * foreground_mask[:, :, np.newaxis]
background_img9 = img9 * background_mask[:, :, np.newaxis]

blurred_background = cv.GaussianBlur(background_img9, (51, 51), 5)
blurred_background = blurred_background * background_mask[:, :, np.newaxis]
final_img9 = blurred_background + foreground_img9
```

The dark edge in the enhanced image is because the masked black pixels in the background image get averaged with the background pixels at the border of the mask when Gaussian blur is applied.