**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

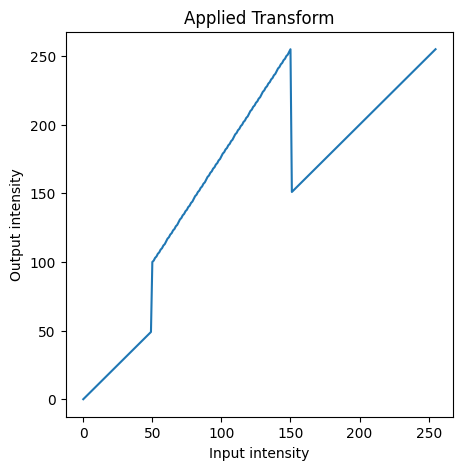
transform = np.arange(0, 256).astype(np.uint8)

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]

A comparison of a person's face

Description automatically generated

**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

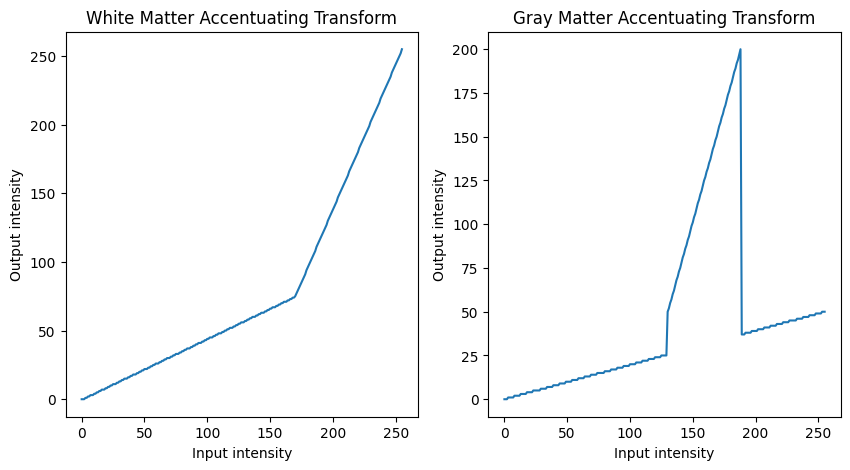
x1, x2 = 130, 188

y1, y2 = 50, 200

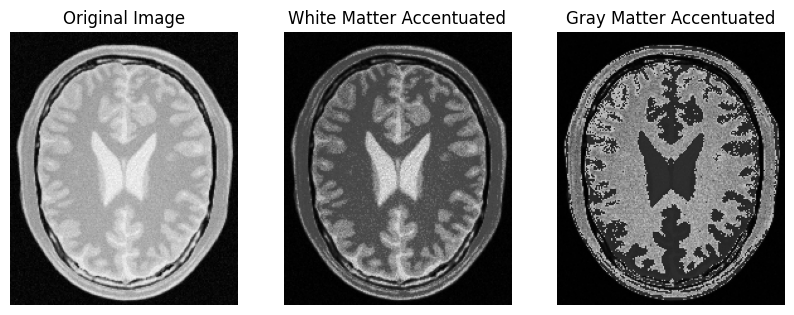
grey\_transform = np.linspace(0, 50, 256)

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')

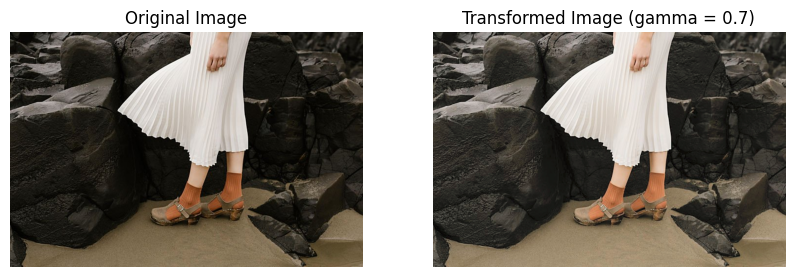
img3 = cv.imread( "images/highlights\_and\_shadows.jpg", cv.IMREAD\_COLOR)

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.



# 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')

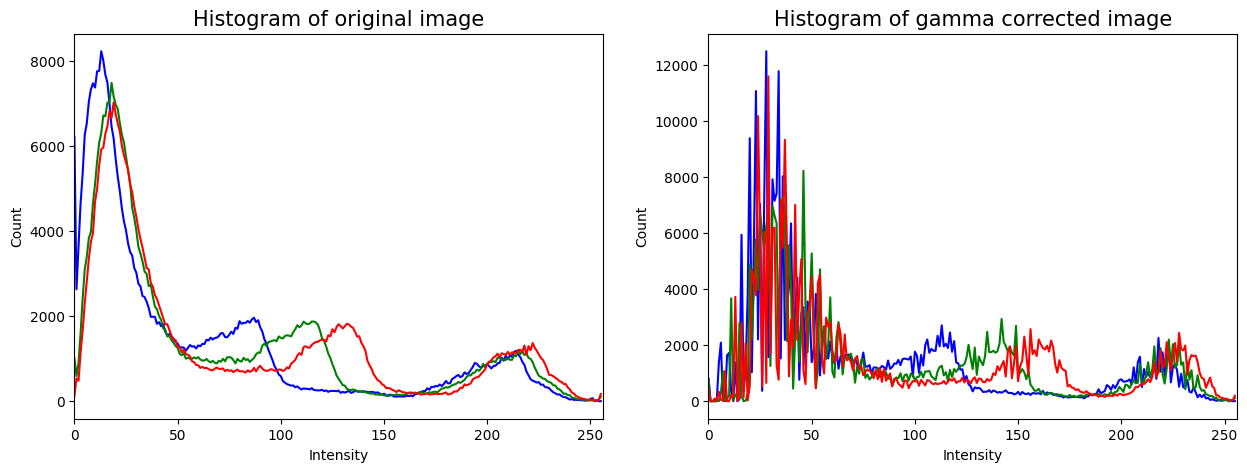
    # Loop over color channels and calculate histograms

    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**

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

A graph with a line

Description automatically generatedA person and person in a garment

Description automatically generated

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]

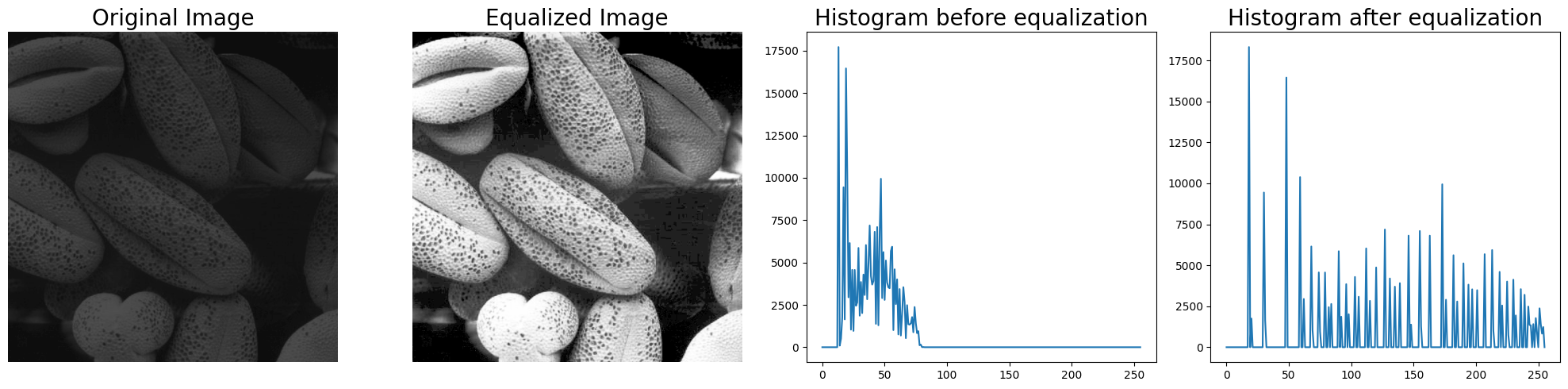
    hist, bins = np.histogram(image.ravel(), 256, [0, 256])

    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**

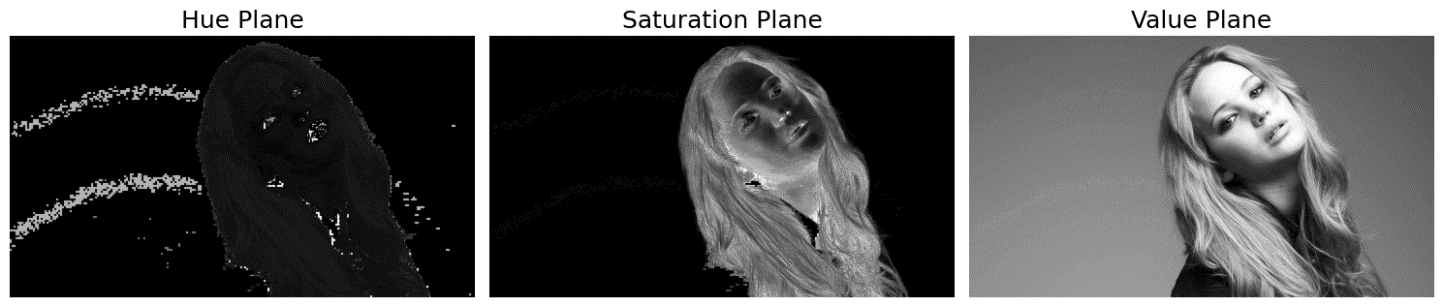
# Separating into HSV planes

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)

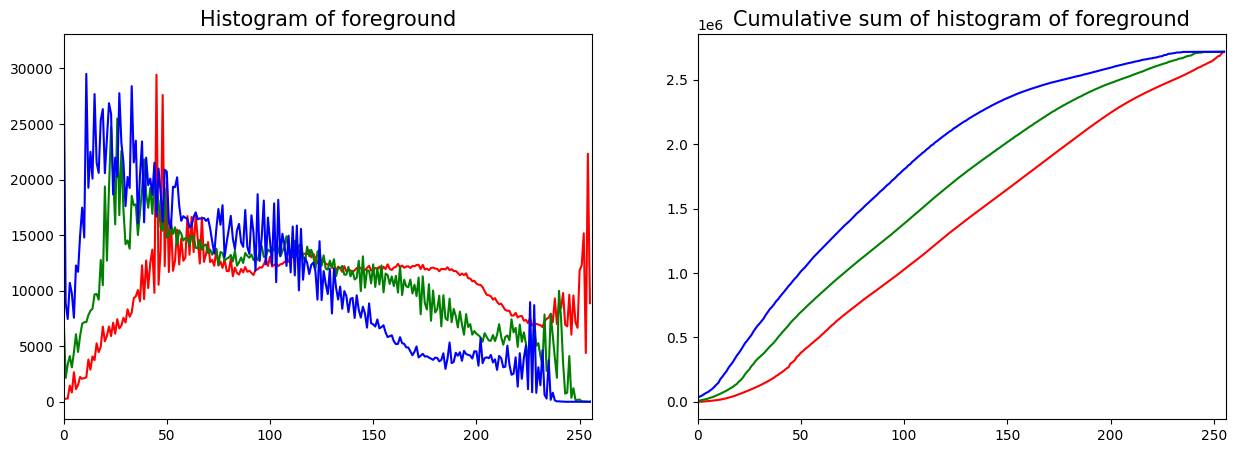
    ax[1].plot(cumulative, color=color)

    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)

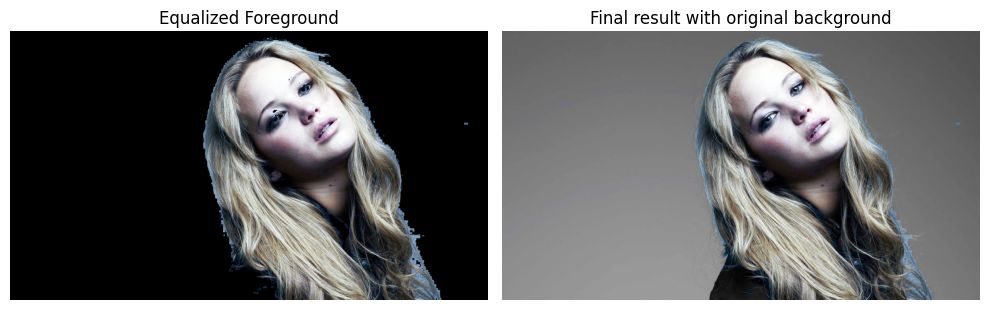
****

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

****

**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.

****

**A screenshot of a computer

Description automatically generated**

**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

**A collage of different flowers

Description automatically generated**

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.