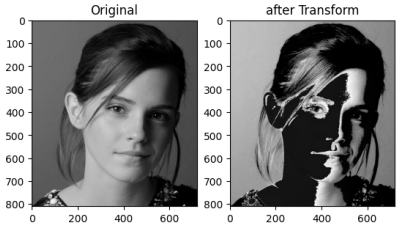
Ransika L.G.C.
220514C

EN3160: Intensity Transformations and Neighborhood Filtering

1. Implement the intensity transformation depicted in Fig. 1 on the image

```
# for section 1 - linear transformation with unit gradient
section1 = img < 50
output[section1] = img[section1]
#section 2
section2= (img >= 50) & (img < 150)
output[section2]= (15.5 * img[section2]) + 50
# for section three - linear transformation with unit gradient
section3 = (img >= 150) & (img < 255)
output[section3]= img[section3]
```



2. Apply a similar operation as above (question 1) to accentuate (a) white matter and (b) gray matter

White Matter (WM) appears brighter in T1-weighted MRI (high intensity values). Gray Matter (GM) appears darker than WM but brighter than CSF (medium intensity values). We can threshold or remap these intensity ranges to make them stand out.

```
# Normalize image for consistent scaling to the range 0-255
img_norm = cv2.normalize(img2, None, 0, 255, cv2.NORM_MINMAX)
# White Matter mask (bright intensities, e.g., 180  255)
wm_mask = cv2.inRange(img_norm, 180, 255)
# Gray Matter mask (medium intensities, e.g., 100  179)
gm_mask = cv2.inRange(img_norm, 100, 179)

# Apply masks (accentuate by making them brighter)
wm_highlight = cv2.bitwise_and(img_norm, img_norm, mask=wm_mask)
gm_highlight = cv2.bitwise_and(img_norm, img_norm, mask=gm_mask)

# Optionally enhance contrast
wm_highlight = cv2.convertScaleAbs(wm_highlight, alpha=1.5, beta=0)
gm_highlight = cv2.convertScaleAbs(gm_highlight, alpha=1.5, beta=0)
```

The mask determines which output pixels are calculated and which are set to zero. Only pixels where mask = 255 are taken from (img and img). All other positions become black (intensity 0).

3. Apply gamma correction to the L plane in the Lab color space and state the γ value.

What is the L*a*b Color Space? The L*a*b (CIELAB) color space is designed to be **perceptually uniform** meaning changes in values correspond more closely to how humans perceive color differences. It separates **lightness** from

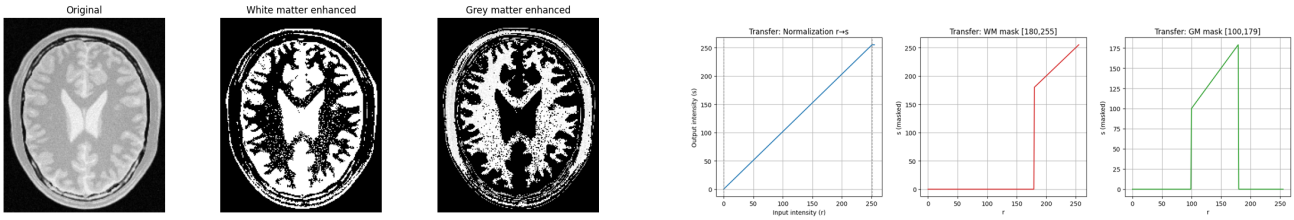
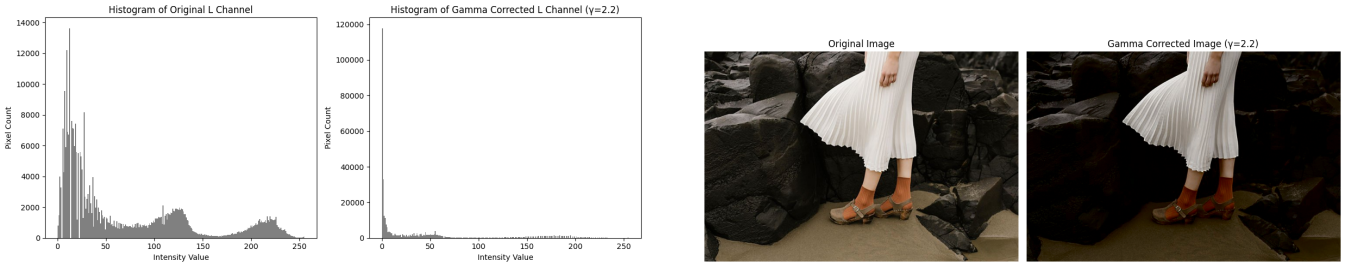


Figure 1: Enter overall caption here describing both images.

color information, which makes it very useful for image processing tasks.

Why Apply Gamma Correction Only on the L Channel? The L channel contains the lightness (brightn information separated from color so it makes sense to apply brightness adjustments there.

```
# Convert BGR to Lab
lab_img = cv2.cvtColor(img3, cv2.COLOR_BGR2LAB)
# Split the Lab image into L, a, b channels
L, a, b = cv2.split(lab_img)
#Normalize only L channel to [0,1] for gamma correction
L_norm = L / 255.0
# choose the gamma value = 2.2
gamma = 2.2
L_gamma = np.power(L_norm, gamma)
L_corrected = np.uint8(L_gamma * 255)
lab_corrected = cv2.merge((L_corrected, a, b)) #merge
result = cv2.cvtColor(lab_corrected, cv2.COLOR_LAB2BGR)
```



4. Increasing the vibrance of a photograph is probably achieved by applying an intensity transformation such as

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

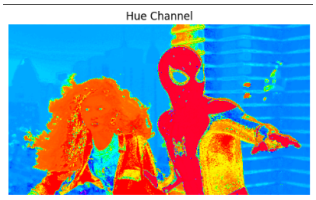
to the saturation plane, where x is the input intensity, a [0,1] and = 70.

(a) Split the image into hue, saturation, and value planes.

```
# Convert the image from BGR to HSV color space
hsv_img = cv2.cvtColor(img4, cv2.COLOR_BGR2HSV)
# Split the HSV image into individual channels
h, s, v = cv2.split(hsv_img)
```

(b) Apply the aforementioned intensity transformation to the saturation plane.

```
# Parameters
a = [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
sigma = 70.0

for i in a:
    # Calculate the Gaussian term
    gaussian_term = i * 128 * np.exp(-((s.astype(np.float32) - 128) ** 2) / (2 * (sigma
        ** 2)))
    # Apply the function f(x) = min(x + gaussian_term, 255)
    s_transformed = s.astype(np.float32) + gaussian_term
    s_transformed = np.clip(s_transformed, 0, 255).astype(np.uint8)
    # Recombine three channels
    hsv_transformed = cv2.merge([h, s_transformed, v])
    # Convert back to BGR to display
    result = cv2.cvtColor(hsv_transformed, cv2.COLOR_HSV2BGR)
```

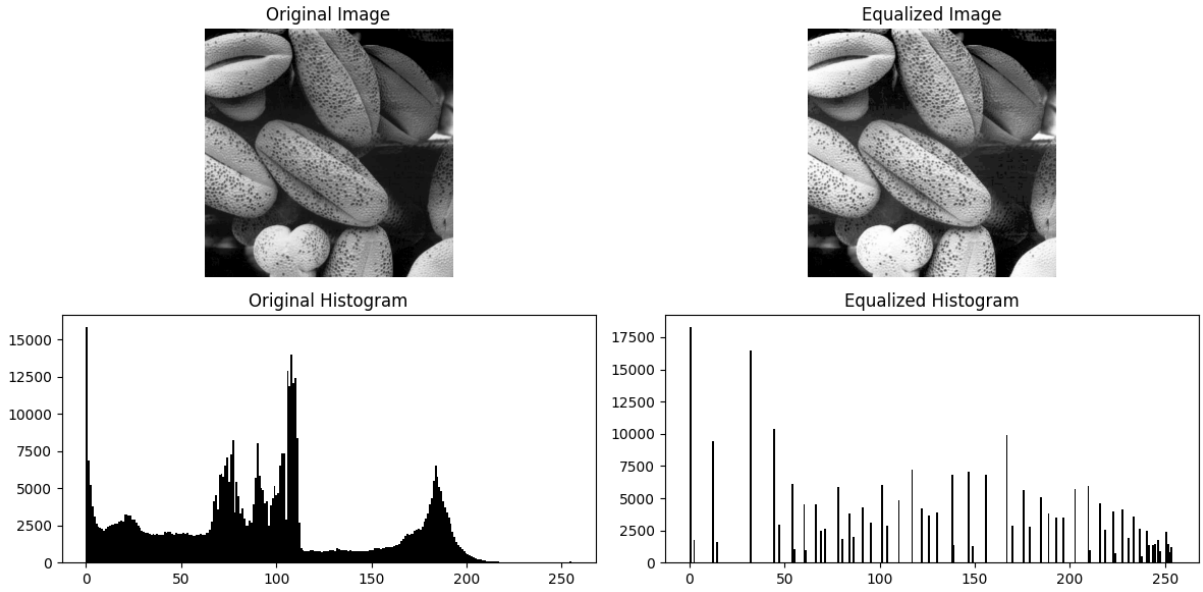


5. Write a function of your own to carry out histogram equalization on the image shown in Fig. 5. Show the histograms before and after equalization.

```
def histogram_equalization(image):
    # Flatten image array and calculate histogram
    hist, bins = np.histogram(image.flatten(), 256, [0,256])
    # Calculate the cumulative distribution function (CDF)
    cdf = hist.cumsum()
    # Mask all zero values (to avoid division by zero)
    cdf_masked = np.ma.masked_equal(cdf, 0)
    # Normalize the CDF
    cdf_min = cdf_masked.min()
    cdf_max = cdf_masked.max()
    cdf_norm = (cdf_masked - cdf_min) * 255 / (cdf_max - cdf_min)
    # Fill masked values with zero
    cdf_final = np.ma.filled(cdf_norm, 0).astype('uint8')
    # Map the original image pixels to equalized values using the normalized CDF as a
        lookup table
    img_equalized = cdf_final[image]
    return img_equalized
```

6. In this question, we will apply histogram equalization only to the foreground of an image to produce an image with a histogram equalized foreground

(a)Open the image in Fig. 6, split it into hue, saturation, and values and display these planes in grayscale




```
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
h, s, v = cv2.split(hsv)
```

(b)Select the appropriate plane to threshold in extract the foreground mask. A mask is a binary image.(c) Now obtain the foreground only using cv.bitwise_and and compute the histogram

```
# saturation threshold (keep moderately high saturation as foreground)
s_lower, s_upper = 60, 255
mask_s = cv2.inRange(s, s_lower, s_upper)  # binary mask from S

# Optionally refine using morphology to clean small noise and fill small holes
kernel = np.ones((3,3), np.uint8)
mask_bin = cv2.morphologyEx(mask_s, cv2.MORPH_OPEN, kernel, iterations=1)
mask_bin = cv2.morphologyEx(mask_bin, cv2.MORPH_CLOSE, kernel, iterations=1)
```

```
# Extract foreground from RGB (visualization) and V (for intensity histogram)
foreground_rgb = cv2.bitwise_and(img_rgb, img_rgb, mask=mask_bin)

# Choose intensity channel to analyze for histogram equalization:
# Typically use the Value (V) channel in HSV for brightness equalization.
foreground_v = cv2.bitwise_and(v, v, mask=mask_bin)

# Compute histogram only over foreground pixels
# Note: foreground_v already zeros out background; we can hist over the non-zero mask
hist_fg, bins = np.histogram(foreground_v[mask01==1], bins=256, range=(0,256))
```

(d) Obtain the cumulative sum of the histogram using np.cumsum.

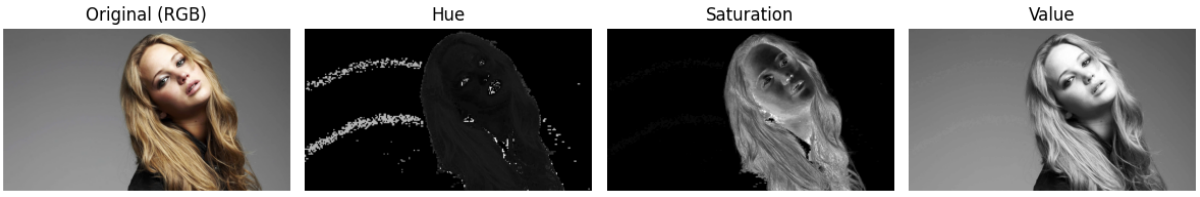
```
cdf_fg = np.cumsum(hist_fg)  # length 256
# Normalize CDF to [0, 1]
cdf_min = cdf_fg[np.nonzero(cdf_fg)].min() if np.any(cdf_fg) else 0
N_fg = cdf_fg[-1] if len(cdf_fg) > 0 else 1  # total foreground pixels
```

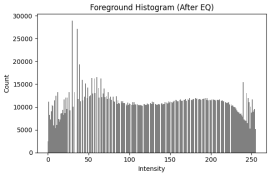
(e) Use the formulas in slides to histogram-equalize the foreground

```
L = 256
# Build mapping for 0..255
equal_map = np.zeros(256, dtype=np.uint8)
if N_fg > 0 and cdf_min < N_fg:
    equal_map = np.round((cdf_fg - cdf_min) / (N_fg - cdf_min) * (L - 1)).clip(0, 255).
        astype(np.uint8)

# Apply mapping only to foreground pixels on the V channel
v_eq = v.copy()
# Map: for foreground pixels, replace v by equalized value, else keep original v
# To avoid touching background, we only remap at mask==1
v_eq[mask01==1] = equal_map[v[mask01==1]]

# Histogram after equalization for foreground pixels
hist_fg_eq, _ = np.histogram(v_eq[mask01==1], bins=256, range=(0,256))
```

(f)Extract the background and add with the histogram equalized foreground.

```
hsv_eq = hsv.copy()
hsv_eq[..., 2] = v_eq

# Convert back to RGB for visualization
img_eq_rgb = cv2.cvtColor(hsv_eq, cv2.COLOR_HSV2RGB)

# Alternatively, explicitly recomposite using the mask:
# - Foreground from img_eq_rgb (equalized V)
# - Background from original img_rgb (unchanged)
foreground_eq_rgb = (img_eq_rgb * mask01[..., None]).astype(np.uint8)
background_rgb = (img_rgb * (1 - mask01[..., None])).astype(np.uint8)
recomposed_rgb = (foreground_eq_rgb + background_rgb).astype(np.uint8)
```



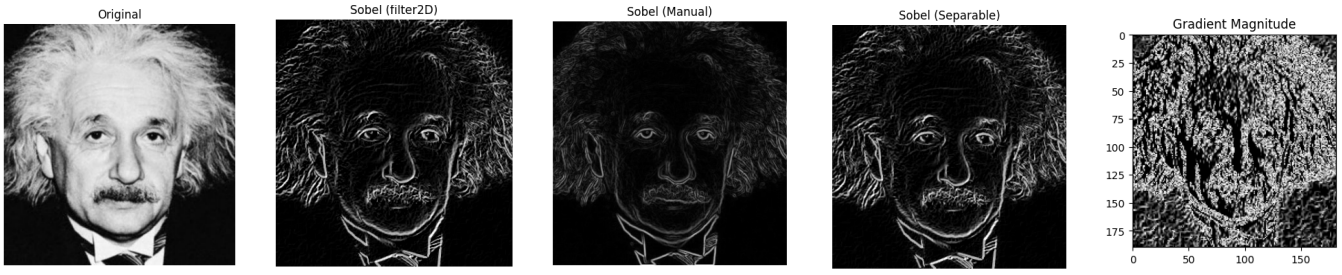


7. Filtering with the Sobel operator can compute the gradient.

(a) Using the existing filter2D to Sobel filter the image.

(b) Write your own code to Sobel filter the image

```python
# Sobel X separable filters
#Sobel filter using cv2.filter2D
sobel_x_kernel = np.array([[1, 0, -1],
                           [2, 0, -2],
                           [1, 0, -1]], dtype=np.float32)
sobel_y_kernel = np.array([[1, 2, 1],
                           [0, 0, 0],
                           [-1, -2, -1]], dtype=np.float32)
sobel_x_a = cv2.filter2D(img7, -1, sobel_x_kernel)
sobel_y_a = cv2.filter2D(img7, -1, sobel_y_kernel)
sobel_a = cv2.magnitude(sobel_x_a.astype(np.float32), sobel_y_a.astype(np.float32))
# My Sobel filter
def my_convolution(image, kernel):
    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.float32)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            region = padded[i:i+kh, j:j+kw]
            output[i, j] = np.sum(region * kernel)
    return output
sobel_x_b = my_convolution(img7, sobel_x_kernel)
sobel_y_b = my_convolution(img7, sobel_y_kernel)
sobel_b = np.sqrt(sobel_x_b**2 + sobel_y_b**2)
# Separable convolution
gx_row = np.array([[1, 0, -1]], dtype=np.float32)  # horizontal derivative
gx_col = np.array([[1], [2], [1]], dtype=np.float32)  # smoothing vertically
# X-direction
temp_x = cv2.filter2D(img7, -1, gx_row)
sobel_x_c = cv2.filter2D(temp_x, -1, gx_col)
# Y-direction (transpose order)
gy_row = np.array([[1, 2, 1]], dtype=np.float32)
gy_col = np.array([[1], [0], [-1]], dtype=np.float32)
temp_y = cv2.filter2D(img7, -1, gy_col)
sobel_y_c = cv2.filter2D(temp_y, -1, gy_row)
sobel_c = cv2.magnitude(sobel_x_c.astype(np.float32), sobel_y_c.astype(np.float32))
```



8. Write a program to zoom images by a given factor s (0,10]. You must use a function to zoom the image,which can handle (a) nearest-neighbor, and (b) bilinear interpolation.

```python
def zoom_image(img, scale_factor, method='nearest'):
```

```
    height, width = img.shape[:2]
    new_height, new_width = int(height * scale_factor), int(width * scale_factor)
    if method == 'nearest':
        interp = cv2.INTER_NEAREST
    elif method == 'bilinear':
        interp = cv2.INTER_LINEAR
    else:
        raise ValueError("Method must be 'nearest' or 'bilinear'")
    return cv2.resize(img, (new_width, new_height), interpolation=interp)
```

```
def normalized_ssd(img1, img2):
    # Ensure both are float for computation
    img1 = img1.astype(np.float32)
    img2 = img2.astype(np.float32)
    # If different sizes, compute over overlapping region
    h = min(img1.shape[0], img2.shape[0])
    w = min(img1.shape[1], img2.shape[1])
    img1_c = img1[:h, :w]
    img2_c = img2[:h, :w]
    diff = img1_c - img2_c
    ssd = np.sum(diff * diff)
    denom = np.sum(img1_c * img1_c) + 1e-12  # avoid divide-by-zero
    return ssd / denom
```

For image 1 - Normalized SSD (nearest neighbor): 0.012057948510772873 and Normalized SSD (bilinear): 0.010184055645914382

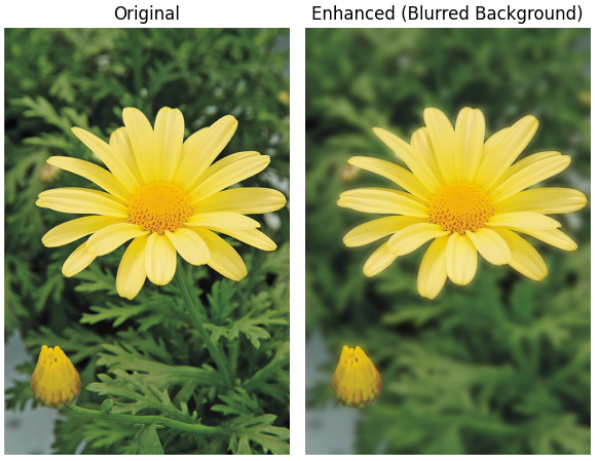
9. (a) Use grabCut to segment the image. Show the final segmentation mask, foreground image, and background image.

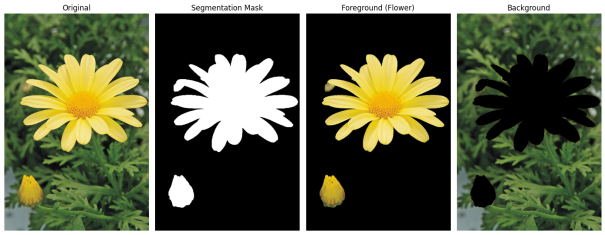
```
# rectangle init
rect = tuple(map(int, (0.1*w_, 0.1*h_, 0.8*w_, 0.8*h_)))
# grabcut (rect)
mask = np.zeros(img_rgb.shape[:2], np.uint8)
bgdModel = np.zeros((1,65), np.float64)
fgdModel = np.zeros((1,65), np.float64)
cv2.grabCut(img_bgr, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
# yellow refinement
hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
yellow = cv2.inRange(hsv, (15,60,80), (35,255,255))
mask_refined = mask.copy()
sel = yellow > 0
mask_refined[sel & (mask_refined == 0)] = cv2.GC_PR_FGD  # 3
# grabcut (mask)
cv2.grabCut(img_bgr, mask_refined, None, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_MASK)
# final mask (0/1)
final_mask = np.where((mask_refined == cv2.GC_FGD) | (mask_refined == cv2.GC_PR_FGD), 1,
    0).astype(np.uint8)
```

```
# compose fg/bg
foreground = (img_rgb * final_mask[..., None]).astype(np.uint8)
background = (img_rgb * (1 - final_mask[..., None])).astype(np.uint8)
```

(b) Produce an enhanced image with a substantially blurred background. Display the original image alongside the enhanced image.

```
def feather_mask(binary_mask, feather_px=5):
    # Feather/soften edges by Gaussian blur on a float mask
    m = binary_mask.astype(np.float32)
    m = cv2.GaussianBlur(m, (2*feather_px+1, 2*feather_px+1), feather_px)
    m = np.clip(m, 0, 1)
    return m

# Create a strongly blurred version
blur_ksize = 41  # strong blur; must be odd
blur_sigma = 0
blurred_bgr = cv2.GaussianBlur(img_bgr, (blur_ksize, blur_ksize), blur_sigma)
blurred_rgb = cv2.cvtColor(blurred_bgr, cv2.COLOR_BGR2RGB)

# Feather the mask for smoother compositing
soft_mask = feather_mask(final_mask, feather_px=5)[..., None]  # HxWx1

# Composite: keep foreground sharp, background blurred
enhanced_rgb = (img_rgb.astype(np.float32) * soft_mask + blurred_rgb.astype(np.float32) *
    (1 - soft_mask)).astype(np.uint8)
```





(c) Why is the background just beyond the edge of the flower quite dark in the enhanced image?

- Mixed pixels at boundaries: Real-world edges contain pixels that are a mixture of foreground and background due to sensor sampling and lens point spread; when multiplied by a binary/near-binary mask, these partially mixed pixels can lose brightness and appear darker.

github repo