LOW LIGHT ENHANCEMENT

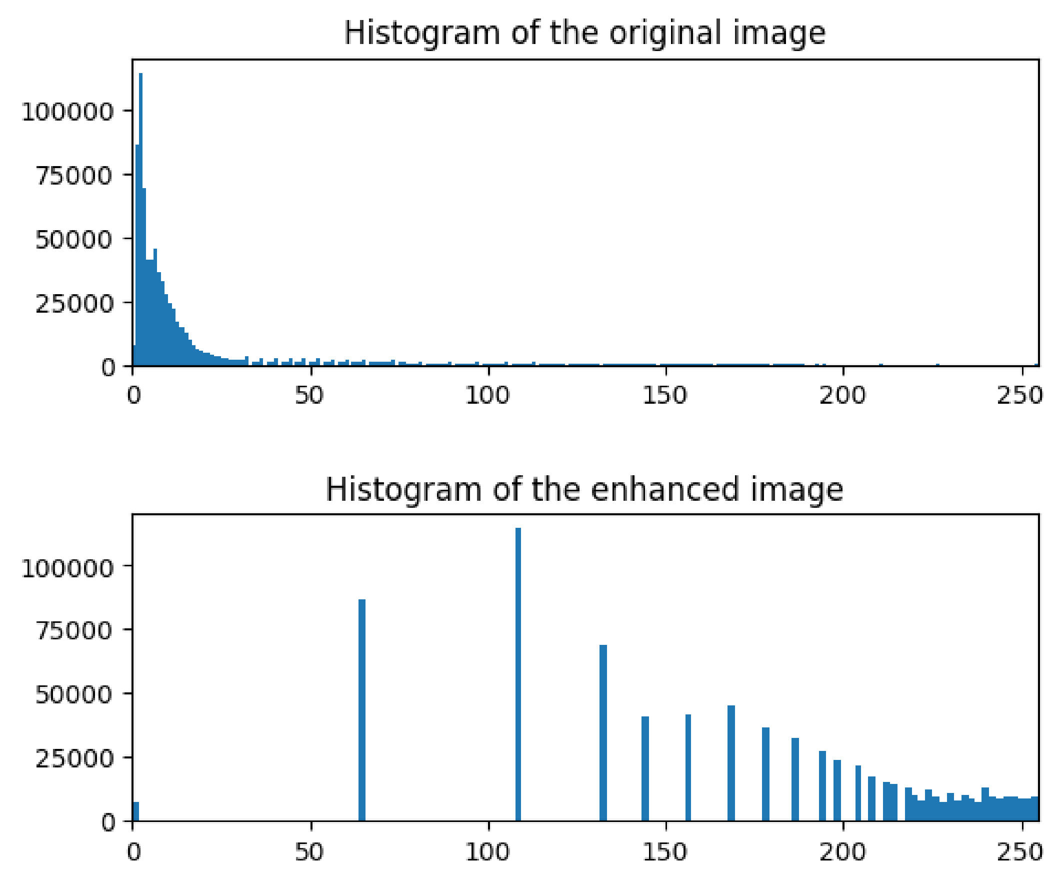
Eduardo Guiraud - s220501

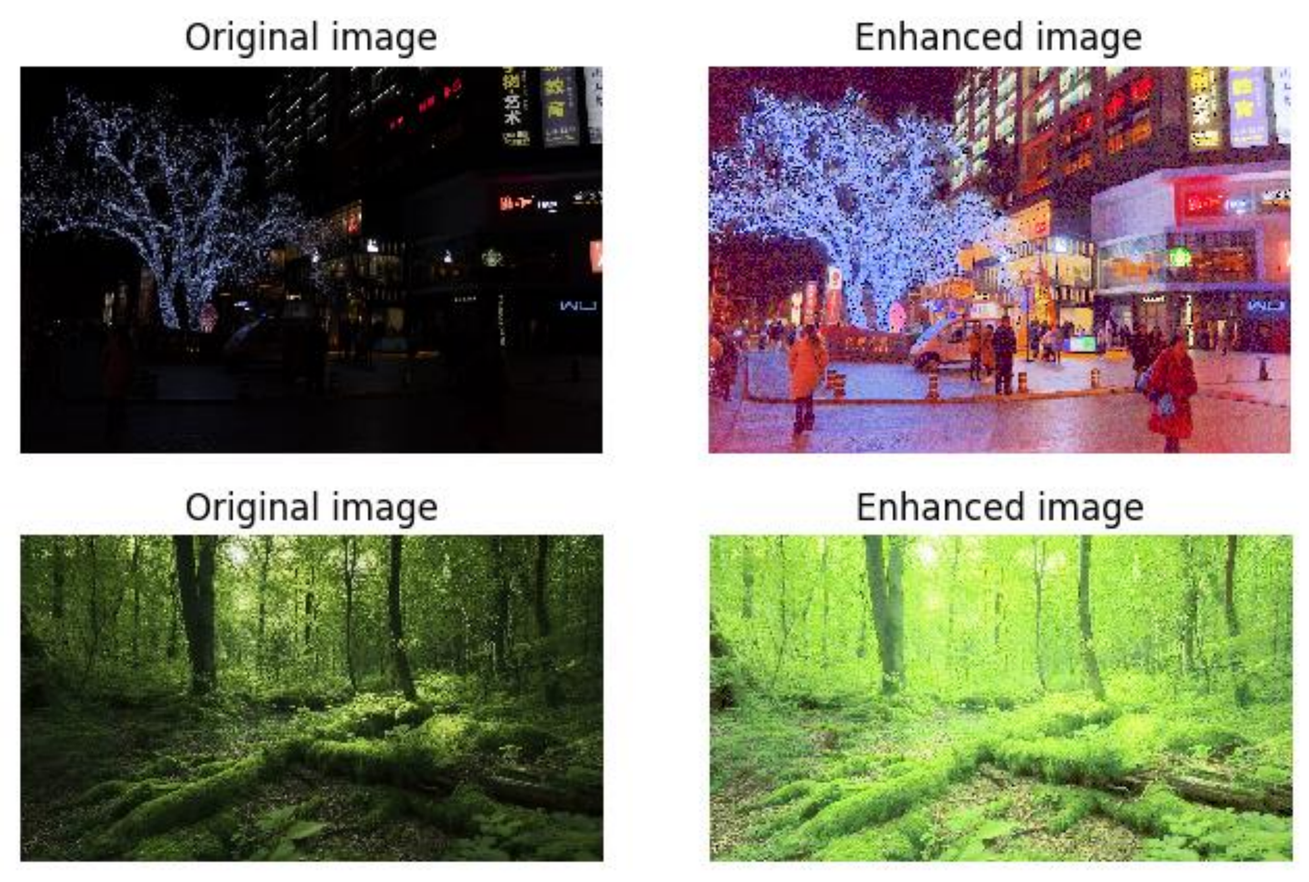
Muhammad Shaian Latif – s205080

HISTOGRAM EQUALISATION WITH ILUMINATION ADJUSTMENT

1. Transform image from RGB to HSV
2. Make a histogram equalisation of the Value channel
3. Detect the type of low light image
4. Adjust the histogram equalisation with a gamma-correction
5. Replace the Value channel with the adjusted illumination
6. Transform image back to RGB

HISTOGRAM EQUALISATION WITH ILUMINATION ADJUSTMENT

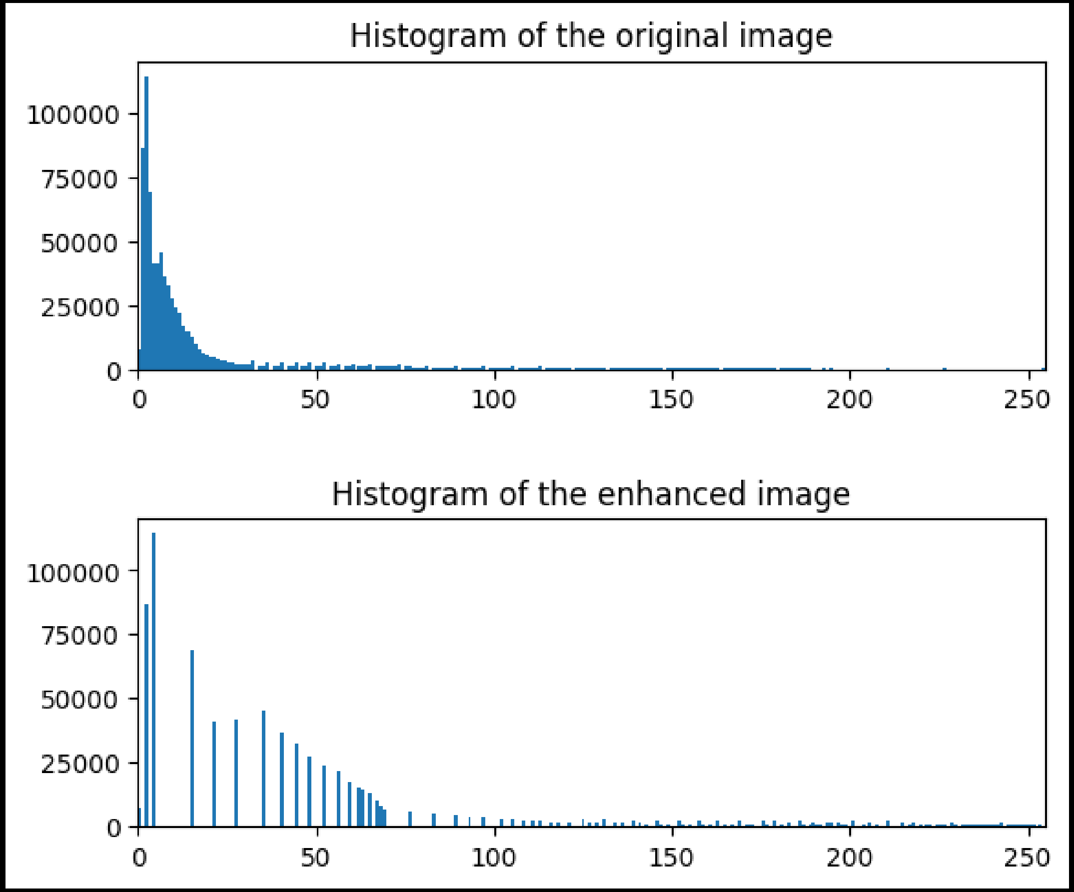


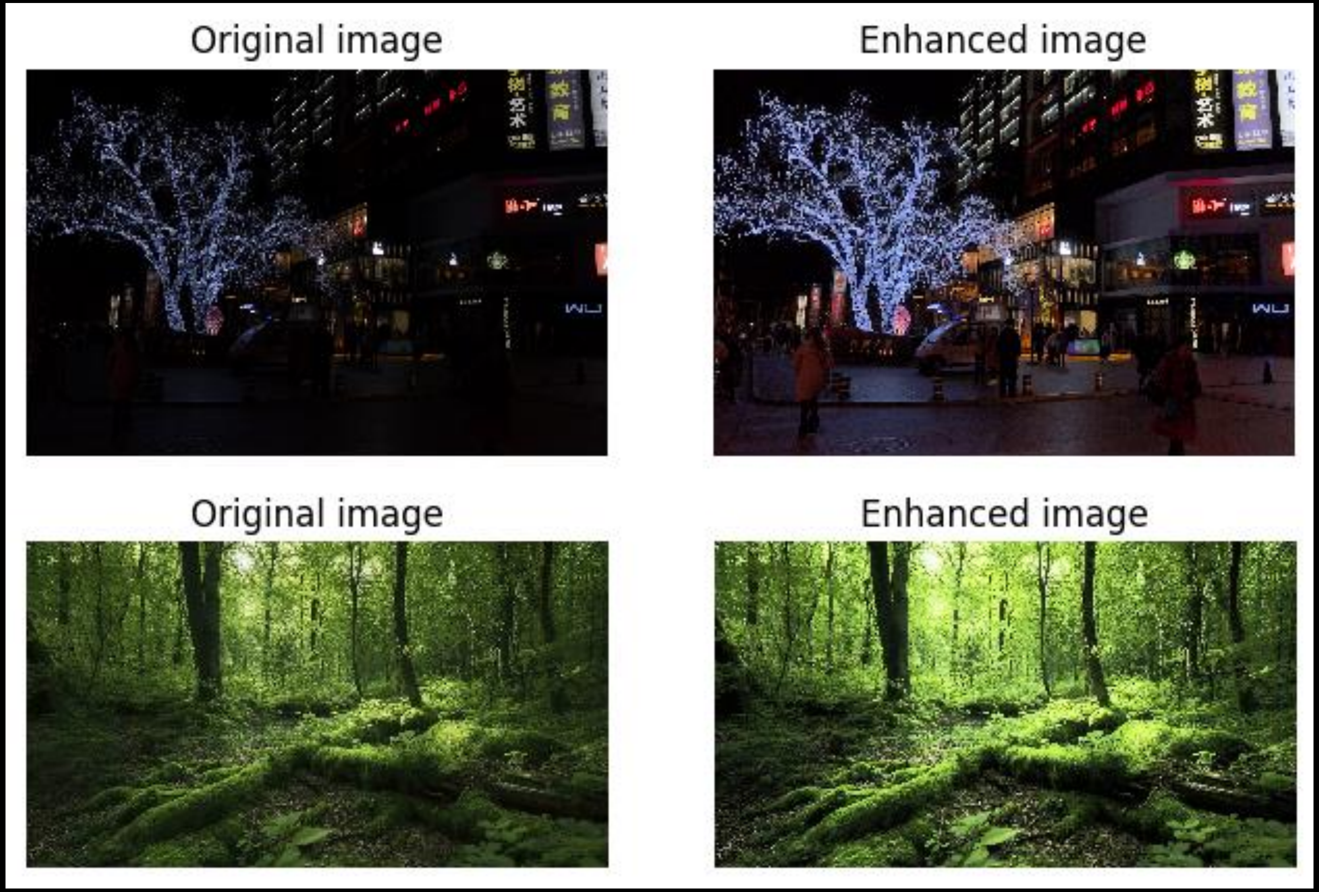


BI-HISTOGRAM EQUALISATION WITH BBHE AND BPHEME

1. Transform image from RGB to HSV
2. Find the mean value of the Value channel and divide the histogram
3. Do the histogram equalisation seperately for histogram higher and lower than the mean value (BBHE)
4. Find the peak value of the Value channel and divide the histogram
5. Do the histogram equalisation seperately for histogram higher and lower than the peak value (BPHEME)
6. Copy the image and replace the Value channel on each with the BBHE and BPHEME
7. Transform images back to RGB
8. Apply weights on the images and assemble them

BI-HISTOGRAM EQUALISATION WITH BBHE AND BPHEME

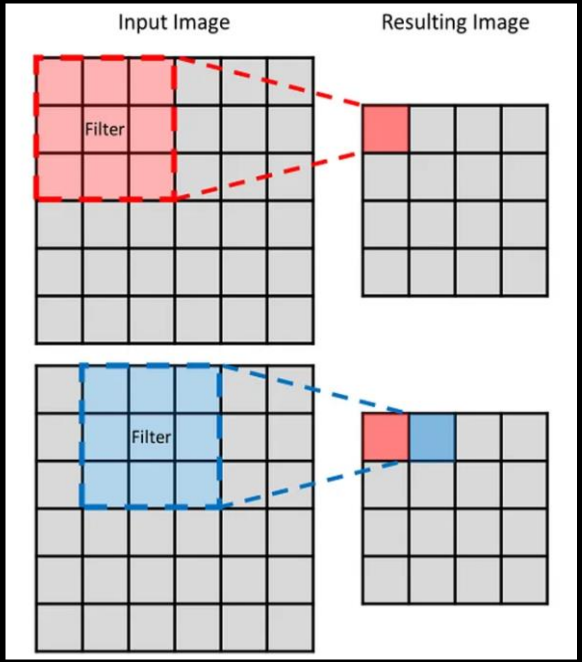




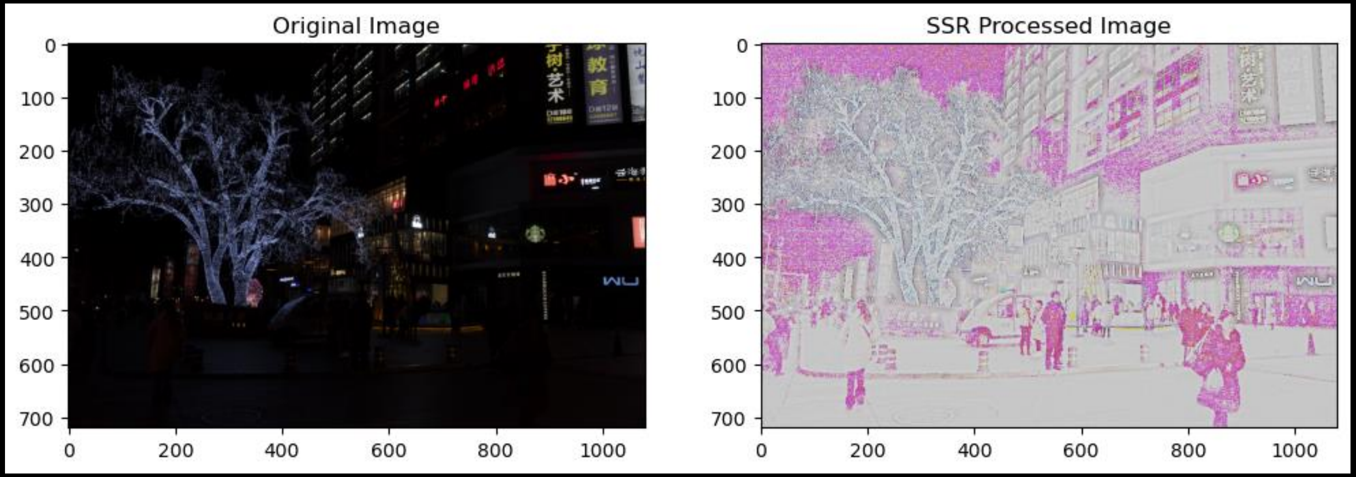
RETINEX APPROACH

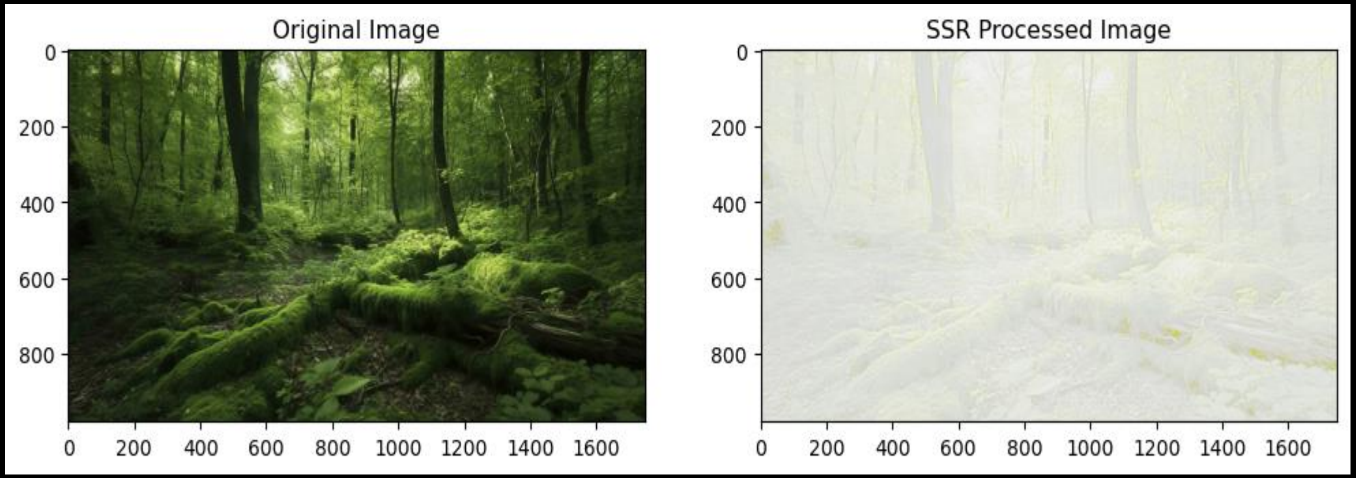
Preprocessing steps, quite simple

```python
# functions
def get_ksize(sigma):
    ksize = int(round((sigma - 0.35)/0.15))
    return ksize

def get_gaussian_blur(img, ksize=0, sigma=5):
    if ksize == 0:
        ksize = get_ksize(sigma)

    # Ensure ksize is odd to meet OpenCV's requirement
    ksize = ksize if ksize % 2 != 0 else ksize + 1

    return cv2.GaussianBlur(img, (ksize, ksize), sigma)
```



SINGLE SCALE RETINEX (SSR)



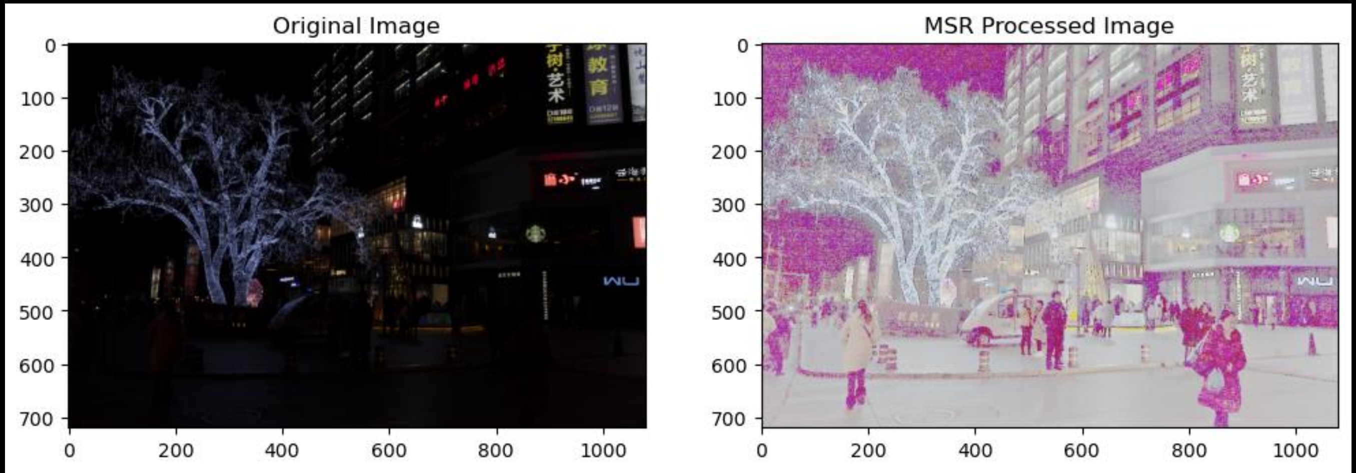


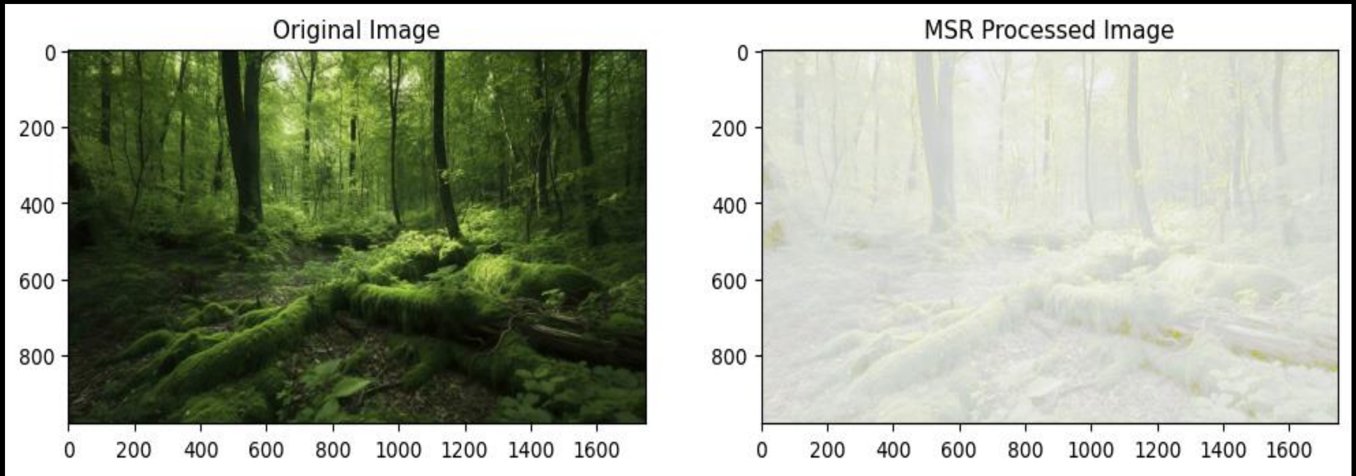
SSR - CODE

$$\mathrm{SSR_i(x,y) = \log(I_i(x,y)) - \log(G_\sigma * I_i)(x,y))}$$

```python
def ssr(img, sigma):

    # Normalize the image to the range [0, 1] and convert to float32 for precision.
    img = img.astype('float32') / 255

    # Apply Gaussian blur to the image.
    blurred = get_gaussian_blur(img, sigma=sigma)

    # Compute the SSR by taking the logarithmic difference.
    ssr_image= np.log10(img + 1e-6) - np.log10(blurred + 1e-6)


    return ssr_image
```

MULTI SCALE RETINEX (MSR)



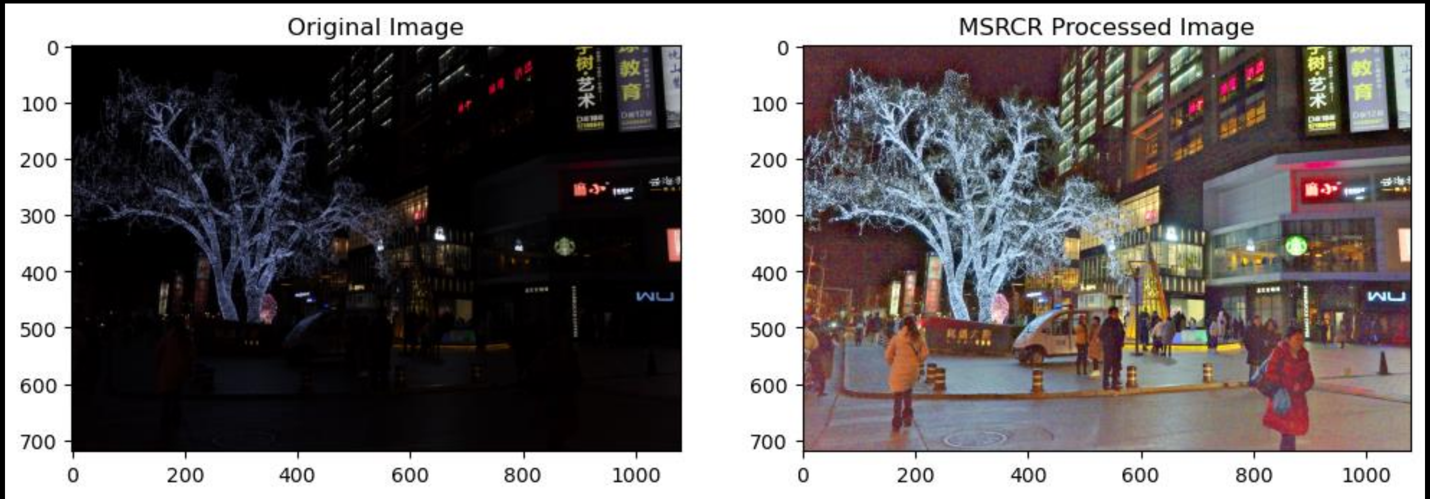


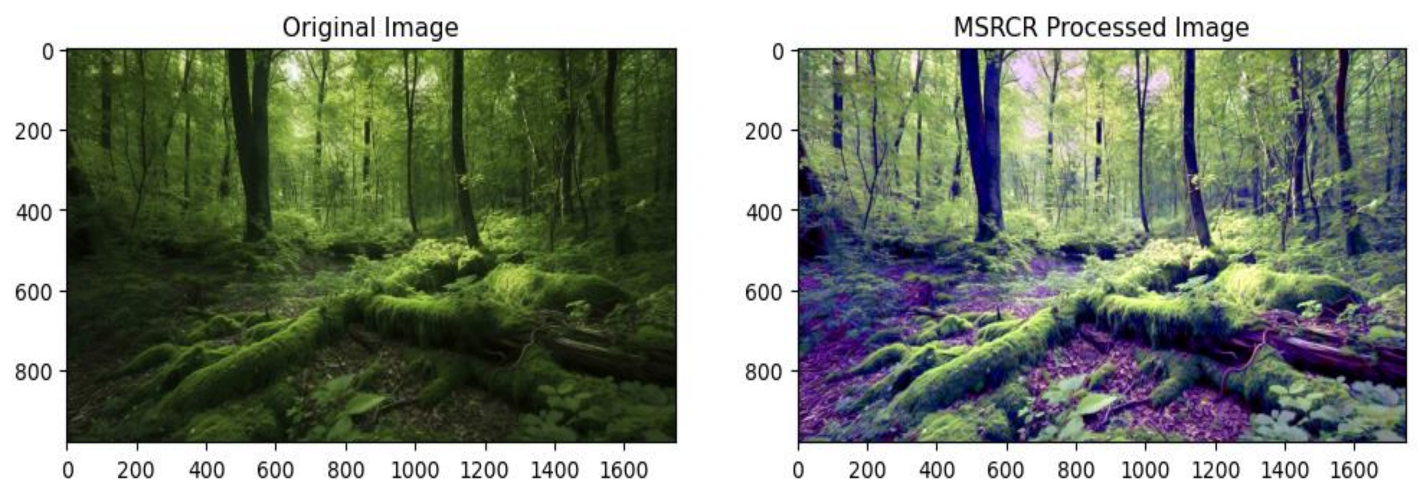
MSR - CODE

$$MSR_i(x,y) = \sum_{n=1}^{N} w_n SSR_i(x,y)$$

```python
def msr(image, sigma_scales=[15, 125, 250]):

    # Initialize an accumulator image with the same shape and type as the input image
    accumulator = np.zeros_like(image, dtype=np.float32)

    # Accumulate SSR (Single Scale Retinex) processed images for each sigma scale
    for sigma in sigma_scales:
        processed_image = ssr(image, sigma)
        accumulator += processed_image.astype(np.float32)

    # Average the accumulated images
    averaged_image = accumulator / len(sigma_scales)

    # Normalize the averaged image to the 0-255 range and convert to 8-bit unsigned integer
    msr_img = cv2.normalize(averaged_image, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8UC3)

    return msr_img
```

MULTI SCALE RETINEX WITH COLOR RESTORATION (MSRCR)





MSRCR - CODE

Color Balance function

```
def color_balance(image, low_percentile, high_percentile):
    # Calculate the number of pixels for the low and high thresholds
    total_pixels = image.shape[0] * image.shape[1]
    low_threshold_count = total_pixels * low_percentile / 100
    high_threshold_count = total_pixels * (100 - high_percentile) / 100

    # Split the image into channels or treat as a single channel for grayscale
    channels = cv2.split(image) if len(image.shape) == 3 else [image]

    adjusted_channels = []
    for channel in channels:
        # Calculate the cumulative histogram
        channel_hist = cv2.calcHist([channel], [0], None, [256], [0, 256])
        cum_hist = np.cumsum(channel_hist)

        # Determine the intensity values for the specified percentiles
        lower_bound, upper_bound = np.searchsorted(cum_hist, [low_threshold_count, high_threshold_count])

        # Create a lookup table to adjust pixel values
        lookup_table = np.interp(np.arange(256), [0, lower_bound, upper_bound, 255], [0, 0, 255, 255]).astype('uint8')

        # Apply the lookup table to adjust the channel
        adjusted_channel = cv2.LUT(channel, lookup_table)
        adjusted_channels.append(adjusted_channel)

    # Merge adjusted channels back into an image
    return cv2.merge(adjusted_channels) if len(adjusted_channels) > 1 else adjusted_channels[0]
```

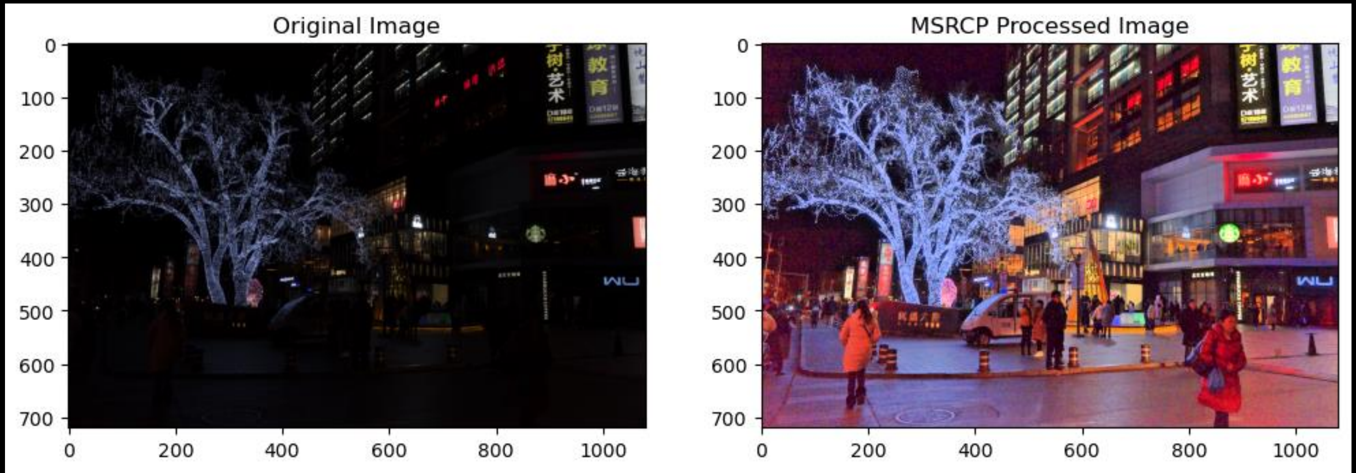
$$CRF_i(x, y) = \beta[\log(\alpha * I_i(x, y)) - \log(\sum_{c=0}^{k-1} I_c(x, y))]$$

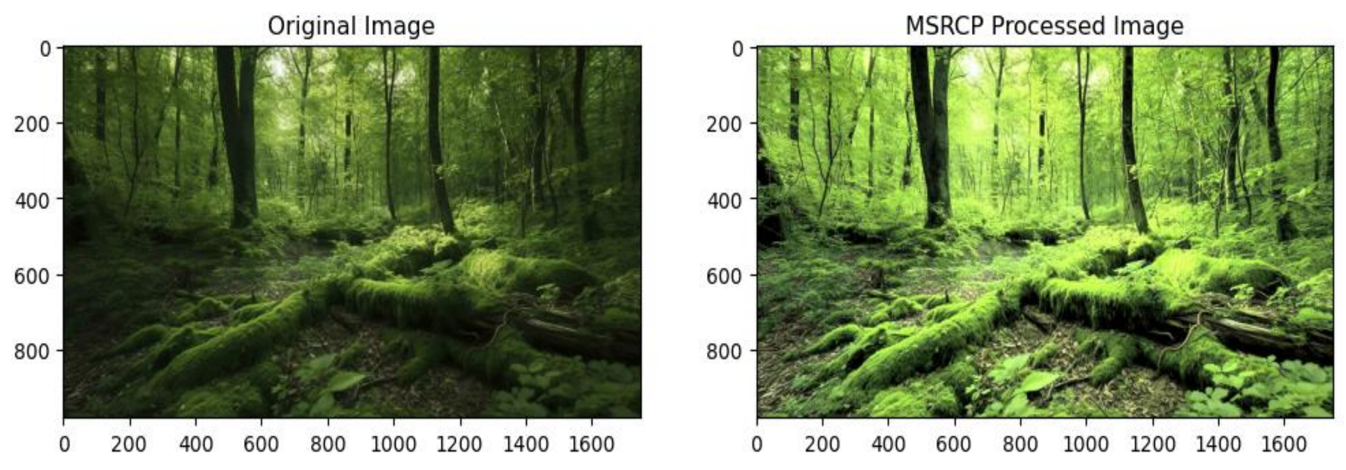
$$MSRCR_i(x, y) = G[MSR_i(x, y) * CRF_i(x, y) - b]$$

MSRCR - CODE

```python
def msrcr(img, sigma_scales=[15, 125, 250], alpha=125, beta=46, G=192, b=-30, low_percentile=2, high_percentile=1):
    """
    Apply Multi-Scale Retinex with Color Restoration (MSRCR) to an image.

    Parameters:
    - img: Input image as a NumPy array.
    - sigma_scales: List of standard deviations for Gaussian blur in MSR.
    - alpha: Gain control for logarithmic nonlinearity in color restoration. (contrast)
    - beta: Offset control for logarithmic nonlinearity in color restoration.(brightness and contrast)
    - G: Gain factor for the final image. (brightness and contrast)
    - b: Offset for the final image. (brightness)
    - low_percentile: Lower percentile for color balance. (reduce shadow noise)
    - high_percentile: Higher percentile for color balance. (prevent clipping)

    Returns:
    - msrcr_img: Image after applying MSRCR.
    """
    # Convert image to float64 for precision and add 1 to avoid log(0).
    img = img.astype(np.float64) + 1.0

    # Apply Multi-Scale Retinex (MSR).
    msr_img = msr(img, sigma_scales)

    # Compute Color Restoration Function (CRF).
    crf = beta * (np.log10(alpha * img) - np.log10(np.sum(img, axis=2, keepdims=True)))

    # Apply gain factor and offset, then normalize the MSRCR image.
    msrcr_img = G * (msr_img * crf - b)
    msrcr_img = cv2.normalize(msrcr_img, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8UC3)

    # Apply color balance to the MSRCR image.
    msrcr_img = color_balance(msrcr_img, low_percentile, high_percentile)

    return msrcr_img
```

MULTI SCALE RETINEX WITH CHROMACITY PRESERVATION (MSRCP)





MSRCP - CODE

```python
def msrcp(img, sigma_scales=[15, 125, 250], low_percentile=1, high_percentile=1):

    # Calculate the intensity image by averaging the color channels.
    intensity_image = np.mean(img, axis=2) + 1.0  # Simplified averaging

    # Enhance contrast using Multi-Scale Retinex on the intensity image.
    msr_intensity = msr(intensity_image, sigma_scales)

    # Adjust color balance based on specified percentiles.
    color_balanced_intensity = color_balance(msr_intensity, low_percentile, high_percentile)

    # Compute scaling factor to avoid overflow, adjusted for each pixel.
    scaling_factor = 256.0 / (np.max(img, axis=2) + 1.0)

    # Calculate the adjustment ratio for color balancing.
    adjustment_ratio = color_balanced_intensity / intensity_image

    # Determine the minimum scaling factor to maintain value range.
    min_scaling_factor = np.minimum(scaling_factor, adjustment_ratio)

    # Apply the minimum scaling factor, ensuring pixel values are within [0, 255].
    adjusted_img = np.clip(img * min_scaling_factor[:, :, np.newaxis], 0, 255)

    return adjusted_img.astype(np.uint8)
```

Algorithm 2: MSRCP algorithm

Data: I input color image; $\sigma_1, \sigma_2, \sigma_3$ the sca
side

Result: MSRCP output color image

begin

$\quad \mathrm{Int} = (I_R + I_G + I_B)/3$

$\quad$ **foreach** σ_i **do**

$\qquad \mathrm{Diff}_i = \log(\mathrm{Int}) - \log(\mathrm{Int} * G_{\sigma_i})$

$\quad$ **end**

$\quad \mathrm{MSR} = \sum_i \frac{1}{3} \mathrm{Diff}_i$

$\quad \mathrm{Int}_1 = \mathrm{SimplestColorBalance}(\mathrm{MSR}, s_1, s_2)$

$\quad$ **foreach** *pixel* i **do**

$\qquad B = \max(I_R[i], I_G[i], I_B[i])$

$\qquad A = \min\left(\frac{255}{B}, \frac{\mathrm{Int}_1[i]}{\mathrm{Int}[i]}\right)$

$\qquad \mathrm{MSRCP}_R[i] = A \cdot I_R[i]$

$\qquad \mathrm{MSRCP}_G[i] = A \cdot I_G[i]$

$\qquad \mathrm{MSRCP}_B[i] = A \cdot I_B[i]$

$\quad$ **end**

end

RESULTS

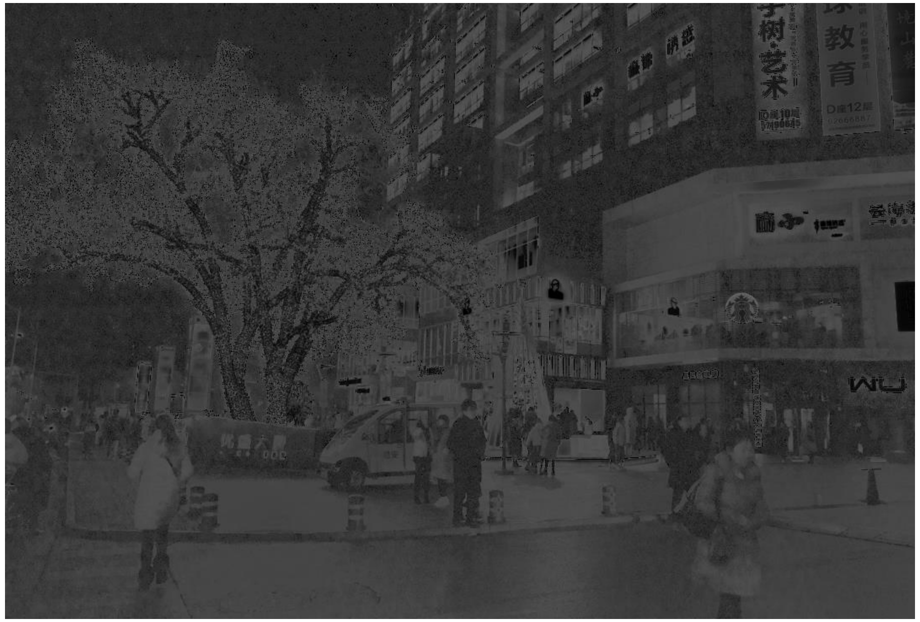
Technique	SSIM	PSNR	ΔE_{94} Average
Illumination Adjustment	0.068899	6.460112	55.907277
BBHE and BPHEME	0.443019	16.391452	12.864055
SSR	0.079659	3.837401	70.582823
MSR	0.079664	3.840461	70.569132
MSRCR	0.116994	10.01133	33.792266
MSRCP	0.136487	10.70646	29.153352

Table 2: Performance Metrics for Image Enhancement Techniques DARK

Technique	SSIM	PSNR	ΔE_{94} Average
Illumination Adjustment	0.385799	8.557906	48.060084
BBHE and BPHEME	0.863048	18.398787	12.025829
SSR	0.219019	3.341695	63.811581
MSR	0.221760	3.533424	62.548926
MSRCR	0.343755	11.46541	32.498825
MSRCP	0.541078	11.90350	29.012738

Table 3: Performance Metrics for Image Enhancement Techniques LIGHT

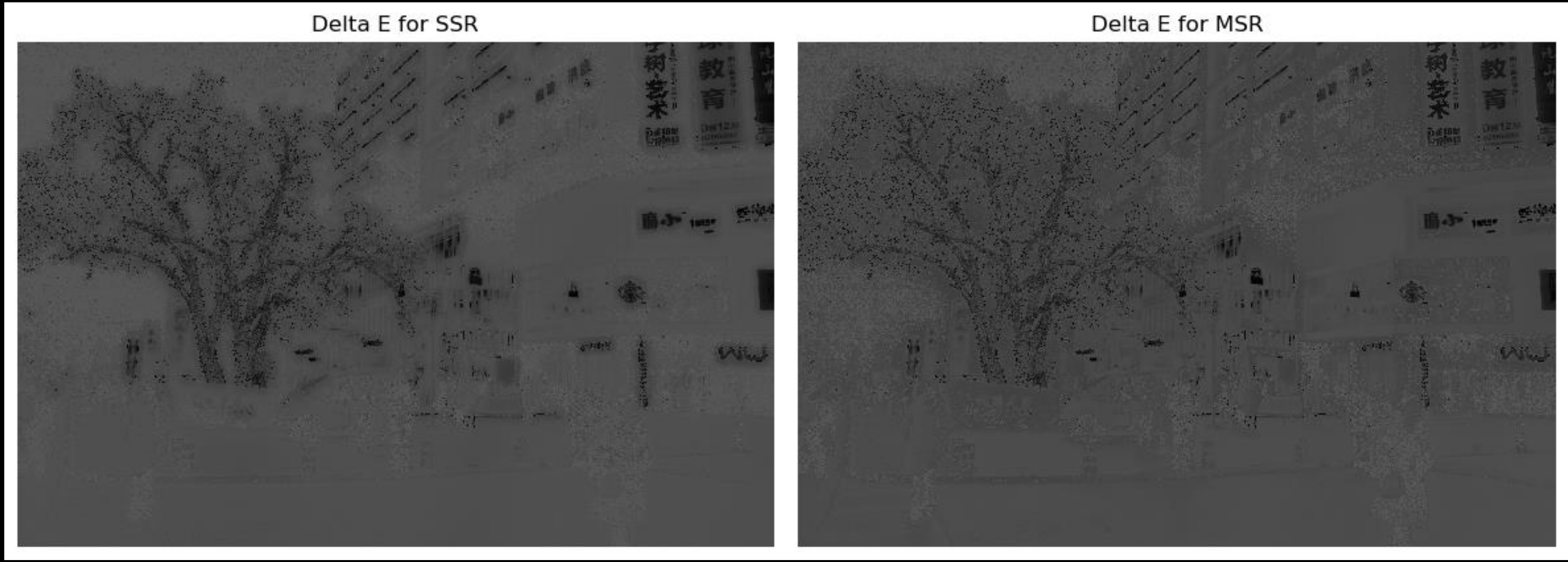
COLOR DIFFERENCE RESULTS



Illumination adjustment

BBHE and BPHEME

COLOR DIFFERENCE RESULTS



COLOR DIFFERENCE RESULTS