Enhancement of images with uneven illumination

Enhancing images with uneven illumination using ensemble learning

Group ID: #1

Til Mohr
tv.mohr@stud.uis.no

Alexander Mühleisen
???

ABSTRACT

This paper introduces a method for enhancing images with uneven illumination by leveraging the strengths of ensemble learning. Uneven illumination in images can severely affect the performance of computer vision algorithms and the visual quality for human observers. Our approach combines three classical image enhancement techniques: Unsharp Masking, Retinex, and Homomorphic Filtering, to address different aspects of the problem. These techniques are integrated through an ensemble learning framework that employs a fusion network of perceptrons for each color channel. The proposed method is unique in its application of ensemble learning to this specific problem of image enhancement. The results demonstrate that the method can effectively improve the visibility of details in images and can serve as a robust preprocessing step for further image analysis tasks.

KEYWORDS

image processing, image enhancement, uneven illumination, ensemble learning

1 INTRODUCTION

Image enhancement is a critical preprocessing step in computer vision that improves the visual quality of images, particularly when they suffer from uneven illumination. Uneven illumination can result from various factors such as lighting variances and camera limitations, leading to shadows, glares, and inconsistent brightness levels. Such issues pose significant challenges in downstream tasks like object recognition, segmentation, and tracking, as these algorithms rely heavily on uniform illumination to extract features accurately.

The goal of image enhancement in this context is to compensate for these illumination variances without introducing artifacts or losing important details. Traditional techniques like Unsharp Masking, Retinex, and Homomorphic Filtering address this issue from different angles. However, they can fall short when faced with complex illumination patterns or when one technique's strengths could complement another's weaknesses.

To bridge this gap, we propose a simple ensemble learning-based image enhancement framework that combines the strengths of individual enhancement methods. Ensemble learning, typically used in machine learning for decision-making tasks, can be effectively applied to image processing. By integrating the outputs of different enhancement techniques, we hope to produce a single, high-quality image that benefits from the cumulative strengths of each method. A main research goal of this report is to investigate to which extent the different enhancement methods contribute to the final result, and in which situations the fused image is superior to the individual images produced by the enhancement methods.

The rest of the paper is structured as follows. Section 2 reviews the current enhancement methods and their theoretical underpinnings. In Section 3 we detail our ensemble learning approach and the training of the fusion network. We then present our experimental results in Section 4, followed by a discussion of their implications in Section 5. Finally, we conclude with a summary of our findings and suggestions for future work.

2 THEORY

In this section we will dive into different methods to enhance images with uneven illumination. We will start with a brief introduction to the problem and then discuss different methods to solve it, as well as how to evaluate the results.

2.1 Problem description

Uneven illumination refers to the irregular distribution of light intensity across an image. In essence, it disrupts the uniformity of the visual output, leading to disparities in brightness and contrast, often observable as glares or shadows. These disparities can mask essential features and details, making the subject of the image less identifiable. This becomes especially problematic when images need to be processed further for various computer vision tasks. In fields like optical microscopy, for example, consistent illumination is crucial for accurately identifying and segmenting microscopic entities. Uneven lighting can obscure crucial cellular structures or make similar-looking entities appear distinct, hampering accurate analysis [3].

To counter this issue, the goal is to enhance the image in a manner that simulates its capture under uniform illumination conditions. By doing so, we aim to restore a natural appearance to the image, preserving details and minimizing artifacts introduced by uneven lighting. This correction enables better analysis, ensuring that conclusions drawn are based on the actual subject and not on lighting imperfections [3].

2.2 Unsharp Masking

Unsharp masking is a technique to sharpen images, enhancing edges and fine details by utilizing a blurred copy of the image. The technique's somewhat paradoxical name comes from how it operates: by subtracting the blurred version from the original image, it isolates the 'unsharp' or high-frequency parts—the details and edges—then these are amplified and recombined with the original, resulting in a clearer, more defined image. The process can be expressed mathematically as [2, 5, 7]:

$$g(x, y) = f(x, y) + \lambda \cdot (f(x, y) - Blur(f)(x, y)) \tag{1}$$

Here, $f(x, y)$ represents the original image, $Blur(f)(x, y)$ is the blurred version of the original image, and λ is a positive value determining how much sharpening is applied. The blurring is often achieved with a Gaussian filter, a common choice for such image processing tasks [2, 5, 7]. An implementation of the unsharp masking algorithm is shown in Listing 2.

2.3 Retinex

Retinex theory forms the basis of a research area aimed at replicating human vision perception through models. This field has birthed various algorithms to improve the visual quality of images. Notably, the Multi-Scale Retinex with Chromacity Preservation (MSRCP) algorithm has been developed. MSRCP enhances the original Multi-Scale Retinex (MSR), which itself is an advancement of the Single Scale Retinex (SSR). The SSR is described by the following mathematical expression [1, 6]:

$$\mathrm{R}n_i(x, y) = \log(f_i(x, y)) - \log(f_i(x, y) * F_n(x, y)) \tag{2}$$

In this formula, $f_i(x, y)$ represents the pixel intensity of the input image at location (x, y) in the i-th color channel, while $F_c(x, y)$ is a Gaussian function used to analyze the surrounding pixels with a standard deviation of $\sigma = n$. Building upon SSR, MSR averages the SSR outputs over multiple scales, as indicated here [1, 6]:

$$\mathrm{R}MSR_i(x, y) = \sum_{n=1}^{N} \omega_n \cdot \mathrm{R}_{ni}(x, y) \tag{3}$$

where ω_n are the weights for each scale. Research has indicated that while MSR can enhance image details, it may also lead to desaturated colors. Hence, MSRCP was introduced. This method first creates an improved image using MSR and then adjusts this image's colors to span the entire available color range [6]. Afterward, it combines the color-stretched image with the original to adjust and intensify the original image's colors [6]. An example of how this is implemented can be found in Listing 4.

2.4 Homomorphic Filtering

The intensity of an image at pixel (x, y) can be described as the product of the illumination $i(x, y)$ and the reflectance $r(x, y)$ [4, 8]:

$$f(x, y) = i(x, y) \cdot r(x, y) \tag{4}$$

In the frequency domain, illumination changes across the image are typically manifested by low frequencies, while high frequencies are associated with reflectance changes. Therefore, by applying the logarithm to the image, one can separate the illumination and reflectance components of the image [4, 8]:

$$\log(f(x, y)) = \log(i(x, y)) + \log(r(x, y)) \tag{5}$$

Applying the Fourier transform to this log-image, a filter $H(u, v)$ can be applied to attenuate the low frequencies, that is the frequencies responsible for illumination changes, and increasing the high frequencies responsible for detail. To finish the enhancement, we revert the process by applying an inverse Fourier transform and exponentiation [4, 8]:

$$f(x, y) = \exp(\mathcal{F}^{-1}(\mathcal{F}(\log(f(x, y))) \cdot H(u, v))) \tag{6}$$

This process is illustrated in Figure 1.

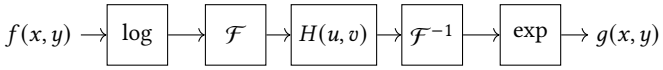

Figure 1: Homomorphic filtering pipeline.

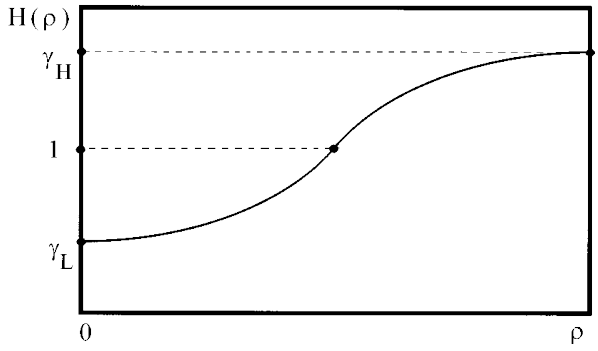

Figure 2: General form of the filter used in homomorphic filtering [8].

There are numerous types of linear filters that can be applied; Voicu et al. suggest using a Butterworth filter of the second order [8]. This particular filter focuses on reducing the impact of low frequencies while accentuating high frequencies:

$$H(u, v) = H'(\rho) = \gamma_1 - \gamma_2 \cdot \frac{1}{1 + 2.415 \cdot \left(\frac{\rho}{\rho_c}\right)^4}, \tag{7}$$

$$\text{where} \qquad \rho = \sqrt{u^2 + v^2} \tag{8}$$

Here, γ_1 and γ_2 are constants that can be adjusted for the desired outcome, with $\gamma_1 \approx \gamma_H$ and $\gamma_2 \approx \gamma_H - \gamma_L$, and ρ_c is the cutoff frequency [8]. The filter's general shape is depicted in Figure 2.

Additionally, Fan et al. recommend including a step for histogram equalization after the filtering to further improve the image's contrast [4]. For color images, the homomorphic filtering process can be applied to each individual color channel, e.g. the illumination channel of HSI images, or to every channel as in RGB images [4, 8]. An example of how to implement this approach is provided in Listing 3.

2.5 Assessment of Image Enhancement

Determining the success of image enhancement depends on the purpose of the process. For aesthetic purposes, a simple visual check may be enough to judge improvement. If the enhancement serves as preparation for a subsequent task like image segmentation, its success should be measured based on how it improves the results of that task. Nevertheless, there are specific objective criteria we can use to evaluate enhancement:

2.5.1 RMS Contrast. Contrast refers to how distinctly the dark and light areas of an image stand apart, essentially how easy it is to distinguish different objects in the picture. When correcting an image with inconsistent lighting, our goal is to better the contrast in regions that were initially similarly lit. Thus, the enhancement

may not always boost the overall contrast but could lead to local improvements. RMS contrast is calculated as the standard deviation of pixel intensities across the entire image [3]:

$$\text{RMS Contrast} = \sqrt{\frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} (I(i,j) - \bar{I})^2} \tag{9}$$

2.5.2 Discrete Entropy. Entropy describes the amount of information in an image, where a high entropy means that the image contains a lot of information, and a low entropy means that the image contains little information, i.e. a flat image has zero entropy. Enhancing an image with uneven illumination should increase the amount of information in the image, and therefore increase the entropy. The discrete entropy is defined as [3, 10]:

$$\text{Discrete Entropy} = -\sum_{i} P_i \cdot \log_2(P_i) \tag{10}$$

where P_i is the probability that the difference between two adjacent pixels is i.

3 METHODOLOGY

This paper aims to investigate the efficacy of combining Unsharp Masking (UM), Retinex (RTX), and Homomorphic Filtering (HF) enhancement techniques into a coherent ensemble framework. We propose a approach by integrating these methods through a fusion network, comprising three distinct perceptron networks corresponding to each color channel. This section delineates the methodology in detail, including the structure of the fusion network and the training of its parameters.

3.1 Fusion Network

Our exploration centers on a fusion network that amalgamates the strengths of UM, RTX, and HF, detailed in Sections 2.2, 2.3, and 2.4, respectively. The operational sequence of our model begins with generating intermediate enhanced images through each method, denoted as g_{UM}, g_{RTX}, and g_{HF}. Subsequently, these images are inputted into the fusion network to yield the final enhanced image g_F.

Algorithm 1 Fusion Network

Require: g_{UM}, g_{RTX}, g_{HF} in HSI color space
Require: $w_c \in \mathbb{R}^3$ for $c \in \{hsi\}$
Ensure: $N \coloneqq \text{width}(g_{UM}) = \text{width}(g_{RTX}) = \text{width}(g_{HF})$
Ensure: $M \coloneqq \text{height}(g_{UM}) = \text{height}(g_{RTX}) = \text{height}(g_{HF})$
  $g_F \leftarrow 0^{N \times M \times 3}$
  for $c \in \{hsi\}$ **do**
    for $(x, y) \in \{1, \ldots, N\} \times \{1, \ldots, M\}$ **do**
      $g_F(x, y, c) \leftarrow w_{c_1} \cdot g_{UM}(x, y, c)$
        $+ w_{c_2} \cdot g_{RTX}(x, y, c)$
        $+ w_{c_3} \cdot g_{HF}(x, y, c)$
    end for
  end for
  return g_F

The fusion network is constructed from three individual perceptron networks, each tailored for a specific channel of the HSI color space. For any given pixel at coordinates (x, y), each perceptron receives three inputs: the values of channel $c \in hsi$ from g_{UM}, g_{RTX}, and g_{HF}. The output is the channel c of the pixel (x, y) in g_F. The fusion process is algorithmically represented in Algorithm 1. The weights $w_c \in \mathbb{R}^3$, for each channel $c \in hsi$, are parameters that determine the influence of each intermediate image and are optimized during the training process, as discussed in Section 3.3. In our experiments, the use of bias was omitted. The fusion process is visually summarized in Figure 3.

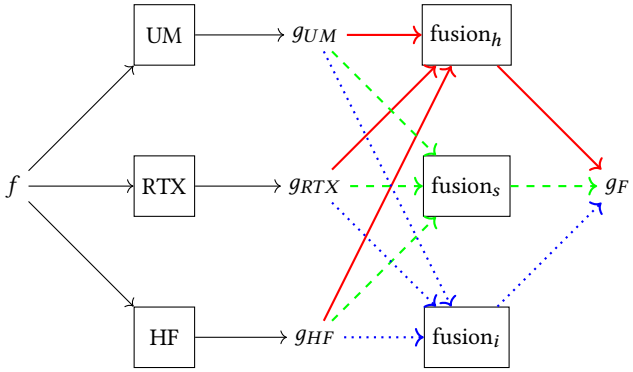

Figure 3: Pipeline of the fusion network approach. Red solid arrows symbolize the fusion of the hue channel, green dashed arrows symbolize the fusion of the saturation channel, and blue dotted arrows symbolize the fusion of the intensity channel.

3.2 Implementation

For the implementation of our fusion network, we employed the `PyTorch` machine learning framework. It involves three distinct linear fully connected layers, one for each color channel. An abridged version of the code for the fusion network is presented in Listing 1. Comprehensive explanations of the theoretical underpinnings and implementations for UM, RTX, and HF are provided in Sections 2.2, 2.3, and 2.4, with corresponding code snippets in Listings 2, 4, and 3. The entire codebase is made accessible on GitHub[1].

```
import torch
import numpy as np
from util import BGR2HSI, HSI2BGR


class ChannelNet(torch.nn.Module):
    def __init__(self):
        super(ChannelNet, self).__init__()
        self.fc = torch.nn.Linear(3, 1, bias=False)

    def forward(self, x):
        return self.fc(x)


class FusionNet(torch.nn.Module):
    def __init__(self):
        super(FusionNet, self).__init__()
        self.h_net = ChannelNet()
```

[1]URL: https://github.com/CodingTil/eiuie

```
        self.s_net = ChannelNet()
        self.i_net = ChannelNet()

    def forward(self, x):
        # Flatten the middle dimensions
        x = x.view(-1, 12)
        # Splitting the input for the three channels
        h_channel = x[:, 0::3]
        s_channel = x[:, 1::3]
        i_channel = x[:, 2::3]
        # Getting the outputs
        h_out = self.h_net(h_channel)
        s_out = self.s_net(s_channel)
        i_out = self.i_net(i_channel)
        # Concatenate the outputs to get the final output
        return torch.cat((h_out, s_out, i_out), dim=1)


def process_image(image: np.ndarray, net: FusionNet) ->
    np.ndarray:
    dimensions = image.shape
    um_imagge = unsharp_masking(image)
    um_image = BGR2HSI(um_imagge)
    hf_image = homomorphic_filtering(image)
    hf_image = BGR2HSI(hf_image)
    rtx_image = retinex(image)
    rtx_image = BGR2HSI(rtx_image)

    # Use numpy functions for efficient concatenation
    um_image = um_image.reshape(-1, 3)
    hf_image = hf_image.reshape(-1, 3)
    rtx_image = rtx_image.reshape(-1, 3)
    all_inputs = np.hstack([um_image, hf_image, rtx_image
    ])
    all_inputs = torch.tensor(all_inputs, dtype=torch.
    float32)

    # Model inference
    outputs = net(all_inputs).numpy()
    outputs = np.clip(outputs, 0, 1)
    fused_image = outputs.reshape(dimensions[0],
    dimensions[1], 3)
    fused_image = HSI2BGR(fused_image)
    return fused_image
```

Listing 1: Fusion Model

3.3 Training

The training of the weight parameters $w_c \in \mathbb{R}^3$ for each channel $c \in hsi$ is imperative for effective image enhancement. Leveraging the perceptron networks, the weights are refined through supervised learning. In an optimal setting, our training dataset comprises images with uneven illumination and their evenly illuminated counterparts. However, no such dataset is known to us. Due to the absence of a directly relevant dataset, we adapted the LOL-dataset, which includes image pairs with low and normal exposure [9].

The training process involved using both the low-light and normal-light images from each pair as input, with the normal-light images always serving as the target output. This dual-input strategy was intended to calibrate the model to enhance underexposed areas without exaggerating well-lit sections.

With a dataset size of approximately 2.4 billion samples (pixels), we were limited by memory constraints to utilize only half of this dataset. We adopted the Adam optimizer with a learning rate of 0.001 and a batch size of 2^{15}. The training was set to run for 1000 epochs, but early stopping criteria were met shortly after the 100th epoch. Remarkably, the entire training process was completed in about 10 minutes on a single NVIDIA L4 Tensor Core GPU.

4 RESULTS

5 DISCUSSION AND CONCLUSIONS

REFERENCES

[1] Kobus Barnard and Brian Funt. 1998. Investigations into multi-scale retinex. *Proc. Colour Imaging in Multimedia'98* (1998), 9–17.

[2] Guang Deng. 2010. A generalized unsharp masking algorithm. *IEEE transactions on Image Processing* 20, 5 (2010), 1249–1261.

[3] Nilanjan Dey. 2019. Uneven illumination correction of digital images: A survey of the state-of-the-art. *Optik* 183 (2019), 483–495.

[4] Chun-Nian Fan and Fu-Yan Zhang. 2011. Homomorphic filtering based illumination normalization method for face recognition. *Pattern Recognition Letters* 32, 10 (2011), 1468–1479.

[5] Koichi Morishita, Shimbu Yamagata, Tetsuo Okabe, Tetsuo Yokoyama, and Kazuhiko Hamatani. 1988. Unsharp masking for image enhancement. US Patent 4,794,531.

[6] Ana Belén Petro, Catalina Sbert, and Jean-Michel Morel. 2014. Multiscale retinex. *Image Processing On Line* (2014), 71–88.

[7] Zenglin Shi, Yunlu Chen, Efstratios Gavves, Pascal Mettes, and Cees GM Snoek. 2021. Unsharp mask guided filtering. *IEEE Transactions on Image Processing* 30 (2021), 7472–7485.

[8] Liviu I Voicu, Harley R Myler, and Arthur Robert Weeks. 1997. Practical considerations on color image enhancement using homomorphic filtering. *Journal of Electronic Imaging* 6, 1 (1997), 108–113.

[9] Chen Wei, Wenjing Wang, Wenhan Yang, and Jiaying Liu. 2018. Deep retinex decomposition for low-light enhancement. *arXiv preprint arXiv:1808.04560* (2018).

[10] Zhengmao Ye, Habib Mohamadian, and Yongmao Ye. 2007. Discrete entropy and relative entropy study on nonlinear clustering of underwater and arial images. In *2007 IEEE International Conference on Control Applications*. IEEE, 313–318.

A LISTINGS

A.1 Unsharp Masking

```python
import numpy as np
import cv2


def process_image(self, image: np.ndarray) -> np.ndarray:
    blurred = cv2.GaussianBlur(image, (self.ksize, self.ksize), 0)
    sharpened = cv2.addWeighted(image, 1 + self.alpha, blurred, -self.alpha, 0)
    return sharpened
```

Listing 2: Unsharp masking

A.2 Homomorphic Filtering

```python
import numpy as np
import cv2
from util import BGR2HSI, HSI2BGR


def filter(value, gamma_1: float = 1.0, gamma_2: float = 0.6, rho: float = 2.0):
    return gamma_1 - gamma_2 * (1 / (1 + 2.415 * np.power(value / rho, 4)))


def process_image(image: np.ndarray) -> np.ndarray:
    # Convert image to HSI space
    image = image.astype(np.float32)
    hsi = BGR2HSI(image)

    # Extract intensity channel and apply homomorphic filtering
    i = hsi[:, :, 2]
    i_log = np.log2(i + 1.0)
    i_log_fft_shifted = np.fft.fftshift(np.fft.fft2(i_log))
    i_log_fft_shifted_filtered = np.zeros_like(i_log_fft_shifted)
    for i in range(i_log_fft_shifted.shape[0]):
        for j in range(i_log_fft_shifted.shape[1]):
            i_log_fft_shifted_filtered[i, j] = i_log_fft_shifted[i, j] * filter(
                np.sqrt(
                    (i - i_log_fft_shifted.shape[0] / 2) ** 2
                    + (j - i_log_fft_shifted.shape[1] / 2) ** 2
                )
            )
    i_log_filtered = np.real(np.fft.ifft2(np.fft.ifftshift(i_log_fft_shifted_filtered)))
    i_filtered = np.exp2(i_log_filtered) - 1.0
    # Replace intensity channel with filtered one
    hsi_filtered = hsi.copy()
    hsi_filtered[:, :, 2] = i_filtered

    # Convert image back to BGR space
    image = HSI2BGR(hsi)
    image = np.clip(image, 0, 255)
    image = image.astype(np.uint8)

    # Equalize histogram of value channel
    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    image[:, :, 2] = cv2.equalizeHist(image[:, :, 2])

    # Convert image back to BGR space
    image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
    return image
```

Listing 3: Homomorphic filtering

A.3 Retinex

```python
from typing import List, Optional

import numpy as np
import cv2


def get_ksize(sigma: float) -> int:
    # opencv calculates ksize from sigma as
    # sigma = 0.3*((ksize-1)*0.5 - 1) + 0.8
    # then ksize from sigma is
    # ksize = ((sigma - 0.8)/0.15) + 2.0
    return int(((sigma - 0.8) / 0.15) + 2.0)


def get_gaussian_blur(
    img: np.ndarray, ksize: Optional[int] = None, sigma: float = 5.0
) -> np.ndarray:
    if ksize is None:
        ksize = get_ksize(sigma)
    # Gaussian 2D-kernel can be seperable into 2-orthogonal vectors
    # then compute full kernel by taking outer product or simply mul(V, V.T)
    sep_k = cv2.getGaussianKernel(ksize, sigma)
    # if ksize >= 11, then convolution is computed by applying fourier transform
    return cv2.filter2D(img, -1, np.outer(sep_k, sep_k))


def ssr(img: np.ndarray, sigma: float) -> np.ndarray:
    return np.log10(img) - np.log10(get_gaussian_blur(img, ksize=0, sigma=sigma) + 1.0)


def msr(img: np.ndarray, sigma_scales: List[float] = [15, 80, 250]) -> np.ndarray:
    msr = np.zeros(img.shape)
    for sigma in sigma_scales:
        msr += ssr(img, sigma)
    msr = msr / len(sigma_scales)
    # computed MSR could be in range [-k, +l], k and l could be any real value
    # so normalize the MSR image values in range [0, 255]
    msr = cv2.normalize(msr, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8UC3)
    return msr


def color_balance(img: np.ndarray, low_per: float, high_per: float) -> np.ndarray:
    tot_pix = img.shape[1] * img.shape[0]
    # no.of pixels to black-out and white-out
    low_count = tot_pix * low_per / 100
    high_count = tot_pix * (100 - high_per) / 100
    # channels of image
    ch_list = []
    if len(img.shape) == 2:
```

```
        ch_list = [img]
    else:
        ch_list = cv2.split(img)
    cs_img = []
    # for each channel, apply contrast-stretch
    for i in range(len(ch_list)):
        ch = ch_list[i]
        # cummulative histogram sum of channel
        cum_hist_sum = np.cumsum(cv2.calcHist([ch], [0], None, [256], (0, 256)))
        # find indices for blacking and whiting out pixels
        li, hi = np.searchsorted(cum_hist_sum, (low_count, high_count))
        if li == hi:
            cs_img.append(ch)
            continue
        # lut with min-max normalization for [0-255] bins
        lut = np.array(
            [
                0 if i < li else (255 if i > hi else round((i - li) / (hi - li) * 255))
                for i in np.arange(0, 256)
            ],
            dtype="uint8",
        )
        # constrast-stretch channel
        cs_ch = cv2.LUT(ch, lut)
        cs_img.append(cs_ch)
    if len(cs_img) == 1:
        return np.squeeze(cs_img)
    elif len(cs_img) > 1:
        return cv2.merge(cs_img)
    raise Exception("Color balance failed")


def msrcp(
    img: np.ndarray,
    sigma_scales: List[float] = [15, 80, 250],
    low_per: float = 1,
    high_per: float = 1,
) -> np.ndarray:
    # Intensity image (Int)
    int_img = (np.sum(img, axis=2) / img.shape[2]) + 1.0
    # Multi-scale retinex of intensity image (MSR)
    msr_int = msr(int_img, sigma_scales)
    # color balance of MSR
    msr_cb = color_balance(msr_int, low_per, high_per)
    # B = MAX/max(Ic)
    B = 256.0 / (np.max(img, axis=2) + 1.0)
    # BB = stack(B, MSR/Int)
    BB = np.array([B, msr_cb / int_img])
    # A = min(BB)
    A = np.min(BB, axis=0)
    # MSRCP = A*I
    msrcp = np.clip(np.expand_dims(A, 2) * img, 0.0, 255.0)
    return msrcp.astype(np.uint8)
```

Listing 4: Retinex