Contents

1 Theory

Colorization using optimization is a technique that colorizes a black and white image using a batch of colored scribbles. This technique operates on YUV channel, where Y stands for intensity (the monochromatic luminance channel) and U and V are the chrominance channels. [1] The information of a black-and-white (bw) image is only recorded in Y. The colorization follows the premise that if two pixels $\mathbf{r}$ and $\mathbf{s}$ have similar intensities (Y), then their colors (U and V) are similar. In terms of channel U, this premise is described by the following loss function:

$$J(U) = \sum_{\mathbf{r}} \left(U(\mathbf{r}) - \sum_{\mathbf{s} \in N(\mathbf{r})} w_{\mathbf{rs}} U(\mathbf{s}) \right)^2 \tag{1}$$

where $N(\mathbf{r})$ is a window (3×3) around $\mathbf{r}$ excluding $\mathbf{r}$ itself, and $w_{\mathbf{rs}}$ is a weighting function given as:

$$w_{\mathbf{rs}} \propto e^{-\frac{(Y(\mathbf{r}) - Y(\mathbf{s}))^2}{2\sigma_{\mathbf{r}}^2}} \tag{2}$$

where $\sigma_{\mathbf{r}}^2$ is the variance of the intensities of pixels in $N(\mathbf{r})$. Channel V follows the same loss function. After determining U and V channels, transforming YUV channel back into RGB channel would output the colorized image.

Theoretically, determining the global minimum of $J(U)$ (it is convex) would output the best result. However, the implementer encountered memory shortage when using scipy.optimize to follow this way (will be discussed later). Having referenced other implementations on the internet [2], the implementer decided to modify the optimizing goal. Instead of globally miminizing $J(U)$, the goal now becomes

$$U(\mathbf{r}) = \sum_{\mathbf{s} \in N(\mathbf{r})} w_{\mathbf{rs}} U(\mathbf{s}), \forall\ \mathbf{r}. \tag{3}$$

which is intuitively modified from equation (1). Though solving equation (3) will not give the global minimum of $J(U)$, a satisfying result is still provided.

2 Implementation

The first step is to read the scribbled image (m1) and the bw image (m2) in RGB channel and transform them into YUV channel. The following are some specification about the implementing details:

1. The implementer imported OpenCV (cv2) for reading images and colorsys for color transforming. Though cv2 and colorsys both support channel transforming, cv2's might cause bugs in this implementation.

2. Channel Y was obtained from m2 and channel U and V were from m1. This is because that scribbles overlay the Y information of the original bw pixels.

3. The RGB values have been normalized to the range of $[0, 1]$, resulting in float values before transforming. This is to provide convenience for the following matrix computing.

The second step is to calculate the variances of Y of neighboring pixels for each pixel. At the same time, the implementer also recorded which pixels are scrbbled.

1. A pixel is thought to be scribbled if the sum of its absolute values of U and V are greater than 10^{-4} (after being normalized).

2. The central pixel itself is excluded from variance calculating.

The third step is to calculate the weights ($w_{\mathbf{rs}}$).

1. If the weight is smaller than a threshold value (10^{-6}) then assign it to the weight. Otherwise the matrix might be singular.

2. Operate the summation-normalization to the weights in one window. That is, all weights in one window should sum up to be 1.

3. If σ is 0, assign the weight to be 0.

4. It is reasonable to multiply $\sqrt{2}$ on the diagonal weights, though this does not make a big difference.

The fourth step is to set up the optimizing goal. As mentioned above, originally, the implementer tried to directly obtain the global minimum of equation (1) by using scipy.optimize. This implementation is simple: only an expression of $J(U)$ and an initial point (a zero vector whose length is # unscribbled pixels). However, it requires a memory space of over 80 Gb. Professor Zizhuo Wang gave the implementer two suggestions:

1. Divide the original image into 4 (or 16, e.t.c.) sub-images and then solve them one by one.

2. Pack neighboring (3×3) pixels as one pixel when solving the optimizing goal. Then somehow recover the resolution...(the implementer forgot the details)

The implementer comments that the above two suggestions are thought to be practical in industry. The first one is usually called quad-tree and is a wide-spread solution in collision detection in video games. The implementer is unfamiliar with the second one but intuitively guesses that it should be a classic strategy in CV.

The implementer finally turned to the internet and modified the optimizing goal. Equation (3) can be expressed in the form of $Ax = b$ and then solved by numpy. Say the sizes of both m1 and m2 are 399×299 ($n = 119,301$ pixels in each), then x is a vector of $n \times 1$. A is the weight matrix that records $w_{\mathbf{rs}}$ and their indices. b is a $n \times 1$ vector of U or V channel of m1. A few specifications:

1. Starting from 0, index the pixels from left to right and from top to bottom. E.g., the index of the pixel at row 2 colomn 1 is 399. x and b represent all pixels in terms

of their indices.

2. A is a sparse matrix (scipy.sparse). Each row of A is full of zeros except that:
   (a) the diagonal element is 1.
   (b) the elements at neighboring pixels' indices should be the corresponding weights' negative values ($-w_{\mathbf{rs}}$).

   A is initialized as an empty lil matrix. Then elements are inserted into A row by row. Before A is passed to numpy.linalg.solve(), it should be transformed into csr or csc. lil is suitable for initializing or editing but not for calculating. csr and csc are on the contrary.

3. It is an extremely frequent bug to see A being "exactly singular". If this happens, there must be some mistakes in the above steps.

The final step is to transform x back in RGB channel and output the image.

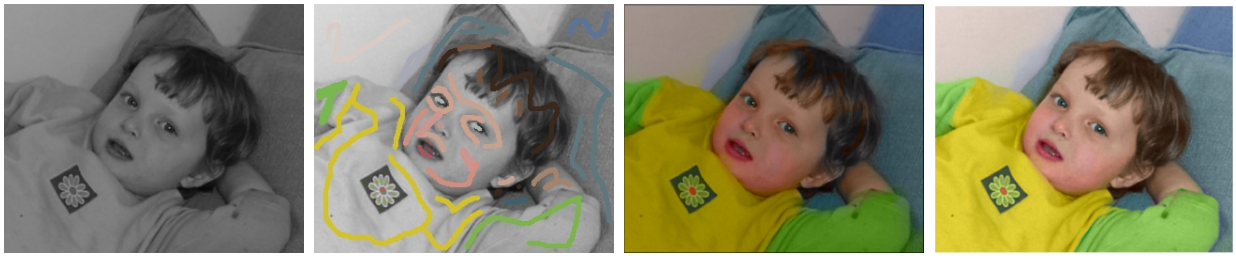
1. Every element in x should be within $[0, 1]$. Otherwise, go back and check the above steps.

2. If the ouput image is full-black or full-white, go to check the normalization steps. It means the RGB values of all pixels are too close that the data type of 'uint8' cannot tell the difference.

3. If the output image has obvious marks of scribbles, go to check A's values and the summation-normalization steps.

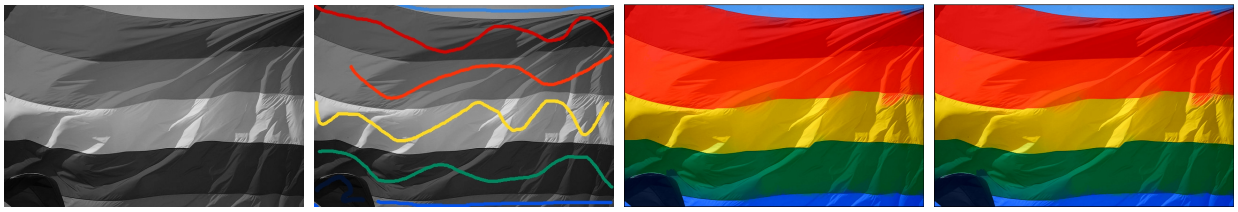
4. The function of "yuv_channels_to_rgb()" is copied and modified from [2].

3 Results



(a) Bw Image **(b)** Scribbled Image **(c)** Output Image **(d)** '04 paper's output image

Figure 1: Comparison between the outputs of this implementation (c) and the paper's original implementation (d). Since the implementer's bw image is darker than the author's, (c) seems darker than (d).



(a) Bw Image **(b)** Scribbled Image **(c)** Output Image **(d)** Others' output image

Figure 2: Comparison between the outputs of this implementation (c) and the referenced implementation from csdn (d).

4 Python Codes

```
from imageio import imread
import colorsys
import numpy as np
from scipy import sparse
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt

m1_ = 'Image_Colorization_Using_Optimization\\resources\\baby1.bmp'
m2_ = 'Image_Colorization_Using_Optimization\\resources\\baby.bmp'

m1 = imread(m1_)
m2 = imread(m2_)

m1 = np.array(m1, dtype='float64')/255
m2 = np.array(m2, dtype='float64')/255

channel_Y, _, _ = colorsys.rgb_to_yiq(m2[:, :, 0], m2[:, :, 1], m2[:, :, 2])
_, channel_U, channel_V = colorsys.rgb_to_yiq(m1[:, :, 0], m1[:, :, 1], m1[:, :,
    2])

m1 = np.dstack((channel_Y, channel_U, channel_V))

W, L, _ = m1.shape
# W = 265, L = 320

# record the coordinates of the scribbles
scr_index = set()
scr_bool = []
# sigmas records the variances of the intensities in a window (9x9) around a
    pixel
sigmas = []

def xy2idx(x, y):
    return x * L + y
```

```
def idx2xy(idx):
    return idx // L, idx % L

def calc_mius_and_sigmas():
    count_scr = 0
    count_all = 0
    for w in range(W):
        for l in range(L):
            if abs(m1[w][l][1]) + abs(m1[w][l][2]) > 1e-4:
                count_scr += 1
                scr_index.add(count_all)
                scr_bool.append(True)
            else:
                scr_bool.append(False)

            nbhd = []
            d = 1
            for i in range(max(w - d, 0), min(w + d + 1, W)):
                for j in range(max(l - d, 0), min(l + d + 1, L)):
                    if i != w or j != l:
                        nbhd.append(m1[i][j][0])

            var = np.var(nbhd)
            sigmas.append(var)
            count_all += 1

    return count_scr

calc_mius_and_sigmas()

M = W * L

A = sparse.lil_matrix((M, M))

def wt_func(sigma, Y_r, Y_s):
    if sigma != 0:
        res = np.exp(-(Y_r - Y_s)**2 / (2 * sigma**2))
        return res if res > 1e-6 else 1e-6
    else:
        return 0

def calc_weights_matrix():
    idx = 0
    for w in range(W):
        for l in range(L):
            if not scr_bool[idx]:
                sigma_r = sigmas[idx]
```

```
                # iterate over a window around[w][l]:
                nbhd_w = []
                nbhd_idx = []
                d = 1

                for i in range(max(w - d, 0), min(w + d + 1, W)):
                    for j in range(max(l - d, 0), min(l + d + 1, L)):
                        if i != w or j != l:
                            win_idx = xy2idx(i, j)
                            w_rs = wt_func(sigma_r, m1[w][l][0], m1[i][j][0])

                            d_x = i - w
                            d_y = j - l
                            # diagonal
                            if (d_x == -1 and d_y == 1) or\
                                (d_x == 1 and d_y == 1) or\
                                (d_x == 1 and d_y == -1) or\
                                (d_x == -1 and d_y == -1):
                                w_rs /= np.sqrt(2)

                            nbhd_w.append(w_rs)
                            nbhd_idx.append(win_idx)

                sum_ = sum(nbhd_w)
                if sum_ != 0:
                    nbhd_w /= sum_
                else:
                    len_ = len(nbhd_w)
                    nbhd_w = np.full(len_, 1/len_)

                for _ in range(len(nbhd_w)):
                    A[idx, nbhd_idx[_]] = -nbhd_w[_]

            A[idx, idx] = 1
            idx += 1

calc_weights_matrix()

A = A.tocsc()

b_u = np.zeros(M, dtype='float64')
b_v = np.zeros(M, dtype='float64')

for _ in range(M):
    if _ in scr_index:
        x, y = idx2xy(_)
```

```
        b_u[_] = m1[x][y][1]
        b_v[_] = m1[x][y][2]

x_u = spsolve(A, b_u)
x_v = spsolve(A, b_v)

x_u = x_u.reshape((W, L))
x_v = x_v.reshape((W, L))

new = np.zeros((W, L, 3))
new[:, :, 0] = m1[:, :, 0]
new[:, :, 1] = x_u[:, :]
new[:, :, 2] = x_v[:, :]

def yuv_channels_to_rgb(cY,cU,cV):
    ansRGB = [colorsys.yiq_to_rgb(cY[_], cU[_], cV[_]) for _ in range(M)]
    ansRGB = np.array(ansRGB)
    pic_ansRGB = np.zeros((W, L, 3))
    pic_ansRGB[:, :, 0] = ansRGB[:, 0].reshape((W, L))
    pic_ansRGB[:, :, 1] = ansRGB[:, 1].reshape((W, L))
    pic_ansRGB[:, :, 2] = ansRGB[:, 2].reshape((W, L))
    return pic_ansRGB

new = yuv_channels_to_rgb(new[:, :, 0].reshape(M), new[:, :, 1].reshape(M),
    new[:, :, 2].reshape(M))
imgplot = plt.imshow(new)
plt.savefig('Image_Colorization_Using_Optimization\\python\\Output_ICUO.jpg')
plt.show()
```

5 References

Levin, L., Lischinski, D., Weiss, Y. 2004. Colorization using Optimization. *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers.*

http://blog.sws9f.org/computer-vision/2017/09/07/colorization-using-optimization-python.html#: :text=