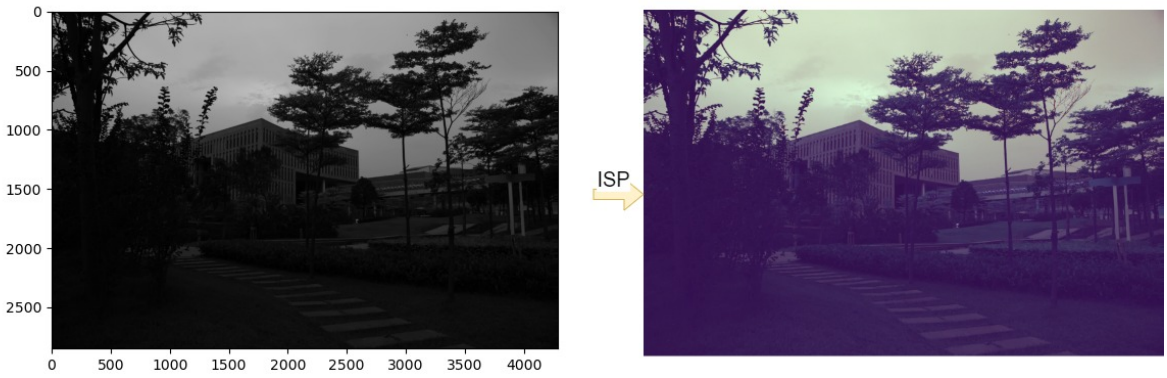
Homework 2 Image Signal Processing

[Zhen Tong](#) 120090694



In this project, we build a Image Signal Processing Pipeline with `Python`. In this ISP pipeline project, our contribution includes:

- Dead Pixel Correction
- Black Level Compensation
- Anti Aliasing Filter
- Auto White Balance and Gain Control
- Color Filter Array Interpolation
- Edge Enhancement for Luma
- Brightness/Contrast Control for Luma
- False Color Suppresion for Chroma
- Hue/Saturation control for Chroma
- RGB YUV image converting
- Acceleration(26.15 sec) **10.47 times** faster than [OpenISP](#)(273.76 sec)

Dead Pixel Correction

Since the Sensor is a physical device, it is difficult to avoid bad points.

Idea

The DPC is in two steps: detect the dead pixel, and recover it. Detection is compute the difference of each pixel and if the all the neighbor difference is bigger than the threshold, it is a dead pixel:

$$diff_x(P_{i,j}) = |P_{i,j} - P_x|$$

$$P_{i,j} = \begin{cases} Dead & \wedge_{x \in neighbor} diff_x(P_{i,j}) \\ Not\ Dead & Otherwise \end{cases}$$

- **Mean Method**

Assign the value of the dead pixel with neighbor mean

- **Gradient Method**

Compute the second order gradient of the image at the dead pixel in four direction. Assign the dead pixel as the average of the most "smooth "direction, which is the minimal one of the four absolute value.

$$dv = |2 * p0 - p2 - p7|$$
$$dh = |2 * p0 - p4 - p5|$$
$$ddl = |2 * p0 - p1 - p8|$$
$$ddr = |2 * p0 - p3 - p6|$$

Acceleration

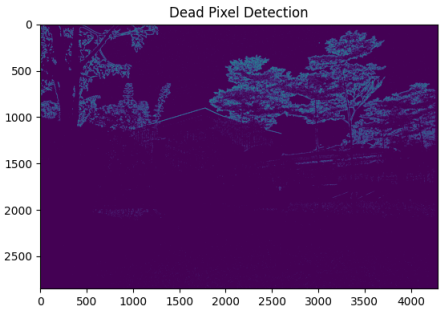
Because the DPC process need to iterate all the pixel to find the dead pixel, which is $O(N^2)$ time complexity. To speedup, we need to avoid the nested for loop in `python`, instead use the vectorize compute with `numpy`. First we generate 9 copy of the image matrix, and generate all the position to be compute, and record them in `y_ids`, and `x_ids`. We compute the mask matrix for all the dead pixel.

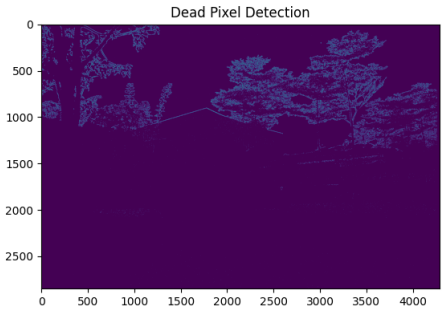
```
y_ids, x_ids = np.meshgrid(np.arange(img_pad.shape[0] - 4),
np.arange(img_pad.shape[1] - 4))
p0 = img_pad[y_ids + 2, x_ids + 2].astype(int)
p1 = img_pad[y_ids,     x_ids].astype(int)
p2 = img_pad[y_ids,     x_ids + 2].astype(int)
p3 = img_pad[y_ids,     x_ids + 4].astype(int)
p4 = img_pad[y_ids + 2, x_ids].astype(int)
p5 = img_pad[y_ids + 2, x_ids + 4].astype(int)
p6 = img_pad[y_ids + 4, x_ids].astype(int)
p7 = img_pad[y_ids + 4, x_ids + 2].astype(int)
p8 = img_pad[y_ids + 4, x_ids + 4].astype(int)

mask = (np.abs(p1 - p0) > self.thres) & \
       (np.abs(p2 - p0) > self.thres) & \
       (np.abs(p3 - p0) > self.thres) & \
       (np.abs(p4 - p0) > self.thres) & \
       (np.abs(p5 - p0) > self.thres) & \
       (np.abs(p6 - p0) > self.thres) & \
       (np.abs(p7 - p0) > self.thres) & \
       (np.abs(p8 - p0) > self.thres)
if self.mode == 'mean':
    dpc_img[mask] = (p2 + p4 + p5 + p7) / 4
elif self.mode == 'gradient':
    dv = abs(2 * p0 - p2 - p7)
    dh = abs(2 * p0 - p4 - p5)
    ddl = abs(2 * p0 - p1 - p8)
    ddr = abs(2 * p0 - p3 - p6)

    min_indices = np.argmin([dv, dh, ddl, ddr], axis=0)
    mask1 = (min_indices == 0) & mask
    mask2 = (min_indices == 1) & mask
    mask3 = (min_indices == 2) & mask
    mask4 = (min_indices == 3) & mask

    dpc_img[mask1] = ((p2 + p7 + 1) / 2)[mask1]
    dpc_img[mask2] = ((p4 + p5 + 1) / 2)[mask2]
    dpc_img[mask3] = ((p1 + p8 + 1) / 2)[mask3]
```

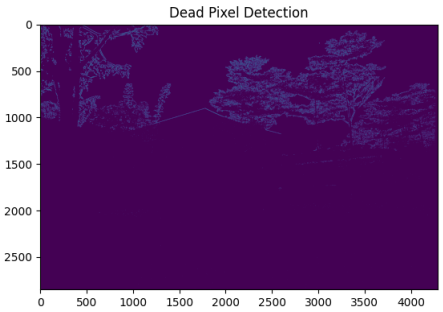
```
    dpc_img[mask4] = ((p3 + p6 + 1) / 2)[mask4]
```

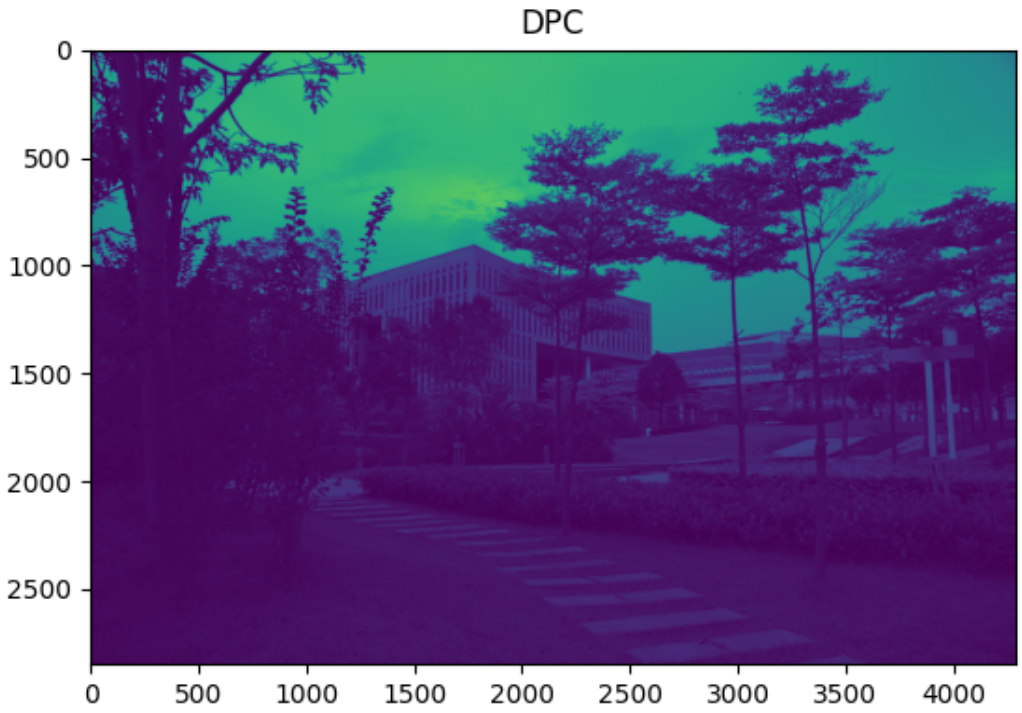
Ouptut

The following figures are "***dead pixel*** when "threshold = 100, 150, 200. As you can see, they are not dead pixel most of the time, instead, they are the edges of the image. Because the DPC detection use gradient, and smooth the image unintentionally, we shouldn't set it too low.









Black Level Compensation

Due to the existence of Sensor leakage current, the lens has just been put into a completely black environment, and the original output data of the Sensor is not 0; And we want the original number to be 0 when it's all black.

$$
\begin{aligned}
R' &= R + B_{\text{offset}} \\
Gr' &= Gr + Gr_{\text{offset}} + \alpha R \\
Gb' &= Gb + Gb_{\text{offset}} + \beta B \\
B' &= B + B_{\text{offset}}
\end{aligned}
$$

Anti Aliasing Filter

Do the convolution of the image with the kernel to avoid aliasing.

$$
\begin{bmatrix}
1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 8 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1
\end{bmatrix}
$$

The acceleration of convolution is the same in Edge Enhancement, see later.

Auto White Balance and Gain Control

By adjust the scale of RGrGbB, the quality of the image can improve.

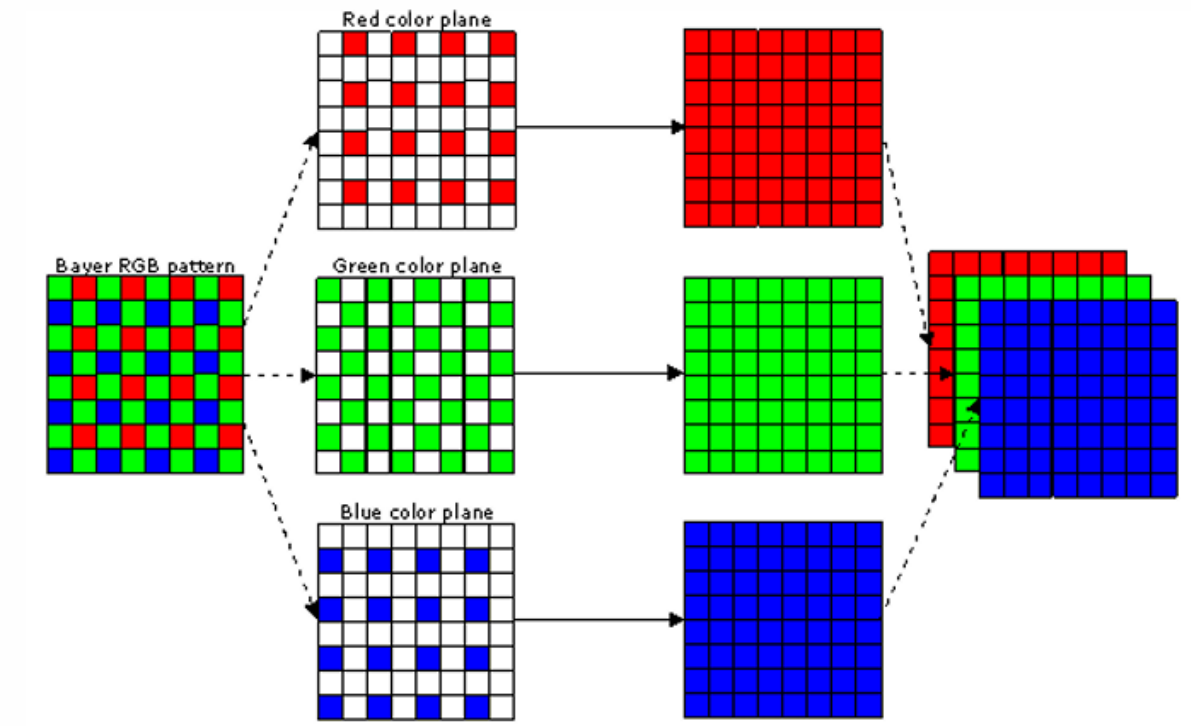
$$
\begin{aligned}
R &= R \times r_{\text{gain}} \\
B &= B \times b_{\text{gain}} \\
Gr &= Gr \times gr_{\text{gain}} \\
Gb &= Gb \times gb_{\text{gain}}
\end{aligned}
$$

However, this process need hyperparameter, which depends on color expertise.

Color Filter Array Interpolation

Idea

We need to interpolate the `[height, width, 3]` RGB image from the raw `[height, width]` Matrix. The matrix is in `rggb` bayer pattern, which is:

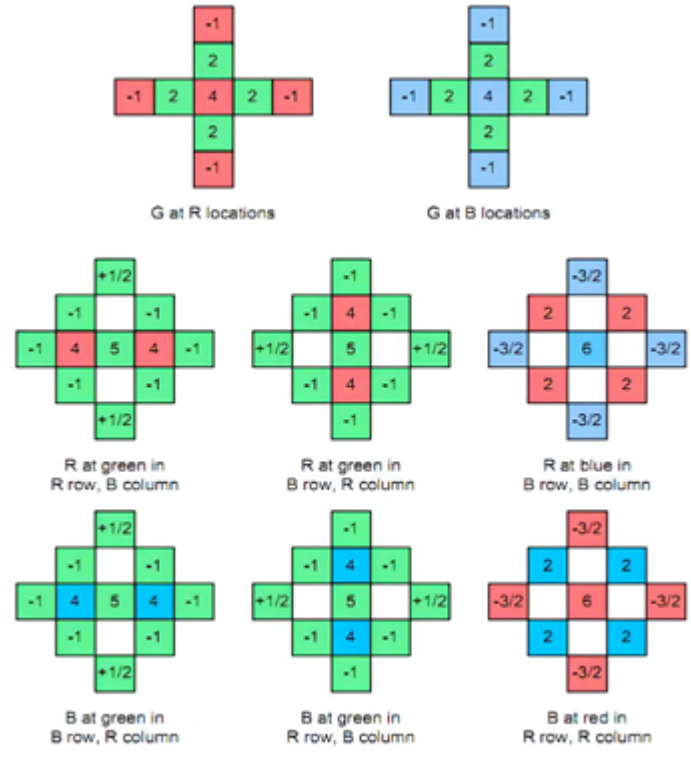


What we need to is to interpolate from the 2 dimension array to 3 dimension with *Malvar* (2004) demosaicing algorithm.

There are several situation of interpolation:

Color in Bayer/Target Color	R	G	B
R	r	g_r	b_r
Gr	r_{gr}	g_{gr}	b_{gr}
Gb	r_{gb}	g_{gb}	b_{gb}
B	r_b	g_b	b

Following the weight of the algorithm in the figure,



For example G at R location is:

$$g_r = \frac{4 \cdot X_{y,x} - X_{y-2,x} - X_{y,x-2} - X_{y+2,x} - X_{y,x+2} + 2 \cdot (X_{y+1,x} + X_{y,x+1} + X_{y-1,x} + X_{y,x-1})}{8}$$

Acceleration

In the interpolation, intuitively we need to use the nested for loop, which is in N^2 time. To speedup the code in python, we can use the **Indices Generationi**: Meshgrid is used to generate indices for accessing elements of the 2D image in a structured manner. This generates arrays `y_indices` and `x_indices` representing the y and x coordinates of pixels in the padded image.

```
y_indices, x_indices = np.meshgrid(np.arange(0, img_pad.shape[0]-4-1, 2),
np.arange(0, img_pad.shape[1]-4-1, 2))
```

The meshgrid will generate the 2d coordinate of step-size = 2. Then, we can extract the rs, grs, gbs, and bs matrix from the bayer matrix by:

```
r = img_pad[y_indices + 2, x_indices + 2]
gr = img_pad[y_indices + 2, x_indices + 3]
gb = img_pad[y_indices + 3, x_indices + 2]
b = img_pad[y_indices + 3, x_indices + 3]
```

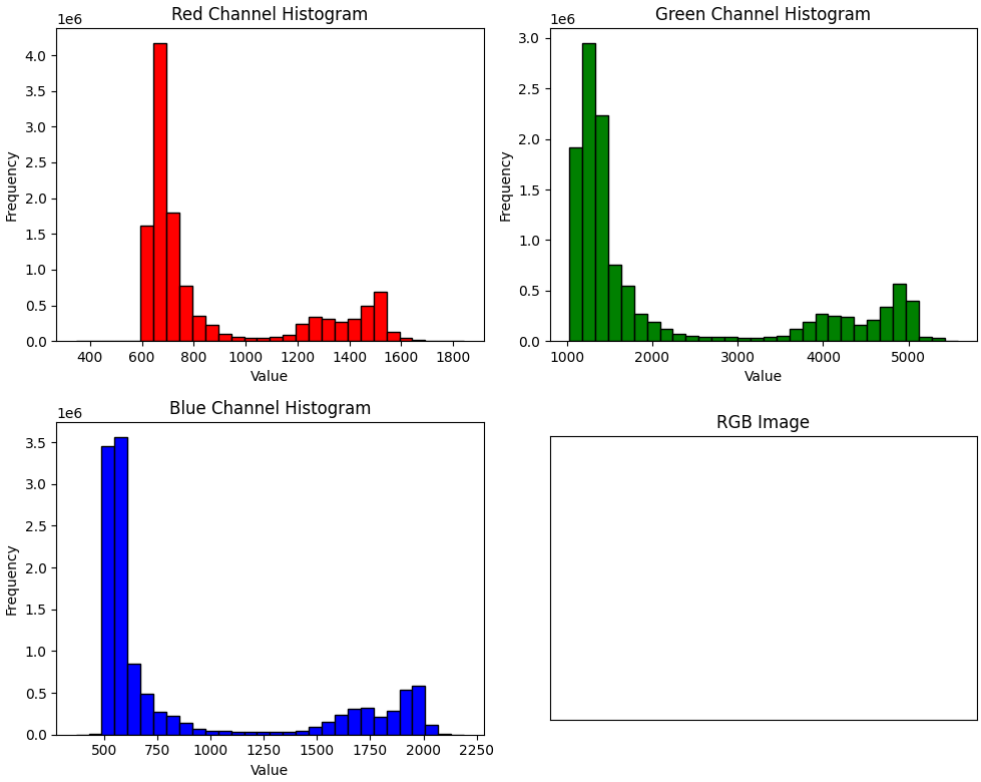
Then, we can interpolate in matrix level for 4 times:

```
cfa_img[y_indices, x_indices, :] = malvar('r', r, y_indices + 2, x_indices + 2,
img_pad)
cfa_img[y_indices, x_indices + 1, :] = malvar('gr', gr, y_indices + 2, x_indices
+ 3, img_pad)
cfa_img[y_indices + 1, x_indices, :] = malvar('gb', gb, y_indices + 3, x_indices
+ 2, img_pad)
cfa_img[y_indices + 1, x_indices + 1, :] = malvar('b', b, y_indices + 3,
x_indices + 3, img_pad)
```

After all, remember to clip the image.

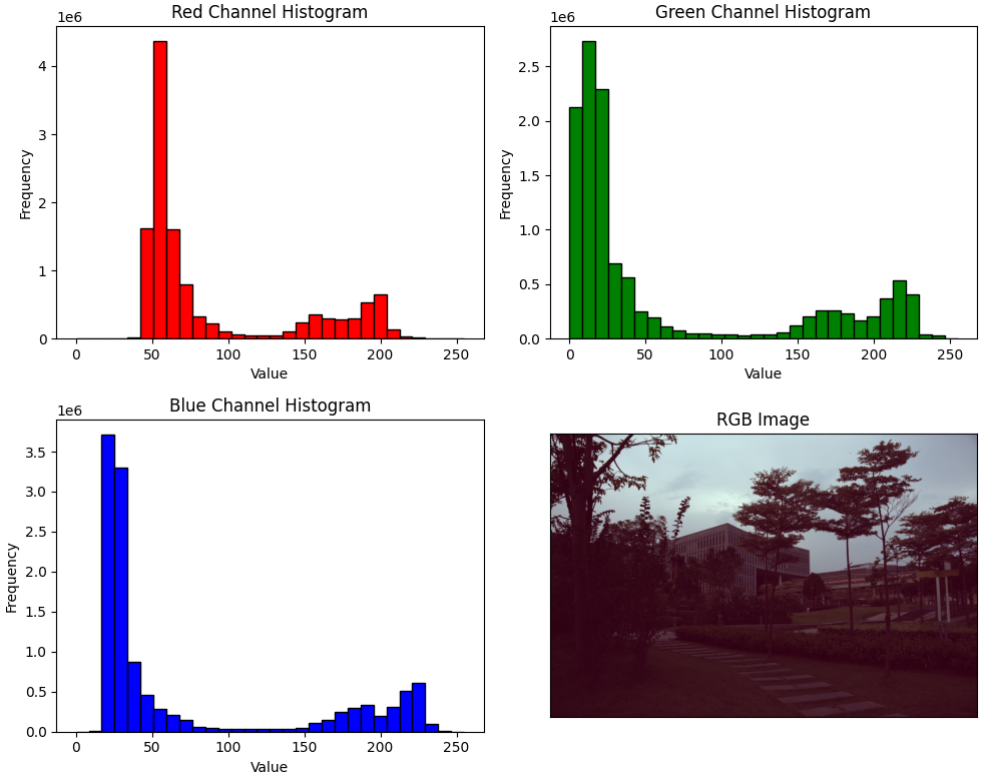
Normalization

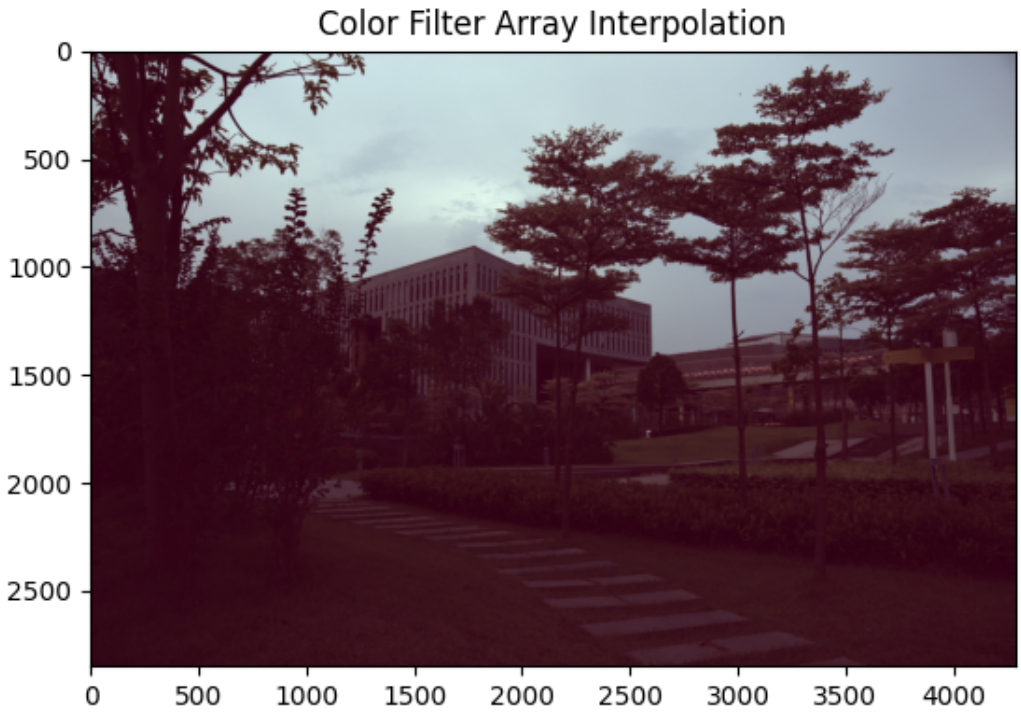
Notably, without normalization, the `matplotlib.pyplot` can not transfrom the scale of CFA output to the 0-255, or 0-1 scale (all white in the RGB Image bellow), we need to manually normalize it into 0-255 or 0-1.



After normalization:

$$X = \frac{X - min(X)}{max(X) - min(X)}$$





RGB2YUV

Idea

We need to change to YUV for the further process but not direct using RGB, because YUV can separate the Chroma(color information `Cr` , `Cb`) and Luma(intensity information `Y`). Notice that the matrix transform of range `[0, 1]` and `[0, 255]` is different, here our RGB range is `[0, 255]`

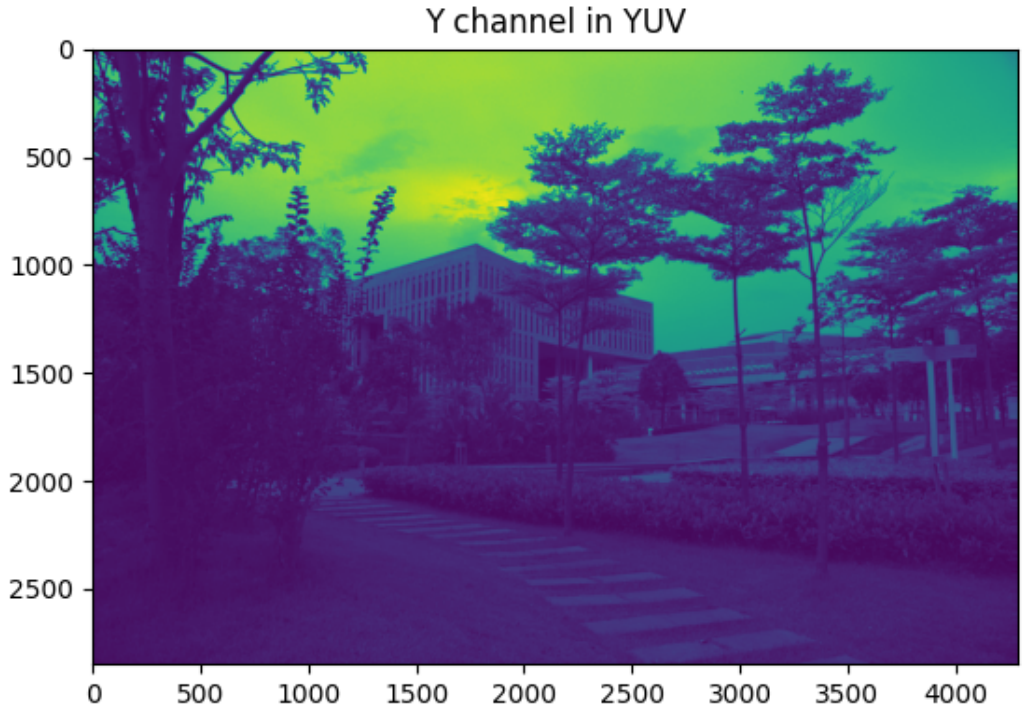
Acceleration

Change the pixel by pixel transform into matrix transform

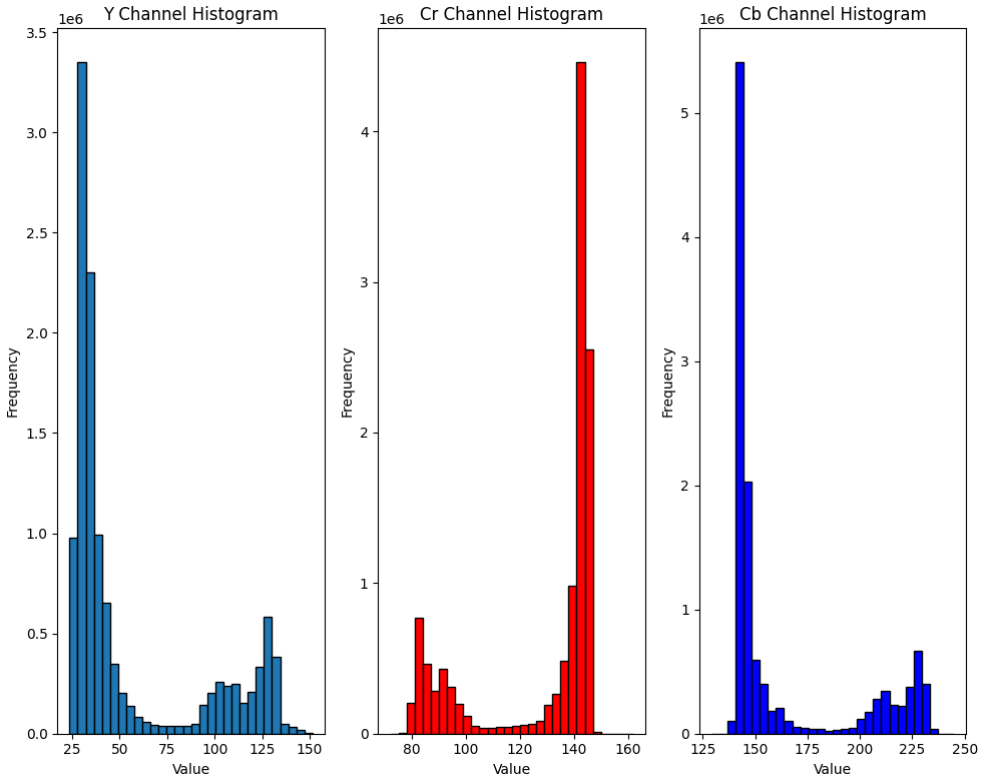
```
def RGB2YUV(img:np.ndarray):
    matrix = np.array([[66, 129, 25],
                       [-38, -74, 112],
                       [112, -94, -18]], dtype=np.int32).T  # x256
    bias = np.array([16, 128, 128], dtype=np.int32).reshape(1, 1, 3)
    np.right_shift(rgb_image @ matrix, 8) + bias
```

Output

In the following content, when we show the `YUV` output, we are actually showing the `YUV[:, :, 0]`, the first channel Luma, but not the 3-Channel image. Next, we try to improve the edge, contrast, and brightness quality of the image one the first channel of YUV.



We can draw the histogram of the `YUV` channels, to monitor the shape change of the distribution later.

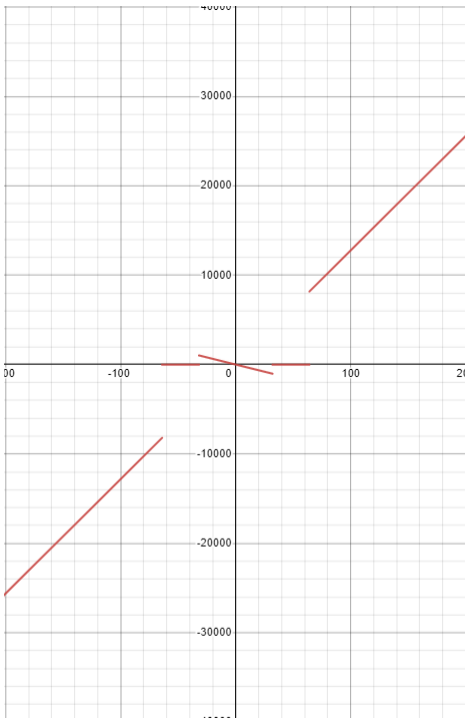


Edge Enhancement

Idea

The edge enhancement includes two parts: compute the convolution of edge filter, and compute the edge map loop up table (EMLUT). With gains: $g_1, g_2 = 32, 128$, thresholds: $x_1, x2 = 32, 64$, the transform function is in shape of:

$$f(x) = \begin{cases} 128x & \text{if } x < -64 \\ 0 & \text{if } -64 < x < -32 \\ -32x & \text{if } -32 < x < 32 \\ 0 & \text{if } 32 < x < 64 \\ 128x & \text{if } x > 64 \end{cases}$$



Speed up

First, in the convolution part, the filter kernel is a 3x5 matrix. We can again use the meshgrid to divide the 15 channels of image from the first channel of YUV image. Each channel of the 15 channel corespond to one weight of the kernel filter. Then compute the weighted sum

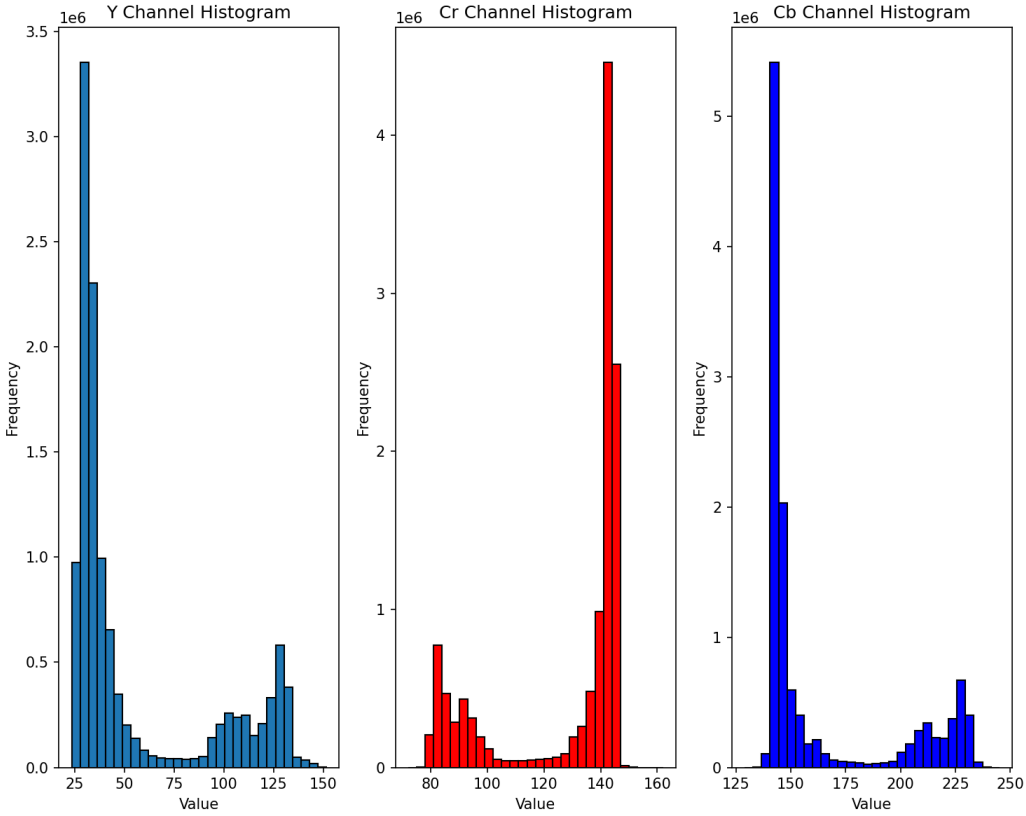
```
def conv3x5(img, kernel:np.ndarray):
    img_pad = np.pad(img, ((1, 1), (2, 2)), 'reflect')
    y_indices, x_indices = np.meshgrid(np.arange(0, img.shape[0],
dtype=np.int32), np.arange(0, img.shape[1], dtype=np.int32))
    a1 = img_pad[y_indices, x_indices]
    a2 = img_pad[y_indices, x_indices+1]
    a3 = img_pad[y_indices, x_indices+2]
    a4 = img_pad[y_indices, x_indices+3]
    a5 = img_pad[y_indices, x_indices+4]
    a6 = img_pad[y_indices+1, x_indices]
    a7 = img_pad[y_indices+1, x_indices+1]
    a8 = img_pad[y_indices+1, x_indices+2]
    a9 = img_pad[y_indices+1, x_indices+3]
    a10 = img_pad[y_indices+1, x_indices+4]
    a11 = img_pad[y_indices+2, x_indices]
```

```
    a12 = img_pad[y_indices+2, x_indices+1]
    a13 = img_pad[y_indices+2, x_indices+2]
    a14 = img_pad[y_indices+2, x_indices+3]
    a15 = img_pad[y_indices+2, x_indices+4]

    kernel = kernel.flatten()
    weighted_sum = np.sum(kernel[:, np.newaxis, np.newaxis] * np.array([a1, a2,
a3, a4, a5,a6, a7, a8, a9, a10,a11, a12, a13, a14, a15]), axis=0)
```

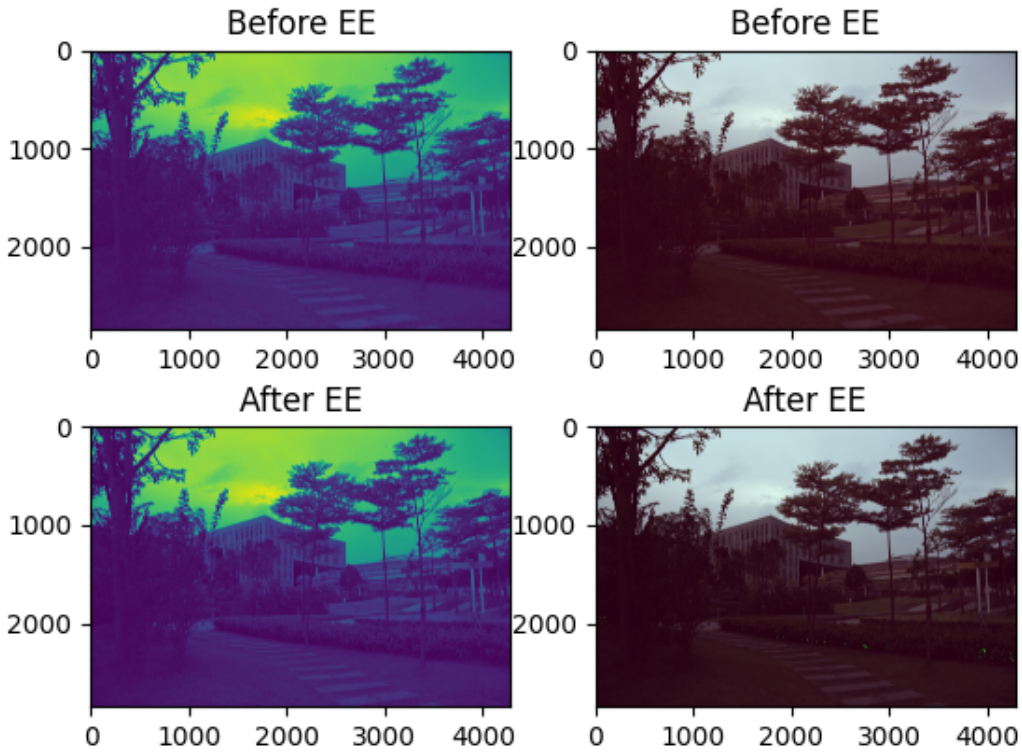
Then the piecewise function transform to the convolution output can be speed up with mask operation in numpy. The mask records all the pixels that in the range of the given condition. Then it can be used to assign the value by indices accessing.

```
# Compute lut based on conditions
mask1 = em_img < -self.thres[1]
mask2 = (em_img >= -self.thres[1]) & (em_img <= -self.thres[0])
mask3 = (em_img >= -self.thres[0]) & (em_img <= self.thres[0])
mask4 = (em_img >= self.thres[0]) & (em_img <= self.thres[1])
mask5 = em_img > self.thres[1]
lut[mask1] = self.gain[1] * em_img[mask1]
lut[mask2] = 0
lut[mask3] = self.gain[0] * em_img[mask3]
lut[mask4] = 0
lut[mask5] = self.gain[1] * em_img[mask5]
```

Output



After computing the `EMLUT(x)`, clip the `EMLUT(x)` and add it to the image as an output:



Brightness/Contrast Control

Idea

Brightness control is :

$$Y \times \text{contrast} + \text{brightess}$$
$$i.e.\ \alpha Y + \beta$$

This operation is first move the distribution to the right of `brightness` scale, and then enlarge the variance with `contrast` scale. In the example image, I choose brightness as 12.3208, and contrast = 0.9249.

Here we minus the median value to control the moving shape of the distribution. One tips here for the adjustment of the parameter is that you can first pin some point of the distribution, and the calculate the parameter using the least square method.

In the histogram above, we can observe that there is a high frequent around 30-40, and we don't want the high value part moving too far from 150. Therefore we can set a group of function from the original distribution to the target distribution:

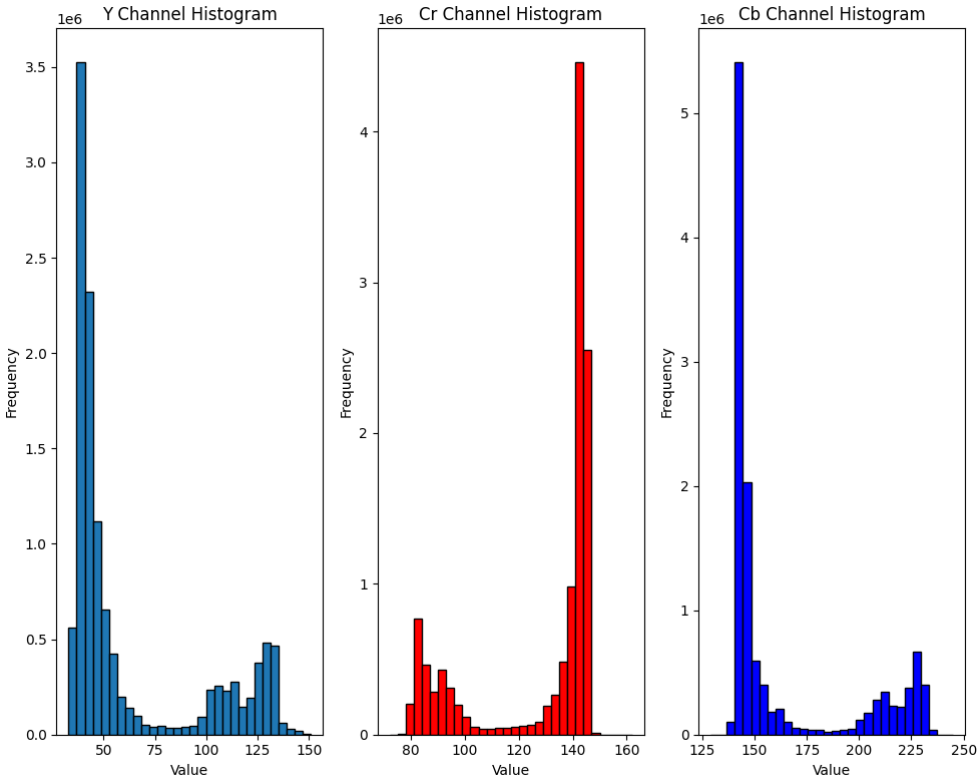
$$40\alpha + \beta = 50$$
$$75\alpha + \beta = 80$$
$$125\alpha + \beta = 130$$
$$150\alpha + \beta = 150$$

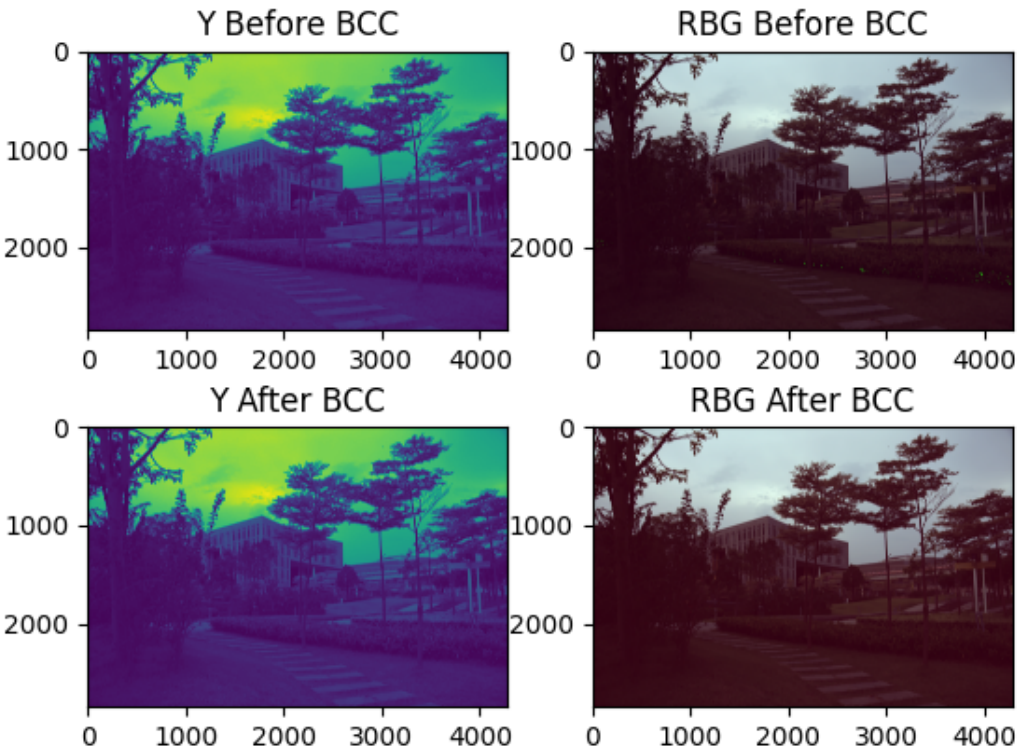
By the LSM, we can get the $\beta = 12.3208$, and $\alpha = 0.9249$.

Notice that now the clip range is no longer `[16, 235]`, the `Y` channel is in range of `[16, 235]`.

Output

In the output image, the sky is brighter, and the shade of tree is darker (left of the image). In the histogram, the darker part is moved from around 30 to around 45.





False Color Suppression

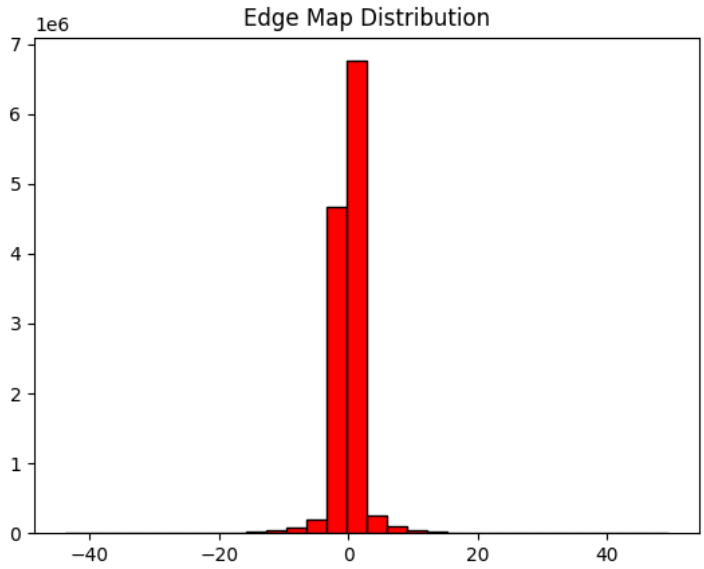
Idea

False Color was caused from the CFA interpolation. In short, false color is introduced into the image when the predicted color is inaccurate, especially in the gray area.

First compute the gain of `u` dimension and `v` dimension by this piecewise function:

$$Cb, Cr = \begin{cases} Cb, Cr & |\mathrm{EM}(y,x)| \leq \mathrm{fcsEdge}_1 \\ \frac{\mathrm{gain} \times (|EM| - 128)}{65536} + 128 & \mathrm{fcsEdge}_1 < |\mathrm{EM}(y,x)| < \mathrm{fcsEdge}_2 \\ 0 & |\mathrm{EM}(y,x)| \geq \mathrm{fcsEdge}_2 \end{cases}$$

In this figure the edge map distribution is:



Therefore, I set the $\mathrm{fcsEdge}_1 = 16$, and $\mathrm{fcsEdge}_2 = 32$

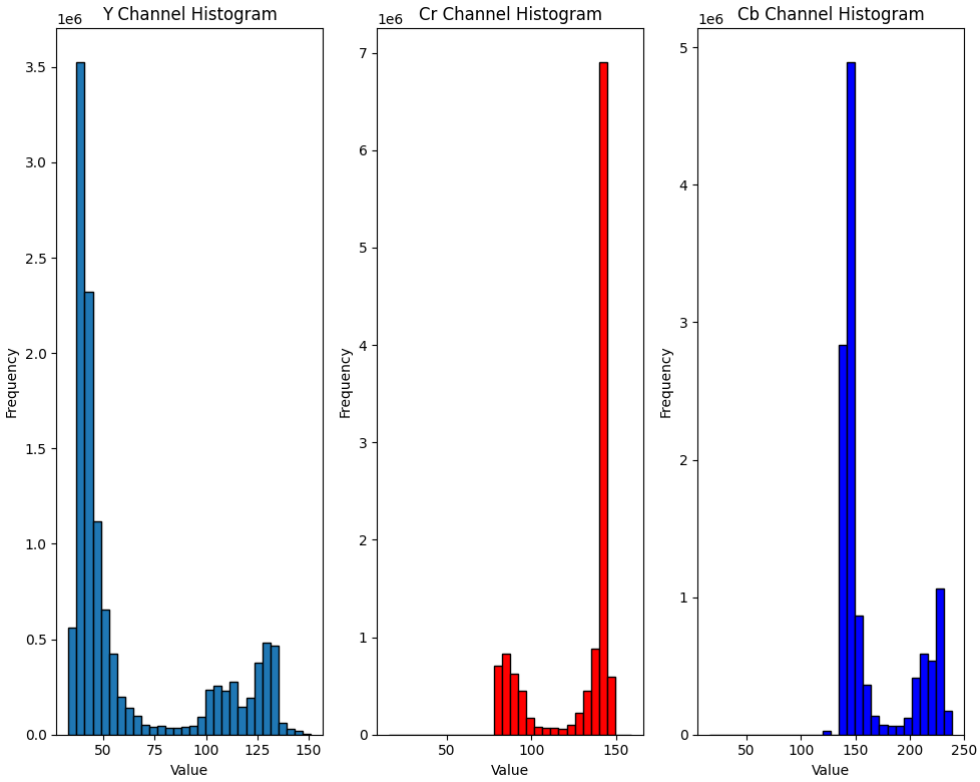
Acceleration

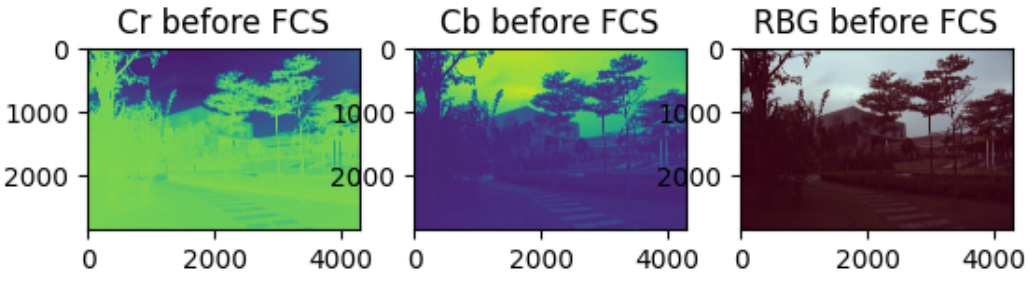
Again use the mask boolean indices to assign the value in a vectorization way:

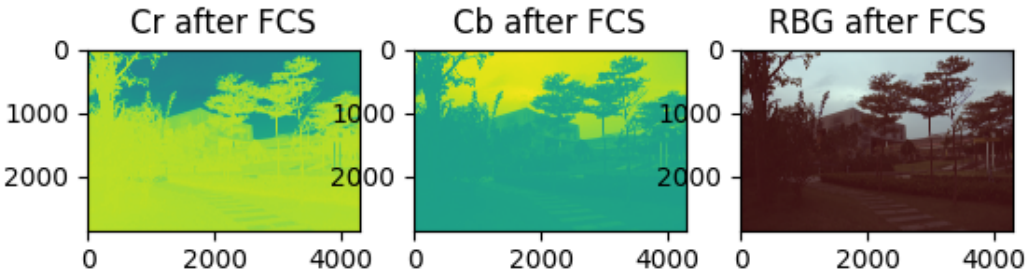
```
absEM = np.abs(self.edgemap)
mask1 = absEM <= self.fcs_edge[0]
mask2 = np.logical_and(
    absEM > self.fcs_edge[0],
    absEM < self.fcs_edge[1])
mask3 = absEM >= self.fcs_edge[1]

fcs_img[mask1] = self.img[mask1]
fcs_img[mask2, 0] = self.gain*(absEM[mask2]-128)/65536 + 128
fcs_img[mask2, 1] = self.gain*(absEM[mask2]-128)/65536 + 128
fcs_img[mask3] = 0

# np.clip(fcs_img, 0, 255, out=fcs_img)
np.clip(fcs_img, 16,239, out=fcs_img)
```

Output

In the output, we can see the `Cb` part improved, and the darker part in `Cb` become brighter. In the RGB level, the darker part become less red, and more blue.







Hue/Saturation control

Idea

HSC is in two step: the Hue control, and the Saturation control. In the Hue tuning, the two channels range are first moved from `[0-256]` to `[-128, 128]`, then using the sine of Hue parameter to get the new `Cr`, and `Cb` channel value.

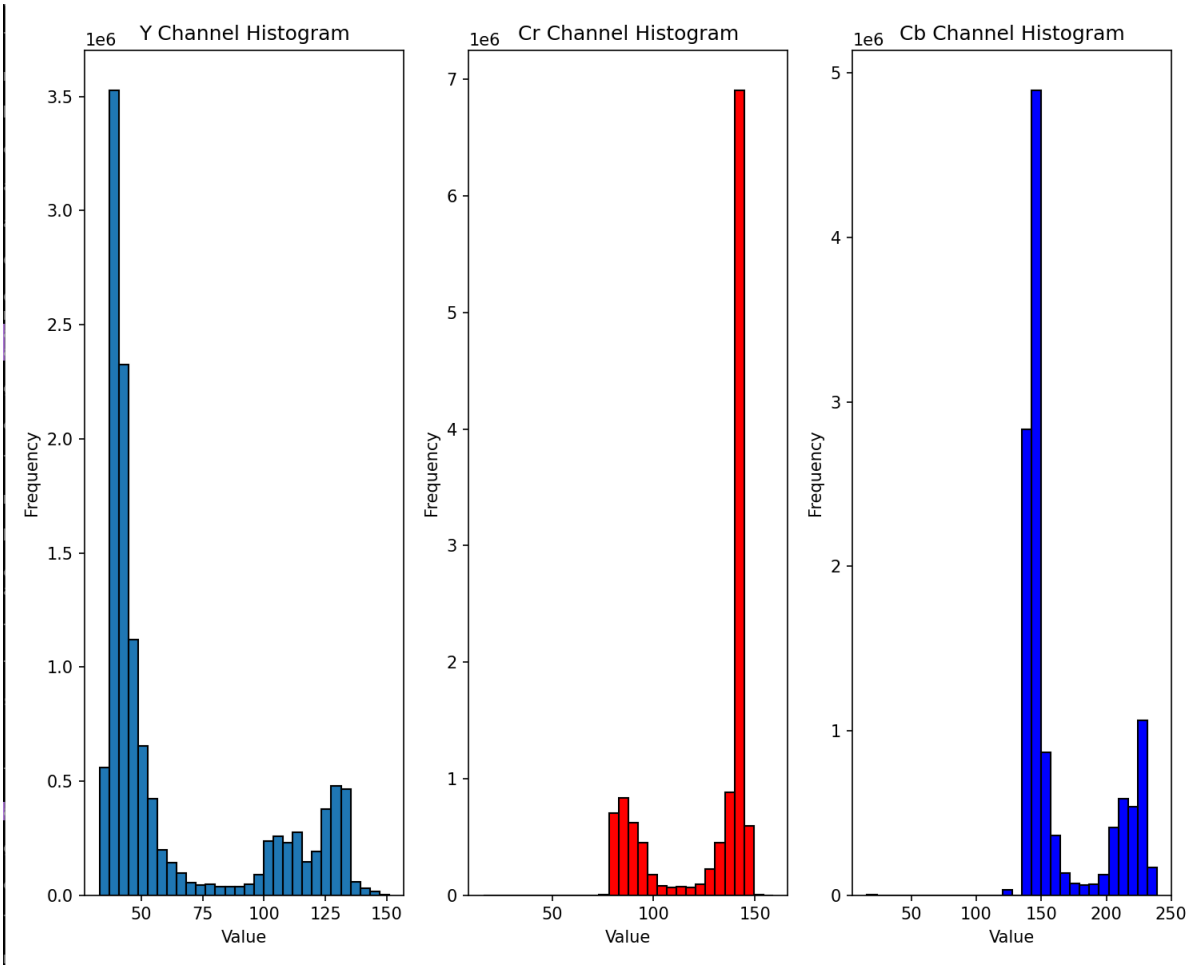
$$Y'_{Cr} = (Y_{Cr} - 128) \times \cos(h) + (Y_{Cb} - 128) \times \sin(h) + 128$$
$$Y'_{Cb} = (Y_{Cb} - 128) \times \cos(h) + (Y_{Cr} - 128) \times \sin(h) + 128$$

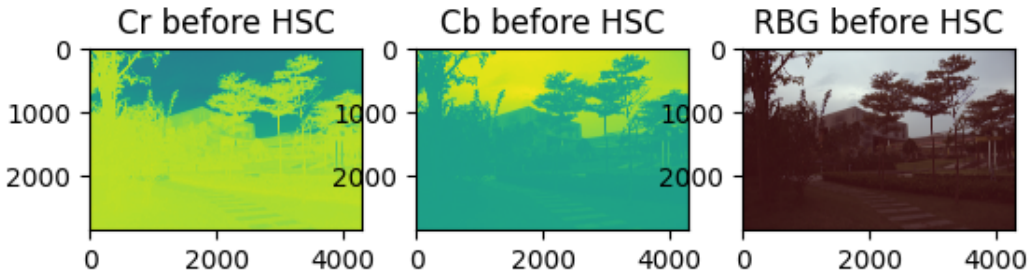
Then in the Saturation control step, with a linear transform, we can output the tuned image of UV channels:

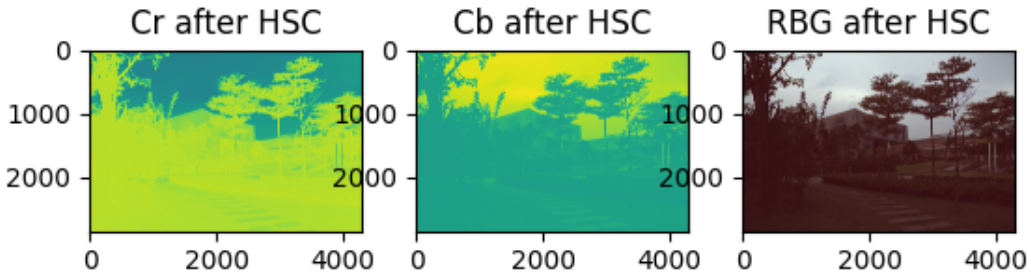
$$Y = \frac{s(Y - 128)}{256} + 128$$

Output

In the image below, we can observe both Cr and Cb showed more details, and more contrast.

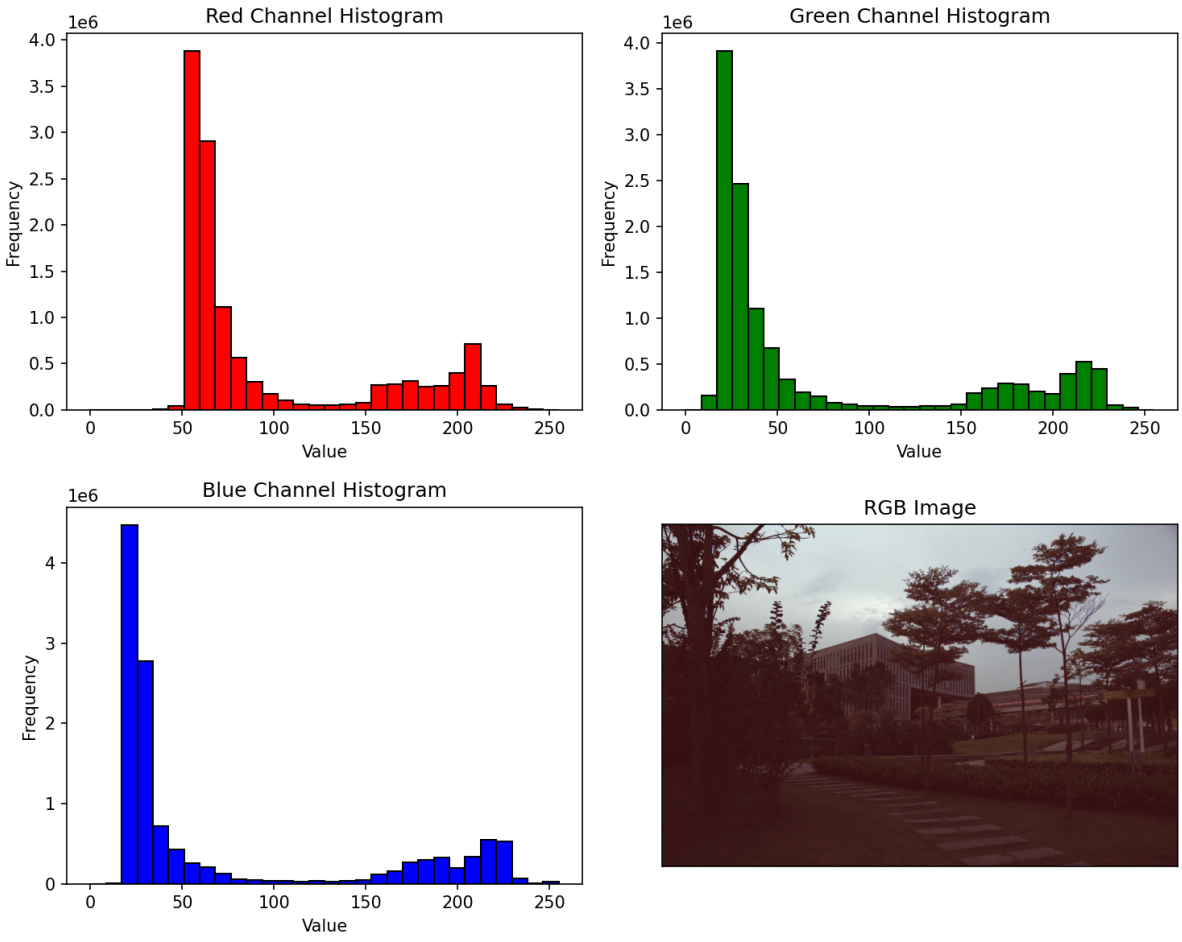






Normalize and output the RBG

After all, assign the `Y` channel as the BCC output, and assign the `Cr`, `Cb` channel as the HSC output.



How to Run

You can run the code without checking detail

```
python hw2/isp_pipeline.py
```

Or you can check image change of each stage in the pipeline with

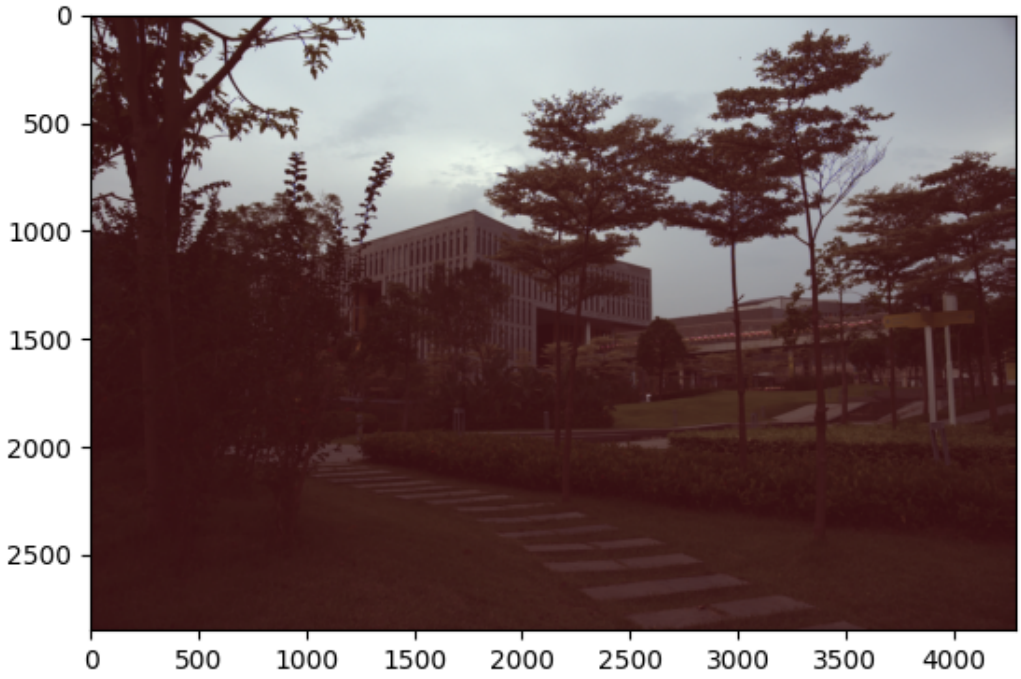
```
python hw2/isp_pipeline.py --imgbose
```

Results

You are free to access the result of each step under the `./data` directory.

Compared to the `rawpy` Output without Gamma Correction

Our ISP Output



References

Github

- ISP Pipeline | camera成像原理: https://juejin.cn/post/7309301549154828323
- FastOpenISP: https://github.com/QiuJueqin/fast-openISP?tab=readme-ov-file
- OpenISP: https://github.com/cruxopen/openISP
- Opencv BCC: https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html
- False Color Suppression in Demosaiced Color Images: https://www.cse.iitb.ac.in/~sharat/icvgip.org/icvgip2004/proceedings/ip1.3_029.pdf
- FalseColor-Python: https://github.com/serrob23/falsecolor