CSE264 Project 1

Joseph Adamson

5/10/2020

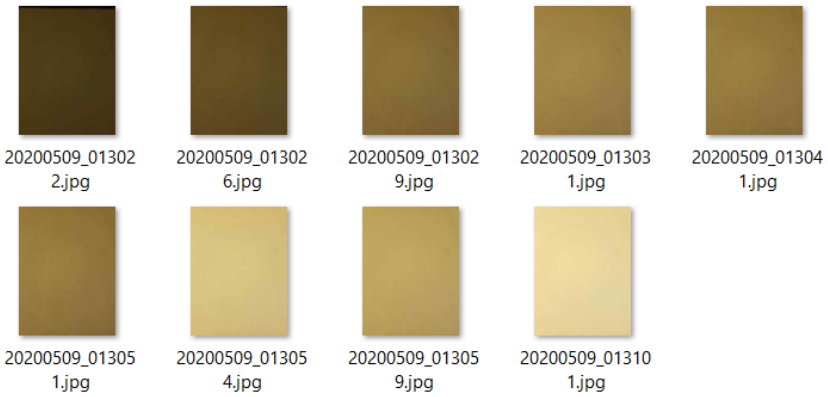
CAMERA USED: Samsung Galaxy S7 camera

Part 1) Radiometric Calibration

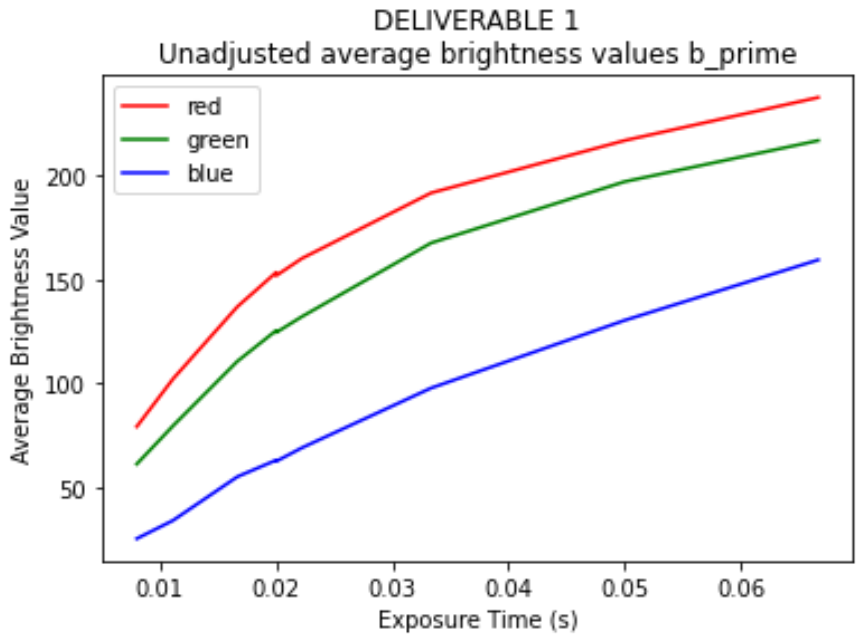
In order to determine the nonlinearities in response of my phone camera, the following procedure was implemented according to the project instructions.

USING MANUAL COLOR TEMPERATURE SETTING: 3000K

1) Take several pictures with increasing exposure time of a flat uniform surface. I used a piece of printer paper in indoor lighting, which is a compact fluorescent bulb (CFL) and watched for the image to move from black to white. The dataset looks as follows:



2) Examine each pixel value using opencv. Saturated pixels (any color channel = 255) trigger a warning and allowed me to remove these photos from the dataset. Additionally, my script gives the final minimum and maximum values for each color channel, so that pictures with a minimum brightness less than 50 can also be discarded alongside those with brightness = 255.
3) Chart the sum of all pixels per image, per color channel. They are plotted against exposure time, which I used a library to extract from the exif data and sort the brightnesses with. These methods produced the following graph. We can see that the camera has the expected nonlinear response, but it is most noticeable in the red channel. This could possibly be due to the sensor properties, my CFL light source, or my choice of 6000K white balance.



And the final stats from the dataset:

```
final stats:
(T_min,T_max) =  0.008 ,  0.06666666666666667
(R_min,R_max) =  79.037075 ,  237.57475
(G_min,G_max) =  61.012675 ,  216.818775
(B_min,B_max) =  25.087025 ,  159.358475
```

I attempted to get the R and G channel minimum values lower, but I couldn't do that without sacrificing SNR in the B channel. This indicates that the camera is much more sensitive to red than it is to blue; this is also indicated with the max values, as the B channel is relatively low even when the red channel is close to saturating at 255.

4) Determine a correction factor via linear regression. Using the given formula:

$$\log(B'(T)) = \log(K \times T1/g) = \log(K) + (1/g) \times \log(T)$$

allows us to form a line equivalent to:

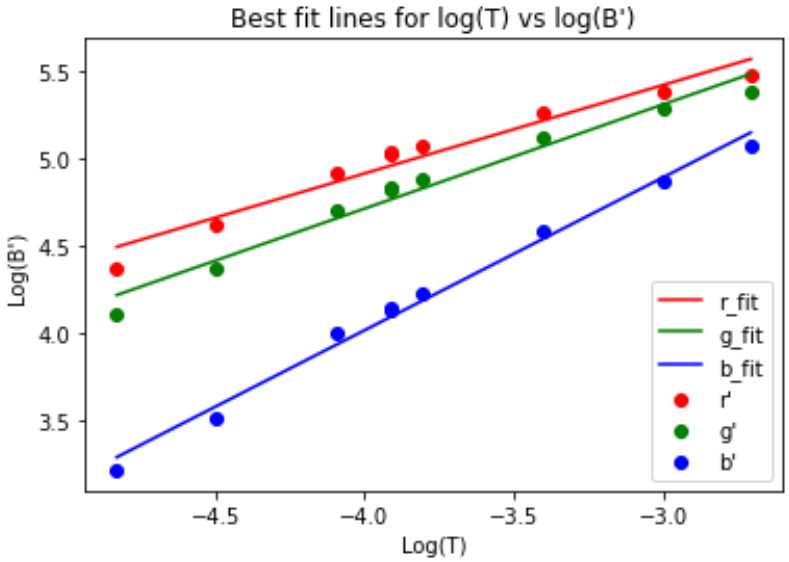
$$y = mx + b \qquad (m = \text{slope}, b = \text{intercept})$$

Since we have already found the arrays of B' and T, I was able to use numpy's linear regression tools to calculate the appropriate logs and then do a linear regression fit with the following lines of code:

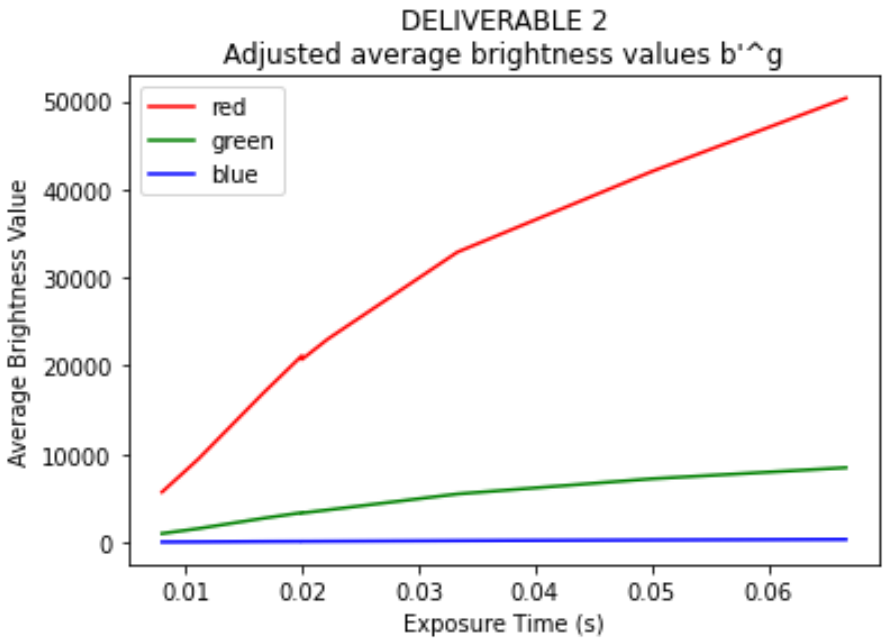
```
[xmin,xmax] = [min(np.log(T)),max(np.log(T))]
x = np.linspace(xmin,xmax)

slope,intercept = np.polyfit(np.log(T), np.log(red_prime), 1)
g_r = 1/slope
k_r = np.exp(intercept)
redline = np.polyval((slope,intercept),x)
```

This was performed for each color channel [r,g,b] to generate a linear fit and ultimately determine the correctional factor g such that $B = (B')^g$ [approximately]

Plotting these linear fits results in the following chart:



5) Apply the correction factor to the gathered data for each color channel by calculating B'^g for each color channel results in the following corrected graph (noting that the pixel brightness is now also exponentiated):



We can see that we have *roughly* linearized it. The red channel still contains a strong bend, which makes sense given the stronger nonlinearity present in the initial graph. I suspect this camera is highly sensitive to red. Using a different fitting function (ie, performing a log fit on the original data rather than a linear fit on the log data) would probably produce better results.

PART 2)

Following the instructions, 30 images were taken under 3 illuminants of a color image mosaic consisting of magenta, cyan, white, and yellow. Since a printer was unavailable, the mosaic was constructed using post-it notes on plain white paper.

There are 10 per illuminant (at the 4 azimuth angles and top-down) 2 each with auto white balancing off (6000K) and auto white balance ON

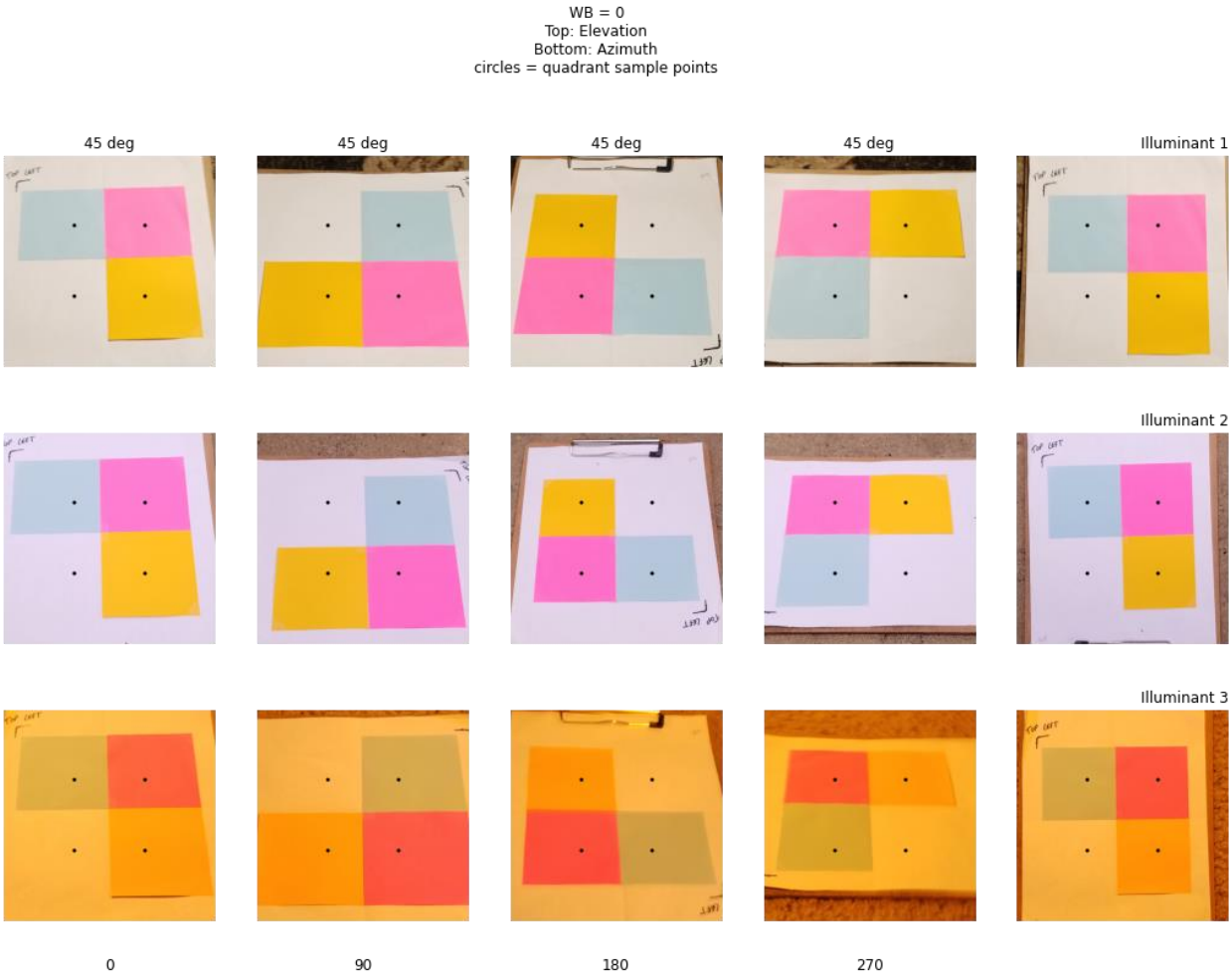
I used the following illuminants, with ISO set to minimum (50) and exposure time selected by setting exposure to AUTO to analyze the scene, and then fixing it at the value determined to have good exposure.

Illuminants:

1) Indoors, white LED lamp
    a. ISO Value: 50
    b. Exposure time: 1/6 s
2) Outdoors, daytime sun (overcast)
    a. ISO : 50
    b. Exposure time: 1/250 s
3) Indoors, CFL lamp bulb
    a. ISO : 50
    b. Exposure: 1/6 s

Every image was cropped and displayed in a grid. An image editing program was used to ensure each one was centered prior to processing; this made it trivial to define a set of 4 points giving the centers of each quadrant. The black dots in the image represent the points of these quadrants, and will be used in point three to average the values for each of these patches.

The resulting images are given in the tables below; the raw data can be found in the project google drive.



Above: Manual white balance (6000K)

It’s clear that the third illuminant displays quite a lot of red. This could be a mixture of things, but most likely it’s due to the high amount of red in the walls and décor in that room, as well as the high red sensitivity discussed in part 1. It may also be that my CFL has a high amount of red in its spectrum.

Above: Automatic White balancing

It can be seen that the automatic white balance had a noticeable effect on illuminant 3, the CFL bulb. As shown below, that illuminant still became an outlier, but the white balance algorithm was still able to correct for a lot of that red effect.

PART 3)

The colors of each patch in each image above were collected by hard-coding the center value of each quadrant as discussed above. 20x20 patches were sampled and averaged from each image, centered around the black dots in the tables above.

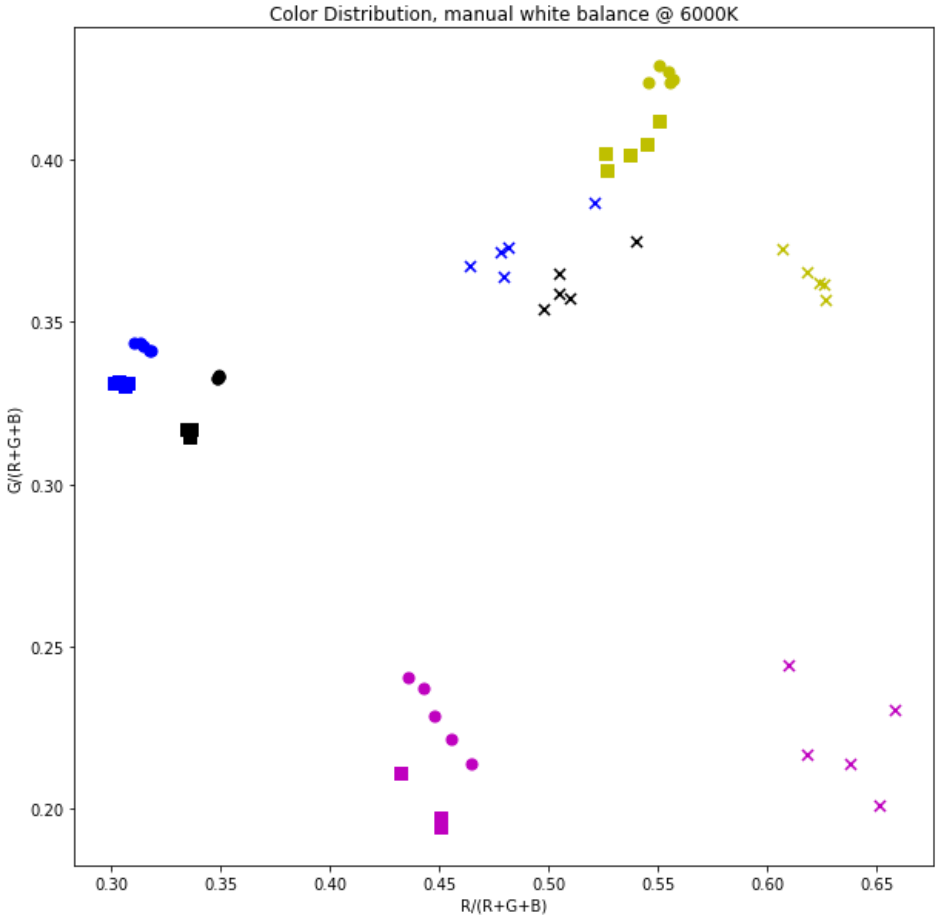
I decided to write a custom colors class which would hold a color array in addition to all the related parameters (illuminant, viewpoint, etc) in order to more easily iterate through all the colors and generate the required graphs. When instantiated, the class generates the normalized R' and G' values by the computed brightness (R+G+B). From there it was straightforward to create a color class for each averaged patch from each image, and generate the following graphs; as titled, one is for the manual white balance (6000K) and the second is using my camera's auto brightness. In all graphs, the legend is:

Circle: Illuminant 1

Square: Illuminant 2

X :Illuminant 3

And the marker colors correspond to the patch they were taken from (where the black markers represent the white patch)

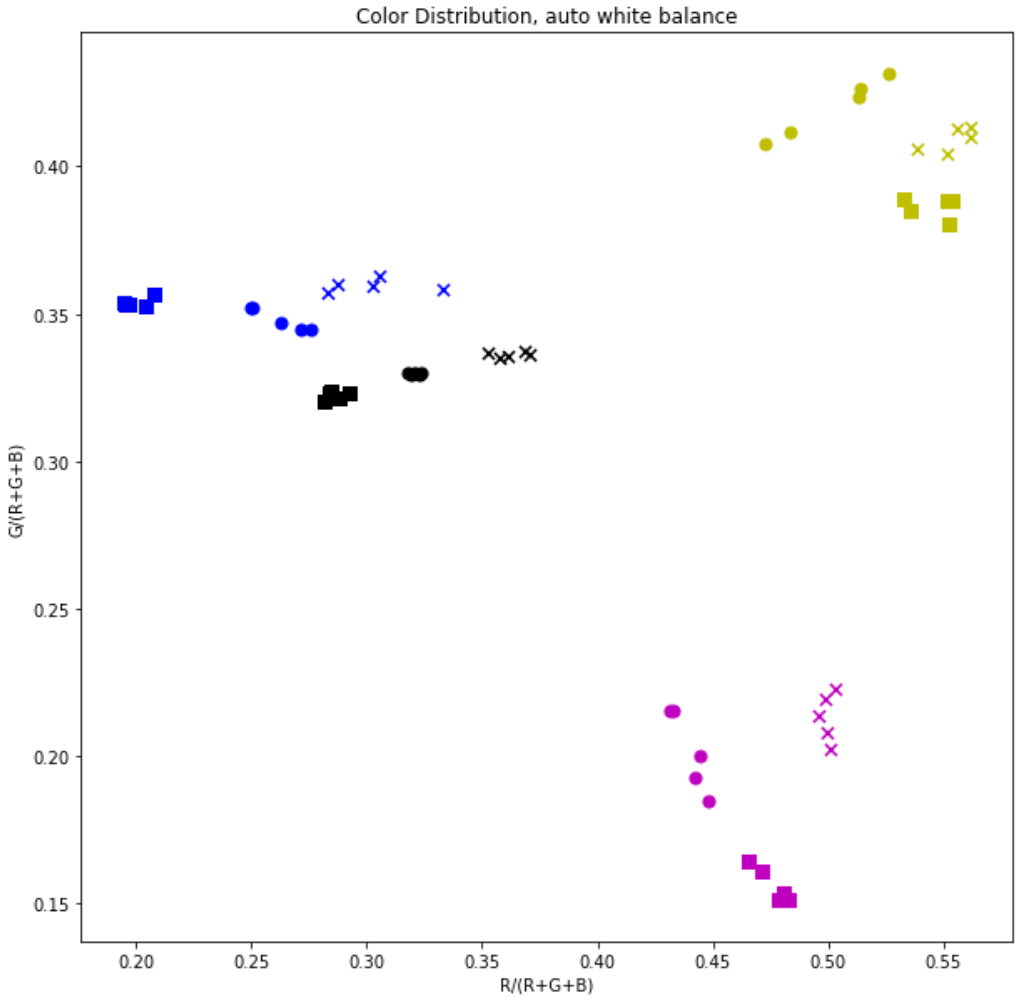


From this graph we can get a rough idea of where the patches lie in this color space (R' vs G') ie, magenta groups in the lower left, yellow in the upper right, blue in the upper left, and white somewhere in between.

It can be seen quite clearly that the manual color balance chart has values which are gathered close together for the same illuminant type, but values from the same color patches vary widely between illuminant types. Furthermore, illuminants 1 and 2 are relatively close, but illuminant 3 (x markers) tends to be in wildly different locations. Intuitively, that can be seen in the dataset by noting that illuminant three has a much redder hue than the others.

The grouping together of values from common illuminants tells me that the brightnesses **do not depend strongly on viewing angle** which makes good sense given that paper has lambertian characteristics, as discussed in lecture.

Now, for the data with auto white balance on:



We can see from this graph that the camera was able to get the colors of each patch much closer together; instead of widely spread responses from different illuminants, there are now clear and

separate groupings of images. This can be seen in the dataset, that the third row of the white balanced images appears much closer to the others. The notable exception here is that the x markers (illuminant 3) are still rather outlying (except on the yellow patch for some reason!) meaning that the camera white balance algorithm still has some difficulty adjusting for all that red.

It's also interesting to note that illuminant three appears to introduce a common offset within my camera's white balance algorithm; the values under this illuminant tend to be offset up and to the right of the others.

PART4) Color Constancy

All images used for the following calculations were taken from the dataset above. Only those images with manual white balance were used.

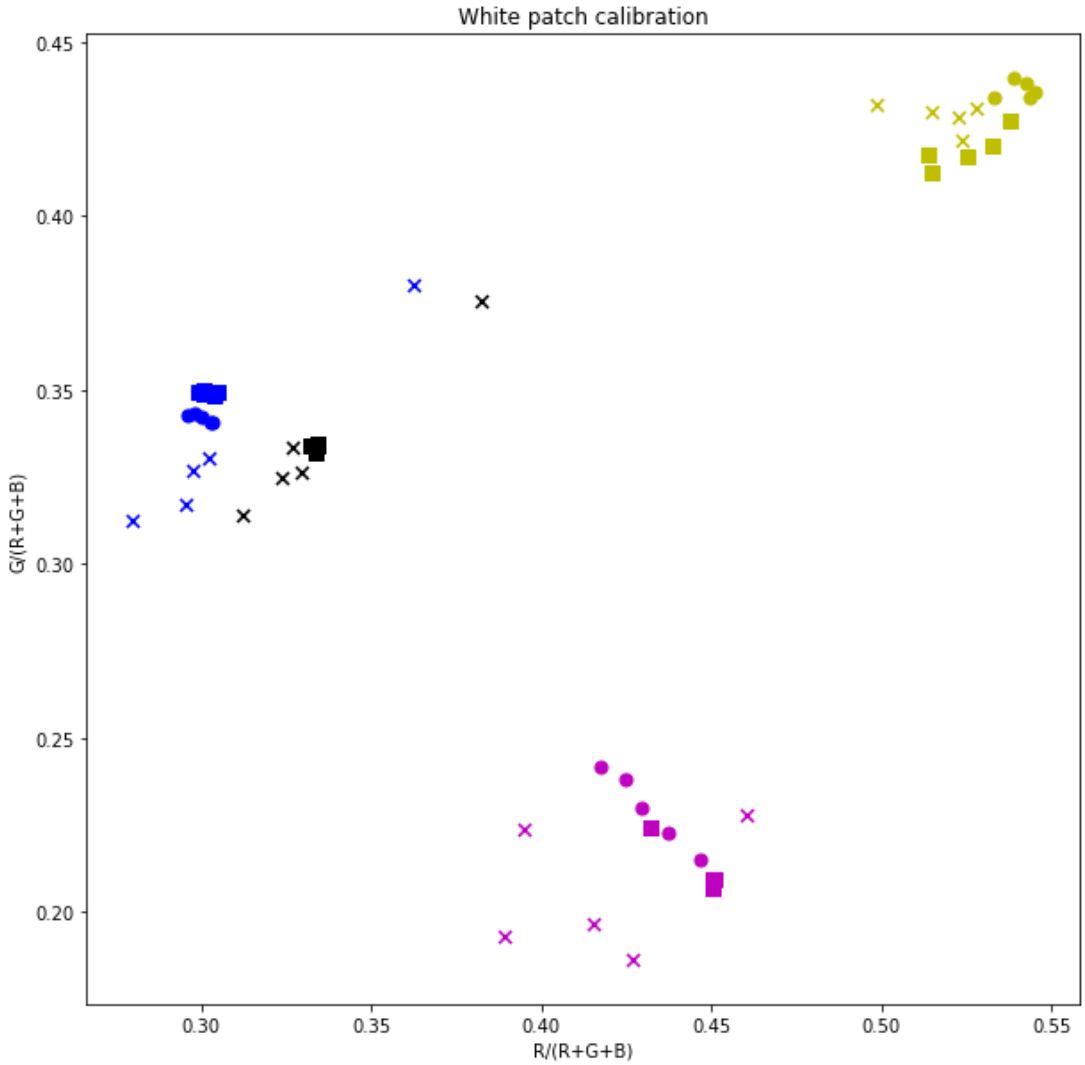
In order to try our own white balance, three methods were applied to the colors of the dataset. The project assignment describes the algorithm to generate each one, so I applied those methods and then balanced the color data. To assist this, I added an applyTransform method to my color class in order to quickly get and plot the results without much extra code; we just need to call the numpy native method 'dot' after constructing D (ie, transformedColor = D.dot(color))

1) White Patch Method

The matrix D was constructed as per the project instructions by iterating through the white squares in the dataset, and averaging out all of the pixel values. Then, the matrix D was constructed by using each diagonal element equal to $255/a_k$ where a_k is the average color calculated from the white patches for the kth channel. The following D matrices were calculated using the shown average values of all the white patches: (one for each illuminant)

```
Ill 1 avg white rgb: [222.1145 212.1145 202.6425]
Ill 2 avg white rgb: [225.3855 212.1855 233.7855]
Ill 3 avg white rgb: [253.837 179.793  63.273]
Illum 1 White Patch Transform:
 [[1.14805652 0.         0.        ]
 [0.         1.2021809  0.        ]
 [0.         0.         1.25837374]]
Illum 2 White Patch Transform:
 [[1.13139488 0.         0.        ]
 [0.         1.20177863 0.        ]
 [0.         0.         1.09074344]]
Illum 3 White Patch Transform:
 [[1.00458168 0.         0.        ]
 [0.         1.41829771 0.        ]
 [0.         0.         4.03015504]]
```

After applying this transform to every color from the dataset, we get the graph:

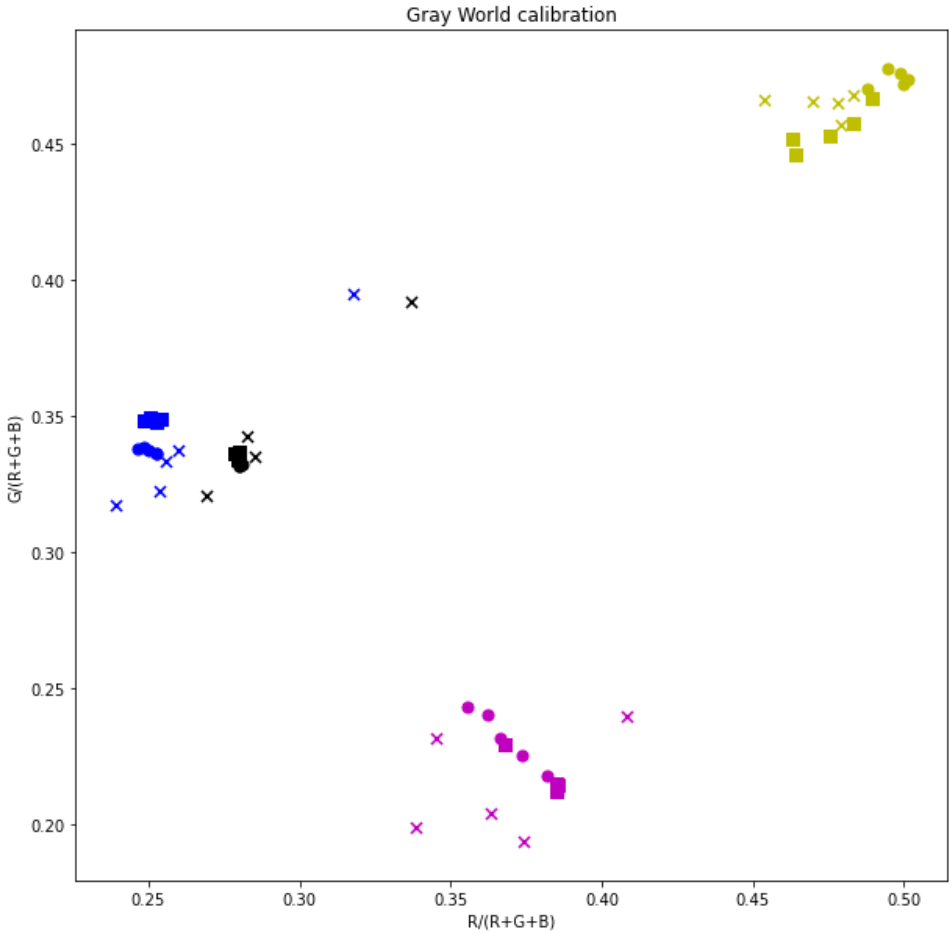


This calibration brings the white values much closer together (as expected) and does pretty well with the yellow, but performs more poorly than auto white balance with regards to the spread of the data; ie, the camera was able to group them more tightly. Still, we can see that this algorithm does a good job of grouping the colors, with the exception of some outliers.

2) Grey World Method

This method is identical to the above method, except that the average values are taken from all patches, not only the white ones. The average values and resulting matrices are given below:

```
Ill 1 avg rgb: [225.280375 181.76825  148.8705  ]
Ill 2 avg rgb: [227.94775  178.81975  171.877625]
Ill 3 avg rgb: [244.632375 146.072875  47.94575 ]
Ill 1 Grey World Transform:
 [[1.13192283 0.         0.        ]
 [0.         1.40288527 0.        ]
 [0.         0.         1.71289812]]

Ill 2 Grey World Transform:
 [[1.11867742 0.         0.        ]
 [0.         1.42601698 0.        ]
 [0.         0.         1.4836137 ]]

Ill 3 Grey World Transform:
 [[1.04238043 0.         0.        ]
 [0.         1.74570399 0.        ]
 [0.         0.         5.31851103]]
```



The results of this calibration method are almost indistinguishable from the white patch method

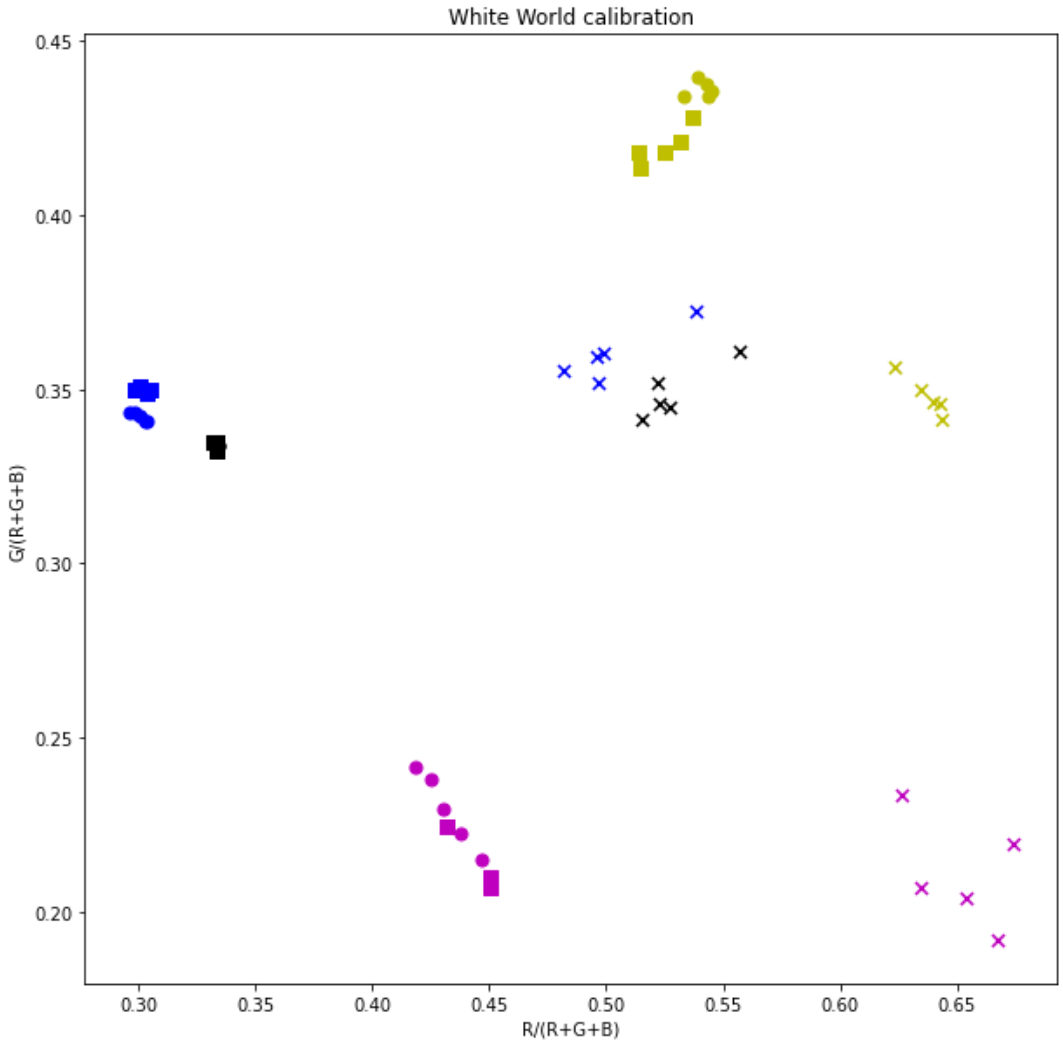
3) White World Method

This method instead assumes the brightest pixel in the images must be white, and scales that pixel to be [255,255,255] while scaling down the rest of the pixels accordingly. Therefore, we iterate through and search for the pixel with the highest brightness, and . The matrices generated for this transform are as follows:

```
Ill 1 White World Transform:
 [[1.13249026 0.         0.        ]
 [0.         1.18512322 0.        ]
 [0.         0.         1.23685838]]

Ill 2 White World Transform:
 [[1.10873181 0.         0.        ]
 [0.         1.18059655 0.        ]
 [0.         0.         1.06697909]]

Ill 3 White World Transform:
 [[1.33774001 0.         0.        ]
 [0.         1.24621249 0.        ]
 [0.         0.         1.24621249]]
```

Note that the values are greater than 1; this indicates that the brightest pixel under each illuminant must have summed up to less than 255*3; in other words, the images were not perfectly exposed to the right. In fact, every matrix element so far has been greater than one, which makes perfect sense since the scale factors have been designed that way, and can be, at a minimum, equal to one for perfectly exposed pixels.

The color grouping plot generated by this transform is given below:



This method is terrible for this dataset; it's most likely due to the fact that each illuminant has a very different brightest pixel. This could probably be solved with better exposure