Name: Batuhan

Surname: Yalçın

ID: 64274

COMP527/ELEC519 – Programming Assignment 1: Camera Pipeline

Problem 1)

RAW image conversion:

For the converting RAW image the tiff file. First I upload the dcraw with the sudo apt install dcraw, and latest version uploaded.

Then using the given command, I found Scaling was:

Scaling with darkness 150, saturation 4095 black and white respectively

And multipliers for the raw image was:

Multipliers 2.39118 1.000000 1.223981 1.000000 rgb scales respectively

Scaling with darkness <black>, saturation <white>, and

multipliers <r_scale> <g_scale> <b_scale> <g_scale>

As given in below:

```
byalcin17@DESKTOP-5C6S3OA:/mnt/c/Users/byalcin17/Desktop/comp427/Assignment1$ dcraw -4 -d -v -T campus.nef
Loading Nikon D3400 image from campus.nef ...
Scaling with darkness 150, saturation 4095, and
multipliers 2.393118 1.000000 1.223981 1.000000
Building histograms...
Writing data to campus.tiff ...
byalcin17@DESKTOP-5C6S3OA:/mnt/c/Users/byalcin17/Desktop/comp427/Assignment1$
```

Then tiff file removed and re-created with given code:

dcraw -4 -D -T campus.nef

Python initials:

It's read by io.imread then I print the width height and bits per pixel:

```
Bits per pixel: 16
Image width dimension: 6016
Image height dimension: 4016
Data type: uint16 (“2D array)
```

Note that to calculate bits byte converted to bit with multiplying 8.

Then converted to double precision array with the following:

img_double = imgr.astype(np.double)

Reading Image

```
imgr = io.imread('campus.tiff')

##properties of the image
print("Bits per pixel:", imgr.dtype.itemsize*8)
print("Image width dimension:", imgr.shape[1])
print("Image height dimension:",  imgr.shape[0])
print("Data type:" , imgr.dtype)

##Then converting double
img_double = imgr.astype(np.double)
print(img_double)
```

```
Bits per pixel: 16
Image width dimension: 6016
Image height dimension: 4016
Data type: uint16
[[200. 306. 217. ... 199. 177. 219.]
 [379. 203. 347. ... 185. 215. 190.]
 [225. 388. 216. ... 211. 177. 235.]
 ...
 [901. 584. 929. ... 302. 412. 304.]
 [520. 909. 494. ... 396. 268. 408.]
 [893. 578. 905. ... 307. 390. 311.]]
```

Linearization:

Linearization happens in two parts first shifting then scaling.

To first shit the img image shifted by the amount of black so the black will zero.

Then to scale between 1 and 0 the scale between white and black which white mines black scaled to the image pixel with division.

Then all the values linearized between 0 and 1. With additional negative and bigger than 1 value clipped.

Linearization

```
In [38]: black = 150
         white = 4095

         # linear transformation
         #Shift
         linear_img_shift = (img_double - black)
         #Scale
         linear_img = linear_img_shift  / (white - black)

         print(linear_img)
         # Clip negative values to 0 and values greater than 1 to 1
         linear_img = np.clip(linear_img, 0, 1)
         print(linear_img)
```

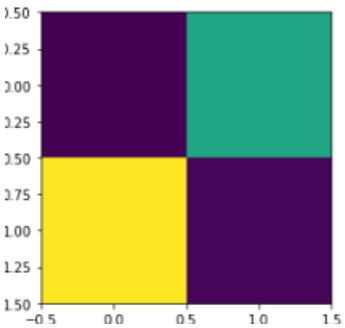
```
[[0.01267427 0.03954373 0.01698352 ... 0.01242079 0.00684411 0.01749049]
 [0.05804816 0.01343473 0.04993663 ... 0.00887199 0.01647655 0.01013942]
 [0.01901141 0.06032953 0.01673004 ... 0.01546261 0.00684411 0.02154626]
 ...
 [0.19036755 0.11001267 0.19746515 ... 0.03852978 0.06641318 0.03903676]
 [0.09378961 0.19239544 0.08719899 ... 0.06235741 0.02991128 0.06539924]
 [0.18833967 0.10849176 0.1913815  ... 0.03979721 0.0608365  0.04081115]]
[[0.01267427 0.03954373 0.01698352 ... 0.01242079 0.00684411 0.01749049]
 [0.05804816 0.01343473 0.04993663 ... 0.00887199 0.01647655 0.01013942]
 [0.01901141 0.06032953 0.01673004 ... 0.01546261 0.00684411 0.02154626]
 ...
 [0.19036755 0.11001267 0.19746515 ... 0.03852978 0.06641318 0.03903676]
 [0.09378961 0.19239544 0.08719899 ... 0.06235741 0.02991128 0.06539924]
 [0.18833967 0.10849176 0.1913815  ... 0.03979721 0.0608365  0.04081115]]
```

Identifying the Correct Bayer pattern:

Two find the correct bayer pattern I try to implement 4 different techniques.

First step in my techniques:

Checking the correct top left square:




```
# Extract the top-left 2x2 square of the image

square = linear_img[:2, :2]

print(square)

plt.imshow(square)
```

It seems purple, green, yellow and again purple but how this can be possible, answer mixture of the colors:

- Purple can be mixture of the blue and red.
- Green can be mixture of the blue and yellow.
- Yellow can be mixture of the red and green.

The reflection comes from the object can affect these colors so it can be multiple bayer pattern but since 1. and 4. Points are purple, it should consist blue and red channel and it only in RGGB pattern it can more suitable let's do further analysis. It can't proof anything right now.

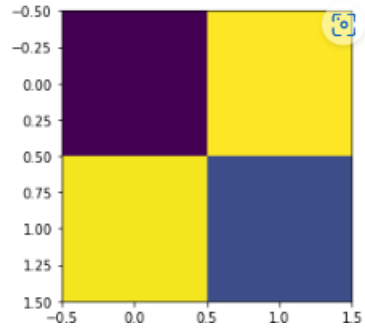
Second step in my techniques:

Since the blue and red channels are unique investigating the red area can give better response let's look this area:

```
fig = px.imshow(img_double)
fig.show()
```

At the given original picture at the axis X between 896-898 and at the axis Y between 2070-2072 should be under red reflection let's check the red reflection.

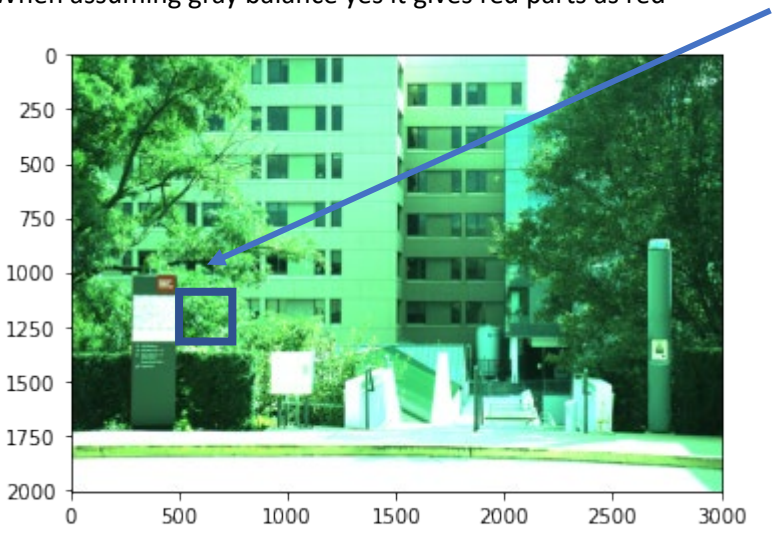
```
<matplotlib.image.AxesImage at 0x11762d87dc0>
```




```
# Extract the top-left 2x2 square of the image

square = linear_img[:2, :2]

print(square)

plt.imshow(square)
```

Now when we extract the red reflection from combination it shows red, green, green and blue let's check under the grayscale because If the original image is a grayscale image, then all color channels (including red) will have the same values. Still not sufficient for me so I use third and fourth technique as proof.

When assuming gray balance yes it gives red parts as red



Third step in techniques:

Using mean values of all channels and comparing the expected values with 4 bayer pattern:

After my implementations I also want to take advise from the chatgpt I wrote him for to advise me an alternative way to find compare bayer pattern. It advises me that you can take mean values of the all channels and look which original channel most suitable for that. Then I implement an algorithm which check the errors between expected value of channels means under 4 bayer patter and my channel means. In which bayer pattern gives smallest error is mor suitable patterm.

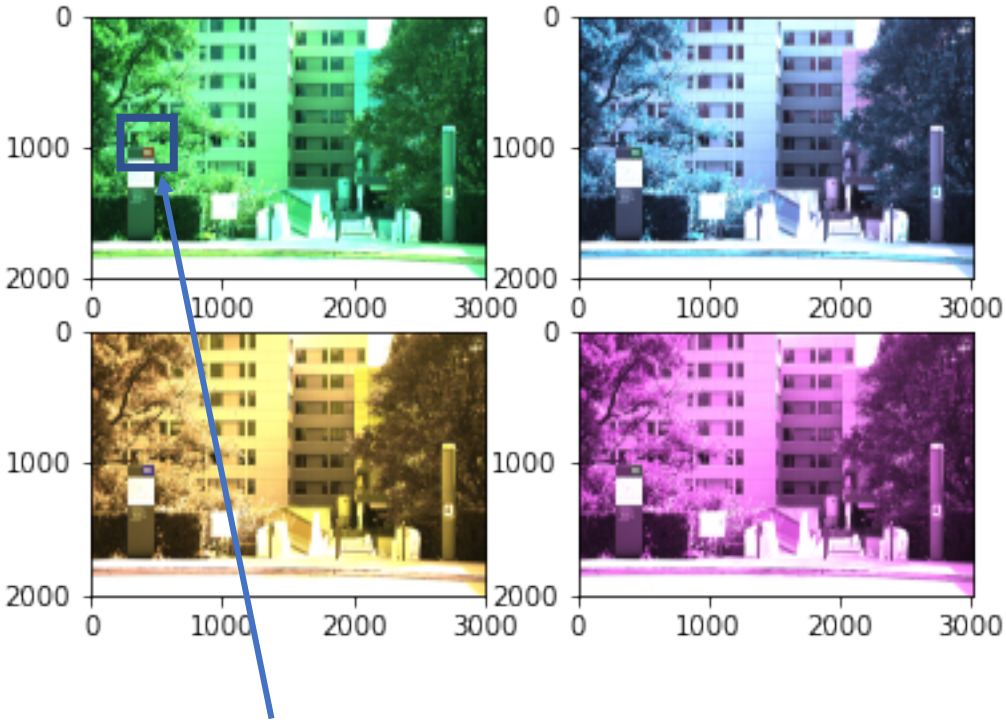
In this algorithm I also found that the best algorithm is RGGB.

```
#Finding best bayer pattern
bayers = ['RGGB', 'GBGR', 'BGRG', 'GRBG']
bayer = bayers[np.argmin(errors)]
print('Bayyern Pattern is following due to most less error:')
print(bayer)
```

```
[1.9250950570342205, 1.9250950570342207, 1.9250950570342207, 2.4250950570342207]
Bayyern Pattern is following due to most less error:
RGGB
```

Fourth step in techniques Final Proof:

After finding the best bayer pattern I also check the all different types of the bayer pattern and I found that the most suitable one is RGGB because only in RGGB I get correct inputs



Congratulations !!!! RGGB is the correct bayer pattern.

White Balancing:

After identifying the correct Bayer pattern, which was the RGGB, RGB picture created using numpy dstack function (Note that since we have for channel for 2 green channel the green channels are averaged to be using in the RGB). Then using lecture notes 3 Noise color pdf page 54. White and Gray algorithms applied. To apply the algorithm max and average value of green, red and blue channels are calculated then the matrix created as described in the lecture notes.

Automatic white balancing

Grey world assumption:

- Compute per-channel average.
- Normalize each channel by its average.
- Normalize by green channel average.

white-balanced RGB $\rightarrow \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} G_{avg}/R_{avg} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & G_{avg}/B_{avg} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow$ sensor RGB

White world assumption:

- Compute per-channel maximum.
- Normalize each channel by its maximum.
- Normalize by green channel maximum.

white-balanced RGB $\rightarrow \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} G_{max}/R_{max} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & G_{max}/B_{max} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow$ sensor RGB

54

For the prescale algorithm previous scales found from dcraw code used these were:

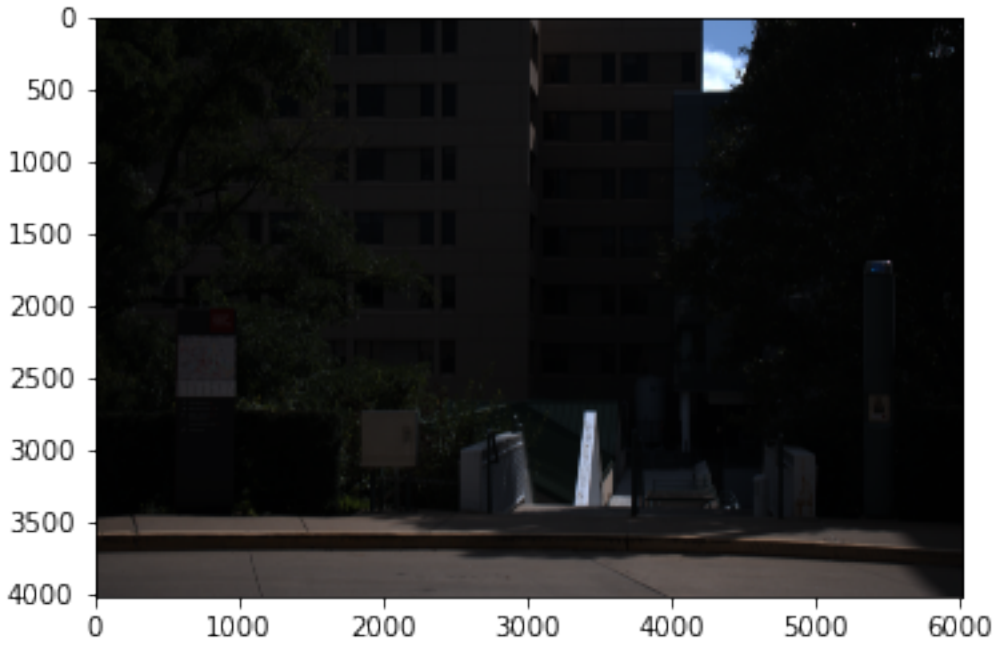
Red scale = 2.393118
Green scale = 1.0
Blue scale = 1.223981

Then the used scales multiplied by the RGB channels respectively to apply as a third white balance method.

Best one chosen as the Gray Algorithm because it shows more color accuracy then the others.

Demosaicing:

For this step scipy build interpolations functions (interp2d) used but not that after RGB created for white balance algorithms the picture size decreased to half to rearrange original picture size arrange step size described as 0.5. After the demosaicing the result is:



Color space correction:

For these steps all mathematical functions were well defined I just should find the dcraw defined matrix values for Nikon D3400 DSLR and divide it 10000. It found under the adobe_coeff function. The values are:

```
    { 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 } },
{ "Nikon D3400", 0, 0,
    { 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 } },
{ "Nikon D300", 0, 0,
```

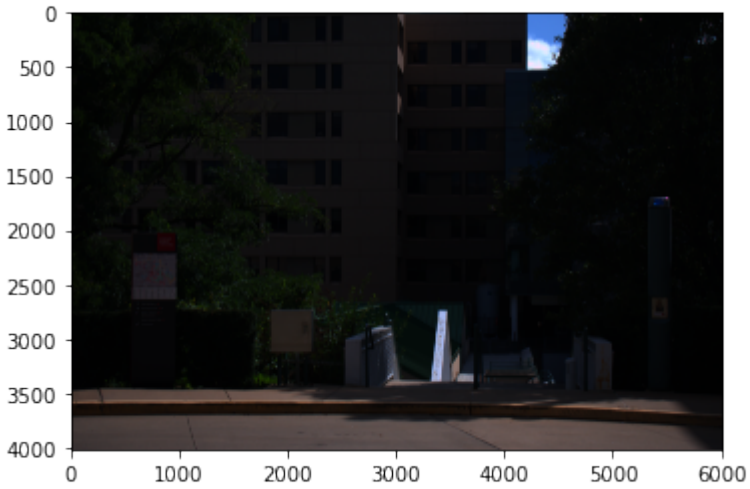
{ 6988,-1384,-714,-5631,13410,2447,-1485,2204,7318 }, adobe_cuff_xyz = np.array([[6988,-1384,-714],

[-5631,13410,2447],

[-1485,2204,7318]])/10000

Then after creating the RGB_to_Matrix, I matrix multiply the previous demosaicing rgb values with inverse of this matrix since the our created matrix is in the transpose form the values also transposed.

The result is:



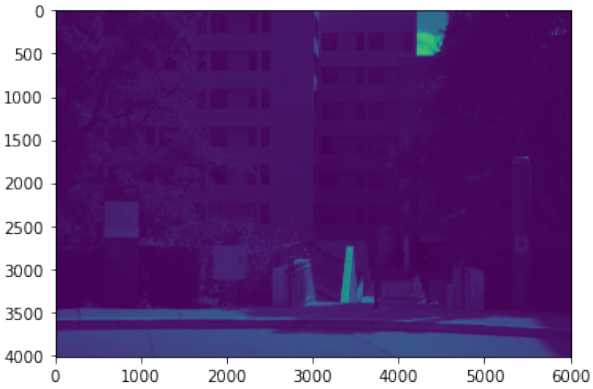
Which is very dark image know the brightness should adjust.

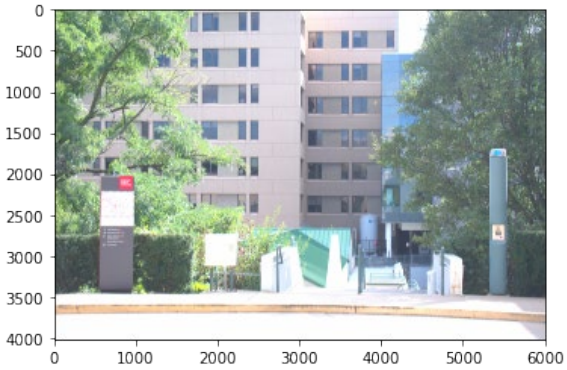
Brightness adjustment and gamma encoding:

This part was very fun like in game we first try to adjust image under grayscale then after selecting the best brightness we apply gamma encoding to tone reproduction. To do that as described in the pdf threshold values masked in condition with 1 and 0 then after multiplying with these conditions only the true values added to the result image and gamma encoding also finalized.

To find correct brightness scale first I try 10 it was too bright:

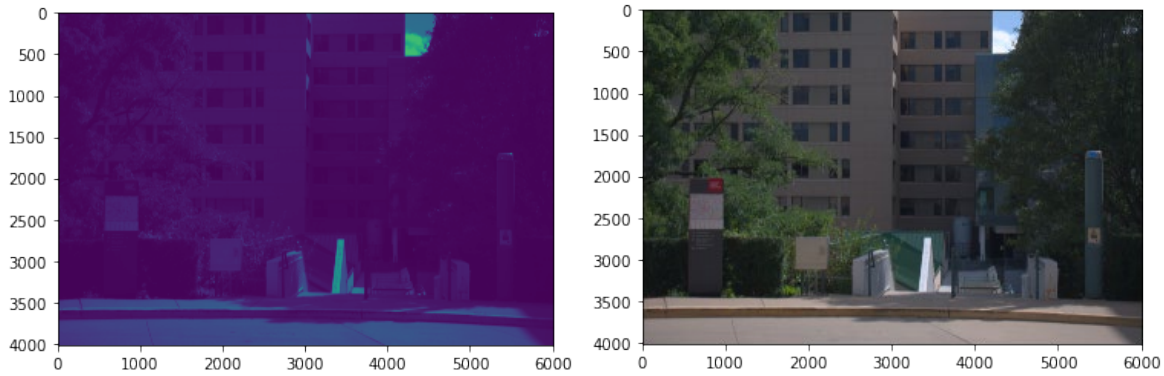
```
Mean value: 0.7226418311630898
```





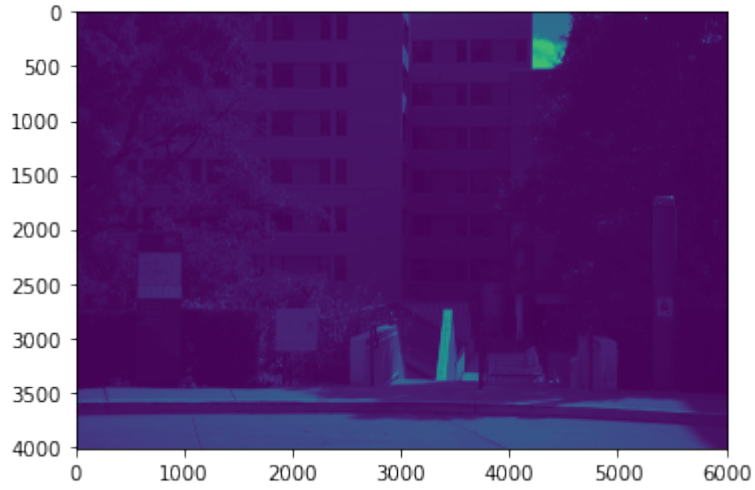
even for gray scale it was not like gray and white screen were almost not readable so then I try 1:

```
Mean value: 0.07226418311630896
```



After the couple of try (5-3,33-3-2.88-2.75-2.5-2.22-1.75) while considering the readability of the texture in light and dark areas the scale 2.75 was okey for me.

```
The mean value is: 0.19872650356984997
```



Compression:

The comparison of two files are I below:

1. First since the jpeg has lossy comparison the file size is smaller than the PNG file. My PNG file has 33.8 MB however JPEG files varies on the size between even 500 KB to 7 MB depend on the compression rate.
2. I believe there is much difference on also in the color distribution. The PNG has better color distribution then the JPEG.
3. Sharpness and detail of the edge on the textures are better in PNG such as Carnegie Mellon Universities text, edge of leaf.

To determine the comparison rate, I compared all the quality percentages between 1 and 100. The highest percentage corresponds to the best quality, but we may not always need that level of quality.

It was difficult to determine the smallest compressed ratio by checking the differences. However, the image quality at settings 50-75 was sufficient, and I couldn't see any changes after 75. At quality level 60, the texture was readable, leaves were distinguishable, and although none of the pictures looked exactly the same as the given JPEG, at quality 60 the pictures were almost indistinguishable from the PNG file.

So, the smallest quality setting that I found was 60. The size comparison ratio with respect to the PNG file was around 20.

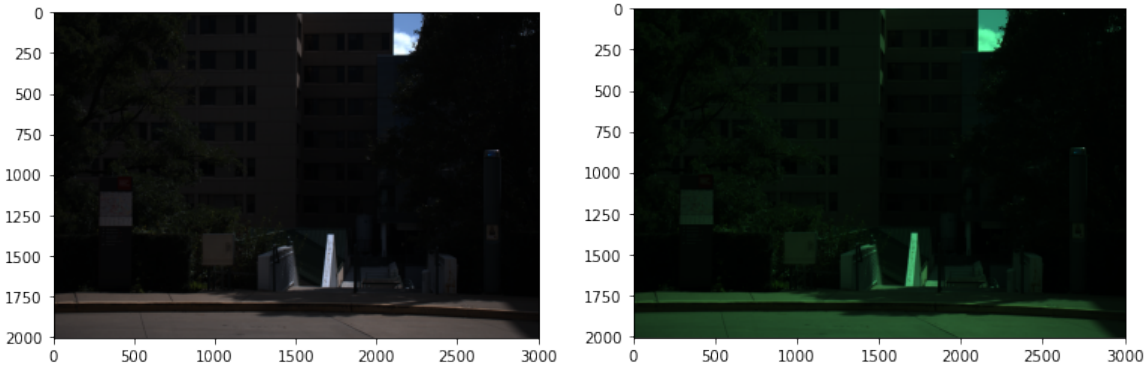
Quality in sk.io.imsave = 60

The compression ratio respected to PNG file is = 33.8 / 1.63 = 20.74

The compression ratio respected to lossy compressed JPEG full quality= 6.87 / 1.63 = 4.21

Different white balance applied final images since demanded:

Changing white balance



Gray and white balance respectively just after balancing however the final result is almost same my eyes couldn't detect too many differences but especially in red and green region color density should change.

Changing brightnes

It significantly change the results image.

Problem 2)

Setup

In this project, I created a pinhole camera using Pringles cans, aluminum foil, and baking paper. Firstly, I cleaned the Pringles can and cut it into sections 5cm apart from the base, in order to obtain a focal length of 5cm. Using a fork tip, I then created a small aperture in the base. As the Pringles cover was too transparent, I used baking paper to create a projection screen. Next, I taped the rest of the Pringles pack together and covered all the open sides with aluminum foil, to prevent any light from entering the aperture hole. This was my setup:

Effect of the Light

Then to understand effect of the source light I took these pictures:

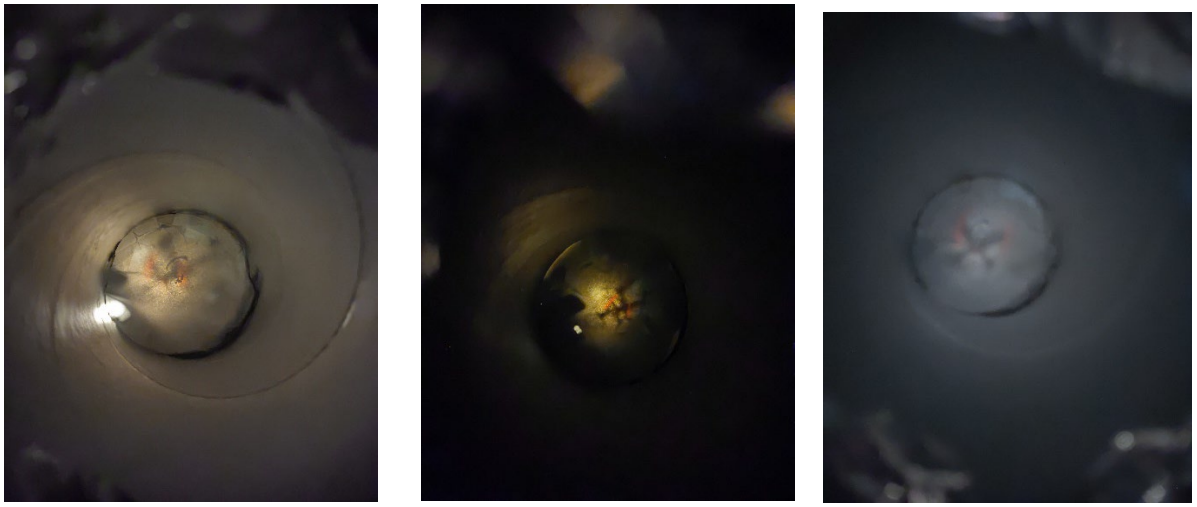
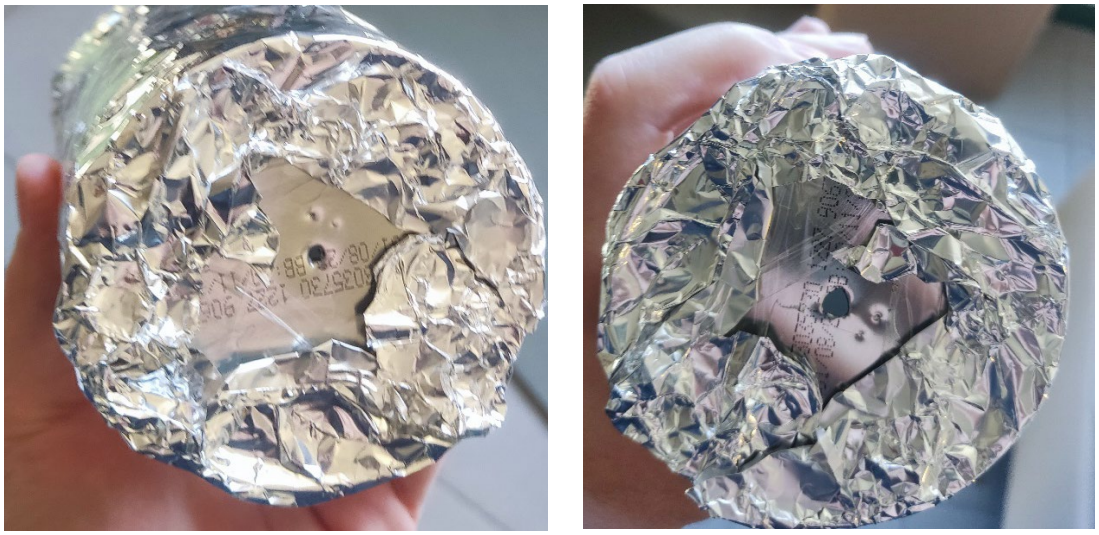


Figure 1 Effect of the Source Light, under too bright, normal, too dim

When the source of light is too bright as shown in first picture the light cause overexposure and a loss of detail in the image. When is optimum I got best picture. When it's too dim, I hardly find the best picture and it was too blurry or and it was having less sharp image.

Effect of the Aperture Size

Then to understand the effect of the aperture size I enlarge to hole diameter:



Note that the other two is not a hole they are a small cavity mistakenly happen during the experiment not a open hole.


The first one is the larger aperture size the second one is the smaller aperture size as you see the smaller one has the more sharpness and detail. To get a good picture I wait long time in smaller one but wait less time in larger one I search this and found this as a exposure time.

Effect of the curtain 😊


This is an only additional not required fun experiment I also try to be dimmed light with the curtain. As expected, it gives blurrier picture.

Overall Observations:

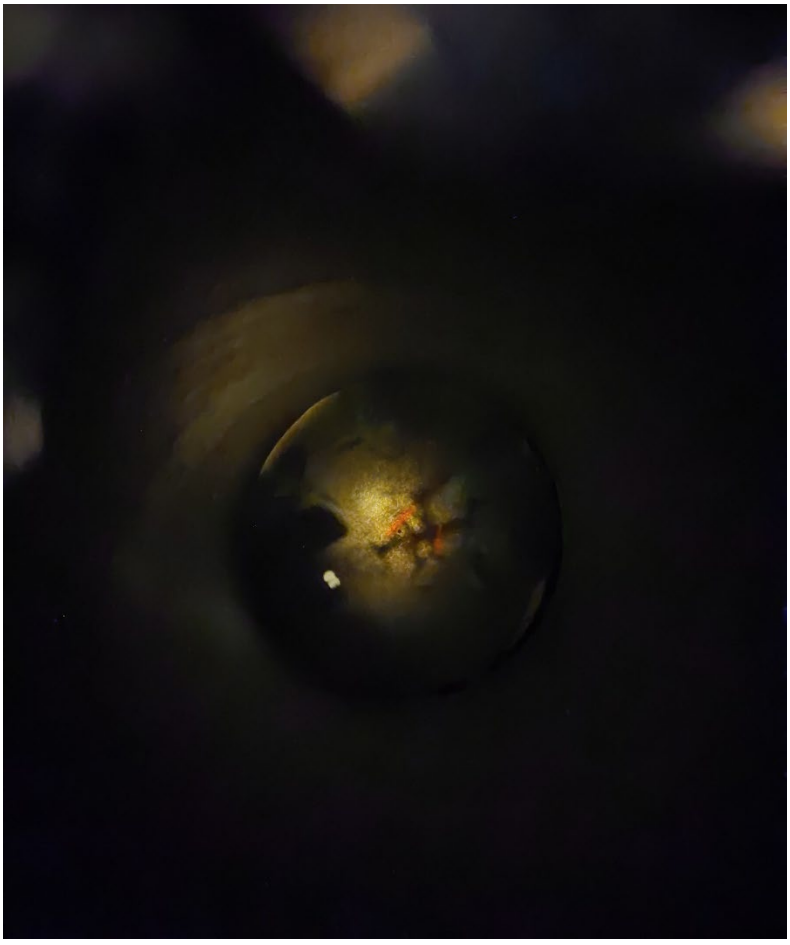
Overall: All the images were inverse and not too clear due to projection surface properties, light, aperture size, focal length etc. However, during the experiment following detail were easily observed:

- Light: The quality and quantity of light was a significant impact on the image. The bright light can cause overexposure and loss of detail, while dim light can result in blurry or underexposed images.

- Aperture size: Changing the size of the aperture was affected the sharpness and depth of field in the image. Smaller apertures created sharper images, but also reduce the amount of light that reaches the camera. Larger apertures allowed lighter but resulted in a softer blurry image.
- Focal length: I didn't do this because the pringles are too expensive I should use another one while the protecting all the other changes are same. However, I can explain possible effect. The distance between the pinhole and the projection surface (focal length) can affect the size and clarity of the final image. Longer focal lengths can produce larger, more detailed images, but also require a longer exposure time (necessary time to collect enough amount of light to get final image). Shorter focal lengths result in smaller, less detailed images, but require less exposure time (necessary time to collect enough amount of light to get final image).

- Image orientation: The orientation of the camera and projection surface also affected the composition and perspective of the final image. Changing the angle or position of the camera can create different visual effects, such as distortion or perspective shifts. While I was trying to take picture of the closest object such as my drone. This was very affected on the trying to take perfect picture on the prefect angle.

My chosen picture for competition:

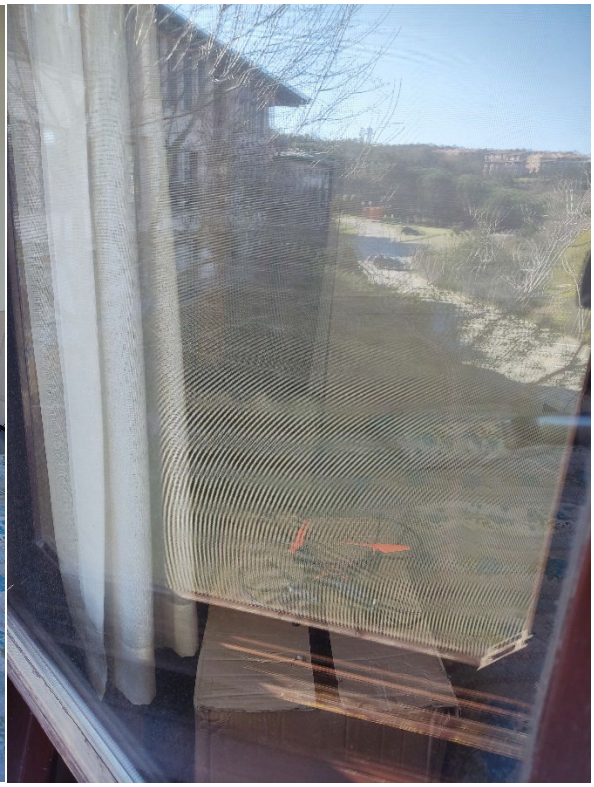


My drone under desk lamp.

Problem 3)

I may misunderstood the some part of the pdf but as I understood I try use window as accidental pinhole.

The taken picture for the problems are was this:

To get the image difference and the get the picture taken by the window I use following approaches:

First I get imageA and ImageB:

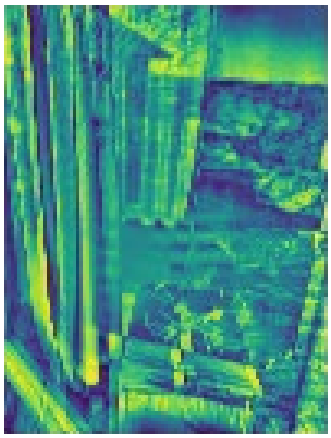
Then to apply more flexible changes on the image I crate gray scales copies.

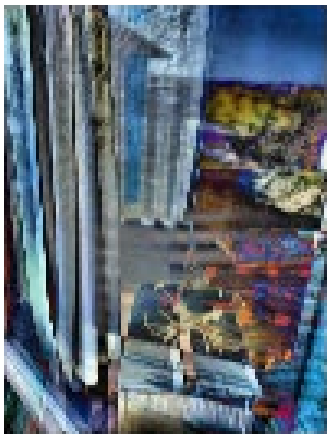
Then I take both original and gray scale differences on the both image.

Then to get more suitable image I applied auto auto contrast and tone adjustment using the skimage adjust_gamma and equalize_adapthist libraries.

Then I compare gray scale image, graytorgb image and without grayscaling rgb images.

The result is following:





I somehow get the image on the occlude reflection but the images should need further adjustment which I do not know yet.