# Computational Photography Homework 1

# – Camera ISP & JPEG Development

20170243 SimJaeYoon

## 1. Overview

This assignment is to make RAW files into JPEG files using image processing technology. I have to make a RAW file into a JPEG file through my own image processing pipeline. To do this, the functions to be implemented are white balance, CFA interpolation, and gamma correction. These functions are essential elements for making RAW images into JPEG images.

## 2. Implement

The images to be used for this assignment are as follows. I received one vertical picture and one horizontal picture, and I had to take additional pictures using my cell phone.


**Given images**

The first thing to do is to replace the provided RAW file with a .tiff file using 'dcraw', a command-line tool. Using this tool, it is possible to change the RAW file to a .tiff file. So first, I installed ‘dcraw’ on my computer.

```
(base) simjaeyoon@simjaeyoonui-MacBookPro ~ % brew install dcraw
==> Downloading https://ghcr.io/v2/homebrew/core/jpeg/manifests/9d
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/jpeg/blobs/sha256:c565929a4901365a3408b57275802f943625c1e29e1b48a186edd2e97d8c0bd
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha256:c565929a4901365a3408b57275802f943625c1e29e1b4
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/jasper/manifests/2.0.33
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/jasper/blobs/sha256:7462315306489ccf06bddad20b9ba97b5872574f8155dfa7f5d316d905da7
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha256:7462315306489ccf06bddad20b9ba97b5872574f8155d
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/libtiff/manifests/4.3.0
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/libtiff/blobs/sha256:09f08e1168780c12c8f1526038eb4f4692624c85a9e78099b8ae2c58e39f
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha256:09f08e1168780c12c8f1526038eb4f4692624c85a9e78
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/little-cms2/manifests/2.12
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/little-cms2/blobs/sha256:70eaa9b280425731f7dcf104e75d4ae1e6a90421e1a741e0fe828593
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha256:70eaa9b280425731f7dcf104e75d4ae1e6a90421e1a74
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/dcraw/manifests/9.28.0
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/dcraw/blobs/sha256:fc0e1b6d2ac47be836929a68ee33d693bb2e455c4ecdd1dee7beafb3a5c123
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha256:fc0e1b6d2ac47be836929a68ee33d693bb2e455c4ecdd
######################################################################## 100.0%
==> Installing dependencies for dcraw: jpeg, jasper, libtiff and little-cms2
==> Installing dcraw dependency: jpeg
==> Pouring jpeg--9d.big_sur.bottle.tar.gz
🍺  /usr/local/Cellar/jpeg/9d: 21 files, 954.0KB
==> Installing dcraw dependency: jasper
==> Pouring jasper--2.0.33.big_sur.bottle.tar.gz
🍺  /usr/local/Cellar/jasper/2.0.33: 41 files, 1.3MB
==> Installing dcraw dependency: libtiff
==> Pouring libtiff--4.3.0.big_sur.bottle.tar.gz
🍺  /usr/local/Cellar/libtiff/4.3.0: 249 files, 4.4MB
==> Installing dcraw dependency: little-cms2
==> Pouring little-cms2--2.12.big_sur.bottle.tar.gz
🍺  /usr/local/Cellar/little-cms2/2.12: 21 files, 1.3MB
==> Installing dcraw
==> Pouring dcraw--9.28.0.big_sur.bottle.tar.gz
🍺  /usr/local/Cellar/dcraw/9.28.0: 4 files, 384.4KB
```

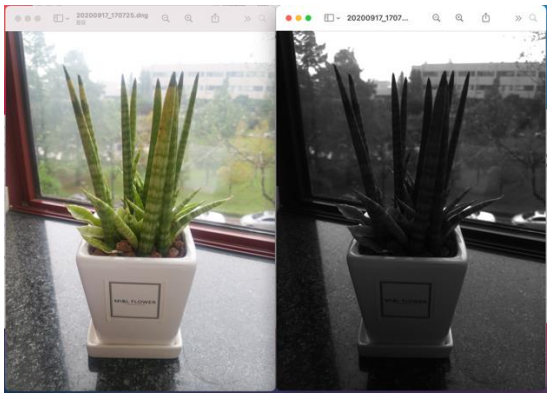
**install 'dcraw'**

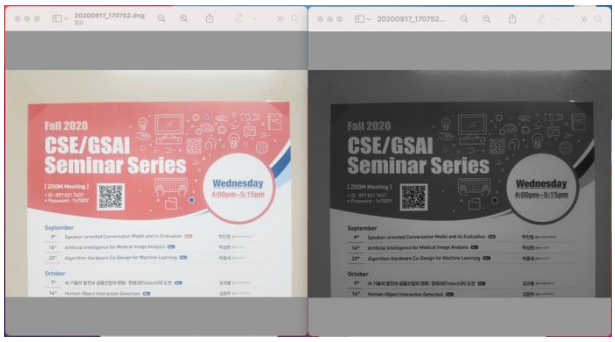
And the command '-4 -d -v -T' was input and converted into a .tiff file.

```
(base) simjaeyoon@simjaeyoonui-MacBookPro image1 % ls
20200917_170725.dng     20200917_170725.jpg
(base) simjaeyoon@simjaeyoonui-MacBookPro image1 % dcraw -4 -d -v -T 20200917_170725.dng
Loading Samsung SM-G935L image from 20200917_170725.dng ...
Scaling with darkness 0, saturation 1023, and
multipliers 2.436696 1.000000 1.479427 1.000000
Building histograms...
Writing data to 20200917_170725.tiff ...
(base) simjaeyoon@simjaeyoonui-MacBookPro image1 % ls
20200917_170725.dng     20200917_170725.jpg     20200917_170725.tiff
(base) simjaeyoon@simjaeyoonui-MacBookPro image1 %
```

**convert RAW file to .tiff file**

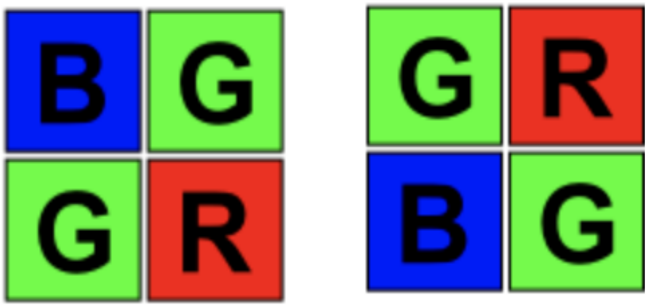
The created tiff file is as follows.






**Raw files and .tiff files**

The two images use different Bayer filters. The first image uses a BGGR Bayer filter, and the second image uses a GBRG Bayer filter. In fact, 90 degree rotation is the same pattern, but it was necessary to divide it because vertical and horizontal images were created using the same camera.




**BGGR Bayer filter(L), GRBG Bayer filter(R)**

So, I changed the image provided first and implemented the functions using OpenCV. I used python 3.7.11 and OpenCV 4.5.3, and my cell phone camera was taken using Samsung Galaxy S10 5G.

## 2.1 Automatic White Balance (Gray World Assumption)

When people take picture of the white objects in the room, camera will simply capture the incoming light, so the objects in the picture will be yellowish, making the picture look unnatural. To avoid this, cameras also try to cancel out the color of the illuminant and adjust the colors of objects. This processing is called white balance. Automatic white balance automatically finds the white balance parameters.

Gray world assumption is a technique based on the assumption that the mean values for each R, G, B channel are the same, and then multiplied by the gain so that the mean values for each R, G, B channel in the image are the same. When various colors exist, the overall average in one image exists only in brightness and has no color component, so the average of each channel r, g, and b becomes the same. The algorithm performs automatic white balance using the following equation.

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} G_{avg}/R_{avg} & & \\ & 1 & \\ & & G_{avg}/B_{avg} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

For the automatic white balance using gray world assistance, the calculation was conducted as follows. By dividing the sum of the intensities for each channel, average was obtained, and using this, the coefficient to multiply with the existing (R, G, B) was calculated.

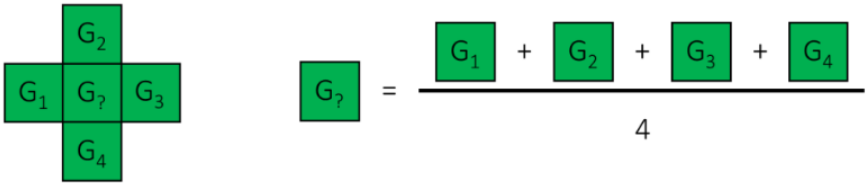
```
BGR_sum = {'B_sum': 0, 'G_sum': 0, 'R_sum': 0}
BGR_num = {'B_num': 0, 'G_num': 0, 'R_num': 0}

# BGGR filter: Blue is (even, even), Green is (odd, even)/(even, odd), Red is (odd, odd)
for y in range(height):
    for x in range(width):
        # Blue case
        if (x % 2 == 0 and y % 2 == 0):
            BGR_sum['B_sum'] += img[y, x]
            BGR_num['B_num'] += 1

        # Red case
        elif (x % 2 == 1 and y % 2 == 1):
            BGR_sum['R_sum'] += img[y, x]
            BGR_num['R_num'] += 1

        # Green case
        else:
            BGR_sum['G_sum'] += img[y, x]
            BGR_num['G_num'] += 1

# Find the white balance parameters
BGR_average = {'B_avg': BGR_sum['B_sum'] / BGR_num['B_num'],
               'G_avg': BGR_sum['G_sum'] / BGR_num['G_num'],
               'R_avg': BGR_sum['R_sum'] / BGR_num['R_num']}

BGR_coefficient = {'B_coe': BGR_average['G_avg'] / BGR_average['B_avg'],
                   'G_coe': 1,
                   'R_coe': BGR_average['G_avg'] / BGR_average['R_avg']}

# Apply white balance
for y in range(height):
    for x in range(width):
        # Blue case
        if (x % 2 == 0 and y % 2 == 0):
            img[y, x] = min(BGR_coefficient['B_coe'] * img[y, x], 1)

        # Red case
        elif (x % 2 == 1 and y % 2 == 1):
            img[y, x] = min(BGR_coefficient['R_coe'] * img[y, x], 1)

        # Green case
        else:
            img[y, x] = min(BGR_coefficient['G_coe'] * img[y, x], 1)
```

Images are usually in the form of an 8-bit unsigned integrator. In this case, when image processing is performed, only integers from 0 to 255 are used, causing inconvenience. Again, it exceeds 255 or in such a case, an overflow problem occurs. So, I changed it to 32bit float format and proceeded with the calculation from 0 to 1 and finally returned it to its original form.

## 2.2 CFA Interpolation

CFA interpolation is one of the most important steps for the visual quality of images. I make the function of bilinear interpolation in this assignment. Bilinear interpolation can be implemented by simply computing the average of 4 neighbors as shown below.

Bilinear interpolation: Simply average your 4 neighbors.




Neighborhood changes for different channels:

I made bilinear interpolation for two cases, such as white balance. BGGR Bayer filter and GRBG Bayer filter were created, and first, an image filled with 0 of the same size was arbitrarily created. In addition, the calculation of the middle part excluding the border part of this image was performed first. Values should be calculated for each channel in the order of B, G, and R, and B, G, and R intensity values were determined using 8 pixels adjacent to the original position.

```
interpolated_img = np.zeros((height, width, 3), dtype = np.uint8)

# Region exluding the edges
for y in range(1, height - 1):
    for x in range(1, width - 1):
        # Blue case
        if(x % 2 == 0 and y % 2 == 0):
            interpolated_img[y, x, 0] = img[y, x]
            interpolated_img[y, x, 1] = (img[y - 1, x] + img[y + 1, x] + img[y, x - 1] + img[y, x + 1]) / 4
            interpolated_img[y, x, 2] = (img[y - 1, x - 1] + img[y - 1, x + 1] + img[y + 1, x - 1] + img[y + 1, x + 1]) / 4

        # Red case
        elif (x % 2 == 1 and y % 2 == 1):
            interpolated_img[y, x, 0] = (img[y - 1, x - 1] + img[y - 1, x + 1] + img[y + 1, x - 1] + img[y + 1, x + 1]) / 4
            interpolated_img[y, x, 1] = (img[y - 1, x] + img[y + 1, x] + img[y, x - 1] + img[y, x + 1]) / 4
            interpolated_img[y, x, 2] = img[y, x]

        # Green case
        else:
            interpolated_img[y, x, 0] = (img[y - 1, x] + img[y + 1, x]) / 2
            interpolated_img[y, x, 1] = (img[y - 1, x - 1] + img[y - 1, x + 1] + img[y + 1, x - 1] + img[y + 1, x + 1] + img[y, x]) / 5
            interpolated_img[y, x, 2] = (img[y, x - 1] + img[y, x + 1]) / 2
```

And I thought of a total of 4 cases for input image. The case was divided to process the borders because the order of the Bayer filter varies depending on whether the horizontal and vertical lengths are odd or even. In addition, the calculation was performed from the four vertices of the corner, and three neighboring pixels were used based on the vertex. And finally, the inner part of the remaining border was calculated using 5 adjacent values to the original position.

```
    # Upper left(B)
    interpolated_img[0, 0, 0] = img[0, 0]
    interpolated_img[0, 0, 1] = (img[0, 1] + img[1, 0]) / 2
    interpolated_img[0, 0, 2] = img[1, 1]

    # Upper right(B)
    interpolated_img[0, width - 1, 0] = img[0, width - 1]
    interpolated_img[0, width - 1, 1] = (img[0, width - 2] + img[1, width - 1]) / 2
    interpolated_img[0, width - 1, 2] = img[1, width - 2]

    # Bottom left(B)
    interpolated_img[height - 1, 0, 0] = img[height - 1, 0]
    interpolated_img[height - 1, 0, 1] = (img[height - 2, 0] + img[height - 1, 1]) / 2
    interpolated_img[height - 1, 0, 2] = img[height - 2, 1]

    # Bottom right(B)
    interpolated_img[height - 1, width - 1, 0] = img[height - 1, width - 1]
    interpolated_img[height - 1, width - 1, 1] = (img[height - 2, width - 1] + img[height - 1, width - 2]) / 2
    interpolated_img[height - 1, width - 1, 2] = img[height - 2, width - 2]

    for x in range(1, width - 1):
        if (x % 2 == 0):
            # Blue case
            interpolated_img[0, x, 0] = img[0, x]
            interpolated_img[0, x, 1] = (img[0, x - 1] + img[0, x + 1] + img[1, x]) / 3
            interpolated_img[0, x, 2] = (img[1, x - 1] + img[1, x + 1]) / 2
            # Blue case
            interpolated_img[height - 1, x, 0] = img[height - 1, x]
            interpolated_img[height - 1, x, 1] = (img[height - 1, x - 1] + img[height -1 , x + 1] + img[height -2, x]) / 3
            interpolated_img[height - 1, x, 2] = (img[height - 2, x - 1] + img[height - 2, x + 1]) / 2

        else:
            # Green case
            interpolated_img[0, x, 0] = (img[0, x - 1] + img[0, x + 1]) / 2
            interpolated_img[0, x, 1] = (img[1, x - 1] + img[1, x + 1] + img[0, x]) / 3
            interpolated_img[0, x, 2] = img[1, x]
            # Green case
            interpolated_img[height - 1, x, 0] = (img[height - 1 , x - 1] + img[height - 1, x + 1]) / 2
            interpolated_img[height - 1, x, 1] = (img[height - 1, x] + img[height - 2, x - 1] + img[height - 2, x + 1]) / 3
            interpolated_img[height - 1, x, 2] = img[height - 2, x]

    for y in range(1, height - 1):
        if (y % 2 == 0):
            # Blue case
            interpolated_img[y, 0, 0] = img[y, 0]
            interpolated_img[y, 0, 1] = (img[y - 1, 0] + img[y + 1, 0] + img[y, 1]) / 3
            interpolated_img[y, 0, 2] = (img[y - 1, 1] + img[y + 1, 1]) / 2
            # Blue case
            interpolated_img[y, width - 1, 0] = img[y, width - 1]
            interpolated_img[y, width - 1, 1] = (img[y - 1, width - 1] + img[y + 1, width -1] + img[y, width - 2]) / 3
            interpolated_img[y, width - 1, 2] = (img[y - 1, width - 2] + img[y + 1, width - 2]) / 2

        else:
            # Green case
            interpolated_img[y, 0, 0] = (img[y - 1, 0] + img[y + 1, 0]) / 2
            interpolated_img[y, 0, 1] = (img[y, 0] + img[y - 1, 1] + img[y + 1, 1]) / 3
            interpolated_img[y, 0, 2] = img[y, 1]
            # Green case
            interpolated_img[y, width - 1, 0] = (img[y - 1, width -1] + img[y - 1, width - 1]) / 2
            interpolated_img[y, width - 1, 1] = (img[y, width - 1] + img[y - 1, width - 2] + img[y + 1, width - 2]) / 3
            interpolated_img[y, width - 1, 2] = img[y, width - 2]
```

## 2.3 Gamma Correction

Using White balance and CFA interpolation, the RAW image was even colored into three channels. And gamma correction will be performed to further correct this image. I used the following equation.

$$I' = I^{1/\gamma}$$

Gamma correction refers to nonlinear deformation of the intensity signal of light using non-linear transfer function in video cameras and computer graphics. In gamma correction, the image can be darkened and brightened by adjusting the gamma value. The pixel value of the image itself may be changed, but the overall balance may be changed through this technique. It better reflects the human perception. The human perception is non-linear to the brightness. Human eyes are more sensitive to dark tones than bright tones.

And during the image processing process, the image format was changed to 32bit float format again this time and then returned to its original form.

```
c = 1
    gamma = 2.2
    exp = 1 / gamma

    for y in range(height):
        for x in range(width):
            for color in range(0,3):
                img[y, x, color] = c * img[y, x, color] ** exp
```

## 2.4 Making images

Once each function has been implemented, the process of creating an image using it is simply organized. First, the path to receive the image and the path to store were created.

```
# Image folder path
path = "../images/image"
folder_number = 4
input_tiff_image_name = sys.argv[1]

input_image_location = path + str(folder_number) + '/'
output_image_location = path + str(folder_number) + '/'

output_white_balance_image_name = input_tiff_image_name[0:-5] + '_wb' + '.jpg'
output_cfa_interpolation_image_name = input_tiff_image_name[0:-5] + '_cfa' + '.jpg'
output_white_balance_and_cfa_interpolation_image_name = input_tiff_image_name[0:-5] + '_wb_cfa' + '.jpg'
output_cfa_interpolation_and_gamma_correction_image_name = input_tiff_image_name[0:-5] + '_cfa_gc' + '.jpg'
output_jpg_image_name = input_tiff_image_name[0:-5] + '.jpg'
```

And I operated three image processing functions created by receiving images. For comparison, a JPEG image was created, including white balance only, CFA interpolation only, white balance and CFA interpolation, CFA interpolation and gamma correction, and finally, all three functions were operated.

```
# 0. Read tiff image
input_image = cv2.imread(input_image_location + input_tiff_image_name, cv2.IMREAD_GRAYSCALE)

input_image_height, input_image_width = input_image.shape[0], input_image.shape[1]

### Perform 3 operations ###
# 1. Only white balance
if(input_image_height > input_image_width):
    output_white_balance_image = white_balance_BGGR_filter(input_image)
else:
    output_white_balance_image = white_balance_GRBG_filter(input_image)
```

```
# 2. Only CFA interpolation
if(input_image_height > input_image_width):
    output_cfa_interpolation_image = CFA_interpolation_BGGR_filter(input_image)
else:
    output_cfa_interpolation_image = CFA_interpoloation_GRBG_filter(input_image)

# 3. White balance + CFA interpolation
if(input_image_height > input_image_width):
    output_white_balance_and_cfa_interpolation_image = CFA_interpolation_BGGR_filter(output_white_balance_image)
else:
    output_white_balance_and_cfa_interpolation_image = CFA_interpoloation_GRBG_filter(output_white_balance_image)

# 4. CFA interpolation + gamma correction
output_cfa_interpolation_and_gamma_correction_image = gamma_correction(output_cfa_interpolation_image)

# 5. White balance + CFA interpolation + gamma correction
output_final_image = gamma_correction(output_white_balance_and_cfa_interpolation_image)
```

And the created images were stored in a folder containing the .tiff file.

```
# Write jpg images
cv2.imwrite(output_image_location + output_white_balance_image_name, output_white_balance_image)
cv2.imwrite(output_image_location + output_cfa_interpolation_image_name, output_cfa_interpolation_image)
cv2.imwrite(output_image_location + output_white_balance_and_cfa_interpolation_image_name, output_white_balance_and_cfa_interpolation_image)
cv2.imwrite(output_image_location + output_cfa_interpolation_and_gamma_correction_image_name, output_cfa_interpolation_and_gamma_correction_image)
cv2.imwrite(output_image_location + output_jpg_image_name, output_final_image)
```

## 3. Result

Now, I will proceed with image processing with the program I created and compare the results. The following is the result of proceeding with the original.tiff file and white balance. When white balance was conducted using the original file, the difference was not known visually.

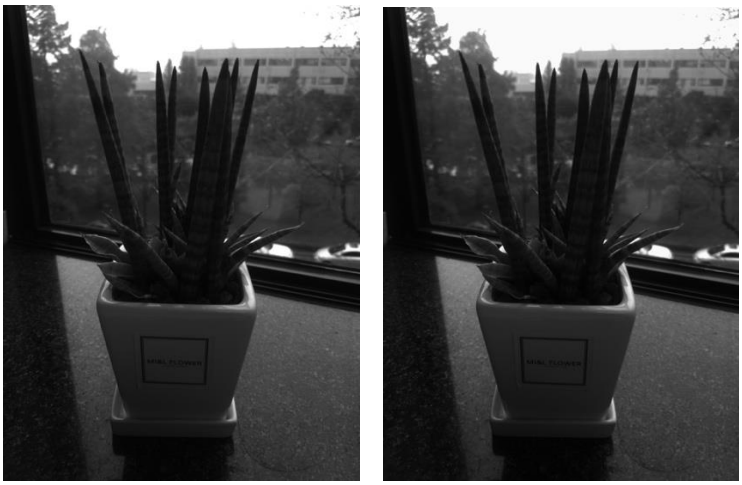


**Image 1 original .tiff file(L), white balance(R)**

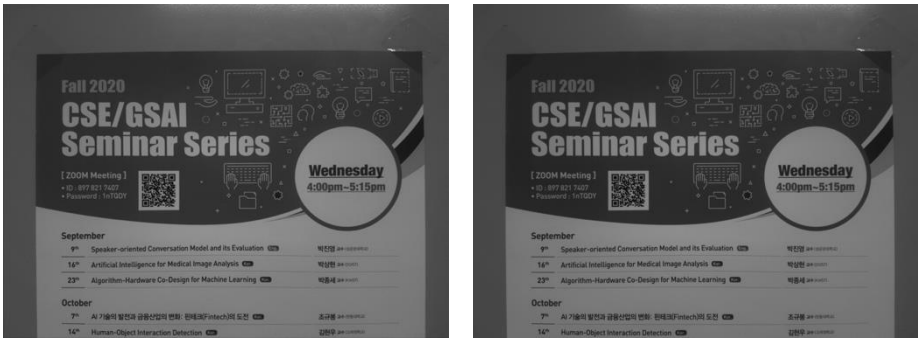



**Image 2 original .tiff file(L), white balance(R)**

So, I compared the results of performing the .tiff file and only CFA interpolation. When I colored the 3 channels, the difference was clear.

**Image 1 original .tiff file(L), CFA interpolation(R)**


**Image 2 original .tiff file(L), CFA interpolation(R)**

So, I compared it as follows. The first picture is the result of only white balance, and the second picture is the result of only CFA interpolation. And in the third picture, after white balance, CFA interpolation was performed. Even if the same CFA interpolation was applied, the difference in white balance was revealed.

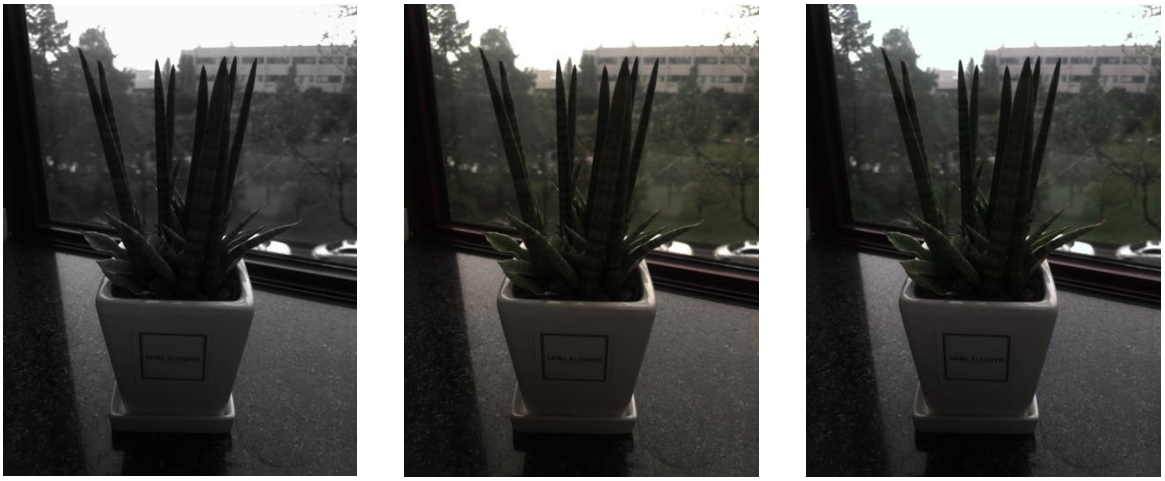


**Image 1 only white balance(L), only CFA interpolation(C), white balance + CFA interpolation(R)**


**Image 2 only white balance(L), only CFA interpolation(C), white balance + CFA interpolation(R)**

In the case of image 1, I can see the problem that the sky is originally white and comes out in a little blue. Overall, a blue atmosphere was formed, and in the case of image 2, the yellow atmosphere disappeared from the object that received yellow light as the purpose of white balance. In my opinion, the white balance was properly applied to both images. So, I checked the white balance parameter value. The intensity of each channel seems to have been accurately calculated.


**Image 1 blue coefficient and red coefficient(L), Image 2 blue coefficient and red coefficient(R)**

$$B_{coe} = {G_{avg}}/{B_{avg}} = 1.14 \ \& \ 1.31 \qquad R_{coe} = {G_{avg}}/{R_{avg}} = 0.93 \ \& \ 0.71$$

Next, gamma correction was compared. The image was corrected to a value of gamma = 2.2 using the results of white balance and CFA interpolation. As a result, it certainly looked comfortable to the eyes. The dark part of the image brightened up, making the details clear.

**Image 1 white balance + CFA interpolation(L), white balance + CFA interpolation + gamma correction(R)**

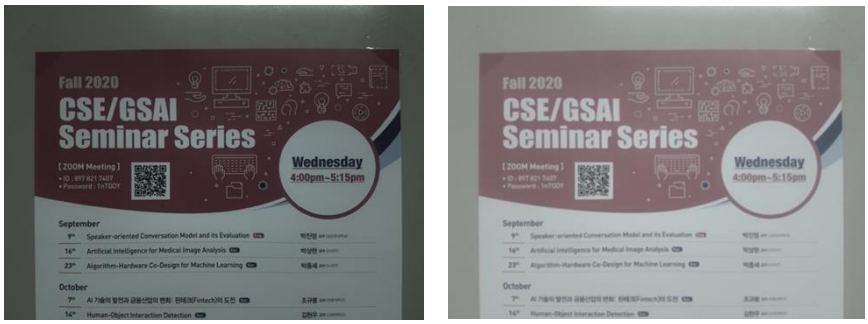



**Image 2 white balance + CFA interpolation(L), white balance + CFA interpolation + gamma correction(R)**

And it can also be seen that the brightness changes depending on the gamma value.

**Image 1 white balance + CFA interpolation + gamma correction with gamma = 1.6 (L),**
**white balance + CFA interpolation + gamma correction with gamma = 2.2(R)**

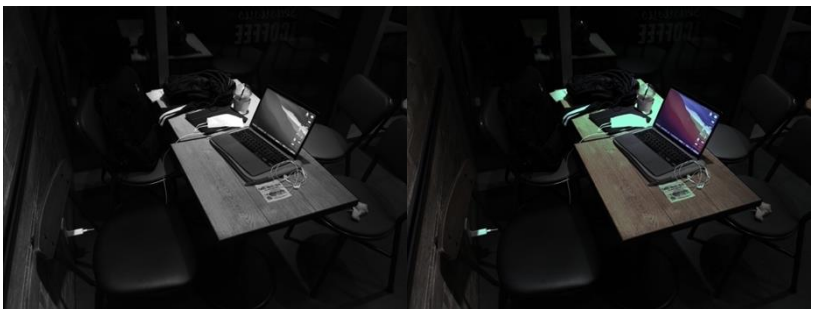
## 4. Discussion

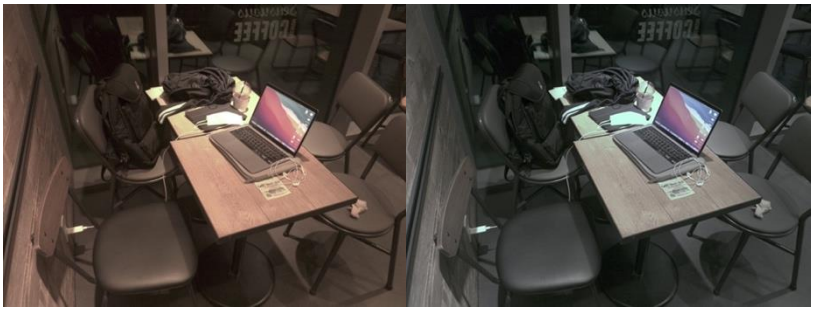
I compared the results of the image processing process using the code I wrote and the images that executed the same function using 'dcraw'. White balance and gamma correction were applied to the image by entering the command '-a -g 2.2 4.5'.

**Image 1 my result(L), ‘dcraw’ result(R)**


**Image 2 my result(L), ‘dcraw’ result(R)**

It can be seen that the results of the two images received as assignments were almost similar. In addition, I used the Samsung Galaxy 10 5G smartphone to take pictures of the cafe and coffee cup I was studying and use them as images. Since it's a cafe, the overall yellow atmosphere could be seen throughout the image using yellow bulbs.

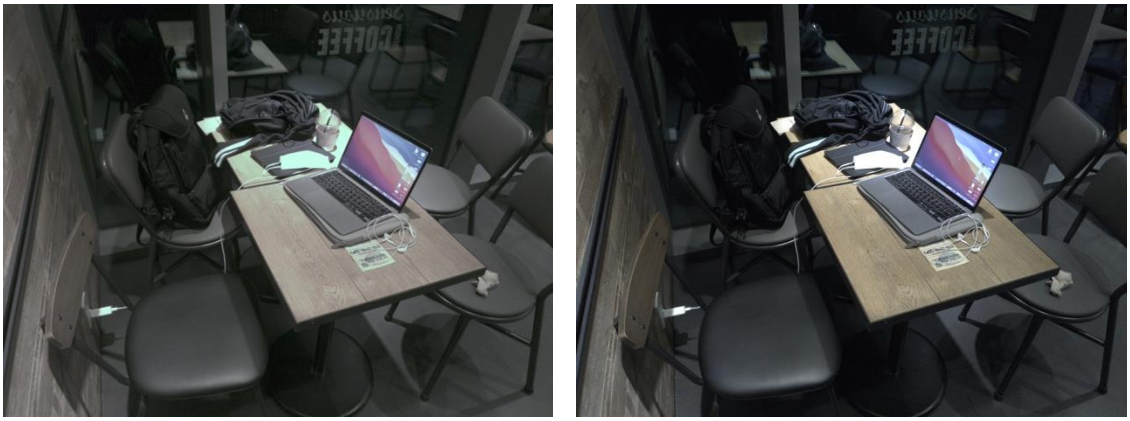




**My image 1 .tiff image(UL), white balance + CFA interpolation(UR),**
**CFA interpolation + gamma correction(BL), white balance + CFA interpolation + gamma correction(BR)**

**My image 2 .tiff image(UL), white balance + CFA interpolation(UR),**
**CFA interpolation + gamma correction(BL), white balance + CFA interpolation + gamma correction(BR)**

I compared white balance, CFA interposition, and gamma correction with the photos I took. The functions shown by each function seem to have been well applied to suit the purpose. And I compared the results of applying 'dcraw' using the images I took and the results using my code.



**My image 1 my result(L), ‘dcraw’ result(R)**


**My image 2 my result(L), ‘dcraw’ result(R)**

Overall, I am satisfied with the similar results. However, there are some problems to think about. First of all, in implementing automatic white balance, a blue atmosphere is formed overall. And in the case of red posters, although they are red, they tend to be blue. As confirmed above, it is not a theoretical computational problem, but the color of the object in the image or the existing color seems to have an effect. And I think the computational cost is a little high. It would be good to implement bilinear interpolation more effectively.