Homework Assignment 1

Instructions to Run Code

From within `src/` run the following:

```
python isp.py --args

Args:
--scr_dir: Path to data/ directory (Required)
--out_dir: Path to write data to (Required)
--bayer_pattern: One of [grgb, rggb, bggr, gbrg] (Required)
--white_balance One of [white, grey, preset, manual] (Required)
--brightness_percentage: Value from 1-100 (Required)
--output_img_format: One of [png, jpg] (Required)
--compression_quality: Value from 1-100. Only used when saving outputs as
JPEG (Optional)
```

Developing Raw Images

1.1: Implementing a Basic Image Processing Pipeline

Raw Image Conversion

Reconnaissance run of dcraw:

```
dcraw -4 -d -v -w -T <RAW_filename>
```

Scaling with darkness **150**, saturation **4095**, and multipliers **2.394531 1.000000 1.597656 1.000000**

Python Initials

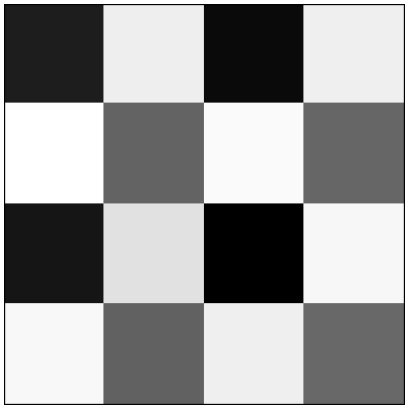
RAW Image read in and cast to double precision array using function `read_im()` in `isp.py`. The image has a **width of 6016, height of 4016, and 16 bits.**

Linearization

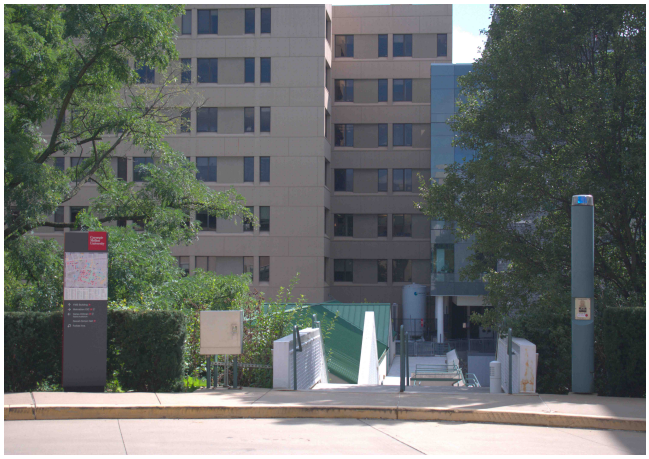
Implemented in `linearize()` in `isp.py`.

Identifying the correct Bayer Pattern

I found the easiest way of identifying the correct Bayer pattern was to enumerate the 4 possibilities and see which image looked the best after development. However, a way of possibly narrowing down the options is to plot a small crop in a uniformly green area so that we can identify whether the Bayer pattern is one off [rggb', bggr] or [grbg, gbrg]. Plotting a 4 x 4 patch in a predominantly green area yields the following:

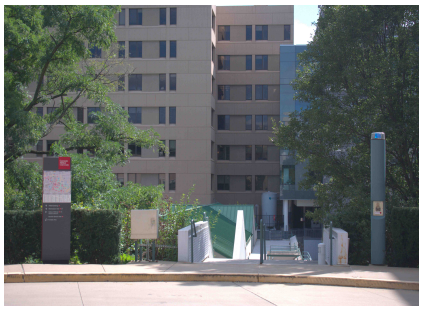


We can see that every 2 x 2 square has practically the same top right and bottom left color, indicating that the Bayer patten is either **rggb** or **bggr**. Looking at the fully developed images using **bggr (left)** and **rggb (right)** as the Bayer pattern, it is clear that **rggb (right)** is the correct Bayer pattern.

bggr	rggb
	

White Balancing

Implemented in `white_balance()` in `isp.py`. After development, these were the images generated using the **white world assumption (left), grey world assumption (middle), and camera presets (right)** for white balancing.

White World Assumption	Grey World Assumption	Camera Presets
		

In my opinion, the image white balanced with the **grey world assumption (middle)** looks the best.

Demosaicing

Implemented in `demosaic()` in `isp.py`. Below is the demosaiced image without any color space correction, brightness adjustment, or gamma encoding.

Color Space Correction

Implemented in `color_space_correction()` in `isp.py`. Below is the color space corrected image without any brightness adjustment or gamma encoding.

Brightness Adjustment and Gamma Encoding

Implemented in `brighten_image()`, `gamma_encoding()`, and `tone_reproduction()` in `isp.py`. The table below shows the fully developed image with brightening percentages of **20% (left), 35% (middle), and 50% (right)**. Of the 3 brightness percentages below, I think **20% (left)** looks the best.

20%	35%	50%
		

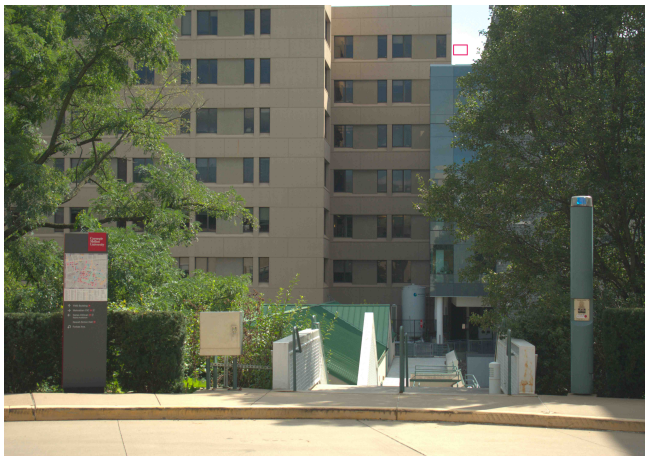
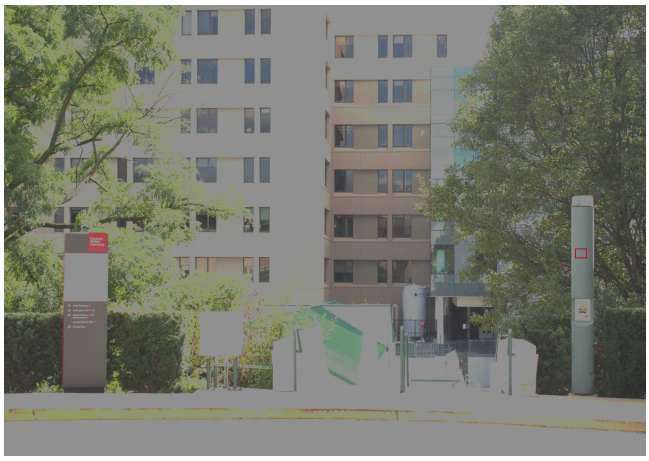
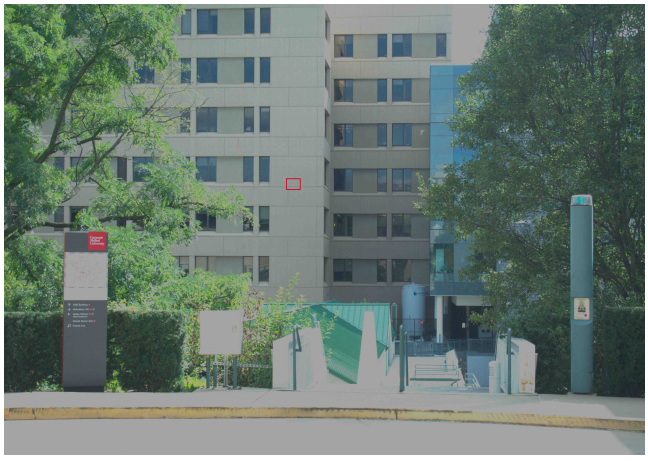
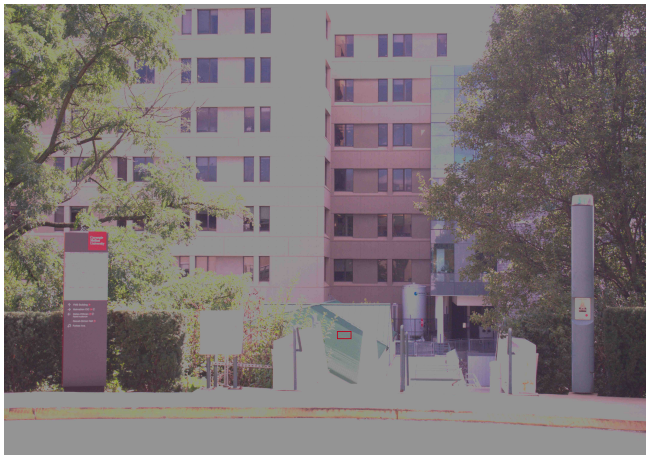
Compression

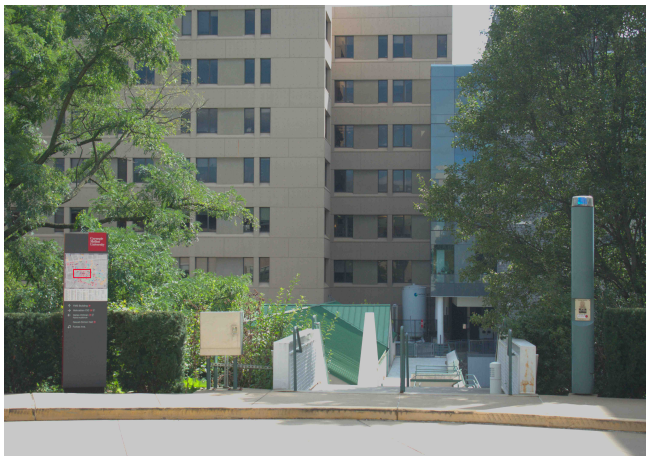
Implemented in `write_im()` in `isp.py`. The compression ratio between the fully developed image saved as a PNG and as a JPEG with 95% compression quality is **5.107**, yielding an approximately **5:1** compression with no clear visual differences.

The lowest compression quality that still yields an image with the same quality as the **PNG** is **25%** with a compression ratio of **41.050** yielding an approximately **41:1** compression with no clear visual differences.

1.2: Manual White Balance

Each image below shows the white balanced image after manually selecting a patch (outlined in red in each image). As expected, patches containing regions that we expect to be white yielded the best white balance while patches containing other dominant colors yielded images with various undesirable tints.



1.3 Learn to use dcraw

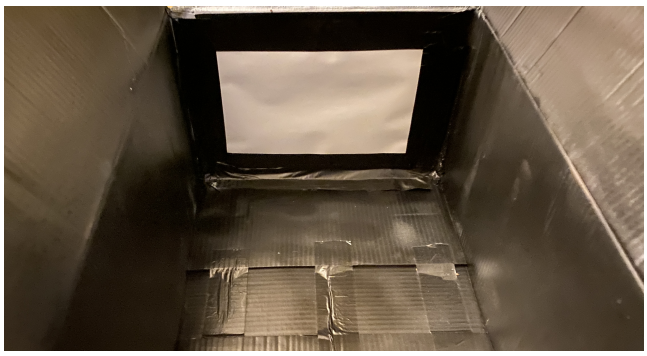
I ran the following command to yield the fully developed image pictured below.

```
dcraw -k 150 -S 4095 -a -q 0 -o 1 -b 1.5 -g 2.4 12.92 ../data/campus.nef

-k 150 -S 4095: Set black to 150 and white to 4095 (Linearize)
-a: Calculate the white balance by averaging the entire image. (White
Balance)
-q 0: Use high-speed, low-quality bilinear interpolation (Demosaic)
-o 1: Color correct to the sRGB colorspace (Color Space Correction)
-b 1.5 (Brighten)
-g 2.4 12.92: Apply sRGB Gamma (Gamma Encodings)
```

Camera Obscura

2.1 Build the Pinhole Camera

Outside of Pinhole Camera	Inside of Pinhole Camera	Full Outside Setup
		

In this pinhole camera, all faces except one were spray painted black. The remaining face was spray painted white. All open edges were sealed with black tape. After taking some images with this setup, each image contained artifacts due to the texture of the screen, a result of the ridges on the cardboard as well as the uneven spray paint job. To overcome this, I instead took a regular sheet of printer paper and taped it with black tape on the inside of the box as seen in the figure above. Although this decreased the size of the screen, it yielded images without any of these texture artifacts. The focal length of the pinhole camera was **45.72 cm**. The **screen size was 15.24 x 25.4 cm**. Therefore the **FOV was 36.38095140364 degrees**, calculated using the following formula: **FOV = 2*arctan(S/2f)** where **S** is the diagonal length of the screen and **f** is the focal length.

2.2 Use the Pinhole Camera

0.1mm	1mm	5mm
		
		

0.1mm	1mm	5mm
		

I used pinhole diameters of **0.1mm, 1mm, and 5mm** as seen above. Unexpectedly, the 0.1mm pinhole yielded the blurriest pictures. This is more due to the difficulty in finding the right focus during long exposure capture than it is because of the pinhole diameter. The 1mm diameter pinhole yielded the sharpest images while the 5mm diameter pinhole yielded images with the most vibrant colors. It is possible, however, that these differences could all be reconciled with the right settings for aperture, ISO, and exposure.