1.1

Raw Image Conversion

To do the reconnaissance run on the raw file, I executed "dcraw -4 -d -v -w -T C:\Users\Luke\ISP_homework\data\baby.nef" in the terminal which returned the following information: black = 0, white = 16383, r_scale = 1.628906, g_scale = 1.000000, and b_scale = 1.386719. To create my .tiff file that I will use for the rest of the project, I ran "dcraw -4 -D -T C:\Users\Luke\ISP_homework\data\baby.nef" in the terminal resulting in a baby.tiff file.

Python Initials

Here I read and saved my .tiff file to image by using skimage's imread function. I then used the .shape function to find the images height and width which was 2844 and 4284 respectively. The bits per pixel for the image is 16 which was found by looking at the properties of the image. I then converted the image into a double array by using the .astype function.

Linearization

I converted the image into a linear array with the following operation, (image_double - black) / (white - black) and saved that to image_linear variable. I then used the .clip function with 0 and 1 in the 2nd and 3rd parameters.

Identifying the correct Bayer pattern

The Bayer pattern that applies to this image is RGGB. I identified this by going into a photo editor and zooming into the top left corner of the image.

White Balancing

I first created a simple demosaicing function that so that the image could be run through that before being white balanced to follow the ISP Pipeline. I then created a function for each of the three white balancing algorithms. The white and gray world algorithms use the average color in the image which was found using the .mean function. The average color would then be used to determine the scale factor which was multiplied to the image to get our white balancing. The third algorithm determined scale factor by using the r_scale, g_scale, and b_scale found and recorded earlier. From these three images that these algorithms produce, I like the white world balanced one the best as it is brighter than the other and easier to see details.

Demosaicing

To do bilinear interpolation for demosaicing I created a function called bilinear_demosaicing. In the function, I separate the color channels for each color to use in interpolation and save to variables. I then create interpolation functions for each color channel which uses the variables from the previous step. I then interpolate for all pixels in the image using the interpolation functions. The function finally returns an interpolated demosaic image. I then use this demosaic image in my white world balancing function to get my demosaic-white balanced image that will be pushed down the ISP pipeline.

Color Space Correction

For this step I first found the camera specific matrix for $M_{XYZ \rightarrow cam}$ in dcraw's raw code. The camera used to capture the raw image is a Nikon D3400 which has the 1x9 matrix of [6988,-1384,-714,-5631,13410,2447,-1485,2204,7318]. In my code I reshape this matrix to be a 3x3 and divided each value by 10,000. I then use the provided $M_{sRGB \rightarrow XYZ}$ to find the matrix multiplication product of $M_{sRGB \rightarrow XYZ}$ and $M_{XYZ \rightarrow cam}$. I then ensure that each row in $M_{sRGB \rightarrow cam}$

matrix has sum 1. I use numpy inverse function to find $M_{sRGB \rightarrow cam}$ inverse. I then iterate over the pixels in my demosaic-white balanced image where each pixels Rcam, Gcam, and Bcam values are represented as a column vector which is then multiplied by the inverse $M_{sRGB \rightarrow cam}$ matrix. This results in my color space corrected image.

Brightness adjustment and gamma encoding

I created a brightness adjustment function with the parameters of an image and the desired mean intensity. The function first converts the image to gray scale using rgb2gray. It then calculates the images current mean intensity. The scale factor is then calculated by dividing desired mean intensity by current mean intensity. The image it then adjusted using the scale factor. Final, the adjusted image is clipped using the .clip function to ensure all pixel intensities are within the range [0,1] and the image is returned. While experimenting with the different percentages, I found that .5 looked the best to me as it just bright enough without being too bright. For gamma encoding I created another function called gamma_encode with one parameter being C_linear which should be our RGB values in out image. This parameter is used to determine which part of the piecewise function should be set to my C_nonlinear variable. This is done using the .where which has 3 parameters. The first is a condition, the second is a outcome, and the third is another outcome. The function checks the condition and if its true the second parameter is used and if it is false then the third parameter. The gamma_encode function then returns C_nonlinear. We then iterate through each pixel, apply gamma_encode to its linear RGB values and store the result in a new image.

Compression

To store the images, I used skimage's imsave function. One was saved without compression and the other with compression with quality set to 95. When looking at the two pictures, I can't tell the difference between the compressed and the uncompressed one. To find the compression ratio I looked at each files size by inspecting their properties. The .PNG had a size of 17239622 bytes and the .JPEG has a size of 4051692 bytes. This results in a compression ratio of 4.25. When changing the quality setting for the .JPEG, the lowest setting I found for which the compression image is indistinguishable from the original was 20. This resulted in a compression ratio of 47.74.

Final images

White world balancing

Gray world balancing

1.2

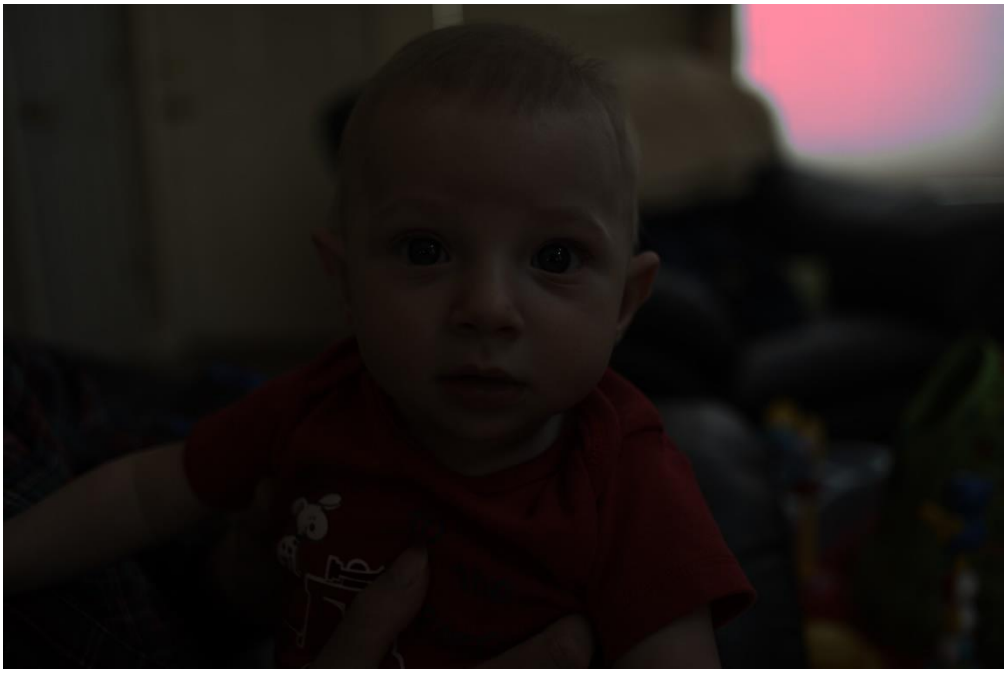
Manual White Balancing

For the manual white balancing code, I first created a function called select_white_patch that has one image parameter. This function requires the user to manually select the coordinates of the white patch they want to use. I want able to use ginput for white balancing as it required me to change the GUI backend which cause problems for other sections of my code and for some parts to not work as intended. Continuing with this solution, I created a second function called manual_white_balance with two parameters, one being an image and the second being the white patch coordinates. This function normalizes all three channels so that red, green, and blue values are all equal in the patch. It first finds the white patch color using the image and the coordinates. It then finds the scaling factor by dividing the max of white patch color by regular white patch

color. It then applies this scaling factor to the image and returns the image. I experimented with 3 different patches from the image:

Coordinates (0, 4200)

Coordinates (500, 4200)



Coordinates (700, 4200)

Out of the 3, I think the patch at coordinates (700, 4200) works the best as it is slightly brighter than the second image and doesn't have the slight green hue the first image has.

1.3

Learn to use dcraw

To use dcraw to preform the ISP pipeline I ran the following in the terminal "dcraw -q 0 -w -o 1 -g 2.2 1.0 C:\Users\Luke\ISP_homework\data\baby.nef" which gave me my baby.ppm which when converted looks like the following.

When comparing the 3 developed images, the dcraw image is noticeable darker than the other two. The image produced from my created pipeline has a noticeable slight green hue to it and its colors are slightly different from the other two. From the 3 developed images, I prefer the provide png image as the lighting is the best out of the three and the color is more bright and vibrant.

2.1

When building my pinhole camera, I used a box with dimensions of 11in x 9.5in x 6.5in. I choose to make the Screen 6 in tall and 10.75 in wide. I used white paper for the screen and used black poster paper to cover all of the other inside surfaces in the box.. On the outside of the box I made some supports to hold my phone camera in place so any movement was reduced. My focal length was 9.25 in and the field of view was 0.05404 degrees. I chose to use a smart phone

camera that had an exposure time of 10 secs. The 3 sizes I choose for the pinhole diameter were 0.1mm, 1mm, and 5mm. When using the 0.1mm pinhole, had my cameras ISO set to 50, for 1mm the ISO was 250, and for 5mm the ISO was 500.

2.2

The 3 sizes I choose for the pinhole diameter were 0.1mm, 1mm, and 5mm. When using the pinhole camera, I took pictures of some trees, the UNC observatory, and a lamp post. The main observation from all the different pinhole cameras pictures was that the pictures where more light was let in (i.e. bigger pinhole diameter) were much clearer and detailed. Another observation was that while using the 0.1 mm pinhole all the pictures came out with a pinkish hue and it was very difficult to see what was being photographed.