

ECE 239AS: Problem Set #1

This problem set will revisit the imaging pipeline, denoising, and HDR imaging from the corresponding lectures and textbook sections. The goal is to develop a working understanding of the digital image formation model that is central to most modern day imaging devices. This will be important in later course material when we discuss extreme imaging systems that. With that goal in mind, the organization of this notebook is split into three problems:

1. Imaging Pipeline
2. Denoising
3. High Dynamic Range Imaging

These three steps are a key part of the digital photography pipeline.

How to Implement

Sections of code you need to write are marked with:

STUDENT CODE HERE and have function names beginning with "student_". After the key student functions, we will include a call to a function named "check_code_correctness" which will evaluate your output against an output from the TAs.

NOTE: This will only check for correctness if you use the exact algorithm we do. We **highly encourage** you to explore new algorithms not covered in class, but these cannot be sanity checked by our code.

```
In [1]: !pip install rawpy  
!pip install imageio  
!pip install gdown  
  
Collecting rawpy  
  Downloading rawpy-0.18.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2  
014_x86_64.whl (1.9 MB)   
  ━━━━━━━━━━━━━━━━ 1.9/1.9 MB 19.3 MB/s eta  
0:00:00  
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-  
packages (from rawpy) (1.23.5)  
Installing collected packages: rawpy  
Successfully installed rawpy-0.18.1  
Requirement already satisfied: imageio in /usr/local/lib/python3.10/dis-  
t-packages (2.31.5)  
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-  
packages (from imageio) (1.23.5)  
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.  
10/dist-packages (from imageio) (9.4.0)  
Requirement already satisfied: gdown in /usr/local/lib/python3.10/dist-  
packages (4.6.6)  
Requirement already satisfied: filelock in /usr/local/lib/python3.10/di  
st-packages (from gdown) (3.12.4)  
Requirement already satisfied: requests[socks] in /usr/local/lib/python  
3.10/dist-packages (from gdown) (2.31.0)  
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-p  
ackages (from gdown) (1.16.0)  
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-p  
ackages (from gdown) (4.66.1)  
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python  
3.10/dist-packages (from gdown) (4.11.2)  
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.  
10/dist-packages (from beautifulsoup4->gdown) (2.5)  
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/l  
ib/python3.10/dist-packages (from requests[socks]->gdown) (3.3.0)  
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.1  
0/dist-packages (from requests[socks]->gdown) (3.4)  
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/pyt  
hon3.10/dist-packages (from requests[socks]->gdown) (2.0.7)  
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/pyt  
hon3.10/dist-packages (from requests[socks]->gdown) (2023.7.22)  
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in /usr/local/li  
b/python3.10/dist-packages (from requests[socks]->gdown) (1.7.1)
```

```
In [2]: from google.colab import drive  
drive.mount('/content/drive')
```

Mounted at /content/drive

```
In [3]: # Imports
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import rawpy
import json
import scipy
import imageio.v3 as iio
from scipy.ndimage import correlate
%matplotlib inline
rpath = '/content/drive/MyDrive/239AS/Assignment-1'
```

```
In [4]: def check_code_correctness(test_function, test_in, test_out):
    """Checks if the given function behaves as expected

    Args:
        test_function: Function to test
        test_in: The sample input to test on
        test_out: The expected output of the function given the specified
        input

    Returns:
        True if the function behaves as expected, False otherwise (wrong
        answer or error)
    """
    try:
        student_out = test_function(test_in)
    except NotImplementedError as err:
        print("Please implement and remove \\'raise NotImplementedError\\'")
        return False
    except RuntimeError as err:
        print("Please make sure you have the right dimensions and type")
        return False
    except:
        print("An exception occurred: could not compute output")
        return False

    try:
        if not( np.allclose( student_out, test_out) ):
            print("Test failed, student output does not match test output")
            return False
    except TypeError as err:
        print("Please make sure your function outputs the correct type")
        return False
    except:
        print("An exception occurred: could not check output")
        return False

    return True
```

```
In [ ]: # Get relevant data -- see references below for sources
!wget http://storage.googleapis.com/gresearch/refraw360/raw.zip
!unzip raw.zip

!wget https://www.pauldebevec.com/Research/HDR/SourceImages/Memorial_SourceImages.zip
!mkdir hdr
!unzip Memorial_SourceImages.zip -d hdr

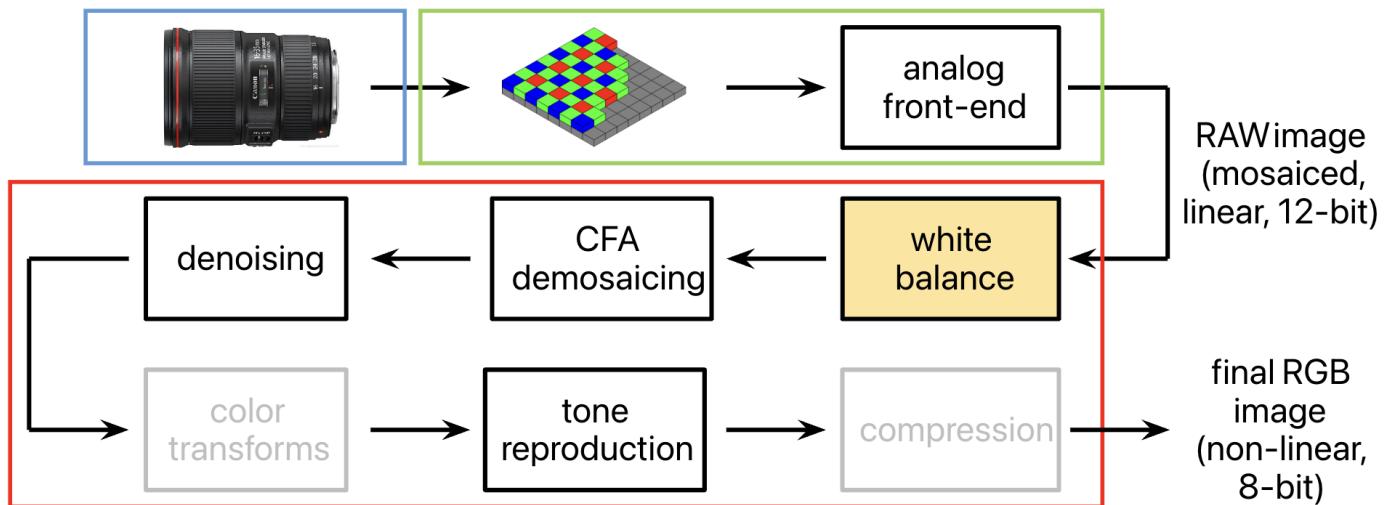
!gdown https://drive.google.com/uc?id=1bGFlFUdevThzXSLQn0J3PVUxtpt4tRua
!unzip sol.zip
```

```
Archive: sol.zip
  creating: sol/
    inflating: sol/hdr.npy
    inflating: __MACOSX/sol/.__hdr.npy
    inflating: sol/demosaick.npy
    inflating: __MACOSX/sol/.__demosaick.npy
    inflating: sol/white_balance.npy
    inflating: __MACOSX/sol/.__white_balance.npy
    inflating: sol/processed.npy
    inflating: __MACOSX/sol/.__processed.npy
    inflating: sol/house_img.jpeg
    inflating: __MACOSX/sol/.__house_img.jpeg
    inflating: sol/color_transform.npy
    inflating: __MACOSX/sol/.__color_transform.npy
```

Problem 1: Imaging Pipeline

Textbook Section: 2.2.4

In this problem, we will re-create part of the imaging pipeline, specifically demosaicing, white balancing, color transforms, and gamma encoding. By the end, you will be able to take in RAW images and output processed images. For each of the steps, we will suggest an implementation technique based on what was covered in Lecture 2.



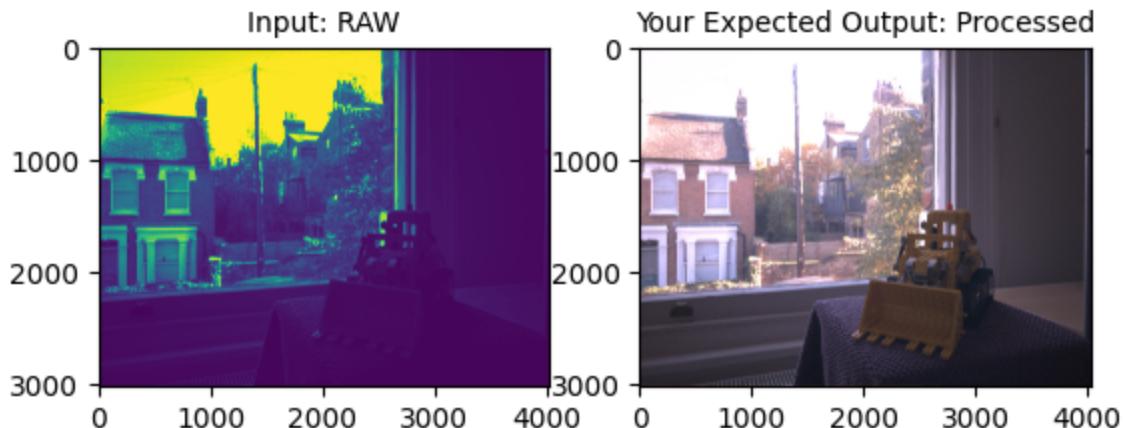
We load one specific image at first for code correctness checks, but feel free to try new images from the dataset or images you capture yourself!

```
In [ ]: dng_file = rpath + "/rawnerf/scenes/windowlegovary/raw/IMG_6796.dng"
meta_file = rpath + "/rawnerf/scenes/windowlegovary/raw/IMG_6796.json"

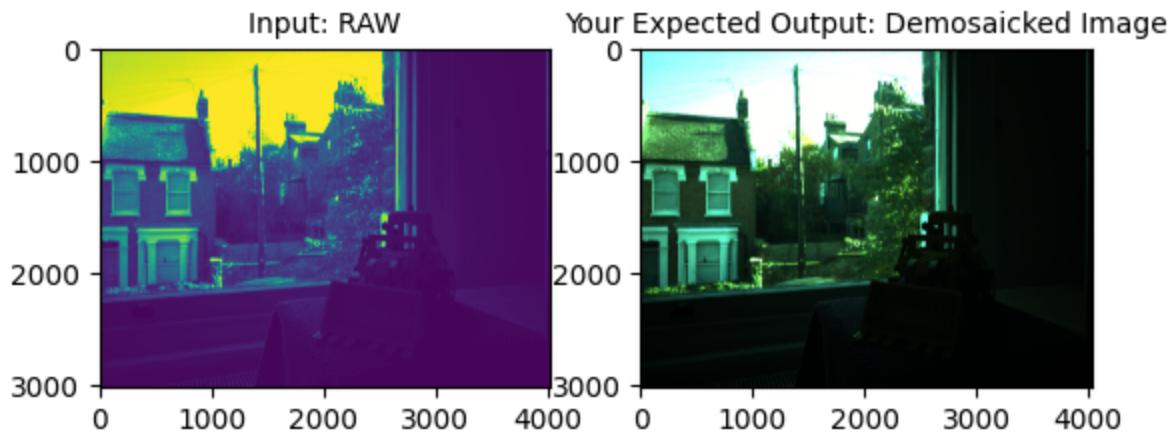
dng = rawpy.imread(dng_file)
with open(meta_file, "r") as json_file:
    meta = json.load(json_file)
raw_img = dng.raw_image # raw is now a numpy array

# Convert to between 0-1
blacklevel = np.array(meta[0]['BlackLevel']).reshape(1, 1)
whitelevel = np.array(meta[0]['WhiteLevel']).reshape(1, 1)
raw_img = (raw_img - blacklevel) / (whitelevel - blacklevel)
```

Let's visualize the input (RAW images) and expected output (Processed images):



Lets start with the first step: demosaicing. The function should take in a mosaicked image in the bayer pattern and return a RGB image. We discussed one way to do this in class: bilinear interpolation. This technique gets values for a pixel by averaging the values of its neighbors. However, the positions of a pixels neighbors are not the same for different channels. We've put the 2x2 pattern (RGGB in this case) for the raw image below. Let's start by visualizing the expected output:



```
In [ ]: def student_demosacking(mosaicked_image):
    """Converts Bayer data into a full RGB image.

    Input data should be ndarray of shape [height, width] with 2x2 mosaic pattern:
    -----
    |red  |green|
    -----
    |green|blue |
    -----"""

    Args:
        bayer_img: [H, W] numpy array. Contains values in the bayer pattern

    Returns:
        rgb_img: [H,W,3] numpy array.

    Potential Implementation:

    [Lecture 2] A potential implementation is bilinear interpolation where each pixel value is averaged from its four neighbors. Where those neighbors are located will vary based on the color you are trying to interpolate. Our code will sanity check bilinear interpolation.

    """
    kernel_G = np.asarray(
        [[0, 1, 0],
         [1, 4, 1],
         [0, 1, 0]]) / 4

    kernel_RB = np.asarray(
        [[1, 2, 1],
         [2, 4, 2],
         [1, 2, 1]]) / 4

    masks = dict((color, np.zeros(mosaicked_image.shape)) for color in 'RGB')
    masks['R'][0::2, 0::2] = 1
    masks['G'][0::2, 1::2] = 1
    masks['G'][1::2, 0::2] = 1
    masks['B'][1::2, 1::2] = 1

    for color in 'RGB':
        masks[color] = masks[color].astype(bool)

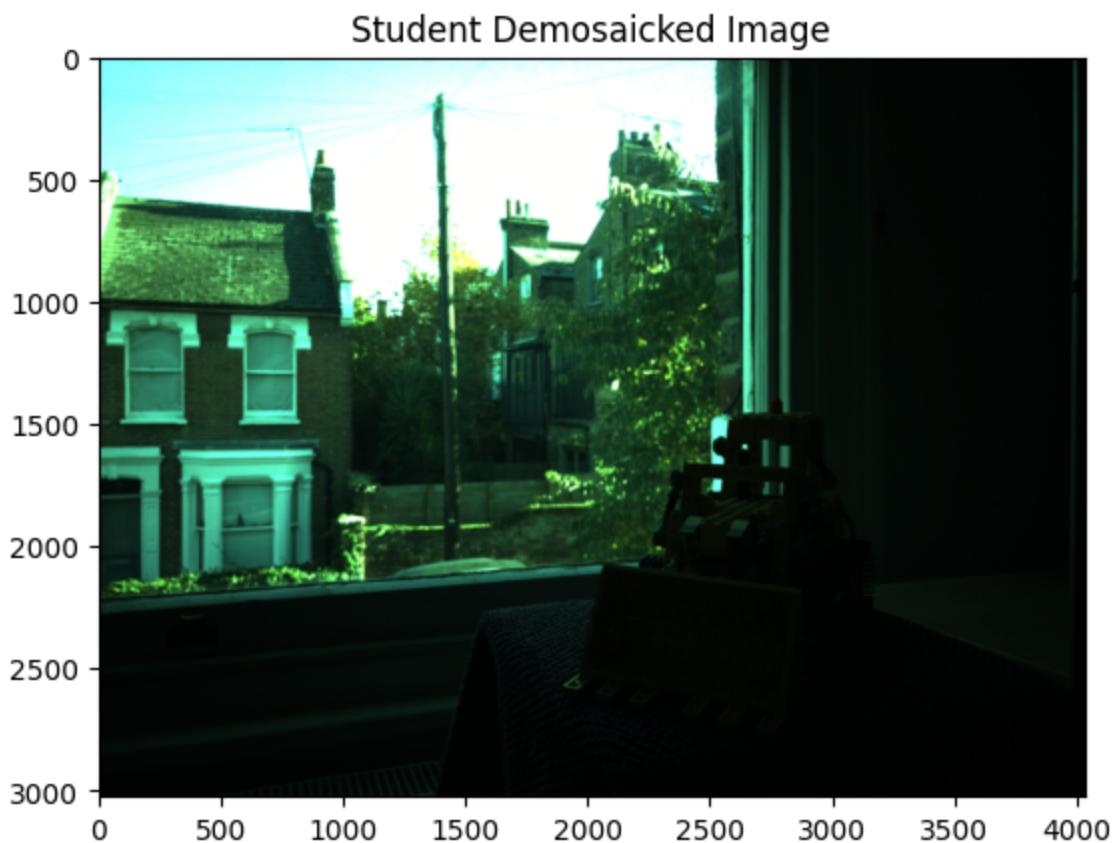
    R = correlate(mosaicked_image * masks['R'], kernel_RB, mode='wrap', cval=0)
    G = correlate(mosaicked_image * masks['G'], kernel_G, mode='wrap', cval=0)
    B = correlate(mosaicked_image * masks['B'], kernel_RB, mode='wrap', cval=0)
    return np.stack([R, G, B], axis=2)
```

```
In [ ]: check_code_correctness(student_demosaicking, raw_img, np.load(rpath+'so  
l/demosaick.npy'))
```

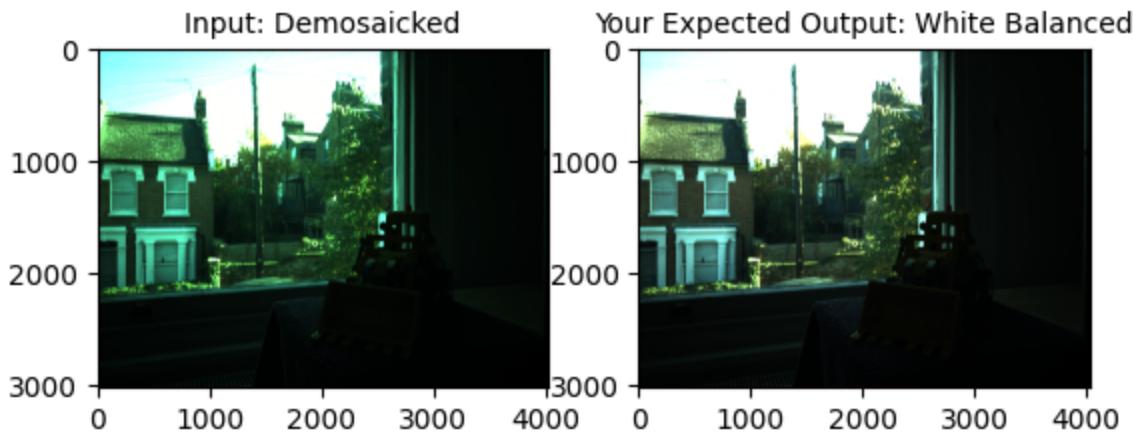
```
Out[ ]: True
```

```
In [ ]: demosaicked_img = student_demosaicking(raw_img)  
demosaicked_img = np.where(demosaicked_img>1.0, 1.0, demosaicked_img)  
plt.imshow(demosaicked_img)  
plt.title("Student Demosaicked Image")  
plt.show()
```

WARNING:matplotlib.image:Clipping input data to the valid range for images with RGB data ([0..1] for floats or [0..255] for integers).



The next step is white balancing. A simple implementation is the gray-world assumption. This is what we will sanity check with the code correctness function. Once again, lets start by visualizing the expected output:



```
In [ ]: def student_white_balancing(demosaickd_image):
    """White balances an image.

    Args:
        demosaickd_image: [H,W,3] numpy array containing RGB values

    Returns:

        white_balanced_img: [H,W,3] numpy array

    Potential Implementation:

        [Lecture 2] Two common implementations are the gray-world assumption
        or the
            white-world assumption. Both can be implemented using a matrix multiplication.
        Our code will sanity check the gray-world assumption.
    """

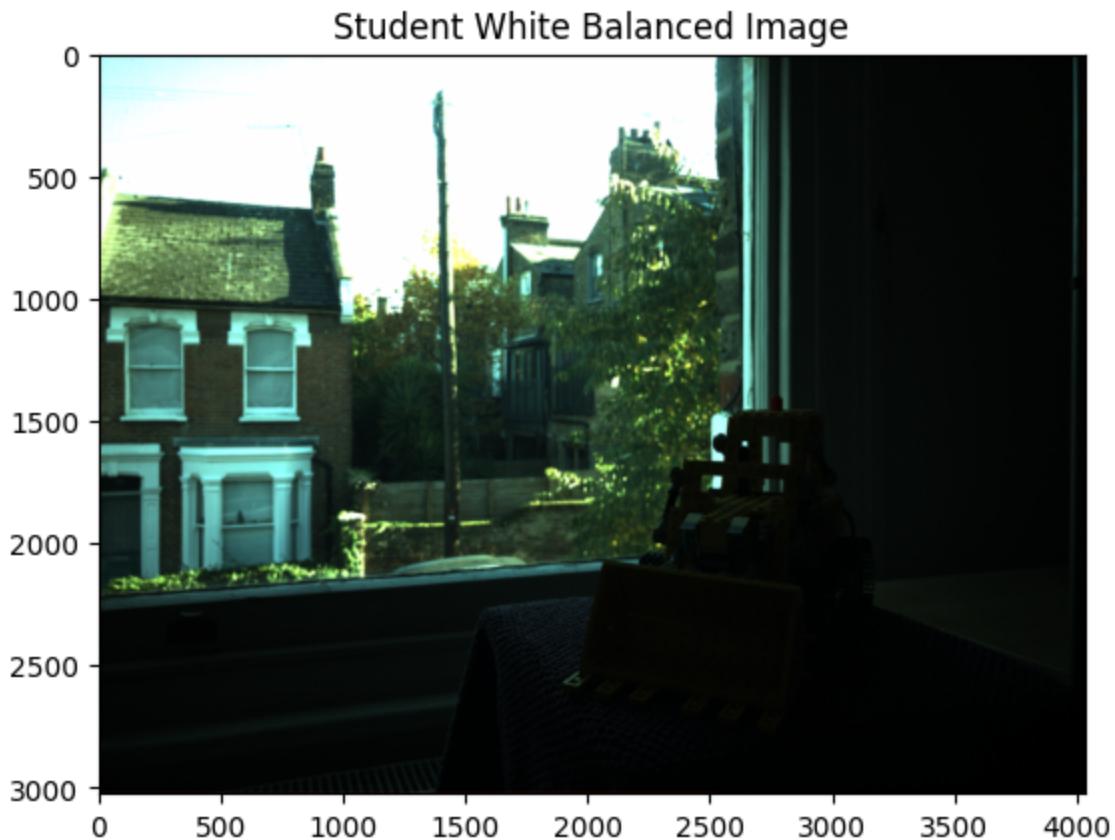
    wb_img = demosaickd_image * (demosaickd_image[:, :, 1].mean() / demosaickd_image.mean(axis=(0, 1)))
    return wb_img
```

```
In [ ]: check_code_correctness(student_white_balancing, demosaickd_img, np.load(rpath+'/sol/white_balance.npy'))
```

```
Out[ ]: True
```

```
In [ ]: white_balanced_img = student_white_balancing(demosaicked_img)
plt.imshow(white_balanced_img)
plt.title("Student White Balanced Image")
plt.show()
```

WARNING:matplotlib.image:Clipping input data to the valid range for images with RGB data ([0..1] for floats or [0..255] for integers).



Next, we have to convert to RGB space. We will provide the individual matrices to convert between spaces, but you will have to choose how to apply them. Let's start by visualizing the expected output image.



```
In [ ]: xyz2cam = np.array([float(x) for x in meta[0]["ColorMatrix2"].split(" ")]) .reshape(3,3) # load from the picture metadata

# Color conversion from reference illuminant XYZ to RGB color space.
# See http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html.
rgb2xyz = np.array([[0.4887180, 0.3106803, 0.2006017],
                    [0.1762044, 0.8129847, 0.0108109],
                    [0.0000000, 0.0102048, 0.9897952]])
```

We've given you two matrices here: one converts from xyz color space to camera color space. The other converts from RGB color space to xyz color space. How can we combine these into a matrix that converts from camera space to RGB space?

```
In [ ]: def student_color_transform(white_balanced_image):
    """Color transform from camera space to rgb

    Args:
        white_balanced_image: [H,W,3] numpy array in camera color space

    Returns:

        rgb_image: [H,W,3] numpy array in RGB color space

    Potential Implementation:

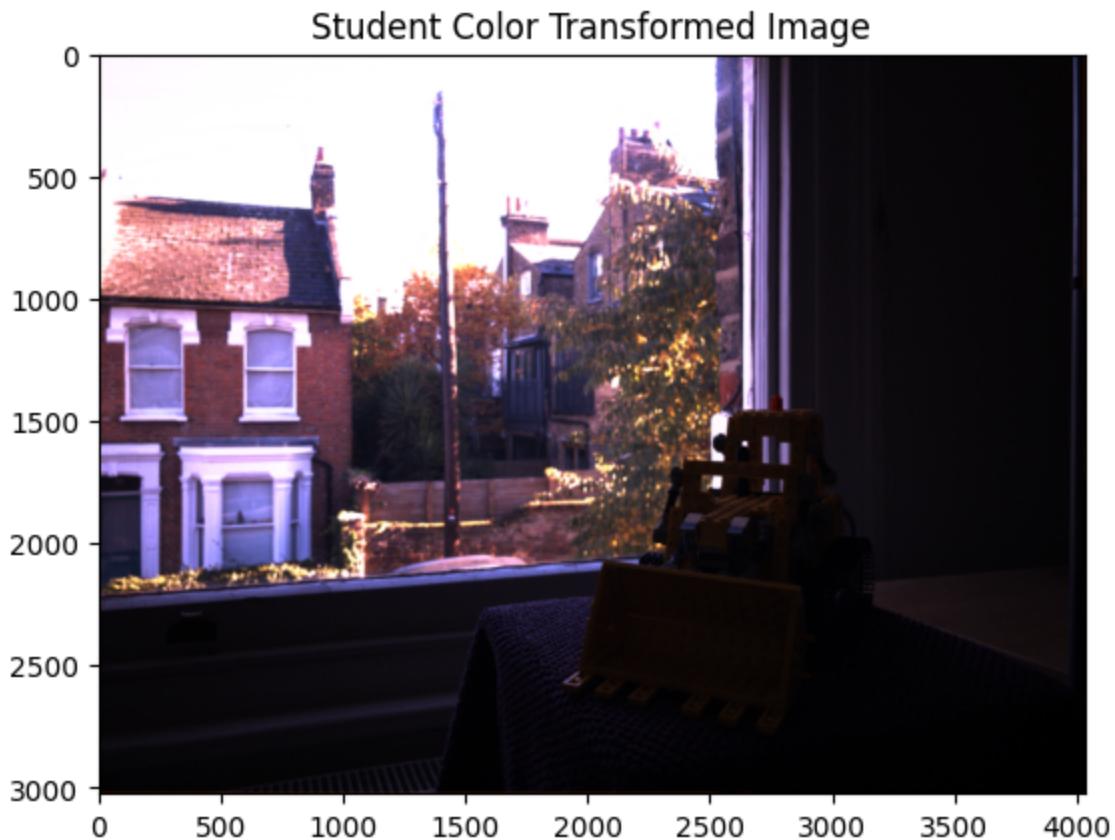
    How would you combine the provided matrices to get the image into RG
    B space?
    """
    white_balanced_image_flat = white_balanced_image.reshape(-1,3).T
    cam2rgb = np.linalg.inv(np.matmul(xyz2cam,rgb2xyz))
    ct_image = np.matmul(cam2rgb,white_balanced_image_flat).T.reshape(white_balanced_image.shape)
    return ct_image
```

```
In [ ]: check_code_correctness(student_color_transform, white_balanced_img, np.load(rpath+'sol/color_transform.npy'))
```

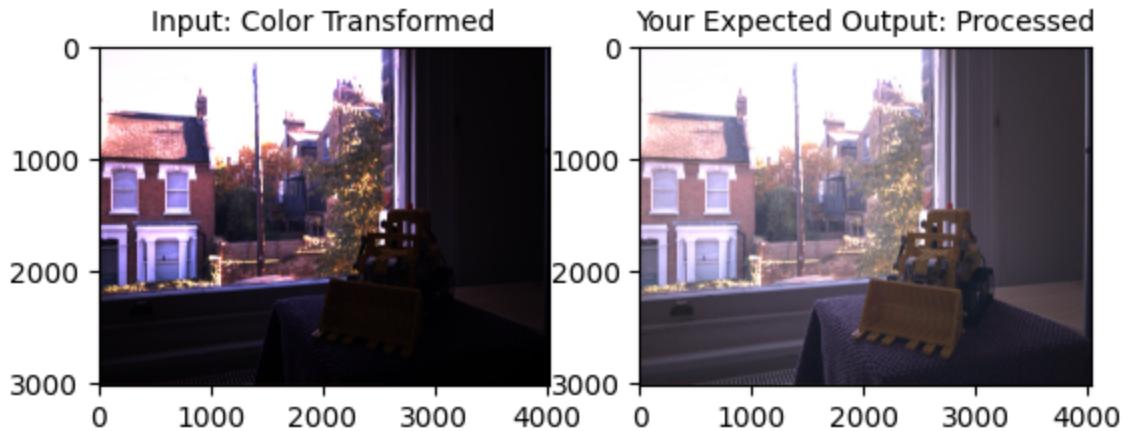
```
Out[ ]: True
```

```
In [ ]: rgb_img = student_color_transform(white_balanced_img)
plt.imshow(rgb_img)
plt.title("Student Color Transformed Image")
plt.show()
```

WARNING:matplotlib.image:Clipping input data to the valid range for images with RGB data ([0..1] for floats or [0..255] for integers).



The last step is gamma encoding. The expected output:

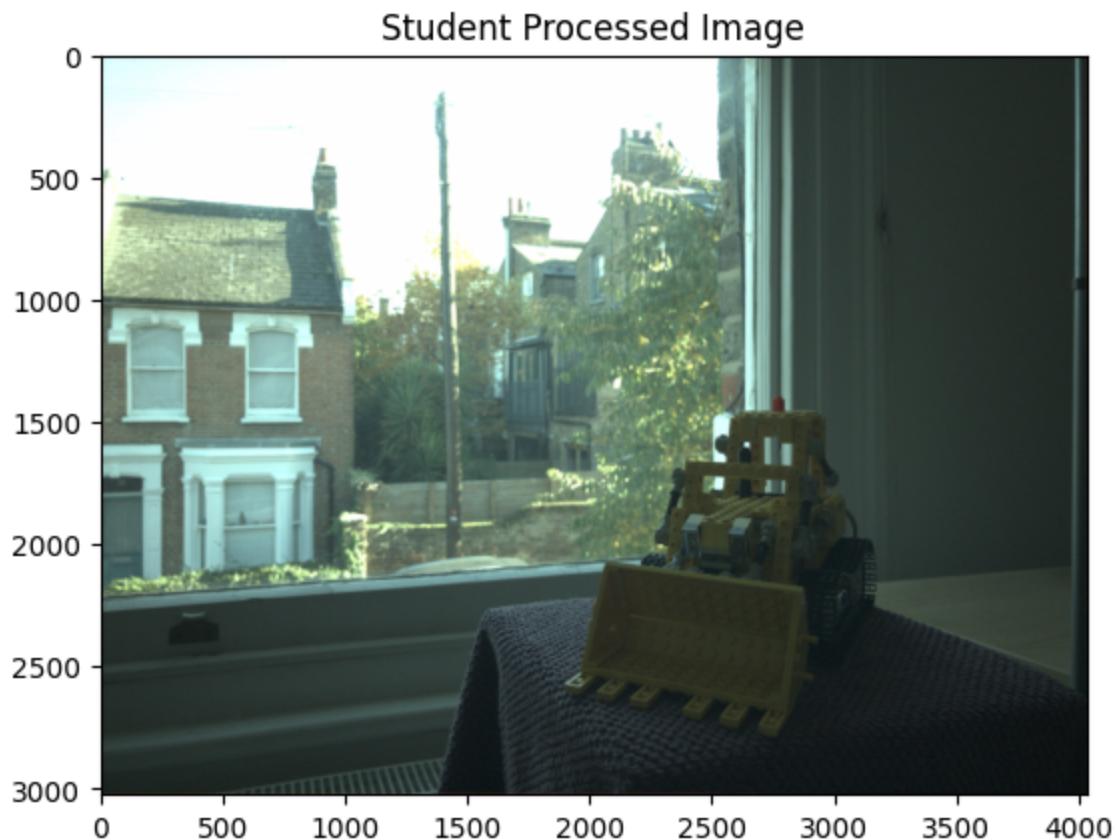


```
In [ ]: def student_gamma_encode(color_transformed_image):  
    """Gamma encodes an image.  
  
    Args:  
        color_transformed_image: [H,W,3] numpy array containing RGB values  
  
    Returns:  
  
        gamma_encoded_img: [H,W,3] numpy array  
  
    Potential Implementation:  
  
        [Lecture 2] For this section, you will need a value for the power. A  
        s discussed  
            in class, a good value is lambda = 1 / 2.2. Try experimenting with d  
        ifferent values.  
            Our code will sanity check with the suggested lambda.  
        """  
        gamma = 1.0/2.2  
        color_transformed_image = np.clip(color_transformed_image,a_min =  
0,a_max =1)  
        gamma_encoded_img = np.power(color_transformed_image, gamma)  
        return gamma_encoded_img
```

```
In [ ]: check_code_correctness(student_gamma_encode, rgb_img, np.load(rpath+ '/s  
ol/processed.npy'))
```

```
Out[ ]: True
```

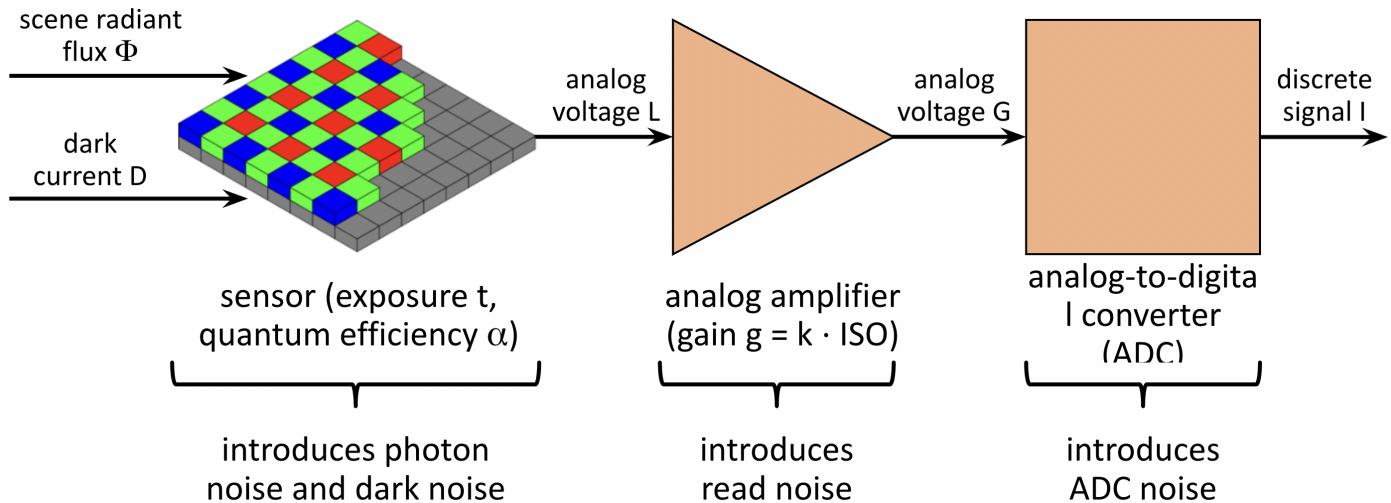
```
In [ ]: processed_image = student_gamma_encode(white_balanced_img)
plt.imshow(processed_image)
plt.title("Student Processed Image")
plt.show()
```



If you tried any new techniques, briefly explain them here:

Problem 2: Denoising

In class, we discussed how the imaging pipeline is not perfect: there are multiple sources of noise including Dark Current, ADC noise, and read noise. In this problem, we will follow the steps in Lecture 3 to estimate these various types of noise. We'll start by estimating the dark current then estimate gain and the ADC noise.



We'll set the parameters of the noise ourselves and then try to estimate them and see how close we get. Let's start by setting up some helper functions:

The input of this section is parameters for various types of noise and our expected outputs are to estimate these parameters with high accuracy.

```
In [5]: # Set parameters:
qe = 1 # Quantum efficiency
D = 0.05 # Dark Current
sigma_read = 0.03 # Std Dev. of read noise
sigma_adc = 0.03 # Std Dev of ADC noise
g = 1 # Gain
```

```
In [6]: def N_photon_detections(t, qe, flux):
    return np.random.poisson(t * qe * flux)

def N_phantom_detections(t, D):
    return np.random.poisson(t * D)

# Total number of detections
# t - exposure time, D - dark current
def N_detections(t, qe, flux, D):
    return np.random.poisson(t * qe * flux + t*D) #multiple ways to do this

def n_read(sigma_read):
    return np.random.normal(scale=sigma_read)

def n_adc(sigma_adc):
    return np.random.normal(scale=sigma_adc)
```

To estimate scene flux, we can load a grayscale image and then use the values as flux. Flux should be a [H,W] array. Try implementing an image with 0 flux, or various levels of gray flux. We'll need them in the estimation process.

```
In [56]: # Returns flux from input image
def image_flux(img_path):
    gray_img = np.array(Image.open(img_path).convert("L"))
    flux = (255 - gray_img) / 255
    return flux

# Returns image with 0 flux everywhere
def zero_flux(size):
    return np.zeros(size) #+ N_detections(t,qe,0,D) + n_read(sigma_read)*g + n_adc(sigma_adc)

# Returns image with gray_level flux everywhere
def gray_level_flux(size, gray_level):
    return (255-np.ones(size)*gray_level)/255
```

Now, lets implement functions to calculate image values and statistics. We've implemented another function as well named "set_vals". It takes in values for D, quantum efficiency, sigma read, and sigma adc and set the global values for these variables. Any values that are not explicitly set are kept the same. You will need to call this function later.

```
In [71]: def set_vals(D_val=None, sigma_read_val=None, sigma_adc_val=None, qe_val=None, g_val=None):
    if D_val is not None:
        global D
        D = D_val
    if qe_val is not None:
        global qe
        qe = qe_val
    if sigma_read_val is not None:
        global sigma_read
        sigma_read = sigma_read_val
    if sigma_adc_val is not None:
        global sigma_adc
        sigma_adc = sigma_adc_val
    if g_val is not None:
        global g
        g = g_val

# Get an image reading given the exposure and flux of the scene
def student_get_I(t, flux):
    return N_detections(t,qe,flux,D)*g + n_read(sigma_read)*g + n_adc(sigma_adc)

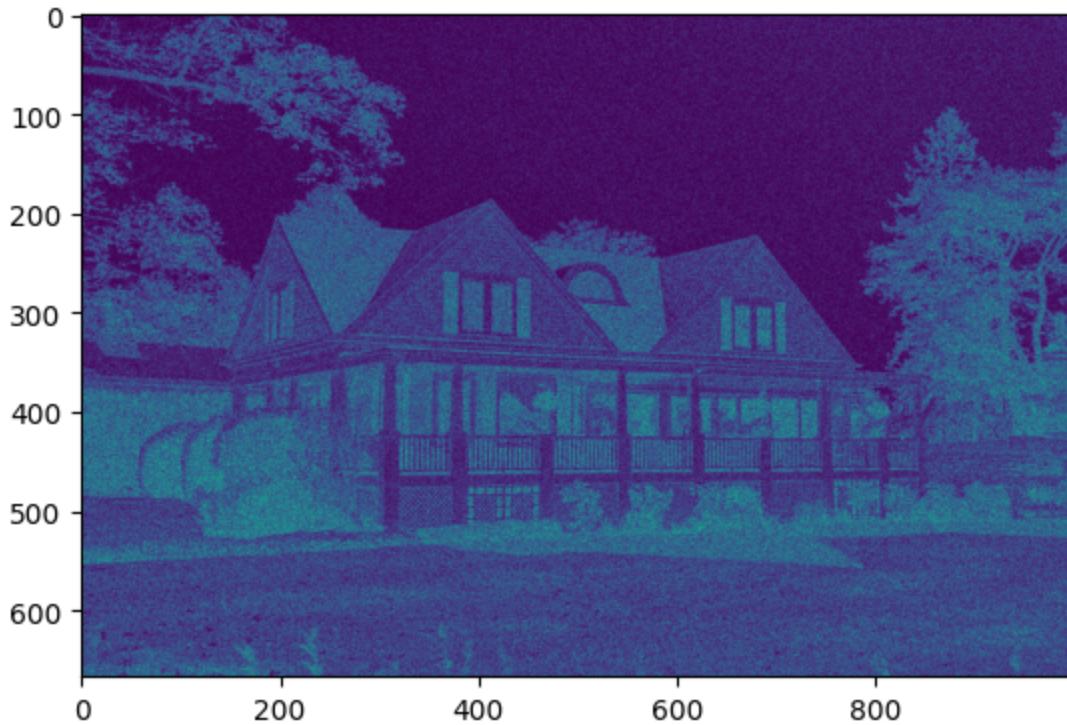
# Estimate mean of image reading given exposure and flux
def student_get_E_I(t, flux):
    M = 10
    EI = np.zeros(flux.shape)
    for i in range(M):
        EI+=(1/M)*student_get_I(t,flux)
    return EI

# Estimate variance of image reading given exposure and flux
def student_get_var_I(t, flux):

    M = 10
    EI = student_get_E_I(t,flux)
    VI = np.zeros(flux.shape)
    for i in range(M):
        VI+=(1/(M-1))*np.square(student_get_I(t,flux)-EI)
    return VI
```

Lets see how our images "I" look:

```
In [ ]: I_hat = student_get_I(10, image_flux(rpath+'/sol/house_img.jpeg')) # Try
          playing with the time to see how it looks
          plt.imshow(I_hat)
          plt.show()
```



Now that we have these helper functions, lets start by estimating dark current. We ideally want to get multiple samples that we can average. The N variable denotes how many samples we should average to get our final number.

```
In [14]: def student_estimate_dark_current(size, N=10):
    """
    Estimates a value for D.
    """
    dark_frame = np.zeros(size)
    for i in range(N):
        dark_frame += (1/N)*student_get_E_I(1, zero_flux(size))
    return dark_frame
```

Let's test this out. Remember the expected value is 0.05.

```
In [16]: print(student_estimate_dark_current((256,256))[0,0]) # should be 0.05
```

0.0515973882842737

```
In [ ]: correct = True
for D_val in [0.01,0.02,0.05,0.06]:
    set_vals(D_val=D_val)
    est_D = student_estimate_dark_current((256,256))
    correct = correct and np.allclose([est_D], [D_val])
if correct:
    print("All tests passed")
else:
    print("Error!")
```

Now lets estimate n_read and n_adc. As we discussed in Lecture 5, we can do this through a multi-step process:

1. Compute mean and variance for varoius gray levels (the number of levels is set by N_levels)
2. Fit a line to the data and use it to estimate g and n_add
3. Use the estimated values of g and n_add to estimate n_read and n_adc.

How do we do step 3? We can leverage the following relationship:

$$n_{\text{add}} = \sqrt{\sigma_{\text{read}}^2 \cdot g^2 + \sigma_{\text{adc}}^2}$$

Approach (i) using polyfit over multiple gains

```
In [49]: from tqdm import tqdm
def student_estimate_read_adc(size, N_levels=5, N=10):
    """
    Estimates n_read and n_adc.

    Potential Implementation based on Lecture 5:
    1. Compute mean and variance for various gray levels (with dark current correction)
    2. Fit a line to the data and estimate parameters

    Returns:
        n_read: estimated n_read
        n_adc: estimated n_adc
    """
    gray_levels = [50, 90, 130, 170, 210]
    gs = [0.1, 0.3, 0.5, 0.7, 1, 2]
    data_points_count = N_levels * N #* size[0]*size[1]
    gain_square, var_add = [None]*len(gs), [None]*len(gs)
    for i,g in enumerate(tqdm(gs)):
        set_vals(sigma_adc_val=0.02, sigma_read_val=0.05, g_val=g)
        mean, var = np.array([None]*data_points_count), np.array([None]*data_points_count)
        c=0
        for _ in tqdm(range(N)):
            for gray_level in gray_levels:
                mean[c] = (student_get_E_I(1,gray_level_flux(size,gray_level)) - student_estimate_dark_current(size)).flatten()
                var[c] = (student_get_var_I(1,gray_level_flux(size,gray_level)) - student_estimate_dark_current(size)).flatten()
            c+=1

        mean , var = np.hstack(mean) , np.hstack(var)
        z = np.polyfit(mean,var,1)
        gain_square[i] , var_add[i] = z[0]**2 , z[1]

    trend = np.polyfit(gain_square, var_add,1)
    var_adc, var_read = trend[1], trend[0]
    return var_adc, var_read
```

Approach (ii) Using 2 gain values

```
In [97]: from tqdm import tqdm
def student_estimate_read_adc(size, N_levels=5, N=10):
    """
    Estimates n_read and n_adc.

    Potential Implementation based on Lecture 5:
    1. Compute mean and variance for various gray levels (with dark current correction)
    2. Fit a line to the data and estimate parameters

    Returns:
        n_read: estimated n_read
        n_adc: estimated n_adc
    """
    gray_levels = [50, 90, 130, 170, 210]
    data_points_count = N_levels
    set_vals(g_val=1)
    mean, var = np.array([None]*data_points_count), np.array([None]*data_
points_count)
    c=0
    for gray_level in tqdm(gray_levels):
        mean_gl = [None]*N
        var_gl = [None]*N
        for j in tqdm(range(N)):
            mean_gl[j] = (student_get_E_I(1,gray_level_flux(size,gray_leve
l))).flatten()
            var_gl[j] = (student_get_var_I(1,gray_level_flux(size,gray_leve
l))).flatten()
        mean[c], var[c] = np.mean(mean_gl, axis=0), np.mean(var_gl, axis=
0)
        c+=1
    mean, var = np.hstack(mean), np.hstack(var)
    z = np.polyfit(mean, var, 1)
    gain1, var_add1 = z[0], z[1]

    #####
    set_vals(g_val=10)
    mean, var = np.array([None]*data_points_count), np.array([None]*data_
points_count)
    c=0
    for gray_level in tqdm(gray_levels):
        mean_gl = [None]*N
        var_gl = [None]*N
        for j in tqdm(range(N)):
            mean_gl[j] = (student_get_E_I(1,gray_level_flux(size,gray_leve
l))).flatten()
            var_gl[j] = (student_get_var_I(1,gray_level_flux(size,gray_leve
l))).flatten()

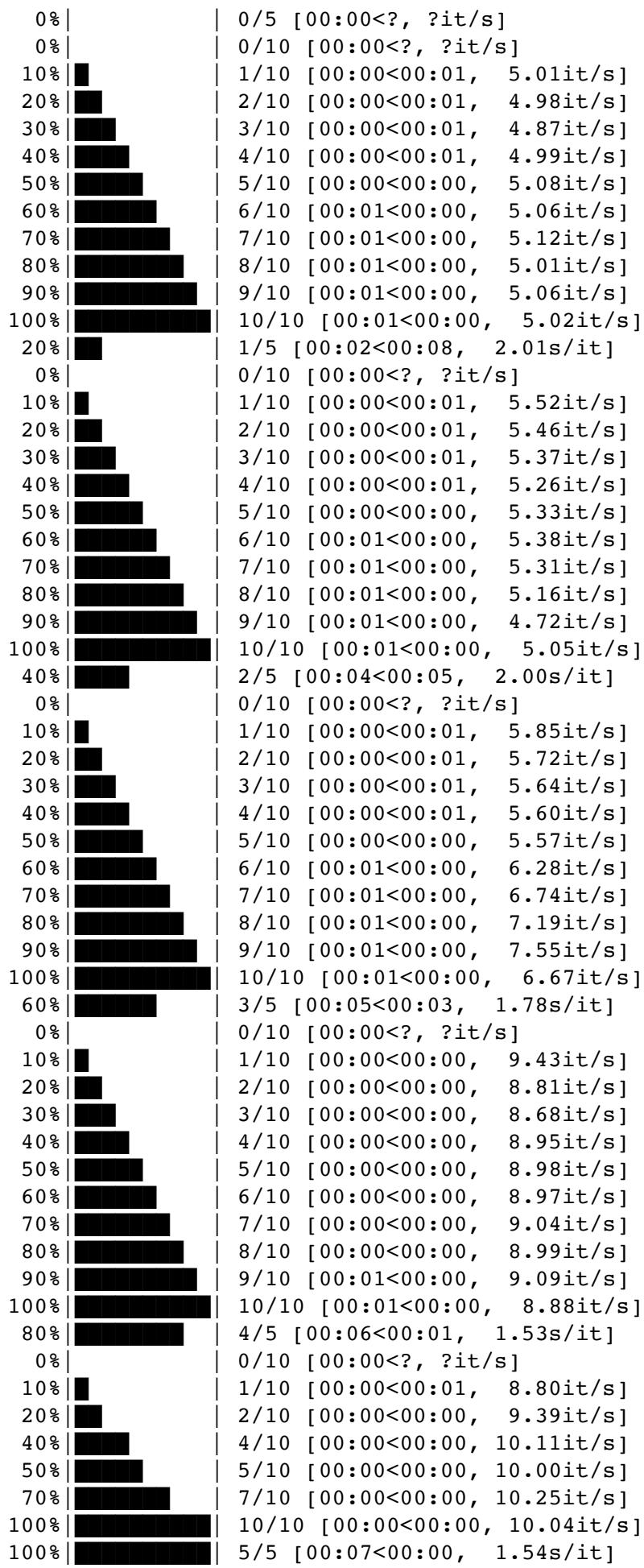
        mean[c], var[c] = np.mean(mean_gl, axis=0), np.mean(var_gl, axis=
0)
```

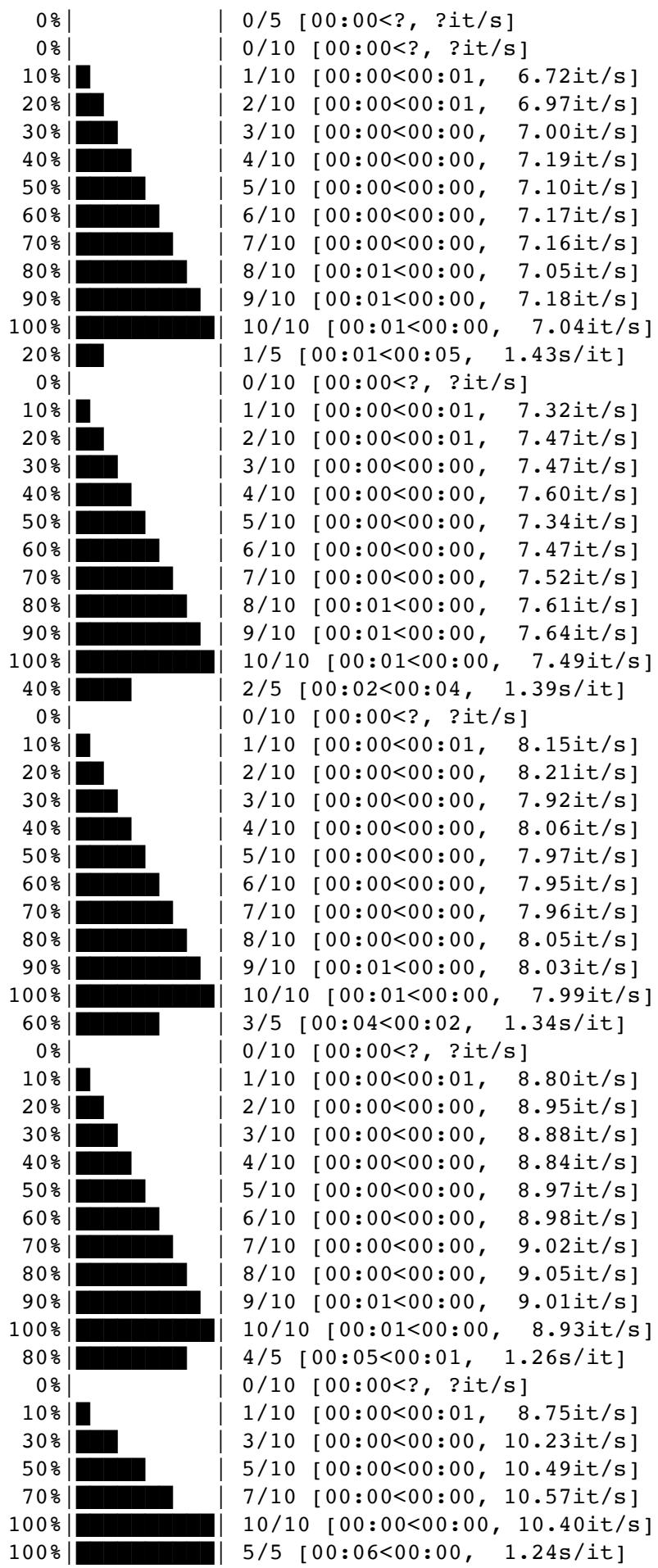
```
c+=1
mean , var = np.hstack(mean) , np.hstack(var)
z = np.polyfit(mean,var,1)
gain2 , var_add2 = z[0] , z[1]

# var_read = (var_add1 - var_add2)/(gain1**2 - gain2**2)
# var_adc = var_add1 - (gain1**2)*var_read
gain_square = [gain1**2, gain2**2]
var_add = [var_add1, var_add2]
trend = np.polyfit(gain_square, var_add,1)
var_adc, var_read = trend[1], trend[0]

return var_read, var_adc, gain1, gain2
```

```
In [98]: var_read, var_adc, gain1, gain2 = student_estimate_read_adc((256,256))
```





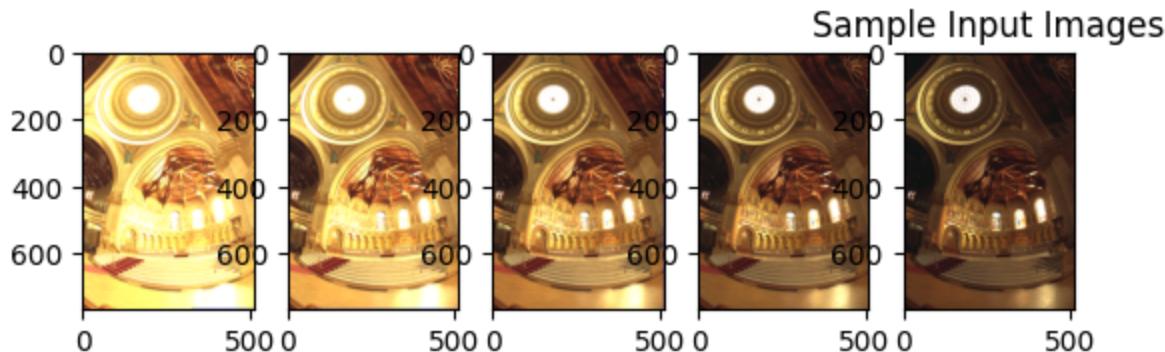
```
In [ ]: correct = True
set_vals(D_val=0) # We've estimated it now and can subtract from input images
for n_adc, n_read in zip([0.01,0.02,0.05,0.06], [0.01, 0.05, 0.02, 0.01]):
    set_vals(sigma_adc_val=n_adc, sigma_read_val=n_read)
    est_n_read, est_n_adc = student_estimate_read_adc((256,256))
    correct = correct and np.allclose([est_n_read, est_n_adc], [n_read, n_adc])
if correct:
    print("All tests passed")
else:
    print("Error!")
```

If all tests are passing, congrats! You implemented a system to estimate the various noises in the imaging pipeline.

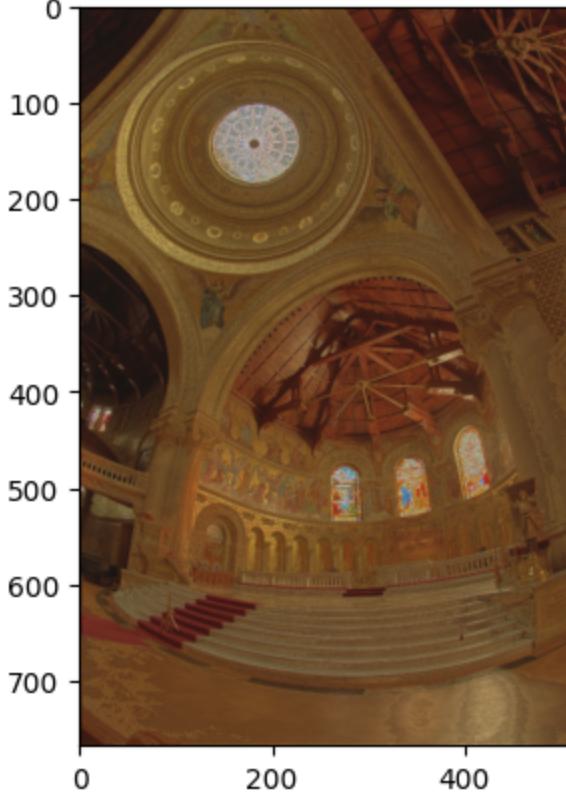
Problem 3: HDR Imaging

In this problem, we will implement exposure bracketing for HDR reconstruction. We have access to a set of images taken at various exposure levels and we want to output a single HDR picture that better represents the scene.

Let's start by visualizing the expected output and input:



Your Expected Output: HDR Image

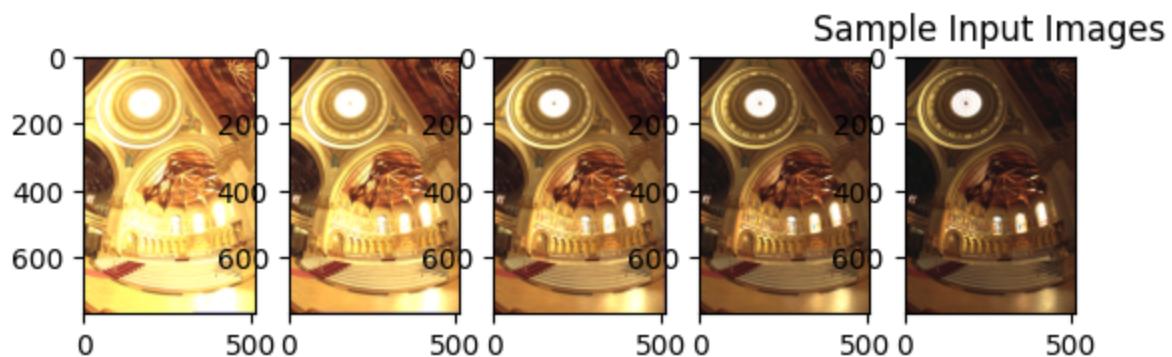


Lets load images in and reading exposure levels. We've written a function below that loads in the shutter speed, f-stop, and gain of the images.

```
In [18]: data_file = rpath+"/hdr/memorial.hdr_image_list.txt"
num_imgs = 0
ldr_imgs = []

with open(data_file, "r") as img_list:
    for line in img_list:
        if line[0] == "#":
            continue
        vals = line.split()
        if len(vals) == 1:
            num_imgs = int(vals[0])
        else:
            fname, inv_shutter_speed, f_stop, gain, nd_filters = vals
            inv_shutter_speed = float(inv_shutter_speed)
            f_stop = int(f_stop)
            gain = int(gain)
            I = np.array(Image.open(rpath+'/hdr/'+fname.replace('.ppm',
'.png')))
            ldr_imgs.append((I, inv_shutter_speed, f_stop, gain)) # Which one of these measures exposure time?
```

```
In [ ]: # Visualize the first 5 images
fig, ax = plt.subplots(1,5)
for i in range(5):
    ax[i].imshow(ldr_imgs[i][0])
plt.title("Sample Input Images")
plt.show()
```



Now that we have the images, lets put them together following the exposure bracketing technique discussed in class. In this technique, we take pixel values from images with varying exposure, weight them by both pixel values and exposure values (see Lecture 5 for details) and then combine them to get an HDR image.

A key question you will have to answer is how to convert the data we have into exposure values we can use for weighting. As a hint, exposure values are generally integer numbers with 0 being the "middle".

```
In [19]: def student_hdr_reconstruct(imgs):
    """
        Recovers a HDR image from multiple LDR images.

    Args:
        imgs: a list of tuples containing: LDR image ([H,W,3]), 1/shutter speed, f-stop, gain

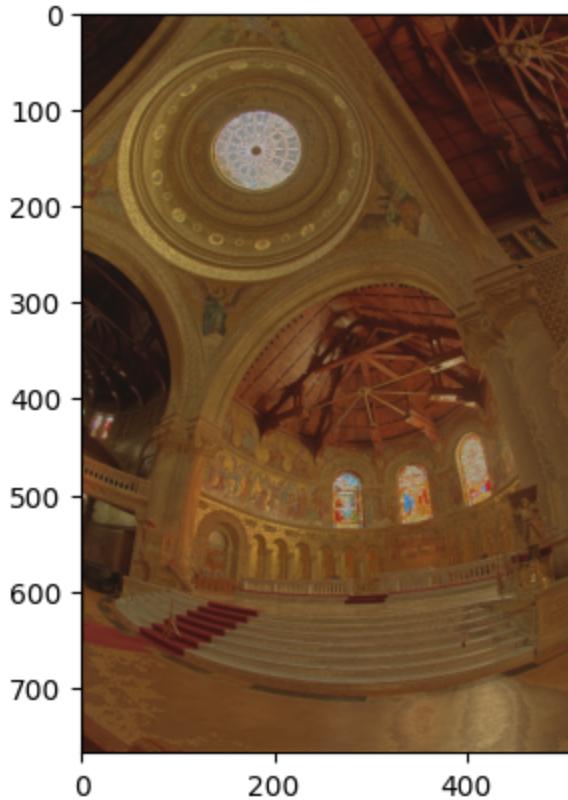
    Returns:
        hdr_img: a [H,W,3] HDR image
    """
    tmin = 0.05*255.0
    tmax = 0.95*255.0

    weights=np.array([None]*len(imgs))
    hdr_img = np.zeros_like(imgs[0][0]).astype(float)
    for i,img in enumerate(imgs):
        img, inv_shutter_speed = img[0], img[1]
        img = img.astype(float)
        weights[i] = np.zeros_like(img)
        valid_idx = (img[:, :, 0]>tmin) & (img[:, :, 0]<tmax) & (img[:, :, 1]>tmin) & (img[:, :, 1]<tmax) & (img[:, :, 2]>tmin) & (img[:, :, 2]<tmax)
        valid_idx = np.repeat(valid_idx[...],None),3, axis=2)
        np.putmask(weights[i],valid_idx, img/(np.abs(np.log2(inv_shutter_speed)) + 1))
        hdr_img = hdr_img + weights[i]*img

    hdr_img = hdr_img/weights.sum(axis=0)
    hdr_img = hdr_img/255.0
return hdr_img
```

```
In [20]: hdr_img = student_hdr_reconstruct(ldr_imgs)
plt.imshow(hdr_img)
```

Out[20]: <matplotlib.image.AxesImage at 0x79add83bbca0>



```
In [21]: check_code_correctness(student_hdr_reconstruct, ldr_imgs, np.load('/content/drive/MyDrive/239AS/Assignment-1/sol/hdr.npy'))
```

Out[21]: True

If you used techniques not discussed in class, please briefly summarize them here:

References

Many of the materials used here were compiled from various different academic papers and datasets:

Raw Dataset:

NeRF in the Dark: High Dynamic Range View Synthesis from Noisy Raw Images, Ben Mildenhall and Peter Hedman and Ricardo Martin-Brualla and Pratul P. Srinivasan and Jonathan T. Barron, CVPR (2022)

HDR dataset:

Recovering High Dynamic Range Radiance Maps from Photographs., Paul E. Debevec and Jitendra Malik, SIGGRAPH 97, August 1997

Further Reading:

Chapter 2, *Computational Imaging*, Ayush Bhandari, Achuta Kadambi, and Ramesh Raskar, MIT Press, 2022.