Name (netid): Rahul Grover (rgrover4)
CS 445 - Project 4: Image Based Lighting

Total Points Claimed	**[70] / 210**
Core	
1. Recovering HDR maps	
a. Data collection	[0] / 20
b. Naive HDR merging	[10] / 10
c. Weighted HDR merging	[15] / 15
d. Calibrated HDR merging	[15] / 15
e. Additional HDR questions	[10] / 10
2. Panoramic transformations	[10] / 10
3. Rendering synthetic objects	[0] / 30
4. Quality of results / report	[10] / 10
B&W	
5. Additional results	[0] / 20
6. Other transformations	[0] / 20
7. Photographer & Tripod removal	[0] / 25
8. Local tone-mapping operator	[0] / 25

1. Recovering HDR maps

Figure of rescaled log irradiance images from naive method

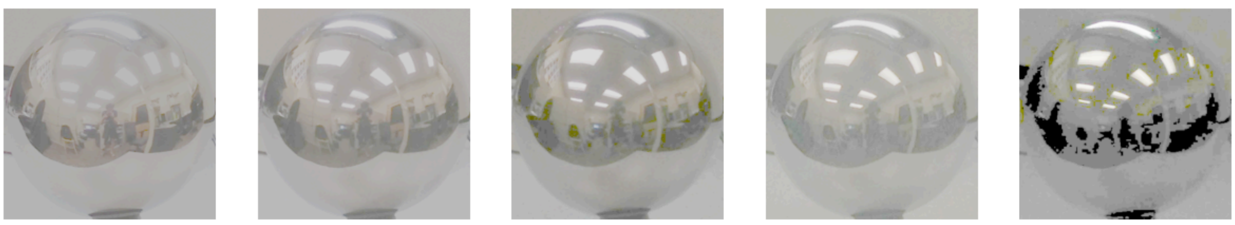
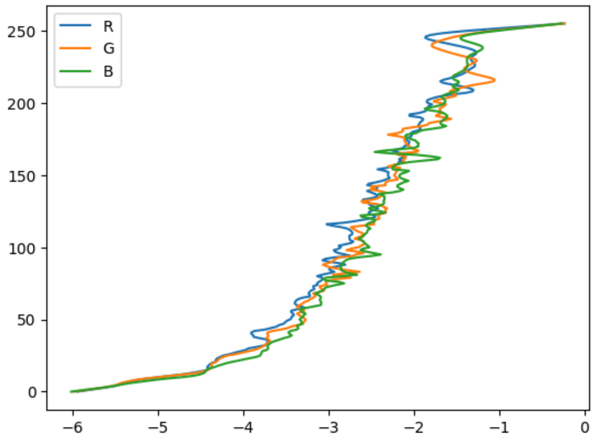


Figure of rescaled log irradiance images from weighted method

Figure of rescaled log irradiance images from calibration method

Plots of g vs intensity and intensity vs g



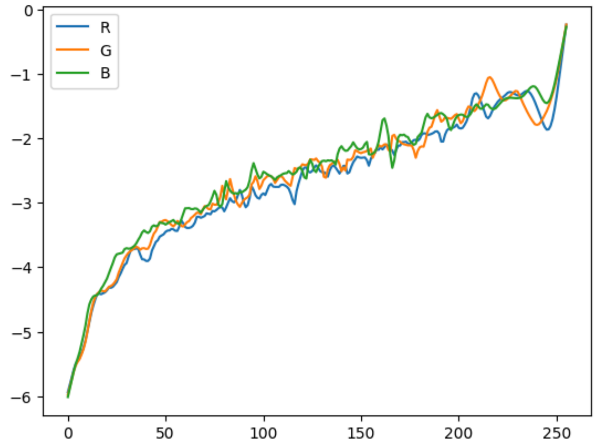


Figure comparing the three HDR methods



Text output comparing the dynamic range and RMS error consistency of the three methods

```
naive:      log range =  6.46     avg RMS error =  0.324
weighted:   log range =  6.622    avg RMS error =  0.286
calibrated: log range =  7.016    avg RMS error =  0.251
```


```
naive:       log range =  6.46     avg RMS error =  0.324
weighted:    log range =  6.622    avg RMS error =  0.286
calibrated:  log range =  7.016    avg RMS error =  0.251
```




Answers to the questions below

1. *For a very bright scene point, will the naive method tend to over-estimate the true brightness, or under-estimate? Why?*

   For a very bright scene point, the naive method will tend to overestimate the true brightness. This is because when there is a higher exposure rate, more light is captured by the camera and the pixel values seem more saturated as a result. This leads to an inaccurate brightness level for the pixels within the scene when dividing by the exposure rate in the naive method.

2. *Why does the weighting method result in a higher dynamic range than the naive method?*

   Unlike the naive method, the weighting method prioritizes the pixel intensities that are closer to 128 and puts less “weight” on pixels close to 0 or 255. It does this by using a weighting function that maps these pixels to get the weighted irradiance which is used to emphasize the weight of the pixels that are mapped closer to the middle range of 0-255.

3. *Why does the calibration method result in a higher dynamic range than the weighting method?*

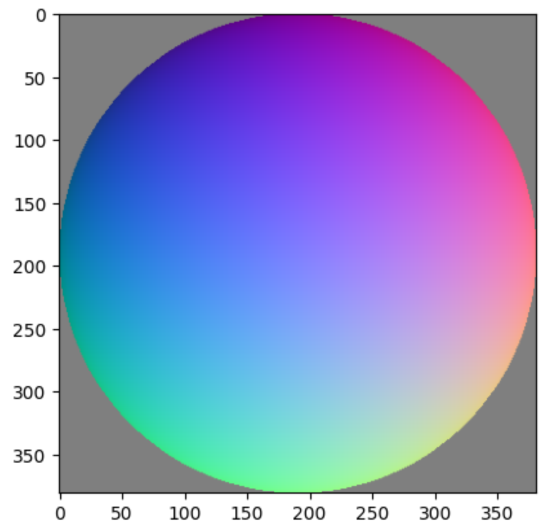
   The calibration method results in a higher dynamic range than the weighted method because in addition to the weighting function in the weighting method, we also estimate the inverse log response function g(Z) before assigning the weights to each pixel. This is important because most cameras compress very high or low intensity values into a smaller range of intensity values and the response function allows us to find the true intensity values from this smaller range.

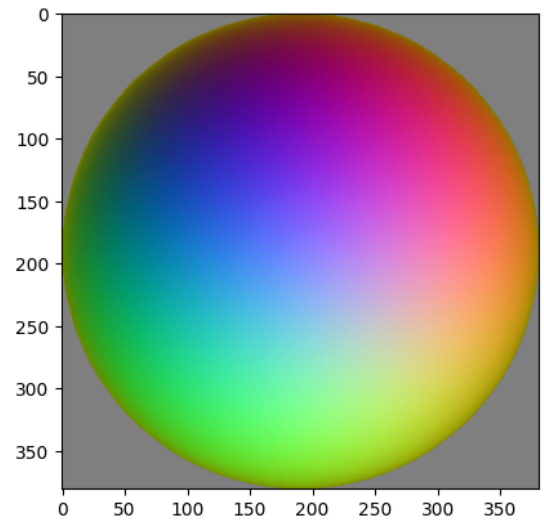
4. *Why does the calibration method result in higher consistency, compared to the weighting method?*

   The calibration method results in higher consistency compared to the weighted method since it attempts to estimate the response function before assigning weights to the intensity values. This is important because images with very high or low intensity values are mapped to a smaller range automatically by the camera, and this inverse function allows us to find the true intensity which more accurately represents the intensity of the pixels at a given exposure. The weighting function assumes a linear relationship with the intensity and exposure but in practice, intensity is typically a non-linear function of exposure which this method takes into account.

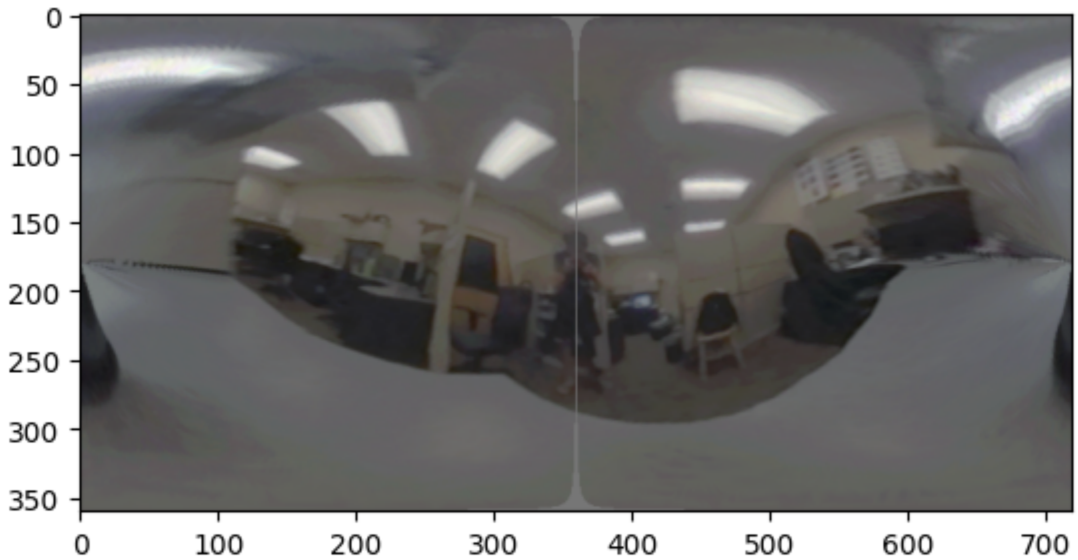
2. Panoramic transformations

The images of normal vectors and reflectance vectors





The equirectangular image from your calibration HDR result



4. Quality of results / report

Acknowledgments / Attribution

None unless the provided images count, the rest of the images involved were my own or blender's provided models

CS445: Computational Photography

Programming Project 4: Image-Based Lighting

Recovering HDR Radiance Maps

Load libraries and data

```
# jupyter extension that allows reloading functions from imports
without clearing kernel :D
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```
# System imports
from os import path
import math

# Third-Party Imports
import cv2
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata

# modify to where you store your project data including utils
datadir = "/content/drive/My Drive/ImageBasedLighting/proj4/"

utilfn = datadir + "utils"
!cp -r "$utilfn" .
samplesfn = datadir + "samples"
!cp -r "$samplesfn" .

# can change this to your output directory of choice
!mkdir "images"
!mkdir "images/outputs"

# import starter code
import utils
from utils.io import read_image, write_image, read_hdr_image,
```

```
write_hdr_image
from utils.display import display_images_linear_rescale,
rescale_images_linear
from utils.hdr_helpers import gsolve
from utils.hdr_helpers import get_equirectangular_image
from utils.bilateral_filter import bilateral_filter
```

```
mkdir: cannot create directory ‘images’: File exists
mkdir: cannot create directory ‘images/outputs’: File exists
```

Reading LDR images

You can use the provided samples or your own images. You get more points for using your own images, but it might help to get things working first with the provided samples.

```
# TODO: Replace this with your path and files

imdir = 'samples'
imfns = ['0024.jpg', '0060.jpg', '0120.jpg', '0205.jpg', '0553.jpg']
exposure_times = [1/24.0, 1/60.0, 1/120.0, 1/205.0, 1/553.0]

ldr_images = []
for f in np.arange(len(imfns)):
  im = read_image(imdir + '/' + imfns[f])
  if f==0:
    imsize = int((im.shape[0] + im.shape[1])/2) # set width/height of
ball images
    ldr_images = np.zeros((len(imfns), imsize, imsize, 3))
  ldr_images[f] = cv2.resize(im, (imsize, imsize))

background_image_file = imdir + '/' + 'empty.jpg'
background_image = read_image(background_image_file)
```

Naive LDR merging

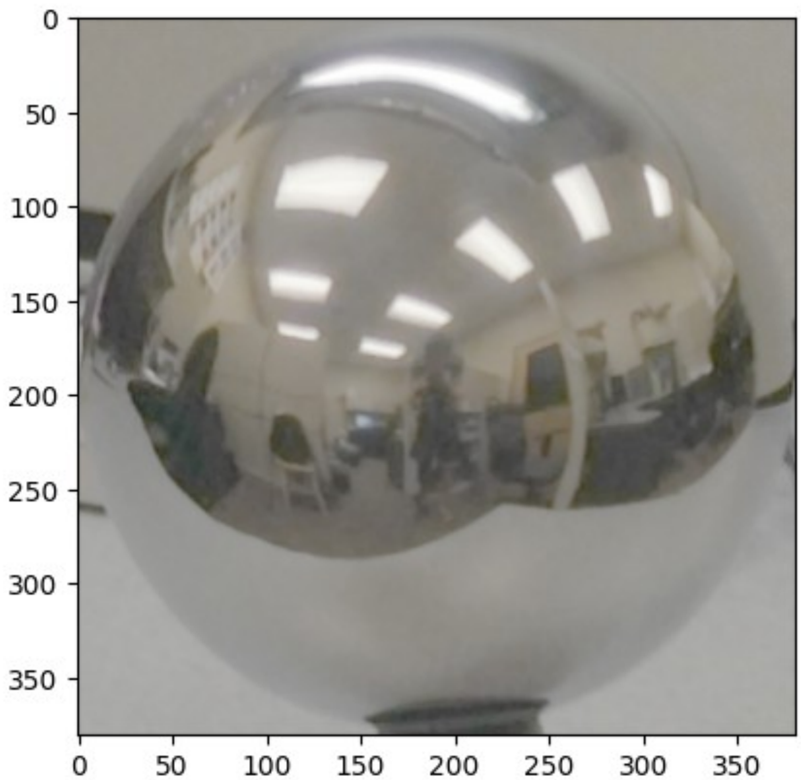
Compute the HDR image as average of irradiance estimates from LDR images

```
def make_hdr_naive(ldr_images: np.ndarray, exposures: list) ->
(np.ndarray, np.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding
exposure values.

    The steps to implement:
    1) Divide each image by its exposure time.
        - This will rescale images as if it has been exposed for 1
second.
```

```
    2) Return average of above images


    For further explanation, please refer to problem page for how to
do it.

    Args:
        ldr_images(np.ndarray): N x H x W x 3  shaped numpy array
representing
            N ldr images with width W, height H, and channel size of 3
(RGB)
        exposures(list): list of length N, representing exposures of
each images.
            Each exposure should correspond to LDR images' exposure
value.
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR
image merged using
            naive ldr merging implementation.
        (np.ndarray): N x H x W x 3  shaped numpy array represending
log irradiances
            for each exposures

    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposures)

    # Scale to 255, and create output arrays as per comment above
    ldr_images_copy = ldr_images.copy()
    ldr_images_copy *= 255
    hdr_image = np.ones((H, W, C))
    log_irradiances = np.zeros((N, H, W, C))

    # Loop through each image and corresponding exposure to get data
    for i in range(N):
      hdr_image += ldr_images_copy[i]/exposures[i]
      log_irradiances[i] = np.log1p(ldr_images_copy[i]/exposures[i])

    # Average out by number of images (N)
    hdr_image /= N

    return hdr_image, log_irradiances


def display_hdr_image(im_hdr):
    '''
    Maps the HDR intensities into a 0 to 1 range and then displays.
    Three suggestions to try:
```

```
      (1) Take log and then linearly map to 0 to 1 range (see
display.py for example)
      (2) img_out = im_hdr / (1 + im_hdr)
      (3) HDR display code in a python package
    '''

    # Take log, and then use function in display.py
    clampedImage = np.log(im_hdr)
    clampedImage = rescale_images_linear(clampedImage)
    plt.figure()
    plt.imshow(clampedImage)
    return clampedImage


# get HDR image, log irradiance
naive_hdr_image, naive_log_irradiances = make_hdr_naive(ldr_images,
exposure_times)

# write HDR image to directory
write_hdr_image(naive_hdr_image, 'images/outputs/naive_hdr.hdr')

# display HDR image
print('HDR Image')
display_hdr_image(naive_hdr_image)

# display original images (code provided in utils.display)
display_images_linear_rescale(ldr_images)

# display log irradiance image (code provided in utils.display)
display_images_linear_rescale(naive_log_irradiances)
```

```
HDR Image
```



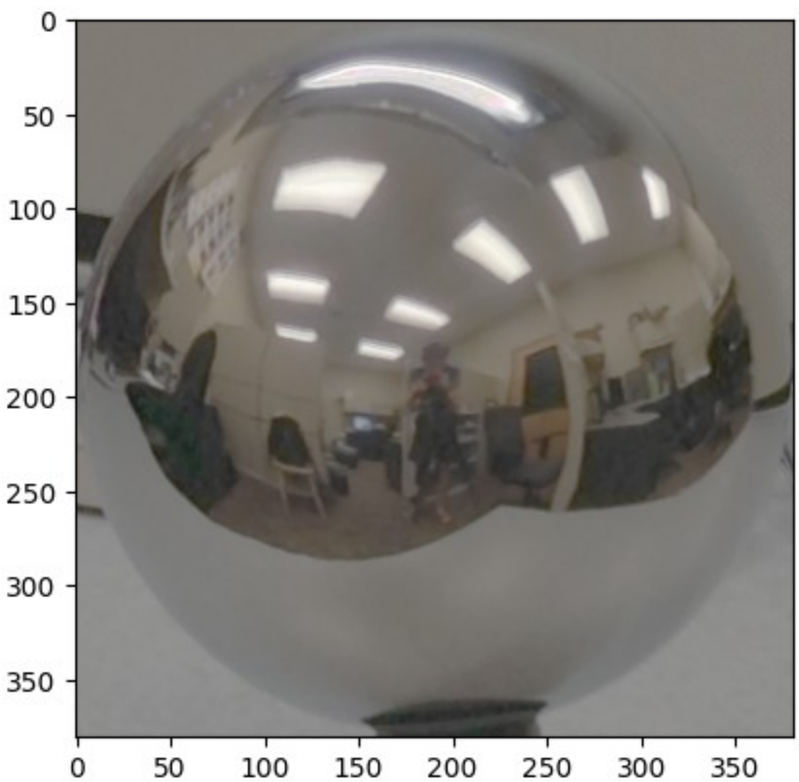
Weighted LDR merging

Compute HDR image as a weighted average of irradiance estimates from LDR images, where weight is based on pixel intensity so that very low/high intensities get less weight

```
def make_hdr_weighted(ldr_images: np.ndarray, exposure_times: list) ->
(np.ndarray, np.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding
exposure values.
```

```
    The steps to implement:
    1) compute weights for images with based on intensities for each
exposures
        - This can be a binary mask to exclude low / high intensity
values

    2) Divide each images by its exposure time.
        - This will rescale images as if it has been exposed for 1
second.

    3) Return weighted average of above images


    Args:
        ldr_images(np.ndarray): N x H x W x 3 shaped numpy array
representing
            N ldr images with width W, height H, and channel size of 3
(RGB)
        exposure_times(list): list of length N, representing exposures
of each images.
            Each exposure should correspond to LDR images' exposure
value.
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR
image merged without
            under - over exposed regions

    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposure_times)

    # Scale to 255 and initialize arrays
    ldr_images_copy = ldr_images.copy()
    ldr_images_copy *= 255
    hdr_image = np.zeros((H, W, C))
    weighted_intensities = np.zeros((H, W, C))
    log_irradiances = np.zeros((N, H, W, C))

    # Given lambda function to keep intensities of 128 near 1
    w = lambda z: (128-np.abs(z-128))

    # Loop through pixel intensities and get
    for i in range(N):
      # Pass through lambda to clamp intensities, and divide by
exposure time
      weighted_irradiance = w(ldr_images_copy[i])
      log_irradiances[i] = np.log1p(weighted_irradiance /
exposure_times[i])
```

```
      hdr_image += weighted_irradiance * (ldr_images_copy[i] /
exposure_times[i])
      # Compute weighted sum to divide image
      weighted_intensities += weighted_irradiance

    hdr_image /= weighted_intensities

    return hdr_image, log_irradiances

# get HDR image, log irradiance
weighted_hdr_image, weighted_log_irradiances =
make_hdr_weighted(ldr_images, exposure_times)

# write HDR image to directory
write_hdr_image(weighted_hdr_image, 'images/outputs/weighted_hdr.hdr')

# display HDR image
display_hdr_image(weighted_hdr_image)

display_images_linear_rescale(weighted_log_irradiances)
```



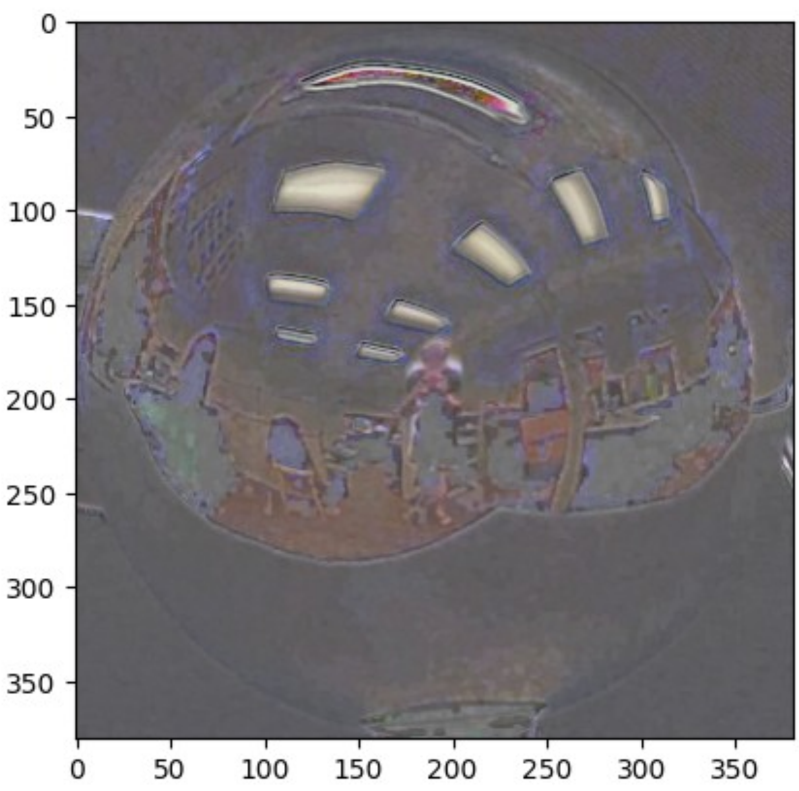
Display of difference between naive and weighted for your own inspection

Where does the weighting make a big difference increasing or decreasing the irradiance estimate? Think about why.

```
# display difference between naive and weighted

log_diff_im = np.log(weighted_hdr_image)-np.log(naive_hdr_image)
print('Min ratio = ', np.exp(log_diff_im).min(), '  Max ratio = ',
np.exp(log_diff_im).max())
plt.figure()
plt.imshow(rescale_images_linear(log_diff_im))
```

```
Min ratio =  0.576763930017864   Max ratio =  2.912049548208497
```

```
<matplotlib.image.AxesImage at 0x7a328f4ae350>
```



LDR merging with camera response function estimation

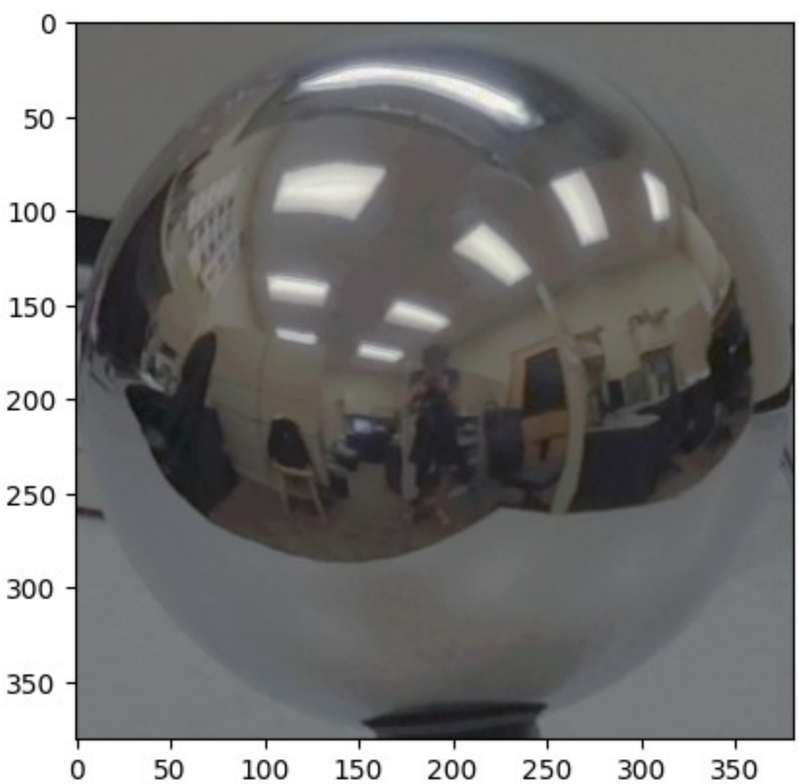
Compute HDR after calibrating the photometric reponses to obtain more accurate irradiance estimates from each image

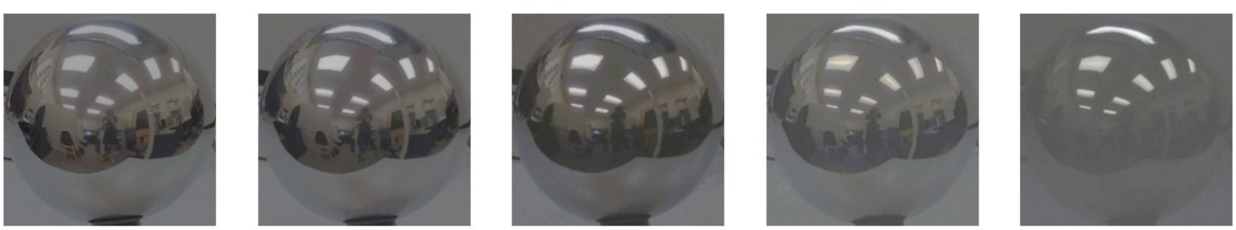
Some suggestions on using gsolve: When providing input to gsolve, don't use all available pixels, otherwise you will likely run out of memory / have very slow run times. To overcome, just randomly sample a set of pixels (1000 or so can suffice), but make sure all pixel locations are the same for each exposure. The weighting function w should be implemented using Eq. 4 from the paper (this is the same function that can be used for the previous LDR merging method). Try different lambda values for recovering g. Try lambda=1 initially, then solve for g and plot it. It should be smooth and continuously increasing. If lambda is too small, g will be bumpy. Refer to Eq. 6 in the paper for using g and combining all of your exposures into a final image. Note that this produces log irradiance values, so make sure to exponentiate the result and save irradiance in linear scale.

```
def make_hdr_estimation(ldr_images: np.ndarray, exposure_times: list,
lm)-> (np.ndarray, np.ndarray):
    '''
    Makes HDR image using multiple LDR images, and its corresponding
exposure values.
    Please refer to problem notebook for how to do it.

    **IMPORTANT**
    The gsolve operations should be ran with:
        Z: int64 array of shape N x P, where N = number of images, P =
number of pixels
        B: float32 array of shape N, log shutter times
        l: lambda; float to control amount of smoothing
        w: function that maps from float intensity to weight
    The steps to implement:
    1) Create random points to sample (from mirror ball region)
    2) For each exposures, compute g values using samples
    3) Recover HDR image using g values


    Args:
        ldr_images(np.ndarray): N x H x W x 3 shaped numpy array
representing
            N ldr images with width W, height H, and channel size of 3
(RGB)
        exposures(list): list of length N, representing exposures of
each images.
            Each exposure should correspond to LDR images' exposure
value.
        lm (scalar): the smoothing parameter
    Return:
        (np.ndarray): H x W x 3 shaped numpy array representing HDR
image merged using
            gsolve
```

```
        (np.ndarray): N x H x W x 3 shaped numpy array representing
log irradiances
            for each exposures
        (np.ndarray): 3 x 256 shaped numpy array representing g values
of each pixel intensities
            at each channels (used for plotting)
    '''
    N, H, W, C = ldr_images.shape
    # sanity check
    assert N == len(exposure_times)

    # TO DO: implement HDR estimation using gsolve
    # gsolve(Z, B, l, w) -> g, lE

    # Scale to 255 and initialize arrays
    ldr_images_copy = ldr_images.copy()
    ldr_images_copy *= 255
    calib_hdr_image = np.zeros((H, W, C))
    calib_log_irradiances = np.zeros((N, H, W, C))
    g = np.zeros((C, 256))

    # Randomly sample 1000 pixels
    numPix = 1000
    x = np.random.randint(0, W, size=numPix)
    y = np.random.randint(0, H, size=numPix)

    # Create Z, B, and w as specified above
    Z = np.zeros((N, numPix)).astype(np.int32)
    B = np.log(exposure_times)
    w = (128 - np.abs(np.arange(0, 256) - 128))

    # For each color channel, populate Z with the values at each
randomly selected pixel on n images, and then solve
    for c in range(C):
      for n in range(N):
        Z[n] = (ldr_images_copy[n, y, x, c]).astype(np.int32)
      g[c], lE = gsolve(Z, B, lm, w)

    for c in range(C):
      for y in range(H):
        for x in range(W):
          # Keep track of num/denom per image
          numSum = 0
          wSum = 0
          for n in range(N):
            z = int(ldr_images_copy[n, y, x, c])
            # Calculate irradiance
            calib_log_irradiances[n, y, x, c] = g[c][z] - B[n]
            # Get numerator and denominator for equation 6
            numSum += w[z] * calib_log_irradiances[n, y, x, c]
```

```
            wSum += w[z]
          # Apply equation 6
          calib_hdr_image[y, x, c] = np.exp(numSum / wSum)

    return calib_hdr_image, calib_log_irradiances, g

lm = 100
# get HDR image, log irradiance
calib_hdr_image, calib_log_irradiances, g =
make_hdr_estimation(ldr_images, exposure_times, lm)

# write HDR image to directory
write_hdr_image(calib_hdr_image, 'images/outputs/calib_hdr.hdr')

# display HDR image
display_hdr_image(calib_hdr_image)

display_images_linear_rescale(calib_log_irradiances)
```



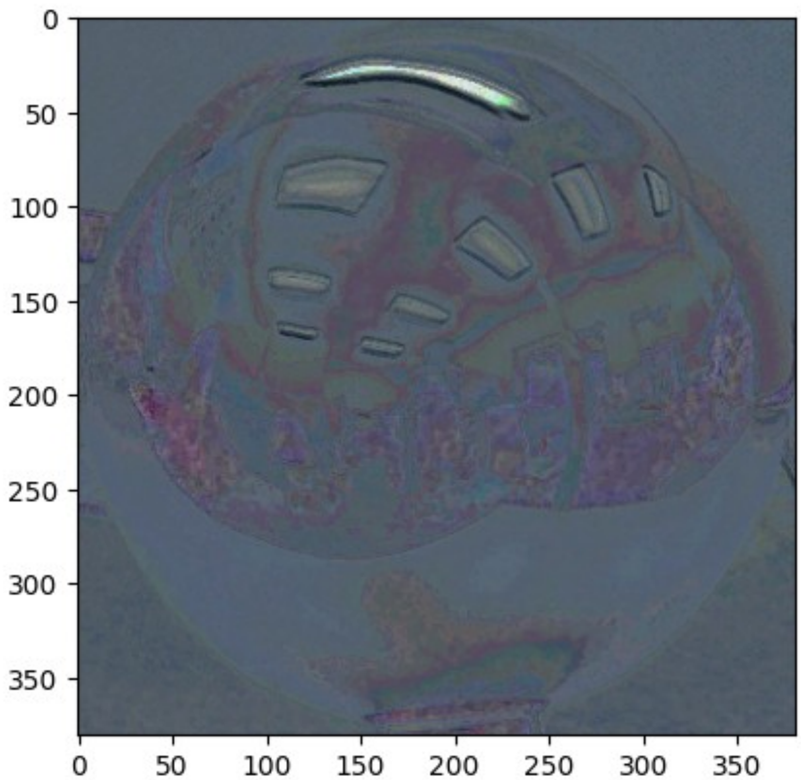


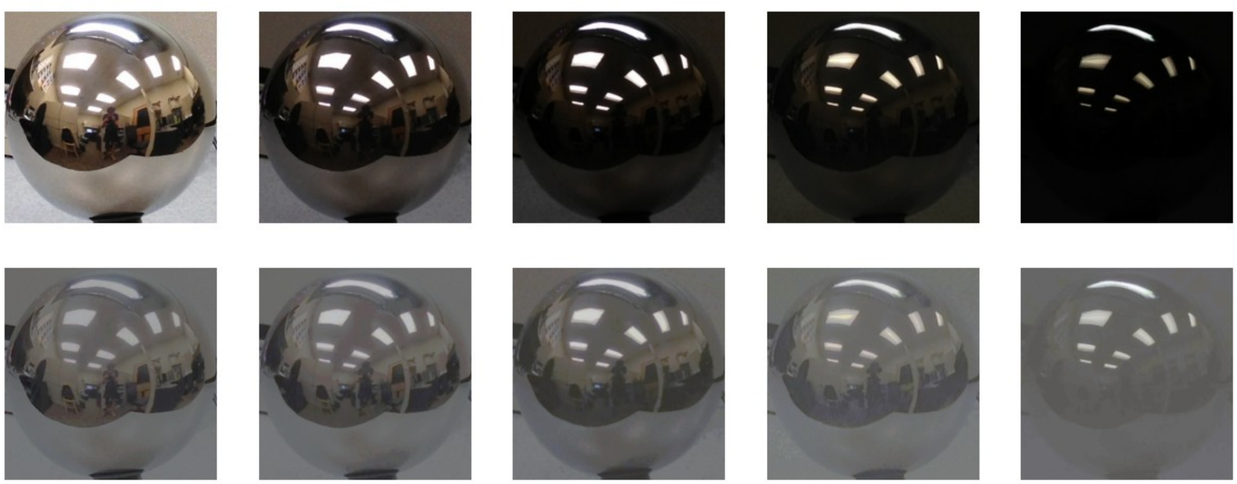
The following code displays your results. You can copy the resulting images and plots directly into your report where appropriate.

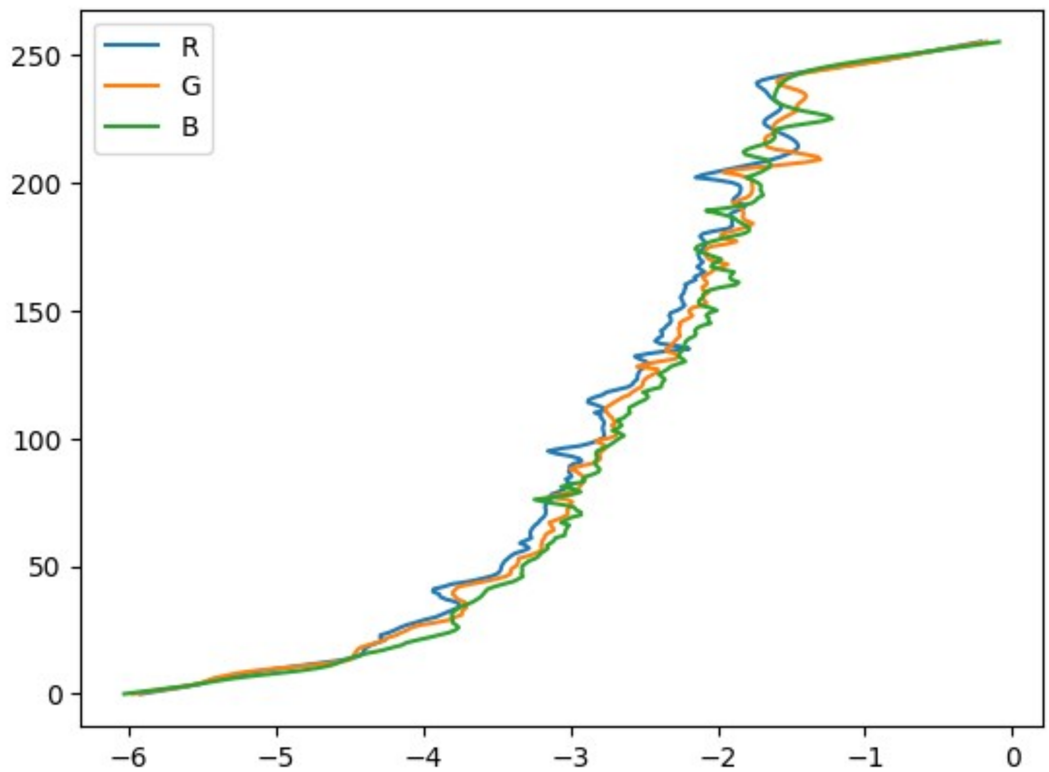
```
# display difference between calibrated and weighted
log_diff_im = np.log(calib_hdr_image/calib_hdr_image.mean())-
np.log(weighted_hdr_image/weighted_hdr_image.mean())
print('Min ratio = ', np.exp(log_diff_im).min(), '  Max ratio = ',
np.exp(log_diff_im).max())
plt.figure()
plt.imshow(rescale_images_linear(log_diff_im))

# display original images (code provided in utils.display)
display_images_linear_rescale(ldr_images)

# display log irradiance image (code provided in utils.display)
display_images_linear_rescale(calib_log_irradiances)

# plot g vs intensity, and then plot intensity vs g
N, NG = g.shape
labels = ['R', 'G', 'B']
plt.figure()
for n in range(N):
    plt.plot(g[n], range(NG), label=labels[n])
plt.gca().legend(('R', 'G', 'B'))

plt.figure()
for n in range(N):
    plt.plot(range(NG), g[n], label=labels[n])
plt.gca().legend(('R', 'G', 'B'))
```

```
Min ratio =  0.45597610160396496   Max ratio =  3.2549660893021675
```

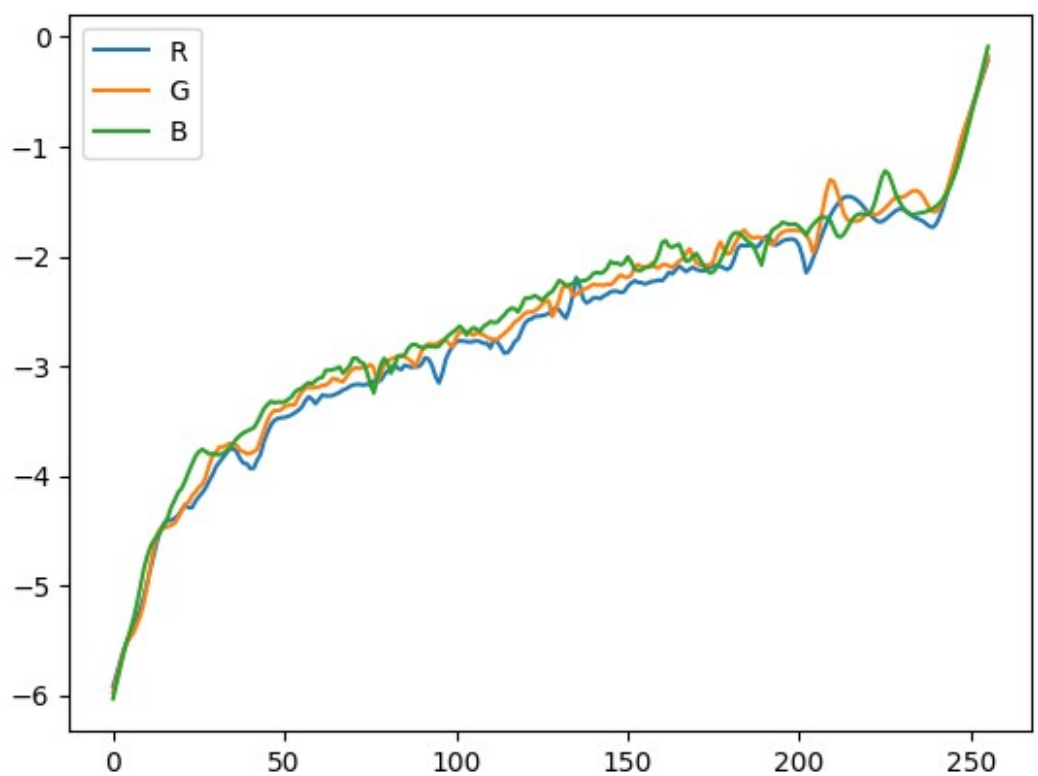
```
<matplotlib.legend.Legend at 0x7a327d243d90>
```


0
50
100
150
200
250
300
350
0 50 100 150 200 250 300 350





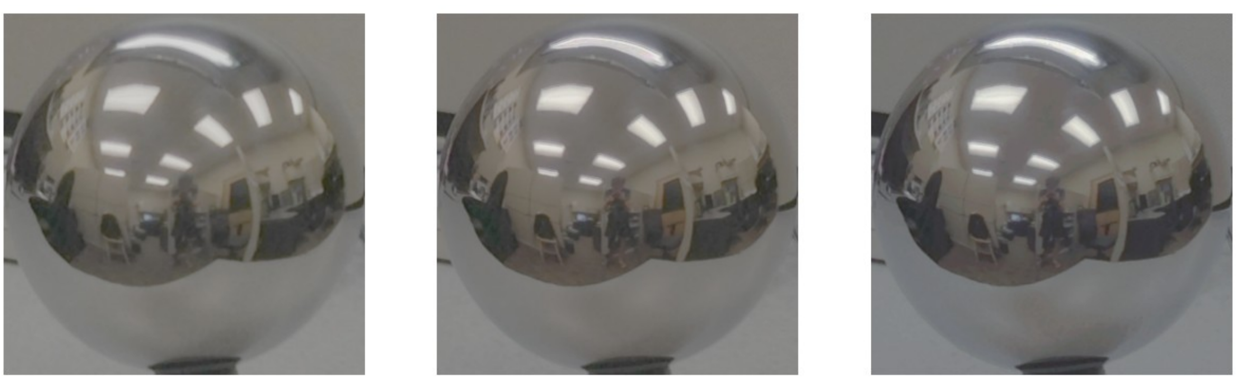
R
G
B



R
G
B

```
def weighted_log_error(ldr_images, hdr_image, log_irradiances):
  # computes weighted RMS error of log irradiances for each image
compared to final log irradiance
  N, H, W, C = ldr_images.shape
  w = 1-abs(ldr_images - 0.5)*2
  err = 0
  for n in np.arange(N):
    err += np.sqrt(np.multiply(w[n], (log_irradiances[n]-
np.log(hdr_image))**2).sum()/w[n].sum())/N
  return err


# compare solutions
err = weighted_log_error(ldr_images, naive_hdr_image,
naive_log_irradiances)
print('naive:  \tlog range = ', round(np.log(naive_hdr_image).max() -
np.log(naive_hdr_image).min(),3), '\tavg RMS error = ', round(err,3))
err = weighted_log_error(ldr_images, weighted_hdr_image,
naive_log_irradiances)
print('weighted:\tlog range = ',
round(np.log(weighted_hdr_image).max() -
np.log(weighted_hdr_image).min(),3), '\tavg RMS error = ',
round(err,3))
err = weighted_log_error(ldr_images, calib_hdr_image,
calib_log_irradiances)
print('calibrated:\tlog range = ', round(np.log(calib_hdr_image).max()
- np.log(calib_hdr_image).min(),3), '\tavg RMS error = ',
round(err,3))

# display log hdr images (code provided in utils.display)
display_images_linear_rescale(np.log(np.stack((naive_hdr_image/naive_h
dr_image.mean(), weighted_hdr_image/weighted_hdr_image.mean(),
calib_hdr_image/calib_hdr_image.mean()), axis=0)))
```

```
naive:      log range =  6.46     avg RMS error =  0.324
weighted:   log range =  6.622    avg RMS error =  0.286
calibrated:      log range =  6.982    avg RMS error =  0.247
```

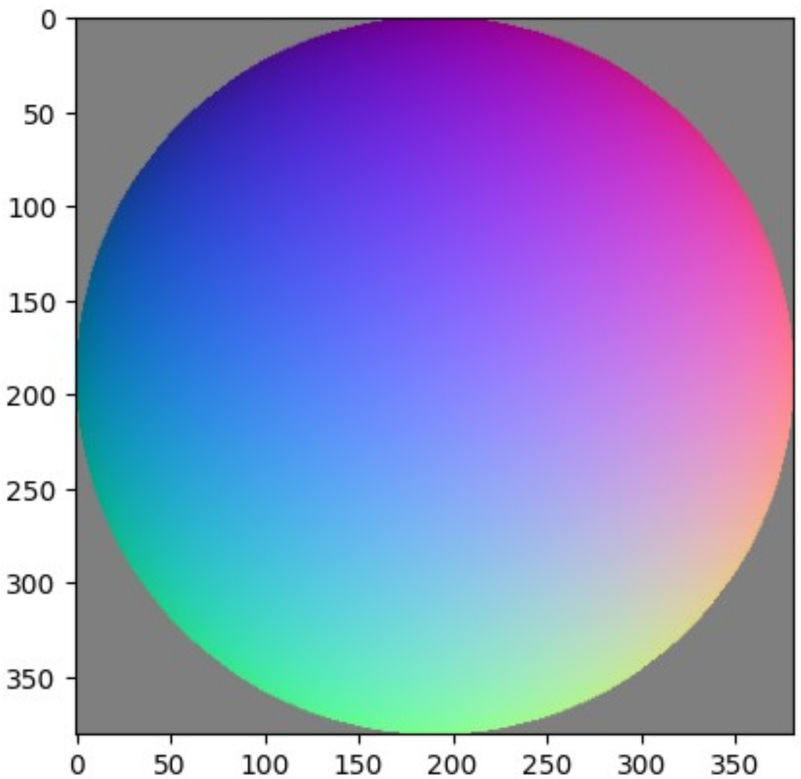


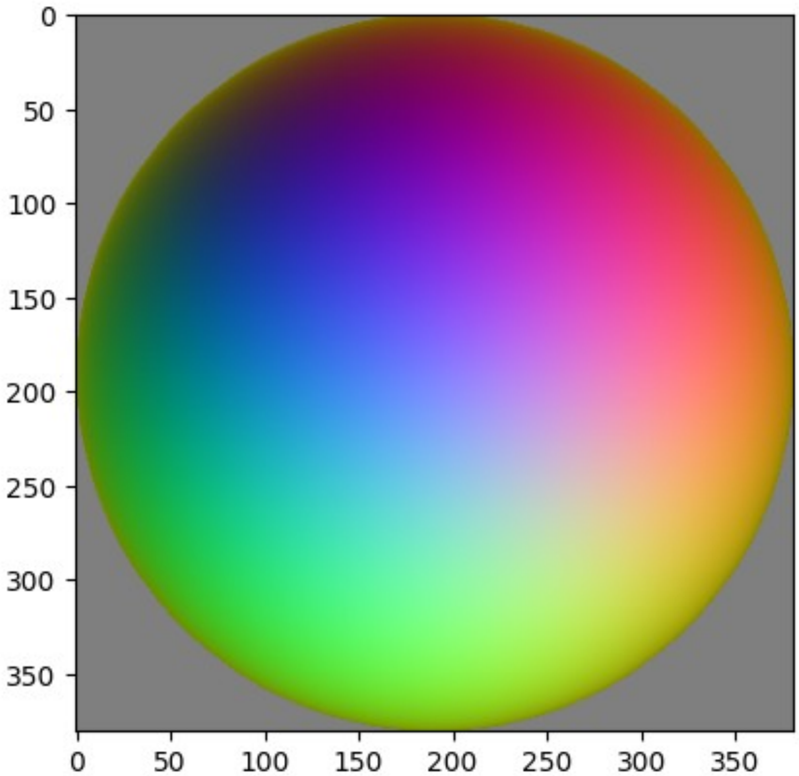
Panoramic transformations

Compute the equirectangular image from the mirrorball image

```
def panoramic_transform(hdr_image):
    '''
    Given HDR mirror ball image,

    Expects mirror ball image to have center of the ball at center of
the image, and
    width and height of the image to be equal.

    Steps to implement:
    1) Compute N image of normal vectors of mirror ball
    2) Compute R image of reflection vectors of mirror ball
    3) Map reflection vectors into spherical coordinates
    4) Interpolate spherical coordinate values into equirectangular
grid.

    Steps 3 and 4 are implemented for you with
get_equirectangular_image

    '''
    H, W, C = hdr_image.shape
    assert H == W
    assert C == 3

    # TO DO: compute N and R

    # R = V - 2 * dot(V,N) * N

    # Create V, N, R
    view_vector = np.array([0, 0, -1])
    normals = np.zeros((H, W, C))
    reflections = np.zeros((H, W, C))

    for y in range(H):
      for x in range(W):
        linX = (x - W/2)/(W/2)
        linY = (y - H/2)/(H/2)
        linZ = linX**2 + linY**2
        if (linZ > 1): linX, linY, linZ = 0, 0, 0
        else: linZ = math.sqrt(1 - linZ)
        normals[y, x] = linX, linY, linZ

    for y in range(H):
      for x in range(W):
        if (np.linalg.norm(normals[y, x]) != 0):
          reflections[y, x] = view_vector - 2 * np.dot(view_vector,
normals[y, x]) * normals[y, x]
```

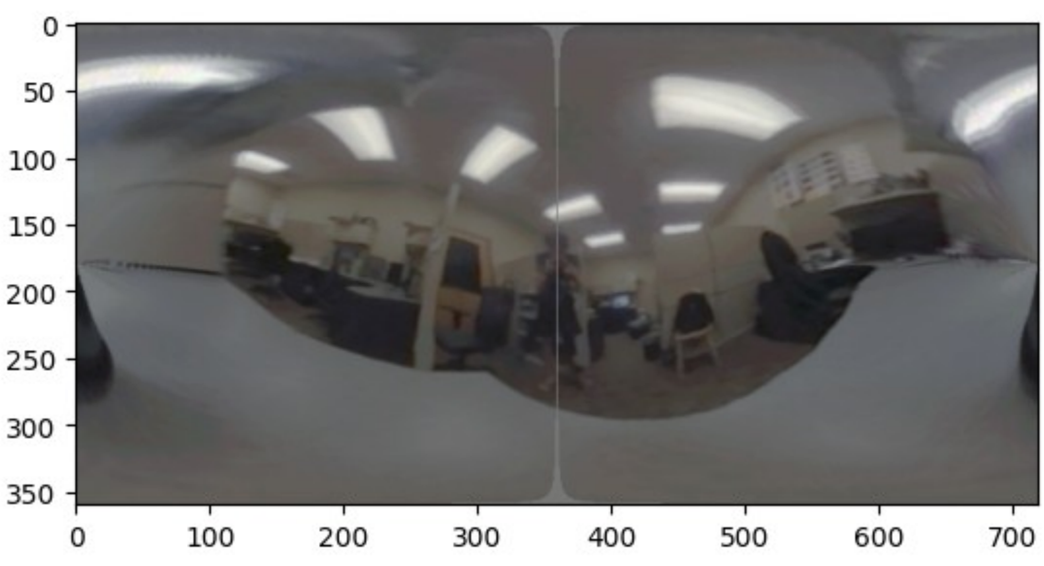
```
        else:
          reflections[y, x] = np.array([0, 0, 0])

    plt.imshow((normals+1)/2)
    plt.show()
    plt.imshow((reflections+1)/2)
    plt.show()

    equirectangular_image = get_equirectangular_image(reflections,
hdr_image)
    return equirectangular_image

hdr_mirrorball_image = calib_hdr_image
eq_image = panoramic_transform(hdr_mirrorball_image)

write_hdr_image(eq_image, 'images/outputs/equirectangular.hdr')

plt.figure(figsize=(15,15))
display_hdr_image(eq_image)
```






```
array([[[0.46657354, 0.46657354, 0.46657354],
        [0.46657354, 0.46657354, 0.46657354],
        [0.46657354, 0.46657354, 0.46657354],
        ...,
        [0.46657354, 0.46657354, 0.46657354],
        [0.46657354, 0.46657354, 0.46657354],
        [0.46657354, 0.46657354, 0.46657354]],

       [[0.3506415 , 0.3525284 , 0.35883594],
        [0.35063305, 0.3525239 , 0.35885957],
        [0.35062462, 0.35251936, 0.35888317],
        ...,
        [0.3506668 , 0.35254198, 0.35876504],
        [0.3506584 , 0.35253745, 0.35878867],
        [0.35064992, 0.35253292, 0.3588123 ]],

       [[0.39367342, 0.3934498 , 0.40243453],
        [0.3938767 , 0.39355376, 0.40263093],
        [0.3940797 , 0.3936577 , 0.4028271 ],
        ...,
        [0.39306185, 0.3931374 , 0.40184367],
        [0.393266  , 0.39324164, 0.4020409 ],
        [0.39346984, 0.39334574, 0.40223783]],

       ...,
```

```
        [[0.37984425, 0.36783084, 0.34791926],
         [0.38083425, 0.3678683 , 0.3479993 ],
         [0.3818176 , 0.36790583, 0.34807926],
         ...,
         [0.37683317, 0.36771834, 0.3476789 ],
         [0.37784377, 0.36775583, 0.34775904],
         [0.37884742, 0.36779335, 0.3478392 ]],

        [[0.37230697, 0.36635658, 0.34717506],
         [0.37256527, 0.3663878 , 0.34722477],
         [0.3728231 , 0.3664191 , 0.3472745 ],
         ...,
         [0.37152937, 0.36626276, 0.34702575],
         [0.37178904, 0.36629406, 0.34707552],
         [0.37204826, 0.3663253 , 0.3471253 ]],

        [[0.38064396, 0.36515546, 0.3481996 ],
         [0.38070264, 0.36516586, 0.34820294],
         [0.3807613 , 0.36517623, 0.34820628],
         ...,
         [0.3804678 , 0.36512434, 0.34818962],
         [0.38052654, 0.36513472, 0.34819296],
         [0.3805853 , 0.36514512, 0.34819627]]], dtype=float32)

<Figure size 1500x1500 with 0 Axes>
```



Rendering synthetic objects into photographs

Use Blender to render the scene with and with objects and obtain the mask image. The code below should then load the images and create the final composite.

```
# Read the images that you produced using Blender.  Modify names as
needed.
O = read_image('images/proj4_objects.png')
E = read_image('images/proj4_empty.png')
M = read_image('images/proj4_mask.png')
M = M > 0.5
I = background_image
I = cv2.resize(I, (M.shape[1], M.shape[0]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
<ipython-input-146-869da1b24074> in <cell line: 2>()
      1 # Read the images that you produced using Blender.  Modify
names as needed.
----> 2 O = read_image('images/proj4_objects.png')
      3 E = read_image('images/proj4_empty.png')
      4 M = read_image('images/proj4_mask.png')
      5 M = M > 0.5

/content/utils/io.py in read_image(image_path)
     35     # read image and convert to RGB
     36     bgr_image = cv2.imread(image_path)
---> 37     rgb_image = bgr_image[:, :, [2, 1, 0]]
     38     return rgb_image.astype(np.float32) / 255
     39 

TypeError: 'NoneType' object is not subscriptable
```

```
# TO DO: compute final composite
result = []

plt.figure(figsize=(20,20))
plt.imshow(result)
plt.show()

write_image(result, 'images/outputs/final_composite.png')
```

Bells & Whistles (Extra Points)

Additional Image-Based Lighting Result

Other panoramic transformations

Photographer/tripod removal

Local tonemapping operator