

Project 4. Image-Based Lighting - Template

October 29, 2019

1 CS445: Computational Photography

1.1 Programming Project 4: Image-Based Lighting

1.1.1 Due Date: 11:59pm on Tuesday, Oct. 29, 2019

This is a template solution file. Please feel free to use this for the base of your report.

1.2 Recovering HDR Radiance Maps (40 pts)

We start by loading in necessary libraries used for this section of the problem.

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

[2]: # 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

      # local imports
      from utils.io import read_image, write_image, read_hdr_image, write_hdr_image
      from utils.display import display_hdr_image_linear, display_hdr_image, ↪
      ↪display_log_irradiances
      from utils.hdr_helpers import gsolve
      from utils.bilateral_filter import bilateral_filter
      from utils.meta import TODO
```

1.2.1 Data collection (10 points)

1. Find a good scene to photograph. The scene should have a flat surface to place your spherical mirror on. Either indoors or outdoors will work.

2. Find a fixed, rigid spot to place your camera. A tripod is best, but you can get away with less. I used the back of a chair to steady my phone when taking my images.
3. Place your spherical mirror on a flat surface, and make sure it doesn't roll by placing a cloth/bottle cap/etc under it. Make sure the sphere is not too far away from the camera – it should occupy at least a 256x256 block of pixels.
4. Photograph the spherical mirror using at least three different exposure times. Make sure the camera does not move too much (slight variations are OK, but the viewpoint should generally be fixed). For best results, your exposure times should be at least 4 times longer and 4 times shorter (± 2 stops) than your mid-level exposure (e.g. if your mid-level exposure time is 1/40s, then you should have at least exposure times of 1/10s and 1/160s; the greater the range the better). Make sure to record the exposure times.
5. Remove the mirror from the scene, and from the same viewpoint as the other photographs, take another picture of the scene at a normal exposure level (most pixels are neither over- or under-exposed). This will be the image that you will use for object insertion/compositing (the “background” image).
6. After you copy all of the images from your camera/phone to your computer, load the spherical mirror images (from step 4) into your favorite image editor and crop them down to contain only the sphere.
7. Small alignment errors may occur (due to camera motion or cropping). One way to fix these is through various alignment procedures, but for this project, we won't worry about these errors. If there are substantial differences in camera position/rotation among the set of images, re-take the photographs.

```
[3]: # TODO: Replace this with actual file path!
low_exposure_mirror_ball_image_file = "images/inputs/160-1.png"
mid_exposure_mirror_ball_image_file = "images/inputs/40-1.png"
high_exposure_mirror_ball_image_file = "images/inputs/10-1.png"
background_image_file = "images/inputs/IMG_3850.JPG"

# TODO: Extract exposure values for each images
low_exposure = 1/160
mid_exposure = 1/40
high_exposure = 1/10

# These images will be used for LDR mergings
low_exposure_mirror_ball_image = cv2.imread(low_exposure_mirror_ball_image_file)
mid_exposure_mirror_ball_image = cv2.imread(mid_exposure_mirror_ball_image_file)
high_exposure_mirror_ball_image = cv2.
    ↳ imread(high_exposure_mirror_ball_image_file)
background_image = read_image(background_image_file)
```

```
[4]: # resize mirror ball images:
# TODO: change size of N to your favorite value!
N = 700
```

```

low_exposure_mirror_ball_image = cv2.resize(low_exposure_mirror_ball_image, (N,
↳N))
mid_exposure_mirror_ball_image = cv2.resize(mid_exposure_mirror_ball_image, (N,
↳N))
high_exposure_mirror_ball_image = cv2.resize(high_exposure_mirror_ball_image,
↳(N, N))

low_exposure_mirror_ball_image = cv2.cvtColor(low_exposure_mirror_ball_image,
↳cv2.COLOR_BGR2RGB)
mid_exposure_mirror_ball_image = cv2.cvtColor(mid_exposure_mirror_ball_image,
↳cv2.COLOR_BGR2RGB)
high_exposure_mirror_ball_image = cv2.cvtColor(high_exposure_mirror_ball_image,
↳cv2.COLOR_BGR2RGB)

ldr_images = np.stack((low_exposure_mirror_ball_image,
                        mid_exposure_mirror_ball_image,
                        high_exposure_mirror_ball_image))
exposures = [low_exposure, mid_exposure, high_exposure]

```

1.2.2 Naive LDR merging (10 points)

After collecting data, load the cropped images, and resize them to all be square and the same dimensions (e.g. `cv2.resize(ldr,(N,N))` N is the new size). Either find the exposure times using the EXIF data (usually accessible in the image properties, or via matlab's `imfinfo`), or refer to your recorded exposure times. To put the images in the same intensity domain, divide each by its exposure time (e.g. `ldr1_scaled = ldr1 / exposure_time1`). After this conversion, all pixels will be scaled to their approximate value if they had been exposed for 1s.

The easiest way to convert your scaled LDR images to an HDR is simply to average them. Create one of these for comparison to your later results.

To save the HDR image, use given `write_hdr_image` function. To visualize HDR image, use given `display_hdr_image` function.

```

[5]: 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 images 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 representing log

→ irradiances

for each exposures

```
'''
N, H, W, C = ldr_images.shape
# sanity check
assert N == len(exposures)
ldr_images = ldr_images.astype(np.float32)/255

# TODO: Implement ldr_images + exposures -> HDR image function here
# np_exposures = N, shaped array
hdr_image = np.zeros((H, W, C))
for i in range(N):
    hdr_image += (ldr_images[i]/exposures[i])

hdr_image /= N

log_irradiance = np.zeros((N, H, W, C))

for i in range(N):
    log_irradiance[i] = np.log(ldr_images[i]/exposures[i])

return hdr_image, log_irradiance
```

```
[6]: # get HDR image, log irradiance
naive_hdr_image, naive_log_irradiance = make_hdr_naive(ldr_images, exposures)

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

# display HDR image
display_hdr_image(naive_hdr_image)

# display log irradiance image
```

```
display_log_irradiance(naive_log_irradiance)
```

/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:42:

RuntimeWarning: divide by zero encountered in log

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



1.2.3 LDR merging without under- and over-exposed regions (5 points)

The naive method has an obvious limitation: if any pixels are under- or over-exposed, the result will contain clipped (and thus incorrect) information. A simple fix is to find these regions (e.g. a pixel might be considered over exposed if its value is less than 0.02 or greater than 0.98, assuming $[0,1]$ images), and exclude them from the averaging process. Another way to think about this is that the naive method is extended using a weighted averaging procedure, where weights are 0 if the pixel is over/under-exposed, and 1 otherwise. Note that with this method, it might be the case that for a given pixel it is never properly exposed (i.e. always either above or below the threshold

in each exposure).

There are perhaps better methods that achieve similar results but don't require a binary weighting. For example, we could create a weighting function that is small if the input (pixel value) is small or large, and large otherwise, and use this to produce an HDR image. In python, such a function can be created with:

```
w = lambda z: float(128-abs(z-128))
```

assuming pixel values range in [0,255].

```
[7]: def make_hdr_filtered(ldr_images: np.ndarray, exposures: list) -> (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.

        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)
        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 without
            under - over exposed regions
        (np.ndarray): N x H x W x 3 shaped numpy array representing log
    ↳ irradiances
            for each exposures
    '''
    N, H, W, C = ldr_images.shape
    print(N, H, W, C)
    # sanity check
    assert N == len(exposures)

    # TODO: Implement ldr_images + exposures -> HDR image function here
    w = np.vectorize(lambda z: float(128-abs(z-128)))
```

```

hdr_image = np.zeros((H, W, C))
sum_w = np.zeros((H, W, C))

for i in range(N):
    weights = w(ldr_images[i])
    sum_w += weights
    hdr_image += (weights*ldr_images[i])/(exposures[i]*255)

hdr_image /= sum_w

log_irradiance = np.zeros((N, H, W, C))

for i in range(N):
    log_irradiance[i] = np.log(ldr_images[i]/exposures[i])

return hdr_image, log_irradiance

```

```

[8]: # get HDR image, log irradiance
filtered_hdr_image, filtered_log_irradiance = make_hdr_filtered(ldr_images,
↳ exposures)

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

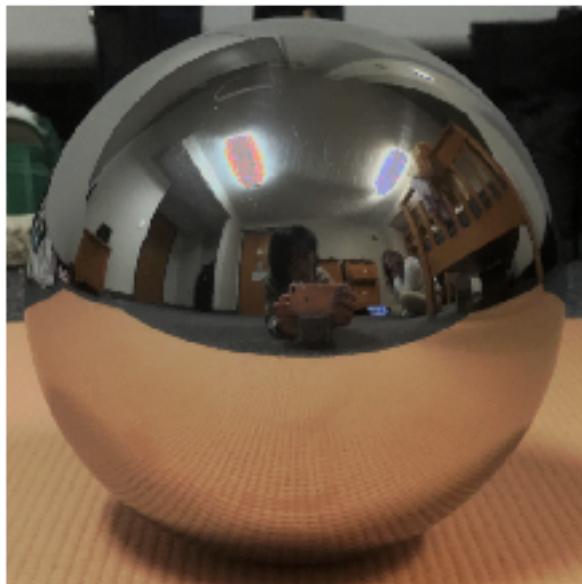
# display HDR image
display_hdr_image(filtered_hdr_image)

# display log irradiance image
display_log_irradiance(filtered_log_irradiance)

```

3 700 700 3

/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:48:
RuntimeWarning: divide by zero encountered in log
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).



1.2.4 LDR merging and response function estimation (15 points)

Nearly all cameras apply a non-linear function to recorded raw pixel values in order to better simulate human vision. In other words, the light incoming to the camera (radiance) is recorded by the sensor, and then mapped to a new value by this function. This function is called the film response function, and in order to convert pixel values to true radiance values, we need to estimate this response function. Typically the response function is hard to estimate, but since we have multiple observations at each pixel at different exposures, we can do a reasonable job up to a missing constant.

The method we will use to estimate the response function is outlined in this paper. Given pixel values Z at varying exposure times t , the goal is to solve for $g(Z) = \ln(R \cdot t) = \ln(R) + \ln(t)$. This boils down to solving for R (irradiance) since all other variables are known. By these definitions, g is the inverse, log response function. The paper provides code to solve for g given a set of pixels at varying exposures (we also provide `gsolve` in `utils`). Use this code to estimate g for each image channel (`r/g/b`). Then, recover the HDR image using equation 6 in the paper.

Some hints 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 (100 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, i.e. $w = \lambda z: \text{float}(128 - \text{abs}(z - 128))$).
- Try different λ values for recovering g . Try $\lambda=1$ initially, then solve for g and plot it. It should be smooth and continuously increasing. If λ 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 radiance values, so make sure to exponentiate the result and save absolute radiance.

```
[9]: def make_hdr_estimation(ldr_images: np.ndarray, exposures: list) -> (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.  
  
    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.  
    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, _ = ldr_images.shape  
  
    # sanity check  
    assert N == len(exposures)
```

```

# implement HDR estimation using gsolve

w = lambda z: float(129-abs(z-128))

hdr_image = np.zeros((H, W, 3))
g_per_exposures = np.zeros((3, 256))

log_exp = np.log(exposures)

log_irradiances = np.zeros((N, H, W, 3))

w = np.vectorize(w)

ys = np.random.randint(H, size=800)
xs = np.random.randint(W, size=800)

for c in range(3):

    g_per_exposures[c] = g_compute(ldr_images[:, :, :, c], w, 100, 800, xs,
    ↪ys, log_exp).T
    funcg = np.vectorize(lambda x: g_per_exposures[c][int(x)])
    weights = np.zeros((H, W))
    num = np.zeros((H, W))

    for n in range(N):
        tmp_weights = w(ldr_images[n, :, :, c])
        num += tmp_weights*(funcg(ldr_images[n, :, :, c])-log_exp[n])
        log_irradiances[n, :, :, c] = funcg(ldr_images[n, :, :, c])-log_exp[n]
        weights += tmp_weights

    hdr_image[:, :, c] = num/weights

hdr_image = abs(np.exp(hdr_image))

return hdr_image, log_irradiances, g_per_exposures

def g_compute(ldr_images, w, l, p, xs, ys, B):
    '''
    Args:
        l: lambda
        p(int): length of the sample
        c(int): channel
    Return:
        g

    **IMPORTANT**

```

The gsolve operations should be ran with:

Z: int64 array of shape $N \times P$, where N = number of images, P = number of pixels

B: float32 array of shape N ,

l: Number

W: function that takes int and returns float

```
'''
N, H, W = ldr_images.shape
Z = np.zeros((N, p))
for m in range(p):
    Z[:, m] = ldr_images[:, ys[m], xs[m]]
Z = Z.astype(np.int64)

g, lE = gsolve(Z, B, l, w)
return g
```

```
[10]: # get HDR image, log irradiance
estimated_hdr_image, estimated_log_irradiance, estimated_g =
    make_hdr_estimation(ldr_images, exposures)

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

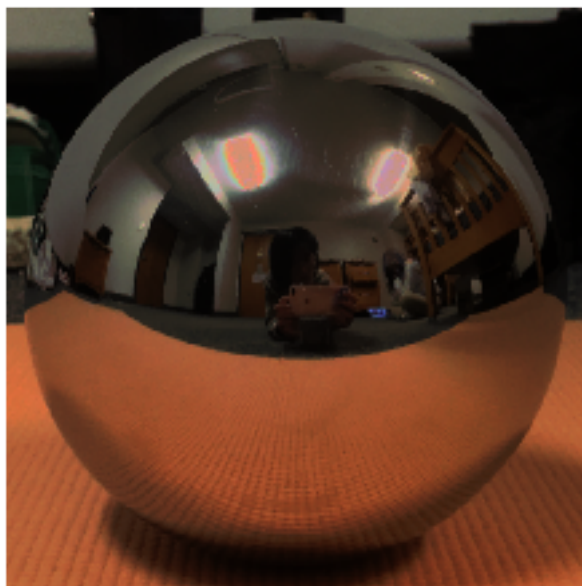
# display HDR image
display_hdr_image(estimated_hdr_image)

# display log irradiance image
display_log_irradiance(estimated_log_irradiance)
```

/Users/minnie/Downloads/proj4_materials/utils/hdr_helpers.py:62: FutureWarning:
`rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.

To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

```
x = np.linalg.lstsq(A, b)[0]
```

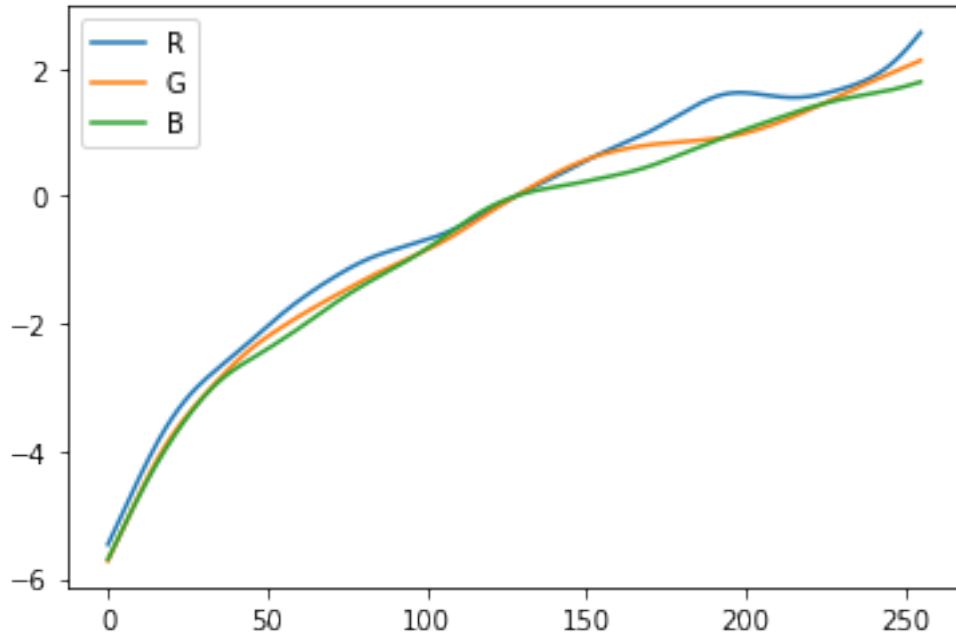


```
[11]: display_hdr_image_linear(np.log(estimated_hdr_image))
```



```
[12]: # display G function for each intensity values
N, NG = estimated_g.shape
labels = ['R', 'G', 'B']
plt.figure()
for n in range(N):
    plt.plot(range(NG), estimated_g[n], label=labels[n])
plt.gca().legend(('R', 'G', 'B'))
```

[12]: <matplotlib.legend.Legend at 0xb2eb7e650>



1.3 Panoramic transformations (20 pts)

Now that we have an HDR image of the spherical mirror, we'd like to use it for relighting (i.e. image-based lighting). However, many programs don't accept the "mirror ball" format, so we need to convert it to a different 360 degree, panoramic format (there is a nice overview of many of these formats [here](#)). For this part of the project, you should implement the mirror ball to equirectangular (latitude longitude) transformation. Most rendering software accepts this format, including Blender's Cycles renderer, which is what we'll use in the next part of the project.

To perform the transformation, you need to figure out the mapping between the mirrored sphere domain and the equirectangular domain. Hint: calculate the normals of the sphere (N) and assume the viewing direction (V) is constant. You can calculate reflection vectors with $R = V - 2 * \text{dot}(V, N) * N$, (NOTE that you'd have to implement channel-wise dot product). which is the direction that light is incoming from the world to the camera after bouncing off the sphere. The reflection vectors can then be converted to, providing the latitude and longitude (ϕ and θ) of the given pixel (fixing the distance to the origin, r , to be 1). Note that this assumes an orthographic camera (which is a close approximation as long as the sphere isn't too close to the camera).

Next, the equirectangular domain can be created by making an image in which the rows correspond to θ and columns correspond to ϕ in spherical coordinates, e.g.

```
EH, EW = 360, 720
phi_fst_half = np.arange(math.pi, 2*math.pi, math.pi / (EW // 2))
phi_snd_half = np.arange(0 * math.pi, math.pi, math.pi / (EW // 2))

theta_range = np.arange(0, math.pi, math.pi / EH)
phi_ranges = np.concatenate((phi_fst_half, phi_snd_half))
```

```
phis, thetas = np.meshgrid(phi_ranges, theta_range)
```

Note that by choosing 360 as EH and 720 as EW, we are making every pixel in equirectangular image to correspond to area occupied by 0.5 degree x 0.5 degree in spherical coordinate. Now that you have the phi/theta for both the mirror ball image and the equirectangular domain, use scipy's `scipy.interpolate.griddata` function to perform the transformation. Below is an example transformation.

Note that the following portion of the project depends on successfully converting your mirror ball HDR image to the equirectangular domain. If you cannot get this working, you can request code from the instructors at a 20 point penalty (i.e. no points will be awarded for this section, but you can do the later sections).

```
[13]: hdr_mirrorball_image = read_hdr_image('images/outputs/estimated_hdr.hdr')
```

```
[14]: 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 normal vector from mirror ball  
    2) Compute reflection vector of mirror ball using given equation  
    3) Map reflection vectors into spherical coordinates  
    4) Interpolate spherical coordinate values into equirectangular grid.  
       - hint: use scipy.interpolate.griddata  
  
    '''  
    H, W, C = hdr_image.shape  
    assert H == W  
    assert H % 2 == 0  
    assert C == 3  
    R = H // 2  
  
    # get equirectangular image somehow  
    phi = np.zeros((H, W))  
    theta = np.zeros((H, W))  
  
    n = 0  
    for i in range(H):  
        for j in range(W):  
  
            reflect = reflection_compute(i, j, R)
```

```

        if sum(np.square(reflect)) == 0:
            phi[i, j] = 0
            theta[i, j] = 0
        else:
            phi[i, j] = math.atan2(reflect[0], reflect[2])
            theta[i, j] = math.acos(reflect[1])
            n += 1

phi %= math.pi*2

print("phi theta")

HDR_R = np.zeros(n)
HDR_G = np.zeros(n)
HDR_B = np.zeros(n)
points = np.zeros((n, 2))
counter = 0
for i in range(H):
    for j in range(W):
        if (phi[i, j] == 0 and theta[i, j] == 0):
            continue
        points[counter, 0] = theta[i, j]
        points[counter, 1] = phi[i, j]
        HDR_R[counter] = hdr_image[i, j, 0]
        HDR_G[counter] = hdr_image[i, j, 1]
        HDR_B[counter] = hdr_image[i, j, 2]
        counter += 1

'''
Viewing direction : 0, 0, 1
X: right, Y: up
Phi: atan2(RX, RZ)
Theta: arccos(RY)
Equirectangular image:
Phi: [PI - 2PI][0 - PI]
Theta: 0 - PI
'''
EH, EW = 360, 720
phi_fst_half = np.arange(math.pi, 2*math.pi, math.pi / (EW // 2))
phi_snd_half = np.arange(0 * math.pi, math.pi, math.pi / (EW // 2))
theta_range = np.arange(0, math.pi, math.pi / EH)
phi_ranges = np.concatenate((phi_fst_half, phi_snd_half))
phis, thetas = np.meshgrid(phi_ranges, theta_range)

xi = np.zeros((360 * 720, 2))
counter = 0

```



```

for i in range(360):
    for j in range(720):
        xi[counter, 0] = thetas[i, j]
        xi[counter, 1] = phis[i, j]
        counter += 1

'''
griddata((theta, phi), hdr_image, (theta_out, phi_out))
You should make your input shapes into:
points: n x 2
values: n x 0 (or (n,) in numpy shape representation)
xi: M x 2
Here, n should be number of pixels on the mirror ball(roughly  $PI * R^2$ ,
↪where  $R = W // 2$ )
M should be your equirectangular image size ( $360 * 720 = 259200$ )
griddata will then return  $M \times 0$  shaped array, which then you have to resize
↪into  $360 \times 720$ 
'''

equirectangular_R = np.reshape(griddata(points, HDR_R, xi,
↪method='linear'), (360, 720))
equirectangular_G = np.reshape(griddata(points, HDR_G, xi,
↪method='linear'), (360, 720))
equirectangular_B = np.reshape(griddata(points, HDR_B, xi,
↪method='linear'), (360, 720))

equirectangular_image = np.zeros((360, 720, 3))
equirectangular_image[:, :, 0] = equirectangular_R
equirectangular_image[:, :, 1] = equirectangular_G
equirectangular_image[:, :, 2] = equirectangular_B
return equirectangular_image

def reflection_compute(col, row, r):
    nx = (row - r) / r
    ny = -(col - r) / r
    if (np.square(nx) + np.square(ny) >= 1):
        nx = 0
        ny = 0
        nz = 0
    else:
        nz = - math.sqrt(1 - np.square(nx) - np.square(ny))

    N = np.array([nx, ny, nz])

    result = np.zeros(3)
    if (np.square(nx) + np.square(ny) + np.square(nz) != 0):

```

```

V = np.array([0, 0, 1])
R = V - 2 * np.dot(V, N) * N
Rn = np.linalg.norm(R)
result = R / Rn

return result

```

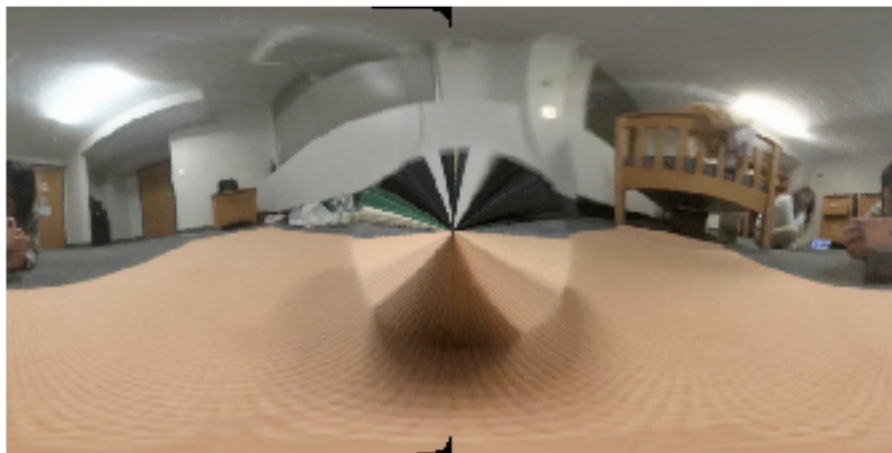
```

[15]: eq_image = panoramic_transform(naive_hdr_image)
display_hdr_image(eq_image)
write_hdr_image(eq_image, 'images/outputs/equirectangular.hdr')

```

phi theta

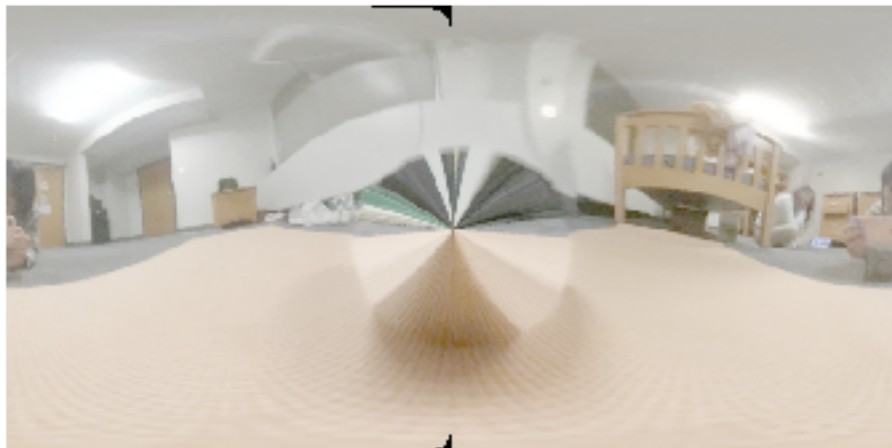
Warning: Negative / Inf values found in hdr image. Clamping to nearest valid value



```

[16]: display_hdr_image_linear(np.log(eq_image))

```



1.4 Rendering synthetic objects into photographs (30 pts)

Next, we will use our equirectangular HDR image as an image-based light, and insert 3D objects into the scene. This consists of 3 main parts: modeling the scene, rendering, and compositing. Specific instructions follow below; if interested, see additional details in Debevec’s paper.

Begin by downloading/installing the Blender. This template assumes that you have version 2.8 of the blender; if you are using version 2.7x, please refer to project webpage for detailed steps. The course webpage has tutorial with sample blend file, while this tutorial assumes that you create your own blend file from scratch. Please right click, open in new tab to view GIFs in full size.

1.4.1 Modeling the scene

To insert objects, we must have some idea of the geometry and surface properties of the scene, as well as the lighting information that we captured in previous stages. In this step, you will manually create rough scene geometry/materials using Blender.

First clear out initial scene including sample mesh and lighting.

Next, set camera to correct position

Next, go into perspective camera mode.

Then, setup viewport dimension. You want the same dimension as your background image for this.

We then want to load background image onto this viewport. Click on ‘g’ to move background image around, and ‘s’ to resize it.

Then, we want to add local scene. That is, add simple geometry (usually planes suffice) to recreate the geometry in the scene near where you’d like to insert objects. For best results, this should be close to where you placed the spherical mirror. Feel free to use the sample scene provided and move the vertices of the plane to match the surface you’d like to recreate (ignore the inserted bunny/teapot/etc for now). Once you’re happy with the placement, add materials to the local scene: select a piece of local scene geometry, go to Properties->Materials, add a Diffuse BSDF material, and change the “Color” to roughly match the color from the photograph. Insert synthetic objects into the scene. Feel free to use the standard models that I’ve included in the sample blend file, or find your own (e.g. Turbosquid, Google 3D Warehouse, DModelz, etc). Add interesting materials to your inserted objects as well. This tutorial is a great introduction to creating materials in Blender. Once finished, your scene should now look something like the right image below. Then, add your HDR image (the equirectangular map made above) to the scene.

First, use notebook to save the HDR panorama: `write_hdr_image(eq_image, ‘equirectangular.hdr’)`.

In the World tab on property panel, make sure Surface=“Background” and Color=“Environment Texture”. Locate your saved HDR image in the filename field below “Environment Texture”.

Finally, render scene using ‘cycle’ renderer. Note that this step takes a while to run.

Once you've finished rendering scene with object, you should save it using `image > save` functionality in render tab.

You also need to create 'empty' scene without inserted objects, and mask for added objects. To render masked scene, remove equirectangular based lighting, and set all inserted objects' material to emission. You can render the mask by using 'Eevee' renderer

With all rendered images, you should have there three images, (rendered with object, without object, and mask):

```
[17]: # this part assumes that you have these from Blender.
blender_output_with_object_path = "images/inputs/with_background.png"
blender_output_without_object_path = "images/inputs/without_objects-1.png"
blender_output_mask_path = "images/inputs/mask.png"

O = read_image(blender_output_with_object_path)
E = read_image(blender_output_without_object_path)
M = read_image(blender_output_mask_path)
M = M > 0.7
I = background_image

[18]: merged = M*O + (1-M)*I + (1-M)*(O-E)*1

[19]: write_image(merged, 'images/outputs/merged.png')
```

1.5 Bells & Whistles (Extra Points)

1.5.1 Other panoramic transformations (20 pts)

Different software accept different spherical HDR projections. In the main project, we've converted from the mirror ball format to the equirectangular format. There are also two other common formats: angular and vertical cross (examples here). Implement these transformations for 10 extra points each (20 possible).

1.5.2 Photographer/tripod removal (20 pts)

If you look closely at your mirror ball images, you'll notice that the photographer (you) and/or your tripod is visible, and probably occupies up a decent sized portion of the mirror's reflection. For 20 extra points, implement one of the following methods to remove the photographer: 1. Cut out the photographer and use in-painting/hole-filling to fill in the hole with background pixels (similar to the bells and whistles from Project 2), or 2. Use Debevec's method for removing the photographer (outlined here, steps 3-5; feel free to use Debevec's HDRShop for doing the panoramic rotations/blending).

The second option works better, but requires you to create an HDR mirror ball image from two different viewpoints, and then merge them together using blending and panoramic rotations.

1.5.3 Local tonemapping operator (30 pts)

HDR images can also be used to create hyper-realistic and contrast enhanced LDR images. This paper describes a simple technique for increasing the contrast of images by using a local tonemapping operator, which effectively compresses the photo's dynamic range into a displayable format while still preserving detail and contrast. For 30 extra credit points, implement the method found in the paper and compare your results to other tonemapping operations (see example below for ideas). You can use `bilateral_filter` code, provided by us, in your implementation, but do not use any other third party code. You can find some example HDR images here, including the memorial church image used below.

From left to right: simple rescaling, rescaling+gamma correction, local tonemapping operator, local tonemapping+gamma correction.