```
import numpy as np
import matplotlib.pyplot as plt
from imageio import imread,imsave

imFile = 'stpeters_probe_small.png'
compositeFile = 'tennis.png'
targetFile = 'interior.jpg'

# This loads and returns all of the images needed for the problem
# data - the image of the spherical mirror
# tennis - the image of the tennis ball that we will relight
# target - the image that we will paste the tennis ball onto
def loadImages():
    imFile = 'stpeters_probe_small.png'
    compositeFile = 'tennis.png'
    targetFile = 'interior.jpg'

    data = imread(imFile).astype('float')*1.5
    tennis = imread(compositeFile).astype('float')
    target = imread(targetFile).astype('float')/255

    return data, tennis, target


# This function takes as input a square image of size m x m x c
# where c is the number of color channels in the image.  We
# assume that the image contains a scphere and that the edges
# of the sphere touch the edge of the image.
# The output is a tuple (ns, vs) where ns is an n x 3 matrix
# where each row is a unit vector of the direction of incoming light
# vs is an n x c vector where the ith row corresponds with the
# image intensity of incoming light from the corresponding row in ns
def extractNormals(img):

    # Assumes the image is square
    d = img.shape[0]
    r = d / 2
    ns = []
    vs = []
    for i in range(d):
        for j in range(d):

            # Determine if the pixel is on the sphere
            x = j - r
            y = i - r
            if x*x + y*y > r*r-100:
                continue

            # Figure out the normal vector at the point
            # We assume that the image is an orthographic projection
            z = np.sqrt(r*r-x*x-y*y)
            n = np.asarray([x,y,z])
            n = n / np.sqrt(np.sum(np.square(n)))
            view = np.asarray([0,0,-1])
            n = 2*n*(np.sum(n*view))-view
```

```
            ns.append(n)
            vs.append(img[i,j])

    return np.asarray(ns), np.asarray(vs)

# This function renders a diffuse sphere of radius r
# using the spherical harmonic coefficients given in
# the input coeff where coeff is a 9 x c matrix
# with c being the number of color channels
# The output is an 2r x 2r x c image of a diffuse sphere
# and the value of -1 on the image where there is no sphere
def renderSphere(r,coeff):

    d = 2*r
    img = -np.ones((d,d,3))
    ns = []
    ps = []

    for i in range(d):
        for j in range(d):

            # Determine if the pixel is on the sphere
            x = j - r
            y = i - r
            if x*x + y*y > r*r:
                continue

            # Figure out the normal vector at the point
            # We assume that the image is an orthographic projection
            z = np.sqrt(r*r-x*x-y*y)
            n = np.asarray([x,y,z])
            n = n / np.sqrt(np.sum(np.square(n)))
            ns.append(n)
            ps.append((i,j))

    ns = np.asarray(ns)
    B = computeBasis(ns)
    vs = B.dot(coeff)

    for p,v in zip(ps,vs):
        img[p[0],p[1]] = np.clip(v,0,255)

    return img

# relights the sphere in img, which is assumed to be a square image
# coeff is the matrix of spherical harmonic coefficients
def relightSphere(img, coeff):
    img = renderSphere(int(img.shape[0]/2),coeff)/255*img/255
    return img

# Copies the image of source onto target
# pixels with values of -1 in source will not be copied
def compositeImages(source, target):

    # Assumes that all pixels not equal to 0 should be copied
```

```
    out = target.copy()
    cx = int(target.shape[1]/2)
    cy = int(target.shape[0]/2)
    sx = cx - int(source.shape[1]/2)
    sy = cy - int(source.shape[0]/2)

    for i in range(source.shape[0]):
        for j in range(source.shape[1]):
            if np.sum(source[i,j]) >= 0:
                out[sy+i,sx+j] = source[i,j]

    return out

# Fill in this function to compute the basis functions
# This function is used in renderSphere()
def computeBasis(ns):
    # Returns the first 9 spherical harmonic basis functions

    ################################################
    # Compute the first 9 basis functions
    ################################################
    B = np.ones((len(ns),9)) # This line is here just to fill space
    B[:, 1] = ns[:, 1]
    B[:, 2] = ns[:, 0]
    B[:, 3] = ns[:, 2]
    B[:, 4] = ns[:, 0]*ns[:, 1]
    B[:, 5] = ns[:, 1]*ns[:, 2]
    B[:, 6] = 3*(ns[:, 2]**2)-1
    B[:, 7] = ns[:, 0]*ns[:, 2]
    B[:, 8] = ns[:, 0]**2-ns[:, 1]**2
    return B

if __name__ == '__main__':

    data,tennis,target = loadImages()
    ns, vs = extractNormals(data)
    B = computeBasis(ns)

    # reduce the number of samples because computing the SVD on
    # the entire data set takes too long
    Bp = B[::50]
    vsp = vs[::50]

    ################################################
    # Solve for the coefficients using least squares
    # or total least squares here
    ################################################
    #coeff = np.zeros((9,3))
    #coeff[0,:] = 255

    #Solve for the coeff using least squares
    coeff = np.linalg.solve(Bp.T@Bp, Bp.T@vsp)
    #Solve for the coeff using total least squares
    #Scale the outputs
```

```
vsp = vsp/384


A = np.append(Bp, vsp, axis=1)
u, s, vt = np.linalg.svd(A)
n = (Bp.T@Bp).shape[0]

v = vt.T
v_xy = v[0:n, n:n+3]
v_yy = v[n:n+3, n:n+3]
v_yy_inv = np.linalg.solve(v_yy, np.identity(3))
coeff = -1*(v_xy@v_yy_inv)
coeff = coeff*384


img = relightSphere(tennis,coeff)

output = compositeImages(img,target)

print('Coefficients:\n'+str(coeff))

plt.figure(1)
plt.imshow(output)
plt.show()

imsave('output.png',output)
#imsave('output1.png', output)
#imsave('output2.png', output)
```