CS189 HW4

Hanze Yao
3033893286

2. Gaussian Features

(a) The code is shown in the appendix.
For this part, the code includes the function polynomial_augmentation in lib.py and part_a() in main.py
The plots are also shown in the appendix with the output.
From what I observe of the fitted curve, as the increase of polynomial degree, the curve fits more and more to the data points. This can also be seen from the decrease of least square error.

(b) The code is shown in the appendix
For this part, the code includes the function rbf_augmentation in lib.py and part_b() in main.py
The plots are also shown in the appendix with the output.
I choose $C_1 = -2$ $C_2 = 1$ $C_3 = 3$
Variance: 3, 2, 0.5
The least square error is 8.706.

3. Probabilistic Principal Components Analysis (PPCA)

(a) $y = Wx + \mu + z$

$\text{mean}(y|x) = Wx + \mu$

$\text{Var}(y|x) = \Sigma = \sigma^2 I$

$\because$ This is still a gaussian distribution

$\therefore y|x \sim N(Wx + \mu, \sigma^2 I)$

(b) $\because x \sim N(0, I)$

$\therefore Wx \sim N(0, WW')$

$\because z \sim N(0, \sigma^2 I)$

$\therefore y = Wx + \mu + z \sim N(\mu, WW' + \sigma^2 I)$

4. Canonical Correlation Analysis

(a) Because this question needs to find a linear structure from paired data (X, Y). And Y is also n draws from random variable y. We care about the variation of Y too.

(b) $P = u^T X \quad Q = V^T y$

$$\rho(P, Q) = \rho(u^T X, V^T y) = \frac{Cov(u^T X, V^T y)}{\sqrt{Var(u^T X)\, Var(V^T y)}}$$

$$= \frac{E((u^T X - E(u^T X))(V^T y - E(V^T y)))}{\sqrt{Cov(u^T X, u^T X)\, Cov(V^T y, V^T y)}}$$

$$= \frac{E(u^T (X - E(X))(y - E(Y))^T V)}{\sqrt{u^T E(XX^T) u V^T E(yy^T) V}}$$

$$= \frac{u^T E(Xy^T) V}{\sqrt{u^T E(XX^T) u V^T E(yy^T) V}}$$

$$E(Xy^T) = \frac{1}{n}\sum_{i=1}^{n} x_i y_i^T = \frac{1}{n} X^T y$$

$$E(XX^T) = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^T = \frac{1}{n} X^T X$$

$$E(yy^T) = \frac{1}{n}\sum_{i=1}^{n} y_i y_i^T = \frac{1}{n} y^T y$$

$$= \frac{\frac{1}{n} u^T X^T y V}{\sqrt{\frac{1}{n^2} u^T X^T X u V^T y^T y V}} = \frac{u^T X^T y V}{\sqrt{u^T X^T X u V^T y^T y V}}$$

(c) After scaling the features of matrix X to have values between -1 and 1, this will not change the correlation coefficient. Because: $\rho(u^T a X, V^T y)$

$$= \frac{a\, u^T E(Xy^T) V}{\sqrt{a^2 u^T E(XX^T) u V^T E(yy^T) V}} = \frac{u^T E(Xy^T) V}{\sqrt{u^T E(XX^T) u V^T E(yy^T) V}}$$

$$= \rho(u^T X, V^T y)$$

5. Total Least Squares

(a) $\because \operatorname{rank}(X + \mathcal{E}_X) = d$

$$\therefore X + \mathcal{E}_X = U \begin{bmatrix} s_1 & & & & \\ & \ddots & & & \\ & & s_d & & \\ & & & 0 & \\ & & & & \ddots \end{bmatrix} V'$$

$$\because y + \mathcal{E}_y = (X + \mathcal{E}_X) w = U \begin{bmatrix} s_1 & & & & \\ & \ddots & & & \\ & & s_d & & \\ & & & 0 & \\ & & & & \ddots \end{bmatrix} V' w = \begin{bmatrix} y_1 \\ \vdots \\ y_d \\ 0 \\ \vdots \end{bmatrix}$$

$$\therefore [X + \mathcal{E}_X, y + \mathcal{E}_y] = \left(U \begin{bmatrix} s_1 & & & & \\ & \ddots & & & \\ & & s_d & & \\ & & & 0 & \\ & & & & \ddots \end{bmatrix} V' , \begin{bmatrix} y_1 \\ \vdots \\ y_d \\ 0 \\ \vdots \end{bmatrix} \right)$$

This will be the null space of $[X + \mathcal{E}_X, y + \mathcal{E}_y]$
And the first d rows will still be linearly independent, with the addition of only one column.

$\therefore \operatorname{rank}([X + \mathcal{E}_X, y + \mathcal{E}_y]) = d$

(b) $[X, y]_{n \times (d+1)} = \left[\begin{array}{c|c} U_{xx} & U_{xy} \\ \hline U_{yx}^T & U_{yy} \end{array}\right] \begin{bmatrix} \Sigma_d & \\ & \sigma_{d+1} \end{bmatrix} \left[\begin{array}{c|c} V_{xx} & V_{xy} \\ \hline V_{yx}^T & V_{yy} \end{array}\right]^T$

$n \times (d+1) \quad (d+1)(d+1) \quad (d+1)(d+1)$

$$= \begin{bmatrix} U_{xx}\Sigma_d & U_{xy}\sigma_{d+1} \\ U_{yx}^T\Sigma_d & U_{yy}\sigma_{d+1} \end{bmatrix} \begin{bmatrix} V_{xx}^T & V_{yx} \\ V_{xy}^T & V_{yy} \end{bmatrix}$$

$$[X + \mathcal{E}_X, y + \mathcal{E}_y] = \left[\begin{array}{c|c} U_{xx} & U_{xy} \\ \hline U_{yx}^T & U_{yy} \end{array}\right] \begin{bmatrix} \Sigma_d & \\ & 0 \end{bmatrix} \left[\begin{array}{c|c} V_{xx} & V_{xy} \\ \hline V_{yx}^T & V_{yy} \end{array}\right]^T$$

$$= \begin{bmatrix} U_{xx}\Sigma_d & 0 \\ U_{yx}^T\Sigma_d & 0 \end{bmatrix} \begin{bmatrix} V_{xx}^T & V_{yx} \\ V_{xy}^T & V_{yy} \end{bmatrix}$$

$$[X,\ Y] = \begin{pmatrix} U_{xx}\Sigma_d V_{xx}^T + U_{xy}\sigma_{d+1}V_{xy}^T & U_{xx}\Sigma_d V_{yx} + U_{xy}\sigma_{d+1}V_{yy} \\ U_{yx}^T\Sigma_d V_{xx}^T + U_{yy}\sigma_{d+1}V_{xy}^T & U_{yx}^T\Sigma_d V_{yx} + U_{yy}\sigma_{d+1}V_{yy} \end{pmatrix}$$

$$[X+\varepsilon_x,\ Y+\varepsilon_y) = \begin{pmatrix} U_{xx}\Sigma_d V_{xx}^T & U_{xx}\Sigma_d V_{yx} \\ U_{yx}^T\Sigma_d V_{xx}^T & U_{yx}^T\Sigma_d V_{yx} \end{pmatrix}$$

$$\therefore [\varepsilon_x,\ \varepsilon_y] = \begin{pmatrix} -U_{xy}\sigma_{d+1}V_{xy}^T & -U_{xy}\sigma_{d+1}V_{yy} \\ -U_{yy}\sigma_{d+1}V_{xy}^T & -U_{yy}\sigma_{d+1}V_{yy} \end{pmatrix}$$

$$= -\begin{pmatrix} U_{xy} \\ U_{yy} \end{pmatrix}\sigma_{d+1}\begin{pmatrix} V_{xy} \\ V_{yy} \end{pmatrix}^T$$

(c) $[X+\varepsilon_x,\ Y+\varepsilon_y]\begin{pmatrix} w \\ -1 \end{pmatrix} = 0$

$$[X+\varepsilon_x\ \ Y+\varepsilon_y]\begin{pmatrix} w \\ -1 \end{pmatrix} = \begin{pmatrix} U_{xx}\Sigma_d V_{xx}^T & U_{xx}\Sigma_d V_{yx} \\ U_{yx}\Sigma_d V_{xx}^T & U_{yx}^T\Sigma_d V_{yx} \end{pmatrix}\begin{pmatrix} w \\ -1 \end{pmatrix}$$

$$= \begin{pmatrix} U_{xx}\Sigma_d V_{xx}^T w - U_{xx}\Sigma_d V_{yx} \\ U_{yx}\Sigma_d V_{xx}^T w - U_{yx}^T\Sigma_d V_{yx} \end{pmatrix} = 0$$

$$\therefore \begin{cases} U_{xx}\Sigma_d V_{xx}^T w = U_{xx}\Sigma_d V_{yx} \\ U_{yx}^T\Sigma_d V_{xx}^T w = U_{yx}^T\Sigma_d V_{yx} \end{cases}$$

$$w = (V_{xx}^T)^{-1}V_{yx}$$

(d) $\therefore \begin{pmatrix} w \\ -1 \end{pmatrix}$ is a right singular vector of $[X,\ Y]$ with singular value σ_{d+1}

$\therefore \begin{pmatrix} w \\ -1 \end{pmatrix}$ is an eigenvector of $[X,\ Y]^T[X,\ Y]$ with eigenvalue σ_{d+1}^2

$$\therefore [X, Y]^T [X, Y] \begin{pmatrix} w \\ -1 \end{pmatrix} = \sigma_{d+1}^2 \begin{pmatrix} w \\ -1 \end{pmatrix}$$

$$\begin{pmatrix} X^TX & X^TY \\ Y^TX & Y^TY \end{pmatrix} \begin{pmatrix} w \\ -1 \end{pmatrix} = \sigma_{d+1}^2 \begin{pmatrix} w \\ -1 \end{pmatrix}$$

$$X^TX\, w - X^TY = \sigma_{d+1}^2 w$$

$$(X^TX - \sigma_{d+1}^2 I)\, w = X^TY$$

(e) The code, coefficients and plot are shown in the appendix

(f) The code, coefficients and plot are shown in the appendix

(g) The code, coefficients and plot are shown in the appendix. The scale factor I use is 384.

Appendix

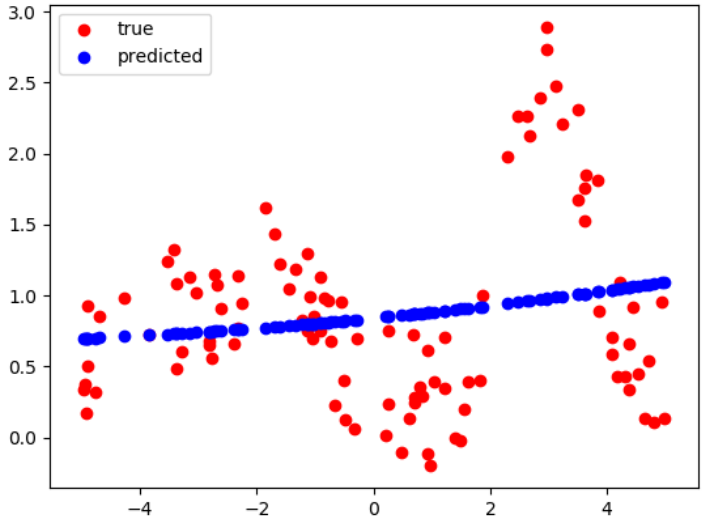
2.

(a)

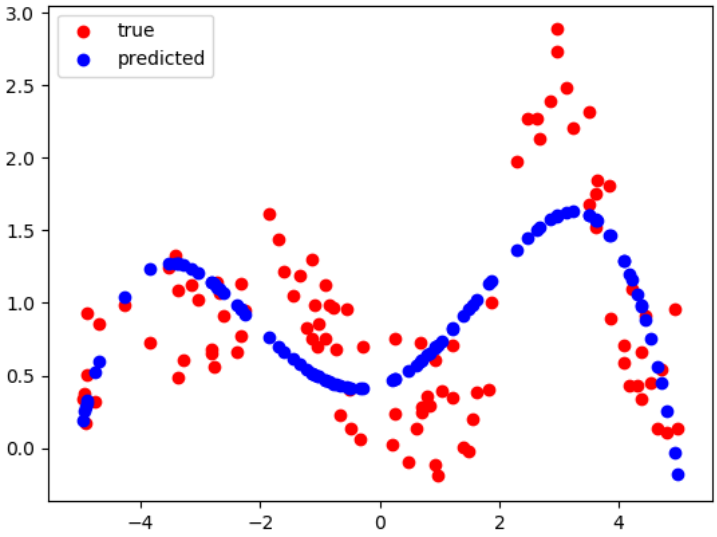
Output

```
Least squares error on original data: 116.87137621050759
Least squares error on augmented data with degree 2: 43.305010550982146
Least squares error on augmented data with degree 4: 26.083682474640515
Least squares error on augmented data wiht degree 6: 18.52122915109212
```

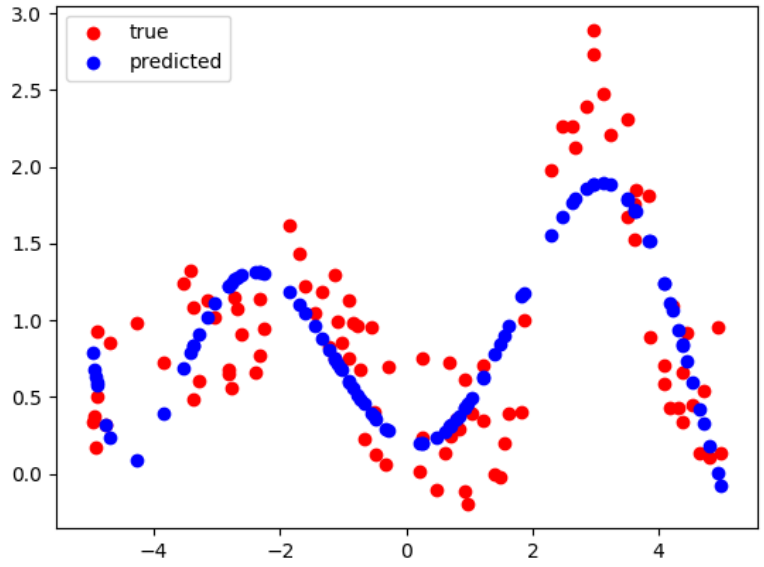
Degree 2 plot:



Degree 4 plot:



Degree 6 plot:

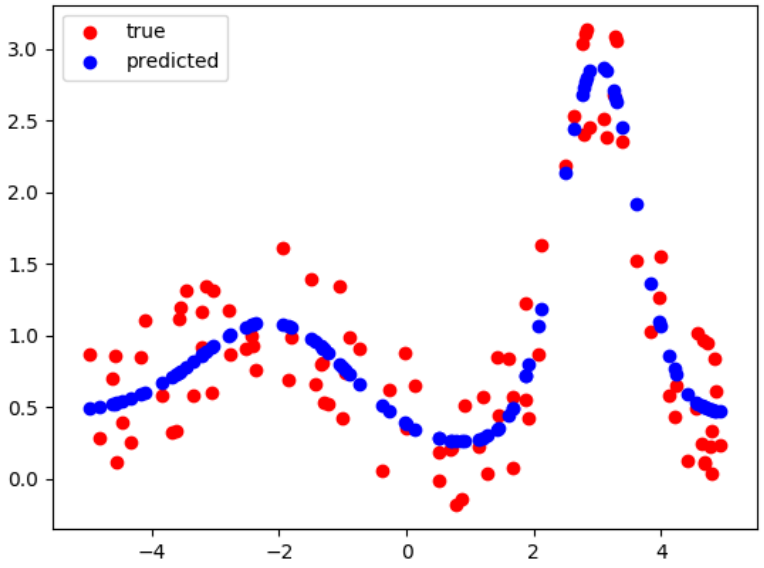


(b)

Output:

```
Least squares error on augmented data with Gaussian function: 8.705852156559082
```

Plot:



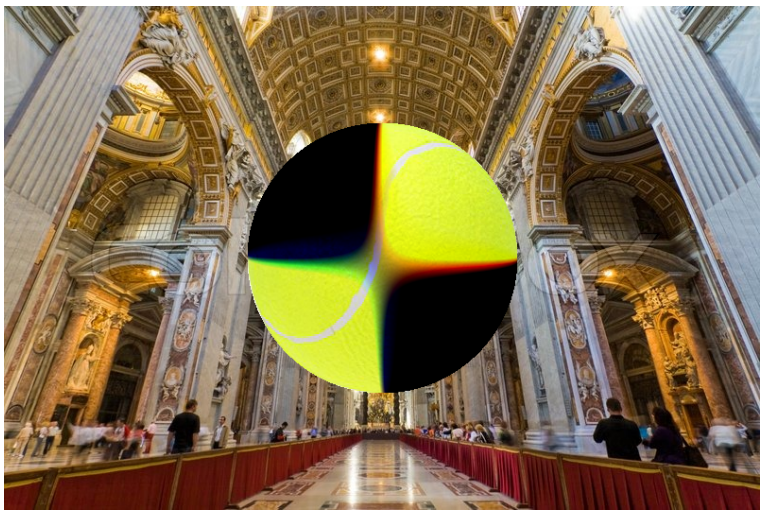
5.

(e)

```
Coefficients:
[[202.31845431 162.41956802 149.07075034]
 [-27.66555164 -17.88905339 -12.92356688]
 [ -5.15203925  -4.51375871  -4.24262639]
 [ -1.08629293   0.42947012   1.15475569]
 [ -3.14053107  -3.70269907  -3.74382934]
 [ 23.67671768  23.15698002  21.94638397]
 [ -3.82167171   0.57606634   1.81637483]
 [  4.7346737    1.4677692   -1.12253649]
 [ -9.72739616  -5.75691108  -4.8395598 ]]
```

(f)

```
Coefficients:
[[ 2.13318421e+02  1.70780299e+02  1.57126297e+02]
 [-3.23046362e+01 -2.02975310e+01 -1.45516114e+01]
 [-4.31689131e+00 -3.80778081e+00 -4.83616306e+00]
 [-4.89811386e+00 -3.37684058e+00 -1.14207091e+00]
 [-7.05901066e+03 -7.39934207e+03 -4.26448732e+03]
 [-3.05378224e+02 -1.56329401e+02  3.50285345e+02]
 [-9.76079364e+00 -5.33182216e+00 -1.55699782e+00]
 [ 7.30792588e+02  3.52130316e+02 -6.11683200e+02]
 [-9.08887079e+00 -3.84309477e+00 -4.16456437e+00]]
```



(g)

```
Coefficients:
[[209.38212459 169.03666402 155.36677288]
 [-30.26805402 -20.30443706 -15.20472049]
 [ -5.753416    -5.07881542  -4.78144904]
 [ -1.05630713   0.46377951   1.19195587]
 [ -7.90569522  -8.20316831  -8.05137623]
 [ 54.96251667  52.62398401  50.09265545]
 [ -3.8491927    0.55663535   1.80236903]
 [  7.32655583   3.83064183   1.07500107]
 [-10.90665749  -6.8522162   -5.87526417]]
```

```
"""Do not modify the functions in this file except for where it says
YOUR CODE HERE."""

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from sklearn.preprocessing import PolynomialFeatures


def polynomial_augmentation(X, degree):
    """PART A

    This function takes in the data X and returns the polynomial features
    for X. You should implement constant/bias terms in your augmentations.

    The input shape of X is (100, 1), and the output shape should be
    (100, degree + 1). The first column of X should be the bias column (all
     1s),
    the second should be a copy of X, the third should be X with all its
     values
    squared, etc.
    """

    """YOUR CODE HERE"""
    poly = PolynomialFeatures(degree)
    augmented_matrix = poly.fit_transform(X)
    return augmented_matrix
    """YOUR CODE ENDS"""


def rbf_augmentation(X, means, variances):
    """PART B

    This function takes in the data X and returns the rbf features with
     respect to
    any centers you provide (write this in the function). We recommend
     looking at the
    data and estimating three centers and three corresponding variances.
     Once you have
    those, evaluate X's data with those three rbf functions, and augment
     the matrix with
    those outputs. You should thus have one column for bias, one for the
     original data X,
    and three for the rbf (mean, variance) pairs you come up with (X is 100
     by 5).

    The input shape of X is (100, 1), and the output shape should be (100,
     #(centers) + 2),
    since you want a column of all ones for the bias and one with the
     original data.

    means is a list of 3 means, variances is a list of 3 corresponding
     variances.
    """
    """YOUR CODE HERE"""
```

```
    n = X.shape[0]
    rbf_matrix = np.copy(X)
    for i in range(len(means)):
        mean = means[i]
        var = variances[i]
        rbf = np.exp(-((X - mean)**2)/(2*var))
        rbf_matrix = np.append(rbf_matrix,rbf, axis=1)
    return rbf_matrix
    """YOUR CODE ENDS"""

def linear_regression(X, y):
    """Return the weights from linear regression.

    X: nxd (d = 1) matrix of data organized in rows
    y: length n vector of labels
    """
    return np.linalg.inv(X.T@X)@X.T@y


def mean_squared_error(y, y_hat):
    """Calculate the mean squared error given truth and predicted labels.

    y: length n vector of true labels
    y_hat: length n vector of predicted labels
    """
    return np.linalg.norm(y - y_hat) ** 2


def plot_compare(X, y, y_hat, figname):
    """Plot both true and predicted values for data.

    X: nxd (d = 1) matrix of data organized in rows
    y: length n vector of labels
    y_hat: same, but the predicted labels
    figname: name of the figure to save as (string))
    """
    # use blue for predicted, red for ground truth
    fig = plt.figure()
    ax1 = fig.add_subplot(111)

    ax1.scatter(X, y, c='r')
    ax1.scatter(X, y_hat, c='b')
    plt.legend(loc='upper left', labels=['true', 'predicted'])
    plt.savefig(figname)


def generate_data():
    """Generate data with noise."""
    X = np.random.rand(100) * 10 - 5
    y = 2.5 * norm.pdf(X, -2, 1.2) - 1.5 * norm.pdf(X, 0, 2) + 3.7 *
     norm.pdf(X, 3, 0.6)
    # inject noise
    y += np.random.rand(100)
    X = np.expand_dims(X, axis=0)
    return X.T, y
```

```python
import numpy as np
from lib import *


def part_a():
    # get the data
    X, y = generate_data()

    # perform linear regression
    weights = linear_regression(X, y)
    # measure error
    y_pred = X @ weights
    error = mean_squared_error(y, y_pred)
    print("Least squares error on original data:", error)
    # plot, remember to use original X to plot even if you augment
    plot_compare(X, y, y_pred, figname="ols_original.png")


    """YOUR CODE HERE"""
    # PART A: Implement the augmenting function in lib and
    # follow a similar procedure as above to plot the regression
    # for different degrees of polynomials as specified in the problem.
    augmented_X_2 = polynomial_augmentation(X, 2)
    augmented_X_4 = polynomial_augmentation(X, 4)
    augmented_X_6 = polynomial_augmentation(X, 6)
    weights_2 = linear_regression(augmented_X_2, y)
    weights_4 = linear_regression(augmented_X_4, y)
    weights_6 = linear_regression(augmented_X_6, y)
    y_pred_2 = augmented_X_2 @ weights_2
    y_pred_4 = augmented_X_4 @ weights_4
    y_pred_6 = augmented_X_6 @ weights_6
    error_2 = mean_squared_error(y, y_pred_2)
    error_4 = mean_squared_error(y, y_pred_4)
    error_6 = mean_squared_error(y, y_pred_6)
    print("Least squares error on augmented data with degree 2:", error_2)
    print("Least squares error on augmented data with degree 4:", error_4)
    print("Least squares error on augmented data wiht degree 6:", error_6)
    plot_compare(X, y, y_pred_2, figname="ols_augmented_2.png")
    plot_compare(X, y, y_pred_4, figname="ols_augmented_4.png")
    plot_compare(X, y, y_pred_6, figname="ols_augmented_6.png")
    """YOUR CODE ENDS"""


def part_b():
    X, y = generate_data()
    """YOUR CODE HERE"""
    # PART B: Implement the rbf augmenting function in lib and then follow
    # the same procedure as before to produce a graph of the fitted data.

    # decide on three means and variances that best represent the data
    # (look at the plot from part a to see gaussian shapes!)
    means = [-2, 1, 3]
    variances = [3, 2, 0.5]
    X_rbf = rbf_augmentation(X, means, variances)
    weights = linear_regression(X_rbf, y)
```

```
    y_pred = X_rbf @ weights
    error = mean_squared_error(y, y_pred)
    print("Least squares error on augmented data with Gaussian function:",
     error)
    plot_compare(X, y, y_pred, figname="ols_augmented_guassian.png")
    """YOUR CODE ENDS"""


def main():
    part = input('Which part to run?\n')
    if part == 'a':
        part_a()
    elif part == 'b':
        part_b()
    else:
        print("Invalid part.")
        exit(1)

if __name__ == "__main__":
    main()
```

```
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)
```