HW4 - ML

Q.7) $X \sim N(\mu, \Sigma)$

$$p(X) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

$$P(X_1, X_2 \ldots X_n) = \prod_{i=1}^{n} \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(X_i-\mu)^T \Sigma^{-1}(X_i-\mu)}$$

$$\ln(P(X_1, X_2, \ldots X_n | \Sigma \mu)) = \ln \prod_{i=1}^{n} \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(X_i-\mu)^T \Sigma^{-1}(X_i-\mu)}$$

$$= \sum_{i=1}^{n} \ln \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(X_i-\mu)^T \Sigma^{-1}(X_i-\mu)}$$

$$= -\frac{n}{2} \ln \det(2\pi\Sigma) - \frac{1}{2}\sum_{i=1}^{n} (X_i-\mu)^T \Sigma^{-1}(X_i-\mu)$$

$$= -\frac{n}{2} \ln \det(2\pi\Sigma) - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{d} \frac{1}{\sigma_j^2}(X_{ij} - \mu_j)^2$$

(b) MLE is the μ and Σ

$$= -\frac{n}{2} \ln \det(\Sigma^{-1}) - \frac{1}{2}\sum_{i=1}^{n} tr[(X_i-\mu_i)(X_i-\mu_i)^T \Sigma^{-1}]$$

$$\frac{\partial \ln P(X_1, X_2 \ldots X_n | \Sigma, \mu)}{d\Sigma}$$

$$= -\frac{n}{2} Tr(\Sigma^{-1} \partial \Sigma) - \frac{1}{2}\sum_{i=1}^{n} \partial (X_i - \mu)^T \times \Sigma^{-1}(X_i-\mu) + (X_i-\mu)^T \partial \Sigma^{-1}(X_i-\mu)$$

$$= -\frac{n}{2} Tr\left(\Sigma^{-1}\begin{bmatrix} 2\sigma_1 & & \\ & 2\sigma_2 & \\ & & \ddots \\ & & & 2\sigma_d \end{bmatrix}\right) - \frac{1}{2}\sum_{i=1}^{n} (X_i - \mu_i)^T \begin{pmatrix} \frac{X_{i1}-\mu_1}{\sigma_1^3} \\ \frac{X_{i2}-\mu_2}{\sigma_2^3} \\ \vdots \\ \frac{X_{id}-\mu_d}{\sigma_d^3} \end{pmatrix} \times (-2)$$

$$= -\frac{n}{2} Tr\left(2/\sigma_1 \quad 2/\sigma_2 \quad \ldots \quad 2/\sigma_d\right) + \sum_{i=1}^{n} (X_i-\mu_i)^T \Sigma^{3/2}(X_i-\mu_i)$$

$$= -\frac{n}{2}\prod_{i=1}^{d} \frac{2}{\sigma_i} + \sum_{i=1}^{n} (X_i - \mu_i)^T \Sigma^{-3/2}(X_i - \mu_i)$$

(Pr 2b) $$\frac{\partial \ln P(X_1, X_2 \ldots X_n \mid \Sigma, M)}{\partial \Sigma^{-1}} = \frac{N}{2}\Sigma - \frac{1}{2}\sum_{i=1}^{n}(X_i - M_i)(X_i - M_i)^T$$

Setting the derivative to zero, we get

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}(x_i - M_i)(x_i - M_i)^T$$

$$\frac{\partial \ln P(X_1, X_2 \ldots X_n \mid \Sigma, M)}{\partial M} = \sum_{i=1}^{n}(X_i - M)^T \Sigma^{-1}$$

Setting the derivative to zero yields.

$$M = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Thus, $$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\left(X_i - \frac{1}{n}\sum_{i=1}^{n} x_i\right)\left(X_i - \frac{1}{n}\sum_{i=1}^{n} x_i\right)^T$$

Qn 3) (a) $Y | X, W \sim N(W^T X, 1)$

$$p(Y|X,W) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y - W^T X)^2}$$

(b) $$P(W | X, Y) = \frac{p(Y|X,W)\, p(W|X)}{\int_W P(Y|X,W)\, P(W|X)\, dW}$$

$$p(W|X) = p(W)$$
$$= \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(W-0)^T \Sigma^{-1} (W-0)}$$

$$= \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2} W^T \Sigma^{-1} W \}}$$

$$p(W|X,Y_i) = P(y_1, y_2 .. y_n | X, W) \cdot P(W)$$ (Assuming y is inde independent)

$$\propto \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y_i - W^T X_i)^2} \cdot \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2} W^T \Sigma^{-1} W}$$

$$\propto \left(\frac{1}{\sqrt{2\pi}}\right)^n \cdot \frac{1}{\sqrt{\det(2\pi\Sigma)}} \cdot e^{-\frac{1}{2}\sum_{i=1}^{n}(y_i - W^T X_i)^2} \cdot e^{-\frac{1}{2} W^T \Sigma^{-1} W}$$

$$\propto \left(\frac{1}{\sqrt{2\pi}}\right)^{n+d} \frac{1}{\sqrt{\det(\Sigma)}} e^{-\frac{1}{2}\left[\sum_{i=1}^{n}(y_i - W^T X_i)^2 + W^T \Sigma^{-1} W\right]}$$

$$\propto \left(\frac{1}{\sqrt{2\pi}}\right)^{n+d} \frac{1}{\sqrt{\det(\Sigma)}} e^{-\frac{1}{2}\left(\sum_{i=1}^{n} y_i^2 + \sum_{i=1}^{n}(W^T X_i)^2 - \sum_{i=1}^{n} 2 y_i W^T X_i\right)} \cdot e^{-\frac{1}{2} W^T \Sigma^{-1} W}$$

$$\propto \exp\left\{-0.5\left(\sum_{i=1}^{n} Y_i^2 - 2 \underbrace{\sum_{i=1}^{n} (X_i Y_i)^T}_{(X^T Y)^T} W + W^T \underbrace{\sum_{i=1}^{n} X_i X_i^T}_{X^T X} W\right)\right\} \exp\{\tfrac{1}{2} W^T \Sigma^{-1} W\}$$

let $X = \begin{bmatrix} X_1^T \\ X_2^T \\ \vdots \\ X_n^T \end{bmatrix}$, $Y = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix}$, Then $\sum_{i=1}^{n} X_i Y_i = X^T Y$ and $\sum_{i=1}^{n} X_i X_i^T = X^T X$

3) (b) $\propto \exp\left\{-\frac{1}{2}\left(-2(X^TY)^Tw + w^T(X^TX)w + w^T\Sigma^{-1}w\right)\right\}$

$\propto \exp\left\{ (X^TY)^Tw - \frac{1}{2} w^T(X^TX + \Sigma^{-1})w\right\}$

$p(w|X,Y) = C \cdot \exp\left\{(X^TY)^Tw - \frac{1}{2} w^T(X^TX + \Sigma^{-1})w\right\}$

where C is a constant, with $M = (X^TX + \Sigma^{-1})$,

we have

$p(w|X,Y) = C \cdot \exp\left\{-\frac{1}{2}\left[(w - M^{-1}(X^TY))^T M (w - M^{-1}(X^TY))\right] + \text{constant}\right\}$

Thus mean is $M^{-1}(X^TY)$ and variance is

M

3) (c)

Qn 4) (a) Since $(A + \hat{A})\vec{w} = \vec{y} + \hat{\vec{y}}$

$$\left(\begin{bmatrix} A_1 + \hat{A}_1 \end{bmatrix} \begin{bmatrix} A_2 + \hat{A}_2 \end{bmatrix} \cdots \begin{bmatrix} A_d + \hat{A}_d \end{bmatrix} \right) \times \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{pmatrix}$$

where $\begin{bmatrix} A_i + \hat{A}_i \end{bmatrix}$ represent the column vectors of $(A + \hat{A})$

Thus, $\vec{y} + \hat{\vec{y}} = w_1 \begin{bmatrix} A_1 + \hat{A}_1 \end{bmatrix} + w_2 \begin{bmatrix} A_2 + A_2 \end{bmatrix} + \cdots w_d \begin{bmatrix} A_d + \hat{A}_d \end{bmatrix}$

~~Thus~~ In other words, $\vec{y} + \hat{\vec{y}}$ is a linear combination of the column vectors of $(A + \hat{A})$. Therefore, adding a column of $(\vec{y} + \hat{\vec{y}})$ to $(A + \hat{A})$ cannot increase its rank.

$\therefore \text{rank}([A + \hat{A}, \vec{y} + \hat{\vec{y}}]) = d$

(b) $[A + \hat{A}, \vec{y} + \hat{\vec{y}}]\vec{x} = \vec{0}$

$$U \begin{bmatrix} \Sigma_{1,\dots d} & \\ & 0 \end{bmatrix} V^T \vec{x} = \vec{0}$$

$$U \begin{bmatrix} \lambda_1 & & & & & \\ & \lambda_2 & & & \vec{0} & \\ & & \ddots & & & \\ \vec{0} & & & \lambda_d & & \\ & & & & 0 & \\ & & & & & 0 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \\ \vdots \\ V_n^T \end{bmatrix} \vec{x} = \vec{0}$$

$$U \begin{bmatrix} \lambda_1 V_1^T \vec{x} \\ \lambda_2 V_2^T \vec{x} \\ \vdots \\ \lambda_d V_d^T \vec{x} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0}$$

U is an orthonormal matrix, thus, U^{-1} exist. So, Qn 4) (b)

$$\lambda_1 \vec{V_1}^T \vec{x} = 0$$
$$\lambda_2 \vec{V_2}^T \vec{x} = 0$$
$$\vdots$$
$$\lambda_d \vec{V_d}^T \vec{x} = 0$$

Thus since $\vec{V_{d+1}}$ is orthogonal to $\vec{V_1}, \vec{V_2}, \vec{V_3} \dots \vec{V_d}$, $\vec{x} = \alpha \vec{V_{d+1}}$ will be a solution to all the equations above where $\alpha \in R$.

sigm

(e)

Qn 5) Given $f(\vec{x}) = x_1 x_2 + x_3^3$, where $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$, ~~comp~~ and $A = \begin{bmatrix} x_1 & x_2 \\ x_2 & x_3 \end{bmatrix}$,

compute $\dfrac{\partial f(\vec{x})}{\partial A}$

$$\frac{\partial f(\vec{x})}{\partial A} = \begin{bmatrix} x_2 & x_1 \\ x_1 & 3x_3^2 \end{bmatrix}$$

Done by : Nathanael Raj

Qn 1) (a) Xin Chen xinchen.zhu@berkeley.edu
Jun Yu phangjunyu@berkeley.edu

(b) I certify that all solutions are entirely in my words and I have not looked at another student's solutions. I have credited all external sources in this write up.

Nathanael Raj

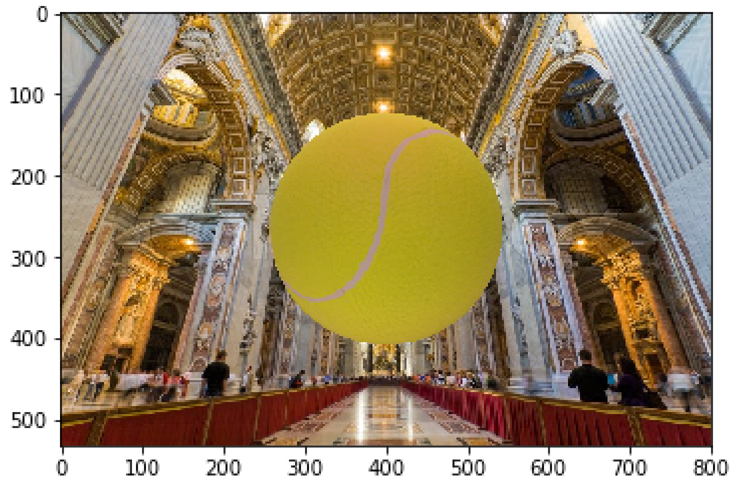
Question 4e

We could rescale it by dividing the vsp (f(n)) values by 384 and multiplying the coefficients of the TLS by 384

Coefficients=

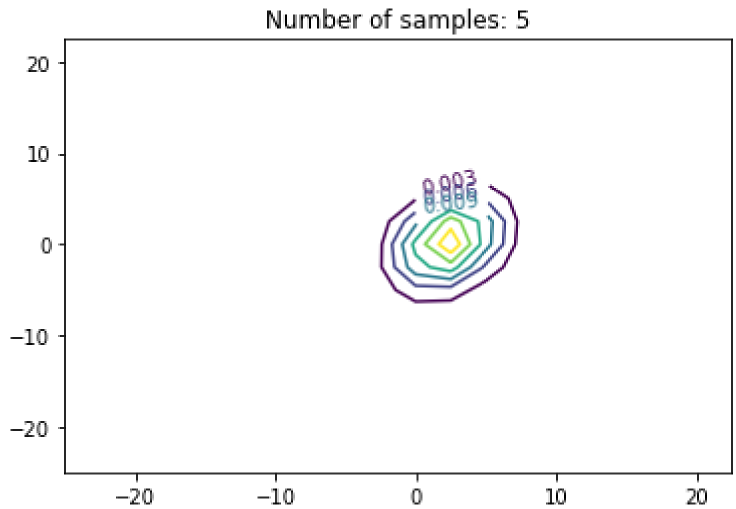
```
[[ 209.38212459  169.03666402  155.36677288]
 [ -30.26805402  -20.30443706  -15.20472049]
 [  -5.753416     -5.07881542   -4.78144904]
 [  -1.05630713    0.46377951    1.19195587]
 [  -7.90569522   -8.20316831   -8.05137623]
 [  54.96251667   52.62398401   50.09265545]
 [  -3.8491927     0.55663535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

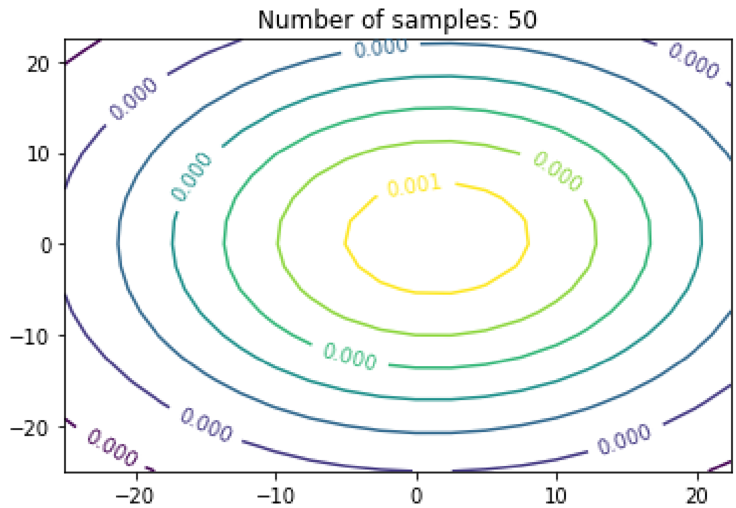
The relit sphere:

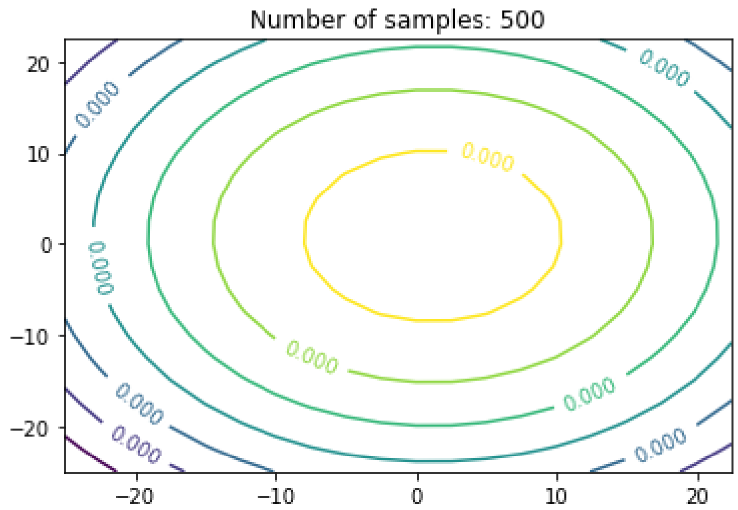


Question 3d

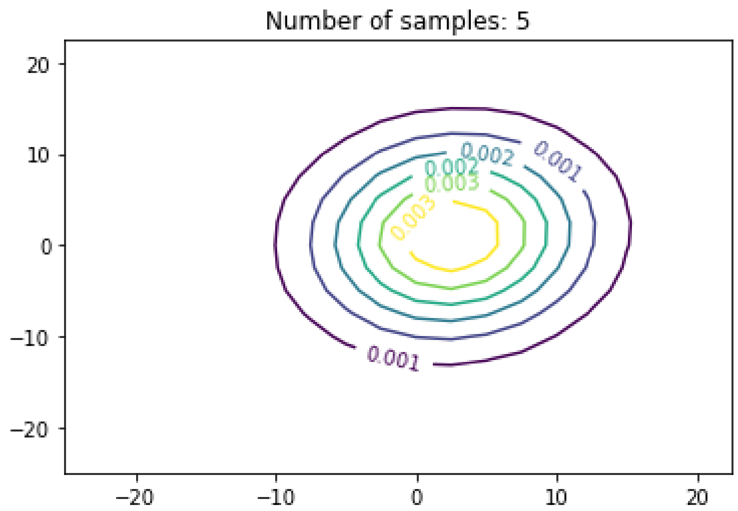
Sigma =
[[1 0]
 [0 1]]

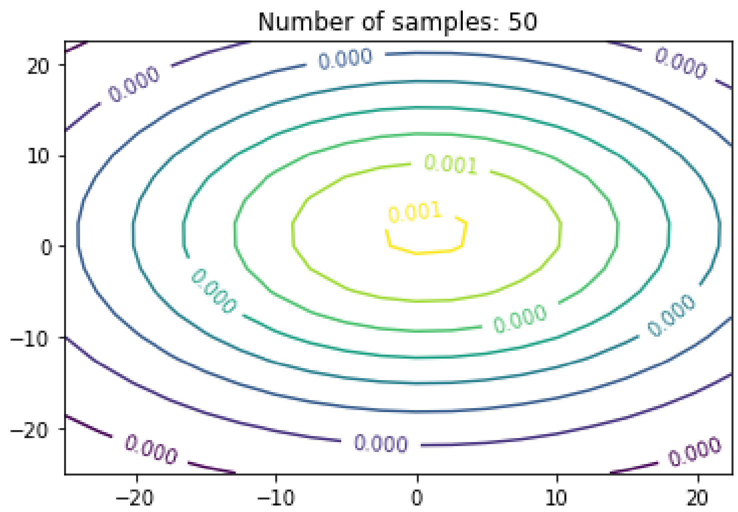


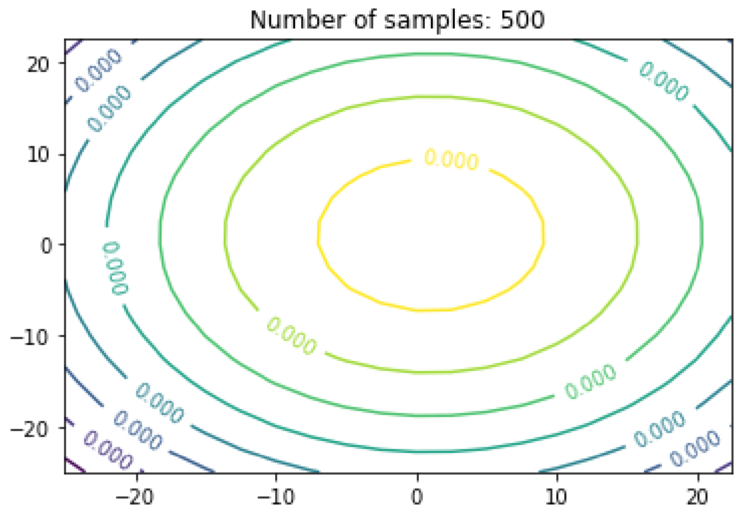




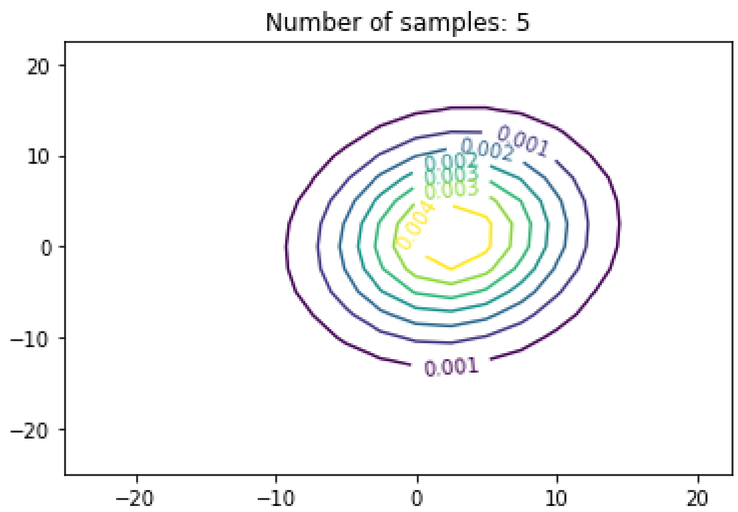
Sigma =
[[1.   0.25]
 [0.25 1.]]

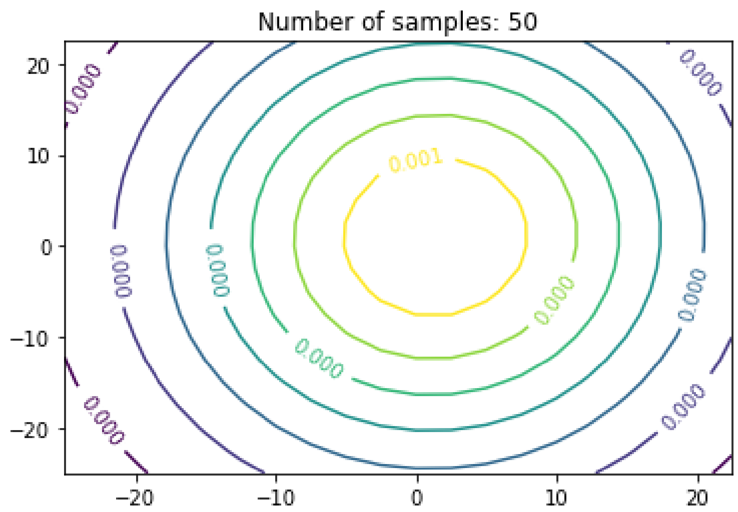


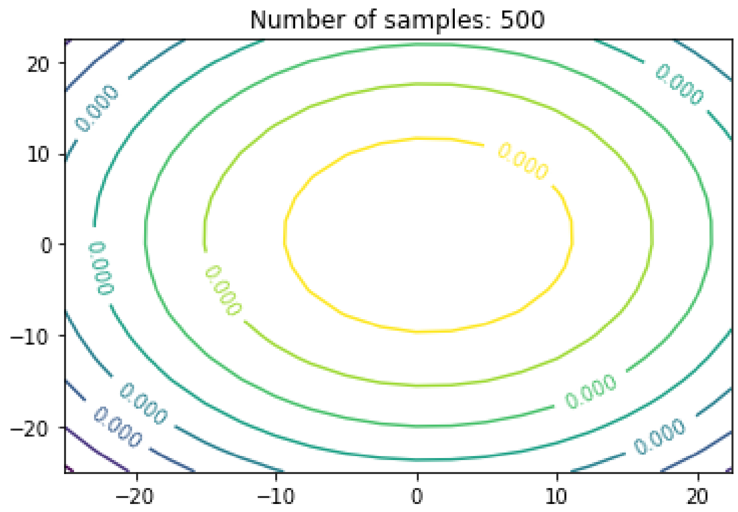




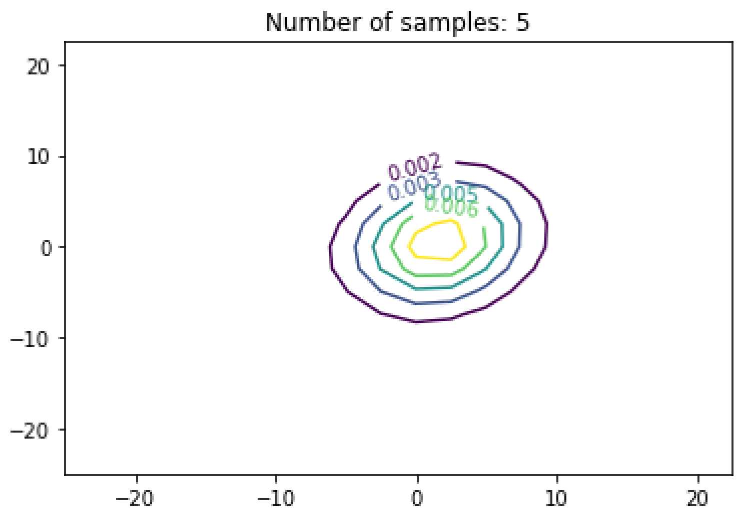
Sigma =
[[1.   0.9]
 [0.9 1.]]

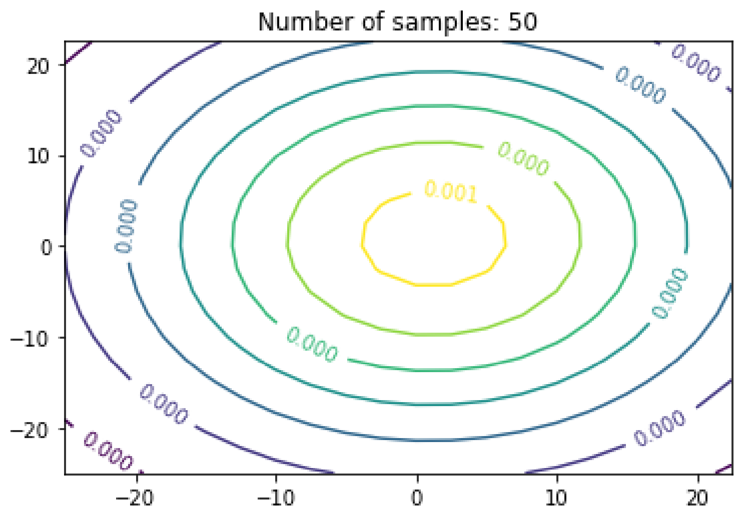


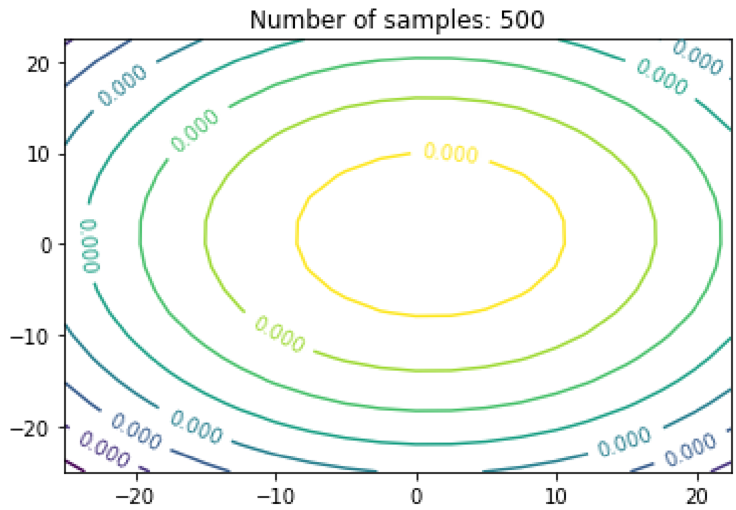




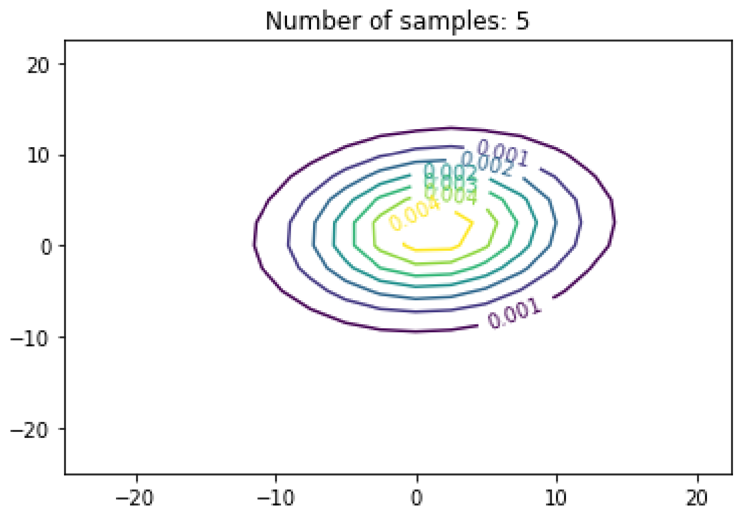
Sigma =
[[1.   -0.25]
 [-0.25 1.]]

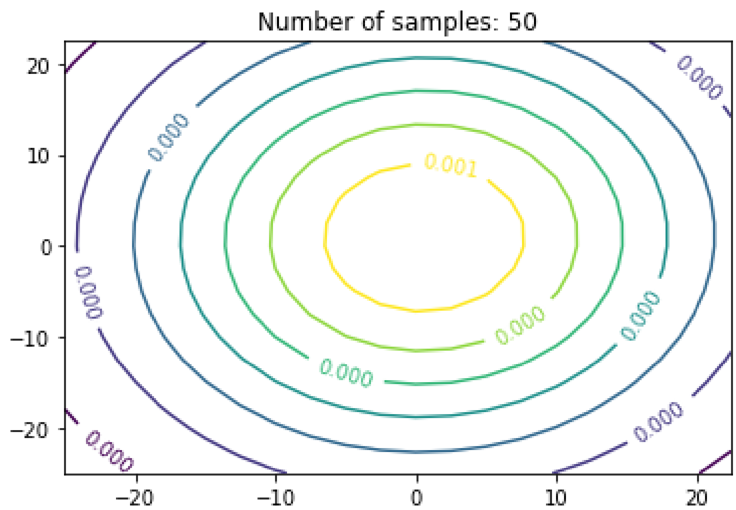


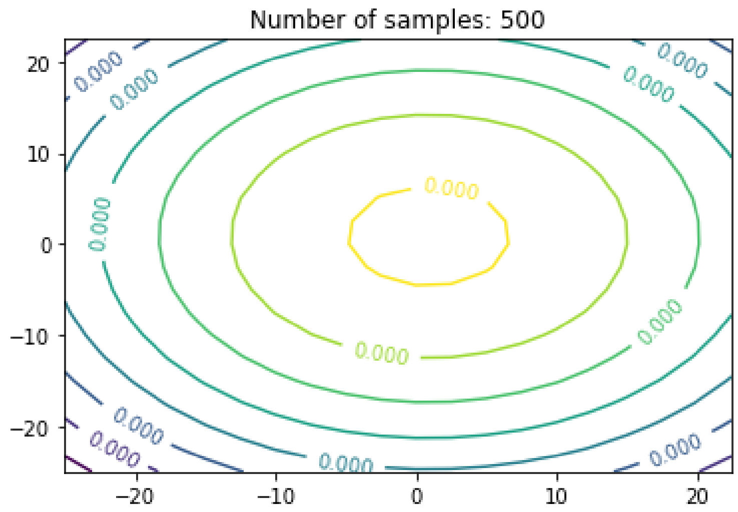




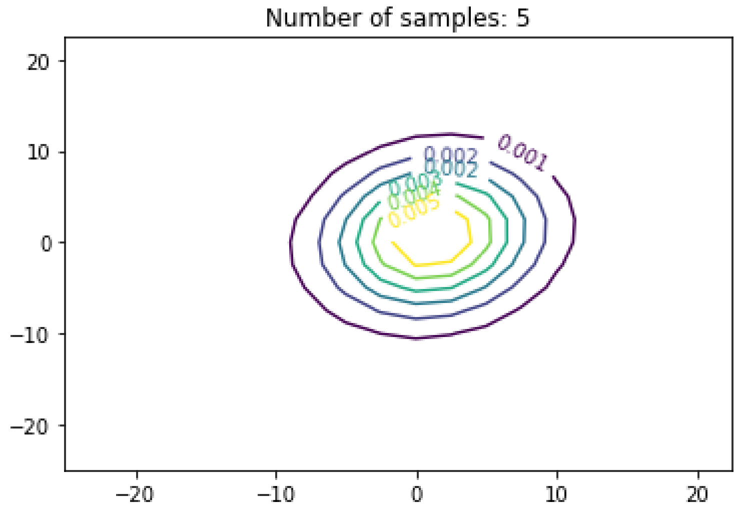
Sigma =
[[1. -0.9]
 [-0.9 1.]]

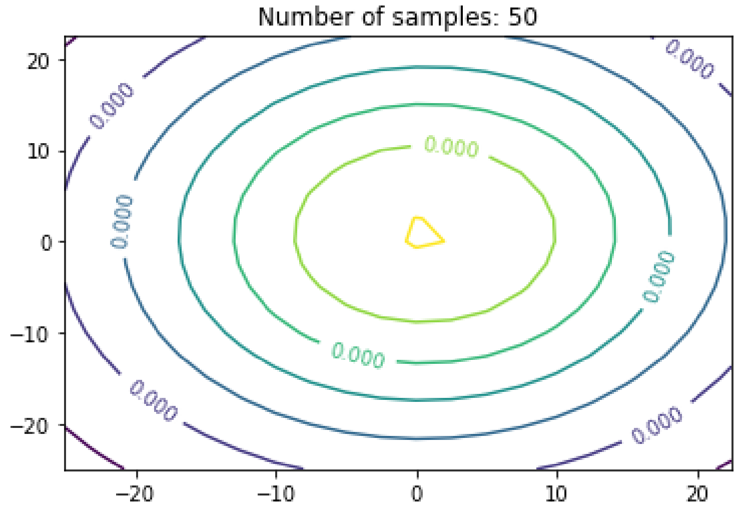


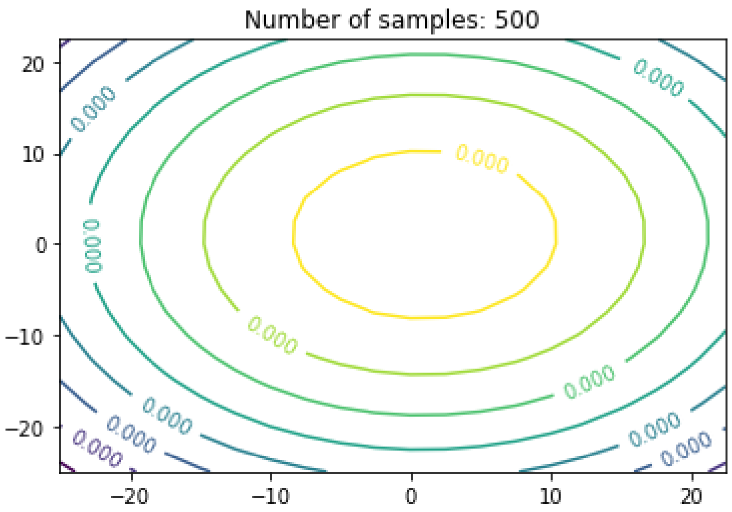




Sigma =
[[0.1 0.]
 [0.   0.1]]







Observation: As the number of training samples increase, the variation of the posterior w increases and this leads to a more spread out distribution. This is because sigma prime was computed using XTranspose * X + sigma which meant that as the size of X increased, the size of the posterior variance would increase also.

Question 2C

```
import numpy as np
import matplotlib.pyplot as plt

def MLE_mu(X):
    n = X.shape[0]
    tot = np.sum(X,0)
    return tot/n

def MLE_sigma(X):
    mu = MLE_mu(X)
    mu = np.matrix(mu).T
    n = X.shape[0]
    tot = np.matrix([[0,0],[0,0]],dtype='float')
    for xi in X:
        m = (xi - mu).T.dot( (xi - mu) )
        tot = tot + m
    return tot/n

sigma_list = [ [[20, 0], [0, 10]], [[20,14],[14,10]],  [[20,-14],[-14,10]] ]
mu = [15, 5]
print ("Sigma \t\t\t\t\t Mu\n")
for sigma in sigma_list:
    samples = np.random.multivariate_normal(mu, sigma, size=100)
    print(MLE_sigma(samples), '\t', MLE_mu(samples), '\n')
```

<u>Output</u>

```
Sigma                                   Mu

[[ 139.96850145   -5.23621416]
 [  -5.23621416  110.37464612]]         [ 14.86386061   5.09534284]

[[ 136.40485397   25.1262464 ]
 [  25.1262464   119.45532448]]         [ 15.3745544    5.31787646]

[[ 131.67869995  -24.50694813]
 [ -24.50694813  114.40296349]]         [ 14.96190147   5.12444096]
```

Question 3d

```
import numpy as np
import matplotlib.mlab as mlab
def gen_data(n):
    X = np.zeros((n,2))
    Z = np.zeros((n,1))
    Y = np.zeros((n,1))
    for i in range (n):
        X[i][0] = np.random.normal(0,2.236068)
        X[i][1] = np.random.normal(0,2.236068)
        Z[i] = np.random.normal(0,5)
        Y[i] = X[i][0] + X[i][1] + Z[i]
    return X, Y
def posterior_w(X, Y, sigma):
    var = X.T.dot(X) + np.linalg.inv(sigma)
    mu = np.linalg.inv(var).dot(X.T).dot(Y)
    return mu, var
import math
import matplotlib.pyplot as plt
def plot(mu, var, n):
    delta = 2.5
    x = np.arange(-25.0, 25.0, delta)
    y = np.arange(-25.0, 25.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, math.sqrt(var[0,0]), math.sqrt(var[1,1]), mu[0,0], mu[1,0],
math.sqrt(abs(var[1,0])))
    #Z1 = mlab.bivariate_normal(X, Y, 1, 1, 0, 0, 0.5)
    plt.figure()
    CS = plt.contour(X, Y, Z1)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.title("Number of samples: "+ str(n))
    plt.show()

sigma = [np.matrix([[1,0],[0,1]]),
         np.matrix([[1,0.25],[0.25,1]]),
         np.matrix([[1,0.9],[0.9,1]]),
         np.matrix([[1,-0.25],[-0.25,1]]),
         np.matrix([[1,-0.9],[-0.9,1]]),
         np.matrix([[0.1,0],[0,0.1]])
        ]
for sigmai in sigma:
    print(sigmai)
```

```
X,Y = gen_data(5)
mu, var = posterior_w(X,Y,sigmai)
plot(mu, var, 5)
X,Y = gen_data(50)
mu, var = posterior_w(X,Y,sigmai)
plot(mu, var, 50)
X,Y = gen_data(500)
mu, var = posterior_w(X,Y,sigmai)
plot(mu, var, 500)
```

Question 4c

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc 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
    B = np.ones((len(ns),9))
    ##################################################
    # Compute the first 9 basis functions
    for i, nsi in enumerate(ns):
        x = nsi[0]
        y = nsi[1]
```

```
        z = nsi[2]
        B[i][0] = 1
        B[i][1] = y
        B[i][2] = x
        B[i][3] = z
        B[i][4] = x*y
        B[i][5] = y*z
        B[i][6] = 3 * z**2 -1
        B[i][7] = x * z
        B[i][8] = x**2 - y**2
    ##################################################
    # This line is here just to fill space

    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
    print(Bp)
    solution = np.linalg.lstsq(Bp, vsp)[0]
    print(solution)
    ###################################################

    coeff = np.zeros((9,3))
    coeff[0,:] = 255
    coeff = solution.reshape(9,3)
    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)
```

Question 4d

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc 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
    B = np.ones((len(ns),9))
    ##################################################
    # Compute the first 9 basis functions
    for i, nsi in enumerate(ns):
        x = nsi[0]
        y = nsi[1]
        z = nsi[2]
        B[i][0] = 1
```

```
        B[i][1] = y
        B[i][2] = x
        B[i][3] = z
        B[i][4] = x*y
        B[i][5] = y*z
        B[i][6] = 3 * z**2 -1
        B[i][7] = x * z
        B[i][8] = x**2 - y**2
    ##################################################
    # This line is here just to fill space

    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
    # Code adapted from: https://en.wikipedia.org/wiki/Total_least_squares
    m, n = Bp.shape
    print(m,n)
    Z = np.hstack((Bp, vsp))
    U,S,V = np.linalg.svd(Z)
    Vxy = V.T[:n,n:]
    Vyy = V.T[n:,n:]
    print(Vyy.shape)
    B = -Vxy.dot(np.linalg.inv(Vyy))
    print(B)

    ###################################################

    coeff = np.zeros((9,3))
    coeff[0,:] = 255
    coeff = B.reshape(9,3)
    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)
```

Question 4e

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc 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
    B = np.ones((len(ns),9))
    ####################################################
    # Compute the first 9 basis functions
    for i, nsi in enumerate(ns):
        x = nsi[0]
```

```
        y = nsi[1]
        z = nsi[2]
        B[i][0] = 1
        B[i][1] = y
        B[i][2] = x
        B[i][3] = z
        B[i][4] = x*y
        B[i][5] = y*z
        B[i][6] = 3 * z**2 -1
        B[i][7] = x * z
        B[i][8] = x**2 - y**2
    ####################################################
    # This line is here just to fill space

    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
    # Code adapted from: https://en.wikipedia.org/wiki/Total_least_squares

    vsp = vsp /384
    m, n = Bp.shape
    print(m,n)
    Z = np.hstack((Bp, vsp))
    U,S,V = np.linalg.svd(Z)
    Vxy = V.T[:n,n:]
    Vyy = V.T[n:,n:]
    print(Vyy.shape)
    B = -Vxy.dot(np.linalg.inv(Vyy))
    print(B)

    #####################################################

    coeff = np.zeros((9,3))
```

```
coeff[0,:] = 255
coeff = B.reshape(9,3) *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)
```