CSE 252B: Computer Vision II, Winter 2019 – Assignment 2

Instructor: Ben Ochoa

Due: Wednesday, February 6, 2019, 11:59 PM

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains both math and programming problems.
- All solutions must be written in this notebook
- Math problems must be done in Markdown/LATEX. Remember to show work and describe your solution.
- Programming aspects of this assignment must be completed using Python in this notebook.
- Your code should be well written with sufficient comments to understand, but there is no need to write extra markdown to describe your solution if it is not explictly asked for.
- This notebook contains skeleton code, which should not be modified (This is important for standardization to facilate effeciant grading).
- You may use python packages for basic linear algebra, but you may not use packages that directly solve the problem. Ask the instructor if in doubt.
- You must submit this notebook exported as a pdf. You must also submit this notebook as an .ipynb file.
- Your code and results should remain inline in the pdf (Do not move your code to an appendix).
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- It is highly recommended that you begin working on this assignment early.

Problem 1 (Math): Line-plane intersection (5 points)

The line in 3D defined by the join of the points $\boldsymbol{X}_1 = (X_1, Y_1, Z_1, T_1)^\top$ and $\boldsymbol{X}_2 = (X_2, Y_2, Z_2, T_2)^\top$ can be represented as a Plucker matrix $\boldsymbol{L} = \boldsymbol{X}_1\boldsymbol{X}_2^\top - \boldsymbol{X}_2\boldsymbol{X}_1^\top$ or pencil of points $\boldsymbol{X}(\lambda) = \lambda\boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ (i.e., $\boldsymbol{X}$ is a function of λ). The line intersects the plane $\boldsymbol{\pi} = (a, b, c, d)^\top$ at the point $\boldsymbol{X}_{\boldsymbol{L}} = \boldsymbol{L}\boldsymbol{\pi}$ or $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$, where $\lambda_{\boldsymbol{\pi}}$ is determined such that $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top\boldsymbol{\pi} = 0$ (i.e., $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ is the point on $\boldsymbol{\pi}$). Show that $\boldsymbol{X}_{\boldsymbol{L}}$ is equal to $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ up to scale.

Answer

By expanding the equation $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top \boldsymbol{\pi}$, we can derive:

$$
\begin{aligned}
&\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})^\top \boldsymbol{\pi} \\
&= (\lambda_{\boldsymbol{\pi}} \boldsymbol{X}_1 + (1-\lambda_{\boldsymbol{\pi}})\boldsymbol{X}_2)^\top \boldsymbol{\pi} \\
&= (\lambda_{\boldsymbol{\pi}}(X_1, Y_1, Z_3, T_1)^T + (1-\lambda_{\boldsymbol{\pi}})(X_2, Y_2, Z_2, T_2)^\top)\boldsymbol{\pi} \\
&= \begin{bmatrix} \lambda_{\boldsymbol{\pi}}(X_1 - X_2) + X_2 \\ \lambda_{\boldsymbol{\pi}}(Y_1 - Y_2) + Y_2 \\ \lambda_{\boldsymbol{\pi}}(Z_1 - Z_2) + Z_2 \\ \lambda_{\boldsymbol{\pi}}(T_1 - T_2) + T_2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \\
&= \lambda_{\boldsymbol{\pi}}[a(X_1 - X_2) + b(Y_1 - Y_2) + c(Z_1 - Z_2) + d(T_1 - T_2)] + aX_2 + bY_2 + cZ_2 + dT_2 \\
&= 0
\end{aligned}
$$

We can then solve $\lambda_{\boldsymbol{\pi}}$, and replace it back to $\boldsymbol{X}(\lambda_\pi)$.

$$\lambda_{\boldsymbol{\pi}} = \frac{-(aX_2+bY_2+cZ_2+dT_2)}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)}$$

$$\boldsymbol{X}(\lambda_\pi) = \frac{-1}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)}[(aX_2 + bY_2 + cZ_2 + dT_2)\boldsymbol{X_1} - (aX_1 + bY_1 + cZ_1 + dT_1)\boldsymbol{X_2}]$$

By expanding $\boldsymbol{X_L}$, we can also derive the below equation.

$$
\begin{aligned}
&\boldsymbol{X_L} = \boldsymbol{L\pi} \\
&= (\boldsymbol{X_1 X_2}^\top - \boldsymbol{X_2 X_1}^\top)\boldsymbol{\pi} \\
&= \boldsymbol{X_1}(\boldsymbol{X_2}^\top \boldsymbol{\pi}) - \boldsymbol{X_2}(\boldsymbol{X_1}^\top \boldsymbol{\pi}) \\
&= (X_2 a + Y_2 b + Z_2 c + T_2 d)\boldsymbol{X}_1 - (X_1 a + Y_1 b + Z_1 c + T_2 d)\boldsymbol{X}_2
\end{aligned}
$$

We can then observe the structure of the expansion in both equations and find that $\boldsymbol{X_L}$ is equal to $\boldsymbol{X}(\lambda_{\boldsymbol{\pi}})$ up to scale.

$$
\begin{aligned}
&\boldsymbol{X}(\lambda_\pi) = \frac{-1}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)}[(aX_2 + bY_2 + cZ_2 + dT_2)\boldsymbol{X_1} - (aX_1 + bY_1 + cZ_1 + dT_1)\boldsymbol{X_2}] \\
&= \frac{-1}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)}\boldsymbol{X_L}
\end{aligned}
$$

Problem 2 (Math): Line-quadric intersection (5 points)

In general, a line in 3D intersects a quadric $\boldsymbol{Q}$ at zero, one (if the line is tangent to the quadric), or two points. If the pencil of points $\boldsymbol{X}(\lambda) = \lambda \boldsymbol{X}_1 + (1-\lambda)\boldsymbol{X}_2$ represents a line in 3D, the (up to two) real roots of the quadratic polynomial $c_2 \lambda_{\boldsymbol{Q}}^2 + c_1 \lambda_{\boldsymbol{Q}} + c_0 = 0$ are used to solve for the intersection point(s) $\boldsymbol{X}(\lambda_{\boldsymbol{Q}})$. Show that $c_2 = \boldsymbol{X}_1^\top \boldsymbol{Q} \boldsymbol{X}_1 - 2\boldsymbol{X}_1^\top \boldsymbol{Q} \boldsymbol{X}_2 + \boldsymbol{X}_2^\top \boldsymbol{Q} \boldsymbol{X}_2$, $c_1 = 2(\boldsymbol{X}_1^\top \boldsymbol{Q} \boldsymbol{X}_2 - \boldsymbol{X}_2^\top \boldsymbol{Q} \boldsymbol{X}_2)$, and $c_0 = \boldsymbol{X}_2^\top \boldsymbol{Q} \boldsymbol{X}_2$.

Answer

If $\boldsymbol{X}$ is a point on $\boldsymbol{Q}$, then $\boldsymbol{X}^\top \boldsymbol{Q}\boldsymbol{X} = 0$. The intersection points on $\boldsymbol{X}(\lambda)$ will also satisfy the same equation $\boldsymbol{X}(\lambda_Q)^\top \boldsymbol{Q}\boldsymbol{X}(\lambda_Q) = 0$. We can then expand the equation as below:

$$
\begin{aligned}
&\boldsymbol{X}(\lambda_Q)^\top \boldsymbol{Q}\boldsymbol{X}(\lambda_Q) \\
&= (\lambda_Q \boldsymbol{X_1} + (1-\lambda_Q)\boldsymbol{X_2})^\top \boldsymbol{Q}(\lambda_Q \boldsymbol{X_1} + (1-\lambda_Q)\boldsymbol{X_2}) \\
&= (\lambda_Q \boldsymbol{X_1}^\top \boldsymbol{Q} + \boldsymbol{X_2}^\top \boldsymbol{Q} - \lambda_Q \boldsymbol{X_2}^\top \boldsymbol{Q})(\lambda_Q \boldsymbol{X_1} + \boldsymbol{X_2} - \lambda_Q \boldsymbol{X_2}) \\
&= \lambda_Q^2(\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_1}) + \lambda_Q(\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_2}) - \lambda_Q^2(\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_2}) + \lambda_Q(\boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_1}) + (\boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_2}) - \lambda_Q(\boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_2}) - \lambda_Q^2(\boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_1}) - \\
&= \lambda_Q^2(\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_1} - 2\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_2} + \boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_2}) + 2\lambda_Q(\boldsymbol{X_1}^\top \boldsymbol{Q}\boldsymbol{X_2} - \boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_2}) + \boldsymbol{X_2}^\top \boldsymbol{Q}\boldsymbol{X_2} \\
&= 0
\end{aligned}
$$

We can observe that the last equation is in the form of $c_2 \lambda_Q^2 + c_1 \lambda_Q + c_0$,

where $c_2 = \boldsymbol{X}_1^\top \boldsymbol{Q}\boldsymbol{X}_1 - 2\boldsymbol{X}_1^\top \boldsymbol{Q}\boldsymbol{X}_2 + \boldsymbol{X}_2^\top \boldsymbol{Q}\boldsymbol{X}_2$, $c_1 = 2(\boldsymbol{X}_1^\top \boldsymbol{Q}\boldsymbol{X}_2 - \boldsymbol{X}_2^\top \boldsymbol{Q}\boldsymbol{X}_2)$, and $c_0 = \boldsymbol{X}_2^\top \boldsymbol{Q}\boldsymbol{X}_2$.

Problem 3 (Programming): Linear Estimation of the Camera Projection Matrix (15 points)

Download input data from the course website. The file hw2_points3D.txt contains the coordinates of 50 scene points in 3D (each line of the file gives the $\tilde{X}_i$, $\tilde{Y}_i$, and $\tilde{Z}_i$ inhomogeneous coordinates of a point). The file hw2_points2D.txt contains the coordinates of the 50 corresponding image points in 2D (each line of the file gives the $\tilde{x}_i$ and $\tilde{y}_i$ inhomogeneous coordinates of a point). The scene points have been randomly generated and projected to image points under a camera projection matrix (i.e., $\boldsymbol{x}_i = \boldsymbol{P}\boldsymbol{X}_i$), then noise has been added to the image point coordinates.

Estimate the camera projection matrix $\boldsymbol{P}_{\mathrm{DLT}}$ using the direct linear transformation (DLT) algorithm (with data normalization). You must express $\boldsymbol{x}_i = \boldsymbol{P}\boldsymbol{X}_i$ as $[\boldsymbol{x}_i]^\perp \boldsymbol{P}\boldsymbol{X}_i = \boldsymbol{0}$ (not $\boldsymbol{x}_i \times \boldsymbol{P}\boldsymbol{X}_i = \boldsymbol{0}$), where $[\boldsymbol{x}_i]^\perp \boldsymbol{x}_i = \boldsymbol{0}$, when forming the solution. Return $\boldsymbol{P}_{\mathrm{DLT}}$, scaled such that $||\boldsymbol{P}_{\mathrm{DLT}}||_{\mathrm{Fro}} = 1$

The following helper functions may be useful in your DLT function implementation. You are welcome to add any additional helper functions.

In [1]:
```python
import numpy as np
import time

def Homogenize(x):
    # converts points from inhomogeneous to homogeneous coordinates
    return np.vstack((x,np.ones((1,x.shape[1]))))


def Dehomogenize(x):
    # converts points from homogeneous to inhomogeneous coordinates
    return x[:-1]/x[-1]


def Normalize(pts):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix

    """your code here"""
    #print('-----------normalize---------')
    dim = pts.shape[0]
    mean = np.mean(pts, axis = 1).reshape((dim, 1))
    var = np.var(pts, axis = 1)
    totalVar = np.sum(var)
    s = np.sqrt(dim / totalVar)

    #construct T
    T = np.hstack((np.identity(dim) * s, mean * s * -1))
    T = np.vstack((T, np.zeros(dim + 1)))
    T[-1, -1] = 1

    pts = Homogenize(pts)
    pts = np.dot(T, pts)
    #print("-------normalize end--------")

    return pts, T

def ComputeCost(P, x, X):
    # Inputs:
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #
    # Output:
    #    cost - Total reprojection error
    """your code here"""

    X = Homogenize(X)
    cost = np.sum(np.square(x - Dehomogenize(P @ X)))
    return cost
```

In [2]:

```python
def leftNullofVector(X):

    X = np.reshape(X, (X.shape[0], 1))
    e = np.zeros(X.shape)
    e[0, 0] = 1

    v = X + np.sign(X[0, 0]) * np.linalg.norm(X) * e
    Hv = np.identity(X.shape[0]) - 2 * (v @ v.T) / (v.T @ v)

    return Hv[1:, :]

def DLT(x, X, normalize=True):
    # Inputs:
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    normalize - if True, apply data normalization to x and X
    #
    # Output:
    #    P - the (3x4) DLT estimate of the camera projection matrix
    P = np.eye(3,4)+np.random.randn(3,4)/10

    # data normalization
    if normalize:
        x, T = Normalize(x)
        X, U = Normalize(X)
    else:
        x = Homogenize(x)
        X = Homogenize(X)

    """your code here"""
    A = np.zeros((1, 12))

    for col in range(0, X.shape[1]):
        xNull = leftNullofVector(x[:, col])
        A = np.vstack((A, np.kron(xNull, X[:, col].T)))

    A = A[1:]

    u, s, vt = np.linalg.svd(A)
    P = vt[-1, :]
    P = np.reshape(P, (3, 4))

    # data denormalize
    if normalize:
        P = np.linalg.inv(T) @ P @ U
    P = P / np.linalg.norm(P)

    return P

def displayResults(P, x, X, title):
    print(title+' =')
    print (P/np.linalg.norm(P)*np.sign(P[-1,-1]))

# load the data
x=np.loadtxt('hw2_points2D.txt').T
X=np.loadtxt('hw2_points3D.txt').T
```

```python
# compute the linear estimate without data normalization
print ('Running DLT without data normalization')
time_start=time.time()
P_DLT = DLT(x, X, normalize=False)
cost = ComputeCost(P_DLT, x, X)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost=%.9f'%cost)


# compute the linear estimate with data normalization
print ('Running DLT with data normalization')
time_start=time.time()
P_DLT = DLT(x, X, normalize=True)
cost = ComputeCost(P_DLT, x, X)
time_total=time.time()-time_start
# display the results
print('took %f secs'%time_total)
print('Cost=%.9f'%cost)
```

```
Running DLT without data normalization
took 0.011266 secs
Cost=97.053718945
Running DLT with data normalization
took 0.005344 secs
Cost=84.104680130
```

In [3]:
```python
# Report your P_DLT value here!
displayResults(P_DLT, x, X, 'P_DLT')
```

```
P_DLT =
[[ 6.04350846e-03 -4.84282446e-03  8.82395315e-03  8.40441373e-01]
 [ 9.09666810e-03 -2.30374203e-03 -6.18060233e-03  5.41657305e-01]
 [ 5.00625470e-06  4.47558354e-06  2.55223773e-06  1.25160752e-03]]
```

Problem 4 (Programming): Nonlinear Estimation of the Camera Projection Matrix (30 points)

Use $\boldsymbol{P}_{\mathrm{DLT}}$ as an initial estimate to an iterative estimation method, specifically the Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the camera projection matrix that minimizes the projection error. You must parameterize the camera projection matrix as a parameterization of the homogeneous vector $\boldsymbol{p} = vec(\boldsymbol{P}^\top)$. It is highly recommended to implement a parameterization of homogeneous vector method where the homogeneous vector is of arbitrary length, as this will be used in following assignments.

Report the initial cost (i.e. cost at iteration 0) and the cost at the end of each successive iteration. Show the numerical values for the final estimate of the camera projection matrix $\boldsymbol{P}_{\mathrm{LM}}$, scaled such that $||\boldsymbol{P}_{\mathrm{LM}}||_{\mathrm{Fro}} = 1$.

The following helper functions may be useful in your LM function implementation. You are welcome to add any additional helper functions.

Hint: LM has its biggest cost reduction after the 1st iteration. You'll know if you are implementing LM correctly if you experience this.

In [12]:

```
# Note that np.sinc is different than defined in class
def Sinc(x):
    # Returns a scalar valued sinc value
    """your code here"""
    if x == 0:
        return 1
    else:
        return (np.sin(x) / x)

def dSinc(x):
    if x == 0:
        return 0
    else:
        return np.cos(x) / x - np.sin(x) / np.square(x)

def Jacobian(P,p,X):
    # compute the jacobian matrix
    #
    # Input:
    #    P - 3x4 projection matrix
    #    p - 11x1 homogeneous parameterization of P
    #    X - 3n 3D scene points
    # Output:
    #    J - 2nx11 jacobian matrix
    J = np.zeros((1,11))

    """your code here"""
    X = Homogenize(X)
    x = Dehomogenize(P @ X)

    x, y = x[0], x[1]

    ######### dp_bar/ dp
    norm = np.linalg.norm(p)
    p_bar = P.reshape(-1, 1)

    a, b = p_bar[0], p_bar[1:]
    I = np.identity(b.shape[0])

    if norm == 0:
        da = np.zeros(b.shape.T)
        db = 0.5 * I
    else:
        da = -0.5 * b.T
        db = Sinc(norm / 2) / 2 * I + (1 / (4 * norm)) * dSinc(norm / 2)
* p @ p.T

    dpbardp = np.vstack((da, db))

    ####### dx / dp_bar
    w = P[2] @ X
    zero = np.zeros((X.shape[0]))
    for col in range(0, X.shape[1]):
        row1 = np.hstack((X[:, col].T, zero.T, -1 * x[col] * X[:, col].T
))
        row2 = np.hstack((zero.T, X[:, col].T, -1 * y[col] * X[:, col].T
```

```
))
        dxdpbar = (1 / w[col]) * np.vstack((row1, row2))
        J = np.vstack((J, dxdpbar @ dpbardp))

    return J[1:]


def Parameterize(P):
    # wrapper function to interface with LM
    # takes all optimization variables and parameterizes all of them
    # in this case it is just P, but in future assignments it will
    # be more useful
    return ParameterizeHomog(P.reshape(-1,1))


def Deparameterize(p):
    # Deparameterize all optimization variables
    return DeParameterizeHomog(p).reshape(3,4)


def ParameterizeHomog(V):
    # Given a homogeneous vector V return its minimal parameterization
    """your code here"""
    #print("-------Parameterize------")
    v = V / np.linalg.norm(V)
    a, b = v[0], v[1:]
    v_hat = (2 / Sinc(np.arccos(a))) * b

    norm = np.linalg.norm(v_hat)
    if norm > np.pi:
        v_hat *= 1 - ((2 * np.pi) / norm) * np.ceil((norm - np.pi) /(2 *
np.pi))
    #print("---------Parameterize end----------")
    #print(v_hat)
    return v_hat


def DeParameterizeHomog(v):
    # Given a parameterized homogeneous vector return its deparameteriza
tion
    """your code here"""
    #print("-------DeParameterize------")
    norm = np.linalg.norm(v)
    v_bar = np.zeros((v.shape[0] + 1, 1))
    v_bar[0] = np.cos(norm / 2)
    v_bar[1:] = ((Sinc(norm / 2) / 2)* v)
    v_bar /= np.linalg.norm(v_bar)
    #print("-------DeParameterize end------")
    return v_bar
```

In [31]:
```
def LM(P, x, X, max_iters, lam):
    # Input:
    #     P - initial estimate of P
    #     x - 2D inhomogeneous image points
    #     X - 3D inhomogeneous scene points
    #     max_iters - maximum number of iterations
    #     lam - lambda parameter
    # Output:
    #     P - Final P (3x4) obtained after convergence

    # data normalization
    x, T = Normalize(x)
    X, U = Normalize(X)

    P = T @ P @ np.linalg.inv(U)

    p = Parameterize(P)
    P = Deparameterize(p)

    inhomo_x = Dehomogenize(x)
    inhomo_X = Dehomogenize(X)

    invcov = np.linalg.inv(np.identity(x.shape[1] * 2) * np.square(T[0,
0]))
    x_meas = np.array([inhomo_x.flatten('F')]).T
    e = x_meas - np.array([Dehomogenize(P @ X).flatten('F')]).T
    SSE = e.T @ invcov @ e

    #print("iteration start!!!!!!!")
    for i in range(max_iters):
        J = Jacobian(Deparameterize(p), p, inhomo_X)
        tmp = J.T @ invcov
        while(True):
            inv = np.linalg.inv(tmp @ J + lam * np.identity(11))
            update = inv @ tmp @ e
            newp = p + update
            newP = Deparameterize(newp)
            newe = x_meas - np.array([Dehomogenize(newP @ X).flatten('F'
)]).T
            newSSE = newe.T @ invcov @ newe
            if newSSE < SSE:
                SSE = newSSE
                e = newe
                p = newp
                P = newP
                lam = 0.1 * lam
                break
            elif newSSE - SSE > 0.00005:
                lam = 10 * lam
            else:
                break
        print ('iter %03d Cost %.9f'%(i+1, SSE))

    # data denormalization
    P = np.linalg.inv(T) @ P @ U
    return P
```

```
# LM hyperparameters
lam = .001
max_iters = 100

# Run LM initialized by DLT estimate with data normalization
print ('Running LM with data normalization')
print ('iter %03d Cost %.9f'%(0, cost))
time_start=time.time()
P_LM = LM(P_DLT, x, X, max_iters, lam)
time_total=time.time()-time_start
print('took %f secs'%time_total)
```

```
Running LM with data normalization
iter 000 Cost 84.104680130
iter 001 Cost 82.791336044
iter 002 Cost 82.790238006
iter 003 Cost 82.790238005
iter 004 Cost 82.790238005
iter 005 Cost 82.790238005
iter 006 Cost 82.790238005
iter 007 Cost 82.790238005
iter 008 Cost 82.790238005
iter 009 Cost 82.790238005
iter 010 Cost 82.790238005
iter 011 Cost 82.790238005
iter 012 Cost 82.790238005
iter 013 Cost 82.790238005
iter 014 Cost 82.790238005
iter 015 Cost 82.790238005
iter 016 Cost 82.790238005
iter 017 Cost 82.790238005
iter 018 Cost 82.790238005
iter 019 Cost 82.790238005
iter 020 Cost 82.790238005
iter 021 Cost 82.790238005
iter 022 Cost 82.790238005
iter 023 Cost 82.790238005
iter 024 Cost 82.790238005
iter 025 Cost 82.790238005
iter 026 Cost 82.790238005
iter 027 Cost 82.790238005
iter 028 Cost 82.790238005
iter 029 Cost 82.790238005
iter 030 Cost 82.790238005
iter 031 Cost 82.790238005
iter 032 Cost 82.790238005
iter 033 Cost 82.790238005
iter 034 Cost 82.790238005
iter 035 Cost 82.790238005
iter 036 Cost 82.790238005
iter 037 Cost 82.790238005
iter 038 Cost 82.790238005
iter 039 Cost 82.790238005
iter 040 Cost 82.790238005
iter 041 Cost 82.790238005
iter 042 Cost 82.790238005
iter 043 Cost 82.790238005
iter 044 Cost 82.790238005
iter 045 Cost 82.790238005
iter 046 Cost 82.790238005
iter 047 Cost 82.790238005
iter 048 Cost 82.790238005
iter 049 Cost 82.790238005
iter 050 Cost 82.790238005
iter 051 Cost 82.790238005
iter 052 Cost 82.790238005
iter 053 Cost 82.790238005
iter 054 Cost 82.790238005
iter 055 Cost 82.790238005
```

```
iter 056 Cost 82.790238005
iter 057 Cost 82.790238005
iter 058 Cost 82.790238005
iter 059 Cost 82.790238005
iter 060 Cost 82.790238005
iter 061 Cost 82.790238005
iter 062 Cost 82.790238005
iter 063 Cost 82.790238005
iter 064 Cost 82.790238005
iter 065 Cost 82.790238005
iter 066 Cost 82.790238005
iter 067 Cost 82.790238005
iter 068 Cost 82.790238005
iter 069 Cost 82.790238005
iter 070 Cost 82.790238005
iter 071 Cost 82.790238005
iter 072 Cost 82.790238005
iter 073 Cost 82.790238005
iter 074 Cost 82.790238005
iter 075 Cost 82.790238005
iter 076 Cost 82.790238005
iter 077 Cost 82.790238005
iter 078 Cost 82.790238005
iter 079 Cost 82.790238005
iter 080 Cost 82.790238005
iter 081 Cost 82.790238005
iter 082 Cost 82.790238005
iter 083 Cost 82.790238005
iter 084 Cost 82.790238005
iter 085 Cost 82.790238005
iter 086 Cost 82.790238005
iter 087 Cost 82.790238005
iter 088 Cost 82.790238005
iter 089 Cost 82.790238005
iter 090 Cost 82.790238005
iter 091 Cost 82.790238005
iter 092 Cost 82.790238005
iter 093 Cost 82.790238005
iter 094 Cost 82.790238005
iter 095 Cost 82.790238005
iter 096 Cost 82.790238005
iter 097 Cost 82.790238005
iter 098 Cost 82.790238005
iter 099 Cost 82.790238005
iter 100 Cost 82.790238005
took 0.332865 secs
```

In [32]:

```
# Report your P_LM final value here!
displayResults(P_LM, x, X, 'P_LM')
```

```
P_LM =
[[ 6.09434291e-03 -4.72647758e-03  8.79023503e-03  8.43642842e-01]
 [ 9.02017241e-03 -2.29290824e-03 -6.13330068e-03  5.36660248e-01]
 [ 4.99088611e-06  4.45205073e-06  2.53705045e-06  1.24348254e-03]]
```

In [ ]: