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.

Your solution here

Problem 1 : $X_L = L\pi$, $L = X_1X_2^T - X_2X_1^T$ $X_1 = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ T_1 \end{bmatrix}$ $X_2 = \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ T_2 \end{bmatrix}$

$$\therefore L = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ T_1 \end{bmatrix} [X_2\, Y_2\, Z_2\, T_2] - \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ T_2 \end{bmatrix} [X_1\, Y_1\, Z_1\, T_1]$$

$$= \begin{bmatrix} X_1X_2 & X_1Y_2 & X_1Z_2 & X_1T_2 \\ Y_1X_2 & Y_1Y_2 & Y_1Z_2 & Y_1T_2 \\ Z_1X_2 & Z_1Y_2 & Z_1Z_2 & Z_1T_2 \\ T_1X_2 & T_1Y_2 & T_1Z_2 & T_1T_2 \end{bmatrix} - \begin{bmatrix} X_2X_1 & X_2Y_1 & X_2Z_1 & X_2T_1 \\ Y_2X_1 & Y_2Y_1 & Y_2Z_1 & Y_2T_1 \\ Z_2X_1 & Z_2Y_1 & Z_2Z_1 & Z_2T_1 \\ T_2X_1 & T_2Y_1 & T_2Z_1 & T_2T_1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & X_1Y_2 - X_2Y_1 & X_1Z_2 - X_2Z_1 & X_1T_2 - X_2T_1 \\ Y_1X_2 - Y_2X_1 & 0 & Y_1Z_2 - Y_2Z_1 & Y_1T_2 - Y_2T_1 \\ Z_1X_2 - Z_2X_1 & Z_1Y_2 - Z_2Y_1 & 0 & Z_1T_2 - Z_2T_1 \\ T_1X_2 - T_2X_1 & T_1Y_2 - T_2Y_1 & T_1Z_2 - T_2Z_1 & 0 \end{bmatrix}$$

$X_L = L\pi$

$$\therefore X_L = L \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} b(X_1Y_2 - X_2Y_1) + c(X_1Z_2 - X_2Z_1) + d(X_1T_2 - X_2T_1) \\ a(Y_1X_2 - X_1Y_2) + c(Y_1Z_2 - Y_2Z_1) + d(Y_1T_2 - Y_2T_1) \\ a(Z_1X_2 - Z_2X_1) + b(Z_1Y_2 - Z_2Y_1) + d(Z_1T_2 - Z_2T_1) \\ a(T_1X_2 - T_2X_1) + b(T_1Y_2 - T_2Y_1) + c(T_1Z_2 - T_2Z_1) \end{bmatrix}$$

Pencil pts : $X(\lambda) = \lambda X_1 + (1-\lambda)X_2$.

$$X(\lambda) = \lambda \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ T_1 \end{bmatrix} + (1-\lambda) \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ T_2 \end{bmatrix} = \begin{bmatrix} \lambda(X_1 - X_2) + X_2 \\ \lambda(Y_1 - Y_2) + Y_2 \\ \lambda(Z_1 - Z_2) + Z_2 \\ \lambda(T_1 - T_2) + T_2 \end{bmatrix}$$

$X(\lambda_\pi)^T \pi = 0$ solve for λ_π

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

Take λ back to $X(\lambda_\pi)$.

$$X(\lambda\pi) = \lambda \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ T_1 \end{bmatrix} + (1-\lambda) \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ T_2 \end{bmatrix} \qquad \alpha = \frac{-1}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)}$$

$$= \frac{-1}{a(X_1-X_2)+b(Y_1-Y_2)+c(Z_1-Z_2)+d(T_1-T_2)} \left[(aX_2+bY_2+cZ_2+dT_2) \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ T_1 \end{bmatrix} - (aX_1+bY_1+cZ_1+dT_1) \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ T_2 \end{bmatrix} \right]$$

$$= \alpha \begin{bmatrix} b(Y_2X_1 - Y_1X_2) + c(Z_2X_1 - Z_1X_2) + d(T_2X_1 - T_1X_2) \\ a(Y_1X_2 - X_1Y_2) + c(Y_1Z_2 - Y_2Z_1) + d(Y_1T_2 - Y_2T_1) \\ a(Z_1X_2 - Z_2X_1) + b(Z_1Y_2 - Z_2Y_1) + d(Z_1T_2 - Z_2T_1) \\ a(T_1X_2 - T_2X_1) + b(T_1Y_2 - T_2Y_1) + c(T_1Z_2 - T_2Z_1) \end{bmatrix} = \alpha X_L$$

$\therefore X_L$ is equal to $X(\lambda\pi)$ up to scale.

Problem 2.

$x(\lambda)$ intersects a quadric Q. $x = \lambda x_1 + (1-\lambda) x_2$.

$\therefore X^T Q X = 0$

$= (\lambda x_1 + (1-\lambda)x_2)^T Q (\lambda x_1 + (1-\lambda) x_2)$

$= \lambda^2 x_1^T Q x_1 + (\lambda x_1^T Q (1-\lambda) x_2) + (1-\lambda) x_2^T Q \lambda x_1 + (1-\lambda) x_2^T Q (1-\lambda) x_2.$

$= (x_1^T Q x_1 + x_2^T Q x_2 - x_1^T Q x_2 - x_2^T Q x_1)\lambda^2 + (x_1^T Q x_2 + x_2^T Q x_1 - 2 x_2^T Q x_2)\lambda$
$+ x_2^T Q x_2.$

x_1 : 4*1 matrix Q : 4*4 (symmetric matrix) x_2 : 4*1.

Let $x_1^T Q x_2 = a$ where a is a scale. $\xrightarrow{\text{transpose.}}$ $x_2^T Q^T x_1 = a$.

$x_2^T Q x_1 = b$

$\because Q$ is symm $\therefore Q = Q^T$

$\therefore x_2^T Q x_1 = a = b$

$\therefore x_1^T Q x_2 = x_2^T Q x_1$

$\therefore X^T Q X = (x_1^T Q x_1 - 2x_1^T Q x_2 + x_2^T Q x_2)\lambda^2 + 2(x_1^T Q x_2 - x_2^T Q x_2)\lambda$
$+ x_2^T Q x_2 = 0$

$\therefore c_2 = x_1^T Q x_1 - 2x_1^T Q x_2 + x_2^T Q x_2$

$c_1 = 2(x_1^T Q x_2 - x_2^T Q x_2)$

$c_0 = x_2^T Q x_2.$

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 $c_0 = \boldsymbol{X}_2^\top\boldsymbol{Q}\boldsymbol{X}_2$.

Your solution here

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 $\boldsymbol{P}_{\text{DLT}}$ 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 $\boldsymbol{P}_{\text{DLT}}$, scaled such that $||\boldsymbol{P}_{\text{DLT}}||_{\text{Fro}} = 1$

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"""
    dimension = pts.shape[0]
    variance = np.var(pts,axis = 1)
    mean = np.mean(pts,axis = 1)
    var_tol = variance.sum()
    S = np.sqrt(dimension/var_tol)
    T = np.eye(pts.shape[0]+1)
    T[:dimension, :dimension] = S* np.eye(dimension)
    for i in range (dimension) :
        T[i,-1] = -S*mean[i]
    pts_homo = Homogenize(pts)
    pts = T @ pts_homo  #home - W scale
    #pts = Dehomogenize(pts_normalized)   #inhomi - W/O scale

    return pts, T


def ComputeCost(P, x, X):
    # Inputs:
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #
    # Output:
    #    cost - Total reprojection error
    #cost = ComputeCost(P_DLT, x, X)
    n = x.shape[1]
    covarx = np.eye(2*n)

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


    cost = ((x - x_predict)**2).sum()
    return cost
```

In [2]:

```python
def left_null_calculator(x):
    #x - homo W scale
    #Hv Household matrix
    #v Household matrix
    x_left = np.zeros((x.shape[0]-1,x.shape[0]*x.shape[1]))
    e = np.zeros((x.shape[0],1))
    e[0] = 1
    for i in range (x.shape[1]):
        sign = np.sign(x[0,i])
        v = x[:,i].reshape(-1,1) + sign * np.linalg.norm(x[:,i]) *e
        Hv = np.eye((x.shape[0])) -2 * np.dot(v,v.T)/np.dot(v.T,v)
        x_left[:,i*x.shape[0]:(i+1)*x.shape[0]] = Hv[1:,:]
    return x_left
```

In [3]:

```python
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"""
    x_left = left_null_calculator(x)
    A = np.zeros((2*x.shape[1],12))
    for i in range (x.shape[1]):
        A[2*i:2*i+2,:] = np.kron(x_left[:,i*x.shape[0]: (i+1)*x.shape[0]],X[:,i]
.T)

    u,s,vh = np.linalg.svd(A)
    P_bar = vh[-1,:]
    # per rows
    P = P_bar.reshape(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


# 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 0.408368 secs
Cost = 97.053719225
Running DLT with data normalization
took 0.009214 secs
Cost = 84.104680130
```

In [4]:

```
# 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 $\boldsymbol{P}_{\text{DLT}}$ 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 $\boldsymbol{P}_{\text{LM}}$, scaled such that $||\boldsymbol{P}_{\text{LM}}||_{\text{Fro}} = 1$.

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 [5]:

```python
# Note that np.sinc is different than defined in class
from numpy import sin, cos , pi
from math import ceil
def Sinc(x):
    # Returns a scalar valued sinc value
    """your code here"""
    if x == 0:
        y = 1
    else:
        y = sin(x)/x

    return y


def Jacobian(P,p,X):
    # compute the jacobian matrix
    #
    # Input:
    #    P - 3x4 projection matrix ##should be mormalized??
    #    p - 11x1 homogeneous parameterization of P
    #    X - 3n 3D scene points #homo 4*50 homo Normalized
    # Output:
    #    J - 2nx11 jacobian matrix

    J = np.zeros((2*X.shape[1],11))

    """your code here"""
    x_homo = P @ X  #x_estimate   homo W scale 3*50
    P_bar = P.reshape(-1,1)  #12*1

    #a = P_bar[0]
    b = P_bar[1:].reshape(-1,1)  #11*1
    a_diff = np.zeros((1,11)) #1*11
    b_diff = np.eye(11)/2
    n_p = np.linalg.norm(p)  #norm_p
    h_p = n_p/2              #half_norm_p
    diff_sinc = cos(h_p)/h_p - sin(h_p)/(h_p**2)
    if n_p != 0:
        a_diff = -b.T/2
        b_diff = Sinc(h_p)*np.eye(11)/2 + (p @ p.T)*diff_sinc/(4*n_p)

    P_bar_over_p = np.vstack((a_diff,b_diff)) #12*11

    x_hat_over_P_bar = np.zeros((2*X.shape[1],12)) #2n*12
    x_in = Dehomogenize(x_homo) #x inhomo 2*50
    w = x_homo[-1]
    for i in range (X.shape[1]):
        left = np.vstack((X[:,i].reshape(-1,1).T,np.zeros([1,4])))
        mid = np.vstack((np.zeros([1,4]),X[:,i].reshape(-1,1).T))
        right = np.vstack((-x_in[0,i]*X[:,i].reshape(-1,1).T,-x_in[1,i]*X[:,i].r
eshape(-1,1).T))

        x_hat_over_P_bar[2*i:2*i+2,:] = (1/w[i])* np.hstack((left,mid,right))

    J = x_hat_over_P_bar @ P_bar_over_p

    return J


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"""
    a = V[0]
    b = V[1:]
    v_hat = (2 * b)/Sinc(np.arccos(a))

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


    return v_hat


def DeParameterizeHomog(v):
    # Given a parameterized homogeneous vector return its deparameterization 11*
1 --> 12*1
    """your code here"""
    v_bar = np.zeros((v.shape[0]+1,1))
    v_bar[0] = cos(np.linalg.norm(v)/2)
    v_bar[1:] = Sinc(np.linalg.norm(v)/2)/2 * v
    return v_bar
```

In [6]:

```
def LM(P, x, X, max_iters, lam):

    #all in normarlization

    # 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

    x_original = x
    X_original = X
    # data normalization
    x, T = Normalize(x) #output-homo   input - inhomo     x-normalized
    X, U = Normalize(X) #homo   X-normalized
    """your code here"""
    P = T @ P @ np.linalg.inv(U)   #normarlized

    # you may modify this so long as the cost is computed
    # at each iteration
    x = Dehomogenize(x)

    p = Parameterize(P) #11*1
    P = Deparameterize(p) #3*4   P normarlized

    x_hat = Dehomogenize(P @ X)
    error = np.zeros((2*x.shape[1],1))
    error = x.reshape(-1,1,order = 'F') - x_hat.reshape(-1,1,order = 'F')

    cov_x = (T[0,0]**2)*np.eye(2*x.shape[1])
    B = np.zeros((11,1))
    A = np.zeros((11,11))
    tolerance = 1e-7

    Pre_cost = 0   # initialize cost with a small value
    cov_inv = np.linalg.inv(cov_x)
    error_esti = error.T @ cov_inv @ error

    J = Jacobian(P,p,X)      #step 2n*11
    for i in range(max_iters):

        A = J.T @ cov_inv @ J + lam * np.eye((11))  #step 4 11*11
        B = J.T @ cov_inv @ error                    #11*1
        sigma = np.linalg.inv(A) @ B
        p_new = p + sigma.reshape(-1,1)
        P = Deparameterize(p_new)  #AFTER depara, P is normalized 3*4

        x_new = Dehomogenize(P @ X)
        error_new = x.reshape(-1,1,order = 'F') - x_new.reshape(-1,1,order = 'F'
)

        error_esti_new = error_new.T @ cov_inv @ error_new
        print ('iter %03d Cost %.9f'%(i+1, error_esti_new))
        if error_esti_new < error_esti:
            if tolerance > error_esti - error_esti_new:
                break
            p = p_new
            error = error_new
            error_esti = error_esti_new
            lam = 0.1 * lam
            J = Jacobian(P,p,X)      #step 2n*11

        else:
            lam = 10 *lam

    # data denormalization
    P = np.linalg.inv(T) @ P @ U
    #print('Denormalized cost',ComputeCost(P, x_original,X_original ))
    return P



# LM hyperparameters
lam = .001
max_iters = 5

# Run LM initialized by DLT estimate with data normalization
print ('Running LM with data normalization\n')
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
took 0.038921 secs
```

In [7]:

```
# 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]]
```