CSE 252B: Computer Vision II, Winter 2024 – Assignment 2

Instructor: Ben Ochoa

Assignment due: Wed, Feb 7, 11:59 PM

Name: Xiao Nan

PID: A69027384

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- All solutions must be written in this notebook.
- Math must be done in Markdown/$\LaTeX$.
- You must show your 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 efficient grading).
- You may use python packages for basic linear algebra, but you may not use functions that directly solve the problem. If you are uncertain about using a specific package, function, or method, then please ask the instructional staff whether it is allowable.
- **You must submit this notebook as an .ipynb file, a .py file, and a .pdf file on Gradescope.**
  - You may directly export the notebook as a .py file. You may use nbconvert to convert the .ipynb file to a .py file using the following command `jupyter nbconvert --to script filename.ipynb`
  - There are two methods to convert the notebook to a .pdf file.
    - You may first export the notebook as a .html file, then print the web page as a .pdf file.
    - If you have XeTeX installed, then you may directly export the notebook as a .pdf file. You may use nbconvert to convert a .ipynb file to a .pdf file using the following command `jupyter nbconvert --allow-chromium-download --to webpdf filename.ipynb`
  - **You must ensure the contents in each cell (e.g., code, output images, printed results, etc.) are clearly visible, and are not cut off or partially cropped in the .pdf file.**
  - Your code and results must remain inline in the .pdf file (do not move your code to an appendix).
  - **While submitting on gradescope, you must assign the relevant pages in the .pdf file submission for each problem.**
- 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 $\mathbf{X}_1 = (X_1, Y_1, Z_1, T_1)^\top$ and $\mathbf{X}_2 = (X_2, Y_2, Z_2, T_2)^\top$ can be represented as a Plücker matrix $\mathtt{L} = \mathbf{X}_1\mathbf{X}_2^\top - \mathbf{X}_2\mathbf{X}_1^\top$ or pencil of points $\mathbf{X}(\lambda) = \lambda\mathbf{X}_1 + (1-\lambda)\mathbf{X}_2$ (i.e., $\mathbf{X}$ is a function of λ). The line intersects the plane $\boldsymbol{\pi} = (a, b, c, d)^\top$ at the point $\mathbf{X}_{\mathtt{L}} = \mathtt{L}\boldsymbol{\pi}$ or $\mathbf{X}(\lambda_{\boldsymbol{\pi}})$, where $\lambda_{\boldsymbol{\pi}}$ is determined such that $\mathbf{X}(\lambda_{\boldsymbol{\pi}})^\top\boldsymbol{\pi} = 0$ (i.e., $\mathbf{X}(\lambda_{\boldsymbol{\pi}})$ is the point on $\boldsymbol{\pi}$). Show that $\mathbf{X}_{\mathtt{L}}$ is equal to $\mathbf{X}(\lambda_{\boldsymbol{\pi}})$ up to scale.

From $\mathbf{X}(\lambda_{\boldsymbol{\pi}})^\top\boldsymbol{\pi} = 0$ we get:

$$\begin{aligned}
(\lambda_\pi \mathbf{X}_1 + (1-\lambda_\pi)\mathbf{X}_2)^T\pi &= 0 && (1)\\
-\lambda_\pi \mathbf{X}_1^T\pi &= (1-\lambda_\pi)\mathbf{X}_2^T\pi && (2)\\
\mathbf{X}_1^T\pi &= \frac{\lambda_\pi - 1}{\lambda_\pi}\mathbf{X}_2^T\pi && (3)
\end{aligned}$$

Then we compute:

$$\begin{aligned}\mathbf{X}_{\mathrm{L}} &= \mathtt{L}\boldsymbol{\pi} && (4)\\ &= (\mathrm{X}_1\mathrm{X}_2^T - \mathrm{X}_2\mathrm{X}_1^T)\boldsymbol{\pi} && (5)\\ &= \mathrm{X}_1\mathrm{X}_2^T\boldsymbol{\pi} - \mathrm{X}_2\mathrm{X}_1^T\pi && (6)\end{aligned}$$

Substitute $\mathbf{X}_1^T\pi = \frac{\lambda_\pi - 1}{\lambda_\pi}\mathbf{X}_2^T\pi$, we get:

$$\begin{aligned} &= \mathrm{X}_1\mathrm{X}_2^T\boldsymbol{\pi} - \frac{\lambda_\pi - 1}{\lambda_\pi}\mathrm{X}_2\mathrm{X}_2^T\pi && (7)\\ &= (\frac{\lambda_\pi\mathrm{X}_1 + (1-\lambda_\pi)\mathrm{X}_2}{\lambda_\pi})\mathrm{X}_2^T\boldsymbol{\pi} && (8)\\ &= \frac{\mathrm{X}_2^T\pi}{\lambda_\pi}\mathrm{X}(\lambda_\pi) && (9)\end{aligned}$$

Hence $\mathbf{X}_{\mathrm{L}}$ is $\mathrm{X}(\lambda_\pi)$ up to the scale of $\frac{\mathrm{X}_2^T\pi}{\lambda_\pi}$. Proved.

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

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

For the point $\mathbf{X}(\lambda_{\mathtt{Q}}) = \lambda_{\mathtt{Q}}\mathbf{X}_1 + (1-\lambda_{\mathtt{Q}})\mathbf{X}_2$ to be on the quadric $\mathtt{Q}$, we have the quadratic polynomial:

$$\begin{aligned}\mathbf{X}(\lambda_{\mathtt{Q}})^T\mathtt{Q}\mathbf{X}(\lambda_{\mathtt{Q}}) &= 0 && (10)\\ (\lambda_{\mathtt{Q}}\mathbf{X}_1 + (1-\lambda_{\mathtt{Q}})\mathbf{X}_2)^T(\lambda_{\mathtt{Q}}\mathtt{Q}\mathbf{X}_1 + (1-\lambda_{\mathtt{Q}})\mathtt{Q}\mathbf{X}_2) &= 0 && (11)\\ (\lambda_{\mathtt{Q}}(\mathbf{X}_1 - \mathbf{X}_2) + \mathbf{X}_2)^T(\lambda_{\mathtt{Q}}\mathtt{Q}(\mathbf{X}_1 - \mathbf{X}_2) + \mathtt{Q}\mathbf{X}_2) &= 0 && (12)\\ (\mathbf{X}_1 - \mathbf{X}_2)^T\mathtt{Q}(\mathbf{X}_1 - \mathbf{X}_2)\lambda_{\mathtt{Q}}^2 + (\mathbf{X}_2^T\mathtt{Q}(\mathbf{X}_1 - \mathbf{X}_2) + (\mathbf{X}_1 - \mathbf{X}_2)^T\mathtt{Q}\mathbf{X}_2)\lambda_{\mathtt{Q}} + \mathbf{X}_2^T\mathtt{Q}\mathbf{X}_2 &= 0 && (13)\\ (\mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_1 - 2\mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_2 + \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2)\lambda_{\mathtt{Q}}^2 + 2(\mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_2 - \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2)\lambda_{\mathtt{Q}} + \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2 &= 0 && (14)\\ c_2\lambda_{\mathtt{Q}}^2 + c_1\lambda_{\mathtt{Q}} + c_0 &= 0 && (15)\end{aligned}$$

Where $c_2 = \mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_1 - 2\mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_2 + \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2, c_1 = 2(\mathbf{X}_1^\top\mathtt{Q}\mathbf{X}_2 - \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2), c_0 = \mathbf{X}_2^\top\mathtt{Q}\mathbf{X}_2$, and the equation's real roots are solutions for $\mathrm{X}(\lambda_{\mathtt{Q}})$. Proved.

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., $\mathbf{x}_i = \mathtt{P}\mathbf{X}_i$), then noise has been added to the image point coordinates.

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

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

In [ ]:

```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 data_normalize(pts):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix
    dim = pts.shape[0]
    pts_homo = homogenize(pts)
    if (dim == 2):
        mux, muy, varx, vary = np.mean(pts[0, :]), np.mean(pts[1, :]), np.var(pts[0, :]), np.var(pts[1, :])
        s = np.sqrt(2/(varx + vary))
        T = np.array([[s, 0, -s*mux], [0, s, -s*muy], [0, 0, 1]])
    else:
        mux, muy, muz, varx, vary, varz = np.mean(pts[0, :]), np.mean(pts[1, :]), np.mean(pts[2, :]), np.var(pts[0
        s = np.sqrt(3/(varx + vary + varz))
        T = np.array([[s, 0, 0, -s*mux], [0, s, 0, -s*muy], [0, 0, s, -s*muz], [0, 0, 0, 1]])
    pts_tr_homo = T @ pts_homo # perform transformation
    # pts_tr = dehomogenize(pts_tr_homo)
    return pts_tr_homo, T

def sum_of_square_projection_error(P, x, X):
    # Inputs:
    #    P - the camera projection matrix
    #    x - 2D inhomogeneous image points (2*n)
    #    X - 3D inhomogeneous scene points (3*n)
    # Output:
    #    cost - Sum of squares of the reprojection error

    x_homo = homogenize(x)
    X_homo = homogenize(X)

    x_proj_homo = P @ X_homo
    x_proj = dehomogenize(x_proj_homo)

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

In [ ]:

```python
def get_left_null_space(x):
    """
    get left null space of pt x
    input:  x - 2D homogeneous image points (3*1)
    output:  x_perp - null space of the points (2*3*1)
    """
    # print(x.shape)
    x_norm = np.linalg.norm(x)
    e1 = np.eye(3)[:, 0]
    sign = np.sign(x[0])
    v = x + sign * x_norm * e1
    Hv = np.eye(3) - 2 * ((np.outer(v, v)) / (v.T @ v))
    x_perp = Hv[1:, :]
    return x_perp

def estimate_camera_projection_matrix_linear(x, X, normalize=True):
    # Inputs:
    #    x - 2D inhomogeneous image points (2*n)
    #    X - 3D inhomogeneous scene points (3*n)
    #    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
    n = x.shape[1]
    A = np.zeros((2*n, 12))

    # data normalization
    if normalize:
        x, T = data_normalize(x)
        X, U = data_normalize(X)
    else:
        x = homogenize(x)
        X = homogenize(X)

    # fill the left kron matrix
    for i in range(n):
        xi, Xi = x[:,i], X[:,i]
        xi_perp = get_left_null_space(xi)
        # print(xi_perp.shape, Xi.shape)
        subi = np.kron(xi_perp, Xi)
        # print(subi.shape)
        A[2*i:2*i+2, :] = subi

    # SVD to find solution
    _, _, Vt = np.linalg.svd(A, full_matrices=False)
    p = Vt[-1, :]

    # load the resulting p to the projection matrix
    P = np.reshape(p, (3,4))

    # P[0, :], P[1, :], P[2, :] = p[:4].T, p[4:8].T, p[8:].T

    # data denormalize
    if normalize:
        # print(T.shape, U.shape, P.shape)
        P = np.linalg.inv(T) @ P @ U

    return P

def display_results(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

assert x.shape[1] == X.shape[1]
n = x.shape[1]

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

print("\n==Correct outputs==")
print("Cost=%.9f without data normalization"%97.053718991)
print("Cost=%.9f with data normalization"%84.104680130)
```

```
Running DLT without data normalization
took 0.005028 secs
Cost=97.053718928
Running DLT with data normalization
took 0.003991 secs
Cost=84.104680130

==Correct outputs==
Cost=97.053718991 without data normalization
Cost=84.104680130 with data normalization
```

In [ ]:
```python
# Report your P_DLT (estimated camera projection matrix linear) value here!
display_results(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 $\mathrm{P}_{\mathrm{DLT}}$ 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 $\mathbf{p} = \mathrm{vec}(\mathrm{P}^{\top})$. 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 P_{LM}, scaled such that $\|\mathrm{P}_{\mathrm{LM}}\|_{\mathrm{Fro}} = 1$.

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

Hint: LM has its biggest cost reduction after the first iteration. If you do not experience this, then there may be an issue with your implementation.

In [ ]:
```python
# Note that np.sinc is different than defined in class
def sinc(x):
    # Returns a scalar valued sinc value
    if (x==0):
        return 1
    else:
        return np.sin(x) / x

def d_sinc(x):
    if (x==0):
        return 0
    else:
        return (np.cos(x) / x) - (np.sin(x) / (x**2))

def partial_x_partial_p(P,X,x):
    # compute the dx_dp component for the Jacobian
    #
    # Input:
    #    P - 3x4 projection matrix
    #    X - Homogenous 3D scene point (4*1)
    #    x - inhomogenous 2D point (2*1)
    # Output:
    #    dx_dp - 2x12 matrix
    w = P[2,:].T @ X
    # x = x
    x = dehomogenize(P @ X)
    dx_dp = np.zeros((2,12))
    dx_dp[0, :4], dx_dp[1, 4:8], dx_dp[0, 8:], dx_dp[1, 8:] = X.T, X.T, -x[0]*X.T, -x[1]*X.T
    dx_dp *= 1/w
    return dx_dp


def parameterize_matrix(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 parameterize_homog(P.reshape(-1,1))


def deparameterize_matrix(m,rows,columns):
    # Deparameterize all optimization variables
    # Input:
    #   m - matrix to be deparameterized
    #   rows - number of rows of the deparameterized matrix
    #   columns - number of rows of the deparameterized matrix
    #
    # Output:
    #    deparameterized matrix
    #
    # For the camera projection, deparameterize it using deparameterize_matrix(p,3,4)
    # where p is the parameterized camera projection matrix

    return deparameterize_homog(m).reshape(rows,columns)


def parameterize_homog(v_bar):
    # Given a homogeneous vector v_bar return its minimal parameterization
    """your code here"""
    # v_bar = v_bar / (np.linalg.norm(v_bar))
    a, b = v_bar[0], v_bar[1:]
    v = (2/sinc(np.arccos(a))) * b
    v_norm = np.linalg.norm(v)
    if (v_norm > np.pi):
        print("normalizing!")
        v = (1 - ( (2*np.pi/v_norm) * (np.ceil((v_norm - np.pi) / (2*np.pi))) )) * v
    return v

def deparameterize_homog(v):
    # Given a parameterized homogeneous vector return its deparameterization
    v_norm = np.linalg.norm(v)
    a = np.cos(v_norm/2)
    b = v * (sinc(v_norm/2)/2)
    v_bar = np.zeros((12,1))
    v_bar[0] = a
    v_bar[1:] = b
    v_bar = v_bar / (np.linalg.norm(v_bar))
    return v_bar

def deparameterize_homog_with_Jacobian(v):
    # Input:
    #    v - homogeneous parameterization vector (11x1 in case of p)
    # Output:
    #    v_bar - deparameterized homogeneous vector
    #    partial_vbar_partial_v - derivative of v_bar w.r.t v (12x11)

    v_norm = np.linalg.norm(v)
    partial_vbar_partial_v = np.zeros((12,11))

    # get vector deparameterizations
    v_bar = deparameterize_homog(v)

    # set partial_a_partial_v
    if (v_norm != 0):
        partial_vbar_partial_v[0, :] = -0.5 * v_bar[1:, 0]

    # set partial_b_partial_v
    iden = np.eye(11)
    if (v_norm != 0):
        partial_vbar_partial_v[1:, :] = (sinc(v_norm/2)/2) * iden + (1/(4*v_norm)) * (d_sinc(v_norm/2) * np.outer(
    else:
        partial_vbar_partial_v[1:, :] = 0.5 * iden

    return v_bar, partial_vbar_partial_v

def data_normalize_with_cov(pts, covarx):
    # data normalization of n dimensional pts
    #
    # Input:
    #    pts - is in inhomogeneous coordinates (2*n)
    #    covarx - covariance matrix associated with x. Has size 2n x 2n, where n is number of points.
    # Outputs:
    #    pts - data normalized points (homogeneious)
    #    T - corresponding transformation matrix
    #    covarx - normalized covariance matrix

    # get transform matrix
    mux, muy, varx, vary = np.mean(pts[0, :]), np.mean(pts[1, :]), np.var(pts[0, :]), np.var(pts[1, :])
    s = np.sqrt(2/(varx + vary))
    T = np.array([[s, 0, -s*mux], [0, s, -s*muy], [0, 0, 1]])

    # get transformed pts
    pts_homo = homogenize(pts)
    pts_trans = T @ pts_homo # transform points

    # get transformed cov matrix
    for i in range(pts.shape[1]):
        J = np.eye(2) * s
        covarx[2*i: 2*i+2, 2*i: 2*i+2] = J @ covarx[2*i: 2*i+2, 2*i: 2*i+2] @ J.T
    covarx_trans = covarx

    # return transformed points and covariance matrix
    return pts_trans, T, covarx_trans

def compute_cost(P, x, X, covarx):
    # Inputs:
    #    P - the camera projection matrix
    #    x - 2D inhomogeneous image points
    #    X - 3D inhomogeneous scene points
    #    covarx - covariance matrix associated with x. Has size 2n x 2n, where n is number of points.
    # Output:
    #    cost - Total reprojection error

    # get projected points
    X_homo = homogenize(X)
    x_proj_homo = P @ X_homo
    x_proj = dehomogenize(x_proj_homo)
    eps = x-x_proj

    # iterate to compute the total cost
    cost = 0
    for j in range(eps.shape[1]):
        eps_i = eps[:, j]
        inv_cov = np.linalg.inv(covarx[2*j:2*j+2, 2*j:2*j+2])
        cost += eps_i.T @ inv_cov @ eps_i
    return cost
```

In [ ]:

```
#Unit Tests (Do not change)

# partial_x_partial_p unit test
def check_values_partial_x_partial_p():
    eps = 1e-8  # Floating point error threshold
    x_2d = np.load('Unit_test/x_2d.npy')
    P = np.load('Unit_test/Projection.npy')
    X = np.load('Unit_test/X.npy')
    target = np.load('Unit_test/dx_dp.npy')
    dx_dp = partial_x_partial_p(P,X,x_2d)
    valid = np.all((dx_dp < target + eps) & (dx_dp > target - eps))
    print(f'Computed partial_x_partial_p is all equal to ground truth +/- {eps}: {valid}')

# deparameterize_homog_with_Jacobian unit test
def check_values_partial_vbar_partial_v():
    eps = 1e-8  # Floating point error threshold
    p = np.load('Unit_test/p.npy')
    dp_dp_target = np.load('Unit_test/dp_dp.npy')
    p_bar_target = np.load('Unit_test/Projection.npy').reshape(12,1)
    p_bar,dp_dp = deparameterize_homog_with_Jacobian(p)
    valid_dp_dp = np.all((dp_dp < dp_dp_target + eps) & (dp_dp > dp_dp_target - eps))
    valid_p_bar = np.all((p_bar < p_bar_target + eps) & (p_bar > p_bar_target - eps))
    valid = valid_dp_dp & valid_p_bar
    print(f'Computed v_bar,partial_vbar_partial_v is all equal to ground truth +/- {eps}: {valid}')

check_values_partial_x_partial_p()
check_values_partial_vbar_partial_v()
```

```
Computed partial_x_partial_p is all equal to ground truth +/- 1e-08: True
Computed v_bar,partial_vbar_partial_v is all equal to ground truth +/- 1e-08: True
```

In [ ]:
```python
def get_parameterization_and_jacobian(P,X,x_inhomo):
    # get the jacobian matrix (2n*11)
    # parameterize the projection matrix
    P_flat_param = parameterize_matrix(P)
    _, ppbarpp = deparameterize_homog_with_Jacobian(P_flat_param)

    # compute jacobians
    n = X.shape[1]
    J = np.zeros((2*n, 11))
    for k in range(n):
        pxppbar = partial_x_partial_p(P,X[:,k],x_inhomo[:,k])
        J[2*k:2*k+2, :] = pxppbar @ ppbarpp
    return P_flat_param, J

def estimate_camera_projection_matrix_nonlinear(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 and homogeneization
    n = X.shape[1]
    covarx = np.eye(2*n)
    x, T, covarx = data_normalize_with_cov(x, covarx)
    X, U = data_normalize(X)
    X_inhomo = dehomogenize(X)
    x_inhomo = dehomogenize(x)
    P = T @ P @ np.linalg.inv(U) # normalize the initial projection matrix


    # you may modify this so long as the cost is computed
    # at each iteration
    x_proj = P @ X
    x_proj_inhomo = dehomogenize(x_proj)
    eps = (x_inhomo - x_proj_inhomo).reshape((2*X.shape[1], 1), order='F')
    cost_prev = compute_cost(P, x_inhomo, X_inhomo, covarx)


    for i in range(max_iters):
        # outer loop
        P = P / np.linalg.norm(P)*np.sign(P[-1,-1]) # parameterization requires normalization, so the original P mu
        # print(P)
        p_flat_param, J = get_parameterization_and_jacobian(P,X,x_inhomo)
        A = J.T @ np.linalg.inv(covarx) @ J
        b = J.T @ np.linalg.inv(covarx) @ eps
        while (True):
            # inner loop
            A_aug = A + lam * np.eye(J.shape[1])
            delta = np.linalg.inv(A_aug) @ b
            p0 = p_flat_param + delta
            P0 = deparameterize_matrix(p0, 3, 4)
            x_proj0 = P0 @ X
            x_proj0_inhomo = dehomogenize(x_proj0)
            eps0 = (x_inhomo - x_proj0_inhomo).reshape((2*X.shape[1], 1), order='F')
            cost0 = compute_cost(P0, x_inhomo, X_inhomo, covarx)
            print('iter %03d Cost %.9f'%(i+1, cost0))
            if (np.abs(1 - cost0/cost_prev) < 1e-7):
                # print("Error reached!")
                return np.linalg.inv(T) @ P @ U
            if (cost0 >= cost_prev):
                lam *= 10
                continue
            else:
                cost_prev = cost0
                P = P0
                eps = eps0
                lam *= 0.1
                break
    return np.linalg.inv(T) @ P @ U



# 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 = estimate_camera_projection_matrix_nonlinear(P_DLT, x, X, max_iters, lam)
time_total=time.time()-time_start
print('took %f secs'%time_total)

print("\n==Correct outputs==")
print("Begins at %.9f; ends at %.9f"%(84.104680130, 82.790238005))
```

```
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 0.107169 secs

==Correct outputs==
Begins at 84.104680130; ends at 82.790238005
```

In [ ]:
```
# Report your P_LM (estimated camera projection matrix nonlinear) final value here!
display_results(P_LM, x, X, 'P_LM')
```

```
P_LM =
[[ 6.09434300e-03 -4.72647784e-03  8.79023509e-03  8.43642847e-01]
 [ 9.02017246e-03 -2.29290848e-03 -6.13330093e-03  5.36660240e-01]
 [ 4.99088669e-06  4.45205098e-06  2.53705055e-06  1.24348261e-03]]
```