CSE 252B: Computer Vision II, Winter 2024 – Assignment 3

Instructor: Ben Ochoa

Assignment Due: Wed, Feb 21, 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 (Programming): Estimation of the Camera Pose - Outlier rejection (20 points)

Download input data from the course website. The file `hw3_points3D.txt` contains the coordinates of 60 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 `hw3_points2D.txt` contains the coordinates of the 60 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 corresponding 3D scene and 2D image points contain both inlier and outlier correspondences. For the inlier correspondences, 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.

The camera calibration matrix was calculated for a 1280×720 sensor and $45\circ$ horizontal field of view lens. The resulting camera calibration matrix is given by

$$K = \begin{bmatrix} 1545.0966799187809 & 0 & 639.5 \\ 0 & 1545.0966799187809 & 359.5 \\ 0 & 0 & 1 \end{bmatrix}$$

For each image point $\mathbf{x} = (x, y, w)^\top = (\tilde{x}, \tilde{y}, 1)^\top$, calculate the point in normalized coordinates $\hat{\mathbf{x}} = K^{-1}\mathbf{x}$.

Determine the set of inlier point correspondences using the M-estimator Sample Consensus (MSAC) algorithm, where the maximum number of attempts to find a consensus set is determined adaptively. For each trial, use the 3-point algorithm of Finsterwalder (as described in the paper by Haralick et al.) to estimate the camera pose (i.e., the rotation R and translation $\mathbf{t}$ from the world coordinate frame to the camera coordinate frame), resulting in up to 4 solutions, and calculate the error and cost for each solution. Note that the 3-point algorithm requires the 2D points in normalized coordinates, not in pixel coordinates. Calculate the projection error, which is the (squared) distance between projected points (the points in 3D projected under the normalized camera projection matrix $\hat{P} = [R \,|\, \mathbf{t}]$) and the measured points in normalized coordinates (hint: the error tolerance is simpler to calculate in pixel coordinates using $P = K[R \,|\, \mathbf{t}]$ than in normalized coordinates using $\hat{P} = [R \,|\, \mathbf{t}]$. You can avoid doing covariance propagation). There must be at least **40 inlier correspondences**.

Hint: this problem has codimension 2.

Report your values for:

- the probability p that as least one of the random samples does not contain any outliers (prob of all-inlier random subset)
- the probability α that a given point is an inlier (inlier prob within a subset)
- the resulting number of inliers
- the number of attempts to find the consensus set

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 normalize(K, x):
    # map the 2D points in pixel coordinates to the 2D points in normalized coordinates
    # Inputs:
    #   K - camera calibration matrix
    #   x - 2D points in pixel coordinates
    # Output:
    #   pts - 2D points in normalized coordinates
    return np.linalg.inv(K) @ x


# load data
x0 = np.loadtxt('hw3_points2D.txt').T
X0 = np.loadtxt('hw3_points3D.txt').T
print('x is', x0.shape)
print('X is', X0.shape)

K = np.array([[1545.0966799187809, 0, 639.5],
              [0, 1545.0966799187809, 359.5],
              [0, 0, 1]])

print('K =')
print(K)
```

```
x is (2, 60)
X is (3, 60)
K =
[[1.54509668e+03 0.00000000e+00 6.39500000e+02]
 [0.00000000e+00 1.54509668e+03 3.59500000e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

In [ ]:
```python
from scipy.stats import chi2
import random
```

```python
def project_and_get_error(P, x, X, K):
    X_homo = homogenize(X)
    x_img_proj_homo = K @ P @ X_homo
    x_img_proj = dehomogenize(x_img_proj_homo)
    error = np.sum((x - x_img_proj)**2, axis=0)
    return error

def compute_MSAC_cost(x, tol, error):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #    tol - reprojection error tolerance
    #    error - pre-computed projection error
    #
    # Output:
    #    cost - total projection error
    #    inlier_ind - inlier indices
    inlier_ind = []
    cost = 0
    count = x.shape[1]
    for n in range(count):
        if (error[n] <= tol):
            cost += error[n]
            inlier_ind.append(n)
        else:
            cost += tol
    return cost, inlier_ind

np.random.seed(38)
def choose_random_column_indices(array_2d, num_columns_to_choose):
    """
    Chooses a random sample of columns from a 2D NumPy array.

    Parameters:
        array_2d (ndarray): The input 2D NumPy array.
        num_columns_to_choose (int): The number of columns to choose randomly.

    Returns:
        random_column_indices: Indices of the randomly chosen columns.
    """
    # Number of columns in the array
    num_columns = array_2d.shape[1]
    # Randomly choose column indices
    random_column_indices = np.random.choice(num_columns, size=num_columns_to_choose, replace=False)
    return random_column_indices

def solve_cubic(a, b, c, d):
    """
    Solves a cubic equation of the form ax^3 + bx^2 + cx + d = 0

    Parameters:
        a, b, c, d (float): Coefficients of the cubic equation.

    Returns:
        ndarray: An array containing the roots of the cubic equation.
    """
    coefficients = [a, b, c, d]
    roots = np.roots(coefficients)
    return roots

def solve_quadratic(b, c, d):
    """
    Solves a cubic equation of the form bx^2 + cx + d = 0

    Parameters:
        b, c, d (float): Coefficients of the quadratic equation.

    Returns:
        ndarray: An array containing the roots of the quadratic equation.
    """
    coefficients = [b, c, d]
    roots = np.roots(coefficients)
    return roots
```

```python
def solveLambda(d1, d2, d3, cosAlpha, cosBeta, cosGamma, sin2Alpha, sin2Beta, sin2Gamma, a, b, c):
    g = c**2 * (c**2 * sin2Beta - b**2 * sin2Gamma)
    h = b**2 * (b**2 - a**2) * sin2Gamma + c**2 * (c**2 + 2 * a**2) * sin2Beta + 2 * b**2 * c**2 * (cosAlpha*cosBe
    i = b**2 * (b**2 - c**2) * sin2Alpha + a**2 * (a**2 + 2 * c**2) * sin2Beta + 2 * a**2 * b**2 * (cosAlpha*cosBe
    j = a**2 * (a**2 * sin2Beta - b**2 * sin2Alpha)
    roots = solve_cubic(g, h, i, j)
    roots = np.real(roots[np.isreal(roots)])
    return roots[0]


def get_camera_points(x, X, K):
    # normalize to normalized coordinates first
    x_normal_homo = np.linalg.inv(K) @ homogenize(x)
    # unitization
    d_123 = x_normal_homo / (np.sign(x_normal_homo[-1, :])*np.linalg.norm(x_normal_homo, axis=0))
    # solve lambda
    d1, d2, d3 = d_123[:, 0], d_123[:, 1], d_123[:, 2]
    cosAlpha, cosBeta, cosGamma = d2@d3, d1@d3, d1@d2
    sin2Alpha, sin2Beta, sin2Gamma = 1-cosAlpha**2, 1-cosBeta**2, 1-cosGamma**2
    a, b, c = np.linalg.norm(X[:,1]-X[:,2]), np.linalg.norm(X[:,2]-X[:,0]), np.linalg.norm(X[:,1]-X[:,0])
    a2, b2, c2 = a**2, b**2, c**2
    lambda0 = solveLambda(d1, d2, d3, cosAlpha, cosBeta, cosGamma, sin2Alpha, sin2Beta, sin2Gamma, a, b, c)
    # solve u,v, and finally solving s
    s = []
    A, B, C, D, E, F = b2 * (lambda0 + 1), -b2 * cosAlpha, ( b2 - a2 - lambda0 * c2 ), -b2 * lambda0 * cosGamma, c
    p_root, q_root = (B**2) - A*C, (E**2) - C*F # in case u has no solution
    if ((p_root < 0) or (q_root < 0)):
        return []
    p, q = np.sqrt(p_root), np.sign(B*E - C*D) * np.sqrt(q_root)
    m, n = [(-B + p) / C, (-B - p) / C], [-(E - q) / C, -(E + q) / C]
    for j in range(len(m)):
        mj, nj = m[j], n[j]
        Au = b2 - (mj**2) * c2
        Bu = 2 * ((c2 * (cosBeta - nj) * mj) - (b2 * cosGamma))
        Cu = -c2 * (nj**2) + (2 * c2 * nj) * cosBeta + b2 - c2
        # solve slack variables u,v
        us = solve_quadratic(Au, Bu, Cu)
        us = np.real(us[np.isreal(us)]) # solve for u
        # back-projection to camera frame
        for u in us:
            v = u * mj + nj
            # print("My b, v, cosBeta: {}, {}, {}".format(b, v, cosBeta))
            s1 = np.sqrt(b2 / (1 + (v**2) - (2*v*cosBeta))) # remember to square b !!!
            s2 = u * s1
            s3 = v * s1
            # print("My s1, s2, s3: {}, {}, {}".format(s1, s2, s3))
            if (s1 > 0 and s2 > 0 and s3 > 0):
                s.append([s1, s2, s3])

    # get camera coordinates (up to 4 solutions)
    cam_coors = []
    for s_123 in s:
        cam_coor = np.zeros((3,3))
        cam_coor[:, 0], cam_coor[:, 1], cam_coor[:, 2] = s_123[0]*d1, s_123[1]*d2, s_123[2]*d3
        cam_coors.append(cam_coor)
    return cam_coors

def umeyama(cam_coor, X):
    # cam_coor, X: Inhomogeneious camera and world coordinates
    cam_mean, X_mean = np.mean(cam_coor, axis=1), np.mean(X, axis=1)
    d, n = cam_coor.shape[0], cam_coor.shape[1]
    S = np.zeros((d,d))
    for j in range(n):
        S += np.outer(cam_coor[:, j] - cam_mean, X[:, j] - X_mean)
    U, sig, Vt = np.linalg.svd(S)
    if (np.linalg.det(U) * np.linalg.det(Vt.T) < 0):
        I = np.eye(3)
        I[-1, -1] = -1
        R = U @ I @ Vt # 3x3
    else:
        R = U @ Vt
    T = cam_mean - R @ X_mean # 3x1
    # load projection matrix and return
    P = np.zeros((3,4))
    P[:,:3], P[:,3] = R, T
```

```python
    return P

def finsterWadler_get_pose(x, X, K):
    # estimate pose using FW algorithm (P3P)
    cam_coors = get_camera_points(x, X, K)
    PList = []
    for cam_coor in cam_coors:
        P = umeyama(cam_coor, X)
        PList.append(P)
    return PList

def determine_inliers(x, X, K, thresh, tol, p):
    # Inputs:
    #     x - 2D inhomogeneous image points
    #     X - 3D inhomogeneous scene points
    #     K - camera calibration matrix
    #     thresh - cost threshold
    #     tol - reprojection error tolerance
    #     p - probability that as least one of the random samples does not contain any outliers
    #
    # Output:
    #     consensus_min_cost - final cost from MSAC
    #     consensus_min_cost_model - camera projection matrix P
    #     inliers - list of indices of the inliers corresponding to input data
    #     trials - number of attempts taken to find consensus set

    """your code here"""
    max_trials = np.inf
    consensus_min_cost = np.inf
    n = x.shape[1]
    s = 3 # sample size = 3
    trials = 0

    inliers = [0]*39
    while (len(inliers) < 40): # get at least 40 inliers
        while ((trials < max_trials) and (consensus_min_cost > thresh)): # "for" loop's number of rounds executing
            samp_inds = choose_random_column_indices(X, s)
            X_samp = X[:, samp_inds]
            x_samp = x[:, samp_inds]

            # for a given set of correspondances, there can be multiple camera coordinates, hence multiple poses (n
            PList = finsterWadler_get_pose(x_samp, X_samp, K)

            # evaluate error for each possible pose
            for P in PList:
                error = project_and_get_error(P, x, X, K)
                # print(error, tol)
                cost, inlier_ind = compute_MSAC_cost(x, tol, error)
                # print(cost, inlier_ind)
                if (cost < consensus_min_cost):
                    consensus_min_cost = cost
                    consensus_min_cost_model = P
                    w = len(inlier_ind) / n
                    # print(p, w, s)
                    max_trials = np.log(1-p) / np.log(1-(w**s))
                    # print(max_trials)
            trials += 1
        error_min = project_and_get_error(consensus_min_cost_model, x, X, K)
        cost_min, inliers = compute_MSAC_cost(x, tol, error_min)

    print("Final minimal cost: {}".format(cost_min))
    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 100
codim, alpha, sigma = 2, 0.95, 1
tol = chi2.ppf(alpha, codim) * sigma
p = 0.999

tic = time.time()

cost_MSAC, P_MSAC, inliers, trials = determine_inliers(x0, X0, K, thresh, tol, p)

# choose just the inliers
x = x0[:, inliers]
```

```
X = X0[:, inliers]

toc = time.time()
time_total = toc-tic

# display the results
print(f'took {time_total} secs')
print(f'iterations: {trials}')
print(f'inlier count: {len(inliers)}')
print(f'MSAC Cost: {cost_MSAC:.9f}')
print('P = ')
print(P_MSAC)
print('inliers: ', inliers)

# display required values
print(f"p = {p}")
print(f"alpha = {alpha}")
print(f"tolerance = {tol}")
print(f"num_inliers = {len(inliers)}")
print(f"num_attempts = {trials}")
```

```
Final minimal cost: 208.0123498384926
took 0.04157686233520508 secs
iterations: 42
inlier count: 42
MSAC Cost: 208.012349838
P =
[[  0.28909222  -0.68063079   0.6731771    6.69890465]
 [  0.6600831   -0.36757721  -0.65511625   7.38891396]
 [  0.69333685   0.63374184   0.34300916 175.72248156]]
inliers:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 3
3, 34, 35, 36, 39, 40, 41, 42, 43, 45, 46, 47, 48, 49]
p = 0.999
alpha = 0.95
tolerance = 5.991464547107979
num_inliers = 42
num_attempts = 42
```

Problem 2 (Programming): Estimation of the Camera Pose - Linear Estimate (30 points)

Estimate the normalized camera projection matrix $\hat{\mathbf{P}}_{\text{linear}} = [\mathbf{R}_{\text{linear}} \,|\, \mathbf{t}_{\text{linear}}]$ from the resulting set of inlier correspondences using the linear estimation method (based on the EPnP method) described in lecture. Report the resulting $\mathbf{R}_{\text{linear}}$ and $\mathbf{t}_{\text{linear}}$.

In [ ]:
```
import time

def sum_of_square_projection_error(P, x, X, K):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
    #    cost - Sum of squares of the reprojection error
    X_homo = homogenize(X)
    x_img_proj_homo = K @ P @ X_homo
    x_img_proj = dehomogenize(x_img_proj_homo)
    cost = np.sum((x - x_img_proj)**2, axis=0)
    return cost

def umeyama(cam_coor, X):
    # cam_coor, X: Inhomogeneious camera and world coordinates
    cam_mean, X_mean = np.mean(cam_coor, axis=1), np.mean(X, axis=1)
    d, n = cam_coor.shape[0], cam_coor.shape[1]
    S = np.zeros((d,d))
    for j in range(n):
        S += np.outer(cam_coor[:, j] - cam_mean, X[:, j] - X_mean)
    U, sig, Vt = np.linalg.svd(S)
    if (np.linalg.det(U) * np.linalg.det(Vt.T) < 0):
        I = np.eye(3)
        I[-1, -1] = -1
```

```
        R = U @ I @ Vt # 3x3
    else:
        R = U @ Vt
    T = cam_mean - R @ X_mean # 3x1
    # load projection matrix and return
    P = np.zeros((3,4))
    P[:,:3], P[:,3] = R, T
    return R, T, P

def estimate_camera_pose_linear(x, X, K):
    # Inputs:
    #    x - 2D inlier points
    #    X - 3D inlier points
    # Output:
    #    P - normalized camera projection matrix
    # normalize x
    x_normal = dehomogenize(np.linalg.inv(K) @ homogenize(x))

    # compute world ctl points
    X_mean = np.mean(X, axis=1)
    X_cov = np.cov(X)
    Lambda, U = np.linalg.eigh(X_cov)
    Lambda, U = Lambda[::-1], U[:, ::-1]
    U = np.hstack([np.zeros((3,1)), U])
    C1234world = U + X_mean[:,None]
    C1world, C2world, C3world, C4world = C1234world[:, 0], C1234world[:, 1], C1234world[:, 2], C1234world[:, 3]

    # compute parameterizations
    n = x.shape[1]
    A = np.hstack([(C2world-C1world)[:, None], (C3world-C1world)[:, None], (C4world-C1world)[:, None]])
    A = np.tile(A, (n, 1))
    b = X.reshape(-1, order="F")
    b = b - np.tile(C1world, n)
    alpha234 = np.zeros(3*n)
    for j in range(n):
        Aj = A[3*j:3*j+3, :]
        Aj_inv = np.linalg.inv(Aj)
        bj = b[3*j:3*j+3]
        alpha234[3*j:3*j+3] = Aj_inv @ bj
        alpha234[3*j] = alpha234[3*j]

    # compute camera ctl points
    M = np.zeros((2*n, 12))
    for j in range(n):
        alphaj2, alphaj3, alphaj4 = alpha234[3*j], alpha234[3*j+1], alpha234[3*j+2]
        alphaj1 = 1 - alphaj2 - alphaj3 - alphaj4
        xj, yj = x_normal[:, j][0], x_normal[:, j][1]
        alphaj = [alphaj1, alphaj2, alphaj3, alphaj4]
        for i in range(len(alphaj)):
            alphaji = alphaj[i]
            M[2*j, 3*i], M[2*j+1, 3*i+1], M[2*j, 3*i+2], M[2*j+1, 3*i+2] = alphaji, alphaji, -alphaji*xj, -alphaji
    U, sig, Vt = np.linalg.svd(M)
    CCam = Vt[-1, :]
    C1cam = CCam[0:3][:, None]
    C2cam = CCam[3:6][:, None]
    C3cam = CCam[6:9][:, None]
    C4cam = CCam[9:][:, None]

    # compute camera coordinates
    sigma2_X = np.trace(X_cov)
    alpha234_cols = np.reshape(alpha234, (3, -1), order='F')
    alpha2, alpha3, alpha4 = alpha234_cols[0, :], alpha234_cols[1, :], alpha234_cols[2, :]
    alpha1 = np.ones(n) - alpha2 - alpha3 - alpha4
    X_cam = alpha1*C1cam+alpha2*C2cam+alpha3*C3cam+alpha4*C4cam
    ZCam_mean = np.mean(X_cam[2, :])
    sigma2_Xcam = np.var(X_cam[0, :])+np.var(X_cam[1, :])+np.var(X_cam[2, :])
    beta = np.sqrt(sigma2_X / sigma2_Xcam)
    if(np.sign(ZCam_mean) < 0):
        beta = -beta
    X_cam = beta * X_cam

    # umeyama alignment for camera pose matrix
    R, t, P = umeyama(X_cam, X)
    return P

tic = time.time()
```

```
P_linear = estimate_camera_pose_linear(x, X, K)
toc = time.time()
time_total = toc - tic

# display the results
print(f'took {time_total} secs')
print('R_linear = ')
print(P_linear[:, 0:3])
print('t_linear = ')
print(P_linear[:, -1])
```

```
took 0.003988504409790039 secs
R_linear =
[[ 0.28064893 -0.68881831  0.6684052 ]
 [ 0.65915863 -0.36787693 -0.65587839]
 [ 0.6976719   0.62465663  0.35079626]]
t_linear =
[  5.79182291   7.33117459 177.06840638]
```

Problem 3 (Programming): Estimation of the Camera Pose - Nonlinear Estimate (30 points)

Use $\mathrm{R}_{\mathrm{linear}}$ and $\mathbf{t}_{\mathrm{linear}}$ as an initial estimate to an iterative estimation method, specifically the Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the camera pose that minimizes the projection error under the normalized camera projection matrix $\hat{\mathrm{P}} = [\mathrm{R} \,|\, \mathbf{t}]$. You must parameterize the camera rotation using the angle-axis representation $\boldsymbol{\omega}$ (where $[\boldsymbol{\omega}]_\times = \ln \mathrm{R}$) of a 3D rotation, which is a 3-vector.

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 rotation $\boldsymbol{\omega}_{\mathrm{LM}}$ and R_{LM}, and the camera translation $\mathbf{t}_{\mathrm{LM}}$.

In [ ]:
```
from scipy.linalg import block_diag

# 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 skew(w):
    # Returns the skew-symmetrix represenation of a vector
    w_skew = np.zeros((3, 3))
    w_skew[0, 1], w_skew[1, 0] = -w[2], w[2]
    w_skew[0, 2], w_skew[2, 0] = w[1], -w[1]
    w_skew[1, 2], w_skew[2, 1] = -w[0], w[0]
    return w_skew


def parameterize_rotation_matrix(R):
    # Parameterizes rotation matrix into its axis-angle representation
    _, _, Vt = np.linalg.svd(R - np.eye(3))
    v = Vt[-1, :] # V is the last row of Vt
    vHat = np.array([R[2, 1]-R[1, 2], R[0, 2]-R[2, 0], R[1, 0]-R[0, 1]])
    cosTheta, sinTheta = (np.trace(R)-1)/2, v@vHat/2
    theta = np.arctan2(sinTheta, cosTheta)

    if (np.abs(theta) < 1e-5): # small rotation
        w = (1/2) * vHat
        return w, theta
    w = theta * v
    wNorm = np.linalg.norm(w)
    if (wNorm > np.pi):
        w = (1 - (2*np.pi/wNorm)*(np.ceil((wNorm-np.pi)/(2*np.pi)))) * w
    return w[:, None], theta
```

```python
def deparameterize_rotation_matrix(w):
    # Deparameterizes to get rotation matrix
    theta = np.linalg.norm(w)
    wSkew = skew(w)
    I = np.eye(3)
    if (np.abs(theta) < 1e-5): # small rotation
        R = I + wSkew
        return R
    R = np.cos(theta)*I + sinc(theta)*wSkew + (((1-np.cos(theta))/(theta**2)) * np.outer(w, w))
    return R

def data_normalize(pts):
    # Input:
    #    pts - 3D scene points
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix
    dim = pts.shape[0]
    pts_homo = homogenize(pts)
    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, T

def normalize_with_cov(K, x, covarx):
    # Inputs:
    #    K - camera calibration matrix
    #    x - 2D points in pixel coordinates
    #    covarx - covariance matrix (2*2)
    #
    # Outputs:
    #    pts - 2D points in normalized coordinates
    #    covarx - normalized covariance matrix
    # project to normalized coordinates
    n = x.shape[1]
    K_inv = np.linalg.inv(K)

    x_homo = homogenize(x)
    pts_homo = K_inv @ x_homo
    pts = dehomogenize(pts_homo)

    # propagate cov matrix
    J = np.array([[K_inv[0,0], K_inv[0,1]],[0, K_inv[1,1]]])
    # J = K_inv[:2, :2]
    for j in range(n):
        covarx[2*j:2*j+2, 2*j:2*j+2] = J @ covarx[2*j:2*j+2, 2*j:2*j+2] @ J.T
    return pts, covarx

def partial_x_hat_partial_w(R, w, t, X):
    # Compute the (partial x_hat) / (partial omega) component of the jacobian
    # Inputs:
    #    R - 3x3 rotation matrix
    #    w - 3x1 axis-angle parameterization of R
    #    t - 3x1 translation vector
    #    X - 3D inlier point
    #
    # Output:
    #    dx_hat_dw -  matrix of size 2x3
    # get dxdXrot
    # print(X.shape, R.shape, w.shape, t.shape)
    w = w[:, 0]
    t = t[:, 0]
    X = X[:, 0]

    x_proj =  dehomogenize(R @ X + t)
    Xrot = R @ X
    wHat = Xrot[2] + t[2]
    dx_hat_dXrot = np.array([[1/wHat, 0, -x_proj[0]/wHat],[0, 1/wHat, -x_proj[1]/wHat]])

    # get dXrotdw
    theta = np.linalg.norm(w)
    if (np.abs(theta) < 1e-5):
        dXrot_dw = skew(-X)
```

```python
    else:
        s = (1-np.cos(theta))/(theta**2)
        ds_dtheta = (theta*np.sin(theta) - 2*(1-np.cos(theta)))/(theta**3)
        dtheta_dw = (1/theta)*w.T
        xSkew = skew(-X)
        wSkew = skew(w)
        dXrot_dw = sinc(theta) * xSkew + \
                   np.cross(w, X)[:, None] * d_sinc(theta) * dtheta_dw + \
                   np.cross(w, np.cross(w, X))[:, None] * ds_dtheta * dtheta_dw + \
                   s * ((wSkew @ xSkew) + skew(-(np.cross(w, X))))

    # multiply by chain rule
    dx_hat_dw = dx_hat_dXrot @ dXrot_dw
    return dx_hat_dw


def partial_x_hat_partial_t(R, t, x_norm, X):
    # Compute the (partial x_hat) / (partial t) component of the jacobian
    # Inputs:
    #    R - 3x3 rotation matrix
    #    t - 3x1 translation vector
    #    x_norm - 2D projected point in normalized coordinates
    #    X - 3D inlier point
    #
    # Output:
    #    dx_hat_dt -  matrix of size 2x3
    t = t[:, 0]
    X = X[:, 0]
    x_norm = x_norm[:, 0]

    Xrot = R @ X
    wHat = Xrot[2] + t[2]
    dx_hat_dt = np.array([[1/wHat, 0, -x_norm[0]/wHat],[0, 1/wHat, -x_norm[1]/wHat]])
    return dx_hat_dt

def compute_cost(P, x, X, covarx):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D ground truth image points in normalized coordinates
    #    X - 3D groundtruth scene points
    #    covarx - covariance matrix
    #
    # Output:
    #    cost - total projection error
    X_homo = homogenize(X)
    x_img_proj_homo = P @ X_homo
    x_img_proj = dehomogenize(x_img_proj_homo)
    n = x_img_proj.shape[1]
    epsilon = x - x_img_proj
    cost = 0
    for j in range(n):
        covarxj, epsilonj = covarx[2*j:2*j+2, 2*j:2*j+2], epsilon[:, j]
        cost += epsilonj.T @ np.linalg.inv(covarxj) @ epsilonj
    return cost
```

In [ ]:

```python
# Unit Tests (Do not change)

# parameterize and deparameterize unit test
def check_values_parameterize():
    eps = 1e-8  # Floating point error threshold
    w = np.load('unit_test/omega.npy')
    R = np.load('unit_test/rotation.npy')
    w_param, _ = parameterize_rotation_matrix(R)
    R_deparam = deparameterize_rotation_matrix(w)

    param_valid = np.all(np.abs(w_param - w) < eps)
    deparam_valid = np.all(np.abs(R_deparam - R) < eps)

    print(f'Parameterized rotation matrix is equal to the given value +/- {eps}: {param_valid}')
    print(f'Deparameterized rotation matrix is equal to the given value +/- {eps}: {deparam_valid}')

# partial_x_hat_partial_w and partial_x_hat_partial_t unit test
def check_values_jacobian():
    eps = 1e-8  # Floating point error threshold
    w = np.load('unit_test/omega.npy')
```

```python
    R = np.load('unit_test/rotation.npy')
    x = np.load('unit_test/point_2d.npy')
    X = np.load('unit_test/point_3d.npy')
    t = np.load('unit_test/translation.npy')
    dx_hat_dw_target = np.load('unit_test/partial_x_partial_omega.npy')
    dx_hat_dt_target = np.load('unit_test/partial_x_partial_t.npy')

    dx_hat_dw = partial_x_hat_partial_w(R, w, t, X)
    dx_hat_dt = partial_x_hat_partial_t(R, t, x, X)
    w_valid = np.all(np.abs(dx_hat_dw - dx_hat_dw_target) < eps)
    t_valid = np.all(np.abs(dx_hat_dt - dx_hat_dt_target) < eps)

    print(f'Computed partial_x_hat_partial_w is equal to the given value +/- {eps}: {w_valid}')
    print(f'Computed partial_x_hat_partial_t is equal to the given value +/- {eps}: {t_valid}')

check_values_parameterize()
check_values_jacobian()
```

```
Parameterized rotation matrix is equal to the given value +/- 1e-08: True
Deparameterized rotation matrix is equal to the given value +/- 1e-08: True
Computed partial_x_hat_partial_w is equal to the given value +/- 1e-08: True
Computed partial_x_hat_partial_t is equal to the given value +/- 1e-08: True
```

In [ ]:

```python
def get_jacobian(R, w, t, x_norm, X):
    # get jacobian w.r.t. w and t
    n = X.shape[1]
    J = np.zeros((2*n, 6))
    t = t[:, None]
    for j in range(n):
        Xj = X[:, j]
        x_normj = x_norm[:, j]
        Xj = Xj[:, None]
        x_normj = x_normj[:, None]
        dxj_dw = partial_x_hat_partial_w(R, w, t, Xj)
        dxj_dt = partial_x_hat_partial_t(R, t, x_normj, Xj)
        J[2*j:2*j+2, :3], J[2*j:2*j+2, 3:] = dxj_dw, dxj_dt
    return J

def get_epsilon(P, X, x):
    x_proj_homo = P @ homogenize(X) # prepare epsilons and lambda
    x_proj = dehomogenize(x_proj_homo)
    epsilon = x - x_proj
    return epsilon

def estimate_camera_pose_nonlinear(P, x, X, K, max_iters, lam):
    # Inputs:
    #    P - initial estimate of camera pose
    #    x - 2D inliers
    #    X - 3D inliers
    #    K - camera calibration matrix
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    #
    # Output:
    #    P - Final camera pose obtained after convergence
    n_points = X.shape[1]
    covarx = np.eye(2 * n_points)
    R, t = P[:3, :3], P[:3, 3]
    C = -R.T @ t
    C = C[:, None]
    # normalize 3d points and the projection matrix
    X, U = data_normalize(X)
    x, covarx = normalize_with_cov(K, x, covarx)
    C_dn = dehomogenize(U @ homogenize(C))
    t = -R @ C_dn
    # P[:3, :3], P[:3, 3] = R, t[:, 0] # don't do assignment using pointers! Re-assign a new one!
    P = np.hstack((R, t))
    x_proj_homo = P @ homogenize(X) # prepare normalized projections, epsilons, and lambda
    x_proj = dehomogenize(x_proj_homo)
    epsilon = x - x_proj

    # estimate camera pose (calibrated)
    cost = compute_cost(P, x, X, covarx)
    print(f"Initial iteration-0 Cost: {cost:.09f}  Avg cost per point: {cost / n_points}:")
    for i in range(max_iters):
        R, t = P[:3, :3], P[:3, 3]
```

```python
        w, _ = parameterize_rotation_matrix(R) # compute jacobians
        # print("Getting jacobian")
        J = get_jacobian(R, w, t, x_proj, X)
        while (True):
            A = J.T @ np.linalg.inv(covarx) @ J + lam * np.eye(6) # solve delta
            b = J.T @ np.linalg.inv(covarx) @ epsilon.reshape(-1, order="F")
            delta = np.linalg.inv(A) @ b
            w_0 = w + delta[:3][:, None]
            t_0 = t + delta[3:]
            # print("w, t, delta")
            # print(w, t, delta)
            R_0 = deparameterize_rotation_matrix(w_0) # project and re-calculate epsilon
            P_0 = np.hstack((R_0, t_0[:, None]))
            epsilon_0 = get_epsilon(P_0, X, x) # get 2*n updated epsilon
            cost_0 = compute_cost(P_0, x, X, covarx)
            if (cost_0 >= cost):
                lam *= 10
                # print("Shouldn't go here")
                continue
            else:
                # print("One update!")
                P = P_0
                epsilon = epsilon_0
                lam = lam / 10
                prev_cost = cost
                cost = cost_0
                break
        print(f'iter: {i + 1:03d}  Cost: {cost:.09f}  Avg cost per point: {cost / n_points}')
        if (1 - cost_0/prev_cost < 1e-10):
            break

    # denormalize projection estimate
    C_norm = (-P[:, 0:3].T @ P[:,-1])[:, None]
    C = np.linalg.inv(U) @ homogenize(C_norm)
    t = -P[:, 0:3] @ dehomogenize(C)
    P = np.hstack((P[:, 0:3], t))
    return P

# LM hyperparameters
lam = .001
max_iters = 100

tic = time.time()
P_LM = estimate_camera_pose_nonlinear(P_linear, x, X, K, max_iters, lam)
# P_LM = LM(P_linear, x, X, K, max_iters, lam)
w_LM, _ = parameterize_rotation_matrix(P_LM[:, 0:3])
toc = time.time()
time_total = toc-tic

# display the results
print('took %f secs'%time_total)
print('w_LM = ')
print(w_LM)
print('R_LM = ')
print(P_LM[:,0:3])
print('t_LM = ')
print(P_LM[:,-1])
```

```
Initial iteration-0 Cost: 902.217482914  Avg cost per point: 21.481368640817724:
iter: 001  Cost: 54.626658073  Avg cost per point: 1.3006347160320142
iter: 002  Cost: 54.498614462  Avg cost per point: 1.2975860586128585
iter: 003  Cost: 54.498595578  Avg cost per point: 1.2975856090041074
iter: 004  Cost: 54.498595577  Avg cost per point: 1.2975856089686546
took 0.298404 secs
w_LM =
[[ 1.34089787]
 [-0.03061548]
 [ 1.41223214]]
R_LM =
[[ 0.28041958 -0.68901753  0.66829612]
 [ 0.65940811 -0.36765957 -0.65574948]
 [ 0.69752835  0.62456487  0.35124481]]
t_LM =
[   5.78161116    7.32996718 175.86386778]
```