CSE 252B: Computer Vision II, Winter 2026 – Assignment 3

Instructor: Ben Ochoa

Assignment Due: Wed, Feb 18, 11:59 PM

Name: Evan Cheng

PID: A69042831

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° horizontal field of view lens. The resulting camera calibration matrix is given by

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

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

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 $\mathtt{R}$ 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 $\hat{\mathtt{P}} = [\mathtt{R} \,|\, \mathbf{t}]$) and the measured points in normalized coordinates (hint: the error tolerance is simpler to calculate in pixel coordinates using $\mathtt{P} = \mathtt{K}[\mathtt{R} \,|\, \mathbf{t}]$ than in normalized coordinates using $\hat{\mathtt{P}} = [\mathtt{R} \,|\, \mathbf{t}]$. 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
- the probability α that a given point is an inlier
- the resulting number of inliers
- the number of attempts to find the consensus set

In [37]:
```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

    """your code here"""
    x = homogenize(x)

    pts = np.linalg.inv(K) @ x
    return pts


# 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 [38]:
```python
from scipy.stats import chi2

def compute_MSAC_cost(P, x, X, K, tol):
    # Compute the MSAC cost
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - measured 2D image points in pixel coordinates
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #    tol - reprojection error tolerance
    #
    # Output:
    #    cost - total projection error

    """your code here"""
    P_pixel = K @ P
    projected = dehomogenize(P_pixel @ homogenize(X))

    errors = np.sum((x - projected)**2, axis=0)

    cost = np.sum(np.minimum(errors, tol))
    return cost

def determine_inliers(x, X, K, thresh, tol, p):
    # Determine inliers using MSAC
    # 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 - normalized 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"""

    trials = 0
    max_trials = 1000
    consensus_min_cost = np.inf
    consensus_min_cost_model = np.zeros((3, 4))
    inliers = []
    model = np.zeros((3,4))

    while (trials < max_trials and consensus_min_cost > thresh):
        sample_i = np.random.choice(X.shape[1], 3, replace=False)

        #inhomogeneous X1, X2, X3
        X_sample = X[:,sample_i]
        X1, X2, X3 = X_sample.T
        a_2 = (X2[0] - X3[0])**2 + (X2[1] - X3[1])**2 + (X2[2] - X3[2])**2
        b_2 = (X1[0] - X3[0])**2 + (X1[1] - X3[1])**2 + (X1[2] - X3[2])**2
        c_2 = (X1[0] - X2[0])**2 + (X1[1] - X2[1])**2 + (X1[2] - X2[2])**2

        #normalized x to find d
        x_h = normalize(K, x)
        x_sample = x_h[:,sample_i]
        d1, d2, d3 = x_sample.T
        d1 = d1 / (np.sign(d1[2]) * np.linalg.norm(d1))
        d2 = d2 / (np.sign(d2[2]) * np.linalg.norm(d2))
        d3 = d3 / (np.sign(d3[2]) * np.linalg.norm(d3))

        #cos and sin squared
        cos_alpha = d2.T @ d3
        cos_beta = d1.T @ d3
        cos_gamma = d1.T @ d2
        sin2_alpha = 1 - cos_alpha**2
        sin2_beta  = 1 - cos_beta**2
        sin2_gamma = 1 - cos_gamma**2

        #coefficients, roots, and lambda naught
        G = c_2 * (c_2 * sin2_beta - b_2 * sin2_gamma)
        H = b_2 * (b_2 - a_2) * sin2_gamma + c_2 * (c_2 + 2 * a_2) * sin2_beta \
            + 2 * b_2 * c_2 * (-1 + cos_alpha * cos_beta * cos_gamma)
        I = b_2 * (b_2 - c_2) * sin2_alpha + a_2 * (a_2 + 2 * c_2) * sin2_beta \
            + 2 * a_2 * b_2 * (-1 + cos_alpha * cos_beta * cos_gamma)
        J = a_2 * (a_2 * sin2_beta - b_2 * sin2_alpha)
        roots = np.roots((G, H, I, J))
        real_roots = roots[np.isreal(roots)]

        if len(real_roots) == 0:
            trials += 1
            continue
        lambda_0 = np.real(real_roots[0])

        #coefficients for m1, m2, n1, n2, p, q
        A = b_2 * (lambda_0 + 1)
        B = -b_2 * cos_alpha
        C = -a_2 + b_2 - lambda_0 * c_2
        D = -b_2 * lambda_0 * cos_gamma
        E = cos_beta * (a_2 + lambda_0 * c_2)
        F = lambda_0 * (b_2 - c_2) - a_2

        p_check = B**2 - A * C
        q_check = E**2 - C * F
        if p_check < 0 or q_check < 0:
            trials += 1
            continue
        p_lam = np.sqrt(p_check)
        q_lam = np.sign(B * E - C * D) * np.sqrt(q_check)

        m1 = (-B + p_lam) / C
        m2 = (-B - p_lam) / C
        n1 = -(E - q_lam) / C
        n2 = -(E + q_lam) / C


        #coefficients for two real roots u
        for (m, n) in [(m1, n1), (m2, n2)]:
            A_u = b_2 - m**2 * c_2
            B_u = c_2 * (cos_beta - n) * m - b_2 * cos_gamma
            C_u = -c_2 * n**2 + 2 * c_2 * n * cos_beta + b_2 - c_2

            roots = np.roots((A_u, B_u, C_u))
            real_roots = roots[np.isreal(roots)].real

            #calculate v, R, t, model
            for u in real_roots:
                v = u * m + n
                denom = 1 + v**2 - 2 * v * cos_beta
                s1 = np.sqrt(b_2 / (denom))
                s2 = u * s1
                s3 = v * s1

                X_cam = np.zeros((3,3))
                if s1 > 0 and s2 > 0 and s3 > 0:
                    X_cam[:,0] = s1 * d1
                    X_cam[:,1] = s2 * d2
                    X_cam[:,2] = s3 * d3

                    mu_X = np.mean(X_sample, axis=1).reshape(3,1)
                    mu_X_cam = np.mean(X_cam, axis=1).reshape(3,1)

                    S = (X_cam - mu_X_cam) @ (X_sample - mu_X).T
                    U, _, V_T = np.linalg.svd(S)

                    if np.linalg.det(U) * np.linalg.det(V_T.T) < 0:
                        R = U @ np.diag([1,1,-1]) @ V_T
                    else:
                        R = U @ V_T

                    t = mu_X_cam - R @ mu_X

                    model[:,0:3] = R
                    model[:,-1] = t.reshape(3,)

                    #calculate cost
                    cost = compute_MSAC_cost(model, x, X, K, tol)

                    #if cost bettter, calculate number of inliers, w, maxTrials
                    if cost < consensus_min_cost:
                        consensus_min_cost = cost
                        consensus_min_cost_model = model

                        measured = dehomogenize(normalize(K, x))
                        projected = dehomogenize(model @ homogenize(X))

                        errors = np.sum((measured - projected)**2, axis=0)
                        num_inliers = np.sum(errors <= tol)

                        w = num_inliers / X.shape[1]
                        if w > 0 and w < 1:
                            max_trials = np.log(1 - p) / np.log(1 - w**3)

        trials += 1

        if trials > 100:
            break

    #calculate error of each point using consensus_min_cost_model and set of inliers
    if consensus_min_cost < np.inf:
        measured = dehomogenize(normalize(K, x))
        projected = dehomogenize(consensus_min_cost_model @ homogenize(X))
        errors = np.sum((measured - projected)**2, axis=0)
        inliers = np.where(errors <= tol)[0].tolist()
    else:
        inliers = []

    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 100
tol = chi2.ppf(0.95, df=2)
p = 0.99
alpha = 0.95

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}")
```

```
took 0.01888895034790039 secs
iterations: 68
inlier count: 56
MSAC Cost: 354.468112447
P =
[[-1.41671398e-01 -9.06063571e-01 -3.98720479e-01 -6.05948537e+01]
 [ 8.95822788e-01  5.40451153e-02 -4.41112976e-01  4.62276916e+01]
 [ 4.21225292e-01 -4.19675983e-01  8.04015748e-01  1.45045086e+01]]
inliers:  [0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32,
33, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
p = 0.99
alpha = 0.95
tolerance = 5.991464547107979
num_inliers = 56
num_attempts = 68
```

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

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

Note: `np.var` returns a biased variance estimate and `np.cov` returns an unbiased covariance estimate. Either may be used, but the choice must be applied consistently across all variance and covariance computations.

In [39]:
```python
def sum_of_square_projection_error(P, x, X, K):
    # Compute the sum of squares of the reprojection error
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - measured 2D image points in pixel coordinates
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
    #    cost - Sum of squares of the reprojection error

    """your code here"""
    P_pixel = K @ P
    projected = dehomogenize(P_pixel @ homogenize(X))

    errors = np.sum((x - projected)**2, axis=0)

    cost = np.sum(errors)
    return cost

def estimate_camera_pose_linear(x, X, K):
    # Estimate the normalized camera projection matrix
    # Inputs:
    #    x - 2D inlier points
    #    X - 3D inlier points
    # Output:
    #    P - normalized camera projection matrix

    """your code here"""


    R = np.eye(3)
    t = np.array([[1, 0, 0]]).T
    P = np.concatenate((R, t), axis=1)

    #1. calculate control points in world coord frame
    mu_X_t = np.mean(X, axis=1)
    covar_X_t = np.cov(X) #default bias is n - 1
    V, _, _ = np.linalg.svd(covar_X_t)
    rank = np.linalg.matrix_rank(covar_X_t)

    #control points C are inhomogeneous
    if rank == 2:
        C1 = mu_X_t
        C2 = mu_X_t + V[:,0]
        C3 = mu_X_t + V[:,1]
    elif rank == 3:
        C1 = mu_X_t
        C2 = mu_X_t + V[:,0]
        C3 = mu_X_t + V[:,1]
        C4 = mu_X_t + V[:,2]

    #2. parametrize world coord control points
    if rank == 2:
        A = np.array([C2 - C1, C3 - C1])
        b = X - C1.reshape(3,1)
        alphas = A.T @ b
        alpha2 = alphas[0]
        alpha3 = alphas[1]
        alpha1 = 1 - alpha2 - alpha3

        #3. calculate control points in camera coord frame
        normalized = normalize(K, x)
        x_h = normalized[0,:]
        y_h = normalized[1,:]

        _, n = X.shape
        M = np.zeros((2*n, 9))

        for i in range(n):
            a = np.array([alpha1[i], alpha2[i], alpha3[i]])

            M[2*i, 0::3] = a
            M[2*i, 2::3] = -a * x_h[i]

            M[2*i+1, 1::3] = a
            M[2*i+1, 2::3] = -a * y_h[i]

    elif rank == 3:
        A = np.array([C2 - C1, C3 - C1, C4 - C1])
        b = X - C1.reshape(3,1)
        alphas = A.T @ b
        alpha2 = alphas[0]
        alpha3 = alphas[1]
        alpha4 = alphas[2]
        alpha1 = 1 - alpha2 - alpha3 - alpha4

        normalized = normalize(K, x)
        x_h = normalized[0,:]
        y_h = normalized[1,:]

        _, n = X.shape
        M = np.zeros((2*n, 12))

        for i in range(n):
            a = np.array([alpha1[i], alpha2[i], alpha3[i], alpha4[i]])

            M[2*i, 0::3] = a
            M[2*i, 2::3] = -a * x_h[i]

            M[2*i+1, 1::3] = a
            M[2*i+1, 2::3] = -a * y_h[i]

    V, _, V_T = np.linalg.svd(M)
    c_control_cam = V_T[-1,:]
    if rank == 2:
        c_control_cam = c_control_cam.reshape(3,3).T
        X_cam = alpha1 * c_control_cam[:, 0].reshape(3,1) + alpha2 * c_control_cam[:, 1].reshape(3,1) \
        + alpha3 * c_control_cam[:, 2].reshape(3,1)
    elif rank == 3:
        c_control_cam = c_control_cam.reshape(4,3).T
        X_cam = alpha1 * c_control_cam[:, 0].reshape(3,1) + alpha2 * c_control_cam[:, 1].reshape(3,1) \
            + alpha3 * c_control_cam[:, 2].reshape(3,1) + alpha4 * c_control_cam[:, 3].reshape(3,1)

    #4. scale camera coord control points
    var_world = np.trace(covar_X_t)

    mu_Z_cam = np.mean(X_cam[2,:])
    var_cam = np.var(c_control_cam[0,:]) + np.var(c_control_cam[1,:]) + np.var(c_control_cam[2,:])

    if mu_Z_cam >= 0:
        beta = np.sqrt(var_world / var_cam)
    else:
        beta = -np.sqrt(var_world / var_cam)

    X_cam = beta * X_cam

    #5. estimate 3D euclidean transformation (R and t)
    mu_X = np.mean(X, axis=1).reshape(3,1)
    mu_cam = np.mean(X_cam, axis=1).reshape(3,1)

    S = (X_cam - mu_cam) @ (X - mu_X).T
    U, _, V_T = np.linalg.svd(S)

    if np.linalg.det(U) * np.linalg.det(V_T.T) < 0:
        R = U @ np.diag([1,1,-1]) @ V_T
    else:
        R = U @ V_T

    t = mu_cam - R @ mu_X

    P[:,0:3] = R
    P[:,-1] = t.reshape(3,)

    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 0.0020232200622558594 secs
R_linear =
[[ 0.43788558 -0.67338753  0.59565549]
 [-0.21422806 -0.7216222  -0.65830672]
 [ 0.87313376  0.1606569  -0.46024646]]
t_linear =
[ 10.13023905 -52.97015861 198.12471392]
```

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

Use $\mathbf{R}_{\text{linear}}$ and $\mathbf{t}_{\text{linear}}$ 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 $\hat{\mathrm{P}} = [\mathbf{R} \,|\, \mathbf{t}]$. You must parameterize the camera rotation using the angle-axis representation $\boldsymbol{\omega}$ (where $[\boldsymbol{\omega}]_\times = \ln \mathbf{R}$) 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 $\boldsymbol{\omega}_{\text{LM}}$ and $\mathbf{R}_{\text{LM}}$, and the camera translation $\mathbf{t}_{\text{LM}}$.

In [40]:
```
from scipy.linalg import block_diag

# Note that np.sinc is different than defined in class
def sinc(x):
    """your code here"""
    if x == 0:
        y = 1
    else:
        y = np.sin(x) / x

    return y

def skew(w):
    # Returns the skew-symmetrix represenation of a vector
    """your code here"""
    w_skew = [[0, -w[2], w[1]],
              [w[2], 0, -w[0]],
```

```python
              [-w[1], w[0], 0]]

    w_skew = np.array(w_skew)

    return w_skew


def parameterize_rotation_matrix(R):
    # Parameterizes rotation matrix into its axis-angle representation
    """your code here"""
    _, Sigma, V_T = np.linalg.svd(R - np.identity(3))
    v = V_T[-1,:].reshape(3,1)
    norm_v = np.linalg.norm(v)
    v /= norm_v

    v_h = np.array([R[2,1] - R[1,2], R[0,2] - R[2,0], R[1,0] - R[0,1]])
    cos = (np.linalg.trace(R) - 1) / 2
    sin = (v.T @ v_h) / 2
    theta = np.atan2(sin, cos)

    _, null = v.shape
    if null > 1:
        w = 0.5 * v_h
    else:
        w = theta * v

    norm_w = np.linalg.norm(w)

    if norm_w > np.pi:
        w = (1 - 2*np.pi/norm_w * np.ceil((norm_w - np.pi) / (2 * np.pi))) * w

    return w, theta


def deparameterize_rotation_matrix(w):
    # Deparameterizes to get rotation matrix
    """your code here"""
    w_skew = skew(w.flatten())
    theta = np.linalg.norm(w)
    R = np.cos(theta) * np.identity(3) + sinc(theta) * w_skew + ((1 - np.cos(theta)) / theta**2) * (w @ w.T)

    if np.isnan((1 - np.cos(theta)) / theta**2):
        R = np.identity(3) + w_skew

    return R


def data_normalize(pts):
    # Normalize data points to have zero mean and uniform scale
    # Input:
    #    pts - 3D scene points
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix

    """your code here"""
    n = pts.shape[0]

    s = np.sqrt(n / np.sum(np.var(pts, axis=1)))
    mu = np.mean(pts, axis=1)

    T = np.eye(pts.shape[0]+1)
    T[:n,:n] *= s
    T[:n,n] = -s * mu

    pts = homogenize(pts)
    pts = T @ pts

    return pts, T


def normalize_with_cov(K, x, covarx):
    # Normalize 2D points and covariance matrix
    # Inputs:
    #    K - camera calibration matrix
    #    x - 2D points in pixel coordinates
    #    covarx - covariance matrix
    #
    # Outputs:
    #    pts - 2D points in normalized coordinates
    #    covarx - normalized covariance matrix

    """your code here"""
    K_inv = np.linalg.inv(K)
    _, n = x.shape

    A = K_inv[0:2, 0:2]
    A_big = np.kron(np.eye(n), A)

    pts = normalize(K, x)
    covarx = A_big @ covarx @ A_big.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

    dx_hat_dw = np.zeros((2, 3))

    """your code here"""
    theta = np.linalg.norm(w)
    X_t = X.flatten()
    w = w.flatten()

    # if theta == 0 or theta < 1e-8:
    #     X_rot = X_t + np.cross(w, X_t)
    # else:
    #     X_rot = X_t + sinc(theta) * np.cross(w, X_t) + ((1 - np.cos(theta)) / theta**2) * np.cross(w, np.cross(w, X_t))

    X_rot = R @ X.reshape(3,1)

    dsinc_dtheta = 0 if theta == 0 else (np.cos(theta) / theta) - (np.sin(theta) / theta**2)
    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).reshape(1,3)

    if np.abs(theta) < 1e-8:
        dX_rot_dw = skew(-X_t)
    else:
        skew_x_neg = skew(-X_t)
        skew_w = skew(w)
        skew_wx = skew(-np.cross(w, X_t))

        dX_rot_dw = sinc(theta) * skew_x_neg + np.cross(w, X_t).reshape(3,1) * dsinc_dtheta * dtheta_dw \
            + np.cross(w, np.cross(w, X_t)).reshape(3,1) * ds_dtheta * dtheta_dw + s * (skew_w @ skew_x_neg + skew_wx)

    dx_hat_dw = np.zeros((2, 3))

    x_h = dehomogenize(X_rot + t).reshape(2,)
    Z_rot = X_rot[2]
    w_h = (Z_rot + t[2])[0]

    dx_hat_dX_rot = np.zeros((2,3))
    dx_hat_dX_rot[:,:] = [[1/w_h, 0, -x_h[0]/w_h],
                          [0, 1/w_h, -x_h[1]/w_h]]

    dx_hat_dw = dx_hat_dX_rot @ dX_rot_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

    dx_hat_dt = np.zeros((2, 3))

    """your code here"""
    w_h = (R[2,:].T @ X + t[2])[0]
    x_norm = x_norm.reshape(2,)

    dx_hat_dt[:,:] = [[1/w_h, 0, -x_norm[0]/w_h],
                      [0, 1/w_h, -x_norm[1]/w_h]]


    return dx_hat_dt


def compute_cost(P, x, X, covarx):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - measured 2D image points in normalized coordinates
    #    X - 3D groundtruth scene points
    #    covarx - covariance matrix
    #
    # Output:
    #    cost - total projection error

    """your code here"""
    X = homogenize(X)
    x_h = dehomogenize(P @ X)

    epsilon = (x - x_h).reshape(-1,1)
    sigma_i = np.linalg.inv(covarx)

    cost = epsilon.T @ sigma_i @ epsilon
    cost = cost[0,0]
    return cost
```

In [41]:
```
# 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')
    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 [42]:
```
def estimate_camera_pose_nonlinear(P, x, X, K, max_iters, lam):
    # Estimate camera pose using Levenberg-Marquardt algorithm
    # 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)
    """your code here"""
    tau_1 = 1e-7
    tau_2 = 0

    x, covarx = normalize_with_cov(K, x, covarx)
    X, U = data_normalize(X)

    R = P[0:3,0:3]
    t = P[:,-1].reshape(3,1)
    C = (-R.T @ t).reshape(3,1)

    C_DN = U @ homogenize(C)
    t_DN = -R @ dehomogenize(C_DN)

    P = np.hstack((R, t_DN))

    x_t = dehomogenize(x)
    X_t = dehomogenize(X)
    #x and X are homogeneous, x_t and X_t are inhomogeneous normalized

    for i in range(max_iters):
        #compute previous: R, P, and cost
        R = P[0:3,0:3]
        t = P[:,-1].reshape(3, 1)
        w, _ = parameterize_rotation_matrix(R)
        cost_prev = compute_cost(P, x_t, X_t, covarx)

        #compute Jacobian
        _, n = X_t.shape
        J = np.zeros((2 * n, 6))
        x_proj = dehomogenize(P @ X)

        for k in range(n):
            X_k = X_t[:, k]
            dx_dw = partial_x_hat_partial_w(R, w, t, X_k)
            dx_dt = partial_x_hat_partial_t(R, t, x_proj[:, k], X_k)
            J[2*k:2*k+2, :] = np.hstack((dx_dw, dx_dt))

        #compute normal equations matrix
        epsilon = (x_t - x_proj).flatten('F').reshape(-1,1)
        sigma_inv = np.linalg.inv(covarx)
        U_n = J.T @ sigma_inv @ J
        eps_a = J.T @ sigma_inv @ epsilon

        while True:
            #compute candidate/current: w_0, t_0, R_0, P_0, cost_0
            S = U_n + lam * np.eye(6)
            delta = np.linalg.solve(S, eps_a)
            w_0 = w + delta[:3].reshape(3, 1)
            t_0 = t + delta[3:].reshape(3, 1)
            R_0 = deparameterize_rotation_matrix(w_0)
            P_0 = np.hstack((R_0, t_0))

            cost_0 = compute_cost(P_0, x_t, X_t, covarx)

            #compare candidate to previous
            if cost_0 >= cost_prev:
                lam *= 10
            else:
                w = w_0
                t = t_0
                R = R_0
                P = P_0
                lam /= 10
                break

        #termination criteria
        if (1 - (cost_0 / cost_prev)) <= tau_1:
            break
        if (cost_prev - cost_0) <= tau_2:
            break
        if (i > 100 * 6 + 1):
            break

        cost = compute_cost(P, x_t, X_t, covarx)
        print ('iter %03d Cost %.9f'%(i+1, cost))

    # data denormalization
    C_DN = -R.T @ t
    C = dehomogenize(np.linalg.inv(U) @ homogenize(C_DN))
    t_final = -R @ C

    P = np.hstack((R, t_final))
    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)
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])
```

```
iter 001 Cost 2575326.721369626
iter 002 Cost 2337085.610741599
iter 003 Cost 2307369.842144787
iter 004 Cost 2306696.802930197
iter 005 Cost 2306626.600245328
iter 006 Cost 2306617.811587300
iter 007 Cost 2306616.558543483
took 0.127549 secs
w_LM =
[[0.96347668]
 [0.30417528]
 [1.31321178]]
R_LM =
[[ 0.28121031 -0.6737012   0.6834087 ]
 [ 0.90556375 -0.04940238 -0.42132374]
 [ 0.31760833  0.73735073  0.59618693]]
t_LM =
[  5.77945902  42.98023107 177.26715842]
```