HW3_Final

February 16, 2022

1 CSE 252B: Computer Vision II, Winter 2022 – Assignment 3

1.0.1 Instructor: Ben Ochoa

1.0.2 Due: Wednesday, February 16, 2022, 11:59 PM

1.1 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 problems 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 effecient grading).
- You may use python packages for basic linear algebra, but you may not use packages that directly solve the problem. If you are uncertain about using a specific package, then please ask the instructional staff whether or not it is allowable.
- 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.

1.2 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., $\boldsymbol{x}_i = \boldsymbol{P}\boldsymbol{X}_i$), 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

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

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

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 $\boldsymbol{R}$ and translation $\boldsymbol{t}$ 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 image coordinates. Calculate the projection error, which is the (squared) distance between projected points (the points in 3D projected under the normalized camera projection matrix $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$) and the measured points in normalized coordinates (hint: the error tolerance is simpler to calculate in image coordinates using $\boldsymbol{P} = \boldsymbol{K}[\boldsymbol{R}|\boldsymbol{t}]$ than in normalized coordinates using $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$. You can avoid doing covariance propagation).

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

[1]:
```
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

    pts = np.linalg.inv(K) @ Homogenize(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)

def ComputeCost(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 - total projection error


    # Project world points onto image
    x_proj = Dehomogenize(K @ P @ Homogenize(X))

    # Compute cost
    cost = np.sum(np.square(np.linalg.norm(x-x_proj, axis=0)))

    return cost

def computeMSACConsensusCost(P, x, X, K, tol):
    # Inputs:
    #    P - normalized camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
```

```
    #    cost - total projection error

    # Project world points onto image
    X_h = Homogenize(X)
    #print(K.shape, P.shape, X_h.shape)
    x_proj = Dehomogenize(K @ P @ X_h)

    # Compute cost
    cost = np.linalg.norm(x-x_proj,axis = 0)
    inliers = np.where(cost <= tol)[0]
    cost[cost > tol] = tol
    cost = np.sum(cost)
    return cost, inliers
```

```
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]]
```

[2]:
```
# Helper methods
def calculateNormalProjMatrix(R,t):
    '''
    Given camera intrinsic and extrinisc
    compute the normalized camera projection
    matrix
    '''
    P = np.hstack((R,t))
    # Normalize
    P = P / np.linalg.norm(P)
    return P

def calculateModelFinsterwalder(x, X, K):
    '''
    Compute the camera pose using Finsterwalder'
    3 point algorithm
    '''

    # Normalize x
    x_hat = Normalize(K,x)

    # Unitized direction vectors
    d = x_hat/np.linalg.norm(x_hat,axis=0)
    assert d.shape == x_hat.shape, "Unitization incorrect"

    # Calculate a,b,c
    a = np.linalg.norm(X[:,1]-X[:,2])
    b = np.linalg.norm(X[:,0]-X[:,2])
    c = np.linalg.norm(X[:,1]-X[:,0])

    # Calculate cosines and sines of
    # angles alpha, beta, gamma
    cos_alpha = np.matmul(d[:,1].T,d[:,2])
    cos_beta = np.matmul(d[:,0].T,d[:,2])
    cos_gamma = np.matmul(d[:,0].T,d[:,1])

    sin_alpha = np.sqrt(1 - np.square(cos_alpha))
    sin_beta = np.sqrt(1 - np.square(cos_beta))
    sin_gamma = np.sqrt(1 - np.square(cos_gamma))

    # Compute coeffecients of cubic
    G = np.square(c)*(np.square(c*sin_beta) - np.square(b*sin_gamma))

    H = np.square(b*sin_gamma)*(b*b-a*a) + np.square(c*sin_beta)*(c*c + 2*a*a) +␣
↪\
        2*np.square(b*c)*(cos_alpha*cos_beta*cos_gamma-1)

    I = np.square(b*sin_alpha)*(b*b-c*c) + np.square(a*sin_beta)*(a*a + 2*c*c) +␣
↪\
        2*np.square(a*b)*(cos_alpha*cos_beta*cos_gamma-1)

    J = a*a*(np.square(a*sin_beta) - np.square(b*sin_alpha))

    if np.any(np.isnan([G,H,I,J])):
        # degenerate configuration
        return None

    # Find real root
    roots = np.roots([G,H,I,J])

    if (len(roots) != 3):
        # degenerate configuration
        return None

    if np.isreal(roots[1]+roots[2]):
        idx = 0
    elif np.isreal(roots[2]+roots[0]):
        idx = 1
    else:
        idx = 2 # 1

    lambda_0 = np.real(roots[idx])
```

```
# Compute coeffecients of quadratic
A = b*b*(lambda_0 + 1)
B = -b*b*(cos_alpha)
C = -a*a + b*b - lambda_0*np.square(c) ### NOT SURE ###
D = -b*b*cos_gamma*lambda_0
E = cos_beta*(a*a+ lambda_0*c*c)
F = lambda_0*(b*b-c*c) - a*a

if B*B - A*C < 0 or E*E-C*F < 0:
     return None

p = np.sqrt(B*B - A*C)
q = np.sign(B*E-C*D)*np.sqrt(E*E-C*F)

# Compute m,n
m1 = (-B + p) / C
n1 = -(E-q)/ C
m2 = (-B-p) / C
n2 = -(E+q)/ C

# Solve quadratic
def getCoeffsQuad(b,c,cos_beta,cos_gamma,m,n):
    '''
    Helper function to get coeffs of quadratic
    '''
    coeffs = []
    coeffs.append(b*b-m*m*c*c)
    coeffs.append(2*c*c*(cos_beta-n)*m - 2*b*b*cos_gamma)
    coeffs.append(-np.square(c*n) + 2*n*cos_beta*c*c + b*b - c*c)
    return coeffs

U = []
V = []

[u1,u2] = np.roots(getCoeffsQuad(b,c,cos_beta,cos_gamma,m1,n1))

v1 = u1*m1 + n1
v2 = u2*m1 + n1

if(np.abs(np.imag(u1)) < 1e-2):
    V.append(np.real(v1))
    U.append(np.real(u1))
    V.append(np.real(v2))
    U.append(np.real(u2))

[u3,u4] = np.roots(getCoeffsQuad(b,c,cos_beta,cos_gamma,m2,n2))
```

```
    v3 = u3*m2 + n2
    v4 = u4*m2 + n2

    if(np.abs(np.imag(u3)) < 1e-2):
        V.append(np.real(v3))
        U.append(np.real(u3))
        V.append(np.real(v4))
        U.append(np.real(u4))

    def calcDir(b, cos_beta, v, u):
        '''
        Helper function to
        calculate candidate directions
        '''
        s1 = np.sqrt(b*b/(1 + v*v - 2*v*cos_beta))
        s2 = u*s1
        s3 = v*s1
        return s1,s2,s3

    if len(V) == 0:
        # degenerate configuration
        return None

    # Calculate s1, s2, s3
    flag = False
    for i in range(len(V)):
        s1,s2,s3 = calcDir(b,cos_beta, V[i],U[i])
        if s1 > 0 and s2 > 0 and s3 > 0:
            flag = True
            break

    if not flag:
        # Degenerate configuration
        return None

    X_cam_1 = s1*d[:,0].reshape(-1,1)
    X_cam_2 = s2*d[:,1].reshape(-1,1)
    X_cam_3 = s3*d[:,2].reshape(-1,1)

    X_cam = np.hstack((X_cam_1,X_cam_2,X_cam_3))
    return X_cam
```

[3]:
```
def computePose(X_cam, X_world):
    '''
    Computes the pose given set of world and
    image coordinates
    '''
    assert X_cam.shape[0] == 3, 'shape issue'
    assert X_cam.shape == X_world.shape, 'shape issue'

    # Compute R, t
    B = X_world.T    # nx3
    C = X_cam.T # nx3

    # mean derivative form
    mu = np.mean(B,0)
    B = B - mu
    mu_dash = np.mean(C,0)
    C = C - mu_dash

    # Scatter matrix
    S = C.T @ B
    U,_,Vh = np.linalg.svd(S)

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

    t = mu_dash - R @ mu
    return R,t.reshape(-1,1)
```

[4]:
```
from scipy.stats import chi2

def MSAC(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


    consensus_min_cost = np.inf
    trials = 0
    #inliers = np.random.randint(0, x.shape[0], size=10) # Indexes of inliers
    s = 3 # ### NOT SURE ###
    max_trials = np.inf

    while trials < max_trials:
        trials += 1
        if(consensus_min_cost <= thresh):
            break

        # Select random sample
        inliers = np.random.randint(0, x.shape[1], size = 3)

        # Calculate Model using Finsterwalder
        X_cam = calculateModelFinsterwalder(x[:,inliers], X[:,inliers], K)

        if X_cam is None:
            # Handle degenerate configuration
            continue
        R, t = computePose(X_cam, X[:,inliers])

        # Compute normalized camera projection matrix
        P = calculateNormalProjMatrix(R,t)

        # Compute Consensus cost
        consensus_cost,inliers = computeMSACConsensusCost(P,x,X,K,tol)

        if consensus_cost < consensus_min_cost:
            # Update Model
            consensus_min_cost = consensus_cost
            consensus_min_cost_model = P

            # Update max_trials
            w = inliers.shape[0]/x.shape[1]
            max_trials = int(np.log(1-p)/np.log(1-np.power(w,s)))

    # Get inliers
    consensus_min_cost,inliers = computeMSACConsensusCost(P,x,X,K,tol)

    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 0
p = 0.99
alpha = 0.95
tol = chi2.ppf(alpha,df = 2)
print(tol)
```

```
tic=time.time()

cost_MSAC, P_MSAC, inliers, trials = MSAC(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('took %f secs'%time_total)
print('%d iterations'%trials)
print('inlier count: ',len(inliers))
print('MSAC Cost=%.9f'%cost_MSAC)
print('P = ')
print(P_MSAC)
print('inliers: ',inliers)
```

```
5.991464547107979
took 0.011255 secs
20 iterations
inlier count:  39
MSAC Cost=200.525989070
P =
[[ 0.00147526 -0.00403414  0.00375023  0.01973939]
 [ 0.00374398 -0.00211316 -0.00374594  0.04008412]
 [ 0.00403994  0.0034315   0.00210205  0.99895249]]
inliers:  [ 0  2  3  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 21 23 24 25 26
27
 28 29 30 31 32 37 38 39 40 41 42 43 44 47 49]
```

Final values for parameters

- $p = 0.99$
- $\alpha = 0.95$
- tolerance = 5.9914
- num_inliers = 39
- num_attempts = 20

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

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

```
[5]: def EPnP(x, X, K):
         # Inputs:
         #    x - 2D inlier points
         #    X - 3D inlier points
         # Output:
         #    P - normalized camera projection matrix

         assert X.shape[0] == 3 , 'Not a fat matrix!'
         mean = np.mean(X,axis = 1).reshape(-1,1)
         cov = np.cov(X)

         # Calculate v
         U,S,Vh = np.linalg.svd(cov)
         cov_w = np.sum(S)

         b = X - mean
         alpha = Vh @ b
         alpha_1 = 1 - np.sum(alpha,0)
         alpha_1 = alpha_1.reshape(1,-1)
         alpha = np.vstack((alpha_1, alpha))

         # Calculate M
         x = Dehomogenize(np.linalg.inv(K) @ Homogenize(x))
         M = np.zeros((2*X.shape[1],12))
         for i in range(X.shape[1]):
             alpha_1 = alpha[0,i]
             alpha_2 = alpha[1,i]
             alpha_3 = alpha[2,i]
             alpha_4 = alpha[3,i]
             x_ = x[0,i]
             y = x[1,i]
             M[2*i,:] = np.array([alpha_1, 0 ,-alpha_1*x_, alpha_2, 0, -alpha_2*x_,\
                                 alpha_3, 0, -alpha_3*x_, alpha_4, 0, -alpha_4*x_])
             M[2*i+1,:] = np.array([0 ,alpha_1, -alpha_1*y, 0, alpha_2, -alpha_2*y,\
                                   0, alpha_3, -alpha_3*y, 0, alpha_4, -alpha_4*y])


         assert M.shape[0] == 2*x.shape[1] and M.shape[1] == 12 , 'Shape of M not␣
      ↪consistent'

         # Calculate control points in camera co-ordinate
         _,_,Vh = np.linalg.svd(M, full_matrices=True)
         control_cam = Vh[-1,:].T
         control_cam = np.reshape(control_cam, (3,4), order='F')

         # Parametrize camera co-ordinates
         X_cam = np.zeros((3,x.shape[1]))
```

```
    for i in range(x.shape[1]):
        alpha_1 = alpha[0,i]
        alpha_2 = alpha[1,i]
        alpha_3 = alpha[2,i]
        alpha_4 = alpha[3,i]
        X_cam[:,i] = alpha_1*control_cam[:,0] + alpha_2*control_cam[:,1] + \
        alpha_3*control_cam[:,2] + alpha_4*control_cam[:,3]

    # Scale
    mean = np.mean(X_cam, axis=1).reshape(-1,1)
    cov = np.cov(X_cam)
    _,S,_ = np.linalg.svd(cov)
    cov_cam = np.sum(S)
    beta = np.sqrt(cov_w/cov_cam)
    if mean[0,-1] < 0:
        beta = -beta

    X_cam = X_cam * beta

    # Compute pose
    R,t = computePose(X_cam, X)

    P = np.concatenate((R, t), axis=1)
    return P

#### TODO  - Comment ####
# Uncomment the following block to run EPnP on an example set of inliers.
# You MUST comment it before submitting, or you will lose points.
# The sample cost with these inliers should be around 57.82.
#inliers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19,␣
 ↪21, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42,␣
 ↪43, 44, 45, 46, 47, 48, 49]
#inliers = np.array(inliers)
x = x0[:,inliers]
X = X0[:,inliers]

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

cost = ComputeCost(P_linear, x, X, K)

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

```
took 0.002095 secs
R_linear =
[[ 0.27692967 -0.69262097  0.66602263]
 [ 0.6634624  -0.36355934 -0.65394361]
 [ 0.69507381  0.62297736  0.3588476 ]]
t_linear =
[  5.39185362   7.80491722 175.84635822]
```

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

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

[6]:
```
from scipy.linalg import block_diag

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

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

def skew(w):
    # Returns the skew-symmetrix represenation of a vector

    w = w.reshape(-1,1)
    assert w.shape[0] == 3 and w.shape[1] == 1, 'w shape error' + str(w.shape)

    a = w[0,0]
    b = w[1,0]
    c = w[2,0]

    w_skew = np.array([[0,-c,b],[c,0,-a],[-b,a,0]])
    return w_skew


def Parameterize(R):
    # Parameterizes rotation matrix into its axis-angle representation

    _,_,Vh = np.linalg.svd(R-np.eye(3))
    v = Vh[-1,:].T

    cos_theta = (np.trace(R)-1)/2
    v_hat = np.array([[R[2,1] - R[1,2]],
                      [R[0,2] - R[2,0]],
                      [R[1,0] - R[0,1]]])

    sin_theta = v.T @ v_hat/2
    theta = np.arctan2(sin_theta,cos_theta)

    if theta == 0:
        # Small rotations
        w = v_hat/2
    else:
        w = theta*v

    # Normalize
    if np.linalg.norm(w) >= np.pi:
        w_norm = np.linalg.norm(w)
        w = w * (1-2*np.pi/w_norm)*np.ceil((w_norm-np.pi)/(2*np.pi))

    return w, theta


def Deparameterize(w):
    # Deparameterizes to get rotation matrix

    w = w.reshape(1,-1)

    theta = np.linalg.norm(w)
    if theta == 0:
        R = np.eye(3) + skew(w)
    else:
        R = np.cos(theta)*np.eye(3) \
            + Sinc(theta)*skew(w) \
            + ((1-np.cos(theta))/theta**2)*(w.T @ w)
    return R


def DataNormalize(pts):
    # Input:
    #    pts - 3D scene points
    # Outputs:
    #    pts - data normalized points
    #    T - corresponding transformation matrix

    ## DOUBT - Camera center???

    # Compute mean and variance of each dimension
    m = np.mean(pts,1).reshape(-1,1)
    v = np.var(pts,1).reshape(-1,1)
    s = np.sqrt(m.shape[0]/np.sum(v))

    # Create Transformation matrix
    T = np.eye(pts.shape[0]+1)
    for i in range(T.shape[0]-1):
        T[i,i] = s
        T[i,-1] = -m[i]*s

    # Normalize each set of points
    pts_h = Homogenize(pts)
    for i in range(pts.shape[1]):
        pts_h[:,i] = np.matmul(T, pts_h[:,i])
    return pts_h, T


def Normalize_withCov(K, x, covarx):
    # 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

    # x_norm = A * x_pix + b
    K_inv = np.linalg.inv(K)
    A = K_inv[0:2,0:2]
    B = K_inv[0:2,-1]

    pts = Dehomogenize(K_inv @ Homogenize(x))
    J = np.zeros((2*x.shape[1],2*x.shape[1]))

    for i in range(x.shape[1]):
        J[2*i:2*i+2,2*i:2*i+2] = A

    covarx = J @ covarx @ J.T

    return pts, covarx


def Jacobian(R, w, t, X):
    # compute the jacobian matrix
    # Inputs:
    #    R - 3x3 rotation matrix
    #    w - 3x1 axis-angle parameterization of R
    #    t - 3x1 translation vector
    #    X - 3D inlier points
    #
    # Output:
    #    J - Jacobian matrix of size 2*nx6

    assert w.shape[0] == 3 , 'w shape issue in Jacobian'
    assert t.shape[0] == 3 , 't shape issue in Jacobian'

    J = np.zeros((2*X.shape[1],6))
    x_hat = Dehomogenize(R@X + t)
    theta = np.linalg.norm(w)
    s = (1-np.cos(theta))/theta**2
    ds_dtheta = (theta*np.sin(theta) - 2*(1-np.cos(theta)))/np.power(theta,3)
    r_3 = R[2,:]
    t_3 = t[2,0]

    for i in range(X.shape[1]):
        w_hat = r_3.T @ X[:,i] + t_3
        x_hat_x = x_hat[0,i]
        x_hat_y = x_hat[1,i]

        # Eq 24
        dx_hat_dt = np.array([[1, 0, -x_hat_x],
                              [0 ,1, -x_hat_y]])/w_hat

        # Eq 23
        dx_hat_dX_rotated = np.array([[1, 0, -x_hat_x],
                                      [0 ,1, -x_hat_y]])/w_hat

        # Eq 17
        v = X[:,i].reshape(-1,1)
        if theta == 0:
            d_X_rotated_dw = skew(-v)
        else:
            d_X_rotated_dw = Sinc(theta)*skew(-v) \
            + np.cross(w,v,axis = 0) @ w.T/np.linalg.norm(w)*Dsinc(theta) \
            + np.cross(w, np.cross(w,v,axis = 0), axis = 0) @ w.T/np.linalg.
↪norm(w)*ds_dtheta \
            + s*(skew(w) @ skew(-v) + skew(-np.cross(w,v,axis = 0)))

        # Eq 22
        dx_hat_dw = dx_hat_dX_rotated @ d_X_rotated_dw
        J[2*i:2*i+2,:] = np.hstack((dx_hat_dw, dx_hat_dt))

    return J


def ComputeCost_withCov(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_hat =  Dehomogenize(P @ Homogenize(X))
    m = x.ravel('F')
    m_hat = x_hat.ravel('F')
    cost = np.matmul(np.matmul((m-m_hat).T,np.linalg.inv(covarx)),(m-m_hat))
    return cost
```

[7]:
```
def LM(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

    # Data Normalize
    covarx = np.eye(2*X.shape[1])
    x, covarx = Normalize_withCov(K, x, covarx)
    X,U = DataNormalize(X)

    # Data normalize camera center
    R = P[:,0:3]
    t = P[:,-1].reshape(-1,1)
    cam_center = -R.T @ t
    cam_center = U @ Homogenize(cam_center.reshape(-1,1))
    t = -R @ Dehomogenize(cam_center)

    P = np.concatenate((R, t), axis=1)
    cost_prev = ComputeCost_withCov(P, x, Dehomogenize(X), covarx)
    print('Initial Cost %.9f' %(cost_prev))
    i = 0

    while (i < max_iters):

        # Parameterize R
        w, theta = Parameterize(R)

        # Jacobian
        J = Jacobian(R, w.reshape(-1,1), t, Dehomogenize(X))

        # Augmented normal equation
        J_T_Sigma = np.matmul(J.T, np.linalg.inv(covarx))

        epsilon = x - Dehomogenize(P @ X)
        epsilon = epsilon.ravel('F').reshape(1,-1)

        # L*Delta = Q (Normal Equation)
        Q = np.matmul(J_T_Sigma, epsilon.T)
        L = np.matmul(J_T_Sigma,J) + lam*np.eye(6)
        delta = np.matmul(np.linalg.inv(L),Q)
        delta = delta.reshape(-1,1)

        # Candidate
        w_c = w + delta[0:3,0]
        t_c = t + delta[3:6,0].reshape(-1,1)


        # Deparametrize
        R_c = Deparameterize(w_c)

        P_c = np.concatenate((R_c, t_c), axis=1)

        cost = ComputeCost_withCov(P_c, x, Dehomogenize(X), covarx)
        if (1 - cost/cost_prev < 1e-12):
            # Terminate if change in cost less than
            # threshold
            break

        if cost < cost_prev:
            # Valid iteration
            lam = lam/10
            R = R_c
            t = t_c
            P = np.concatenate((R, t), axis=1)
            print ('iter %03d Cost %.9f'%(i+1, cost))
            i = i + 1
            cost_prev = cost
        else:
            # Update lambda
            lam = lam*10


    # Data denormalize t
    t = - R @ (Dehomogenize(np.linalg.inv(U) @ Homogenize(-R.T @ t)))

    # Average cost per point
    print("")
    print('Average cost per point %.9f'%(cost/x.shape[1]))
    return P

# With the sample inliers...
# Start: 57.854356308
# End: 57.815478730

# LM hyperparameters
lam = .001
max_iters = 100

tic = time.time()
P_LM = LM(P_linear, x, X, K, max_iters, lam)
w_LM,_ = Parameterize(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])
```

```python
print('t_LM = ')
print(P_LM[:,-1])
```

```
Initial Cost 58.931068871
iter 001 Cost 58.611330437
iter 002 Cost 58.611209431
iter 003 Cost 58.611209384

Average cost per point 1.502851523
took 0.096409 secs
w_LM =
[ 1.33428812 -0.03220256  1.41485346]
R_LM =
[[ 0.27705603 -0.69257837  0.66601438]
 [ 0.66155944 -0.36519231 -0.65496083]
 [ 0.69683503  0.62206895  0.35700303]]
t_LM =
[0.90479599 0.02792594 8.79156472]
```