CSE 252B: Computer Vision II, Winter 2019 – Assignment 3

Instructor: Ben Ochoa

Due: Wednesday, February 20, 2019, 11:59 PM

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains both math and programming problems.
- All solutions must be written in this notebook
- Math problems must be done in Markdown/LATEX. Remember to show work and describe your solution.
- Programming aspects of this assignment must be completed using Python in this notebook.
- Your code should be well written with sufficient comments to understand, but there is no need to write extra markdown to describe your solution if it is not explictly asked for.
- This notebook contains skeleton code, which should not be modified (This is important for standardization to facilate effeciant grading).
- You may use python packages for basic linear algebra, but you may not use packages that directly solve the problem. Ask the instructor if in doubt.
- You must submit this notebook exported as a pdf. You must also submit this notebook as an .ipynb file.
- Your code and results should remain inline in the pdf (Do not move your code to an appendix).
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- It is highly recommended that you begin working on this assignment early.

Problem 1 (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 $\hat{\boldsymbol{P}} = [\boldsymbol{R}|\boldsymbol{t}]$).

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 [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]


# 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):
    # Inputs:
    #    P - camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
    #    cost - total projection error --- >  2d pts inhomo
    #n = x.shape[1]

    #covarx = np.eye(2*n) # covariance propagation

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


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

```
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 [2]:

```python
def Back_Proj(x,K):
    # Inputs:
    #    x - 2D inhomogeneous image points (2*n)
    #    K - camera calibration matrix
    # Output:
    #    d - 3D unit ray in Camera frame (inhomo)(3*n)
    d = np.linalg.inv(K) @ Homogenize(x)  #(3*n)
    d_normalized = d/(np.sign(d[2])*np.linalg.norm(d,axis = 0))
    return d_normalized
```

In [3]:

```python
def P_extrinsic(R,T):
    P_ext = np.eye(4)
    P_ext[:-1,:-1] = R
    P_ext[:-1,-1] =  T.reshape(-1)
    return P_ext
```

In [4]:

```python
def P_proj(R,T, K):
    P_ext = P_extrinsic(R,T)
    Iden = np.hstack((np.eye(3),np.zeros((3,1))))
    P = K @ Iden @ P_ext
    return P
```

In [5]:

```
def Pose_estimation(p,X,K):
    # Inputs:
    #    Least_square_estimation (Umeyama paper)
    #    X - 3*n 3D PTS in world frame (inhomo)  hstack
    #    p - 3*n 3D PTS in cam frame (inhomo)  hstack
    #    K - camera calibration matrix
    # Output:
    #    R,T - Camera pose
    #    P - Camera projection matrix
    #    Cost  - SUM OF SQUARE ERROR OF 2D INHOMO PTS
    #p = np.hstack((p1,p2,p3))  # P Camera frame
    mean_p = np.mean(p,axis = 1).reshape(-1,1)  #(3,1)
    #X = np.hstack((X1,X2,X3))  # X World frame
    mean_X = np.mean(X,axis = 1).reshape(-1,1)
    # Mean_deviation form B,C
    C = (p - mean_p).T    # Cam
    B = (X - mean_X).T   # World
    Sigma = (C.T @ B)/p.shape[1]
    u,d,vh = np.linalg.svd(Sigma)
    eye = np.eye(3)
    if (np.linalg.det(u)*np.linalg.det(vh.T)) < 0:  # Make sure det(R) = +1
        eye[-1,-1] = -1
        R = u @ eye @ vh
    else:
        R = u @ vh  # 3*3
    T = mean_p - R @ mean_X    #3*1
    P = P_proj(R,T,K)
    #Cost = ComputeCost(P, x, X)


    return R,T ,P
```

In [6]:

```
#USE minimum lbda
def P3P(x,X,K):
    # Inputs:
    #    Finsterwalder's Solution
    #    pick three pts from database randomly
    #    x - 2D inhomogeneous image points (3 pts hstack)
    #    X - 3D inhomogeneous scene (World) points (3 pts hstack)
    #    K - camera calibration matrix
    # Output:
    #    p_3pts_Cam - 3d pts in Camera frame (inhomo)  (3*3, p1,p2,p2 hstack )
    #    R,T - corresponding camera pose
    #    P - Camera projection matrix
    R = np.zeros((4,3,3))
    T = np.zeros((4,3,1))
    P = np.zeros((4,3,4))
    p_3pts = np.zeros((4,3,3))
```

```
    a = np.linalg.norm(X[:,1]-X[:,2])

    b = np.linalg.norm(X[:,2]-X[:,0])
    c = np.linalg.norm(X[:,0]-X[:,1])
    j = Back_Proj(x,K)
    cos_a = (j[:,1]*j[:,2]).sum() #alpha
    cos_b = (j[:,0]*j[:,2]).sum()  #beta
    cos_g = (j[:,0]*j[:,1]).sum()  #gamma
    sin_a2 = 1 - cos_a**2
    sin_b2 = 1 - cos_b**2
    sin_g2 = 1 - cos_g**2
    G = c**2*(c**2*sin_b2 - b**2*sin_g2)
    H = b**2*(b**2-a**2)*sin_g2 + c**2*(c**2+2*a**2)*sin_b2 + 2*b**2*c**2*(-1 +
cos_a*cos_b*cos_g)
    I = b**2*(b**2-c**2)*sin_a2 + a**2*(a**2+2*c**2)*sin_b2 + 2*b**2*a**2*(-1 +
cos_a*cos_b*cos_g)
    J = a**2*(a**2*sin_b2-b**2*sin_a2)
    coeff = [G,H,I,J]
    root = np.roots(coeff)
    real_values = root[np.isreal(root)]
    if real_values.size != 0:
        real_flt64 = real_values.real[abs(real_values.imag)<1e-5]   #real_flt64
        lbda = real_flt64.max()
        #print('Lambda:    ',lbda) # 3


        A = 1 + lbda
        B = -cos_a
        C = (b**2-a**2)/b**2 - lbda*c**2/b**2
        D = -lbda*cos_g
        E = (a**2/b**2 + lbda*c**2/b**2)*cos_b
        F = -a**2/b**2 + lbda*(b**2-c**2)/b**2
        if (B**2-A*C > 0) and (E**2-C*F>0):
            p = np.sqrt(B**2-A*C)
            q = np.sign(B*E-C*D)*np.sqrt(E**2-C*F)
            m = np.array([(-B + p)/C,(-B - p)/C])

            n = np.array([(-E + q)/C,(-E - q)/C]) #(2,)
            #v = um +n
            A_u = b**2 - m**2 *c**2
            B_u = c**2*(cos_b-n)*m-b**2*cos_g
            C_u = -c**2*n**2 + 2*c**2*n*cos_b+b**2-c**2

            sqrt_valued = B_u**2-A_u*C_u
            mask = sqrt_valued> 0
            pos_valued = sqrt_valued[mask]
            #print('pos_valued:    ',pos_valued)
            A_u = A_u[mask]
            B_u = B_u[mask]
            C_u = C_u[mask]
            if pos_valued.size != 0:

                m = m[mask]
```

```
                n = n[mask]

                #print('n_mask:    ',n)
                #print('m_mask:    ',m)


                u_large = -np.sign(B_u)/A_u*(np.abs(B_u)+np.sqrt(pos_valued))  #
shape: (2,)  0,1
                #print('u_large:    ',u_large)
                u_small = C_u/(A_u*u_large)    #shape:   (2,)   0,1
                #print('u_small:    ',u_small)
                v_large  = u_large * m +n    #shape:  (2,)  0,1
                #print('v_large:    ',v_large)
                v_small = u_small * m +n     #shape:   (2,)   0,1
                #print('v_small:    ',v_small)
                if pos_valued.shape[0] == 2.0 :
                    uv = np.array([[u_large[0],v_large[0]],[u_large[1],v_large[1
]],[u_small[0],v_small[0]],[u_small[1],v_small[1]]]).T #(2,4)
                elif pos_valued.shape[0] == 1.0 :
                    uv = np.array([[u_large[0],v_large[0]],[u_small[0],v_small[0
]]]).T   #(2,2)
                #print('uv:    ',uv)
                s1 = np.zeros([1,uv.shape[1]])
                s2 = np.zeros([1,uv.shape[1]])
                s3 = np.zeros([1,uv.shape[1]])
                R = np.zeros((uv.shape[1],3,3))
                T = np.zeros((uv.shape[1],3,1))
                P = np.zeros((uv.shape[1],3,4))
                p_3pts = np.zeros((uv.shape[1],3,3))
                for jj in range (uv.shape[1]):
                    s1[0,jj] = np.sqrt(a**2/(uv[0,jj]**2+uv[1,jj]**2-2*uv[0,jj]*
uv[1,jj]*cos_a))
                    s2[0,jj] = uv[0,jj]*s1[0,jj]
                    s3[0,jj] = uv[1,jj]*s1[0,jj]

                p1 = np.kron(s1, j[:,0].reshape(-1,1)) #3*4 or 3*2  4or2 pairs o
f (p1,p2,p3) p1 ---(X,Y,Z)
                p2 = np.kron(s2, j[:,1].reshape(-1,1))
                p3 = np.kron(s3, j[:,2].reshape(-1,1))
                p = np.vstack((p1,p2,p3))

                ## Following is to calculate R,T,P(projection matrix) for each l
ambda

                for k in range (p_3pts.shape[0]):
                    p_3pts[k] = p[:,k].reshape(3,3,order = 'F') # kth sets of p1
,p2,p3
                    R[k],T[k],P[k] = Pose_estimation(p_3pts[k],X,K)

    return R,T,p_3pts,P
```

In [7]:

```
def Cal_Consensus_Cost(P,x,X,K,tol) :
    # Inputs:
    #    Consensus_Cost for MSAC ()
    #    pick three pts from database randomly
    #    x - 2D inhomogeneous image points (ALL datapts)
    #    X - 3D inhomogeneous scene (World) points (ALL datapts)
    #    K - camera calibration matrix
    #    df - degree of freedom
    #    t - threshold
    # Output:
    #    N_inlier - number of inlier
    #    cost - Consensus cost
    #   inlier - list of indices of the inliers corresponding to input data

    inlier = []
    x_est = Dehomogenize(P @ Homogenize(X)) # same size as data pts
    #print('x_est   :%s'%x_est)
    diffx = x - x_est
    error = np.linalg.norm(diffx,axis = 0)**2
    # error (2dpts difference-norm) < =  torlerance  --> inlier
    mask = error <=  tol
    N_inlier = error[mask].shape[0]
    #print('N_inlier:  %f'%N_inlier)
    if N_inlier == 0.0:
        inlier.append(-1)
        cost = np.inf
    else:
        for ii in error[mask]:
            inlier.append(np.where(error == ii)[0][0])
        cost = error[mask].sum() + (x.shape[1]-N_inlier)*tol
    return N_inlier, cost ,inlier
```

In [15]:

```
from scipy.stats import chi2
from math import log
def MSAC(x, X, K, thresh, p,alpha):
    # Inputs:
    #    x - 2D inhomogeneous image points  total data pts
    #    X - 3D inhomogeneous scene points  total data pts
    #    K - camera calibration matrix
    #    thresh - cost threshold
    #    tol - reprojection error tolerance
    #    p - probability that as least one of the random samples does not contai
n any outliers
    #    s - random sample size
    #    w  - probability that a data is an inlier  (keep updating in while loop
)
    #    alpha - probability that a data is an inlier  (constant in the loop, us
ed to calculate distance threshold/ tolerance)
    #    sigma2 - variance  of the measurement error
```

```
    #    sigma2 - variance of the measurement error
    #    df - degree of freedom
    # 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
    w = 0.95   # prob inlier
    sigma2 = 1
    s = 3 #sample size
    df = 2
    trials = 0
    max_trials = 1  #np.inf
    consensus_min_cost = np.inf
    consensus_min_cost_model = np.zeros((3,4))  # projection matrix
    #cost = 0
    #inliers = [np.inf]
    """your code here"""
    # Distance Threshold (variance sigma2)       #df degree of freedom
    tol = chi2.ppf(alpha,df)    #df =2 alpha = 0.95   tol == 6
    inliers = range(x.shape[1])
    while (trials < max_trials) and (consensus_min_cost > thresh):

        # 3* 3d pts in world --->  p3p  --- >  P --  est 2d inhom pts in image f
rame ---- >  inlier outlier (cost)
        # consensus_min_cost_model --- camera projection matrix P

        idx = np.random.choice(x.shape[1], size = 3, replace = False)   #[4,53,5
1]

        x_3pts = x[:,idx]     # x_3pts,X_3pts 3pts randomly chose from x,X
        X_3pts = X[:,idx]
        R,T,Pts_Cam,P= P3P(x_3pts,X_3pts,K)
        if P.all() != 0 :
            cost = np.zeros((P.shape[0],1))
            N_inlier = np.zeros((P.shape[0],1))

            Local_inlier = [[] for i in range(P.shape[0])]
            for ii in range (P.shape[0]):
                N_inlier[ii], cost[ii],inlier_list = Cal_Consensus_Cost(P[ii],x,
X,K,tol)
                Local_inlier[ii].append(inlier_list)

            if all(ii  != [[-1]] for ii in Local_inlier):

                N_min = np.where(cost == cost.min()) # should be a value from 0
- 3
                #print('N_min:     ',N_min)
                if N_min[0].shape != 0:
                    Local_min_cost = cost[N_min[0][0]]
                    Local_P = P[N_min[0][0]]
                    Local_N_inlier = N_inlier[N_min[0][0]]
                    list_inlier   =   Local_inlier[N_min[0][0]]
                else:
```

```
                    Local_min_cost = cost[N_min]

                    Local_P = P[N_min]
                    Local_N_inlier = N_inlier[N_min]
                    Local_inlier[N_min]
                    list_inlier    =    Local_inlier[N_min[0]]
                #print('Local_min_cost:    ',Local_min_cost,'\n')

                if Local_min_cost < consensus_min_cost:
                    consensus_min_cost = Local_min_cost
                    consensus_min_cost_model = Local_P
                    inliers = list_inlier

                    w = Local_N_inlier / x.shape[1]

                    max_trials = log(1-p)/log(1-w**s)
        trials += 1
    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 0
p = 0.99  # probability that as least one of the random samples does not contain
any outliers
alpha = 0.95   #

tic=time.time()

cost_MSAC, P_MSAC, inliers, trials= MSAC(x0, X0, K, thresh, p,alpha)

# choose just the inliers
x = x0[:,inliers[0]]
X = X0[:,inliers[0]]

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[0])

print('numbers of inliers: ',len(inliers[0]))
```

```
took 0.015757 secs
8 iterations
MSAC Cost=177.196449120
P =
[[ 8.77760709e+02 -6.66876347e+02  1.25741643e+03  1.21202222e+05]
 [ 1.27059396e+03 -3.42659663e+02 -8.85855155e+02  7.45937675e+04]
 [ 6.97218025e-01  6.24018410e-01  3.52828641e-01  1.75849394e+02]]
inliers:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 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, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
numbers of inliers:  46
```

Final values for parameters

- p = 0.99
- α = 0.95
- tolerance = 5.99146454710798
- num_inliers = 46
- num_attempts = 8

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 $\boldsymbol{t}_{\text{linear}}$.

In [188]:

```
def EPnP(x, X, K):
    # Inputs:
    #     x - 2D inlier points Image  inhomo       3*N
    #     X - 3D inlier points  World inhomo
    # Output:
    #     P - normalized camera projection matrix

    """your code here"""
    x_normalized = np.linalg.inv(K) @ Homogenize(x)
    x_normalized = Dehomogenize(x_normalized)
    mean_3d = np.mean(X,axis = 1).reshape(-1,1)   #3*1
    #cov_3d = ((X-mean_3d) @ (X-mean_3d).T)/X.shape[1] #3*3
    cov_3d = np.cov(X)
    U,D,VH = np.linalg.svd(cov_3d)
    var_3d = D.sum()
    S = np.sqrt(var_3d/3)
    C1_bar =  mean_3d   #3*1
    C2_bar = S*VH[0,:].T +mean_3d
    C3_bar = S*VH[1,:].T +mean_3d
    C4_bar = S*VH[2,:].T +mean_3d
```

```
    Inv_A = VH/S
    Alpha_24 = Inv_A @ (X - C1_bar)
    Alpha_1 = 1 - np.sum(Alpha_24,axis = 0)
    Alpha = np.vstack((Alpha_1,Alpha_24)) #4*n
    #print('Alpha:      ',Alpha)
    A_cam = np.ones((2*x.shape[1],12))   # Camera frames

    for i in range (x.shape[1]):
        for j in range (4):
            col_1 = np.eye(2)*Alpha[j,i]
            col_2 = -np.array([[x_normalized[0,i]],[x_normalized[1,i]]])*Alpha[j
,i]
            A_cam[2*i:2*i+2,3*j:3*j+3] = np.hstack((col_1,col_2))
    #print('A_cam:        ',A_cam)
    u,d,vh = np.linalg.svd(A_cam)
    Cam = vh[-1,:].reshape(3,4,order = 'F')
    #print('Cam:       ',Cam)
    #X_cam = Alpha[0] * Cam[0] + Alpha[1] * Cam[1]+ Alpha[2] * Cam[2]+Alpha[3] *
Cam[3]
    X_cam = np.zeros((3,X.shape[1]))     #inhom 3*n
    for i in range (4):
        X_cam += Alpha[i,:] * Cam[:,i].reshape(-1,1)


    mean_cam =  np.mean(X_cam,axis = 1).reshape(-1,1)   #3*1
    var_cam =np.trace(np.cov(X_cam))
    beta = np.sqrt(var_3d/var_cam)
    X_cam = X_cam*beta/np.sign(mean_cam[2])  #3d inhomo 3*n

    R,T, P = Pose_estimation(X_cam,X,K)
    P_normalized = np.linalg.inv(K) @ P
    return P_normalized,X_cam,Alpha

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

# 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 0.018518 secs
R_linear =
[[ 0.27861294 -0.69059043  0.66742766]
 [ 0.6610925  -0.36619528 -0.6548723 ]
 [ 0.6966574   0.62368732  0.35451741]]
t_linear =
[  5.59659327   7.51236352 175.91579455]
```

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 $\boldsymbol{t}_{\text{LM}}$.

In [181]:

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

    return y

def skew(w):
    # Returns the skew-symmetrix represenation of a vector
    """your code here"""
    w_skew = np.zeros((3,3))
    w_skew = np.array([[0,-w[2],w[1]],[w[2],0,-w[0]],[-w[1],w[0],0]])


    return w_skew


def Parameterize(R):
    # Parameterizes rotation matrix into its axis-angle representation
    # w 3*1
    """your code here"""

    iden = np.eye(R.shape[1])
    u,d,vh = np.linalg.svd(R-iden)
    v = vh[-1,:].reshape(-1,1) #3*1
    v_hat = np.array([[R[-1,1]-R[1,-1]],[R[0,-1]-R[-1,0]],[R[1,0]-R[0,1]]])#3*1
    sin_theta = (v.T @ v_hat)/2
    cos_theta = (np.trace(R)-1)/2
```

```
    theta = math.atan2(sin_theta,cos_theta)

    if theta < 0 :
        theta = theta + 2*pi
    w = theta * v/np.linalg.norm(v)

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

    return w, theta


def Deparameterize(w):
    # Deparameterizes to get rotation matrix
    """your code here"""
    theta = np.linalg.norm(w)
    R = cos(theta)*np.eye(3) + Sinc(theta)*skew(w)+ (1-cos(theta))*(w @ w.T)/the
ta**2




    return R


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   3*n world
    #
    # Output:
    #    J - Jacobian matrix of size 2*nx6
    #  x_est == x_hat in LM
    """your code here"""
    #w, theta = Parameterize(R)


    X_rotated = R @ X
    x_est = Dehomogenize(X_rotated + t)  # estimate 2d inhomo normalized
    theta = np.linalg.norm(w)

    dtheta_dw = w.T/theta   #1*3

    s = (1-cos(theta))/theta**2  #scale

    ds_dtheta = (theta*sin(theta)-2*(1-cos(theta)))/theta**3   #scale
    if theta == 0:
        dsinc = 0
    else:
        dsinc = cos(theta)/theta - sin(theta)/(theta**2)     # scale
```

```
    dXrotated_dw = np.zeros((3*X.shape[1],3))

    dxhat_dXrotated = np.zeros((2*X.shape[1],3))
    J = np.zeros((2*X.shape[1],6))

    for i in range (X.shape[1]):
        if 0 < theta < 1e-5 :  # or theta close to zero
            dXrotated_dw[3*i:3*i+3,:] = skew(-X[:,i])
        else:
            dXrotated_dw[3*i:3*i+3,:] = (Sinc(theta) * skew(-X[:,i]) + np.cross(
w.T,X[:,i]).T *dsinc @ dtheta_dw
                                         + np.cross(w.T,np.cross(w.T,X[:,i])).T*
ds_dtheta @ dtheta_dw
                                         + s*(skew(w)@skew(-X[:,i]) + skew(-np.c
ross(w.T,X[:,i]).T)))

        w_hat = X_rotated[-1,i] + t[-1]
        dxhat_dXrotated[2*i:2*i+2,:]= np.array([[1/w_hat,0,-x_est[0,i]/w_hat],[0
, 1/w_hat,-x_est[1,i]/w_hat]])
        temp = dxhat_dXrotated[2*i:2*i+2,:] @ dXrotated_dw[3*i:3*i+3,:]
        J[2*i:2*i+2,:] = np.hstack((temp,dxhat_dXrotated[2*i:2*i+2,:]))


    return J
```

In [184]:

```
def LM(P, x, X, K, max_iters, lam):
    # Inputs:
    #    P - initial estimate of camera pose  (Normalized projection matrix )
    #    x - 2D inliers  inhomo in Cam frame
    #    X - 3D inliers  inhomo in World frame
    #    K - camera calibration matrix
    #    max_iters - maximum number of iterations
    #    lam - lambda parameter
    #    R,t--- as HW 2 --- P
    #    w  ---- as HW 2 --- p_new
    # Output:
    #    P - Final camera pose obtained after convergence [R|t]  # 3*4
    # all are normalized pts
    """your code here"""
    tolerance = 1e-6
    x_normalized = Dehomogenize(np.linalg.inv(K) @ Homogenize(x))   # datum

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



    w,theta = Parameterize(R)
    R = Deparameterize(w)


    x_hat = Dehomogenize(R @ X + t)
```

```
    InvK = np.linalg.inv(K)

    J_cov = np.array([[InvK[0,0],InvK[0,1]],[0,InvK[1,1]]])
    cov_diagnoal = J_cov @ np.eye(2) @ J_cov.T

    cov_xnorm = block_diag(*([cov_diagnoal]*x.shape[1])) # 2n*2n
    Inv_cov =np.linalg.inv(cov_xnorm)
    error = np.zeros((2*x.shape[1],1))
    error = x_normalized.reshape(-1,1,order = 'F') - x_hat.reshape(-1,1,order =
'F') #2n*1
    #print('error:    ',error)
    SSE = error.T @ Inv_cov @ error  #(1,)
    P = np.zeros((3,4))


    J = Jacobian(R, w, t, X)

    print ('iter %03d Cost %.9f'%(0, SSE))


    for i in range(max_iters):
        A = J.T @ Inv_cov @ J + lam * np.eye(6)
        B = J.T @ Inv_cov @ error

        sigma = (np.linalg.inv(A) @ B ).reshape(-1,1)
        # w 3*1
        w_new = w + sigma[0:3,:]
        t_new = t + sigma[3:,:]

        R = Deparameterize(w_new)
        x_new = Dehomogenize(R @ X + t_new)
        error_new = x_normalized.reshape(-1,1,order = 'F') - x_new.reshape(-1,1,
order = 'F') #2n*1
        SSE_new = error_new.T @ Inv_cov @ error_new
        print ('iter %03d Cost %.9f'%(i+1, SSE_new))
        if SSE_new < SSE:
            if tolerance > SSE - SSE_new:
                break
            w = w_new
            t = t_new
            SSE = SSE_new   #(square of sum error)
            error = error_new        # (difference of two 2d normalized pts)
            lam = 0.1 * lam
            J = Jacobian(R, w, t, X)
        else:
            lam = 10 * lam


    P[:,:3] = R
    P[:,-1] = t_new.reshape(-1)
```

```
    return P,J


# LM hyperparameters
lam = .001
max_iters = 10

tic = time.time()
P_LM,J = 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])
print('t_LM = ')
print(P_LM[:,-1])
```

```
iter 000 Cost 61.255919945
iter 001 Cost 61.132239095
iter 002 Cost 61.132228967
iter 003 Cost 61.132228966
took 0.063895 secs
w_LM =
[[ 1.33783933]
 [-0.03144242]
 [ 1.41392914]]
R_LM =
[[ 0.27845636 -0.69072467  0.6673541 ]
 [ 0.6603758  -0.36684201 -0.65523336]
 [ 0.69739936  0.62315839  0.35398836]]
t_LM =
[  5.58040783   7.43451118 175.91832972]
```