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 compute_error(P, x, X, K):
    # Inputs:
    #    P - camera projection matrix
    #    x - 2D groundtruth image points
    #    X - 3D groundtruth scene points
    #    K - camera calibration matrix
    #
    # Output:
    #    error - total projection error

    X = Homogenize(X)
    error = []
    for p in P:
        x_est = K @ p @ X
        x_est = Dehomogenize(x_est)
        e = np.linalg.norm(x_est - x, axis = 0)
        e = np.square(e)
        error.append(e)

    return error
```

```
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
from scipy.stats import chi2

def calculate_tolerance():
    lam, var, cod = 0.95, 1, 2
    t = chi2.ppf(lam, cod) * var
    return t

def compute_cost(error, tol):
    consensus_cost = 0
    inlier_count = 0
    for i in range(error.shape[0]):
        if error[i] > tol:
            consensus_cost += tol
        else:
            consensus_cost += error[i]
            inlier_count += 1
    return consensus_cost, inlier_count

def calculate_lambda(a, b, c, alpha, beta, gamma):
    a2, b2, c2 = a**2, b**2, c**2
    cos_abg = alpha * beta * gamma
    cos2_alpha, cos2_beta, cos2_gamma = alpha ** 2, beta ** 2, gamma ** 2
    sin2_alpha, sin2_beta, sin2_gamma = 1 - cos2_alpha, 1 - cos2_beta, 1 - cos2_gamma

    G = c2 * (c2 * sin2_beta - b2 * sin2_gamma)
    H = b2 * (b2 - a2) * sin2_gamma \
        + c2 * (c2 + 2 * a2) * sin2_beta \
        + 2 * b2 * c2 * (-1 + cos_abg)
    I = b2 * (b2 - c2) * sin2_alpha \
        + a2 * (a2 + 2 * c2) * sin2_beta \
        + 2 * a2 * b2 * (-1 + cos_abg)
    J = a2 * (a2 * sin2_beta - b2 * sin2_alpha)

    coeff = [G, H, I, J]
    root = np.roots(coeff)

    for i in range(root.shape[0]):
        if np.isreal(root[i]):
            return root[i]

    print("no real number lambda!!!")
    return root.min()
```

```python
def calculate_mn(a, b, c, alpha, beta, gamma, lam):
    a2, b2, c2 = a**2, b**2, c**2
    A = 1 + lam
    B = -1 * alpha
    C = (b2 - a2 - lam * c2) / b2
    D = -lam * gamma
    E = ((a2 + lam * c2) / b2) * beta
    F = (-a2 + lam * (b2 - c2))/ b2

    if B ** 2 - A * C < 0 or E ** 2 - C * F < 0:
        return []

    p = np.sqrt(B ** 2 - A * C)
    q = np.sign(B * E - C * D) * np.sqrt(E ** 2 - C * F)

    mn = []
    mn.append(((-B + p) / C, -(E - q) / C))
    mn.append(((-B - p) / C, -(E + q) / C))

    return mn

def calculate_uv(a, b, c, alpha, beta, gamma, mn):
    a2, b2, c2 = a**2, b**2, c**2
    uv = []
    for pair in mn:
        m, n = pair[0], pair[1]
        A = b2 - (m**2) * c2
        B = c2 * (beta - n) * m - b2 * gamma
        C = -c2 * (n**2) + 2 * c2 * n * beta + b2 - c2
        if B ** 2 - A * C < 0:
            continue
        u_large = (-np.sign(B) / A) * (np.abs(B) + np.sqrt(B ** 2 - A * C))
        u_small = C / (A * u_large)
        if np.isnan(u_large) or np.isnan(u_small):
            continue
        uv.append((u_large, u_large * m + n))
        uv.append((u_small, u_small * m + n))
    return uv

def calculate_S(a, b, c, alpha, beta, gamma, uv):
    S = []
    for uv_pair in uv:
        u, v = uv_pair[0], uv_pair[1]
        s1_2 = (a ** 2) / (u ** 2 + v ** 2 - 2 * u * v * alpha)
        s1 = np.sqrt(s1_2)
        S.append(np.array([s1, u * s1, v * s1]))
    return S
```

In [3]:

```
def linearEstimate(X, x_hat, mean):
    mean = np.array([mean]).T
    B = (X - np.tile(mean, (1, X.shape[1])))
    mean_p = np.array([np.mean(x_hat, axis = 1)]).T
    C = (x_hat - np.tile(mean_p, (1, X.shape[1])))

    S = C @ B.T
    U, S, V = np.linalg.svd(S)
    V = V.T

    R = np.eye(3)
    if 0 > np.linalg.det(U) * np.linalg.det(V):
        R = U @ np.diag([1, 1, -1]) @ V.T
    else:
        R = U @ V.T
    t = mean_p - R @ mean
    t = t.reshape((3, 1))
    P = np.concatenate((R, t), axis=1)

    return P
```

In [4]:

```
def P3P(X, x_homo, K):

    K_inv = np.linalg.inv(K)
    x_normalized = K_inv @ x_homo

    ##################### calculate known stuff
    ### calculate q
    f = K[0, 0]
    q = f * (x_normalized[:-1] / x_normalized[-1])
    q = np.vstack((q, np.ones((1, 3)) * f))

    ### calculate unit vector j
    norm = np.linalg.norm(q, axis = 0)
    norm = np.tile(norm, (3, 1))
    j = q / norm

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

    ### calculate angles
    c_alpha = j[:, 1] @ j[:, 2]
    c_beta = j[:, 0] @ j[:, 2]
    c_gamma = j[:, 0] @ j[:, 1]

    ##################### find P1, P2, P3
    lam = calculate_lambda(a, b, c, c_alpha, c_beta, c_gamma)
    lam = np.real(lam)
    mn = calculate_mn(a, b, c, c_alpha, c_beta, c_gamma, lam)
    uv = calculate_uv(a, b, c, c_alpha, c_beta, c_gamma, mn)
    S = calculate_S(a, b, c, c_alpha, c_beta, c_gamma, uv)

    solution = []
    for i in range(len(S)):
        s = S[i]
        tmp = np.zeros(j.shape)
        for k in range(j.shape[1]):
            tmp[:, k] = j[:, k] * s[k]
        solution.append(tmp)

    ################ find R, t
    P_sol = []
    for p_cam in solution:
        #P_sol.append(find_Rt(X, p_cam))
        tmp = linearEstimate(X, p_cam, np.mean(X, axis = 1))
        P_sol.append(linearEstimate(X, p_cam, np.mean(X, axis = 1)))

    return P_sol
```

In [5]:

```python
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 contai
n 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

    total_point_num = X.shape[1]
    x_homo, X_homo = Homogenize(x), Homogenize(X)

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

    while trials < max_trials and consensus_min_cost > thresh:
        idx = np.random.choice(X.shape[1], 3, replace = False)
        x_homo_sampled, X_sampled = x_homo[:, idx], X[:, idx]
        P = P3P(X_sampled, x_homo_sampled, K)

        if len(P) == 0:
            continue

        ###### compute error for each set of error
        error = compute_error(P, x, X, K)
        inlier_count = 0
        consensus_cost = np.inf
        model = P[0]

        ###### compute cost for each set of cost and take the smallest one
        for i in range(len(error)):
            cost, cnt = compute_cost(error[i], tol)
            if cost < consensus_cost:
                consensus_cost = cost
                inlier_count = cnt
                model = P[i]

        ###### determine if this is the model we want
        if consensus_cost < consensus_min_cost:
            consensus_min_cost = consensus_cost
            consensus_min_cost_model = model
```

```
            ###### adaptive maxtrials
            w = inlier_count / total_point_num
            max_trials = np.log(1 - 0.99) / np.log(1 - w ** 3)

        trials += 1

    #### find set of inliers
    error = compute_error([consensus_min_cost_model], x, X, K)
    error = error[0]
    for i in range(total_point_num):
        if error[i] <= tol:
            inliers.append(i)

    return consensus_min_cost, consensus_min_cost_model, inliers, trials


# MSAC parameters
thresh = 100
tol = calculate_tolerance()
p = 0
alpha = 0

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)
print('inliers count:', len(inliers))
```

```
took 0.026268 secs
MSAC Cost=190.124982976
P =
[[  0.28167008  -0.6875954    0.66923428   5.96805104]
 [  0.66384904  -0.36394671  -0.65333548   7.82695738]
 [  0.69279609   0.62829559   0.35394665 176.1326423 ]]
inliers:  [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 17, 18, 19
, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 39, 40, 41
, 42, 43, 44, 46, 47, 48, 49]
inliers count: 41
```

Final values for parameters

- p = 0.99
- α = 0.95
- tolerance = 5.991464547107979
- num_inliers = 41
- num_attempts = 26

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 [7]:

```
def worldControlPoint(X, mean, V, s):
    C1 = mean
    C2 = s * V[:, 0] + mean
    C3 = s * V[:, 1] + mean
    C4 = s * V[:, 2] + mean
    return C1, C2, C3, C4

def cameraControlPoint(alpha, x_hat):
    x_hat_inhomo = Dehomogenize(x_hat)
    A = np.zeros((1, 12))
    for i in range(x_hat_inhomo.shape[1]):
        a1, a2, a3, a4 = alpha[0, i], alpha[1, i], alpha[2, i], alpha[3, i]
        xi, yi = x_hat_inhomo[0, i], x_hat_inhomo[1, i]
        array = np.array([[a1, 0, -a1 * xi, a2, 0, -a2 * xi, a3, 0, -a3 * xi, a4
, 0, -a4 * xi],
                          [0, a1, -a1 * yi, 0, a2, -a2 * yi, 0, a3, -a3 * yi, 0,
a4, -a4 * yi]])
        A = np.vstack((A, array))

    A = A[1:, :]
```

```python
    U, D, V = np.linalg.svd(A)

    V = V.T
    control = V[:, -1].reshape((3, 4), order='F')
    return control

def findAlpha(s, V, mean, X):
    mean = np.array([mean]).T
    mean = np.tile(mean, (1, X.shape[1]))
    b = X - mean  #### all - vector
    A_inv = (1 / s) * V.T

    alpha = A_inv @ b
    alpha1 = np.ones(alpha[0].size) - alpha[0] - alpha[1] - alpha[2]
    alpha = np.vstack((alpha1, alpha))
    return alpha

def scale3DPoints(x_parameterized, total_var):
    mean = np.mean(x_parameterized, axis = 1)
    cov = np.cov(x_parameterized)
    U, D, V = np.linalg.svd(cov)
    total_var_par = np.diag(D).sum()

    if mean[2] < 0:
        beta = -np.sqrt(total_var / total_var_par)
    else:
        beta = np.sqrt(total_var / total_var_par)
    return beta * x_parameterized

def parameterize3DPoints(x_hat, mean, X, isWorld = True):
    cov = np.cov(X)
    U, D, V = np.linalg.svd(cov)
    V = V.T

    total_var = np.diag(D).sum()
    s = np.sqrt(total_var / 3)
    alpha = findAlpha(s, V, mean, X)
    cam_control = cameraControlPoint(alpha, x_hat)
    x_parameterized = cam_control @ alpha

    return scale3DPoints(x_parameterized, total_var)
```

In [14]:

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

    """your code here"""
    x_homo = Homogenize(x)
    x_hat = np.linalg.inv(K) @ x_homo
    mean = np.mean(X, axis = 1)

    X_cam = parameterize3DPoints(x_hat, mean, X, isWorld = True)
    P = linearEstimate(X, X_cam, mean)

    proj = P @ Homogenize(X)

    return  P

tic=time.time()
P_linear = 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 0.006673 secs
R_linear =
[[ 0.27992735 -0.68962973  0.66787088]
 [ 0.66319206 -0.3640928  -0.65392104]
 [ 0.69413037  0.62597705  0.35543743]]
t_linear =
[  5.72967796   7.75667331 175.94218393]
```

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 [11]:

```
from scipy.linalg import block_diag

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


def skew(w):
    w = w.reshape((3, -1))
    w_skew = np.matrix([[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
    I = np.identity(R.shape[0])
    U, S, V = np.linalg.svd(R - I)
    V = V.T

    v = V[:, -1]
    v_hat = np.array([R[2, 1] - R[1, 2], R[0, 2] - R[2, 0], R[1, 0] - R[0, 1]]).
reshape((3, 1))

    sin_theta = (v.T @ v_hat) / 2
    cos_theta = (np.trace(R) - 1) / 2
    theta = np.arctan2(sin_theta, cos_theta)
    if theta.shape == (1, 1):
        theta = theta[0, 0]

    w = theta * (v / np.linalg.norm(v))
```

```
    theta = np.linalg.norm(w)

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

    return w, theta


def Deparameterize(w):
    theta = np.linalg.norm(w)
    w = w.reshape((3, 1))
    return np.cos(theta) * np.identity(3) + Sinc(theta) * skew(w) + (((1 - np.co
s(theta)) / (theta ** 2))) * w @ w.T

def dSinc(x):
    if x == 0:
        return 0
    else:
        return np.cos(x) / x - np.sin(x) / np.square(x)

def S(theta):
    return (1 - np.cos(theta)) / np.power(theta, 2)

def ds_dtheta(theta):
    return (theta * np.sin(theta) - 2 * (1 - np.cos(theta))) / np.power(theta, 3
)

def dtheta_dw(w):
    return (w.T / np.linalg.norm(w))

def dXrotated_dw(theta, X, w):
    if theta == 0:
        return skew(-1 * X)
    else:
        s = S(theta)
        return Sinc(theta) * skew(-1 * X) + \
                np.cross(w, X, axis = 0) @ (dSinc(theta) * dtheta_dw(w)) + \
                np.cross(w, np.cross(w, X, axis = 0), axis = 0) @ (ds_dtheta(the
ta) * dtheta_dw(w)) + \
                s * (skew(w) @ skew(-1 * X) + skew(-1 * np.cross(w, X, axis = 0)
))

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
```

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

    theta = np.linalg.norm(w)

    X_rotated = R @ X
    x_hat = Dehomogenize(P @ Homogenize(X))
    w_hat = X_rotated[2, :] + t[2]
    w_hat = w_hat.reshape((1, X.shape[1]))

    J = np.zeros((1,6))
    for i in range(X.shape[1]):
        tmp_w = 1 / w_hat[0, i]
        dxi_dt = np.array([[tmp_w, 0, -x_hat[0, i] / w_hat[0, i]],
                           [0, tmp_w, -x_hat[1, i] / w_hat[0, i]]])
        dxi_drotated = np.array([[tmp_w, 0, -x_hat[0, i] / w_hat[0, i]],
                                 [0, tmp_w, -x_hat[1, i] / w_hat[0, i]]])

        drotated_dw = dXrotated_dw(theta, X[:, i], w)  ## 3x3
        dxi_dw = dxi_drotated @ drotated_dw
        Ai = np.hstack((dxi_dw, dxi_dt))
        J = np.vstack((J, Ai))

    return J[1:, :]
```

In [13]:

```
def estimate_e(P, X_homo, x_meas):
    x_est = P @ X_homo
    e = x_meas - Dehomogenize(x_est)
    return e

def covPropagation(K):
    K = np.linalg.inv(K)
    J = K[:2, :2]
    cov = J @ np.identity(2) @ J.T
    return cov

def normalEqMat(J, invcov):
    U = np.zeros((6, 6))
    for i in range(J.shape[0] // 2):
        Ai = J[i * 2: i * 2 + 2]
        U = U + Ai.T @ invcov @ Ai
    return U

def normalEqVec(J, invcov, e):
    E = np.zeros((6, 1))
    for i in range(J.shape[0] // 2):
        Ai = J[i * 2: (i + 1) * 2 ]
        tmp = e[:, i]
        tmp = tmp.reshape((2, 1))
        E = E + Ai.T @ invcov @ tmp
    return E
```

```python
def weightedSSE(e, invcov):

    SSE = 0
    tmp = e[:, 0].reshape(2, 1)
    c = np.eye(e.shape[1] * 2) * invcov[0, 0]

    for i in range(1, e.shape[1]):
        tmp = np.vstack((tmp, e[:, i].reshape(2, 1)))
    SSE = tmp.T @ c @ tmp
    return SSE

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

    ## fixed
    invcov = np.linalg.inv(covPropagation(K))
    X_homo = Homogenize(X)
    x_meas = Dehomogenize(np.linalg.inv(K) @ Homogenize(x))

    ## initial round
    e = estimate_e(P, X_homo, x_meas)
    SSE = weightedSSE(e, invcov)
    R, t = P[:3, :3], P[:, -1]
    w, theta = Parameterize(R)
    p = np.hstack((w, t)).reshape((6, 1))
    w, t = p[:3], p[3:]
    print ('iter %03d Cost %.15f'%(0, SSE))

    for i in range(max_iters):
        J = Jacobian(R, w, t, X)
        U = normalEqMat(J, invcov)
        while(True):
            inv = np.linalg.inv(U + lam * np.identity(6))
            E = normalEqVec(J, invcov, e)
            update = inv @ E

            ### update newp, P
            newp = p + update
            w, t = newp[:3], newp[3:]
            R = Deparameterize(w)
            newP = np.concatenate((R, t), axis=1)

            ### calculate new SSE
            newe = estimate_e(newP, X_homo, x_meas)
            newSSE = weightedSSE(newe, invcov)
```

```python
            if newSSE < SSE:
                thresh = SSE - newSSE
                SSE = newSSE
                e = newe
                p = newp
                P = newP
                lam = 0.1 * lam
                break
            elif newSSE - SSE > 0.0005:
                lam = 10 * lam
            else:
                break

        print ('iter %03d Cost %.15f'%(i+1, SSE))

        if thresh < 10 ** -15:
            break
        thresh = 0

    return P


# 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[:,:3].reshape(3, 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 48.490665208647826
iter 001 Cost 48.457040347144684
iter 002 Cost 48.457023692765304
iter 003 Cost 48.457023685140967
iter 004 Cost 48.457023685137820
iter 005 Cost 48.457023685136711
iter 006 Cost 48.457023685136711
took 0.144951 secs
w_LM =
[[ 1.33627092]
 [-0.02809921]
 [ 1.41206027]]
R_LM =
[[ 0.27981274 -0.68973777  0.66780733]
 [ 0.66262427 -0.36459979 -0.65421409]
 [ 0.69471858  0.62556278  0.35501732]]
t_LM =
[[  5.71686918]
 [  7.69512062]
 [175.94615506]]
```

In [ ]: