# Computer Vision 252B Hw3

**Hao-en Sung (wrangle1005@gmail.com)**
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

## 1 Programming: Estimation of the camera pose (rotation and translation of a calibrated camera) (80 points)

In this problem, we assume the camera is calibrated. In other words, the calibration matrix is given as follows.

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

### 1.1 Outlier rejection (20 points)

In my implementation, I first homogenize both normalized 2D points $k^{-1} \cdot x$ and 3D points $X$. Later, I will enter a while loop to estimate the best rotation and translation matrix, which is also named as *mSAC* algorithm.

In each iteration, I need to random sample three correspondence of 2D and 3D points. After that, I will use *Finsterwalder* algorithm, which is proposed in link, to estimate all potential 3D points in camera coordinates from 2D points. The conclusion of that algorithm can be summarized as follows. Firstly, it will use an addition constraint — set a perfect square to 0 — to find out $\lambda_0$. It is noticeable that there might be one or three valid (real) values for $\lambda_0$. If there are three solutions, I will just select arbitrary of them. Secondly, I can use $\lambda_0$ to solve two potential $u$ and two corresponding potential $v$, i.e. there are going to be four pairs of $(u, v)$. Thirdly, with each valid pair (real-valued) $(u, v)$, I can use *Umeyama* to find out rotation matrix $R$ and translation matrix $t$.

*Umeyama* algorithm is designed to find out the rotation matrix $R$ and translation matrix $t$ for multiple pairs of 3D points in Camera Coordinate and 3D points in World Coordinate, which is proposed in link (formula correction: link). The big picture of this algorithm is to decompose covariance between 3D points in Camera Coordinate and 3D points in World Coordinate. It is noticeable that the rank of covariance matrix may not be full-rank, i.e. rank 3, or the determine of it may not be positive. If any of these situations occur, some formula refinement is needed in order to provide correct results.

From second paragraph, it is known that there might be several valid $(u, v)$ pairs, or said $(R, t)$ pairs. If that is the case, I then need to find out the one with smallest cost and examine where it is competitive with the current outlier threshold. If the found $(R, t)$ pair in this iteration is better than all the previous ones, I will keep this $(R, t)$ pair as well as the inliers set; or I will just skip it and start the next iteration.

To calculate the error, or said cost, I need to first derive a reasonable threshold $t$, which is defined as

$$t^2 = F_m^{-1}(\alpha) \cdot \sigma^2,$$

where $F_m^{-1}(\cdot)$ is the Inverse Chi-Squared Cumulative Distribution Function with $m = 2$ degrees of freedom, i.e. codimension, $\alpha = 0.95$ and $\sigma = 1$.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Beside all the above-mentioned algorithms, another key issue is to adaptively decide the proper *MAX_TRIALS* for *mSAC* algorithm. It can be approached by the following equations.

$$w = \frac{\text{NUMBER OF INLIERS}}{\text{TOTAL NUMBER OF DATA POINTS}}$$
$$p = 0.99$$
$$\text{MAX\_TRIALS} = \frac{\log(1-p)}{\log(1-w^s)}$$

At the end, *mSAC* algorithm each time will select around 30 to 50 inliers within around ten iterations. With fixed random seed *RandStream('mt19937ar','Seed',514)*, it will find out 38 inliers within *MAX_TRIALS*= 15.7130502006309. It is interesting to notice that the Rooted Mean Squared Error (RMSE) with best found rotation matrix $R$ and translation matrix $t$ is around 1.47459827831417.

### 1.2 Linear estimation (30 points)

*EPnP* is a real time (linear time) algorithm to find out the camera pose. The main framework of EPnP can be summarized as follows.

1. Find out four Control Points in World Coordinate
2. For each 3D points $X_i$ in World Coordinate, find out $\alpha_{i,1}, \cdots \alpha_{i,4}$
3. Find out four Control Points in Camera Coordinate
4. For each 3D points $X_i$ in World Coordinate, find out corresponding points in Camera Coordinate with $\alpha_{i,1}, \cdots \alpha_{i,4}$
5. Use *Umeyama* algorithm to estimate rotation matrix $R$ and translation matrix $t$

To calculate four Control Points in World Coordinate, I need to first calculate the mean $\mu$, variance $Var$ and covariance matrix $C$ for all 3D points $X_i$. After that I can decompose $C$ as $C = UZV$. Later, I will simply define

$$C_1 = \mu,$$
$$C_2 = \mu + \frac{\Sigma_i Var_i}{3} \cdot V(:,1),$$
$$C_3 = \mu + \frac{\Sigma_i Var_i}{3} \cdot V(:,2),$$
$$C_4 = \mu + \frac{\Sigma_i Var_i}{3} \cdot V(:,3).$$

After retrieve the control points in World Coordinate, I can estimate $\alpha_{i,1}, \cdots \alpha_{i,4}$ for all points in World Coordinate by solving

$$[C_2 - C_1, C_3 - C_1, C_4 - C_1] \cdot \begin{bmatrix} \alpha_{i,2} \\ \alpha_{i,3} \\ \alpha_{i,4} \end{bmatrix} = X_i - C_1.$$

The result of previous formula can be easily solved with matrix division. Later, I can obtain $a_{i,1}$ as $a_{i,1} = 1 - \alpha_{i,2} - \alpha_{i,3} - \alpha_{i,4}$.

For third step, I am going to solve the following equation.

$$\begin{bmatrix} a_{i,1}, 0, -a_{i,1} \cdot x_{i,1}, a_{i,2}, 0, -a_{i,2} \cdot x_{i,1}, a_{i,3}, 0, -a_{i,3} \cdot x_{i,1}, a_{i,4}, 0, -a_{i,4} \cdot x_{i,1} \\ a_{i,1}, 0, -a_{i,1} \cdot x_{i,2}, a_{i,2}, 0, -a_{i,2} \cdot x_{i,2}, a_{i,3}, 0, -a_{i,3} \cdot x_{i,2}, a_{i,4}, 0, -a_{i,4} \cdot x_{i,2} \end{bmatrix} \cdot \begin{bmatrix} C_{cam_1} \\ C_{cam_2} \\ C_{cam_3} \\ C_{cam_4} \end{bmatrix} = 0$$

This equation can be solved similarly to the finding out the Control Points in World Coordinate: extract the left column of $V$ in $UZV$ as the right null space.

Fourthly, I can use the $\alpha$ matrix, which is obtained in step 2, to reform all 3D points in 3D Camera Coordinate, denoted as $X_{c_i}$.

Lastly, I can reuse *Umeyama* algorithm to retrieve rotation matrix $R$ and translation matrix $t$ with all pairs of $X_i$ and $X_{c_i}$ and tell the error.

To evaluate the result, I use the same random seed as the previous *mSAC* problem, i.e. *RandStream('mt19937ar','Seed',514)*. and obtain Rooted Mean Squared Error (RMSE) around 1.21531149866683. The rotation matrix $R$ and translation matrix $t$ are recorded as follows.

$$R = \begin{bmatrix} 0.276014107766956 & -0.692587156102531 & 0.666437726659753 \\ 0.65947295906159 & -0.367952681532405 & -0.65551982458173 \\ 0.699222159674707 & 0.620430379136064 & 0.355182370149362 \end{bmatrix}$$

$$t = \begin{bmatrix} 5.34057941777919 \\ 7.3501788426026 \\ 176.01076234755 \end{bmatrix}$$

### 1.3 Nonlinear estimation (30 points)

For nonlinear estimation, I need to use *Levenberg–Marquardt* algorithm (LM algorithm) again here. The main framework of it can be summarized as follows.

1. Form a covariance matrix $\Sigma$
2. Calculate the initial error $\epsilon \cdot \Sigma^{-1} \cdot \epsilon'$
3. Conduct angle normalization if needed
4. Calculate *Jacobian* matrix for gradient update
5. Select a proper scalar $\lambda$ as step size
6. Update all parameters if there is an improvement

For first step, I refer to one of the posts in Piazza. It describes that

$$\begin{aligned} \hat{x_i} &= K^{-1} \cdot \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2} & a_{2,3} \\ 0 & 0 & a_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} a_{1,1} & a_{1,2} \\ 0 & a_{2,2} \end{bmatrix} \cdot x_i + \begin{bmatrix} a_{1,3} \\ x_{2,3} \end{bmatrix} \end{aligned}$$

With $\Sigma_x = 1$ and the property of convariance propagation, I have

$$\begin{aligned} J = \frac{\partial \hat{x}}{\partial x} &= \begin{bmatrix} a_{1,1} & a_{1,2} \\ 0 & a_{2,2} \end{bmatrix} \\ \Sigma_{\hat{x}} &= J \cdot \Sigma_x J' = J \cdot J'. \end{aligned}$$

Thus, I will form a matrix $\Sigma$ with size $2n \times 2n$, which is filled up with $J \cdot J'$ in its diagonal.

Second part is a kind of error estimation. In my implementation, I will stop this algorithm when two consecutive errors differ less than $10^{-4}$.

Angle normalization in third part is also a very crucial procedure. With this normalization, it prevents singularity problem in later matrix calculation.

Fourth part is the most challenging part for me in this homework. Professor does not provide sufficient information or even hint in class. Thus, I refer to one paper, *A compact formula for the derivative of a 3-D rotation in exponential coordinates* in link for 3D exponential coordinate rotation derivatives and derive the rest of derivatitives by myself. It is known that the *Jacobian* matrix for this part can be

written as

$$\begin{aligned} J_i &= \frac{\partial x_i}{\partial (W,t)} \\ &= \frac{\partial x_i}{\partial (R,t)} \cdot \frac{\partial (R,t)}{\partial (W,t)} \\ &= \begin{bmatrix} \frac{\partial x_{i,1}}{\partial R} & \frac{\partial x_{i,1}}{\partial t} \\ \frac{\partial x_{i,2}}{\partial R} & \frac{\partial x_{i,2}}{\partial t} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial R}{\partial W} & \frac{\partial R}{\partial t} \\ \frac{\partial t}{\partial W} & \frac{\partial t}{\partial t}. \end{bmatrix} \end{aligned}$$

For the first part of $J$, the derivative of it can be calculated as

$$\begin{bmatrix} \frac{\partial x_{i,1}}{\partial R} & \frac{\partial x_{i,1}}{\partial t} \\ \frac{\partial x_{i,2}}{\partial R} & \frac{\partial x_{i,2}}{\partial t} \end{bmatrix} = \begin{bmatrix} X_i' & 0^T & -x_{i,1}X_i' & 1 & 0 & -x_{i,1} \\ 0^T & X_i' & -x_{i,2}X_i' & 0 & 1 & -x_{i,2}. \end{bmatrix}$$

For calculating $\frac{\partial R}{\partial W}$ in second part of $J$, I simply copy the derivative from the paper, *A compact formula for the derivative of a 3-D rotation in exponential coordinates*, which can be shown as follows.

$$\begin{aligned} \frac{\partial R}{\partial W_i} = \; & \cos(\theta)\bar{w}_i[\bar{W}_i]_\times + \sin(\theta)\bar{w}_i[\bar{W}_i]_\times^2 + \frac{\sin(\theta)}{\theta}[e_i - \bar{w}_i\bar{W}_i] \quad , \\ & + \frac{1-\cos(\theta)}{\theta}(e_i\bar{W}_i' + \bar{W}_i e_i' - 2\bar{w}_i\bar{W}_i\bar{W}_i') \end{aligned}$$

where

$$\begin{aligned} \theta &= \|W_i\| \\ \bar{W}_i &= (\bar{w}_{i,1}, \bar{w}_{i,2}, \bar{w}_{i,3})^T = \frac{W_i}{\|W_i\|}. \end{aligned}$$

The rest of the gradient are trivial. I then can rewrite the second part as

$$\begin{bmatrix} \frac{\partial R}{\partial W_1} & \frac{\partial R}{\partial W_2} & \frac{\partial R}{\partial W_3} & 0 \\ & 0 & & I. \end{bmatrix}$$

At the end, the *Jacobian* matrix for instance $i$ is going to be

$$J_i = \begin{bmatrix} X_i' & 0^T & -x_{i,1}X_i' & 1 & 0 & -x_{i,1} \\ 0^T & X_i' & -x_{i,2}X_i' & 0 & 1 & -x_{i,2}. \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial R}{\partial W_1} & \frac{\partial R}{\partial W_2} & \frac{\partial R}{\partial W_3} & 0 \\ & 0 & & I. \end{bmatrix}.$$

Step five is straightforward: if current step size is too large to produce poor result, then just scale down $\lambda$ by multiplying $0.1$; otherwise, I will increase the step size by multiplying $10$.

The last step is also easy to understand. It is noticeable that once the parameters are updated, I need to sync $R$ and $W$ to make sure they are still convertible to each other. At the end, the best parameter will be return to main function.

To evaluate the result, I again use the same random seed, i.e. *RandStream('mt19937ar','Seed',514).* and obtain Rooted Mean Squared Error (RMSE) around $1.21490877283495$. The rotation matrix $R$, translation matrix $t$, parameterized $W$, and error logs are recorded as follows.

$$R = \begin{bmatrix} 0.275865232803052 & -0.692697770091679 & 0.666384403059176 \\ 0.65901469142486 & -0.368380095500437 & -0.655740605518131 \\ 0.699712805239917 & 0.620053146552056 & 0.354874746402418 \end{bmatrix}$$

$$t = \begin{bmatrix} 5.326269646828 \\ 7.30037486990607 \\ 176.015115697155 \end{bmatrix}$$

$$W = \begin{bmatrix} 1.33723902840539 \\ -0.0349335776869578 \\ 1.41681416434966 \end{bmatrix}$$

$$log = [56.1253174740863 \quad 56.0881269253346 \quad 56.0881263998321]$$

### Summary

From three-point estimation to EPnP estimation, it can be found that the RMSE drops dramatically from 1.47459827831417 to 1.21531149866683. Later, non-linear learning process further improves this error to 1.21490877283495.

## Appendix

### Problem 1

Code Listing 1: mSAC Algorithm

```
function [best_R, best_t, best_bmap, MAX_TRIALS] = mSAC(hx, hX, K)
    % Parameters
    CONSENSUS_MIN_COST = Inf;
    MAX_TRIALS = Inf;
    THRESHOLD = 1;
    TOLERANCE = chi2inv(0.95,2);
    PROBABILITY = 0.99;

    % Initialize return values
    best_R = zeros(3,3);
    best_t = zeros(3,1);
    best_bmap = false(size(hx,1),1);

    % Normalize x
    nx = (K\hx')';

    % Dehomogenize
    Xw = hX(:,1:3);

    trials = 0;
    s = RandStream('mt19937ar','Seed',514);
    while trials < MAX_TRIALS && CONSENSUS_MIN_COST > THRESHOLD
        trials = trials + 1;

        % Random 3 numbers
        idx = randperm(s, size(nx,1));

        % Find Camera Coordinate
        XcLst = Finsterwalder(nx(idx(1:3),:), Xw(idx(1:3),:)); % cell

        % Enuerate all pairs
        sbest_cost = Inf;
        for i = 1:length(XcLst)
            % Find R, t
            [status, R, t] = Umeyama(Xw(idx(1:3),:), XcLst{i});

            if status == 1
                % Calculate error
                error = calError(hx, hX, K, R, t);

                % Calculate cost
                cost = calCost(error, TOLERANCE);

                % Preserve the best model
                if cost < sbest_cost
                    sbest_R = R;
                    sbest_t = t;
                    sbest_bmap = error < TOLERANCE;
                    sbest_n_in = sum(sbest_bmap);
                    sbest_cost = cost;
                end
            end
```

```
        end

        if sbest_cost < CONSENSUS_MIN_COST
            CONSENSUS_MIN_COST = sbest_cost;
            best_R = sbest_R;
            best_t = sbest_t;
            best_bmap = sbest_bmap;

            w = sbest_n_in / size(nx,1);
            MAX_TRIALS = log(1-PROBABILITY) / log(1-w^3);
        end
    end
end

function error = calError(hx, hX, K, R, t)
    px = (K * [R, t] * hX')';
    px = px ./ px(:,3);
    diff = hx(:,1:2) - px(:,1:2);
    error = sum(diff.^2,2);
end

function cost = calCost(error, TOLERANCE)
    cost = sum(min(error, TOLERANCE));
end
```

Code Listing 2: Finsterwalder Algorithm

```
function XcLst = Finsterwalder(nx, Xw)
    assert(size(Xw,1) == 3 && size(nx,1) == 3);

    % Normalize nx by nx(:,3)
    nx = nx ./ nx(:,3);

    % Calculate distance
    a = norm(Xw(2,:) - Xw(3,:));
    b = norm(Xw(1,:) - Xw(3,:));
    c = norm(Xw(1,:) - Xw(2,:));

    % Calculate unit vector
    j1 = 1/sqrt(nx(1,1)^2 + nx(1,2)^2 + 1) * nx(1,:);
    j2 = 1/sqrt(nx(2,1)^2 + nx(2,2)^2 + 1) * nx(2,:);
    j3 = 1/sqrt(nx(3,1)^2 + nx(3,2)^2 + 1) * nx(3,:);

    % Calculate angles
    alpha = acos(dot(j2,j3));
    beta =  acos(dot(j1,j3));
    gamma = acos(dot(j1,j2));

    % solve lambda (4 solutions)
    G = c^2 * (c^2*sin(beta)^2 - b^2*sin(gamma)^2);
    H = b^2 * (b^2-a^2) * sin(gamma)^2 ...
        + c^2 * (c^2 + 2*a^2) * sin(beta)^2 ...
        + 2*b^2*c^2 * (-1 + cos(alpha)*cos(beta)*cos(gamma));
    I = b^2 * (b^2-c^2) * sin(alpha)^2 ...
        + a^2 * (a^2 + 2*c^2) * sin(beta)^2 ...
        + 2*a^2*b^2 * (-1 + cos(alpha)*cos(beta)*cos(gamma));
    J = a^2 * (a^2*sin(beta)^2 - b^2*sin(alpha)^2);
    r = roots([G, H, I, J]);
    assert(size(r,1) == 3);

    % Choose a lambda \in R
    tag = 0;
    for i = 1:3
        if isreal(r(i))
            tag = i;
```

```
            break;
        end
    end
    assert(tag ~= 0);
    lambda = r(tag);

    % Solve for p, q
    A = 1 + lambda;
    B = -cos(alpha);
    C = (b^2-a^2)/(b^2) - lambda*c^2/b^2;
    D = -lambda * cos(gamma);
    E = ((a^2/b^2) + lambda*c^2/b^2) * cos(beta);
    F = -a^2/b^2 + lambda * ((b^2-c^2)/b^2);
    p = sqrt(B^2 - A*C);
    q = sign(B*E - C*D) * sqrt(E^2 - C*F);

    % Solve for u, v
    uvLst = [];
    mLst = [(-B+p)/C, (-B-p)/C];
    nLst = [(-E+q)/C, (-E-q)/C];
    for i = 1:2
        m = mLst(i);
        n = nLst(i);
        AA = b^2 - m^2 * c^2;
        BB = c^2*(cos(beta)-n)*m - b^2*cos(gamma);
        CC = -c^2*n^2 + 2*c^2*n*cos(beta) + b^2 - c^2;

        u_large = -sign(BB)/AA * (abs(BB) + sqrt(BB^2-AA*CC));
        v_large = u_large*m + n;
        if isreal(u_large) && isreal(v_large)
            uvLst = [uvLst; [u_large, v_large]];
        end

        u_small = CC / (AA*u_large);
        v_small = u_small*m + n;
        if isreal(u_small) && isreal(v_small)
            uvLst = [uvLst; [u_small, v_small]];
        end
    end

    % Calculate Xc
    XcLst = {};
    for i = 1:size(uvLst,1)
        u = uvLst(i,1);
        v = uvLst(i,2);
        s1 = sqrt(c^2 / (1 + u^2 - 2*u*cos(gamma)));
        s2 = u * s1;
        s3 = v * s1;
        points = [s1*j1; s2*j2; s3*j3];

        varXc = var(points);
        varXw = var(Xw);
        points = points * sqrt(sum(varXw) / sum(varXc));
        points = points ./ sign(points(:,3));

        XcLst{i} = points;
    end
end
```

Code Listing 3: Umeyama Algorithm

```
function [status, R, t] = Umeyama(Xw, Xc)
    assert(size(Xw,1) == size(Xc,1));

    meanXw = mean(Xw);
```

```
    meanXc = mean(Xc);
    covXwXc = zeros(3,3);
    for i = 1:size(Xw,1)
        covXwXc = covXwXc + (Xc(i,:)-meanXc)' * (Xw(i,:)-meanXw);
    end
    covXwXc = covXwXc / size(Xw,1);

    [U, ~, V] = svd(covXwXc);

    if rank(covXwXc) == 1
        status = 0;
        R = zeros(3,3);
        t = zeros(3,1);
    else
        status = 1;
        if rank(covXwXc) == 3
            S = eye(3);
            if det(covXwXc) < 0
                S(3,3) = -1;
            end
        else
            S = eye(3);
            if det(U) * det(V) < 0
                S(3,3) = -1;
            end
        end
        R = U * S * V';
        t = meanXc' - R * meanXw';
    end
end
```

**Problem 2**

Code Listing 4: EPnP Algorithm

```
function [R, t] = EPnP(hx, hX, K)
    % Find World Control Points
    [cw1, cw2, cw3, cw4] = calWorldControlPoints(hX);

    % Find Alpha
    alpha = calAlpha(hX, cw1, cw2, cw3, cw4);

    % Find Camera Control Points
    [cc1, cc2, cc3, cc4] = calCameraControlPoints(alpha, hx, K);

    % Find Camera Points
    Xc = calCameraPoints(hX, cc1, cc2, cc3, cc4, alpha);

    % Find out R, t
    [status, R, t] = Umeyama(hX(:,1:3), Xc);
    assert(status == 1);
end

function [cw1, cw2, cw3, cw4] = calWorldControlPoints(hX)
    % Dehomogenize
    Xw = hX(:,1:3);

    meanX = mean(Xw);
    varXw = var(Xw);
    covXw = cov(Xw);
    sX = sqrt(sum(varXw) / 3);

    [~, ~, V] = svd(covXw);
```

```
    cw1 = meanX;
    cw2 = sX * V(:,1)' + meanX;
    cw3 = sX * V(:,2)' + meanX;
    cw4 = sX * V(:,3)' + meanX;
end

function alpha = calAlpha(hX, cw1, cw2, cw3, cw4)
    % Dehomogenize
    Xw = hX(:,1:3);

    alpha = zeros(size(Xw,1),4);
    A = [cw2-cw1; cw3-cw1; cw4-cw1];
    b = Xw - cw1;
    alpha(:,2:4) = b/A;
    alpha(:,1) = 1 - sum(alpha, 2);
end

function [cc1, cc2, cc3, cc4] = calCameraControlPoints(alpha, hx, K)
    % Normalize x
    nx = (K\hx')';

    % Normalize nx by nx(:,3)
    nx = nx ./ nx(:,3);

    A = zeros(2*size(nx,1), 12);
    for i = 1:size(nx,1)
        A(2*i-1,:) = [alpha(i,1), 0, -alpha(i,1) * nx(i,1), ...
                      alpha(i,2), 0, -alpha(i,2) * nx(i,1), ...
                      alpha(i,3), 0, -alpha(i,3) * nx(i,1), ...
                      alpha(i,4), 0, -alpha(i,4) * nx(i,1)];
        A(2*i,:)   = [0, alpha(i,1), -alpha(i,1) * nx(i,2), ...
                      0, alpha(i,2), -alpha(i,2) * nx(i,2), ...
                      0, alpha(i,3), -alpha(i,3) * nx(i,2), ...
                      0, alpha(i,4), -alpha(i,4) * nx(i,2)];
    end

    % Solve for cc1 to cc4
    [~, ~, V] = svd(A, 'econ');
    cc1 = V(1:3,end);
    cc2 = V(4:6,end);
    cc3 = V(7:9,end);
    cc4 = V(10:12,end);
end

function Xc = calCameraPoints(hX, cc1, cc2, cc3, cc4, alpha)
    % Dehomogenize
    Xw = hX(:,1:3);

    Xc = zeros(size(Xw,1), 3);
    for i = 1:size(Xw,1)
        Xc(i,:) = alpha(i,1) * cc1 + alpha(i,2) * cc2 ...
            + alpha(i,3) * cc3 + alpha(i,4) * cc4;
    end

    varXw = var(Xw);
    varXc = var(Xc);
    beta = sqrt(sum(varXw)/sum(varXc));
    Xc = Xc * beta;
    Xc = Xc ./ sign(Xc(:,3));
end
```

**Problem 3**

Code Listing 5: Levenberg–Marquardt Algorithm

```
function [R, t, W, log] = LM_c(hx, hX, K, R, t)
    % Normalize x
    nx = (K\hx')';

    % Normalize nx by nx(:,3)
    nx = nx ./ nx(:,3);

    % Dehomogenize
    Xw = hX(:,1:3);

    % Covariance Matrix
    Z = zeros(2*size(hx,1));
    invK = inv(K);
    c = invK(1:2,1:2);
    for i = 1:size(hx,1)
        Z(2*i-1:2*i,2*i-1:2*i) = c * c';
    end

    % Initialization
    lambda = 0.001;
    ex = calEpsilon(nx, Xw, R, t);
    perr = 10000000;
    err = ex'*inv(Z)*ex;

    % Error Log
    log = err;

    while abs(perr-err) > 0.0001
        W = parameterize(R);

        % Angle Normalization
        if norm(W) > pi
            W = (1 - 2*pi/norm(W)*ceil((norm(W)-pi)/(2*pi))) * W;
        end
        R = deparameterize(W);

        % Create \partial(R, t) / \partial(W, t)
        I = eye(3);
        theta = norm(W);
        nW = W/norm(W);
        nWx = skewMatrix(nW);

        partial_vv = zeros(12,6);
        partial_vv(10:12,4:6) = eye(3);
        for i = 1:3
            dW = cos(theta)*nW(i,1)*nWx + sin(theta)*nW(i,1)*nWx*nWx
                                                    ...
                + sin(theta)/theta*skewMatrix(I(i,:)'-nW(i,1)*nW) ...
                + (1-cos(theta))/theta ...
                * (I(i,:)'*nW' + nW*I(i,:) - 2*nW(i,1)*(nW*nW'));
            partial_vv(1:9,i) = vector(dW);
        end

        % Calculate J with \partial(x, y) / \partial(R, t)
        J = zeros(2*size(hx,1),6);
        for i = 1:size(hx,1)
            w = R * Xw(i,:)' + t;
            partial_xp = 1/w(3,1) * ...
                [Xw(i,:), zeros(1,3), -w(1,1)/w(3,1)*Xw(i,:), ...
                 1, 0, -w(1,1)/w(3,1); ...
                 zeros(1,3), Xw(i,:), -w(2,1)/w(3,1)*Xw(i,:), ...
                 0, 1, -w(2,1)/w(3,1)];
            J(2*i-1:2*i,:) = partial_xp * partial_vv;
```

```
        end

        while true
            % Solve delta
            d = (J'*inv(Z)*J + lambda*eye(6)) \ (J'*inv(Z)*ex);
            nW = W + d(1:3,1);
            nt = t + d(4:6,1);

            % Angle Normalization
            if norm(nW) > pi
                nW = (1 - 2*pi/norm(nW) ...
                    * ceil((norm(nW)-pi)/(2*pi))) * nW;
            end
            nR = deparameterize(nW);

            nex = calEpsilon(nx, Xw, nR, nt);
            if nex'*inv(Z)*nex < ex'*inv(Z)*ex
                R = nR;
                t = nt;
                ex = nex;
                lambda = 0.1 * lambda;
                break;
            else
                lambda = 10 * lambda;
            end
        end

        % Update error
        perr = err;
        err = nex'*inv(Z)*nex;
        log = [log, err];
    end
    W = parameterize(R);
end

function ex = calEpsilon(nx, Xw, R, t)
    px = (R * Xw' + t)';
    px = px ./ px(:,3);
    ex = vector(nx(:,1:2)) - vector(px(:,1:2));
end

function W = parameterize(R)
    [~, ~, V] = svd(R - eye(3));
    a = V(:,end);
    b = [R(3,2)-R(2,3); R(1,3)-R(3,1); R(2,1)-R(1,2)];

    sin_theta = 0.5 * a' * b;
    cos_theta = 0.5 * (trace(R)-1);
    theta = atan2(sin_theta, cos_theta);
    %theta = acos(cos_theta);

    W = theta * a / norm(a);
end

function R = deparameterize(W)
    theta = norm(W);

    if theta < 1e-7
        assert(false);
    else
        sin_theta = sin(theta);
        cos_theta = cos(theta);
        nW = W / norm(W);
        nWx = skewMatrix(nW);
```

```
        R = cos_theta*eye(3) + sin_theta*nWx ...
            + (1-cos_theta)*(nW*nW');
    end
end

function Wx = skewMatrix(W)
    Wx = [0, -W(3,1), W(2,1); ...
          W(3,1), 0, -W(1,1); ...
          -W(2,1), W(1,1), 0];
end

function vx = vector(x)
    vx = x';
    vx = vx(:);
end
```

**Main Script**

Code Listing 6: Main Script

```
%% Preprocessing
% Read data
hx = dlmread('../dat/hw3_points2D.txt');
hX = dlmread('../dat/hw3_points3D.txt');

% Homogenize
hx(:,3) = ones(size(hx,1),1);
hX(:,4) = ones(size(hX,1),1);

% Calibration Matrix
K = [1545.0966799187809, 0, 639.5; ...
     0, 1545.0966799187809, 359.5; ...
     0, 0, 1];


%% Problem 1
% Remove Outliers
[R, t, bmap, MAX_TRIALS] = mSAC(hx, hX, K);
hx = hx(bmap,:);
hX = hX(bmap,:);

RMSE = calRMSE(hx, hX, K, R, t);
fprintf('\n\nProblem 1\n');
fprintf('Number of Inliers: %d\n', sum(bmap));
fprintf('Number of MaxTrials: %.10f\n', MAX_TRIALS);
fprintf('RMSE: %.10f\n', RMSE);


%% Problem 2
[R, t] = EPnP(hx, hX, K);

RMSE = calRMSE(hx, hX, K, R, t);
fprintf('\n\nProblem 2\n');
display(R);
display(t);
fprintf('RMSE: %.10f\n', RMSE);


%% Problem 3
[R, t, W, log] = LM_c(hx, hX, K, R, t);

RMSE = calRMSE(hx, hX, K, R, t);
fprintf('\n\nProblem 3\n');
display(R);
```

```
display(t);
display(W);
display(log);
fprintf('RMSE: %.10f\n', RMSE);
```

**Helper Function**

Code Listing 7: Calculate RMSE

```
function RMSE = calRMSE(hx, hX, K, R, t)
    px = (K * [R, t] * hX')';
    px = px ./ px(:,3);
    diff = hx(:,1:2) - px(:,1:2);
    RMSE = sqrt(mean(sum(diff.^2,2)));
end
```