CSE 252B - Homework 3

Lingting Ge A53212800

2/21/2017

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

1.1 Outlier rejection

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 $\tilde{\mathrm{X}}_i$, $\tilde{\mathrm{Y}}_i$, and $\tilde{\mathrm{Z}}_i$ 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 $\tilde{\mathrm{x}}_i$ and $\tilde{\mathrm{y}}_i$ inhomogeneous coordinates of a point). The corresponding 3D scene and 2D image points contain both inlier and outlier correspondences. For the inlier correspondences, the scene points have been randomly generated and projected to image points under a camera projection matrix (i.e., $\mathbf{x}_i = \mathbf{P}\mathbf{X}_i$), then noise has been added to the image point coordinates.

The camera calibration matrix was calculated for a 1280 * 720 sensor and 45° horizontal field of view lens. The resulting camera calibration matrix is given by

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

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

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 R and translation 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 $\hat{P} = [R|t]$) and the measured points in normalized coordinates (hint: the error tolerance is simpler to calculate in image coordinates using $P = K[R|t]$ than in normalized coordinates using $\hat{P} = [R|t]$).

In your report, describe any assumptions, including the probability p that at least one of the random samples does not contain any outliers (used to determine the number of attempts to find a consensus set), and the probability α that a given data point is an inlier and the variance σ^2 of the measurement error (both used to determine the distance threshold; hint: this problem has codimension 2). Additionally, show the resulting number of inliers and the number of attempts to find the consensus set.

Solution:

The number of inliers is 50 (specially the former 50 points in the give data), and the number of attempts to find the consensus set is 6. The random seed that I used is rand('seed', 0). (Due to random sample, the result may be different for each try, but the biggest number of inliers is 50. However, by using a fixed random seed, we can generate same random samples for each try.)

The parameters I used are as follows: probability $P = 0.99$, probability $\alpha = 0.95$, variance $\sigma^2 = 1$.

Here are the general steps that I used to solve this question is as follows:

- Calculate the points the in normalized coordinates using K and original points.
- For initialization, set max trials and minimum cost to be infinity. And number of trials to be 0;
- Begin the iteration of MASC method.
- Select three random points. Calculate the corresponding points using FinsterWalder method, and then using Umeyama paper to calculate the camera pose R and t. There may be several solutions of camera pose.
- For each solution, calculate the corresponding cost and model, according to the MSAC method.
- If the cost is less then current minimum cost, then update the minimum cost and model. And calculate the number inliers and update the max trials using parameter w, where $w = \frac{number of inliers}{total number of data points}$, and $max\_trials = \frac{log(1-P)}{log(1-w^s)}$, where $s = 3$ since we have three random points for each iteration.
- If current number of trials is bigger than the current max trials, then stop the iteration. Otherwise, go to step 4.
- After the iteration, we can get the inliers by compare the error of each point with the tolerance, using the model we get from the iteration.

1.2 Linear estimation

Estimate the normalized camera projection matrix $\hat{P}_{linear} = [R_{linear}|t_{linear}]$ from the resulting set of inlier correspondences using the linear estimation method (based on the EPnP method) described in lecture. Include the numerical values of the resulting R_{linear} and t_{linear} in your report with sufficient precision such that it can be evaluated (hint: use format longg in MATLAB prior to displaying your results).

Solution:

The result of my solution is as follows:

$$[R_{linear}|t_{linear}] = \begin{bmatrix} 0.277695359471619 & -0.69133711406441 & 0.667036942039211 & 5.51296444255469 \\ 0.660708403783331 & -0.366610968933959 & -0.655027329679751 & 7.47705965212295 \\ 0.697387763369857 & 0.622614963018395 & 0.354966076305317 & 175.916874394307 \end{bmatrix} \tag{2}$$

The steps for solving this problem are as follows:

- Calculate the mean and variance of 3D points.
- Calculate the control point in the world coordinate frame.
- Calculate α for each point in the parameterization of 3D point, then parameterize the 3D points.
- Formulate the matrix m using α and 2D points.
- Calculate the control points in the camera coordinate frame using matrix m. The control point is the null space of matrix m;
- Deparameterize 3D points in camera coordinate frame, by multiply α and control points matrix.
- Scale the 3D points in camera coordinate frame such that the total standard deviation of the scale corrected 3D points is equal to the total standard deviation of the 3D points in the world coordinate frame. Then we can get the points in camera coordinate frame.
- By using the points in camera coordinate frame and their corresponding points in world coordinate frame, calculate the camera pose based on Umeyama paper, which is similar with the part used in question (a).

Table 1: cost at each iteration

iteration	cost
0	83.5848697971415
1	83.5699630891798
2	83.5699624725947
3	83.5699624725558
4	83.5699624725571
5	83.5699624725599
6	83.5699624725574
7	83.5699624725587
8	83.5699624725574
9	83.5699624725595
10	83.5699624725574

1.3 Nonlinear estimation

Use R_{linear} and t_{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 $\hat{P} = [R|t]$. You must parameterize the cam- era rotation using the angle-axis representation ω (where $[\omega]x = lnR$) of a 3D rotation, which is a 3-vector.

In your report, show the initial cost (i.e., the 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 ω_{LM} and $R_{LM} = e^{[\omega_{LM}]x}$, and the camera translation t_{LM} in your report with sufficient precision such that it can be evaluated.

Solution:

The initial cost is 83.5848697971415. And the cost at the end of each successive iteration are show in the Table 1.

After convergence the camera rotation is as follows:

$$\omega_{LM} = \begin{bmatrix} 1.33700363490747 \\ -0.0319789480523163 \\ 1.41466759636439 \end{bmatrix} \tag{3}$$

And the camera projection matrix is as follows:

$$[R_{LM}|t_{LM}] = \begin{bmatrix} 0.277683332246175 & -0.691344681860574 & 0.667034105466701 & 5.51175076923206 \\ 0.660496916081481 & -0.366802845591519 & -0.655133189750617 & 7.45410176871747 \\ 0.697592854636223 & 0.622493536776942 & 0.35477599387701 & 175.916231607656 \end{bmatrix} \tag{4}$$

For this problem, the process of Levenberg algorithm (nonlinear estimation) is very clear according to the lecture note. And here is the general process:

- For initialization, we have camera calibration matrix K, initial normalized camera projection matrix P, which is from the result of question (b), and the inlier 2D points and their corresponding 3D points, which are from the result of question (a). And $\lambda = 0.001$.
- Calculate measurement vector $[\hat{x}_1, ..., \hat{x}_n]$, and calculate the projection $K * P * point3D$, and then calculate the initial error ϵ by take the difference between the projection and original 2D points. Then we can get the initial cost by calculating $\epsilon^\top * \Sigma * \epsilon$, where $\Sigma = I$.
- Then begin the iteration of Levenberg method, do the following until convergence.

- First, parameterize the normalized projection matrix P, then we can get the parameter vector $[w, t]^\top$, which is a 6-vector. Then using the parameter vector and the inlier points, we can calculate the Jacobian matrix J.
- Second, after get the Jacobian matrix J, we can solve the augmented normal equation to get δ.
- Third, add δ to the parameter vector, we can get a new vector. Deparameterize it and do the projection from 3D to 2D. Then calculate the current error and current cost.
- Forth, if the current cost is less than the cost, then update the error, cost, and model (parameter vector), and go to first step with setting $\lambda = 0.1\lambda$. Otherwise, go the second step by setting $\lambda = 10\lambda$.
- Finally, after convergence, we can get the result.

Appendix: Source code

```
% This is main.m %
clear all;
K = [1545.0966799187809, 0, 639.5; ...
     0, 1545.0966799187809, 359.5; ...
     0, 0, 1];

% read the data
point2DOrig = readPoint('hw3_points2D.txt', 2);
point3DOrig = readPoint('hw3_points3D.txt', 3);

% transfer to homogeneous
point2DHomo = [point2DOrig, ones(60, 1)];
point3DHomo = [point3DOrig, ones(60, 1)];

% calculate the point in normalized coordinates
point2DNorm = (K \ (point2DHomo'))';



% question (a) %
[inlierIndex, trials] = MSAC(K, point2DOrig, point3DOrig);
disp('number of trials:');
disp(trials);
num_inliers = sum(inlierIndex);
disp('number of inliers');
disp(num_inliers);



% question (b) %

% pick up inliers
inlier2DNorm = point2DNorm(inlierIndex, :);
inlier3DInhomo = point3DOrig(inlierIndex, :);

% linear estimation
[R, t, flag] = linearEst(inlier2DNorm, inlier3DInhomo);
P_linear = [R, t];

format longg;
disp('camera projection matrix:');
disp(P_linear);



% question (c) %

% pick up inliers
inlier2DOrig = point2DOrig(inlierIndex, :)';
inlier3DOrig = point3DOrig(inlierIndex, :)';
inlier2DHomo = point2DHomo(inlierIndex, :)';
inlier3DHomo = point3DHomo(inlierIndex, :)';
inlier2DNorm = point2DNorm(inlierIndex, :)';
inlier2DNorm = inlier2DNorm(1 : 2, :);

[P, w] = levenberg(P_linear, K, inlier2DOrig, inlier3DHomo);
disp('final result of w:');
disp(w);
disp('final matrix:');
disp(P);


% read the data
```

```
function point = readPoint(fileName, dim)
    file = fopen(fileName);
    if dim == 2
        point = textscan(file, '%f %f');
    else
        point = textscan(file, '%f %f %f');
    end
    fclose(file);
    point = cell2mat(point);
end
```

```
% This is MSAC.m %
% MSAC method
function [inlierIndex, trials] = MSAC(K, point2DOrig, point3DOrig)
    % transfer to homogeneous
    point2DHomo = [point2DOrig, ones(60, 1)];
    point3DHomo = [point3DOrig, ones(60, 1)];

    % calculate the point in normalized coordinates
    point2DNorm = (K \ (point2DHomo'))';

    % MSAC algorithm
    consensus_min_cost = Inf;
    trials = 0;
    max_trials = Inf;
    threshold = 0;
    prob = 0.99;
    alpha = 0.95;
    variance = 1;
    codimension = 2;
    tolerance = chi2inv(alpha, codimension);

    rand('seed', 0);

    % begin iteration
    while (trials < max_trials && consensus_min_cost > threshold)
        % generate three random samples
        sampleIndex = randperm(60, 3);
        [R, t, cnt] = finsterWalder(point2DNorm, point3DHomo, sampleIndex);
        if cnt == 0
            continue;
        end
        % computation for each possible solution
        for i = 1 : size(R, 3)
            x_cam = K * [R(:, :, i), t(:, :, i)] * point3DHomo';
            x_camHomo = x_cam ./ x_cam(3, :);
            x_cam2D = x_camHomo;
            x_cam2DInhomo = x_cam2D(1 : 2, :);
            error = x_cam2DInhomo - point2DOrig';
            cost = reshape(error, [2*60, 1])' * eye(2*60) * reshape(error, [2*60, 1]);
            if cost < consensus_min_cost
                consensus_min_cost = cost;
                model_R = R(:, :, i);
                model_t = t(:, :, i);
            end
            % count number of inliers
            dist_error = zeros(1, 60);
            for k = 1 : 60
                dist_error(k) = norm(error(:, k));
            end
            num_inliers = sum(dist_error <= tolerance);
            w = num_inliers / 60;
            max_trials = log(1 - prob) / log(1 - w^3);
        end
        trials = trials + 1;
```

```
    end

    % count inliers
    x_cam = K * [model_R, model_t] * point3DHomo';
    x_camHomo = x_cam ./ x_cam(3, :);
    x_camInhomo = x_camHomo(1 : 2, :);
    dist_error = zeros(1, 60);
    for k = 1 : 60
        dist_error(k) = norm(error(:, k));
    end
    inlierIndex = (dist_error <= tolerance);
end
```

```
% This is finsterWalder.m %
% 3 poing algorithm of Finsterwalder
function [R, t, cnt] = finsterWalder(point2D, point3D, sampleIndex)
    p1 = point3D(sampleIndex(1), :);
    p2 = point3D(sampleIndex(2), :);
    p3 = point3D(sampleIndex(3), :);

    q1 = point2D(sampleIndex(1), :);
    q2 = point2D(sampleIndex(2), :);
    q3 = point2D(sampleIndex(3), :);

    j1 = q1 / norm(q1);
    j2 = q2 / norm(q2);
    j3 = q3 / norm(q3);

    cos_alpha = dot(j2,j3);
    cos_beta = dot(j1,j3);
    cos_gamma = dot(j1,j2);

    a = norm(p2 - p3);
    b = norm(p1 - p3);
    c = norm(p1 - p2);

    G = c^2 * (c^2 * (1 - cos_beta^2) - b^2 * (1 - cos_gamma^2));
    H = b^2 * (b^2 - a^2) * (1 - cos_gamma^2) + c^2 * (c^2 + 2 * a^2) * ...
        (1 - cos_beta^2) + 2 * b^2 * c^2 * (-1 + cos_alpha * cos_beta * cos_gamma);
    I = b^2 * (b^2 - c^2) * (1 - cos_alpha^2) + a^2 * (a^2 + 2 * c^2) * ...
        (1 - cos_beta^2) + 2 * a^2 * b^2 * (-1 + cos_alpha * cos_beta * cos_gamma);
    J = a^2 * (a^2 * (1 - cos_beta^2) - b^2 * (1-cos_alpha^2));

    % find a real root
    lambda = roots([G, H, I, J]);
    for i = 1 : length(lambda)
        if isreal(lambda(i))
            realLambda = lambda(i);
            break;
        end
    end

    A = 1 + realLambda;
    B = -cos_alpha;
    C = (b^2 - a^2) / b^2 - realLambda * (c^2 / b^2);
    D = -realLambda * cos_gamma;
    E = (a^2 / b^2 + realLambda * c^2 / b^2) * cos_beta;
    F = -a^2 / b^2 + realLambda * ((b^2 - c^2) / b^2);

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

    m = [-B + p; -B - p] / C;
    n = [-E + q; -E - q] / C;

```

```
    pos = 1;
    u = [];
    v = [];
    % find all real roots for u
    for i = 1 : 2
        A1 = b^2 - m(i)^2 * c^2;
        B1 = 2 * (c^2 * (cos_beta-n(i)) * m(i) - b^2 * cos_gamma);
        C1 = -c^2 * n(i)^2 + 2 * c^2 * n(i) * cos_beta + b^2 - c^2;
        possi_u = roots([A1 B1 C1]);
        for h = 1:length(possi_u)
            if isreal(possi_u(h))
                u(pos) = possi_u(h);
                v(pos) = u(pos) .* m(i) + n(i);
                pos = pos + 1;
            end
        end
    end

    cnt = length(u);
    if cnt == 0
        R = [];
        t = [];
        return
    end

    s1 = zeros(1,length(u));
    s2 = zeros(1,length(u));
    s3 = zeros(1,length(u));
    for i = 1 : length(u)
        s1(i) = sqrt(a^2 / (u(i)^2 + v(i)^2 - 2 * u(i) * v(i) * cos_alpha));
        s2(i) = u(i) * s1(i);
        s3(i) = v(i) * s1(i);
    end

    p_1 = j1' * s1;
    p_2 = j2' * s2;
    p_3 = j3' * s3;

    % compute the camera pose R and t
    R = zeros(3, 3, size(p_1,2));
    t = zeros(3, 1, size(p_1,2));
    threePoint3D = [p1; p2; p3];
    for i = 1 :size(p_1, 2)
        [R(:, :, i), t(:, :, i)] = calCameraPose(threePoint3D(:, 1 : 3), [p_1(:, i), p_2(:,
    i), p_3(:, i)]');
    end
end
```

```
% This is calCameraPose.m %
% calculate camera pose R and t based on Umeyama paper
% X is point in world coordinates
% Y is point in camera coodinates
function [R, t, flag] = calCameraPose(X, Y)
    num = size(X, 1);
    mean_X = mean(X, 1);
    mean_Y = mean(Y, 1);
    meanX = repmat(mean_X, [num, 1]);
    meanY = repmat(mean_Y, [num, 1]);
    varX = sum(sum((X - meanX) .* (X - meanX))) / num;
    varY = sum(sum((Y - meanY) .* (Y - meanY))) / num;
    covariance = (Y - meanY)' * (X - meanX) / num;

    % if rank is less than, then discard this answer
    if rank(covariance) < size(X, 2) - 1
        flag = false;
```

```
    else
        flag = true;
    end

    [U, D, V] = svd(covariance);
    S = eye(size(D, 1));
    if abs(det(U) * det(V) - 1) >= 0.01
        S(size(D, 1), size(D, 1)) = -1;
    end
    R = U * S * V';
    c = 1/ varX * trace(D * S);
    t = mean_Y' - c * R * mean_X';
end
```

```
% This is linearEst.m %
% compute R and t using linear estimation
function [R, t, flag] = linearEst(point2D, point3D)
    format longg
    num = size(point3D, 1);

    % calculate mean and covariance of 3D points
    mean3D = mean(point3D);
    covar = cov(point3D);
    [~, D, V] = svd(covar);

    % calculate control point in world coordinate
    varWorld = D(1, 1) + D(2, 2) + D(3, 3);
    s = sqrt(varWorld / 3);

    % 3D points parameterization
    b = point3D - repmat(mean3D, num, 1);
    alpha2to4 = b * (V / s);
    alpha1 = 1.0 - alpha2to4(:, 1) - alpha2to4(:, 2) - alpha2to4(:, 3);
    alpha = [alpha1, alpha2to4(:, 1), alpha2to4(:, 2), alpha2to4(:, 3)];

    % calculage control point in camera coordinate frame
    m = zeros(2 * num, 12);
    for i = 1 : num
        m(2 * i - 1 : 2 * i, :) = ...
            [alpha(i, 1), 0, -alpha(i, 1) * point2D(i, 1), ...
             alpha(i, 2), 0, -alpha(i, 2) * point2D(i, 1), ...
             alpha(i, 3), 0, -alpha(i, 3) * point2D(i, 1), ...
             alpha(i, 4), 0, -alpha(i, 4) * point2D(i, 1); ...
             0, alpha(i, 1), -alpha(i, 1) * point2D(i, 2), ...
             0, alpha(i, 2), -alpha(i, 2) * point2D(i, 2), ...
             0, alpha(i, 3), -alpha(i, 3) * point2D(i, 2), ...
             0, alpha(i, 4), -alpha(i, 4) * point2D(i, 2)];
    end
    [~, ~, V] = svd(m);
    control = V(:, size(V, 2));
    C_cam = [control(1 : 3)'; control(4 : 6)'; control(7 : 9)'; control(10 : 12)'];

    % deparameterize 3D point in camera coordinate frame
    X_cam = alpha * C_cam;

    % scale 3D points in camera coordinate frame
    varCam = var(X_cam(:, 1)) + var(X_cam(:, 2)) + var(X_cam(:, 3));
    beta = sqrt(varWorld / varCam);
    X_cam = beta * X_cam;
    for i = 1 : num
        if X_cam(i, 3) < 0
            X_cam(i, :) = -X_cam(i, :);
        end
    end

```

```
    % transfermation from world coordinate frame to camera coorinate frame
    % calculate R and t based on Umeyama paper
    [R, t, flag] = calCameraPose(point3D, X_cam);
end
```

```
% This is levenberg.m %
% levenberg iterative method
function [P, w] = levenberg(P_linear, K, inlier2DOrig, inlier3DHomo)
    num_inlier = size(inlier2DOrig, 2);
    % initial estimate
    point2DEstInhomo = projection(K, P_linear, inlier3DHomo);
    error = calMeasureVector((inlier2DOrig - point2DEstInhomo), num_inlier);
    %error = calMeasureVector((inlier2DNorm - point2DEstInhomo), num_inlier);
    covar = eye(2 * num_inlier);
    cost = error' * covar * error;
    disp('cost for each iteration:')
    disp(cost);

    d = diag(inv(K));
    K_diag = diag(repmat(d(1 : 2), [num_inlier, 1]));
    param_num = 6;
    lambda = 0.001;

    paramP = parameterize(P_linear);

    % begin iteration
    for i = 1 : 10
        % compute Jocabian
        J = calJocabian(inlier3DHomo, paramP);
        % compute delta
        delta = (J' * inv(K_diag * covar * K_diag') * J + lambda * eye(param_num)) \...
                (J' * inv(K_diag * covar) * error);
        % delta = (J' * covar * J + lambda * eye(param_num)) \ (J' * covar * error);
        % update P
        paramPUpdate = paramP + delta;
        deparamPUpdate = deparameterize(paramPUpdate);
        % compute error and cost
        proj2DUpdateInhomo = projection(K, deparamPUpdate, inlier3DHomo);
        errorUpdate = calMeasureVector((inlier2DOrig - proj2DUpdateInhomo), num_inlier);
        % errorUpdate = calMeasureVector((inlier2DNorm - proj2DUpdateInhomo), num_inlier);
        costUpdate = errorUpdate' * covar * errorUpdate;
        disp(costUpdate);
        % make decsion
        if (costUpdate < cost)
            paramP = paramPUpdate;
            deparamP = deparamPUpdate;
            error = errorUpdate;
            cost = costUpdate;
            lambda = 0.1 * lambda;
        else
            lambda = 10.0 * lambda;
        end
    end
    P = deparamP;
    w = paramP(1 : 3);
end

% compute measurement vectort
function measureVector = calMeasureVector(point, num)
    measureVector = reshape(point, [2 * num, 1]);
end

% paramterization
function paramP = parameterize(P)
    R = P(1 : 3, 1 : 3);
```

```
    t = P(1 : 3, 4);
    [~, D, V] = svd(R - eye(3));
    v = V(:, 3);
    v_head = [R(3, 2) - R(2, 3);...
              R(1, 3) - R(3, 1);...
              R(2, 1) - R(1, 2)];

    sin_theta = v' * v_head / 2.0;
    cos_theta = (trace(R) - 1) / 2.0;
    theta = atan2(sin_theta, cos_theta);
    if theta < 0
        theta = theta + 2 * pi;
    end
    w = theta * v / norm(v);
    if theta > pi
        w = w * (1 - 2 * pi / theta * ceil((theta - pi) / (2 * pi)));
    end
    paramP = [w', t']';
end

function deparamP = deparameterize(P)
    w = P(1 : 3);
    t = P(4 : 6);
    theta = norm(w);
    w_x = formMat(w);
    R = cos(theta) * eye(3) + sinc(theta) * w_x + ((1 - cos(theta)) / (theta^2)) * w * w';
    deparamP = [R, t];
end

% sinc(x)
function res = sinc(x)
    if x == 0
        res = 1.0;
    else
        res = (sin(x)) / x;
    end
end

% compute Jocabian matrix
function jocabian = calJocabian(x3D, paramP)
    for i = 1 : size(x3D, 2)
        jocabian(2 * i - 1 : 2 * i, :) = calJocabianAi(paramP, x3D(1 : 3, i));
    end
end

% compute Ai in the Jocabian matrix
function Ai = calJocabianAi(paramP, X)
    w = paramP(1:3);
    theta = norm(w);
    deparamP = deparameterize(paramP);
    xHomo = deparamP * [X; 1];

    part1 = [1/xHomo(3) 0 -xHomo(1)/(xHomo(3)^2); 0 1/xHomo(3) -xHomo(2)/(xHomo(3)^2)];
    part2 = [1, 1, 1, 1, 0, 0;...
             1, 1, 1, 0, 1, 0;...
             1, 1, 1, 0, 0, 1];
    if theta < eps
        part2(:, 1 : 3) = formMat(-X);
    else
        part2(:, 1 : 3) = sinc(theta) * formMat(-X) + cross(w, X) * (cos(theta) / theta -
    sin(theta) / (theta^2))...
                          * (w / theta)' + cross(w, cross(w, X)) * (sin(theta) / (theta^2) -
     2 * ...
                          (1 - cos(theta)) / (theta^3)) * (w / theta)' + (1 -cos(theta)) / (
    theta^2)...
```

```
                            * eye(3) * (formMat(w) * formMat(-X) + formMat((-cross(w, X))));
    end
    Ai = part1 * part2;
end

% form the matrix
function mat = formMat(X)
    mat = [0, -X(3), X(2);...
           X(3), 0, -X(1);...
           -X(2), X(1), 0];
end

% projection
function X_est = projection(K, deparamP, point)
    X2DEst = K * deparamP * point;
    X2DEstHomo = X2DEst ./ X2DEst(3, :);
    X2DEstInhomo = X2DEstHomo(1 : 2, :);
    X_est = X2DEstInhomo;
end
```