CSE 252B - Homework 5

Lingting Ge A53212800

3/19/2017

1 Point on line closet to the origin

Given a line $\mathbf{l} = (a, b, c)^T$, show that the point on $\mathbf{l}$ that is closest to the origin is the point $x = (-ac, -bc, a^2 + b^2)^T$. (Hint: this calculation is needed in the two-view optimal triangulation method used below.)

Solution

Since we are going to find the point on $\mathbf{l}$ that is closet to the origin, then if there is a line $\mathbf{l}_{orth}$, which is orthogonal to $\mathbf{l}$ and through the origin, then the intersection of two lines is the point that we need.

Suppose a line through the origin is $(m, n, 0)^T$, and since this line is orthogonal to the line $\mathbf{l}$, then we have the following equation:

$$am + bn = 0 \tag{1}$$

Further, the intersection of the two lines is as follows:

$$\begin{aligned}
point &= (a, b, c)^T \times (m, n, 0)^T && (2)\\
&= (-nc, mc, an - bm)^T && (3)\\
&= (-nbc, mbc, nba - mb^2)^T \quad uptoscale && (4)\\
&= (nbc, -mbc, -nba + mb^2)^T \quad uptoscale && (5)\\
&= (-mac, -mbc, ma^2 + mb^2)^T \quad substitue(1) && (6)\\
&= (-ac, -bc, a^2 + b^2)^T \quad uptoscale && (7)
\end{aligned}$$

Therefore, the point on $\mathbf{l}$ that is closest to the origin is $(-ac, -bc, a^2 + b^2)^T$.

2 Feature detection

Download input data from the course website. The file $IMG\_5030.JPG$ contains image1 and the file $IMG\_5031.JPG$ contains image 2. In your report, include a figure containing the pair of input images.

For each input image, calculate an image where each pixel value is the minor eigenvalue of the gradient matrix

$$\mathrm{N} = \begin{bmatrix} \Sigma_w I_x^2 & \Sigma_w I_x I_y \\ \Sigma_w I_x I_y & \Sigma_w I_y^2 \end{bmatrix} \tag{8}$$

where w is the window about the pixel, and I_x and I_y are the gradient images in the x and y direction, respectively. Calculate the gradient images using the five point central difference operator. Set resulting values that are below a specified threshold value to zero (hint: calculating the mean instead of the sum in N allows for adjusting the size of the window without changing the threshold value). Apply an operation that suppresses (sets to 0) local (i.e., about a window) nonmaximum pixel values in the minor eigenvalue image. Vary these parameters such that around 1350-1400 features are detected in each image. For resulting nonzero pixel values, determine the subpixel feature coordinate using the Forstner corner point operator.

In your report, state the size of the feature detection window (i.e., the size of the window used to calculate the elements in the gradient matrix N), the minor eigenvalue threshold value, the size of the local nonmaximum suppression window, and the resulting number of features detected in each image. Additionally, include a figure containing the pair of images, where the detected features (after local nonmaximum suppression) in each of the images are indicated by a square about the feature, where the size of the square is the size of the detection window.

Solution:

The parameters I used are as follows:

Size of the feature detection window is 7 * 7;
The minor eigenvalue threshold value is 1000;
The size of the local non maximum suppression window is 7 * 7.

The resulting numbers of features detected are 1369 and 1391 for the first and second image respectively.

Here is the procedure that I solve this problem:

- Read the image and transform the image into gray scale image.
- Calculate gradient images I_x and I_y respectively.
- Select a window size and calculate gradient matrix.
- Calculate the minor eigenvalue image.
- Compute the non maxima suppression.
- Using Forstner corner point method to find the coordinates of the corners.

The original figures are shown in Figure 1.

The detected features are shown in Figure 2.

Figure 1: Original figure

Figure 2: Corner detection

3 Feature matching

Determine the set of one-to-one putative feature correspondences by performing a brute-force search for the greatest correlation coefficient value (in the range [-1, 1]) between the detected features in image 1 and the detected features in image 2. Only allow matches that are above a specified correlation coefficient threshold value (note that calculating the correlation coefficient allows for adjusting the size of the matching window without changing the threshold value). Further, only allow matches that are above a specified distance ratio threshold value, where distance is measured to the next best match for a given feature. Vary these parameters such that around 300 putative feature correspondences are established. Optional: constrain the search to coordinates in image 2 that are within a proximity of the detected feature coordinates in image 1.

In your report, state the size of the proximity window (if used), the correlation coefficient threshold value, the distance ratio threshold value, and the resulting number of putative feature correspondences (i.e., matched features). Additionally, include a figure containing the pair of images, where the matched features in each of the images are indicated by a square about the feature, where the size of the square is the size of the matching window, and a line segment is drawn from the feature to the coordinates of the corresponding feature in the other image (see Fig. 4.9(e) in the Hartley & Zisserman book as an example).

Solution:

The parameters I used are as follows:

The correlation coefficient threshold value is 0.5;
The distance ratio threshold value is 0.78;
The window size is 27 * 27.
I didn't use the proximity window.

The resulting number of putative feature correspondences is 291.

Here is the process that I used to solve this problem:

- Fetch the corners of two features from the result of question (a).
- Take out all windows corresponding to their corners.
- Calculate correlation coefficients of window pair between all window in two figures.
- Do the one-to-one matching as follows:
  - Find the indices of element with maximum value.
  - If the maximum value is larger than the similarity threshold, do the follows:
    - Store the best match value.
    - Set this element value to -1.
    - Find the next best match value as the following equation:

$$nextbestmatch = max(nextbestmatchvalueinrow, nextbestmatchvalueincolumn) \tag{9}$$

    - If the $(1 - bestmatchvalue) < (1 - nextbestmatchvalue) * distanceratiothreshold$, then store the feature match. Otherwise, this match is not unique enough.
    - Set the row and column to be -1.
- Otherwise, stop the iteration.

The final feature matching images are shown in Figure 3:

Figure 3: Feature mapping

4 Outlier rejection

The resulting set of putative point correspondences should contain both inlier and outlier correspondences (i.e., false matches). 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, you must use the 7-point algorithm (as described in lecture) to estimate the fundamental matrix, resulting in 1 to 3 solutions. Calculate the (squared) Sampson error as a first order approximation to the geometric error.

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 1). State the resulting number of inliers and the number of attempts to find the consensus set.

Additionally, include a figure containing the pair of images, where the inlier features in each of the images are indicated by a square about the feature, where the size of the square is the size of the matching window, and a line segment is drawn from the feature to the coordinates of the corresponding feature in the other image (see Fig. 4.9(g) in the Hartley & Zisserman book as an example).

Solution:

The parameters I used are as follows:

Probability p = 0.99;
Probability $\alpha = 0.95$;
Variance σ^2 assumed to be 1;

The number of inliers I find is 221;
The number of trials is 58.

The random seed I used is rand('seed', 2);

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

- 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 seven random points for two figures respectively.
- Do the normalization for the selected seven points.
- Compute the fundamental matrix using seven-point method as follows:
  - First, compute part A, where $A_i = kron(x_i', x_i)$, where x_i the point in the first image, and x_i' is the corresponding point in the second image.
  - Second, do the SVD operation for A, then we can get the part a, and b as the last two column of V. ($[U, D, V] = SVD(A)$.)
  - Third, reshape a, and b we can get F_1 and F_2, then compute F as $F = \alpha * F_1 + F_2$.
  - Then, solve the equation for α, which makes $det(F) = 0$.
  - At last, select the solution F which has the least cost.
- Calculate the corresponding cost and model, the error is computed as the squared Sampson error.
- If the cost is less than current minimum cost, do the follows:
  - Update the minimum cost and model.

Figure 4: Feature mapping using inliers

  - Calculate the number inliers and update the max trials using parameter w, where $w = \frac{numberofinliers}{totalnumberofdatapoints}$, and $max\_trials = \frac{log(1-P)}{log(1-w^s)}$, where $s = 7$ 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, continue.
- 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.

The matching figures are shown in Figure 4.

5 Linear estimation

Estimate the fundamental matrix F_{DLT} from the resulting set of inlier correspondences using the direct linear transformation (DLT) algorithm (with data normalization). Include the numerical values of the resulting F_{DLT}, scaled such that $\| F_{DLT} \|_{Fro} = 1$, 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 fundamental matrix F_{DLT} from DLT algorithm is as follows:

$$F_{DLT} = \begin{bmatrix} -3.13339047052432e-07 & 7.42204505013255e-07 & 0.010901497400064 \\ -1.72399491328539e-06 & -2.23343906012598e-08 & 0.000826782310100526 \\ -0.010009718518411 & -0.000532285326272364 & -0.999889991943784 \end{bmatrix} \tag{10}$$

Here is the general process that I used to solve this problem:

- Do the data normalization for the points in both figures.
- Then we can get the normalized data and matrix T_1 and T_2 for both figures respectively.
- Form the big matrix A using kron operations for each pair of points in both figures, where $A_i = kron(x_i', x_i)$, where x_i the point in the first image, and x_i' is the corresponding point in the second image.
- Compute the right null space of the matrix A, using SVD operation.
- Reshape the right null space to get a temp fundamental matrix F_{tmp}.
- Then, $[U, D, V] = SVD(F_{tmp})$. Set $S(3,3) = 0$.
- Then we can get $F_{norm} = U * D * V^T$.
- Calculate the matrix F_{DLT}, using the following equation:

$$F_{DLT} = T_2^T * F_{norm} * T_1. \tag{11}$$

- Scale the matrix F_{DLT} so that its fro norm is equal to 1.

6 Nonlinear estimation

As described in lecture, parameterize the fundamental matrix as $(w_u^T, w_v^T, \sigma^T)^T$, where $\parallel \sigma \parallel = 1$, and calculate the camera projection matrices $P = [I|0]$ and $P' = [M|e']$, where $M = UZdiag(\sigma_1, \sigma_2, (\sigma_1 + \sigma_2)/2)V^T$, $U = exp([w_u]_x) = [u_1|u_2|u_3]$, $\sigma = (\sigma_1, \sigma_2)^T$, $V = exp([w_v]_x)$,

$$Z = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

and $e' = -u_3$. Use F_{DLT} and the triangulated 3D points as an initial estimate to an iterative estimation method, specifically the sparse Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the fundamental matrix $F = exp([w_u]_x)diag(\sigma^T, 0)exp([w_v]_x)^T$ that minimizes the reprojection error. The initial estimate of the 3D points must be determined using the two-view optimal triangulation method described in lecture (algorithm 12.1 in the Hartley & Zisserman book, but use the ray-plane intersection method for the final step instead of the homogeneous method). Additionally, you must parameterize the homogeneous 3D scene points that are being adjusted using the parameterization of homogeneous vectors (see section A6.9.2 (page 624) of the textbook, and the corrections and errata).

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 fundamental matrix F_{LM}, scaled such that $\parallel F_{LM} \parallel_{Fro} = 1$, in your report with sufficient precision such that it can be evaluated.

Solution:

The initial cost is 66.3661689736039.

And the cost of each successive iteration are as follows:

66.3661689736039
66.361844084361
66.3191002246801
65.9391907875752
64.9338298444916
64.8991731315673
64.8991731315673
64.8991731315645
64.899173131562

The number of iterations that used to be convergence is 8.

The final estimate of the matrix is as follows:

$$F_{LM} = \begin{bmatrix} -3.07388549276665e-07 & 7.42483435994713e-07 & 0.0109014978480142 \\ -1.72344521766337e-06 & -2.23086432845289e-08 & 0.000826782950387801 \\ -0.010009718645915 & -0.000532285672416024 & -0.999889991936913 \end{bmatrix} \tag{13}$$

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 initial matrix F_{DLT}, which is from the result of last question, and the inlier points for both images, which are from the result of question of outlier rejection.
- Set the initial value $\lambda = 0.001$.
- Compute the 3D points using two-view optimal triangulation method as follows:

- For each pair of inliers, compute transformations T follows, and similarly for T':

$$T = \begin{bmatrix} w & 0 & -x \\ 0 & w & -y \\ 0 & 0 & w \end{bmatrix} \tag{14}$$

- Compute F_s as $F_s = T'^{-T} * F * T^{-1}$
- Calculate epipoles e and e' of F_s, and scale them.
- Form the rotation matrices R as follows, and similarly for R'.

$$R = \begin{bmatrix} e_1 & e_2 & 0 \\ -e_2 & e_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

- Then, form the polynomial $g(t)$ as follows and solve for t.

$$g(t) = t((at+b)^2 + f'^2(ct+d)^2)^2 - (ad-bc)(1+f^2t^2)^2(at+b)(ct+d) \tag{16}$$

- For each real part of each root t, compute the cost function as follows, and select the root t with the smallest cost.

$$s(t) = 1/(1+f^2+t^2) + (ct+d)^2/((at+b)^2 + (f'^2(ct+d)^2)) \tag{17}$$

- Determin the pints as the closest points on the corresponding lines.
- Correct points mapped back to original coordinates.
- Compute the orthogonal line which also go through the points.
- Back project the line to the plane.
- At last, compute the 3D scene points.

* Calculate initial measurement vector $[\hat{w_u}^T, \hat{w_v}^T, \hat{s}, \hat{X}_1^T, ..., \hat{X}_n^T]$, where $\hat{w_u}^T$, $\hat{w_v}^T$ and $\hat{s}$ are from the parameterization of fundamental matrix F. And $\hat{X}_i$ are the parameterization of the 3D points.
* Calculate the projections using following equations, where x_{scene} is the 3D scene points, $P = [I|0]$, and $P' = [m|e']$ is computed from F, which is the initial fundamental matrix F_{DLT}.

$$proj1 = P * x_{scene} \tag{18}$$

$$proj2 = P' * x_{scene} \tag{19}$$

* Compute the measurement vector $[x_1^T, x_2^t, ..., x_n^T, x\prime_1^T, x\prime_2^T, ..., x\prime_n^T]$.
* Compute the initial error vector and further compute the initial cost.
* Begin the iteration of Levenberg method, do the following until convergence.
  - First, compute the Jacobian matrix using parameter vector, measurement vector, inlier points and 3D scene points. And the Jacobian matrix is a sparse matrix, which contains three parts: A, $B1$ and $B2$.

- Second, using the three parts of Jacobian matrix to compute the normal equations matrix, which also contains three parts: U, V, and W. These three part can be calculated as follows:

$$U = \Sigma_i A_i^T * inv(\Sigma_{x'_i}) * A_i \tag{20}$$

$$V_i = B1_i^T * inv(\Sigma_{x_i}) * B1_i + B2_i^T * inv(\Sigma_{x'_i}) * B2_i \tag{21}$$

$$W_i = A_i^T * inv(\Sigma_{x'_i}) * B2_i \tag{22}$$

- Third, compute the normal equations vector, which is in the form of $(\epsilon_a^T, \epsilon_{b_1}^T, \epsilon_{b_2}^T, ..., \epsilon_{b_n}^T)^T$.
- Forth, compute the augmented normal equations, solve the equations and we can get the increment δ. δ can be considered as two parts, one is for the parameterization of the fundamental matrix F, and the other part is for the parameterization of the 3D scene points.
- Then, add δ to the parameter vector, we can get a new parameter vector.
- Deparameterize the two parts of the new parameter vector, then we can get the new fundamental matrix and the new 3D scene points.
- Recompute the new error and cost, using P and the new P' from the new fundamental matrix and the new 3D scene points.
- If the current cost is less than the current minimum cost, then update the error, cost and model, and go to the first step, with setting $\lambda = 0.1\lambda$.
- Otherwise, go to the forth step by setting $\lambda = 10\lambda$.

- Finally, after convergence, we can get the result.

Figure 5: Mapping to lines

7 Point to line mapping

Qualitatively determine the accuracy of F_{LM} by mapping points in image 1 to epipolar lines in image 2. Choose three distinct points $x_{\{1,2,3\}}$ distributed in image 1 that are not in the set of inlier correspondences and map them to epipolar lines $l'_{\{1,2,3\}} = F_{LM}x_{\{1,2,3\}}$ in the second image under the fundamental matrix F_{LM}.

In your report, include a figure containing the pair of images, where the three points in image 1 are indicated by a square (or circle) about the feature and the corresponding epipolar lines are drawn in image 2. Comment on the qualitative accuracy of the mapping (hint: each line l'_i should pass through the point x'_i in image 2 that corresponds to the point x_i in image 1).

Solution:

The figure is shown in figure 5. In the first image, three points which are not inliers are presented by three rectangles. In the second image, the corresponding epipolar lines are drawn. From the second image, we can see that each epipolar line pass very accurately through the corresponding points in the image 1. Therefore, the calculation of the F_{LM} is correct.

Appendix: Source code

```
% main.m %

clc, clear, close all;
format longg;

% input image
image1 = 'IMG_5030.JPG';
image2 = 'IMG_5031.JPG';

% show the original picture
figure(1)
subplot(1, 2, 1);
imshow(imread(image1));
subplot(1, 2, 2)
imshow(imread(image2));



%---------- Question (a) ----------%
% feature detection %

% set parameter
w_size1 = 7;
threshold = 1000;
w_size2 = 7;
w_sizeb = 27;
simThresh = 0.5;
ratioThresh = 0.78;

% corner detection
[row1, col1] = featureDetection(image1, w_size1, threshold, w_size2);
[row2, col2] = featureDetection(image2, w_size1, threshold, w_size2);

% count # features
x1 = row1(:);
x2 = row2(:);
y1 = col1(:);
y2 = col2(:);
disp('Question (a):');
disp('number of features in figure 1:');
disp(size(x1, 1));
disp('number of features in figure 2:');
disp(size(x2, 1));

% show the feature image
figure(2)
subplot(1, 2, 1);
imshow(imread(image1));
hold on;
scatter(y1, x1, w_size1 * w_size1, 's');
subplot(1, 2, 2)
imshow(imread(image2));
hold on;
scatter(y2, x2, w_size1 * w_size1, 's');



%---------- Question (b) ----------%
% feature matching %

% feature matching
match = featureMatching(image1, image2, row1, col1, row2, col2, w_sizeb, simThresh,
    ratioThresh);
```

```
disp('Question (b):');
disp('number of matchings:');
disp(sum(sum(match)));

% show the feature matching image
figure(3)
subplot(1, 2, 1);
imshow(imread(image1));
hold on;
length1 = size(row1);
length2 = size(row2);
for i = 1 : length1(1)
    for j = 1 : length2(1)
        if match(i, j) == 1
            plot([col1(i), col2(j)], [row1(i), row2(j)], '-');
            scatter(col1(i), row1(i), w_size1 * w_size1, 's');
        end
    end
end
subplot(1, 2, 2);
imshow(imread(image2));
hold on;
length1 = size(row1);
length2 = size(row2);
for i = 1 : length1(1)
    for j = 1 : length2(1)
        if match(i, j) == 1
            plot([col1(i), col2(j)], [row1(i), row2(j)], '-');
            scatter(col2(j), row2(j), w_size1 * w_size1, 's');
        end
    end
end

% extract coordinates of matching in question (b)
point2DOrig1 = [];
point2DOrig2 = [];
for i = 1 : size(row1, 1)
    for j = 1 : size(row2, 1)
        if match(i, j) == 1
            point2DOrig1 = [point2DOrig1; [row1(i), col1(i)]];
            point2DOrig2 = [point2DOrig2; [row2(j), col2(j)]];
        end
    end
end



%---------- Question (c) ------------%
% outliers rejection %

% MSAC method
[inlierIndex, trials] = MSAC(point2DOrig1, point2DOrig2);
inlier1 = [];
inlier2 = [];
for i = 1 : length(inlierIndex)
    if inlierIndex(i) == 1
        inlier1 = [inlier1; point2DOrig1(i, :)];
        inlier2 = [inlier2; point2DOrig2(i, :)];
    end
end
disp('Question (c):');
disp('number of inliers:');
disp(sum(inlierIndex));
disp('number of trials:');
disp(trials);
```

```

% show the feature matching image
figure(4)
subplot(1, 2, 1);
imshow(imread(image1));
hold on;
for i = 1 : length(inlierIndex)
    if inlierIndex(i) == 1
        plot([point2DOrig1(i, 2), point2DOrig2(i, 2)], ...
             [point2DOrig1(i, 1), point2DOrig2(i, 1)], '-');
        scatter(point2DOrig1(i, 2), point2DOrig1(i, 1), w_size1 * w_size1, 's');
    end
end
subplot(1, 2, 2);
imshow(imread(image2));
hold on;
for i = 1 : length(inlierIndex)
    if inlierIndex(i) == 1
        plot([point2DOrig2(i, 2), point2DOrig1(i, 2)], ...
             [point2DOrig2(i, 1), point2DOrig1(i, 1)], '-');
        scatter(point2DOrig2(i, 2), point2DOrig2(i, 1), w_size1 * w_size1, 's');
    end
end



%---------- Question (d) ------------%
% linear estimation %

F_DLT = linearEstimation(inlier1, inlier2);
num_pnt = size(inlier1, 1);
format longg;
disp('Question (d):');
disp('F_DLT = ');
disp(F_DLT);

%{
for i = 1 : num_pnt
    tmp_pnt1 = inlier1(i, :);
    pnt1 = [tmp_pnt1, 1];
    tmp_pnt2 = inlier2(i, :);
    pnt2 = [tmp_pnt2, 1];
    disp(pnt2 * F_DLT * pnt1');
end
%}



%---------- Question (e) ------------%
% nonlinear estimation %

disp('Question (e):');
F = F_DLT;
uni = ones(num_pnt, 1);
inlier1 = [inlier1, uni];
inlier2 = [inlier2, uni];

pnt3D = triangulate(F, inlier1, inlier2);

[F_LM, cost_lst] = levenbergEst(F, inlier1, inlier2, pnt3D);
disp('cost for each iteration: ');
for i = 1 : size(cost_lst, 2)
    disp(cost_lst(i));
end
disp('F_LM = ');
```

```
disp(F_LM);



%————————— Question (f) ——————————%
% point to line mapping %

F = F_LM;
mapping(F, image1, image2);
disp('Question (f):');
disp('The figure will show');
```

```
% featureDetection.m %

% feature detection
function [row, col] = featureDetection(image, w_size1, threshold, w_size2)
    % read the image in RGB format
    i = imread(image);

    % convert RGB to gray scale
    grayImage = rgb2gray(i);

    % calculate gradient images
    K = [-1, 8, 0, -8, 1] / 12;
    Ix = imfilter(grayImage, K);
    Iy = imfilter(grayImage, K');

    % calculate Isquare and IxIy
    IxSquare = Ix .* Ix;
    IxIy = Ix .* Iy;
    IySquare = Iy .* Iy;

    % calculate minor eigenvalue image
    eigenImage = calEigenImage(IxSquare, IxIy, IySquare, w_size1);

    % set 0 if below threshold
    threshEigenImage = eigenImage .* (eigenImage >= threshold);

    % non maximum suppression
    % maximum filter
    Imax = ordfilt2(threshEigenImage, w_size2 * w_size2, ones(w_size2, w_size2));
    % compare two image, generate image J
    imageJ = threshEigenImage .* (threshEigenImage >= Imax);

    % find the coordinate of corner
    [row, col] = findCorner(IxSquare, IxIy, IySquare, imageJ, w_size1);
end

% calculate minor eigenvalue image
function m = calEigenImage(IxSquare, IxIy, IySquare, w_size)
    len = size(IxIy);
    m = zeros(len(1), len(2));
    for i = 1 : len(1)
        for j = 1 : len(2)
            [N, b] = calGradMatrix(IxSquare, IxIy, IySquare, i, j, w_size);
            m(i, j) = 0.5 * (trace(N) - sqrt(trace(N) ^ 2 - 4 * det(N)));
        end
    end
end

% Calculate gradient matrix
function [m, b] = calGradMatrix(IxSquare, IxIy, IySquare, i, j, w_size)
    m = zeros(2, 2);
    b = zeros(2, 1);
    half = (w_size - 1) / 2;
```

```
    len = size(IxIy);
    i_start = max(i - half, 1);
    j_start = max(j - half, 1);
    i_end = min(i + half, len(1));
    j_end = min(j + half, len(2));
    m(1, 1) = sum(sum(IxSquare(i_start : i_end, j_start : j_end)));
    m(1, 2) = sum(sum(IxIy(i_start : i_end, j_start : j_end)));
    m(2, 1) = m(1, 2);
    m(2, 2) = sum(sum(IySquare(i_start : i_end, j_start : j_end)));
    for p = i_start : i_end
        for q = j_start : j_end
            b(1) = b(1) + double(p) * double(IxSquare(p, q)) + double(q) * double(IxIy(p, q)
    );
            b(2) = b(2) + double(q) * double(IySquare(p, q)) + double(p) * double(IxIy(p, q)
    );
        end
    end
end

% find corner coordinates
function [row, col] = findCorner(IxSquare, IxIy, IySquare, imageJ, w_size)
    len = size(imageJ);
    row = [];
    col = [];
    for i = 1 : len(1)
        for j = 1 : len(2)
            if (imageJ(i, j) > 0)
                [N, b] = calGradMatrix(IxSquare, IxIy, IySquare, i, j, w_size);
                coord = N \ b;
                coord = coord';
                if coord(1) >= 15 && coord(1) <= len(1) - 15 && ...
                   coord(2) >= 15 && coord(2) <= len(2) - 15
                    row = [row; coord(1)];
                    col = [col; coord(2)];
                end
            end
        end
    end
end

% extract the corner coordinates
function [row, col] = extractCorner(corner)
    row = [];
    col = [];
    for i = 15 : size(corner, 1) - 15
        for j = 15 : size(corner, 2) - 15
            if corner(i, j) == 1
                row = [row, i];
                col = [col, j];
            end
        end
    end
    row = row';
    col = col';
end
```

```
% featureMatching.m %

% feature matching
function match = featureMatching(image1, image2, row1, col1, row2, col2, w_size, simThresh,
    ratioThresh)
    I1 = imread(image1);
    I2 = imread(image2);
    Image1 = rgb2gray(I1);
    Image2 = rgb2gray(I2);
```

```
    length1 = size(row1);
    len1 = length1(1);
    length2 = size(row2);
    len2 = length2(1);
    match = zeros(len1, len2);
    % calculate correlation coefficient matrix
    correl = zeros(len1, len2);
    for i = 1 : len1
        [win1, size1] = fetchWindow(Image1, size(Image1), row1, col1, i, w_size);
        for j = 1 : len2
            [win2, size2] = fetchWindow(Image2, size(Image2), row2, col2, j, w_size);
            if size1 == w_size^2 && size2 == w_size^2
                correl(i, j) = corr2(win1, win2);
            else
                correl(i,j) = -1.0;
            end
        end
    end
    % one to one match
    maxValue = max(max(correl));
    count = 0;
    while maxValue > simThresh
        [X, Y] = find(correl == maxValue);
        x = X(1);
        y = Y(1);
        correl(x, y) = -1;
        nextMaxValue = max(max(correl(x,:)), max(correl(:,y)));
        if (1.0 - maxValue) < (1.0 - nextMaxValue) * ratioThresh
            count = count + 1;
            match(x, y) = 1;
        end;
        correl(x,:) = -1;
        correl(:,y) = -1;
        maxValue = max(max(correl));
    end
end

% fetch the window
function [win, size] = fetchWindow(image, len, row, col, i, w_size)
    half = (w_size - 1) / 2;
    i_start = max(round(row(i)) - half, 1);
    j_start = max(round(col(i)) - half, 1);
    i_end = min(round(row(i)) + half, len(1));
    j_end = min(round(col(i)) + half, len(2));
    win = image(i_start : i_end, j_start : j_end);
    size = (i_end - i_start + 1) * (j_end - j_start + 1);
end
```

```
% MSAC.m %

% MSAC method
function [inlierIndex, trials] = MSAC(point2DOrig1, point2DOrig2)
    format longg;
    % transfer to homogeneous
    num_point = size(point2DOrig1, 1);
    point2DHomo1 = [point2DOrig1, ones(num_point, 1)];
    point2DHomo2 = [point2DOrig2, ones(num_point, 1)];

    % MSAC algorithm
    consensus_min_cost = Inf;
    trials = 0;
    max_trials = Inf;
    threshold = 0;
    prob = 0.99;
    alpha = 0.95;
```

```
    variance = 1;
    codimension = 1;
    tolerance = chi2inv(alpha, codimension);

    rand('seed', 2);

    % begin iteration
    while (trials < max_trials && consensus_min_cost > threshold)
        % generate seven random samples
        sampleIndex = randperm(num_point, 7);
        % compute model F
        F_sol = sevenPoint(point2DHomo1, point2DHomo2, sampleIndex);
        num_F = size(F_sol, 1) / 3;
        for n = 1 : num_F
            F = F_sol((n - 1) * 3 + 1: n * 3, :);
            % compute cost
            cost = 0;
            for i = 1 : num_point
                % compute sampson error
                error_i = sampsonError(F, point2DHomo1(i, :), point2DHomo2(i, :));
                if error_i < tolerance
                    cost = cost + error_i;
                else
                    cost = cost + tolerance;
                end
            end
            % update model
            if cost < consensus_min_cost
                consensus_min_cost = cost;
                model_F = F;
                % count number of inliers
                dist_error = zeros(1, num_point);
                for i = 1 : num_point
                    dist_error(i) = sampsonError(F, point2DHomo1(i, :), point2DHomo2(i, :));
                end
                % update max_trials
                num_inliers = sum(dist_error <= tolerance);
                w = num_inliers / num_point;
                max_trials = log(1 - prob) / log(1 - w^7);
            end
        end
        trials = trials + 1;
    end
    % count inliers
    dist_error = zeros(1, num_point);
    for i = 1 : num_point
        dist_error(i) = sampsonError(model_F, point2DHomo1(i, :), point2DHomo2(i, :));
    end
    inlierIndex = (dist_error <= tolerance);
end

% seven point method for foundamental matrix
function F_sol = sevenPoint(pointHomo1, pointHomo2, sampleIndex)
    format longg;
    point1 = pointHomo1(sampleIndex, :)';
    point2 = pointHomo2(sampleIndex, :)';
    A = [];
    for i = 1 : 7
        A(i, :) = kron(point2(:, i)', point1(:, i)');
    end

    [~,~,V] = svd(A);
    a = V(:,8);
    b = V(:,9);
    F1 = reshape(a,3,3)';
```

```
    F2 = reshape(b,3,3)';

    syms alph
    F = alph*F1 + F2;
    equation = solve(det(F));
    F_sol = [];
    alpha_sol = double(vpa(equation));
    for i = 1 : length(alpha_sol)
        alpha = alpha_sol(i);
        if ~isreal(alpha)
            continue;
        end
        F_i = alpha * F1 + F2;
        F_sol = [F_sol; F_i];
    end
end

% calculate Sampson error
function error = sampsonError(F, point1, point2)
    point1 = point1';
    point2 = point2';
    nominator = (point2' * F * point1)^2;
    denominator = (point2' * F(:, 1))^2 + (point2' * F(:, 2))^2 + ...
                  (F(1, :) * point1)^2 + (F(2, :) * point1)^2;
    error = nominator * 1.0 / denominator;
end
```

```
% linearEstimation.m %

% DLT linear estimation %
function F_DLT = linearEstimation(inlier1, inlier2)
    % data normalization
    [point1, T1] = dataNormalization(inlier1, 2);
    [point2, T2] = dataNormalization(inlier2, 2);

    % using DLT compute matrix A
    % H * point2 = point1
    A = DLTAlgorithm(point1, point2);

    % using svd compute projection matrix P
    F_DLT = calProjMat(A, T1, T2);
end

% DLT algorithm
function matA = DLTAlgorithm(point1, point2)
    num_point = size(point1, 1);
    point1 = point1';
    point2 = point2';
    matA = [];
    % using house holder matrix
    % to calculate left null space of x
    for i = 1 : num_point
        matA = [matA; kron(point2(:, i)', point1(:, i)')];
    end
end

% using svd compute projection matrix P
function F_DLT = calProjMat(A, T1, T2)
    [~, ~, V] = svd(A);
    f = V(:, end);
    F = reshape(f, 3, 3)';
    [U,D,V] = svd(F);
    D(3,3)= 0;
    F = U * D * V';
    F_DLT = T2' * F * T1;
```

```
    F_DLT = F_DLT / norm(F_DLT, 'fro');
end

% Data Normalization
function [point, T] = dataNormalization(data, dim)
    num_point = size(data, 1);
    % calculate mean and variance
    m = mean(data);
    v = var(data);
    % calculate normalized vector
    if (dim == 2)
        s = sqrt(2.0 / sum(v));
        T = zeros(3, 3);
        T(1, 1) = s;
        T(2, 2) = s;
        T(3, 3) = 1.0;
        T(1, 3) = -1.0 * m(1) * s;
        T(2, 3) = -1.0 * m(2) * s;
    else
        s = sqrt(3.0 / sum(v));
        T = zeros(4, 4);
        T(1, 1) = s;
        T(2, 2) = s;
        T(3, 3) = s;
        T(4, 4) = 1.0;
        T(1, 4) = -1.0 * m(1) * s;
        T(2, 4) = -1.0 * m(2) * s;
        T(3, 4) = -1.0 * m(3) * s;
    end
    % transfer inhomo to homo data
    unit = ones(num_point, 1);
    point = [data, unit];
    % normalize the data
    point = T * point';
    point = point';
end
```

```
% triangulate.m%

% two-view optimal triangulation
function pnt3D = triangulate(F, inlier1, inlier2)
    num_pnt = size(inlier1, 1);
    % uni = ones(num_pnt, 1);
    % inlier1_homo = [inlier1, uni];
    % inlier2_homo = [inlier2, uni];
    inlier1_homo = inlier1;
    inlier2_homo = inlier2;
    pnt3D = zeros(num_pnt, 4);
    pnt2D1 = zeros(num_pnt, 3);
    pnt2D2 = zeros(num_pnt, 3);

    % compute matrix P
    W = [0, 1, 0;
         -1, 0, 0;
         0, 0, 0];
    Z = [0, -1, 0;
         1, 0, 0;
         0, 0, 1];
    P = [eye(3),zeros(3, 1)];
    [~, ~, V] = svd(P);
    pnt1 = V(:, end);
    [U, D, V] = svd(F);
    D_prime = D;
    D_prime(3, 3) = (D(1, 1) + D(2, 2)) / 2.0;
    m = U * Z * D_prime * V';
```

```
    e_prime = -U(:, 3);
    proj_prime = [m, e_prime];

    disp('doing triangulation, this may take a while.');
    % do triangulation for each point
    for i = 1 : num_pnt
        point1 = inlier1_homo(i, :);
        point2 = inlier2_homo(i, :);

        % compute matrix Fs
        T  = [point1(3), 0, -point1(1);
              0, point1(3), -point1(2);
              0, 0, point1(3)];
        T_prime = [point2(3), 0, -point2(1);
                   0, point2(3), -point2(2);
                   0, 0, point2(3)];
        Fs = inv(T_prime') * F * inv(T);

        % compute epipoles of Fs
        [~, ~, V] = svd(Fs);
        e = V(:, end);
        [~, ~, V] = svd(Fs');
        e_prime = V(:, end);
        % scale the epipoles
        e = e / sqrt(e(1)^2 + e(2)^2);
        e_prime = e_prime / sqrt(e_prime(1)^2 + e_prime(2)^2);

        % compute the rotation matrices
        R  = [e(1), e(2), 0;
             -e(2), e(1), 0;
              0, 0, 1];
        R_prime = [e_prime(1), e_prime(2), 0;
                  -e_prime(2), e_prime(1), 0;
                   0, 0, 1];

        % F matrix in special form
        Fs = R_prime * Fs * R';
        f = e(3);
        f_prime = e_prime(3);
        a = Fs(2,2);
        b = Fs(2,3);
        c = Fs(3,2);
        d = Fs(3,3);

        % solve g(t) for t
        syms t
        g_t = t * ((a * t + b)^2 + f_prime^2 * (c * t + d)^2)^2 - ...
              (a * d - b * c) * (1 + f^2 * t^2)^2 * (a * t + b) * ...
              (c * t + d);
        tmp_t = solve(g_t);
        t = double(vpa(tmp_t));
        cost = zeros(6, 1);
        for n = 1 : 6
            real_t = real(t(n));
            cost(n) = (real_t)^2 / (1 + f^2 * (real_t)^2) + ...
                      (c * real_t + d)^2 / ((a * real_t + b)^2 + ...
                      f_prime^2 * (c * real_t + d)^2);
        end
        % find t with minimum cost
        [~,index] = min(cost);
        t_opt = t(index);

        % compute x_hat
        line = [t_opt * f, 1, -t_opt]';
        line_prime = [-f_prime * (c * t_opt + d), a * t_opt + b, c * t_opt + d]';
```

```
        x_hat  = [-line(1) * line(3), -line(2) * line(3), line(1)^2 + line(2)^2]';
        x_hat_prime = [-line_prime(1) * line_prime(3), -line_prime(2) * ...
                       line_prime(3), line_prime(1)^2 + line_prime(2)^2]';

        % correct points mapped back to original coordinates
        tmp_pnt1 = (inv(T) * R' * x_hat)';
        pnt2D1(i, :) = tmp_pnt1 / tmp_pnt1(3);
        tmp_pnt2 = (inv(T_prime) * R_prime' * x_hat_prime)';
        pnt2D2(i, :) = tmp_pnt2 / tmp_pnt2(3);

        % compute the line
        line_prime = F * pnt2D1(i, :)';
        line_orth_prime = [-line_prime(2) * pnt2D2(i, 3), line_prime(1) * ...
                     pnt2D2(i, 3), line_prime(2) * pnt2D2(i, 1) - ...
                     line_prime(1) * pnt2D2(i,2)]';
        plane = proj_prime' * line_orth_prime;

        % compute the 3D point
        P_plus = P' * inv(P * P');
        pnt2 = P_plus * pnt2D1(i, :)';
        tmp_3D_pnt = [pnt2(1) * plane(4) * pnt1(4), pnt2(2) * plane(4) * pnt1(4), ...
                       pnt2(3) * plane(4) * pnt1(4), -pnt1(4) * (plane(1) * pnt2(1) + ...
                       plane(2) * pnt2(2) + plane(3) * pnt2(3))];
        pnt3D(i, :) = tmp_3D_pnt / tmp_3D_pnt(4);
    end
    disp('triangulation done.');
end
```

```
% levenbergEst.m %

function [F_LM, cost_lst] = levenbergEst(F_init, inlier1, inlier2, pnt3D)
    F = F_init;
    cost_lst = [];
    inlier1 = inlier1(:, 1 : 2);
    inlier2 = inlier2(:, 1 : 2);
    inlier1 = inlier1';
    inlier2 = inlier2';
    pnt3D = pnt3D';
    num_pnt = size(inlier1, 2);
    Z = [0, -1, 0;
         1, 0, 0;
         0, 0, 1];

    % normarlize 3D points
    for i = 1 : num_pnt
        pnt3D(:, i) = pnt3D(:, i) / norm(pnt3D(:, i));
    end

    % compute initial P and P_prime
    P = [eye(3), zeros(3, 1)];
    [w_u, w_v, sigma, s] = parameterize_F(F);
    [U, D, V] = svd(F);
    D_prime = D;
    D_prime(3, 3) = (D(1, 1) + D(2, 2)) / 2.0;
    m = U * Z * D_prime * V';
    e_prime = -U(:, 3);
    P_prime = [m, e_prime];

    % compute initial cost
    pnt_pred1 = P * pnt3D;
    pnt_pred1 = pnt_pred1 ./ pnt_pred1(3, :);
    pnt_pred1_inhomo = pnt_pred1(1 : 2, :);
    pnt_pred2 = P_prime * pnt3D;
    pnt_pred2 = pnt_pred2 ./ pnt_pred2(3, :);
    pnt_pred2_inhomo = pnt_pred2(1 : 2, :);
```

```matlab
    error1 = pnt_pred1_inhomo - inlier1;
    error2 = pnt_pred2_inhomo - inlier2;
    cost = norm(error1)^2 + norm(error2)^2;
    cost_lst = [cost_lst, cost];

    % parameterization of 3D point
    pnt_scene_param = zeros(3, num_pnt);
    for i = 1 : num_pnt
        pnt = pnt3D(:, i);
        pnt_param = parameterize(pnt);
        pnt_scene_param(:, i) = pnt_param;
    end

    param_F = [w_u', w_v', s]';
    lam = 0.001;
    n = 0;
    % begin iteration
    for k = 1 : 40
        % compute Jacobian matrix
        [A, B1, B2] = calJacobian(num_pnt, pnt_pred1, pnt_pred2, pnt_scene_param, P_prime,
    param_F);

        % compute normal equations matrix
        U = zeros(7, 7);
        V = zeros(3, 3, num_pnt);
        W = zeros(7, 3, num_pnt);
        error_part1 = zeros(7, 1);
        error_part2 = zeros(3, 1, num_pnt);
        for i = 1 : num_pnt
            U = U + A(:, :, i)' * A(:, :, i);
            V(:, :, i) = B1(:, :, i)' * B1(:, :, i) + ...
                         B2(:, :, i)' * B2(:, :, i);
            W(:, :, i) = A(:, :, i)' * B2(:, :, i);
            error_part1 = error_part1 + A(:, :, i)' * error2(:, i);
            error_part2(:, :, i) = B1(:, :, i)' * error1(:, i) + ...
                                   B2(:, :, i)' * error2(:, i);
        end

        % compute augmented normal euqations
        S = U + lam * eye(7);
        epsilon = error_part1;
        for i = 1 : num_pnt
            S = S - W(:, :, i) * inv(V(:, :, i) + lam * eye(3)) * W(:, :, i)';
            epsilon = epsilon - W(:, :, i) * inv(V(:, :, i) + lam * eye(3)) * error_part2(:,
     :, i);
        end
        delata_part1 = linsolve(S, epsilon);
        delata_part2 = zeros(3, 1, num_pnt);
        for i = 1 : num_pnt
            delata_part2(:, :, i) = inv(V(:, :, i) + lam * eye(3)) * ...
                                    (error_part2(:, :, i) - W(:, :, i)' * delata_part1);
        end

        % update
        param_F_update = param_F + delata_part1;
        %{
        w_u_update = param_F_update(1 : 3);
        w_v_update = param_F_update(4 : 6);
        s_update = param_F_update(7);
        sigma_update = deparameterize(s_update);
        F_update = expm(formMat(w_u_update)) * diag([sigma_update', 0]) * expm(formMat(
    w_v_update))';
        %}
        F_update = deparameterize_F(param_F_update(1 : 3), param_F_update(4 : 6),
    param_F_update(7));
```

```
        pnt_scene_param_update = zeros(3, num_pnt);
        pnt3D_update = zeros(4, num_pnt);
        for i = 1 : num_pnt
            pnt_scene_param_update(:, i) = pnt_scene_param(:, i) + delata_part2(:, :, i);
            pnt3D_update(:, i) = deparameterize(pnt_scene_param_update(:, i));
            pnt3D_update(:, i) = pnt3D_update(:, i) / pnt3D_update(4, i);
            pnt3D_update(:, i) = pnt3D_update(:, i) / norm(pnt3D_update(:, i));
        end

        % compute P, P_prime and cost
        P_update = [eye(3), zeros(3, 1)];
        P_prime_update = cal_P_prime(F_update);
        [cost_update, error1_update, error2_update, pnt_pred1_update, pnt_pred2_update] = ...
            cal_cost(P_update, P_prime_update, pnt3D_update, inlier1, inlier2);

        % jump the loop
        if(cost_update > cost)
            lam = 10 * lam;
        else
            n = n + 1;
            lam = lam / 10;
            param_F = param_F_update;
            F = F_update;
            P_prime = P_prime_update;
            error1 = error1_update;
            error2 = error2_update;
            cost = cost_update;
            cost_lst = [cost_lst, cost];
            pnt_scene_param = pnt_scene_param_update;
            pnt3D = pnt3D_update;
            pnt_pred1 = pnt_pred1_update;
            pnt_pred2 = pnt_pred2_update;
        end
    end
    F_LM = F;
end

% parameteriztion of matrix F
function [w_u, w_v, sigma, s] = parameterize_F(F)
    [U, D, V] = svd(F);
    if det(U) < 0
        U = -U;
    end
    if det(V) < 0
        V = -V;
    end
    D = [D(1, 1), D(2, 2)]';
    D = D / norm(D);
    tmp_u = logm(U);
    w_u = [tmp_u(3, 2), tmp_u(1, 3), tmp_u(2, 1)]';
    tmp_v = logm(V);
    w_v = [tmp_v(3, 2), tmp_v(1, 3), tmp_v(2, 1)]';
    sigma = D;
    s = parameterize(sigma);
end

% deparameterization of matrix F
function F = deparameterize_F(w_u, w_v, s)
    U = expm(formMat(w_u));
    V = expm(formMat(w_v));
    sigma = deparameterize(s);
    F = sigma(1) * U(:, 1) * V(:, 1)' + sigma(2) * U(:, 2) * V(:, 2)';
end

```

```
% parameterize
function paramVector = parameterize(P)
    a = P(1);
    b = P(2 : length(P));
    paramVector = (2.0 / (sinc(acos(a)))) * b;
    normP = norm(paramVector);
    if (normP > pi)
        paramVector = (1.0 - 2 * pi / normP * ceil((normP - pi) / 2 * pi)) * paramVector;
    end
end

% deparameterize
function deparamVector = deparameterize(P)
    normP = norm(P);
    deparamVector = [cos(normP / 2.0), ((sinc(normP / 2.0)) / 2.0) * P']';
end

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

% derivative of sinc(x)
function res = derivSinc(x)
    if x == 0
        res = 0.0;
    else
        res = cos(x) / x - sin(x) / (x * x);
    end
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

% compute Jacobian matrix
function [A, B1, B2] = calJacobian(num_pnt, pnt_pred1, pnt_pred2, pnt_scene_param, P_prime,
    param_F)
    w_u = param_F(1 : 3);
    w_v = param_F(4 : 6);
    s = param_F(7);
    sigma = deparameterize(s);
    A = zeros(2, 7, num_pnt);
    B1 = zeros(2, 3, num_pnt);
    B2 = zeros(2, 3, num_pnt);
    for i = 1:num_pnt
        pnt1 = pnt_pred1(:, i);
        pnt2 = pnt_pred2(:, i);
        pnt_scene = deparameterize(pnt_scene_param(:, i));

        % compute A
        A1_i = [1 / pnt2(3), 0, -pnt2(1) / pnt2(3)^2;
                0, 1 / pnt2(3), -pnt2(2) / pnt2(3)^2];
        A2_i = zeros(3, 12);
        A2_i(1, 1 : 4) = pnt_scene';
        A2_i(2, 5 : 8) = pnt_scene';
        A2_i(3, 9 : 12) = pnt_scene';

```

```
        U = expm(formMat(w_u));
        V = expm(formMat(w_v));
        tmp_mat = zeros(9, 3);
        tmp_mat(2, 3) = -1;
        tmp_mat(3, 2) = 1;
        tmp_mat(4, 3) = 1;
        tmp_mat(6, 1) = -1;
        tmp_mat(7, 2) = -1;
        tmp_mat(8, 1) = 1;

        % compute dP' / dw_u
        tmp = [-sigma(2) * V(:, 2), sigma(1) * V(:, 1), (sigma(1) + sigma(2)) / 2.0 * V(:, 3);
               0, 0, -1];
        dp_du = kron(eye(3), tmp);
        theta = norm(w_u);
        dtheta_dw = (1.0 / theta) * w_u';
        s = (1 - cos(theta) / theta^2);
        tmp_m = vec(w_u * w_u');
        dm_dw = kron(w_u, eye(3)) + kron(eye(3), w_u);
        ds_dw = dtheta_dw * ((theta * sin(theta) - 2 * (1 - cos(theta))) / theta^3);
        du_dw_u = -vec(eye(3)) * sin(theta) * dtheta_dw + sinc(theta) * tmp_mat + ...
                  vec(formMat(w_u)) * derivSinc(theta) * dtheta_dw + s * dm_dw + tmp_m * ds_dw;
        dp_dw_u = dp_du * du_dw_u;

        % compute dP' / dw_v
        dp_dv = [kron(eye(3), [sigma(1) * U(1, 2), -sigma(2) * U(1, 1), (sigma(1) + sigma(2)) / 2.0 * U(1, 3)]);
                 zeros(1, 9);
                 kron(eye(3), [sigma(1) * U(2, 2), -sigma(2) * U(2, 1), (sigma(1) + sigma(2)) / 2.0 * U(2, 3)]);
                 zeros(1, 9);
                 kron(eye(3), [sigma(1) * U(3, 2), -sigma(2) * U(3, 1), (sigma(1) + sigma(2)) / 2.0 * U(3, 3)]);
                 zeros(1, 9)];
        theta = norm(w_v);
        dtheta_dw = (1.0 / theta) * w_v';
        s = (1 - cos(theta) / theta^2);
        tmp_m = vec(w_v * w_v');
        dm_dw = kron(w_v, eye(3)) + kron(eye(3), w_v);
        ds_dw = dtheta_dw * ((theta * sin(theta) - 2 * (1 - cos(theta))) / theta^3);
        dv_dw_v = -vec(eye(3)) * sin(theta) * dtheta_dw + sinc(theta) * tmp_mat + ...
                  vec(formMat(w_v)) * derivSinc(theta) * dtheta_dw + s * dm_dw + tmp_m * ds_dw;
        dp_dw_v = dp_dv * dv_dw_v;

        % compute dP' / ds
        dp_dsigma = [U(1, 2) * V(:, 1) + 0.5 * U(1, 3) * V(:, 3), 0.5 * U(1, 3) * V(:, 3) - U(1, 1) * V(:, 2);
                     0, 0;
                     U(2, 2) * V(:, 1) + 0.5 * U(2, 3) * V(:, 3), 0.5 * U(2, 3) * V(:, 3) - U(2, 1) * V(:, 2);
                     0, 0;
                     U(3, 2) * V(:, 1) + 0.5 * U(3, 3) * V(:, 3), 0.5 * U(3, 3) * V(:, 3) - U(3, 1) * V(:, 2);
                     0, 0];
        dsigma_ds = zeros(2, 1);
        dsigma_ds(1) = -0.5 * sigma(2);
        if norm(s) == 0
            dsigma_ds(2) = 0.5;
        else
            dsigma_ds(2) = 0.5 * sinc(0.5 * norm(s)) + 0.25 * norm(s) * derivSinc(0.5 * norm(s)) * s * s;
        end
```

```
        dp_ds = dp_dsigma * dsigma_ds;

        A3_i = [dp_dw_u, dp_dw_v, dp_ds];
        A_i = A1_i * A2_i * A3_i;
        A(:, :, i) = A_i;

        % compute B1
        B1_1_i = [1 / pnt1(3), 0, -pnt1(1) / pnt1(3)^2;
                  0, 1 / pnt1(3), -pnt1(2) / pnt1(3)^2];
        B1_2_i = [eye(3), zeros(3, 1)];
        B1_3_i = zeros(4, 3);
        B1_3_i(1, :) = -0.25 * (sinc(norm(pnt_scene_param(:, i)) / 2)) * ...
                       pnt_scene_param(:, i)';
        if norm(pnt_scene_param(:, i)) == 0
            B1_3_i(2 : 4, :) = 0.5 * eye(3);
        else
            B1_3_i(2 : 4, :) = sinc(norm(pnt_scene_param(:, i)) / 2) * 0.5 * ...
                               eye(3) + (1/(4 * norm(pnt_scene_param(:, i)))) * ...
                               (derivSinc(norm(pnt_scene_param(:, i)) / 2)) * ...
                               pnt_scene_param(:, i) * pnt_scene_param(:, i)';
        end
        B1_i = B1_1_i * B1_2_i * B1_3_i;
        B1(:, :, i) = B1_i;

        % compute B2
        B2_1_i = [1 / pnt2(3), 0, -pnt2(1) / pnt2(3)^2;
                  0, 1 / pnt2(3), -pnt2(2) / pnt2(3)^2];

        B2_2_i = P_prime;

        B2_3_i = zeros(4, 3);
        B2_3_i(1, :) = -0.25 * (sinc(norm(pnt_scene_param(:, i)) / 2)) * ...
                      pnt_scene_param(:, i)';
        if norm(pnt_scene_param(:, i)) == 0
            B2_3_i(2 : 4, :) = 0.5 * eye(3);
        else
            B2_3_i(2 : 4, :) = sinc(norm(pnt_scene_param(:, i)) / 2) * 0.5 * ...
                               eye(3) + (1/(4 * norm(pnt_scene_param(:, i)))) * ...
                               (derivSinc(norm(pnt_scene_param(:, i)) / 2)) * ...
                               pnt_scene_param(:, i) * pnt_scene_param(:, i)';
        end
        B2_i = B2_1_i * B2_2_i * B2_3_i;
        B2(:, :, i) = B2_i;
    end
end

% vectorize
function res = vec(a)
    tmp = a';
    res = tmp(:);
end

% compute P_prime
function P_prime = cal_P_prime(F)
    Z = [0, -1, 0;
         1, 0, 0;
         0, 0, 1];
    [U, D, V] = svd(F);
    D_prime = D;
    D_prime(3, 3) = (D(1, 1) + D(2, 2)) / 2.0;
    m = U * Z * D_prime * V';
    e_prime = -U(:, 3);
    P_prime = [m, e_prime];
end

```

```
% compute cost
function [cost, error1, error2, pnt_pred1, pnt_pred2] = cal_cost(P, P_prime, pnt3D, inlier1,
    inlier2)
    pnt_pred1 = P * pnt3D;
    pnt_pred1 = pnt_pred1 ./ pnt_pred1(3, :);
    pnt_pred1_inhomo = pnt_pred1(1 : 2, :);
    pnt_pred2 = P_prime * pnt3D;
    pnt_pred2 = pnt_pred2 ./ pnt_pred2(3, :);
    pnt_pred2_inhomo = pnt_pred2(1 : 2, :);
    error1 = pnt_pred1_inhomo - inlier1;
    error2 = pnt_pred2_inhomo - inlier2;
    cost = norm(error1)^2 + norm(error2)^2;
end
```

```
% mapping.m %

% map points to epipolar lines
function mapping(F, image1, image2)
    win_size = 20;
    color = ['r', 'y', 'g'];
    points = [200, 300, 1;
              300, 800, 1;
              420, 760, 1]';
    lines = [];
    for i = 1 : 3
        lines(:, i) = F * points(:, i);
    end

    % show the feature matching image
    figure(5)
    subplot(1, 2, 1);
    imshow(imread(image1));
    hold on;
    for i = 1 : 3
        plot(points(2, i), points(1, i), 's', 'MarkerSize', win_size, 'Color', color(i));
    end
    subplot(1, 2, 2);
    imshow(imread(image2));
    hold on;
    for i = 1 : 3
        syms x1 x2
        f = [x1, 0, 1] * lines(:, i);
        x1 = double(solve(f));
        f = [x2, 1024, 1] * lines(:, i);
        x2 = double(solve(f));
        line([0, 1024], [x1, x2], 'color', color(i));
        hold on;
    end
end
```