Zhu, Zhongjian
University of California, San Diego

March 30, 2018

HOMEWORK 4

Problem 1. Automatic estimation of the planar projective transformation

(a) **Feature detection**
Calculate an image where each pixel value is the minor eigenvalue of the gradient matrix. Calculate the gradient images using the five-point central difference operator. Set resulting values that are below a specified threshold value to zero. Apply an operation that suppresses local nonmaximum pixel values in the minor eigenvalue image. Determine the subpixel feature coordinate.

Solution
First, in order to get the gradient matrix, we need to filter the image in x- and y-direction. The convolution kernel is $\mathbf{k} = (-1, 8, 0, -8, 1)^\top/12$. So the x-derivative image, I_x, is to filter each row with $\mathbf{k}^\top$ and the y-derivative image, I_y, is to filter each column with $\mathbf{k}$.
Then use the following equation to get the minor eigenvalue for each pixel.

$$\mathbf{N} = \begin{bmatrix} \sum_w I_x^2 & \sum_w I_x I_y \\ \sum_w I_x I_y & \sum_w I_y^2 \end{bmatrix}.$$

$$\lambda_{min} = \frac{Tr(\mathbf{N} - \sqrt{Tr(\mathbf{N})^2 - 4det(\mathbf{N})})}{2}$$

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. Set resulting values that are below a specified threshold value to zero. Now we get the minor eigenvalue matrix.
Next step is to apply the non-maxium suppression. For each pixel, we set a window on it and calculate the maximum value in the window, the local maximum. If the value of the center pixel of the window is less than the local maximum, set it to 0, otherwise, keep the value. This step is actually to find the corner pixels and set the value of non-corner pixels to 0.
Since what we need is not the corner pixel but rather the corner coordinates, we need to use the Föstner corner point operator to find the subpixels.

$$\begin{bmatrix} \sum_w I_x^2 & \sum_w I_x I_y \\ \sum_w I_x I_y & \sum_w I_y^2 \end{bmatrix} \begin{bmatrix} x_{corner} \\ y_{corner} \end{bmatrix} = \begin{bmatrix} \sum_w (xI_x^2 + yI_x I_y \\ \sum_w (xI_x I_y + yI_y^2 \end{bmatrix}$$

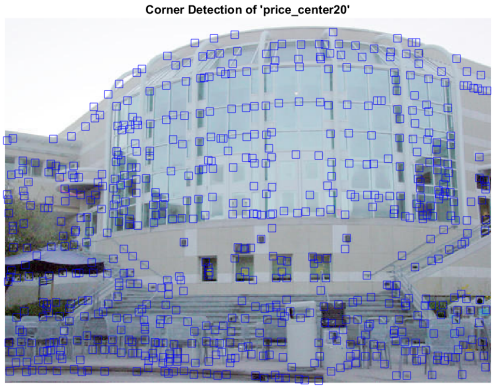
x_{corner} and y_{corner} are the coordinates of the subpixel.

Result
The size of the feature detection window is 9×9, the minor eigenvalue threshold value is 6.5, the size of the local non-maximum suppression window is 9×9.
The number of features detected in *price_ center20.JPG* is 644, and the number of features detected in *price_ center21.JPG* is 642.

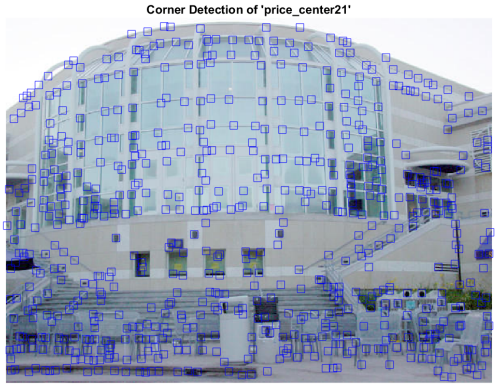
(a)

(b)

Figure 1: Input images. (a) price_center20. (b) price_center21.



(a)



(b)

Figure 2: Detected corners. (a) price_center20. (b) price_center21.

(b) **Feature matching**
Determine the set of one-to-one putative feature correspondences by performing a brute-force search for the greatest correlation coefficient value between windows centered about the detected features in image 1 and windows centered about the detected features in image 2. Only allow matches that are above a specified correlation coefficient 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 200 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.

Solution

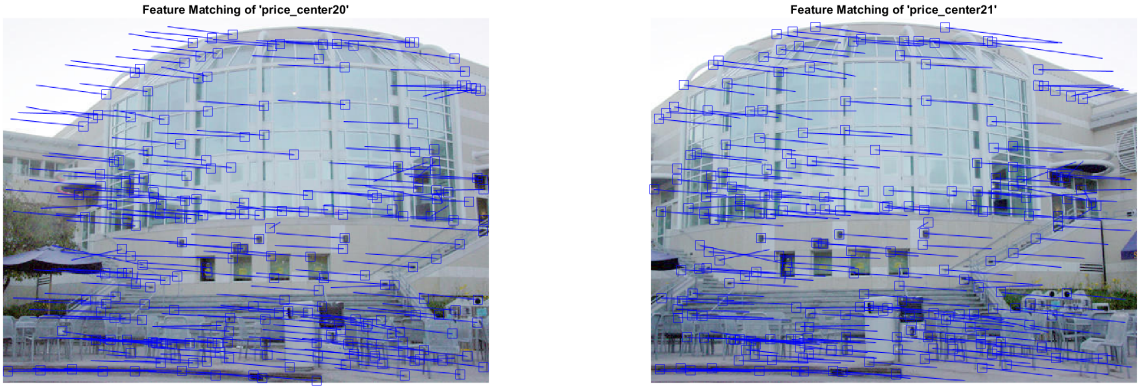
First we need to calculate the correlation coefficient between a given window in image 1 and all windows in image 2. Here since the subpixel value do not exist in image coordinate, we need to use interpolation.

Once we have the correlation matrix, we need to do the one-to-one matching. The algorithm is as follow:

```
Find indices of element with maximum value
  If similarity threshold < maximum value
    Store the best match value
    Set element to -1
    Find next best match value as
    max(next best match value in row, next best match value in column)
    If (1-best match value)<(1-next best match value)*distance ratio threshold
      Store feature value
    else
      Match is not unique enough
    Set row and column to -1
```

Result

The size of the proximity window is 25×160, the size of the matching window is 9×9, the correlation coefficient threshold value is 0.7, the distance ratio threshold value is 0.9, and the resulting number of putative feature correspondences is 217.



(a) (b)

Figure 3: Matched features. (a) price_center20. (b) price_center21.

(c) **Outlier rejection**
The resulting set of putative point correspondences contain both inlier and outlier correspondences. 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. Use the 4-point algorithm to estimate the planar projective transformation from the 2D points in image 1 to the 2D points in image 2. Calculate the Sampson error as a first order approximation to the geometric error.

Solution
The adaptive number of trials MSAC algorithm can be concluded as follow:

```
consensus_min_cost = inf
max_trials = inf
for(trials=0;trials<max_trials && consensus_min_cost>threshold;++trials)
{
  select a random sample
  calculate the model
  calculate the error
  calculate the cost
  if(consensus_min_cost<consensus_min_cost)
  {
    consensus_min_cost = consensus_cost
    consensus_min_cost_model = model
    number of inliers
    w = # of inliers / # of data points
    max_trials = log(1-p) / log(1-w^s)
  }
}
```

To calculate the model, $\mathbf{H}$, we need to use the canonical projective matrix:

$$\mathbf{e}_1 = \begin{bmatrix}1\\0\\0\end{bmatrix} \mathbf{e}_2 = \begin{bmatrix}0\\1\\0\end{bmatrix} \mathbf{e}_3 = \begin{bmatrix}0\\0\\1\end{bmatrix} \mathbf{e}_4 = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

The projection matrices from canonical basis to image 1 and image 2 are $\mathbf{H}_1$ and $\mathbf{H}_2$ respectively. So the projection matrix from image 1 to image 2 is $\mathbf{H} = \mathbf{H}_2^{-1}\mathbf{H}_1$.
To calculate the 2D projective transformation, 4 points are needed.

$$\begin{bmatrix}\mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_4\end{bmatrix} = \mathbf{H}\begin{bmatrix}s_1\mathbf{x}_1 & s_2\mathbf{x}_2 & s_3\mathbf{x}_3 & s_4\mathbf{x}_4\end{bmatrix} = \mathbf{H}\begin{bmatrix}\lambda_1\mathbf{x}_1 & \lambda_2\mathbf{x}_2 & \lambda_3\mathbf{x}_3 & \mathbf{x}_4\end{bmatrix}$$

where $\lambda_i = s_i/s_4$. Since $\mathbf{H}$ is up-to-scale, this will lead to the same result.

$$\begin{bmatrix}\mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3\end{bmatrix} = \mathbf{H}\begin{bmatrix}\lambda_1\mathbf{x}_1 & \lambda_2\mathbf{x}_2 & \lambda_3\mathbf{x}_3\end{bmatrix} \text{and} \mathbf{e}_4 = \mathbf{H}\mathbf{x}_4$$

So we can use

$$\begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3\end{bmatrix}\begin{bmatrix}\lambda_1\\\lambda_2\\\lambda_3\end{bmatrix} = \mathbf{x}_4$$

to solve for λ and then use

$$\mathbf{H}^{-1} = \begin{bmatrix} \lambda_1 \mathbf{x}_1 & \lambda_2 \mathbf{x}_2 & \lambda_3 \mathbf{x}_3 \end{bmatrix}$$

to get $\mathbf{H}$.
The error used here is the Sampson error which is the first order approximation of thee geometric error.

$$\delta_{\mathbf{X}} = -\mathbf{J}^\top (\mathbf{J}\mathbf{J}^\top)^{-\mathbf{1}} \epsilon$$

where

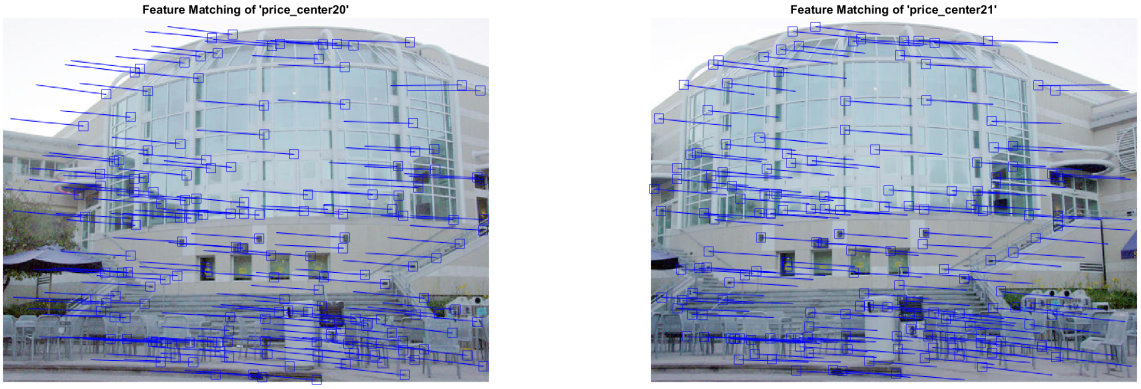
$$\mathbf{J} = \frac{\partial C_{\mathbf{H}}(\mathbf{X})}{\partial \mathbf{X}} = \begin{bmatrix} -h_{21} + \tilde{y}' h_{31} & -h_{22} + \tilde{y}' h_{32} & 0 & \tilde{x} h_{32} + \tilde{y} h_{32} + h_{33} \\ h_{11} - \tilde{x}' h_{31} & h_{12} - \tilde{x}' h_{32} & -(\tilde{x} h_{31} + \tilde{y} h_{32} + h_{33}) & 0 \end{bmatrix}$$

and

$$\epsilon = C_{\mathbf{H}}(\mathbf{X}) = \begin{bmatrix} -(\tilde{x} h_{21} + \tilde{y} h_{22} + h_{23}) + \tilde{y}'(\tilde{x} h_{31} + \tilde{y}' h_{32} + h_{33}) \\ \tilde{x} h_{11} + \tilde{y} h_{12} + h_{13} - \tilde{x}'(\tilde{x} h_{31} + \tilde{y}' h_{32} + h_{33}) \end{bmatrix}$$

is the cost associated with $\mathbf{X}$. So the Sampson error is $\|\delta_{\mathbf{X}}\|^2 = \epsilon^\top (\mathbf{J}\mathbf{J}^\top)^{-1} \epsilon$.
Next step is to calculate the cost. First we set a tolerance. Here we use $t^2 = F_m^{-1}(\alpha)$ where t^2 is the mean squared distance threshold and $F_m^{-1}(\alpha)$ is the inverse chi-squared cumulative distribution function. We assume the probability that a data point is an inlier is 0.95, $\alpha = 0.95$, and the variance of the measurement error is 1, $\sigma^2 = 1$. The codimension $m = 2$. For points whose error is less or equal to the tolerance, we add the error to the cost and for those whose error is greater than the tolerance, we add the tolerance to the cost. The points whose error are less or equal to the tolerance are inliers and the others are outliers. Then if the cost is less than the previous cost, we keep the cost, the camera pose and the number of inliers. Then update the maximum number of trials.

Result
The total number of the inliers are dependent on the choose of the points used to calculate the model, so we will have different number of inliers at each trial. Here we assumed that the probability p that at least one of the random samples does not contain any outliers is 0.99, the probability α that a given data point is an inlier is 0.95 and the variance σ^2 of the measurement error is 1. In the code file, I keep one random seed which leads to 165 inliers. The number of maximum trials is 7.9519, so it runs 8 times to find the consensus set.



(a) (b)

Figure 4: Matched features of inliers. (a) price_center20. (b) price_center21.

(d) **Linear estimation**
Estimate the planar projective transformation $\mathbf{H}_{DLT}$ from the resulting set of inlier correspondences using the direct linear transformation (DLT) algorithm (with data normalization).

Solution
For DLT algorithm, we need first normalize data point by

$$\mathbf{x} = \begin{bmatrix} s & 0 & -\mu_{\tilde{\mathbf{x}}} s \\ 0 & s & -\mu_{\tilde{\mathbf{y}}} s \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x}$$

where

$$\sigma^2 = \sigma_{\tilde{\mathbf{x}}}^2 + \sigma_{\tilde{\mathbf{x}}}^2, s = \frac{\sqrt{2}}{\sigma}$$

Then we need to solve the equation below:

$$\begin{bmatrix} [\mathbf{x}_1']^{\perp} \otimes \mathbf{x}_1^{\top} \\ [\mathbf{x}_2']^{\perp} \otimes \mathbf{x}_2^{\top} \\ \vdots \\ [\mathbf{x}_n']^{\perp} \otimes \mathbf{x}_n^{\top} \end{bmatrix} \mathbf{h} = \mathbf{0}$$

where $[\mathbf{x}_i]^{\perp}$ is the left null space of $\mathbf{x_i}$. To get the left null space of $\mathbf{x_i}$, we need the Householder Matrix.

$$\mathbf{H_V} = \mathbf{I} - 2\frac{\mathbf{V}\mathbf{V}^{\top}}{\mathbf{V}^{\top}\mathbf{V}}$$

$$\mathbf{V} = \mathbf{x} + sign(x_1)\|\mathbf{x}\|\mathbf{e_1}$$

The 2 to 3 rows of $\mathbf{H_V}$ is the left null space of $\mathbf{x_i}$.
Since the DoF of $\mathbf{h}$ is 8, we need at least 4 points to solve the equation. Suppose the left null space is $\mathbf{A}$, using Singular Value Decomposition we can get

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

and $\mathbf{h}$ is the last column of $\mathbf{V}$. Finally, reshape $\mathbf{h}$ to a 3×3 matrix, denormalize it and divide it by its norm.

Result

$$\mathbf{H}_{DLT} = \begin{bmatrix} 0.0109600331801354 & -2.00096871346898 \times 10^{-5} & -0.984084957989864 \\ 0.000324711572734466 & 0.0106646222405988 & -0.176745394736336 \\ 1.26050707974689 \times 10^{-6} & 1.1124241298911 \times 10^{-7} & 0.0101930644629375 \end{bmatrix}.$$

(e) **Nonlinear estimation**
Use $\mathbf{H}_{DLT}$ and the Sampson corrected points as an initial estimate to an iterative estimation method, specifically the sparse Levenberg-Marquardt algorithm, to determine the Maximum Likelihood estimate of the planar projective transformation that minimizes the reprojection error.

Solution
We first assume scene points $\mathbf{x}$ and then $\mathbf{x}' = \mathbf{H}'\mathbf{x}$ and $\mathbf{x}'' = \mathbf{H}''\mathbf{x}$. Furthermore, we assume $\mathbf{H}' = \mathbf{I}$. As a result, we only need to calculate $\mathbf{H}''$. So for Levenberg-Marquardt algorithm, the parameter vector is $(\hat{\mathbf{h}}''^\top, \hat{\mathbf{x}}_1^\top, \hat{\mathbf{x}}_2^\top, ..., \hat{\mathbf{x}}_n^\top)^\top$, $\hat{\mathbf{h}}''$ is the parameterized $\mathbf{H}''$ and $\hat{\hat{\mathbf{x}}}_i$ is the parameterized scene points. The measurement vector is $(\hat{\hat{\mathbf{x}}}_1'^\top, \hat{\hat{\mathbf{x}}}_2'^\top, ..., \hat{\hat{\mathbf{x}}}_n'^\top, \hat{\hat{\mathbf{x}}}_1''^\top, \hat{\hat{\mathbf{x}}}_2''^\top, ..., \hat{\hat{\mathbf{x}}}_n''^\top)^\top$. The cost is $\epsilon'^\top \Sigma_{\mathbf{x}'}^{-1} \epsilon' + \epsilon'' \Sigma_{\mathbf{x}''}^{-1} \epsilon''$. We assume $\Sigma_{\mathbf{x}_i}$ is an identity matrix.
For Jacobian matrix, here it is very sparse, so there is no need to calculate the full matrix. We only need to calculate three terms:

$$\mathbf{A}_i'' = \frac{\partial \hat{\hat{\mathbf{x}}}_i''}{\partial \hat{\mathbf{h}}''} \quad \mathbf{B}_i' = \frac{\partial \hat{\hat{\mathbf{x}}}_i'}{\partial \hat{\hat{\mathbf{x}}}} \quad \mathbf{B}_i'' = \frac{\partial \hat{\hat{\mathbf{x}}}_i''}{\partial \hat{\hat{\mathbf{x}}}}.$$

For normal equations matrix,$\mathbf{J}^\top \Sigma_{\mathbf{x}}^{-1} \mathbf{J}$ we still only need to calculate three terms:

$$\mathbf{U}'' = \textstyle\sum_i \mathbf{A}_i''^\top \Sigma_{\mathbf{x}_i''}^{-1} \mathbf{A}_i'' \quad \mathbf{V}_i = \mathbf{B}_i'^\top \Sigma_{\mathbf{x}_i'}^{-1} \mathbf{B}_i' + \mathbf{B}_i''^\top \Sigma_{\mathbf{x}_i''}^{-1} \mathbf{B}_i'' \quad \mathbf{W}_i'' = \mathbf{A}_i''^\top \Sigma_{\mathbf{x}_i''}^{-1} \mathbf{B}_i''.$$

The normal equations vector is $(\epsilon_{\mathbf{h}''}^\top, \epsilon_{\hat{\hat{\mathbf{x}}}_1}^\top, \epsilon_{\hat{\hat{\mathbf{x}}}_2}^\top, ..., \epsilon_{\hat{\hat{\mathbf{x}}}_n}^\top)^\top$ where

$$\epsilon_{\mathbf{h}''} = \textstyle\sum_i \mathbf{A}_i''^\top \Sigma_{\mathbf{x}_i}^{-1} \epsilon_i'' \quad \epsilon_{\hat{\hat{\mathbf{x}}}_i} = \mathbf{B}_i'^\top \Sigma_{\mathbf{x}_i'}^{-1} \epsilon_i' + \mathbf{B}_i''^\top \Sigma_{\mathbf{x}_i''}^{-1} \epsilon_i''.$$

The augmented normal equations are

$$\mathbf{S}'' = \mathbf{U}'' + \lambda \mathbf{I} - \sum_i \mathbf{W}_i''^\top (\mathbf{V}_i + \lambda \mathbf{I})^\top \mathbf{W}_i''^\top$$

$$\mathbf{e}'' = \epsilon_{\mathbf{h}''} - \sum_i \mathbf{W}_i''^\top (\mathbf{V}_i + \lambda \mathbf{I})^{-1} \epsilon_{\hat{\hat{\mathbf{x}}}_i}$$

So we can get $\delta_{\mathbf{h}''}$ by solving $\mathbf{S}''\delta_{\mathbf{h}''} = \mathbf{e}''$ and then $\delta_{\hat{\mathbf{x}}_i} = (\mathbf{V}_i + \lambda\mathbf{I})^{-1}(\epsilon_{\hat{\mathbf{x}}_i} - \mathbf{W}_i^{''\top}\delta_{\mathbf{h}''})$. Add these two to the parameter vector to get the new parameter vector. Deparameterize the parameter vector and project the scene points to image 1 and image 2. Calculate the current cost and compare it the with previous cost. If the current cost is less than the previous cost, keep the parameter vector and error vector and set $\lambda = 0.1\lambda$. Iteration until the difference of cost between two iterations is less than 0.0001.

Result
The costs of each iteration are as follow:

Iteration	Cost
0	59.0808
1	58.8801
2	58.8413
3	58.8413

The $\mathbf{H}_{LM}$ is

$$\mathbf{H}_{LM} = \begin{bmatrix} 0.0109600116108005 & -2.11106791008192 \times 10^{-5} & -0.983949182484355 \\ 0.000329426569889086 & 0.0106594313943083 & -0.177500139570516 \\ 1.27286063438928 \times 10^{-6} & 9.85386977683278 \times 10^{-7} & 0.0101908017104713 \end{bmatrix}.$$

Appendix

Listing 1: Part(a)

```
close all; clear; clc;
I0 = imread('../data/price_center20.JPG');
I1 = imread('../data/price_center21.JPG');
%% Corner Detection
fprintf('Detecting corners... ')
win_detect = 9;
win_spr = 9;
eigen_th = 6.5;
[r0,c0,x_f0,y_f0] = CornerCoordinate(I0,win_detect,win_spr,eigen_th);
[r1,c1,x_f1,y_f1] = CornerCoordinate(I1,win_detect,win_spr,eigen_th);
save('../data/c.mat','x_f0','y_f0','x_f1','y_f1');
fprintf('Done\n')
disp(['The number of feature detected in ''price\_center20'' is '...
    ,num2str(numel(r0))]);
disp(['The number of feature detected in ''price\_center21'' is '...
    ,num2str(numel(r1))]);
%% Plot
figure
imshow(I0);
title('price\_center20');
% saveas(gcf,'PriceCenter_20.png');
```

```
figure
imshow(I1);
title('price\_center21');
% saveas(gcf,'PriceCenter_21.png');
figure
imshow(I0);hold on
plot(y_f0,x_f0,'bs','MarkerSize',win_detect);hold off
title('Corner Detection of ''price\_center20''');
% saveas(gcf,'CornerDetection_20.png');
figure
imshow(I1);hold on
plot(y_f1,x_f1,'bs','MarkerSize',win_detect);hold off
title('Corner Detection of ''price\_center21''');
% saveas(gcf,'CornerDetection_21.png');
```

Listing 2: Part(b)

```
close all; clear; clc;
load('../data/c.mat');
I0 = imread('../data/price_center20.JPG');
I1 = imread('../data/price_center21.JPG');
%% Feature Matching
fprintf('Matching features... ')
win_match = 11;
simi_th = 0.7;
dist_th = 0.9;
[X0,Y0,X1,Y1] = FeatureMatch(I0,I1,x_f0,y_f0,x_f1,y_f1,...
                             win_match,simi_th,dist_th);
fprintf('Done\n')
disp(['The number of matched features is ',num2str(numel(X0))]);
x1_inhomo = [Y0';X0'];
x2_inhomo = [Y1';X1'];
save('../data/x.mat','x1_inhomo','x2_inhomo','win_match');
%% Plot
figure
imshow(insertShape(I0,'Line',[Y0 X0 Y1 X1],'Color','blue'));hold on
plot(Y0,X0,'bs','MarkerSize',win_match);hold off
title('Feature Matching of ''price\_center20''');
% saveas(gcf,'FeatureMatch_20.png');
figure
imshow(insertShape(I1,'Line',[Y1 X1 Y0 X0],'Color','blue'));hold on
plot(Y1,X1,'bs','MarkerSize',win_match);hold off
title('Feature Matching of ''price\_center21''');
% saveas(gcf,'FeatureMatch_21.png');
```

Listing 3: Part(c)

```
clear;clc;
```

```
load('../data/x.mat');
x1 = padarray(x1_inhomo,[1  0],1,'post');
x2 = padarray(x2_inhomo,[1  0],1,'post');
n = size(x1,2);
%% mSAC
consensus_min_cost = inf;
max_trials = inf;
trials = 0;
threshold = 0;
tolerance = chi2inv(0.95,2) * 1;
p = 0.99;
s = 3;
k = 1;
rng(2)
while trials < max_trials && consensus_min_cost > threshold
    %% Select a random sample
    i = randperm(n,4)';
    x11 = x1(:,i(1));
    x12 = x1(:,i(2));
    x13 = x1(:,i(3));
    x14 = x1(:,i(4));

    x21 = x2(:,i(1));
    x22 = x2(:,i(2));
    x23 = x2(:,i(3));
    x24 = x2(:,i(4));
    %% Calculate model
    lambda1 = [x11,x12,x13] \ x14;
    H1 = inv([lambda1(1)*x11,lambda1(2)*x12,lambda1(3)*x13]);
    lambda2 = [x21,x22,x23] \ x24;
    H2 = inv([lambda2(1)*x21,lambda2(2)*x22,lambda2(3)*x23]);
    H = H2\H1;
    %% Error for each point
    delta = SampsonError(x1_inhomo,x2_inhomo,H);
    error = sum(delta.^2);
    %% Calculate cost
    cost = sum(error .* (error<=tolerance) + tolerance * (error>tolerance));
    %% Update maximum trials
    if cost < consensus_min_cost
        consensus_min_cost = cost;
        H_min_cost = H;
        inliers = error <= tolerance;
        w = sum(inliers) / n;
        max_trials = log(1-p) / log(1-w^s);
    end
    trials = trials + 1;
```

```
end
disp(['The number of inliers is ',num2str(sum(inliers))]);
disp(['The value of max_trials is ',num2str(max_trials),...
    ', so the the number of attempts is ',num2str(ceil(max_trials))]);
x1_inlier_inhomo = x1_inhomo(:,inliers);
x2_inlier_inhomo = x2_inhomo(:,inliers);
save('../data/x_inlier.mat','x1_inlier_inhomo','x2_inlier_inhomo');
%% Plot
I0 = imread('../data/price_center20.JPG');
I1 = imread('../data/price_center21.JPG');
figure
imshow(insertShape(I0,'Line',[x1_inlier_inhomo',x2_inlier_inhomo'],...
    'Color','blue')); hold on
plot(x1_inlier_inhomo(1,:)',x1_inlier_inhomo(2,:)',...
    'bs','MarkerSize',win_match); hold off
title('Feature Matching of ''price\_center20''');
% saveas(gcf,'FeatureMatch_20_inlier.png');

figure
imshow(insertShape(I1,'Line',[x2_inlier_inhomo',x1_inlier_inhomo'],...
    'Color','blue')); hold on
plot(x2_inlier_inhomo(1,:)',x2_inlier_inhomo(2,:)',...
    'bs','MarkerSize',win_match); hold off
title('Feature Matching of ''price\_center21''');
% saveas(gcf,'FeatureMatch_21_inlier.png');
```

Listing 4: Part(d)

```
clear;clc
load('../data/x_inlier.mat');
x1 = padarray(x1_inlier_inhomo,[1  0],1,'post');
x2 = padarray(x2_inlier_inhomo,[1  0],1,'post');
n = size(x1,2);
%% Data Normalization
% x1
mu_x1 = mean(x1,2);
sigma_x1 = sum(var(x1,0,2));
s_x1 = sqrt(2/sigma_x1);
T1 = [s_x1      0    -mu_x1(1)*s_x1
      0       s_x1   -mu_x1(2)*s_x1
      0        0         1       ];
x1_norm = T1*x1;
% x2
mu_x2 = mean(x2,2);
sigma_x2 = sum(var(x2,0,2));
s_x2 = sqrt(2/sigma_x2);
T2 = [s_x2      0    -mu_x2(1)*s_x2
```

```
        0          s_x2     -mu_x2(2)*s_x2
        0           0           1        ];
x2_norm = T2*x2;
%% DLT H
A = zeros(2*n,9);
for i = 1:n
    x1i = x1_norm(:,i);
    x2i = x2_norm(:,i);
    v = x2i + sign(x2i(1))*norm(x2i,2)*[1;0;0];
    Hv = eye(3) - 2 * (v*v')/(v'*v);
    m = Hv(2:3,:)';
    A(2*i-1:2*i,:) = kron(m',x1i');
end
[~,~,V] = svd(A);
h = V(:,end);
H_DLT = T2\reshape(h,3,3)'*T1;
H_DLT = H_DLT/norm(H_DLT,'fro');
save('../data/H_DLT.mat','H_DLT');

format longg
disp('H_DLT = ')
disp(H_DLT)
```

Listing 5: Part(e)

```
clear;clc
load('../data/x_inlier.mat');
load('../data/H_DLT.mat');
n = size(x1_inlier_inhomo,2);
%% Jacobian
f_A = matlabFunction(jcbA);
f_B1 = matlabFunction(jcbB1);
f_B2 = matlabFunction(jcbB2);
%% Initialize parameter vector
% h_hat
h_bar = reshape(H_DLT',9,[]);
h_hat = parameterization(h_bar);
% x_hat
delta = SampsonError(x1_inlier_inhomo,x2_inlier_inhomo,H_DLT);
x_inhomo = x1_inlier_inhomo + delta(1:2,:);
x_bar = padarray(x_inhomo,[1 0],1,'post');
x_hat = parameterization(x_bar);
% sigma
sigma1 = eye(2*n);
sigma2 = eye(2*n);
% cost
previous_cost = inf;
```

```
% tolerance
tolerance = 0.0001;
%% LM
itr = 0;
disp('LM:')
fprintf('itr\tcost\n')
fprintf('--------------------\n')
% step_1
lambda = 0.001;
x1_hat = deparameterization(x_hat);
x2_hat = reshape(deparameterization(h_hat),3,3)' * deparameterization(x_hat);
x1_hat_inhomo = x1_hat(1:2,:) ./ x1_hat(3,:);
x2_hat_inhomo = x2_hat(1:2,:) ./ x2_hat(3,:);
epsilon1 = reshape(x1_inlier_inhomo - x1_hat_inhomo,[],1);
epsilon2 = reshape(x2_inlier_inhomo - x2_hat_inhomo,[],1);
current_cost = epsilon1'*(sigma1\epsilon1) + epsilon2'*(sigma2\epsilon2);
fprintf('%d\t%.4f\n', itr, current_cost)
% step_2
[A,B1,B2] = jcb(h_hat, x_hat, f_A, f_B1, f_B2);
while tolerance < previous_cost - current_cost || previous_cost < current_cost
    itr = itr+1;
    % step_3_4
    [U,V,W] = NormalEquationsMatrix(A,B1,B2);
    [epsilon_a,epsilon_b] = NormalEquationsVector(A,B1,B2,epsilon1,epsilon2);
    [delta_a,delta_b] = AugmentedNormalEquations(U,V,W,epsilon_a,epsilon_b,lambda);
    % step_5
    h_hat0 = h_hat + delta_a;
    x_hat0 = x_hat + delta_b;
    % step_6
    x1_hat0 = deparameterization(x_hat0);
    x2_hat0 = reshape(deparameterization(h_hat0),3,3)' * deparameterization(x_hat0);
    x1_hat_inhomo0 = x1_hat0(1:2,:) ./ x1_hat0(3,:);
    x2_hat_inhomo0 = x2_hat0(1:2,:) ./ x2_hat0(3,:);
    epsilon10 = reshape(x1_inlier_inhomo - x1_hat_inhomo0,[],1);
    epsilon20 = reshape(x2_inlier_inhomo - x2_hat_inhomo0,[],1);
    % step_7
    previous_cost = current_cost;
    current_cost = epsilon10'*(sigma1\epsilon10) + epsilon20'*(sigma2\epsilon20);
    fprintf('%d\t%.4f\n', itr, current_cost)
    if current_cost < previous_cost
        h_hat = h_hat0;
        x_hat = x_hat0;
        epsilon1 = epsilon10;
        epsilon2 = epsilon20;
        lambda = 0.1*lambda;
        [A,B1,B2] = jcb(h_hat, x_hat, f_A, f_B1, f_B2);
```

```
69      else
70          lambda = 10*lambda;
71      end
72  end
73  fprintf('--------------------\n\n')
74  %% H_LM
75  H_LM = reshape(deparameterization(h_hat0),3,3)';
76  H_LM = H_LM/norm(H_LM,'fro');
77
78  format longg
79  disp('H_LM = ')
80  disp(H_LM)
```

Listing 6: Function

```
1   function [r,c,x_f1,x_f2] = CornerCoordinate(im,win1,win2,threshold)
2   im = rgb2gray(im);
3   A = zeros(size(im));
4   k = [-1 8 0 -8 1]' / 12;
5   imx = double(imfilter(im,k','symmetric'));
6   imy = double(imfilter(im,k,'symmetric'));
7   %% Gradient matrix
8   for i = (win1+1)/2 : size(im,1) - (win1-1)/2
9       for j = (win1+1)/2 : size(im,2) - (win1-1)/2
10          im_win_x = imx(i - (win1-1)/2:i + (win1-1)/2,...
11                         j - (win1-1)/2:j + (win1-1)/2);
12          im_win_y = imy(i - (win1-1)/2:i + (win1-1)/2,...
13                         j - (win1-1)/2:j + (win1-1)/2);
14          N = zeros(2,2);
15          N(1,1) = sum(sum(im_win_x.^2));
16          N(1,2) = sum(sum(im_win_x .* im_win_y));
17          N(2,1) = N(1,2);
18          N(2,2) = sum(sum(im_win_y.^2));
19          N = N/win1^2;
20          lambda = (trace(N) - sqrt(trace(N)^2 - 4*det(N)))/2;
21          if lambda > threshold
22              A(i,j) = lambda;
23          end
24      end
25  end
26  %% Non-maximum suppression
27  B = ordfilt2(A,win1^2,ones(win1,win1));
28  C = (B == A & B ~= 0);
29  [r,c] = find(C);
30  x_f1 = zeros(size(r));
31  x_f2 = zeros(size(r));
32  %% find corner
```

```
for k = 1:numel(r)
    T = zeros(2,2);
    y = zeros(2,1);
    im_win_x = imx(r(k) - (win2-1)/2:r(k) + (win2-1)/2,...
                   c(k) - (win2-1)/2:c(k) + (win2-1)/2);
    im_win_y = imy(r(k) - (win2-1)/2:r(k) + (win2-1)/2,...
                   c(k) - (win2-1)/2:c(k) + (win2-1)/2);
    xx = [r(k) - (win2-1)/2 : r(k) + (win2-1)/2]' .* ones(win2);
    yy = [c(k) - (win2-1)/2 : c(k) + (win2-1)/2] .* ones(win2);
    y(1) = sum(sum( xx.*im_win_x.^2 + yy.*im_win_x.*im_win_y ));
    y(2) = sum(sum( xx.*im_win_x.*im_win_y + yy.*im_win_y.^2 ));
    T(1,1) = sum(sum(im_win_x.^2));
    T(1,2) = sum(sum(im_win_x .* im_win_y));
    T(2,1) = T(1,2);
    T(2,2) = sum(sum(im_win_y.^2));
    x = T\y;
    x_f1(k) = x(1);
    x_f2(k) = x(2);
end
end
```

Listing 7: Function

```
function [X0,Y0,X1,Y1] = FeatureMatch(im0,im1,x0,y0,x1,y1,win,simi_th,dist_th)
im0 = double(rgb2gray(im0));
im1 = double(rgb2gray(im1));
half_win = (win-1)/2;

xx0 = x0+half_win;
xx1 = x1+half_win;
yy0 = y0+half_win;
yy1 = y1+half_win;
im0 = padarray(im0,[half_win,half_win],'symmetric');
im1 = padarray(im1,[half_win,half_win],'symmetric');
%% Xcorrelation matrix
xc = zeros(numel(xx0),numel(xx1));
crd = zeros(numel(xx0),numel(xx1));
for i = 1:numel(xx0)
    X = fix(xx0(i)) - half_win:ceil(xx0(i)) + half_win;
    Y = fix(yy0(i)) - half_win:ceil(yy0(i)) + half_win;
    im0_win = im0(X,Y);
    [X,Y] = meshgrid(X,Y);
    [Xq,Yq] = meshgrid(xx0(i)-half_win:xx0(i)+half_win,...
                       yy0(i) - half_win:yy0(i) + half_win);
    im0_win_intep = interp2(X,Y,im0_win,Xq,Yq,'linear');
    for j = 1:numel(xx1)
        X = fix(xx1(j)) - half_win:ceil(xx1(j)) + half_win;
```

```
25          Y = fix(yy1(j)) - half_win:ceil(yy1(j)) + half_win;
26          im1_win = im1(X,Y);
27          [X,Y] = meshgrid(X,Y);
28          [Xq,Yq] = meshgrid(xx1(j)-half_win:xx1(j)+half_win,...
29                             yy1(j) - half_win:yy1(j) + half_win);
30          im1_win_intep = interp2(X,Y,im1_win,Xq,Yq,'linear');
31          xc(i,j) = corr2(im0_win_intep,im1_win_intep);
32      end
33  end
34  %% One-to-One Matching
35  while max(xc(:)) > simi_th
36      [r,c] = find(xc == max(xc(:)));
37      i = r(1);
38      j = c(1);
39      mx = xc(i,j);
40      xc(i,j) = -1;
41      next_mx = max(max(xc(i,:),[],2),max(xc(:,j)));
42      if (1-mx) < (1-next_mx)*dist_th
43          crd(i,j) = 1;
44      end
45      xc(i,:) = -1;
46      xc(:,j) = -1;
47  end
48  [i,j] = find(crd);
49  X0 = x0(i);
50  Y0 = y0(i);
51  X1 = x1(j);
52  Y1 = y1(j);
53  w = abs(X0-X1);
54  d = abs(Y0-Y1);
55  n = find(w > 25 | d > 160);
56  X0(n) = [];
57  Y0(n) = [];
58  X1(n) = [];
59  Y1(n) = [];
60  end
```

Listing 8: Function

```
1  function v = parameterization(v_bar)
2  v_bar = v_bar ./ sqrt(sum(v_bar.^2));
3  a = v_bar(1,:);
4  b = v_bar(2:end,:);
5  v = 2*acos(a) ./ sin(acos(a)) .* b;
6  v_norm = sqrt(sum(v.^2));
7  v = (1 - 2*pi ./ v_norm .* ceil((v_norm-pi)/2/pi)) .* v;
8  end
```

Listing 9: Function

```
function v_bar = deparameterization(v)
v_norm = sqrt(sum(v.^2));
a = cos(v_norm/2);
b = sin(v_norm/2) ./ v_norm .* v;
v_bar = [a;b];
end
```

Listing 10: Function

```
function f_A = jcbA
syms h1 h2 h3 h4 h5 h6 h7 h8 x y
h_hat = [h1 h2 h3 h4 h5 h6 h7 h8].';
x_hat = [x y].';
H2 = reshape(deparameterization(h_hat),3,3).';
x_bar = deparameterization(x_hat);
x2_hat = H2 * x_bar;
x2_hat_inhomo = x2_hat(1:2,:) ./ x2_hat(3,:);
f_A([h1 h2 h3 h4 h5 h6 h7 h8 x y]) = jacobian(x2_hat_inhomo,h_hat);
end
```

Listing 11: Function

```
function f_B1 = jcbB1
syms x y
x_hat = [x y].';
x_bar = deparameterization(x_hat);
x1_hat = x_bar;
x1_hat_inhomo = x1_hat(1:2,:) ./ x1_hat(3,:);
f_B1([x y]) = jacobian(x1_hat_inhomo,x_hat);
end
```

Listing 12: Function

```
function f_B2 = jcbB2
syms h1 h2 h3 h4 h5 h6 h7 h8 x y
h_hat = [h1 h2 h3 h4 h5 h6 h7 h8].';
x_hat = [x y].';
H2 = reshape(deparameterization(h_hat),3,3).';
x_bar = deparameterization(x_hat);
x2_hat = H2 * x_bar;
x2_hat_inhomo = x2_hat(1:2,:) ./ x2_hat(3,:);
f_B2([h1 h2 h3 h4 h5 h6 h7 h8 x y]) = jacobian(x2_hat_inhomo,x_hat);
end
```

Listing 13: Function

```
function [A,B1,B2] = jcb(h_hat, x_hat, f_A, f_B1, f_B2)
```

```
n = size(x_hat,2);
A = zeros(2,8,n);
B1 = zeros(2,2,n);
B2 = zeros(2,2,n);
h1 = h_hat(1);
h2 = h_hat(2);
h3 = h_hat(3);
h4 = h_hat(4);
h5 = h_hat(5);
h6 = h_hat(6);
h7 = h_hat(7);
h8 = h_hat(8);
for i = 1:n
    x = x_hat(1,i);
    y = x_hat(2,i);
    A(:,:,i) = double(f_A(h1, h2, h3, h4, h5, h6, h7, h8, x, y));
    B1(:,:,i) = double(f_B1(x, y));
    B2(:,:,i) = double(f_B2(h1, h2, h3, h4, h5, h6, h7, h8, x, y));
end
end
```

Listing 14: Function

```
function delta = SampsonError(x1_inhomo,x2_inhomo,H)
n = size(x1_inhomo,2);
delta = zeros(4,n);
for i = 1:n
    epsilon = [-(x1_inhomo(1,i)*H(2,1)+x1_inhomo(2,i)*H(2,2)+H(2,3))...
               + x2_inhomo(2,i)*(x1_inhomo(1,i)*H(3,1)+x1_inhomo(2,i)*H(3,2)+H(3,3));
                 x1_inhomo(1,i)*H(1,1)+x1_inhomo(2,i)*H(1,2)+H(1,3)....
               - x2_inhomo(1,i)*(x1_inhomo(1,i)*H(3,1)+x1_inhomo(2,i)*H(3,2)+H(3,3))];
    J = [-H(2,1)+x2_inhomo(2,i)*H(3,1), -H(2,2)+x2_inhomo(2,i)*H(3,2),...
         0, x1_inhomo(1,i)*H(3,1)+x1_inhomo(2,i)*H(3,2)+H(3,3);
         H(1,1)-x2_inhomo(1,i)*H(3,1),H(1,2)-x2_inhomo(1,i)*H(3,2),...
         -(x1_inhomo(1,i)*H(3,1)+x1_inhomo(2,i)*H(3,2)+H(3,3)),
0];
    delta(:,i) = -J'*(J*J'\epsilon);
end
end
```

Listing 15: Function

```
function [U,V,W] = NormalEquationsMatrix(A,B1,B2)
n = size(A,3);
U = zeros(8,8);
V = zeros(2,2,n);
W = zeros(8,2,n);
```

```
6  for i = 1:n
7      U = U + A(:,:,i)' * A(:,:,i);
8      V(:,:,i) = B1(:,:,i)'*B1(:,:,i) + B2(:,:,i)'*B2(:,:,i);
9      W(:,:,i) = A(:,:,i)'*B2(:,:,i);
10 end
11 end
```

Listing 16: Function

```
1  function [epsilon_a,epsilon_b] = NormalEquationsVector(A,B1,B2,epsilon1,epsilon2)
2  n = size(A,3);
3  epsilon_a = zeros(8,1);
4  epsilon_b = zeros(2,n);
5  for i = 1:n
6      epsilon_a = epsilon_a + A(:,:,i)'*epsilon2(2*i-1:2*i);
7      epsilon_b(:,i) = B1(:,:,i)'*epsilon1(2*i-1:2*i) + B2(:,:,i)'*epsilon2(2*i-1:2*i);
8  end
9  end
```

Listing 17: Function

```
1  function [delta_a,delta_b] = AugmentedNormalEquations(U,V,W,epsilon_a,epsilon_b,lambda)
2  n = size(V,3);
3  delta_b = zeros(2,n);
4  S = U + lambda*eye(8);
5  e = epsilon_a;
6  for i = 1:n
7      S = S - W(:,:,i)*((V(:,:,i) + lambda*eye(2))\W(:,:,i)');
8      e = e - W(:,:,i)*((V(:,:,i) + lambda*eye(2))\epsilon_b(:,i));
9  end
10 delta_a = S\e;
11 for i = 1:n
12     delta_b(:,i) = (V(:,:,i) + lambda*eye(2))\(epsilon_b(:,i) - W(:,:,i)'*delta_a);
13 end
14 end
```