ECE 661: Homework 9

Xingguang Zhang

(Fall 2020)

1. Theory Question

In Lecture 18, we showed that the image of the Absolute Conic Ω_∞ is given by $\omega = K^{-T}K^{-1}$. As you know, the Absolute Conic resides in the plane π_∞ at infinity. Does the derivation we went through in Lecture 18 mean that you can actually see ω in a camera image? Give reasons for both 'yes' and 'no' answers. Also, explain in your own words the role played by this result in camera calibration.

–No, we can't actually see ω in the camera image. The pixels of the absolute conic is imaginary because $K^{-T}K^{-1}$ is positive definite. The image of the absolute conic is invariant to the rotation and translation R, its relative position to a moving camera is constant, and only a function of the intrinsic parameter K of the camera. This property can help us to find the intrinsic parameter of a camera.

2. Salient Point Detection

The salient point is defined as corners of the square pattern. We will first detect all 80 corners in the given 40 images and my own image set containing 21 images.

2.1. Canny Edge Detection

I used Canny edge detector cv2.Canny() from OpenCV library. To do so, I first convert the original image into gray scale images and then filter it with a 3×3 Gaussian kernel to reduce the noise on the image.

For both of the given images and my own images, the high threshold and low threshold for the Canny detector are set as 383 and 255.

2.2. Hough Transform for line detection

I used cv2.HoughLines() function from OpenCV library to extract lines by Hough transform from the edges detected by Canny detector. I choose the parameters by several trials to extract the reasonable amount of lines (around 20 to 30).

The distance resolution of the accumulator in pixels is set to be 1 for all given and my own images. For all given images, the angle resolution of the accumulator in radians is set to 0.5 and the threshold is set to 50. For my own images, these two parameters are set as 0.6 and 70, respectively.

We will get some redundant lines from above processes, but we want only one line for a set of edges that should be collinear on the grid image. The next step is to extract true lines from the raw Hough function output, which is done by first identifying the horizontal and vertical lines, then sorting them by the distance from the original point, finally grouping the lines that should be compressed and take the average of them to get the refined output.

The range of angle for vertical lines are $-\frac{\pi}{4} \sim \frac{\pi}{4}$ and $\frac{3\pi}{4} \sim \frac{5\pi}{4}$ from the y-axis. If we denote angle by θ, we have $cos^2\theta > 0.5$ for vertical lines and others for horizontal lines. There should be 8 vertical lines and 10 horizontal lines, so we need to compress the vertical and horizontal lines respectively.

We determine which lines should be the same real line by the distances among all lines. In sorted order, if a line and its neighbor has a distance less than threshold, I put them in the same group to be averaged. Since the grid patterns have different scales in different views, I set a ratio r to determine the threshold. In both vertical and horizontal line groups, we first measure the size of the pattern by take the difference between the farthest and nearest lines and measure the average distance between two real lines according to it. The threshold is defined by the ratio of the average distance. For the given image, I set $r = 0.25$ and for my own images, I set $r = 0.28$.

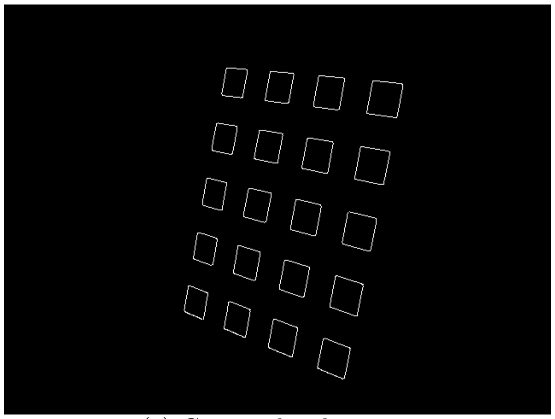
2.3. Extracting intersections

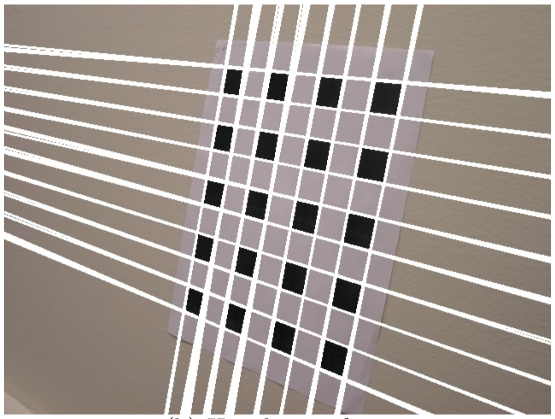
The salient points for calibration are intersections between all vertical lines and horizontal lines. We first represent lines by start and end points, and take their cross product in the homogeneous space as the line representation. We can further calculate all intersections using the cross product of the associated line pairs in homogeneous space.

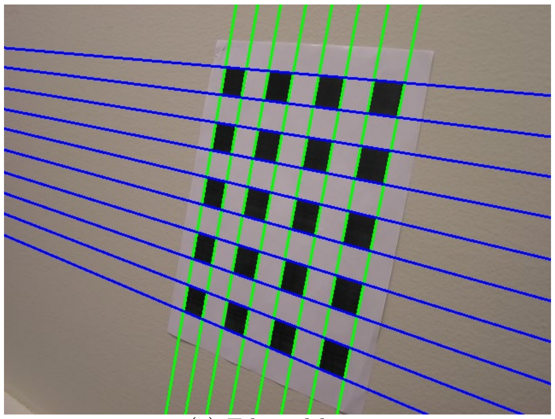
After all processes above, I can successfully extract 80 intersections from 38 images out of 40 in the given image set and 18 images out of 21 in my own image set.

2.4. Results

The result for steps above on images 1 and 19.


(a) Canny edge detection


(b) Hough transform


(c) Filtered lines

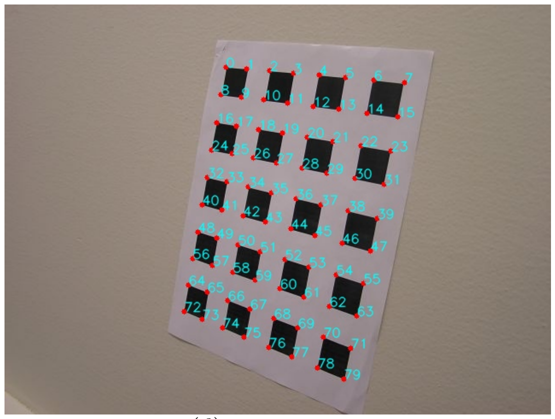
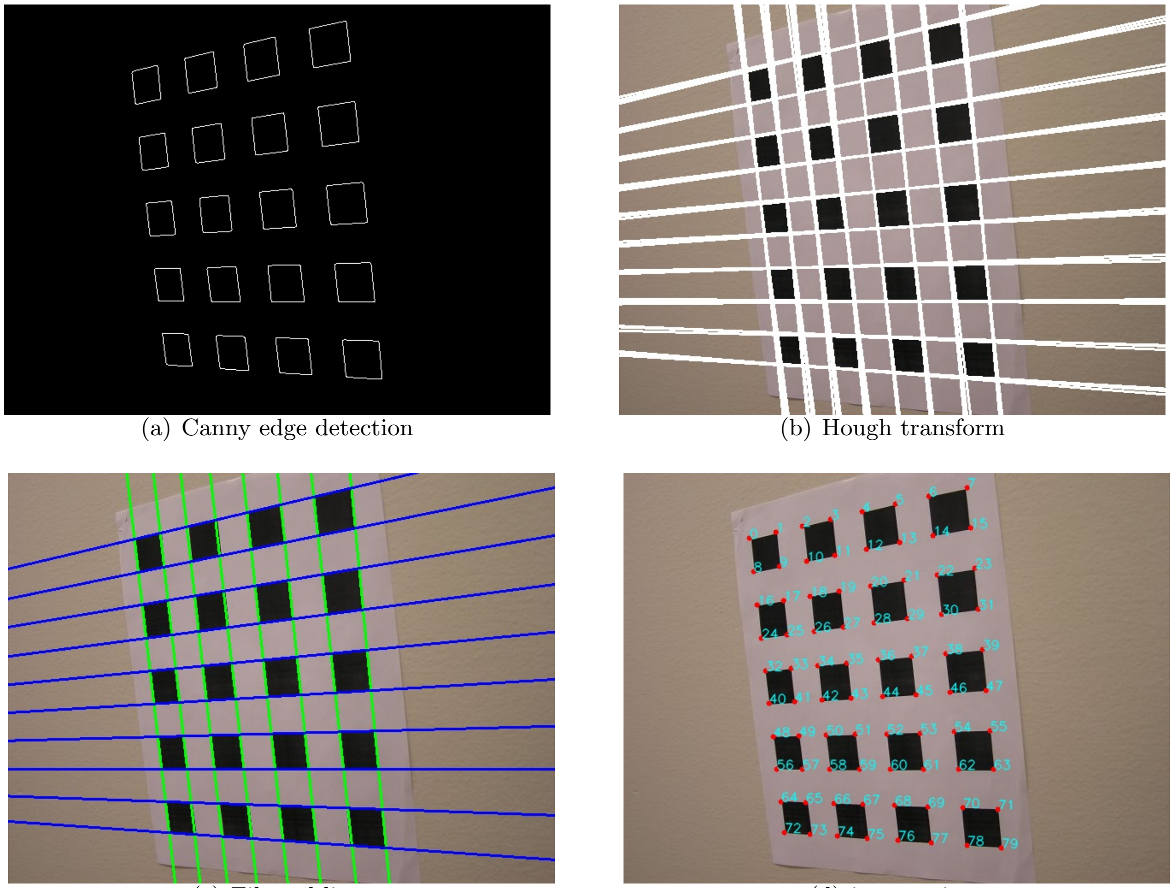

(d) intersections

Figure 1: Intersections extraction from image 1

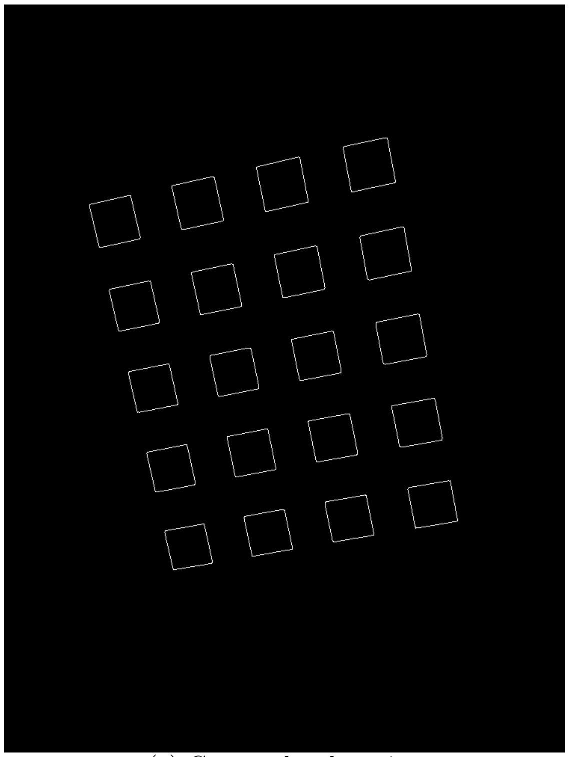

(a) Canny edge detection

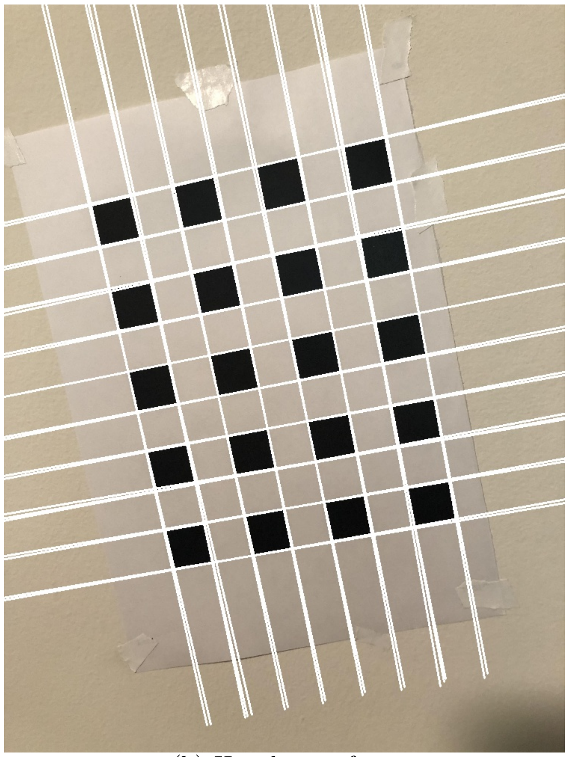
(b) Hough transform

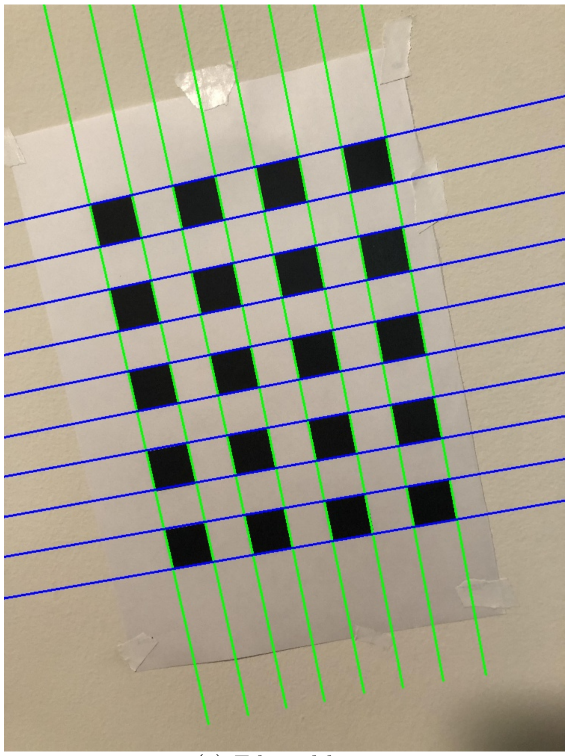
(c) Filtered lines

(d) intersections

Figure 2: Intersections extraction from image 19


(a) Canny edge detection


(b) Hough transform


(c) Filtered lines

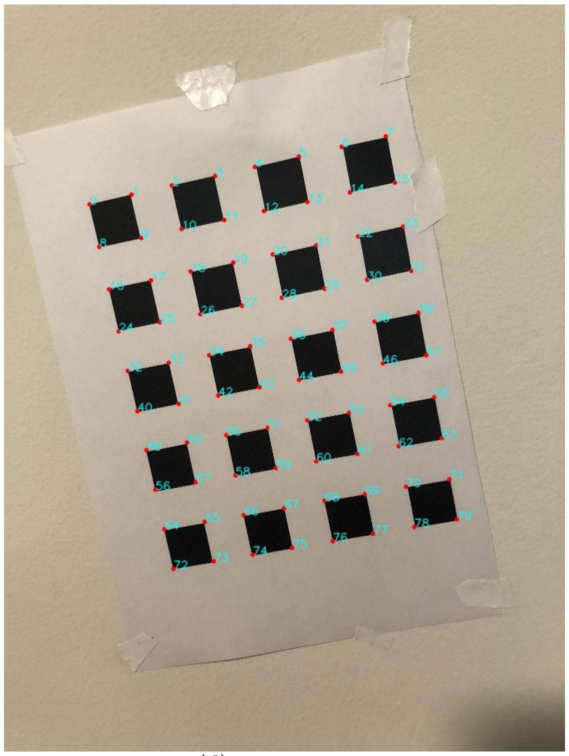
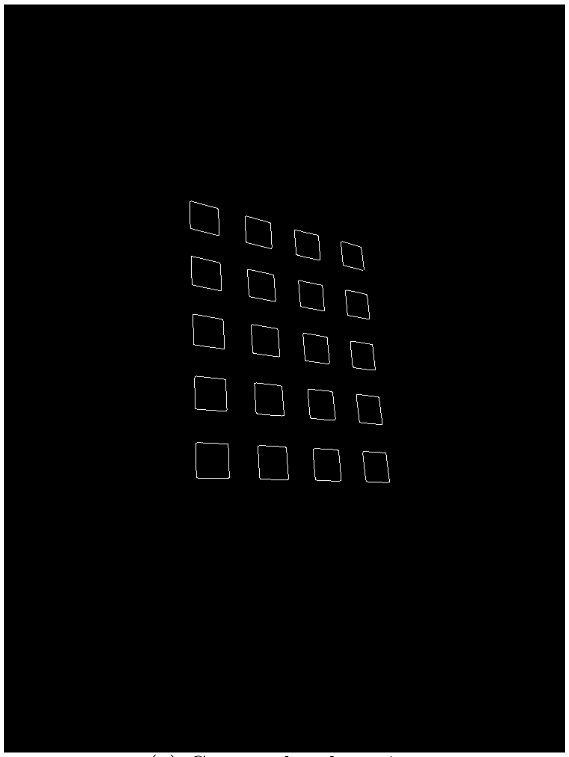
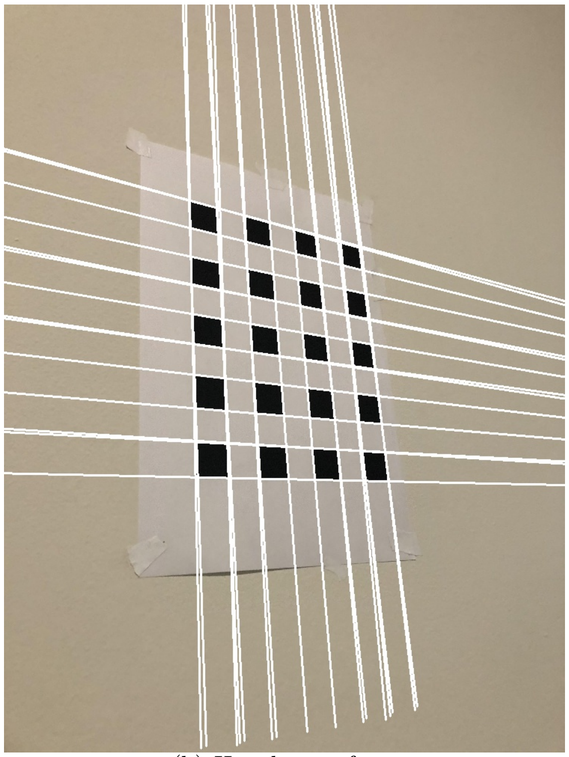
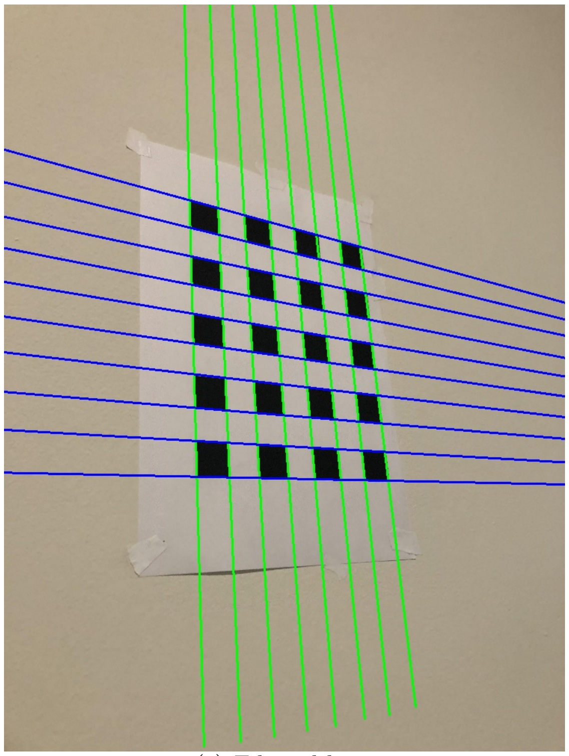

(d) intersections

Figure 3: Intersections extraction from image 1


(a) Canny edge detection


(b) Hough transform


(c) Filtered lines

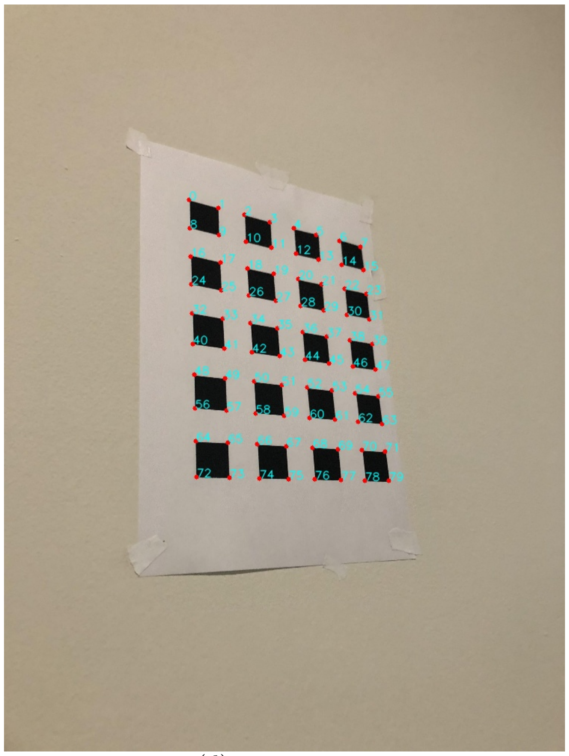

(d) intersections

Figure 4: Intersections extraction from image 7

3. Zhang's Algorithm

The pipeline of Zhang's algorithm contains 3 parts to extract the camera parameters step by step: the intrinsic parameter, the extrinsic parameter and the radial distortion. We will apply Levenberg-Marquadt (LM) non-linear optimization to refine the calibration after these steps.
We first define an ideal grid pattern as the ground truth coordinates of the pattern on the wall, and we compute the homography transform matrices H_i which can transform the ground truth pattern to images under different views.

3.1. Calculate the intrinsic parameter with the absolute conic

We assume the plane where the printed pattern lays on is the plane of $z = 0$, a salient point on it is $\vec{x} = \begin{bmatrix} x & y & 0 & w \end{bmatrix}^T$, its image will be:

$$\vec{y} = H\vec{x} = K[R|\vec{t}]\begin{bmatrix} x \\ y \\ 0 \\ w \end{bmatrix}$$

Where K is the intrinsic matrix, R is the rotation matrix and $\vec{t}$ is the translation vector. The image of two circular points $\vec{I}$ and $\vec{J}$ are $H\vec{I} = \vec{h}_1 + i\vec{h}_2$ and $H\vec{J} = \vec{h}_1 - i\vec{h}_2$, where $H = [\vec{h}_1, \vec{h}_2, \vec{h}_3]$, $\vec{I} = [1, i, 0]^T$ and $\vec{J} = [1, -i, 0]^T$. Since they are on the absolute conic ω, we have

$$(H\vec{I})^T\omega(H\vec{I}) = (\vec{h}_1 + i\vec{h}_2)^T\omega(\vec{h}_1 + i\vec{h}_2) = 0$$
$$(H\vec{J})^T\omega(H\vec{J}) = (\vec{h}_1 - i\vec{h}_2)^T\omega(\vec{h}_1 - i\vec{h}_2) = 0$$

That leads to $\vec{h}_1^T\omega\vec{h}_1 - \vec{h}_2^T\omega\vec{h}_2 = 0$ and $\vec{h}_1^T\omega\vec{h}_2 = 0$, we can convert them into:

$$\begin{bmatrix} h_{11} & h_{21} & h_{31} \end{bmatrix}\begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix}\begin{bmatrix} h_{12} \\ h_{22} \\ h_{32} \end{bmatrix} = 0$$

and

$$\begin{bmatrix} h_{11} & h_{21} & h_{31} \end{bmatrix}\begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix}\begin{bmatrix} h_{11} \\ h_{21} \\ h_{31} \end{bmatrix} - \begin{bmatrix} h_{12} & h_{22} & h_{32} \end{bmatrix}\begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix}\begin{bmatrix} h_{12} \\ h_{22} \\ h_{32} \end{bmatrix} = 0$$

Since ω is symmetric, there are only 6 parameters we need to solve, as Zhang illustrated, we can convert the above functions into:

$$V\vec{b} = \begin{bmatrix} \vec{v}_{12}^T \\ \vec{v}_{11}^T - \vec{v}_{22}^T \end{bmatrix}\vec{b} = 0$$

Where

$$v_{ij} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}$$

For each H we got, we can make two equations, then we stack all equations together to solve $\vec{b}$, which should be the feature vector associated with the smallest singular value of matrix V. Finally we can reconstruct the absolute conic ω, which is also

$$\omega = K^{-T}K^{-1}$$

The intrinsic matrix K is contructed by

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

According to Zhang's technical report, these 5 unknowns can be calculated by:

$$\begin{aligned} y_0 &= \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \\ \lambda &= \omega_{33} - \frac{\omega_{13}^2 + y_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}} \\ \alpha_x &= \sqrt{\frac{\lambda}{\omega_{11}}} \\ \alpha_y &= \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \\ s &= -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \\ x_0 &= \frac{sy_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda} \end{aligned}$$

3.2. Compute the extrinsic parameter for each camera pose

The homography transform matrix H on the $z = 0$ can be represented as $H = K[R|\vec{t}]$, also $K^{-1}[\vec{h}_1, \vec{h}_2, \vec{h}_3] = [\vec{r}_1, \vec{r}_2, \vec{t}]$, as the rotation matrix has to be orthonormal, we need to normalize R and $\vec{t}$ by factor $\xi = \left\| K^{-1}\vec{h}_1 \right\|^{-1}$:

$$\begin{aligned} \left[\vec{r}_1, \vec{r}_2, \vec{t}\right] &= \xi K^{-1}[\vec{h}_1, \vec{h}_2, \vec{h}_3] \\ \vec{r}_3 &= \xi \vec{r}_1 \times \vec{r}_2 \end{aligned}$$

Since the rotation matrix in the physical world can only have 3 DoF, matrix R has another representation $\vec{w}$, which is

$$\vec{w} = \frac{\varphi}{2sin\varphi} \begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix}$$

Where

$$\varphi = cos^{-1}\frac{trace(R) - 1}{2}$$

To transform $\vec{w}$ back to R in the LM refinement, we first construct matrix

$$W = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix}$$

Then R is:

$$R = e^W = I_{3\times3} + \frac{sin\varphi}{\varphi}W + \frac{1 - cos\varphi}{\varphi^2}W \quad \text{where} \quad \varphi = \|\vec{w}\|$$

3.3. Parameter refinement

We've calculated the camera intrinsic matrix K and extrinsic matrix $[R|\vec{t}]$ by above steps. But the results are not optimal yet, we then apply Levenberg-Marquadt (LM) non-linear optimization to refine the calibration. The key element for LM algorithm is the cost function.

We denote the j-th salient point on the ground truth pattern as $\vec{x}_{m,j}$, and the extracted j-th salient point on image from the i-th camera position as $\vec{x}_{i,j}$. Our goal is to minimize the cost function

$$d_{geom}^2 = \left\|\vec{X} - \vec{f}(\vec{p})\right\|^2 = \sum_i \sum_j \left\|\vec{x}_{i,j} - \hat{\vec{x}}_{i,j}\right\| \tag{1}$$

Where

$$\hat{\vec{x}}_{i,j} = K[R_i|\vec{t_i}]\vec{x}_{m,j} \tag{2}$$

And R_i, K are represented by vectors W_i's and $[\alpha_x, \alpha_y, s, x_0, y_0]$ in order to restrict the DoF of the optimization.

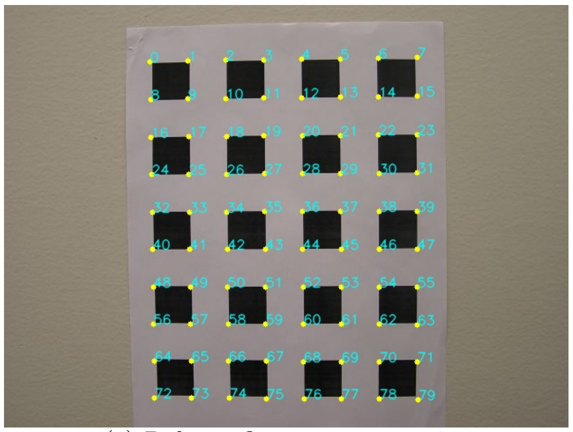
3.3.1. Project corners of the ground truth pattern to the given dataset images

I projected the corners of the calibration pattern to every image, and measured the average error(distance), the variance and the maximal distance. The error declined significantly after the LM refinement. For the given image set,

stage	average error	variance	largest error
Original	1.02660864	0.5971901	5.80386693
LM refinement without radial distortion	0.61186023	0.12041647	2.68659401
LM refinement with radial distortion	0.54191563	0.11181598	2.64409102

Table 1: Error reduction on the given image set by the LM refinement

Project the ground truth pattern on the 13th and 39th images, we can clearly view that the LM optimization did reduce the projection error obviously.


(a) Before refinement: error 1.406

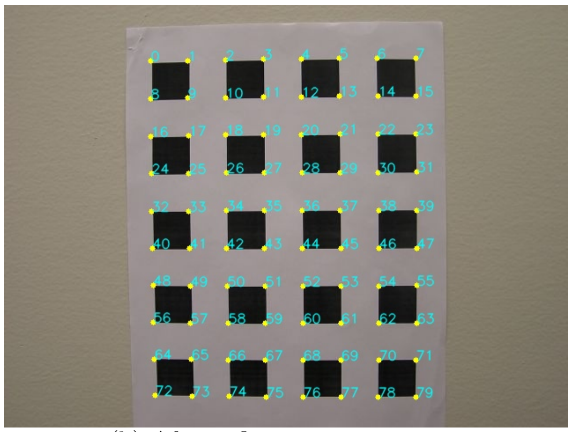

(b) After refinement: error 0.663

Figure 5: The LM refinement result on image 13

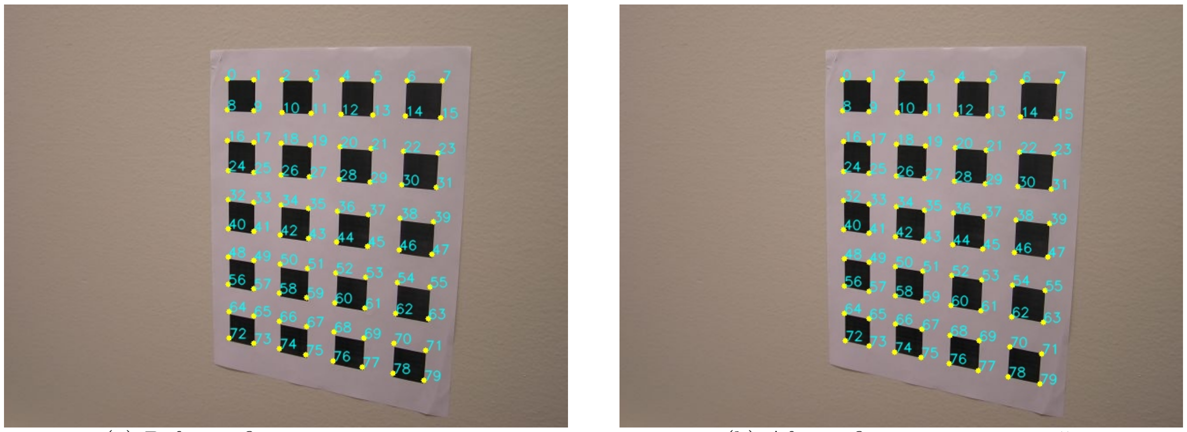

(a) Before refinement: error 2.328 (b) After refinement: error 0.647

Figure 6: The LM refinement result on image 39

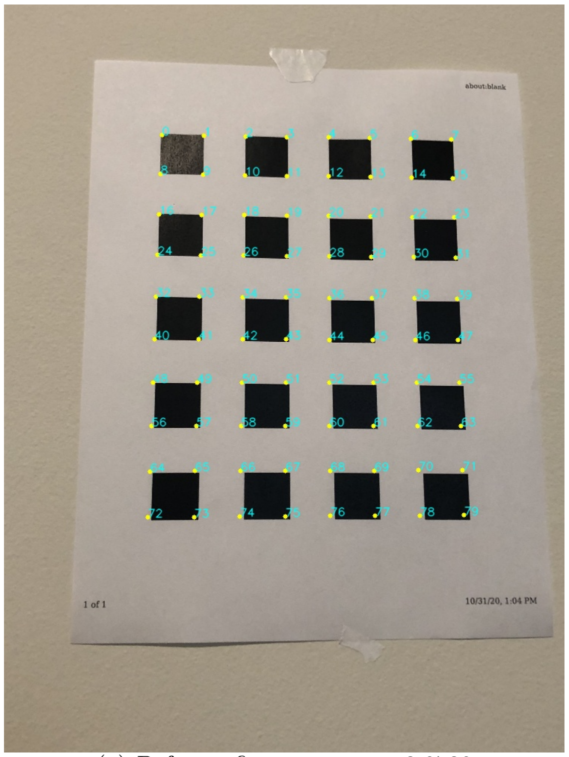
3.3.2. Project corners of the ground truth pattern to my own images

The decreasing of the error and variance on my own image set:

stage	average error	variance	largest error
Original	8.5807599	57.31564843	44.83856173
LM refinement without radial distortion	1.88021248	1.82869572	12.67971128
LM refinement with radial distortion	1.82835229	1.86785082	12.28723254

Table 2: Error reduction on my own image set by the LM refinement

Project the ground truth pattern on the 6th and 9th image:


(a) Before refinement: error 3.9730

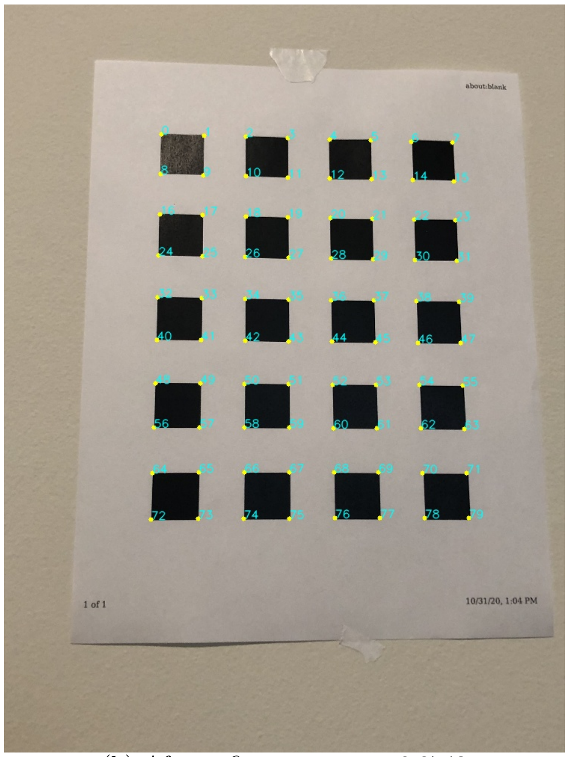
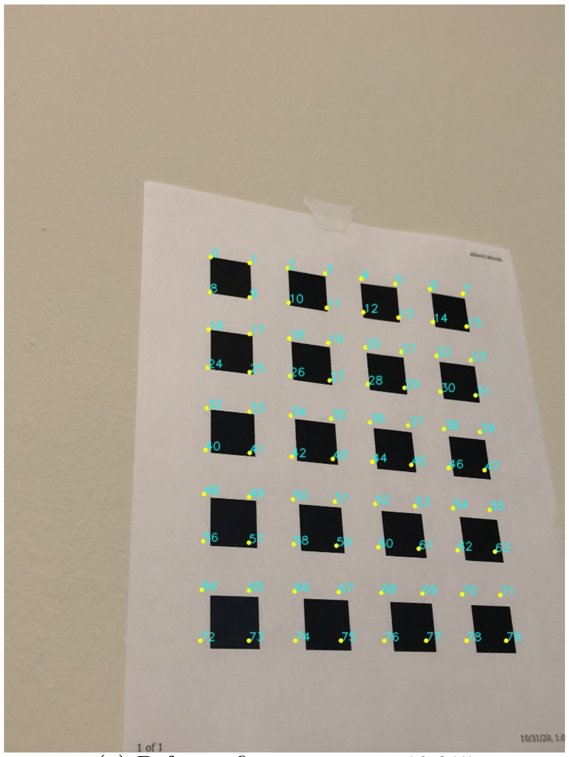

(b) After refinement: error 0.8743

Figure 7: The LM refinement result on image 6


(a) Before refinement: error 10.015

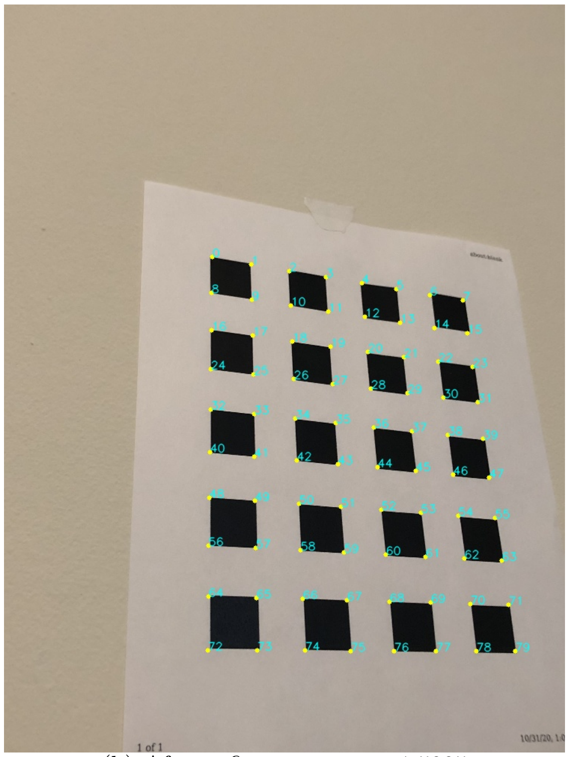

(b) After refinement: error 1.5995

Figure 8: The LM refinement result on image 9

3.4. The radial distortion

The linear model we discussed above are based on the assumption that our camera works like the pinhole camera, but in realistic, it only applies when the focal length is relatively long. But actually our camera for calibration used a short focal length for imaging the patterns, so we need to consider the radial distortion. A good approximation to deal with the radial distortion is model it using two parameters k_1 and k_2 and simply adjust our projected coordinates by:

$$\hat{x}_{rad} = \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4]$$
$$\hat{y}_{rad} = \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4]$$

Where $\hat{x}, \hat{y}$ is the projected pattern without the radial distortion, (x_0, y_0) are the currently avaiable for the principal point on the image, and $r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$. To measure the radial distortion, we simple add this step after step (2) and replace $\hat{\vec{x}}_{i,j}$ by $\hat{\vec{x}}_{rad;i,j}$ in the step (1) and run the LM algorithm again.

The estimated radial distortion for the given dataset is:

$$k_1 = -2.787 \times 10^{-7} \quad k_2 = -1.674 \times 10^{-12}$$

And it for my own camera is:

$$k_1 = 1.179 \times 10^{-7} \quad k_2 = -3.639 \times 10^{-13}$$

3.5. Results

I will show the camera intrinsic matrices of both cameras under different stages of computing, including the results by Zhang's algorithm, the results refined by the LM optimization, with or without the consideration of the radial distortion. For the extrinsic matrices, I will show the rotation and translation matrices of the 1st, the 9th, the 21st and the 39th views associated with the given images, and the 4th, 6th, 9th, 18th views associated with my own images.

3.5.1. Given Dataset

The intrinsic matrix K of the given camera before refinement is:

$$K = \begin{vmatrix} 715.3 & 1.352 & 321.5 \\ 0 & 713.2 & 241.3 \\ 0 & 0 & 1 \end{vmatrix}$$

After the LM refinement without consideration of radial distortion:

$$K = \begin{vmatrix} 721.3 & 2.213 & 321.6 \\ 0 & 719.1 & 242.0 \\ 0 & 0 & 1 \end{vmatrix}$$

Consider the radial distortion:

$$K = \begin{vmatrix} 726.8 & 2.241 & 319.6 \\ 0 & 724.9 & 243.8 \\ 0 & 0 & 1 \end{vmatrix}$$

The rotation and translation parameters of the position 1:

$$[R_1|\vec{t_1}] = \left|\begin{array}{ccc|c} 0.789 & -0.184 & 0.585 & -18.86 \\ 0.202 & 0.979 & 0.035 & -51.45 \\ -0.579 & 0.090 & 0.810 & 217.7 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_1|\vec{t_1}] = \left|\begin{array}{ccc|c} 0.784 & -0.184 & 0.593 & -18.74 \\ 0.202 & 0.979 & 0.036 & -51.32 \\ -0.587 & 0.092 & 0.805 & 218.2 \end{array}\right|$$

Consider the radial distortion:

$$[R_1|\vec{t_1}] = \left|\begin{array}{ccc|c} 0.782 & -0.183 & 0.596 & -18.10 \\ 0.204 & 0.978 & 0.034 & -51.87 \\ -0.589 & 0.095 & 0.802 & 218.7 \end{array}\right|$$

The rotation and translation parameters of the position 9:

$$[R_9|\vec{t_9}] = \left|\begin{array}{ccc|c} 0.897 & -0.081 & 0.435 & -21.80 \\ -0.166 & 0.849 & 0.501 & -38.91 \\ -0.410 & -0.521 & 0.748 & 237.7 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_9|\vec{t_9}] = \left|\begin{array}{ccc|c} 0.886 & -0.091 & 0.455 & -21.58 \\ -0.171 & 0.848 & 0.501 & -38.64 \\ -0.432 & -0.522 & 0.736 & 237.7 \end{array}\right|$$

Consider the radial distortion:

$$[R_9|\vec{t_9}] = \left|\begin{array}{ccc|c} 0.884 & -0.093 & 0.458 & -20.87 \\ -0.170 & 0.849 & 0.500 & -39.19 \\ -0.436 & -0.520 & 0.735 & 238.2 \end{array}\right|$$

The rotation and translation parameters of the position 21:

$$[R_{21}|\vec{t_{21}}] = \left|\begin{array}{ccc|c} 0.782 & -0.182 & 0.597 & -38.87 \\ 0.167 & 0.983 & 0.082 & -55.16 \\ -0.601 & 0.036 & 0.798 & 193.38 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_{21}|\vec{t_{21}}] = \left|\begin{array}{ccc|c} 0.788 & -0.182 & 0.588 & -39.15 \\ 0.172 & 0.982 & 0.074 & -55.99 \\ -0.591 & 0.043 & 0.806 & 196.3 \end{array}\right|$$

Consider the radial distortion:

$$[R_{21}|\vec{t_{21}}] = \left|\begin{array}{ccc|c} 0.788 & -0.182 & 0.589 & -38.64 \\ 0.174 & 0.982 & 0.071 & -56.51 \\ -0.591 & 0.046 & 0.805 & 196.3 \end{array}\right|$$

The rotation and translation parameters of the position 39:

$$[R_{39}|\vec{t_{39}}] = \left|\begin{array}{ccc|c} 0.896 & -0.015 & 0.444 & -19.44 \\ 0.103 & 0.980 & -0.173 & -19.44 \\ -0.433 & 0.201 & 0.879 & -205.5 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_{39}|\vec{t_{39}}] = \left|\begin{array}{ccc|c} 0.886 & -0.014 & 0.462 & -19.26 \\ 0.105 & 0.980 & -0.171 & -44.82 \\ -0.451 & 0.200 & 0.870 & 205.9 \end{array}\right|$$

Consider the radial distortion:

$$[R_{39}|\vec{t_{39}}] = \left|\begin{array}{ccc|c} 0.884 & -0.013 & 0.467 & -18.67 \\ 0.106 & 0.979 & -0.173 & -45.34 \\ -0.455 & 0.203 & 0.867 & 206.2 \end{array}\right|$$

3.5.2. My Dataset

The intrinsic matrix K of my own camera before refinement is:

$$K = \begin{vmatrix} 1019.7 & -12.1 & 377.3 \\ 0 & 1018.8 & 505.7 \\ 0 & 0 & 1 \end{vmatrix}$$

After the LM refinement without consideration of radial distortion:

$$K = \begin{vmatrix} 1012.1 & -1.878 & 367.7 \\ 0 & 1012.2 & 510.7 \\ 0 & 0 & 1 \end{vmatrix}$$

Consider the radial distortion:

$$K = \begin{vmatrix} 1009.7 & -1.883 & 371.4 \\ 0 & 1009.1 & 509.6 \\ 0 & 0 & 1 \end{vmatrix}$$

The rotation and translation parameters of the position 4:

$$[R_4|\vec{t_4}] = \left|\begin{array}{ccc|c} 0.937 & 0.072 & 0.342 & -36.31 \\ 0.075 & 0.913 & -0.400 & -40.32 \\ -0.342 & 0.401 & 0.850 & 166.0 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_4|\vec{t_4}] = \left|\begin{array}{ccc|c} 0.943 & 0.058 & 0.325 & -37.82 \\ 0.070 & 0.926 & -0.371 & -41.76 \\ -0.323 & 0.373 & 0.870 & 167.0 \end{array}\right|$$

Consider the radial distortion:

$$[R_4|\vec{t_4}] = \left|\begin{array}{ccc|c} 0.945 & 0.057 & 0.322 & -36.90 \\ 0.069 & 0.927 & -0.369 & -41.60 \\ -0.319 & 0.371 & 0.872 & 167.5 \end{array}\right|$$

The rotation and translation of the position 6:

$$[R_6|\vec{t_6}] = \left|\begin{array}{ccc|c} 0.999 & 0.006 & -0.044 & -30.39 \\ 0.001 & 0.985 & 0.170 & -59.23 \\ -0.044 & -0.170 & 0.984 & 182.2 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_6|\vec{t_6}] = \left|\begin{array}{ccc|c} 0.998 & 0.003 & -0.060 & -27.91 \\ 0.007 & 0.985 & 0.172 & -59.83 \\ -0.059 & -0.172 & 0.983 & 180.03 \end{array}\right|$$

Consider the radial distortion:

$$[R_6|\vec{t_6}] = \left|\begin{array}{ccc|c} 0.998 & 0.004 & -0.064 & -28.58 \\ 0.008 & 0.984 & 0.179 & -59.62 \\ 0.063 & -0.179 & 0.982 & 180.6 \end{array}\right|$$

The rotation and translation of the position 9:

$$[R_9|\vec{t_9}] = \left|\begin{array}{ccc|c} 0.954 & 0.027 & -0.299 & -18.96 \\ 0.096 & 0.917 & 0.388 & -30.97 \\ 0.285 & -0.399 & 0.872 & 187.9 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_9|\vec{t_9}] = \left|\begin{array}{ccc|c} 0.941 & 0.030 & -0.336 & -16.31 \\ 0.115 & 0.908 & -0.403 & -31.24 \\ 0.317 & -0.418 & 0.851 & 182.7 \end{array}\right|$$

Consider the radial distortion:

$$[R_9|\vec{t}_9] = \left|\begin{array}{ccc|c} 0.940 & 0.031 & -0.339 & -16.93 \\ 0.114 & 0.909 & 0.401 & -31.10 \\ 0.320 & -0.415 & 0.851 & 183.3 \end{array}\right|$$

The rotation and translation of the position 18:

$$[R_{18}|\vec{t}_{18}] = \left|\begin{array}{ccc|c} 0.992 & 0.108 & 0.069 & -41.77 \\ -0.113 & 0.990 & 0.081 & -35.42 \\ -0.060 & -0.088 & 0.994 & 135.24 \end{array}\right|$$

After the LM refinement without consideration of radial distortion:

$$[R_{18}|\vec{t}_{18}] = \left|\begin{array}{ccc|c} 0.992 & 0.103 & 0.068 & -39.99 \\ -0.108 & 0.991 & 0.079 & -35.79 \\ -0.060 & -0.086 & 0.995 & 133.8 \end{array}\right|$$

Consider the radial distortion:

$$[R_{18}|\vec{t}_{18}] = \left|\begin{array}{ccc|c} 0.992 & 0.103 & 0.069 & -40.46 \\ -0.109 & 0.991 & 0.077 & -35.65 \\ -0.060 & -0.084 & 0.995 & 134.4 \end{array}\right|$$

4. Reprojection and Observation

To measure the accuracy of your camera-calibration, I reprojected the corner points from 4 views back into the 'Fixed Image', which is the 29th image in the given dataset and the 2nd image my own dataset. I reprojected corners of the 1st, the 9th, the 25th, the 40th views to the fixed image in the provided image set, and reprojected corners of the 4th, the 6th, the 9th, the 18th views to the fixed image in my own image set.

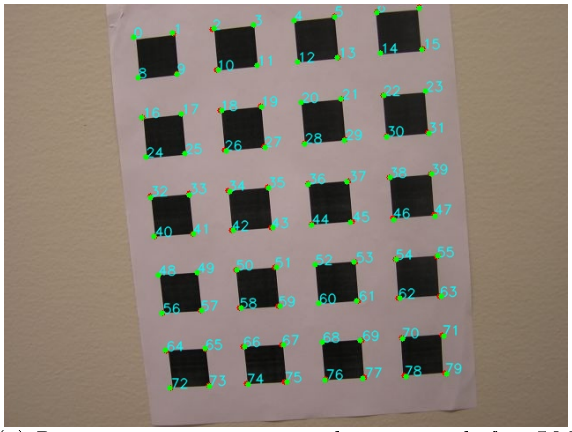
4.1. Given Dataset

The measurement of the projection quality contains the average error, the variance and the maximal error, and the error is defined by the Euclidean distance between the original corner points and the corresponding reprojected corner points.

stage	average error	variance	largest error
Reprojection before refinement	1.36219275	1.06679098	6.79344051
Reprojection after refinement	0.8008501	0.24314927	3.88455069

Table 3: Reprojection error reduction from all views by the LM refinement

Following is the visual comparison of the locations of the original corners vis-a-vis the reprojected corner points, where the reprojected corners are marker by red and the original corners are marked by green. We can clearly observe the improvement provided by the LM refinement.


(a) Reproject intersections on the 1st view before LM

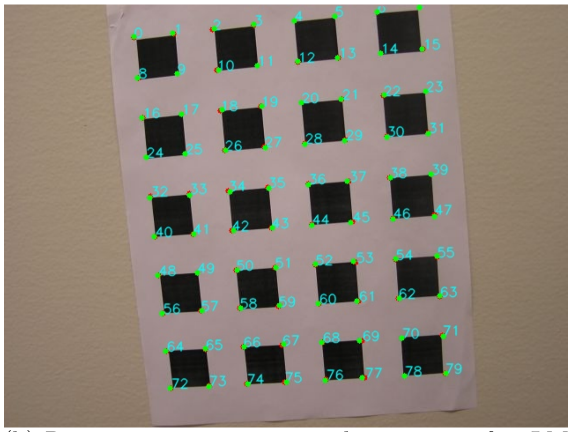
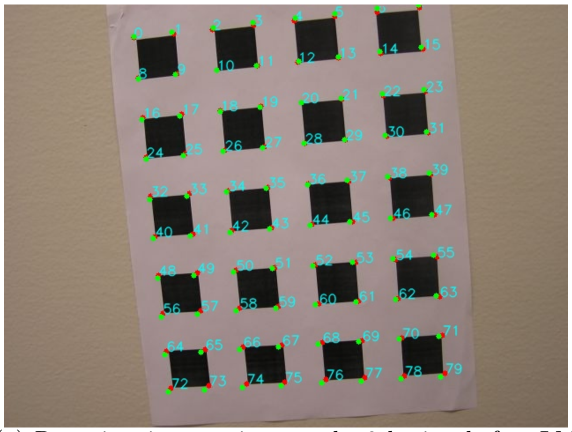

(b) Reproject intersections on the 1st view after LM

Figure 9: Reproject intersections from the 1st view to the fixed image


(a) Reproject intersections on the 9th view before LM

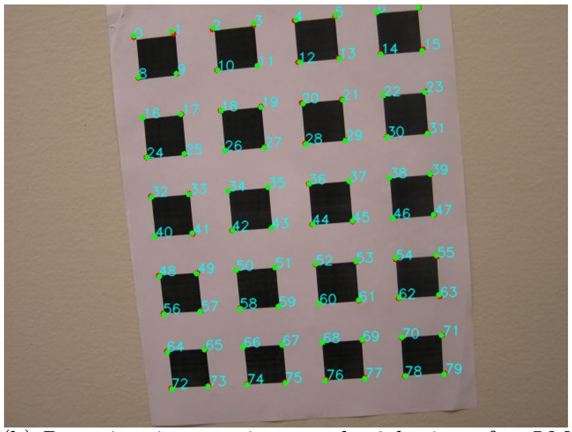
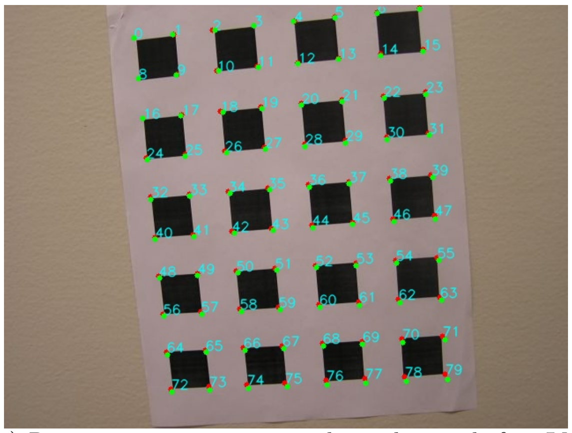

(b) Reproject intersections on the 9th view after LM

Figure 10: Reproject intersections from the 9th view to the fixed image


(a) Reproject intersections on the 35th view before LM

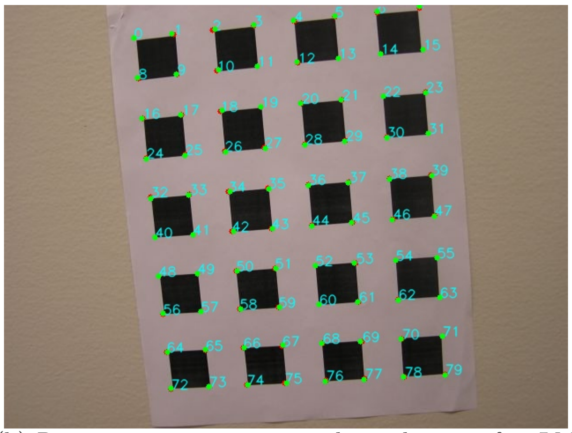
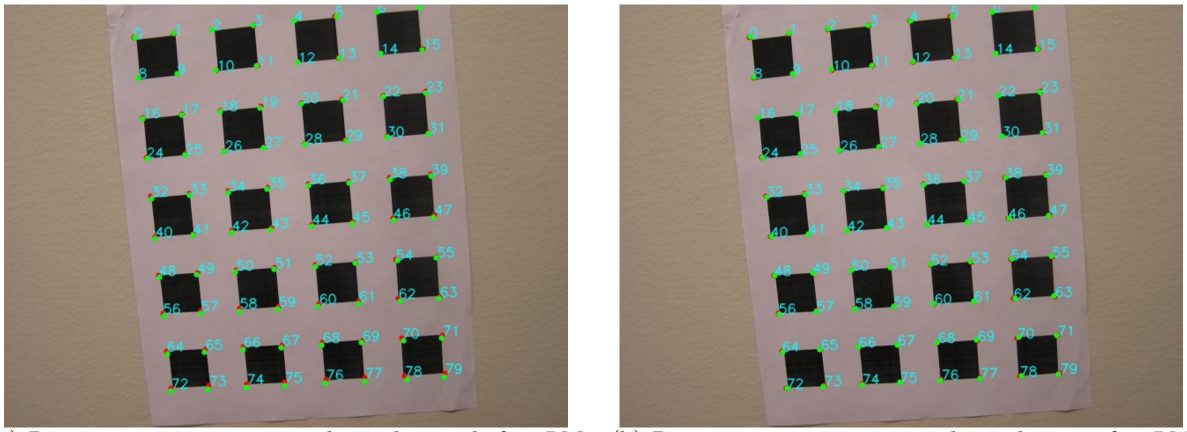

(b) Reproject intersections on the 35th view after LM

Figure 11: Reproject intersections from the 35th view to the fixed image



(a) Reproject intersections on the 40th view before LM (b) Reproject intersections on the 40th view after LM

Figure 12: Reproject intersections from the 40th view to the fixed image

Quantitatively, we list the improvements below:

stage (reprojection to the 29th view)	average error	variance	largest error
Reprojection before refinement (1st view)	1.9309	1.0896	4.2373
Reprojection after refinement (1st view)	0.6959	0.1316	1.7463
Reprojection before refinement (9th view)	3.2201	2.3272	6.7065
Reprojection after refinement (9th view)	1.0579	0.3541	2.8403
Reprojection before refinement (35th view)	2.6459	1.5596	2.8403
Reprojection after refinement (35th view)	0.9813	0.3035	2.6102
Reprojection before refinement (40th view)	2.0271	1.1945	4.7703
Reprojection after refinement (40th view)	0.8309	0.2385	2.3207

Table 4: Reprojection performance improvement from selected views by the LM refinement

4.2. My Dataset

We first show the overall error reduction by the LM optimization:

stage	average error	variance	largest error
Reprojection before refinement	13.43239668	143.51852431	59.54252411
Reprojection after refinement	3.58344406	5.87195738	19.76964539

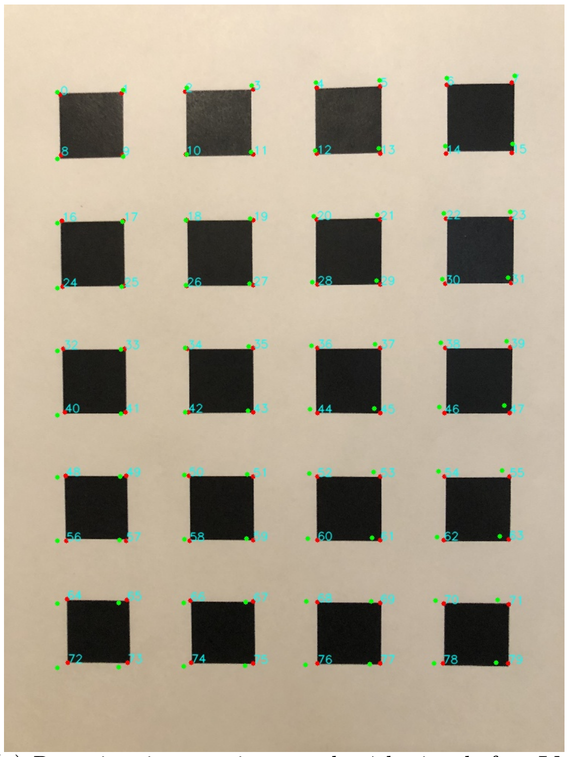
Table 5: Reprojection error reduction from all views by the LM refinement

It's obvious that the reprojection after the LM refinement has been improved a lot. We first quantitatively list the improvements below:

stage (reprojection to the 2nd view)	average error	variance	largest error
Reprojection before refinement (4th view)	8.8282	13.6333	16.9735
Reprojection after refinement (4th view)	3.0144	2.5241	8.6766
Reprojection before refinement (6th view)	3.2557	2.2033	8.3549
Reprojection after refinement (6th view)	2.1577	1.0812	5.9385
Reprojection before refinement (9th view)	11.8417	23.2019	23.6810
Reprojection after refinement (9th view)	3.0629	3.1837	9.2558
Reprojection before refinement (18th view)	3.4260	1.7550	7.5079
Reprojection after refinement (18th view)	2.2706	1.2527	4.9388

Table 6: Reprojection performance improvement from selected views by the LM refinement

Following is the visual comparison of the locations of the original corners vis-a-vis the reprojected corner points, where the reprojected corners are marker by red and the original corners are marked by green.


(a) Reproject intersections on the 4th view before LM

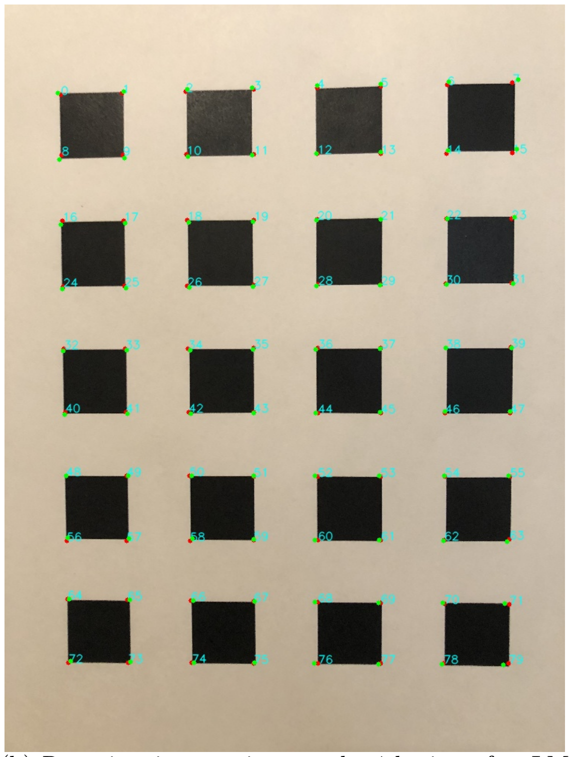
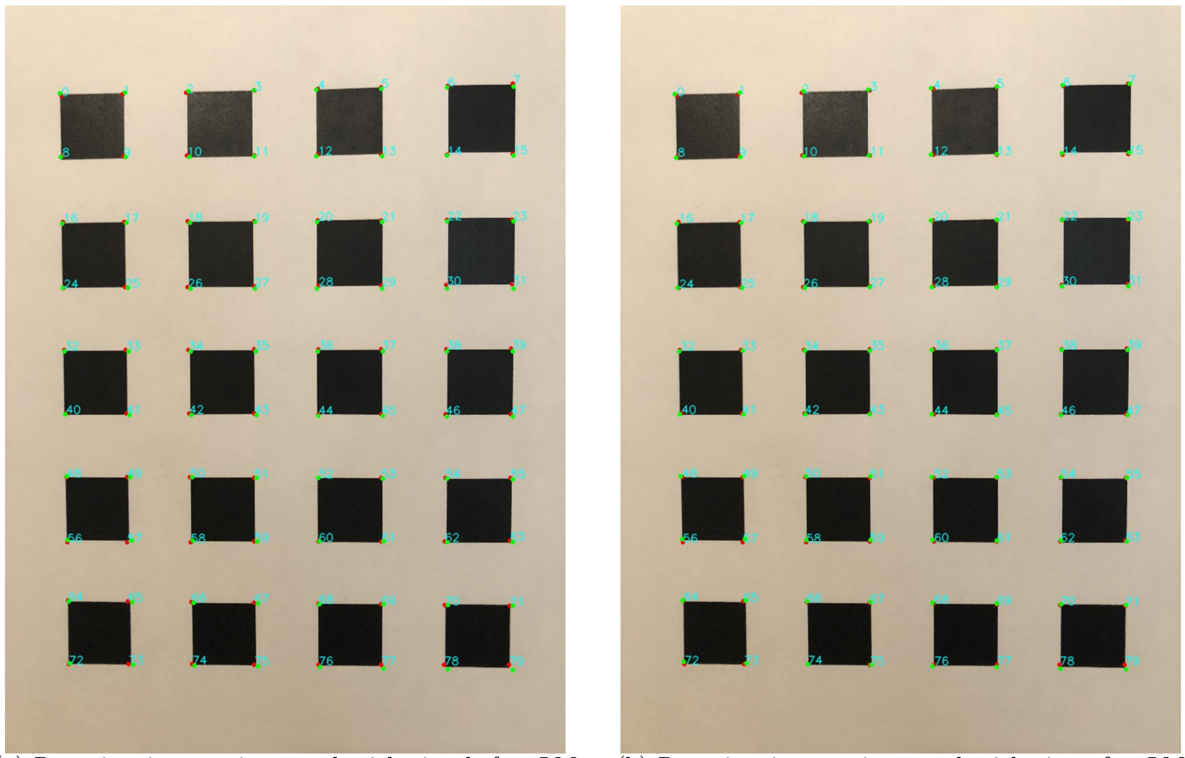
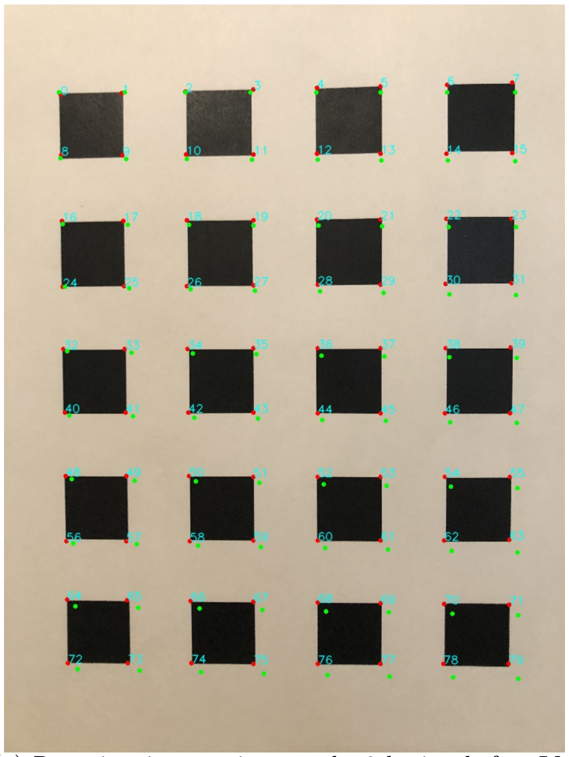

(b) Reproject intersections on the 4th view after LM

Figure 13: Reproject corners from the 4th view to the fixed image



(a) Reproject intersections on the 6th view before LM (b) Reproject intersections on the 6th view after LM

Figure 14: Reproject corners from the 6th view to the fixed image


(a) Reproject intersections on the 9th view before LM

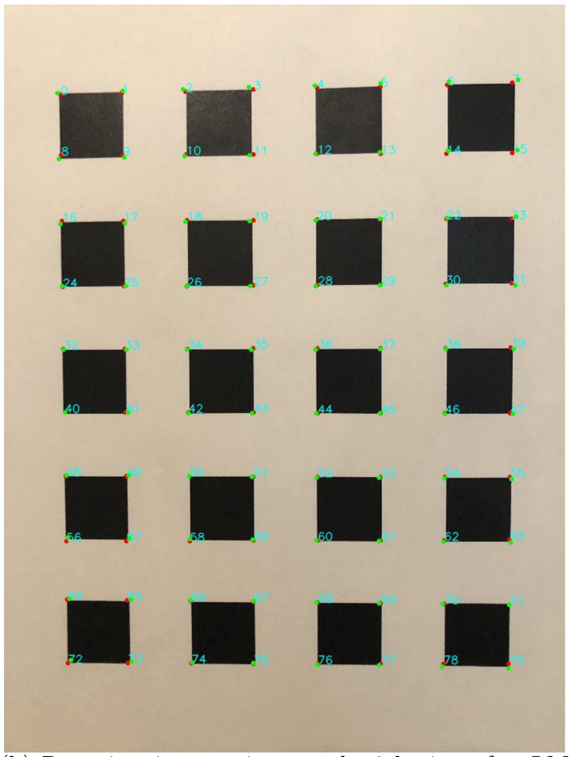
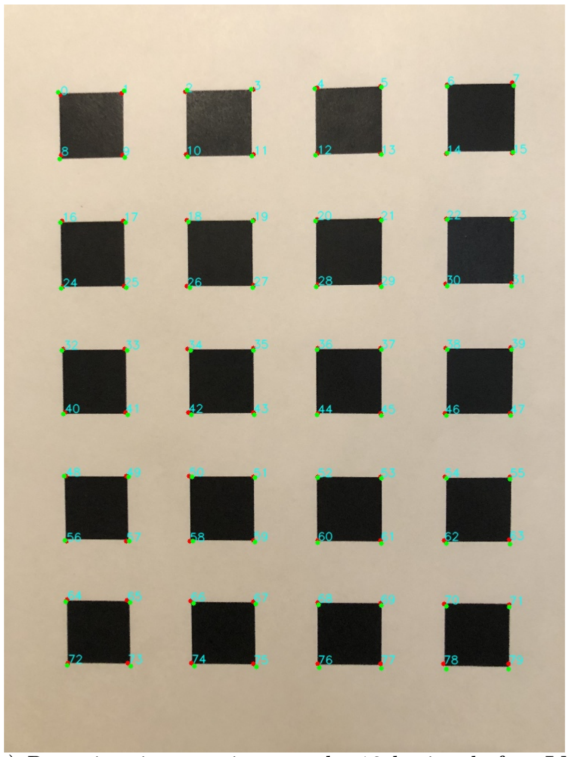

(b) Reproject intersections on the 9th view after LM

Figure 15: Reproject corners from the 9th view to the fixed image


(a) Reproject intersections on the 18th view before LM

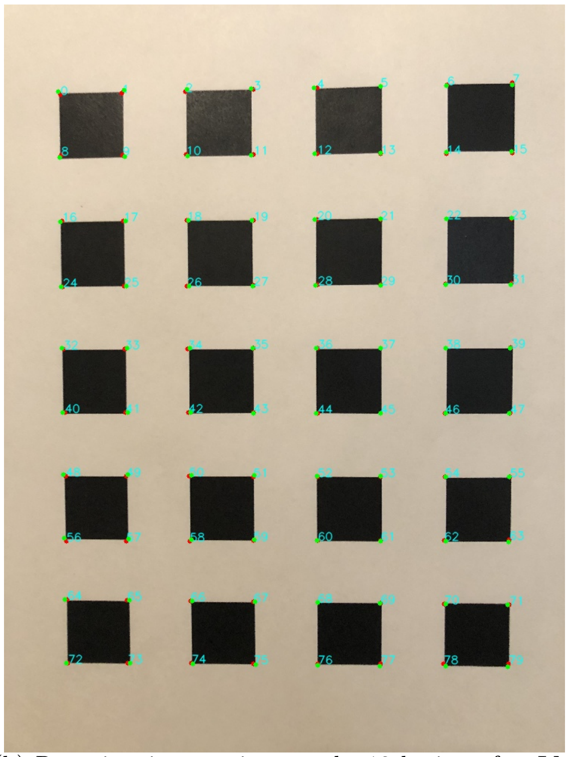

(b) Reproject intersections on the 18th view after LM

Figure 16: Reproject corners from the 18th view to the fixed image

4.3. Observation

1. We can obverse the signification improvement brought by Levenberg-Marquadt (LM) non-linear optimization, both quantitatively and intuitively, it can reduce both the average value of the error and the variance of the error.

2. The results on the given image set is much more accurate than which on my own dataset, one reason for this is because I have much less views than the given dataset, and other reason would be the camera – the iphone camera I used has more nonlinear distortion than the camera used for taking ohotos for the given dataset.

5. Code

```
import cv2
import numpy as np
import os
import math
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

path_root = '/home/xingguang/Documents/ECE661/hw9/Files'

# for my own images:
# img_dataset = os.path.join(path_root, 'Dataset')
# images = []
```

```python
# for i in range(len(os.listdir(img_dataset))):
#     im_path = os.path.join(img_dataset, str(i+1)+'.jpg')
#     images.append(cv2.resize(cv2.imread(im_path), (750, 1000)))

# for the given images
img_dataset = os.path.join(path_root, 'Dataset')
images = []
for i in range(len(os.listdir(img_dataset))):
    im_path = os.path.join(img_dataset, 'Pic_'+str(i+1)+'.jpg')
    images.append(cv2.imread(im_path))


def make_pattern(grid_size, hline_num, vline_num):
    # ma
    x = np.linspace(0, grid_size*(vline_num-1), vline_num)
    y = np.linspace(0, grid_size*(hline_num-1), hline_num)
    xv, yv = np.meshgrid(x, y)
    return np.concatenate([xv.reshape((-1,1)), yv.reshape((-1,1))], axis=1)


def cvtPoint(lines, r):
    rho = lines[:, 0]
    theta = lines[:, 1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
    y0 = b * rho
    pt1 = np.array([x0 + r * (-b), y0 + r * a]).T
    pt2 = np.array([x0 - r * (-b), y0 - r * a]).T
    return pt1, pt2


def rearrange(p1, p2, v=True):
    p1_list = []
    p2_list = []
    for i in range(p1.shape[0]):
        if v:
            if p1[i, 1] > 0 and p2[i, 1] <0:
                p1_list.append(p1[i])
                p2_list.append(p2[i])
            elif p1[i, 1] < 0 and p2[i, 1] > 0:
                p1_list.append(p2[i])
                p2_list.append(p1[i])
        else:
            if p1[i, 0] > 0 and p2[i, 0] <0:
                p1_list.append(p1[i])
                p2_list.append(p2[i])
            elif p1[i, 0] < 0 and p2[i, 0] > 0:
                p1_list.append(p2[i])
                p2_list.append(p1[i])
    return np.concatenate(p1_list, axis=0).reshape((-1, 2)),\
           np.concatenate(p2_list, axis=0).reshape((-1, 2))


def nmsLines(lines, p1, p2, nms_ratio=0.25, v=True):
    if v:
        # d = np.abs(lines[:, 0] * np.cos(lines[:, 1]))
        d = lines[:, 0] * np.cos(lines[:, 1])
        d_abs = np.abs(d)
        nms_thres = nms_ratio * (np.max(d_abs) - np.min(d_abs)) / 7
    else:
        # d = np.abs(lines[:, 0] * np.sin(lines[:, 1]))
        d = lines[:, 0] * np.sin(lines[:, 1])
        d_abs = np.abs(d)
        nms_thres = nms_ratio * (np.max(d_abs) - np.min(d_abs)) / 9

    idx = np.argsort(d, axis=0)
    d_sort = d[idx]
    valid_id = []
    temp_ids = []
    for i in range(d_sort.shape[0]-1):
        if i == 0:
            temp_ids.append(idx[i])
        if d_sort[i+1] - d_sort[i] < nms_thres:
            temp_ids.append(idx[i+1])
        else:
            valid_id.append(temp_ids)
            temp_ids = [idx[i+1]]
        if i == d_sort.shape[0]-2:
            valid_id.append(temp_ids)

    p1_list = []
    p2_list = []
    for ids in valid_id:
        p1_list.append(np.average(p1[ids], axis=0))
        p2_list.append(np.average(p2[ids], axis=0))
    return np.concatenate(p1_list, axis=0).reshape((-1, 2)),\
            np.concatenate(p2_list, axis=0).reshape((-1, 2))


def drawLines(img, p1, p2, c):
    out = img.copy()
    for i in range(0, p1.shape[0]):
        pt1 = (int(p1[i, 0].item()), int(p1[i, 1].item()))
        pt2 = (int(p2[i, 0].item()), int(p2[i, 1].item()))
        cv2.line(out, pt1, pt2, c, 2)
    return out


def drawPoints(img, p, with_text=True, c=(0, 0, 255)):
    out = img.copy()
    for i in range(p.shape[0]):
        point = (int(p[i,0].item()), int(p[i,1].item()))
        out = cv2.circle(out, point, radius=3, color=c, thickness=-1)
        if with_text:
            out = cv2.putText(out, str(i), point, cv2.FONT_HERSHEY_SIMPLEX, 0.5,
    (255, 255, 0), 1, cv2.LINE_AA)
    return out


def find_itsc(vp1, vp2, hp1, hp2):
    vp1 = np.append(vp1, np.ones((vp1.shape[0],1)), axis=1)
    vp2 = np.append(vp2, np.ones((vp1.shape[0],1)), axis=1)
    hp1 = np.append(hp1, np.ones((hp1.shape[0],1)), axis=1)
    hp2 = np.append(hp2, np.ones((hp2.shape[0],1)), axis=1)
    v_lines = np.cross(vp1, vp2)
    h_lines = np.cross(hp1, hp2)
    points = []
    for i in range(h_lines.shape[0]):
        itscs = np.cross(v_lines, h_lines[i])
        points.append(itscs[:,:2]/itscs[:, 2].reshape((-1,1)))
    return np.concatenate(points, axis=0)


def extract_intersections_from_image(in_img, nms_ratio, i, r=0.5, t=50, save_imgs=
    True):
    raw_img = np.copy(in_img)
    gray = cv2.GaussianBlur(cv2.cvtColor(raw_img, cv2.COLOR_BGR2GRAY), (3, 3),
    1.4)
    edges = cv2.Canny(gray, 255*1.5,255)
    lines = cv2.HoughLines(edges, 1, r*np.pi / 180, t)
    lines = np.squeeze(lines)
    # cos(theta)^2 > cos(pi/2)^2 ~ verticle
    vlines = lines[np.where(np.cos(lines[:,1]) ** 2 > 0.5)]
    vp1, vp2 = cvtPoint(vlines, 1000)
    vp1, vp2 = rearrange(vp1, vp2, v=True)
    vp1, vp2 = nmsLines(vlines, vp1, vp2, nms_ratio, v=True)
    if vlines is not None:
        img = drawLines(raw_img, vp1, vp2, c=(0,255,0))

    # cos(theta)^2 <= cos(pi/2)^2 ~ horizontal
    hlines = lines[np.where(np.cos(lines[:,1]) ** 2 <= 0.5)]
    hp1, hp2 = cvtPoint(hlines, 1000)
    hp1, hp2 = rearrange(hp1, hp2, v=False)
    hp1, hp2 = nmsLines(hlines, hp1, hp2, nms_ratio, v=False)
    if hlines is not None:
        img = drawLines(img, hp1, hp2, c=(255,0,0))
    intersections = find_itsc(vp1, vp2, hp1, hp2)
    img_with_points = drawPoints(raw_img, p=intersections)
    if save_imgs:
        p1, p2 = cvtPoint(lines, 1000)
        img_Hough = drawLines(raw_img, p1, p2, c=(255,255,255))
        cv2.imwrite(os.path.join(path_root, "HoughTrans", "h_"+str(i)+'.jpg'),
    img_Hough)
        cv2.imwrite(os.path.join(path_root, "Hough_filtered", "hf_"+str(i)+'.jpg')
    , img)
        cv2.imwrite(os.path.join(path_root, "Canny", "edge_"+str(i)+'.jpg'), edges
    )
        cv2.imwrite(os.path.join(path_root, "Intersections", "itsc_"+str(i)+'.jpg'
    ), img_with_points)
    return intersections


def findHomoproj(source, target):
    # target = source * H^T
    def F_unit(source_point, target_point):
        x, y = source_point[0], source_point[1]
        x_, y_ = target_point[0], target_point[1]
        return np.asarray([[x, y, 1, 0, 0, 0, -x*x_, -y*x_],
                            [0, 0, 0, x, y, 1, -x*y_, -y*y_]])
    F_list = [F_unit(source[i], target[i]) for i in range(source.shape[0])]
    F = np.concatenate(F_list, axis=0)
    T_span = target.reshape((-1,1))
    H_param = np.dot(np.linalg.pinv(F), T_span)
    H = np.ones((9, 1))
    H[:8, :] = H_param
    return H.reshape((3, 3))
```

```
def findOmega(H_list):
    def V_unit(H):
        h11, h12, h13 = (H[0,0], H[1,0], H[2,0])
        h21, h22, h23 = (H[0,1], H[1,1], H[2,1])
        return np.asarray([[h11*h21, h11*h22+h12*h21, h12*h22, \
                  h13*h21+h11*h23, h13*h22+h12*h23, h13*h23],\
                  [h11**2-h21**2, 2*h11*h12-2*h21*h22, h12**2-h22**2, \
                  2*h11*h13-2*h21*h23, 2*h12*h13-2*h22*h23, h13**2-h23**2]])
    V_list = [V_unit(H) for H in H_list]
    V = np.concatenate(V_list, axis=0)
    _, _, v = np.linalg.svd(V)
    b = v[-1]
    Omega = np.array([[b[0],b[1],b[3]], [b[1],b[2],b[4]], [b[3],b[4],b[5]]])
    return Omega

def findK(omega):
    w = np.copy(omega)
    v0 = (w[0,1]*w[0,2] - w[0,0]*w[1,2]) / (w[0,0]*w[1,1] - w[0,1]**2)
    lamda = w[2,2] - (w[0,2]**2 + v0 * (w[0,1]*w[0,2] - w[0,0]*w[1,2])) / w[0,0]
    a_x = np.sqrt(lamda / w[0,0])
    a_y = np.sqrt(lamda * w[0,0] / (w[0,0]*w[1,1] - w[0,1]**2))
    s = -w[0,1] * a_x**2 * a_y / lamda
    u0 = s * v0 / a_y - w[0,2] * a_x**2 / lamda
    K = np.array([[a_x, s, u0], [0, a_y, v0], [0, 0, 1]])
    return K

def findRt(H_list, K):
    R_list = []
    t_list = []
    for H in H_list:
        r12_t = np.dot(np.linalg.inv(K), H)
        lamda = 1 / np.linalg.norm(r12_t[:,0])
        r12_t = lamda * r12_t
        r3 = np.cross(r12_t[:,0], r12_t[:, 1])
        Q = np.copy(r12_t)
        Q[:, 2] = r3
        u, _, v = np.linalg.svd(Q)
        R = np.dot(u, v)
        R_list.append(R)
        t_list.append(r12_t[:, 2].copy())
    return R_list, t_list


def projTransform(H, source):
    nps = source.shape[0]
    source_rep = np.concatenate((source, np.ones((nps,1))), axis=1)
    t_homo = np.dot(H, source_rep.T).T
    t_norm = t_homo[:,:2] / t_homo[:,2].reshape((nps,1))
    return t_norm

def construct_params(R_list, t_list, K):
    Rt_list = []
    for R, t in zip(R_list, t_list):
        phi = np.arccos((np.trace(R)-1)/2)
        w = phi / (2 * np.sin(phi)) * np. asarray([
            R[2,1] - R[1,2], R[0,2] - R[2,0], R[1,0] - R[0,1]])
        Rt_list.append(np.append(w, t))
    K_param = np.asarray([K[0,0], K[0,1], K[0,2], K[1,1], K[1,2]])
    params = np.append(K_param, np.concatenate(Rt_list))
    return params

def reconstruct_mat(lm_params):
    N = int((lm_params.shape[0]-5) / 6)
    k = lm_params[:5]
    K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])
    R_list = []
    t_list = []
    for i in range(N):
        w = lm_params[5+i*6:8+i*6]
        t = lm_params[8+i*6:11+i*6]
        phi = np.linalg.norm(w)
        wx = np.array([[0, -w[2], w[1]], [w[2],0,-w[0]], [-w[1],w[0],0]])
        R = np.eye(3) + np.sin(phi)/phi*wx + (1-np.cos(phi))/(phi**2) * np.dot(wx,
 wx)
        R_list.append(R)
        t_list.append(t)
    return R_list, t_list, K


def radial_distort(itscs, k1, k2, x0, y0):
    # Remove radial distortions
    x = itscs[:,0]
    y = itscs[:,1]
    r = (x-x0)**2 + (y-y0)**2
    x_rad = x + (x-x0) * (k1*r + k2*(r**2))
    y_rad = y + (y-y0) * (k1*r + k2*(r**2))
    return np.hstack([x_rad.reshape((-1, 1)), y_rad.reshape((-1, 1))])


def cost_Func(params, itsc_list, pattern, with_rd=False):
    num_Img = len(itsc_list)
    if with_rd:
        R_list, t_list, K = reconstruct_mat(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
        R_list, t_list, K = reconstruct_mat(params)
    Proj_pattern = []
    for R, t in zip(R_list, t_list):
        Rt = np.concatenate([R[:,0:1], R[:,1:2], t.reshape((-1,1))], axis=1)
        H = np.dot(K, Rt)
        reconst_p = projTransform(H, pattern)
        if with_rd:
            reconst_p = radial_distort(reconst_p, k1, k2, x0, y0)
        Proj_pattern.append(reconst_p)
    projec_itscs = np.concatenate(Proj_pattern, axis=0)
    gt_ptrns = np.concatenate(itsc_list, axis=0)
    diff = projec_itscs - gt_ptrns
    return diff.flatten()

def error(diff):
    diff = diff.reshape((-1, 2))
    diff_norm = np.linalg.norm(diff, axis=1)
    e = np.average(diff_norm)
    var = np.var(diff_norm)
    max_d = np.max(diff_norm)
    return np.array([e, var, max_d])

def measure(diff):
    diff = diff.reshape((-1, 2))
    diff_norm = np.linalg.norm(diff, axis=1)
    num_Img = int(diff_norm.shape[0]/80)
    measured = []
    for i in range(num_Img):
        measure_imgi = {}
        current_d = diff_norm[i*80:i*80+80]
        measure_imgi["Means"] = np.average(current_d)
        measure_imgi["Variances"] = np.var(current_d)
        measure_imgi["max_distance"] = np.max(current_d)
        measured.append(measure_imgi)
    return measured


def measure_proj(pattern, itsc_list, valid_idlist, images, params, status='before'
    ):
    if params.shape[0] % 6 == 1:
        R_list, t_list, K = reconstruct_mat(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
        R_list, t_list, K = reconstruct_mat(params)
    H_list = []
    for R, t in zip(R_list, t_list):
        Rt = np.concatenate([R[:,0:1], R[:,1:2], t.reshape((-1,1))], axis=1)
        H = np.dot(K, Rt)
        H_list.append(H)

    diff_list = []
    for i, H in enumerate(H_list):
        img_idx = valid_idlist[i]
        img_i = images[img_idx]
        projed = projTransform(H, pattern)
        if params.shape[0] % 6 == 1:
            projed = projTransform(H, pattern)
            projed = radial_distort(projed, k1, k2, x0, y0)
        else:
            projed = projTransform(H, pattern)
        proj_img = drawPoints(img_i, projed, with_text=True, c=(0, 255, 255))
        cv2.imwrite(os.path.join(path_root, "proj_"+status,\
                    "to"+str(img_idx+1)+'.jpg'), proj_img)
        diff = itsc_list[i] - projed
        diff_list.append(diff)
    e = error(np.array(diff_list).flatten())
    measure_params = measure(np.array(diff_list).flatten())
    return e, measure_params

def reproject(static_idx, valid_idlist, images, params, itsc_list, status='before'
    ):
    img_idx = valid_idlist[static_idx]
    static_img = np.copy(images[img_idx])
    static_itscs = itsc_list[static_idx]
    static_img = drawPoints(static_img, static_itscs)
    if status == 'withrad':
        R_list, t_list, K = reconstruct_mat(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
        R_list, t_list, K = reconstruct_mat(params)
    H_list = []
    for R, t in zip(R_list, t_list):
        Rt = np.concatenate([R[:,0:1], R[:,1:2], t.reshape((-1,1))], axis=1)
        H = np.dot(K, Rt)
        H_list.append(H)
    sH = H_list[static_idx]
    diff_list = []
    for i, H in enumerate(H_list):
        if i == static_idx:
            continue
        img_i = valid_idlist[i]
        H_i_s = np.dot(sH, np.linalg.inv(H))
        reprojed = projTransform(H_i_s, itsc_list[i])
        reproj_img = drawPoints(static_img, reprojed, with_text=False, c=(0, 255,
    0))
        cv2.imwrite(os.path.join(path_root, "reproj_"+status, \
                        str(img_i+1)+'to'+str(valid_idlist[static_idx]+1)+'.jpg'),
    reproj_img)
        diff = static_itscs - reprojed
        diff_list.append(diff)
    e = error(np.array(diff_list).flatten())
    measure_params = measure(np.array(diff_list).flatten())
    return e, measure_params

# --------------------------------Main function --------------------------------
# -----------------------------------------------------------------------------
# Load all images, extract the intersections and compute H's
# -----------------------------------------------------------------------------
gt_pattern = make_pattern(grid_size=10, hline_num=10, vline_num=8)

H_list = []
valid_image_ids = []
itsc_list = []

# for the given images:
nms = 0.25
rsl = 0.5
thres = 50

# for my own images:
# nms = 0.28
# rsl = 0.6
# thres = 70
for i, image in enumerate(images):
    itscs = extract_intersections_from_image(image, nms, i+1, rsl, thres,
    save_imgs=False)
    if itscs.shape[0] == 80:
        H = findHomoproj(gt_pattern, itscs)
        H_list.append(H)
        itsc_list.append(itscs)
        valid_image_ids.append(i)
```

```
print("Total number of images being detected 80 intersections:", len(
    valid_image_ids))
print("They are (indices):", valid_image_ids)

# ------------------------------------------------------------------------------
# Compute the intrinsic matrix K and rotation matrices
# ------------------------------------------------------------------------------
omega = findOmega(H_list)
K = findK(omega)
R_list, t_list = findRt(H_list, K)

# ------------------------------------------------------------------------------
# LM optimization to refine the camera matrices
# ------------------------------------------------------------------------------
params = construct_params(R_list, t_list, K)
loss = cost_Func(params, itsc_list, pattern=gt_pattern)
sol = least_squares(cost_Func, params, method = 'lm', args=[itsc_list, gt_pattern
    ])
R_list_refined, t_list_refined, K_refined = reconstruct_mat(sol.x)
sol_rad = least_squares(cost_Func, np.append(params, np.array([0, 0])), \
                        method = 'lm', args=[itsc_list, gt_pattern, True])
R_list_refined_wr, t_list_refined_wr, K_refined_wr = reconstruct_mat(sol_rad.x
    [:-2])
print("k1, k2:", sol_rad.x[-2:])
print("K before refinement:", K)
print("K after refinement without radial distortion:", K_refined)
print("K after refinement with radial distortion:", K_refined_wr)

# ------------------------------------------------------------------------------
# Project the ground truth pattern to every image
# ------------------------------------------------------------------------------
_, measure_nolm= measure_proj(gt_pattern, itsc_list, valid_image_ids, images,
    params, "before")
_, measure_lm= measure_proj(gt_pattern, itsc_list, valid_image_ids, images,
    sol_rad.x, "after")
# measure overall accuracy
print(error(cost_Func(params, itsc_list, pattern=gt_pattern)))
print(error(cost_Func(sol.x, itsc_list, pattern=gt_pattern)))
# measure accuracy of selected views
print(error(cost_Func(sol_rad.x, itsc_list, pattern=gt_pattern, with_rd=True)))
print("measure the projection on image", valid_image_ids[10]+1, measure_nolm[10],
    measure_lm[10])
print("measure the projection on image", valid_image_ids[36]+1, measure_nolm[36],
    measure_lm[36])
# [0, 7, 18, 36] for given images
# [3, 5, 8, 15] for my own images
for i in [0, 7, 18, 36]:
    print("Project the gt pattern to image:", valid_image_ids[i]+1, "the ratation
    matrices:")
    print(R_list[i], t_list[i])
    print(R_list_refined[i], t_list_refined[i])
    print(R_list_refined_wr[i], t_list_refined_wr[i])

# ------------------------------------------------------------------------------
# Project the corners of all views to the fixed image
# ------------------------------------------------------------------------------

fix_id = 1
e_before, measured_before = reproject(fix_id, valid_image_ids, images, params,
    itsc_list, status='before')
print("Overall reprojection error, var, and max distance before refinement:",
    e_before)

e_after, measured_after = reproject(fix_id, valid_image_ids, images, sol.x,
    itsc_list, status='after')
print("Overall reprojection error, var, and max distance after refinement:",
    e_after)

# [3, 7, 9, 36] for the given images
# [2, 4, 7, 14] for my own images
for i in [3, 7, 9, 36]:
    if i < fix_id:
        print("image:", valid_image_ids[i]+1)
    else:
        print("image:", valid_image_ids[i]+2)
    print(measured_before[i])
    print(measured_after[i])
```