ECE 661 – Homework 8

Ran Xu

xu943@purdue.edu

11/13/2018

1. Math Reductions and Algorithms

In this homework, I want to estimate the intrinsic and external parameters of an unknown camera using Zhang's algorithm. In general, a calibration pattern sheet is attached on the $z = 0$ plane with horizontal right-side as x-axis and vertical down-side as y-axis. A couple of images is taken in different views in which the one direct face to it is referred as "Fixed Image". The camera calibration can be divided into 3 steps and 1 additional evaluation step. Firstly, in the pre-processing step, I resize the image, find edges, lines and corners and label them in order to establish point correspondence between images. Secondly, I use a list of math reductions to derive the homography between image pairs, intrinsic parameters, external parameters of the unknown camera. Finally, I refine these parameters using LM optimization and radial distortion is an optional part to further decrease the projection error. Last but not the least, I re-project an image from any arbitrary view to the "Fixed Image" and compare the re-projected error and ground truth error to evaluate the quality of camera calibration. The detailed math reductions and algorithms are described in the following subsections.

1.1 Pre-processing steps

Firstly, if the images are too big, i.e. either dimension is large than 1000 pixels. I will resize the image to a small shape so that each dimension is around 500 pixels. The aspect ratio is preserved, and the down-sampling rate is denoted as DsR. In Sec. 1.2, I will further discuss how this step affect the parameters to calibrate. The images are loaded in black-and-white since the RGB information does not provide too much additional information to localize edges.

Secondly, I use Canny Edge Detector to detect the edges between the black block and white background. Theoretically, the canny edge detector is derived by optimizing a convolutional function $h(x, y)$ with respect to three criteria:

1. The SNR (Signal-To-Noise Ratio) at the true location of the edge;
2. The localization of the detected edge vis-à-vis that of the true edge;
3. The distance between where the true edge is detected and the nearest spurious edge.

In practice, I use OpenCV Canny() function to automate this step. The parameters in this Canny() function can be manually tuned to achieve a good enough result that detects all edges in all images across the dataset.

Thirdly, I use Hough Line Detector to detect presentable lines that connects the edges. In short, the Hough Line Detector count the histogram of (R, θ) across all edges, where R is the distance of an edge to the image origin and θ is the angle between the line connecting edge and origin and the x-axis. As a result, a vertical line will have peak votes around $(R_v, 0)$, where R_v is the distance from the origin to the vertical line. While a horizontal line will have peak votes around $(R_h, 0)$, where R_h is the distance from the origin to the horizontal line.

There are two problems in Hough Line Detector – first being how to choose the precision of R and θ in getting histogram and minimum votes for line to be detected; second being how to finally elegantly get 8 vertical lines and 10 horizontal lines. To solve the first problem, I manually choose some numbers and check the results and finally pick the one that can detect all true lines. Being too conservative, the approach will output many detected lines among which a lot of them coincide. To merge the redundant detection, I use my own Mergelines module. I first distinguish the vertical lines by picking $\theta \leq \frac{\pi}{4}\ or\ \theta \geq \frac{3\pi}{4}$ and distinguish the horizontal lines otherwise. Then sort all vertical lines by their R values and merge them using mean as representative if the difference is less than a threshold. With a good edge detection in the last step, good parameter in Hough Line Detector and good threshold in Mergelines module. I have accurately detected all 8 vertical lines and 10 horizontal lines in all given images.

Fourthly, I find the intersection between each pair of vertical line and horizontal line. Suppose the representation of the two lines are (R_1, θ_1) and (R_2, θ_2). Then the HC representation of these two lines are $(cos\theta_1, sin\theta_1, -R_1)$, $(cos\theta_2, sin\theta_2, -R_2)$. The HC of intersection is the cross product of these two lines. These intersections are labeled row-by-row according to the R value of the horizontal line.

Finally, I measure the true world coordinates of the 80 corners in the calibration pattern.

1.2 Solve the Intrinsic and External Parameters

1.2.1 Solve the pairwise homography between calibration pattern and camera images

Denote the 2D j-th point on the i-th camera image as $\overrightarrow{x_{\iota,J}}$, the 3D j-th point on the calibration pattern as $\overrightarrow{x_{m,J}}' = (x, y, 0,1)^T$, the projection metric from the calibration pattern to the j-th camera image as $P_j = [R_{j1}, R_{j2}, R_{j3}, t_j]$. Then the projection from calibration pattern to the j-th camera image can be presented as

$$\overrightarrow{x_{\iota,J}} = P_j \overrightarrow{x_{m,J}}'^T = [R_{j1}, R_{j2}, t_j]\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

So, we simplify the homography from 2D plane on the calibration pattern sheet to the 2D plane of camera image. Denote the 2D j-th point on the calibration pattern as $\overrightarrow{x_{m,J}} = (x, y, 1)^T$, the homography matrix from calibration pattern to the j-th camera image as $H_j = [R_{j1}, R_{j2}, t_j]$. And we have one equation for each of the 80 point correspondences.

$$\overrightarrow{x_{\iota,J}} = H_j \overrightarrow{x_{m,J}}$$

Due to homogenous property, we assume H_j as follows,

$$H_j = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

Then we have the HC of $\overrightarrow{x_{\iota,J}}$

$$\overrightarrow{x_{\iota,J}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}\begin{bmatrix} x_{1m,j} \\ x_{2m,j} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11}x_{1m,j} + h_{12}x_{2m,j} + h_{13} \\ h_{21}x_{1m,j} + h_{22}x_{2m,j} + h_{23} \\ h_{31}x_{1m,j} + h_{32}x_{2m,j} + 1 \end{bmatrix} = \left(h_{31}x_{1m,j} + h_{32}x_{2m,j} + 1\right)\begin{bmatrix} \mathrm{x}_{1i,j} \\ \mathrm{x}_{2i,j} \\ 1 \end{bmatrix}$$

It can be represented in the following way,

$$\begin{bmatrix} x_{1m,j} & x_{2m,j} & 1 & 0 & 0 & 0 & -x_{1m,j}\mathrm{x}_{1\mathrm{i,j}} & -x_{2m,j}\mathrm{x}_{1\mathrm{i,j}} \\ 0 & 0 & 0 & x_{1m,j} & x_{2m,j} & 1 & -x_{1m,j}\mathrm{x}_{2\mathrm{i,j}} & -x_{2m,j}\mathrm{x}_{2\mathrm{i,j}} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} \mathrm{x}_{1\mathrm{i,j}} \\ \mathrm{x}_{2\mathrm{i,j}} \end{bmatrix}$$

Given 80 pairs of correspondences, a.k.a. 160 equations to be solved. We can re-write the equation as,

$$Ah_j = b$$

where A is a 160-by-8 matrix, b is the 2D coordinates in the camera image, and h is the vector-encoded homography $h = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}]^T$. To solve the vector-encoded homography h_j with least square error, we have

$$h_j = (A^T A)^{-1} A^T b$$

1.2.2 Solve the intrinsic parameter K.

Given a number of homographies from calibration pattern to camera images $H_j = [H_{1j}, H_{2j}, H_{3j}]$. Each homography contribute 2 equations in restricting the intermediate matrix w, where $w = K^{-T}K^{-1}$. We have,

$$\begin{cases} H_{1j}^T w H_{1j} - H_{2j}^T w H_{2j} = 0 \\ H_{1j}^T w H_{2j} = 0 \end{cases}$$

Due to the symmetric property of w, w has 6 unknowns. We can re-write the equations as follows,

$$\begin{bmatrix} H_{11j}^2 - H_{21j}^2 & 2H_{11j} * H_{12j} - 2H_{21j} * H_{22j} & H_{12j}^2 - H_{22j}^2 & 2H_{11j} * H_{13j} - 2H_{21j} * H_{23j} & 2H_{12j} * H_{13j} - 2H_{22j} * H_{23j} & H_{13j}^2 - H_{23j}^2 \\ H_{11j} * H_{21j} & H_{11j} * H_{22j} + H_{12j} * H_{21j} & H_{12j} * H_{22j} & H_{13j} * H_{21j} + H_{11j} * H_{23j} & H_{13j} * H_{22j} + H_{12j} * H_{23j} & H_{13j} * H_{23j} \end{bmatrix} \begin{bmatrix} w_{11} \\ w_{12} \\ w_{22} \\ w_{13} \\ w_{23} \\ w_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

So, equations can be simplified with $Vb = 0$. Depending on the available images N, the shape of matrix V is in $2N \times 6$ shape. The optimal vector b_{opt} , which minimize $||Vb||$ is the eigenvector of $V^T V$ corresponding to the smallest eigenvalue. I then reshape the b_{opt} and get the intermediate matrix w.

Given w, denote the elements in w as $w_{i,j}$ and $K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$. Each parameter in K can be derived as follows,

$$\begin{cases} y_0 = \dfrac{w_{12}w_{13} - w_{11}w_{23}}{w_{11}w_{22} - w_{12}^2} \\ \lambda = w_{33} - \dfrac{w_{13}^2 + y_0(w_{12}w_{13} - w_{11}w_{23})}{w_{11}} \\ \alpha_x = \sqrt{\dfrac{\lambda}{w_{11}}} \\ \alpha_y = \sqrt{\dfrac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}} \\ s = -\dfrac{w_{12}\alpha_x^2\alpha_y}{\lambda} \\ x_0 = \dfrac{s y_0}{\alpha_y} - \dfrac{w_{13}\alpha_x^2}{\lambda} \end{cases}$$

Note that if the images are down-sampled by a factor of DSR, the original intrinsic parameter K should follow the following equation,

$$K_{Org} = \begin{bmatrix} DSR & 0 & 0 \\ 0 & DSR & 0 \\ 0 & 0 & 1 \end{bmatrix} K_{Down-sampled}$$

1.2.3 Solve the external parameter – rotation matrix Rs and transition vector ts.

Given intrinsic parameter K and the homography from the calibration pattern to the j-th camera image $H_j = [H_{1j}, H_{2j}, H_{3j}]$. We get a temporal rotation matrix Q_j and the transition vection t_j, where

$$\begin{cases} Q_{1j} = \xi K^{-1} H_{1j} \\ Q_{2j} = \xi K^{-1} H_{2j} \\ Q_{3j} = Q_{1j} \times Q_{2j} \\ t_j = \xi K^{-1} H_{3j} \\ \xi = \dfrac{1}{\left\| K^{-1} H_{1j} \right\|} \end{cases}$$

The resulting temporal matrix Q_j is not orthogonal and thus does not qualify for a rotation matrix. To get the final rotation matrix R_j, we do SVD decomposition on $Q_j = UDV^T$ and $R_j = UV^T$

1.3 LM optimizations

The purpose of LM optimization is to refine the intrinsic parameter K and the external parameter Rs and ts, so that the cost function is minimized. The cost function is defined with physical meaning -- summed square projection error. In this optimization, we want to project the calibration pattern to the camera plane of j-th camera and measure the square of Euclidean Distance from the projected corners in the calibration pattern and the true observed corners. The square of Euclidean Distance is then summed across all corners in the image and all images in the dataset.

To rigorously define this, we first determine the optimization parameter p, which is the representation of intrinsic parameter K and the external parameter Rs and ts. To avoid introducing redundant degree of freedom, we only put 5 parameters in K $(\alpha_x, s, x_0, \alpha_y, y_0)$ into p. Also, the rotation matrix R has only 3 degree of freedom and we use a 3x1 rotation vector r to replace R. The transition from r to R is as follows,

$$\begin{cases} R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \frac{sin\varphi}{\varphi} \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix} + \frac{1 - cos\varphi}{\varphi^2} \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix} \\ \varphi = \|r\| \end{cases}$$

The transition from R to r is as follows,

$$\begin{cases} \varphi = \arccos\left(\frac{R_{11} + R_{22} + R_{33} - 1}{2}\right) \\ r = \frac{\varphi}{2sin\varphi} \begin{bmatrix} R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{bmatrix} \end{cases}$$

Then the initialization of p is consisting of the flatten representation of K, Rs and ts, a.k.a. the 5 parameters from K, the $(3 + 3)N$ parameters from Rs and ts, where N is the number of images.

Finally, we define the cost function. Denote the j-th point on the calibration pattern in 2D HC representation as $\overrightarrow{x_{m,j}}$, the j-th projected point on the i-th camera image as $\widehat{x_{i,j}}$, the true j-th point on the i-th camera image as $\overrightarrow{x_{i,j}}$. The cost function is then defined as,

$$d_{geom}^2 = \sum_i \sum_j \left\| \overrightarrow{x_{i,j}} - \widehat{x_{i,j}} \right\|^2$$

The projected point $\widehat{x_{i,j}}$ can be represented in terms of $\overrightarrow{x_{m,j}}$ as follows,

$$\widehat{x_{i,j}} = K[R_{1i}, R_{2i}, t_i]\overrightarrow{x_{m,j}}$$

Thus, by optimizing LM parameter p, in terms of minimizing the cost function, we can get a better set of intrinsic parameter K and the external parameters Rs and ts with lower geometric projection error.

Radial Distortion

Considering the radial distortion with 2 additional parameters k_1 and k_2. The LM parameter p is extended by 2. The computation of projected point $\widehat{x_{i,j}}$ is modified. We first get the distortion-free projected point $\widehat{x_{i,j}}'$, which can be computed as,

$$\widehat{x_{i,j}'} = K[R_{1i}, R_{2i}, t_i]\overrightarrow{x_{m,j}}$$

Then the projected point $\widehat{x_{i,j}}$ with radial distortion is,

$$\begin{cases} \widehat{x_{1i,j}} = \widehat{x'_{1i,j}} + \left(\widehat{x'_{1i,j}} - x_0\right)[k_1 r^2 + k_2 r^4] \\ \widehat{x_{2i,j}} = \widehat{x'_{2i,j}} + \left(\widehat{x'_{2i,j}} - y_0\right)[k_1 r^2 + k_2 r^4] \\ r^2 = \left(\widehat{x'_{1i,j}} - x_0\right)^2 + \left(\widehat{x'_{2i,j}} - y_0\right)^2 \end{cases}$$

By optimizing LM parameter p, in terms of minimizing the new cost function, we can get an even better set of intrinsic parameter K and the external parameters Rs and ts with lower geometric projection error.

1.4 Reprojection and error metrics

The purpose of reprojection is also to examine the quality of camera calibration. This time, we project each image back to the world coordinate, a.k.a, the plane of calibration pattern and then project to the plane of "Fixed Image". The Euclidean difference between the 2-projected corners and ground truth corners on the "Fixed Image" is then collected as the error metric of a pair of point correspondence. We can use the mean and variance of this error metric across an image to represent the quality on a particular image and use the that across the whole dataset to represent the quality of camera calibration.

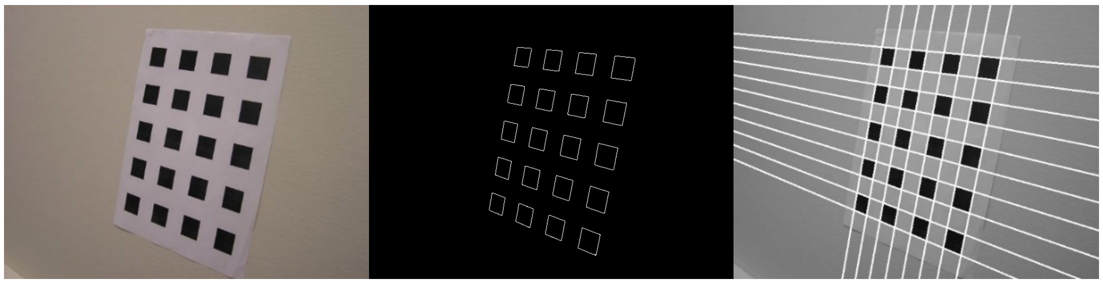
2. Evaluation on the Given Dataset and Observations

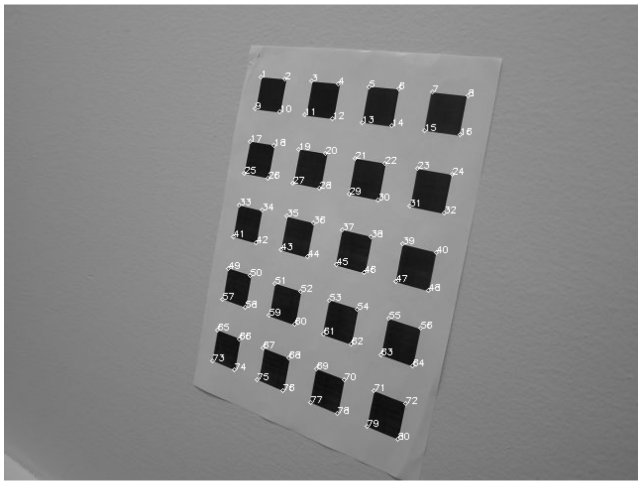
2.1 Pre-processing

The original image dataset is consisting of 40 480-height, 640-width, RGB images. The "Fixed Image" is "Pic_13.jpg". I first load them and transfer to black-and-write images as the input of Canny edge detector. The two parameters for canny edge detectors are 200 and 400. The resulting edges are denoted as 255 (white) and the background are denoted as 0 (black).

Then I feed the resulting edges to the Hough line detector, the precision of R and θ in Hough line detector is 1 pixel and 0.5 degree. The minimum vote for a line to be detected is 55. The output is typically more than 10 horizontal lines and 8 vertical lines. Using my own MergeLine module with threshold of 12, I can perfectly detect 10 horizontal lines and 8 vertical lines in all 40 images. The horizontal lines and vertical lines can be differentiated from θ and each "parallel" line can be differentiated from R as discussed in Sec 1.1.

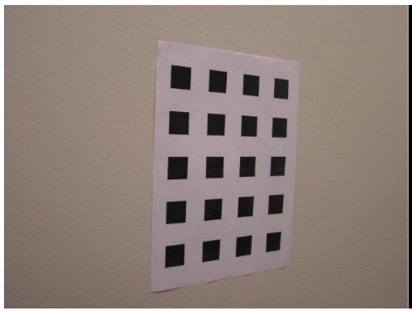
Then using the cross-product representation of each pair of horizontal line and vertical line as discussed in Sec 1.1, I got 80 corners to serve as the key points for calibration. They are labelled row-by-row from 1 to 80.


(a) (b) (c)

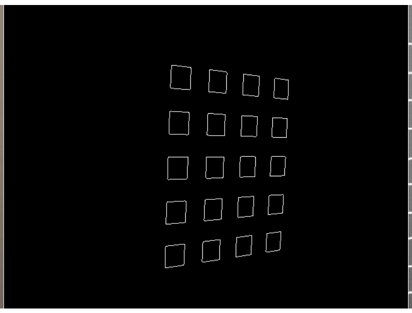


(d)

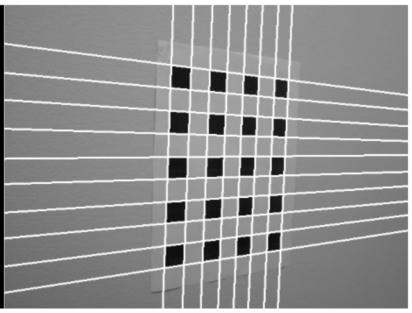
Figure 1: Pre-processing results of Pic_1.jpg - (a) original image taken from a camera (b) detected edges using canny Edge detector (c) detected lines using Hough line detector (d) 80 intersection corners with 1-80 labels.



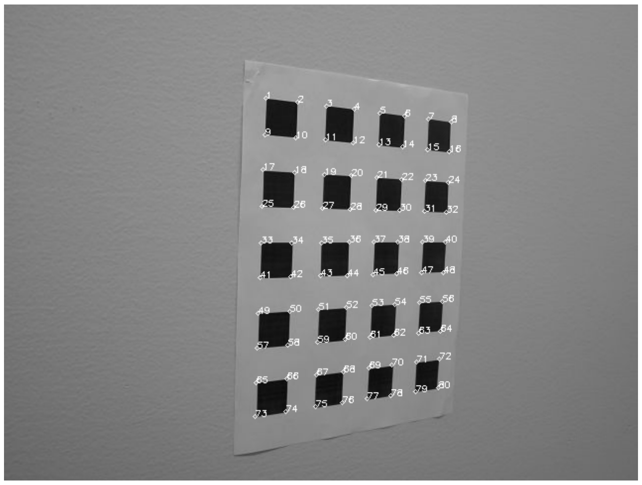
(a)



(b)



(c)



(d)

Figure 2: Pre-processing results of Pic_2.jpg - (a) original image taken from a camera (b) detected edges using canny Edge detector (c) detected lines using Hough line detector (d) 80 intersection corners with 1-80 labels.

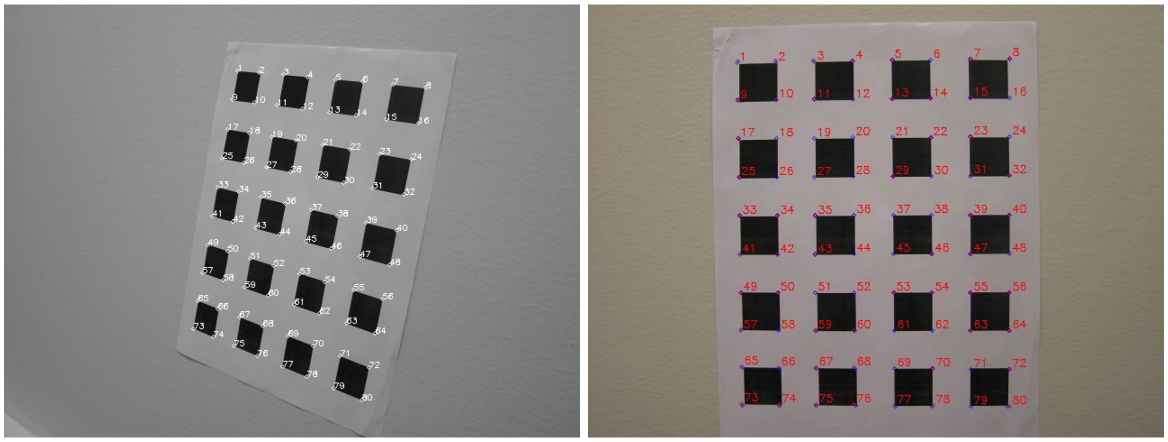
As shown in Fig.1 and 2, the Canny edge detector, Hough line detector, MergeLine module and corner detection module work well. All 80 corners are detected and labelled in row-by-row order.

Measuring the ground truth corners in the calibration pattern

The 80 ground truth corners in the calibration pattern are all considered to lie in the xy plane ($z = 0$). They are the intersection of 8 vertical lines ($x = x_i$) and 10 horizontal lines ($y = y_i$), where $x_i = [23.0, 47.2, 71.7, 96.0, 120.0, 144.7, 169.0, 193.2]\ mm$, and $y_i = [29.7, 54.0, 78.1, 102.7, 126.9, 151.1, 175.5, 200.0\ 224.2, 248.8]\ mm$. **I use millimeter as unit length in physical coordinate and thus by default any unit physical coordinate represents 1 millimeter.**

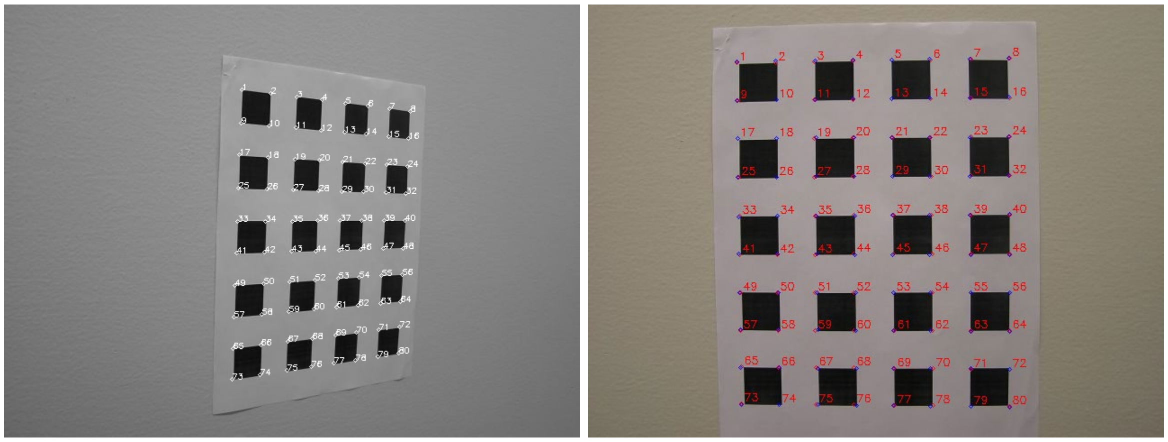
2.2 Reprojection of the corners

In this step, I re-project the 80 corners of an image to the "Fixed Image" (Pic_13.jpg) using the two projection equations in Sec 1.4. The purpose of this step is to visually examine the precision of the intrinsic parameters and external parameters after LM optimizations.



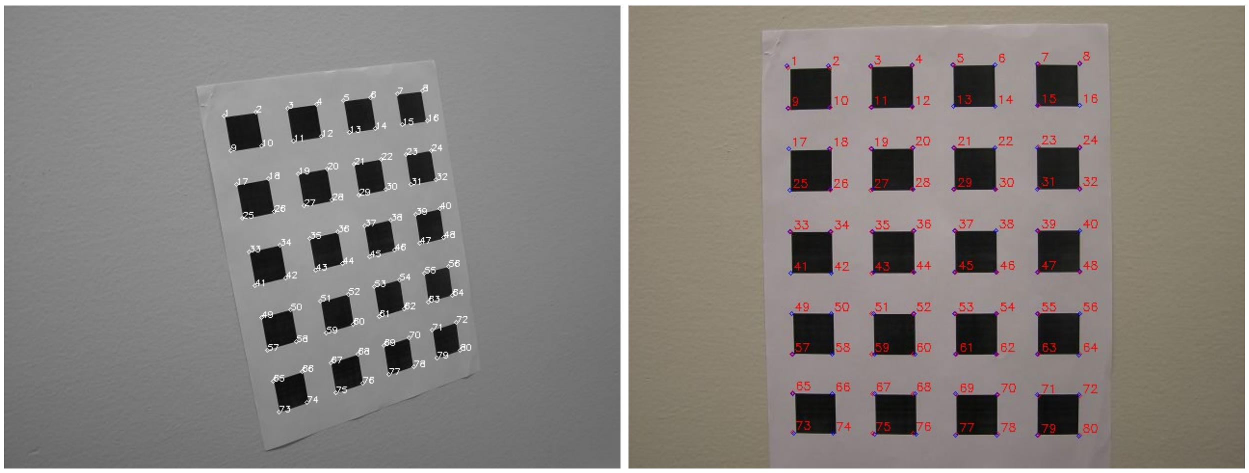
(a) *(b)*

Figure 3: (a) original image (Pic_1.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.75 and 0.13***



(a) *(b)*

Figure 4: (a) original image (Pic_2.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.14 and 0.50.***



(a) *(b)*

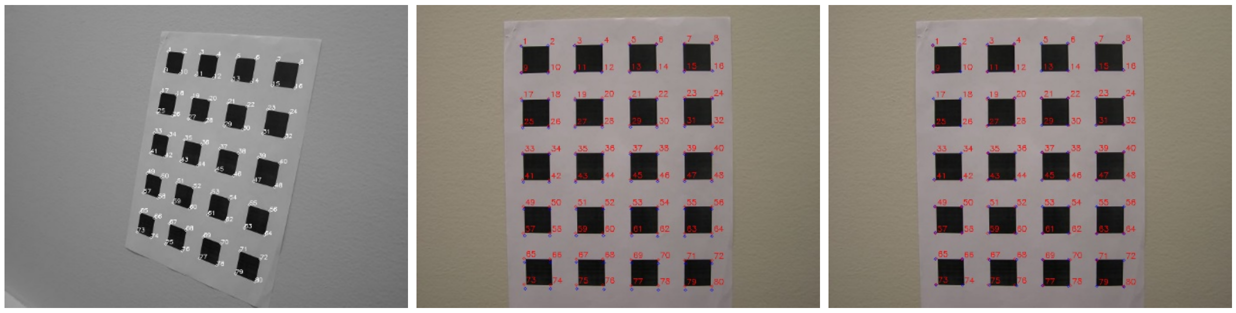
Figure 5: (a) original image (Pic_3.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.98 and 0.28.***

As shown in Fig. 3, 4, and 5, all re-projected corners almost coincide with the true corners on the "Fixed Image". **The mean Euclidean Distance is within 2 pixels in all three images, which is almost negligible, and the variance is also very small (within 1). This shows that the overall calibration algorithm has achieved good accuracy.**

The mean and variance of Euclidean Distance between all re-projected corners and true corners in all 40 given images are 0.88 and 0.28. This aggregated result again proves the good performance of the camera calibration algorithm.

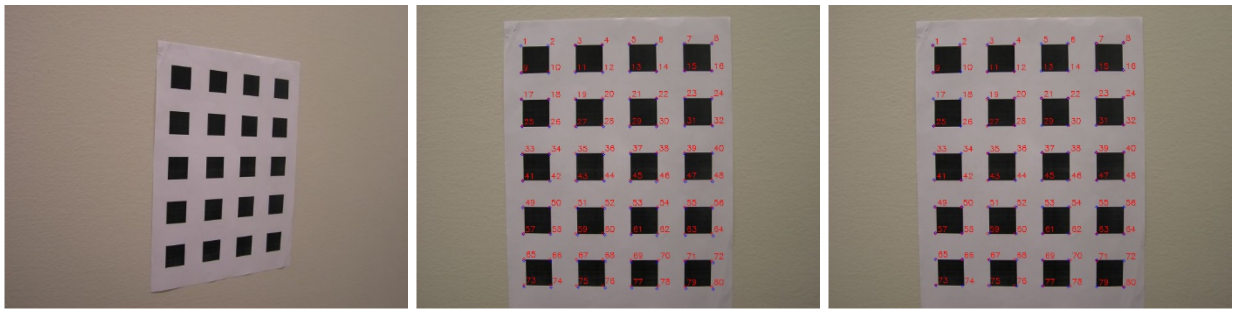
2.3 Marginal gains using LM optimization (with bonus requirement)

In this step, I re-project the 80 corners of an image to the "Fixed Image" (Pic_13.jpg) using the two projection equations in Sec 1.4, while I use the intrinsic parameters and external parameters before and after LM optimizations for comparison to show the marginal gains using LM optimizations.



(a) *(b)* *(c)*

Figure 6: (a) original image (Pic_1.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters before LM optimization. ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 3.52 and 2.42*** *(c) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters after LM optimization.* ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.75 and 0.13***



(a) *(b)* *(c)*

Figure 7: (a) original image (Pic_2.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters before LM optimization. ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 2.38 and 1.22*** *(c) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters after LM optimization.* ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.14 and 0.50***

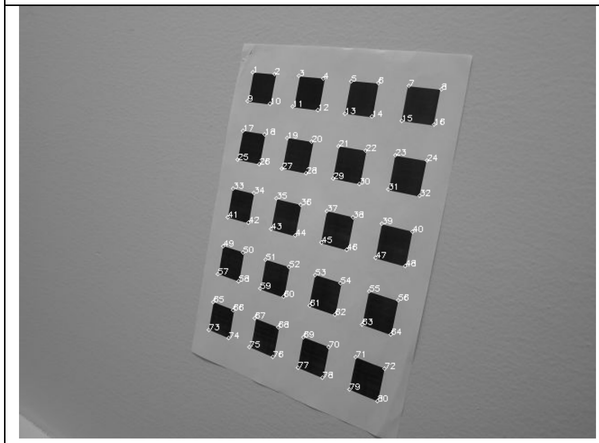
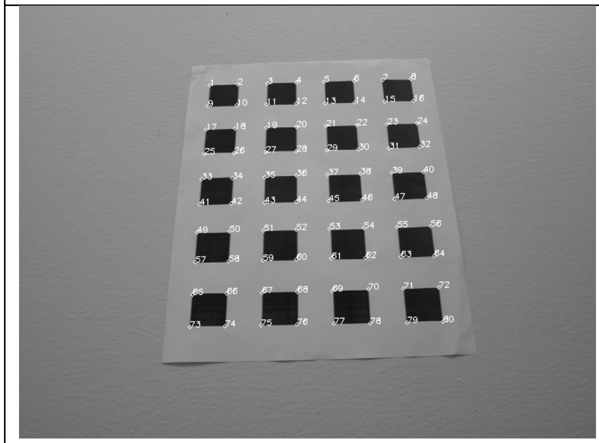
As shown in Fig. 6 and 7, visually we can see some small re-projection errors in the last row using the parameter before LM optimization, while these errors are removed after LM optimizations. The mean and variance of Euclidean Distance between re-projected corners and true corners are reduced as shown in the captions of each figure. **Considering all the corners in all 40 images, the mean and variance drop from 2.17 and 2.55 to 0.88 and 0.28.**

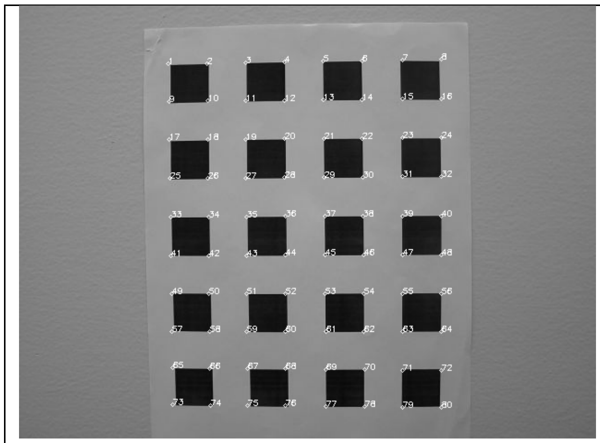
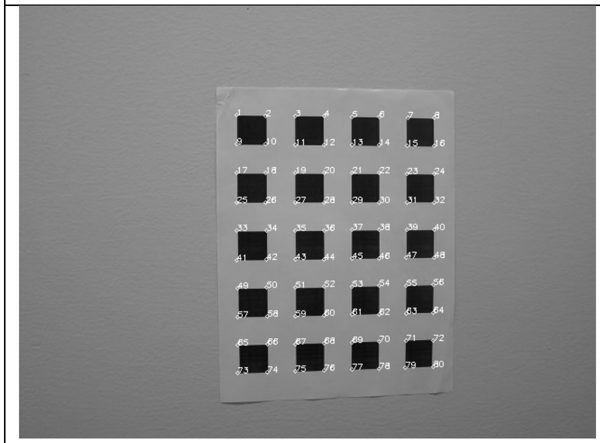
2.4 Intrinsic camera matrix and external camera calibration matrix

Finally, my estimated intrinsic camera matrix after LM optimization (without considering radial distortion) is,

$$K = \begin{bmatrix} 720.10 & 1.93 & 322.81 \\ 0 & 717.55 & 245.11 \\ 0 & 0 & 1 \end{bmatrix}$$

This shows that the principle point is at (322.81, 245.11) which is almost the center of a 640x480 image. The $45°$ ray right of the principle axis (y=0, x=z) pass through the image plane 720.10 pixels right of the center and the $45°$ ray below the principle axis (x=0, y=z) pass through the image plane 717.55 pixels below the center. So, the camera is apparently not able to capture $90°$ angle in either horizontal or vertical directions.

Captured Image	External Camera Matrix $[\mathrm{R}\vert\mathrm{t}]$ Another representation $\mathrm{R}[\mathrm{I}_{3x3}\vert - \mathrm{C}]$
	$\begin{bmatrix} 0.785 & -0.183 & 0.592 & -59.2 \\ 0.205 & 0.978 & 0.030 & -161 \\ -0.584 & 0.098 & 0.806 & 540 \end{bmatrix}$ $\begin{bmatrix} 0.785 & -0.183 & 0.592 \\ 0.205 & 0.978 & 0.030 \\ -0.584 & 0.098 & 0.806 \end{bmatrix} \left[\mathrm{I}_{3x3} \vert - \begin{pmatrix} 395 \\ 94 \\ -395 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.395 meters right and 0.094 meters down from the left top corner of the calibration sheet and 0.395 meters from the wall (-z direction).
	$\begin{bmatrix} 1.000 & 0.014 & 0.028 & -108.1 \\ -0.027 & 0.858 & 0.514 & -145.2 \\ -0.017 & -0.514 & 0.858 & 564.5 \end{bmatrix}$ $\begin{bmatrix} 1.000 & 0.014 & 0.028 \\ -0.027 & 0.858 & 0.514 \\ -0.017 & -0.514 & 0.858 \end{bmatrix} \left[\mathrm{I}_{3x3} \vert - \begin{pmatrix} 113 \\ 416 \\ -407 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.113 meters right, and 0.416 meters down from the left top corner of the calibration sheet and 0.407 meters from the wall (-z direction). **This can be verified from the image because the center of the camera along x-axis is about half the width of the paper and 1.5x of the height of the paper along y-axis.**

	$\begin{bmatrix} 1.000 & 0.004 & 0.037 & -111.8 \\ -0.003 & 1.000 & -0.038 & -133.0 \\ -0.037 & -0.038 & 0.999 & 407.6 \end{bmatrix}$ $\begin{bmatrix} 1.000 & 0.004 & 0.037 \\ -0.003 & 1.000 & -0.038 \\ -0.037 & -0.038 & 0.999 \end{bmatrix} \left[\mathrm{I}_{3x3} \middle\vert -\begin{pmatrix} 127 \\ 118 \\ -408 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.127 meters right, and 0.118 meters down from the left top corner of the calibration sheet and 0.408 meters from the wall (-z direction). **This can be verified from the image because there is no rotation between the camera coordinates and the world coordinates, and thus the *R* matrix is almost 3x3 identity matrix. The center of the camera along x-axis is about half the width of the paper and half of the height of the paper along y-axis. This matches human's impression on the camera center.**
	$\begin{bmatrix} 0.993 & -0.001 & -0.123 & -83.73 \\ -0.006 & 0.999 & -0.052 & -122.9 \\ 0.123 & 0.053 & 0.991 & 535.7.6 \end{bmatrix}$ $\begin{bmatrix} 0.993 & -0.001 & -0.123 \\ -0.006 & 0.999 & -0.052 \\ 0.123 & 0.053 & 0.991 \end{bmatrix} \left[\mathrm{I}_{3x3} \middle\vert -\begin{pmatrix} 17 \\ 94 \\ 548 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.017 meters right, and 0.094 meters down from the left top corner of the calibration sheet and 0.548 meters from the wall (-z direction). **This can be verified from the image because this figure is farther from the wall than other pictures. Thus, the last coordinate is the largest among the 4 shown images. The x- and y-axis of the camera center is very close to the world center and thus the x and y coordinates are very small.**

2.5 Radial distortion parameters (bonus requirement)

The estimated two radial distortion parameters are -2.8×10^{-7} and 1.9×10^{-12}, where all physical coordinates are all in millimeter scale.
Note that the equations in Sec 1.3 only defines more accurate projection from world coordinates to the camera coordinates. However, it is hard to remove the distortion from a camera image and transfer back to the world coordinates and thus we can no longer use reprojection error metric. Instead, we use the summed square error between the projected true corners and real true corners in the LM optimizations defined in Sec 1.3 as error metric. The summed square error is 4419.28 before the LM optimization, 840.47 after the

LM optimization without considering radial distortion, and 709.15 after the LM optimization considering radial distortion. So the summed square projection error drops by 16% with considering radial distortion.

3. Evaluation on the Own Dataset and Observations

3.1 Pre-processing

The original image dataset is consisting of 21 4032-height, 3024-width, RGB images. The "Fixed Image" is "Pic_21.jpg". I first down-sample them by a factor of 7.875 in each dimension and get 512-height, 384-width, RGB images. The purpose of this down-sampling step is to guarantee a stable result in Canny edge detector, Hough line detector and Corner detector. All the evaluations are done in the down-sampled domain. As discussed in Sec 1.2, the original intrinsic parameter K_{Org} can be retrieved from,

$$K_{Org} = \begin{bmatrix} 7.875 & 0 & 0 \\ 0 & 7.875 & 0 \\ 0 & 0 & 1 \end{bmatrix} K_{Down-sampled}$$

I then load the down-sampled images and transfer them to black-and-write images as the input of Canny edge detector. The two parameters for canny edge detectors are 200 and 400. The resulting edges are denoted as 255 (white) and the background are denoted as 0 (black).

Then I feed the resulting edges to the Hough line detector, the precision of R and θ in Hough line detector is 1 pixel and 0.5 degree. The minimum vote for a line to be detected is 50. The output is typically more than 10 horizontal lines and 8 vertical lines. Using my own MergeLine module with threshold of 12, I can perfectly detect 10 horizontal lines and 8 vertical lines in all 21 images. The horizontal lines and vertical lines can be differentiated from θ and each "parallel" line can be differentiated from R as discussed in Sec 1.1.

Then using the cross-product representation of each pair of horizontal line and vertical line as discussed in Sec 1.1, I got 80 corners to serve as the key points for calibration. They are labelled row-by-row from 1 to 80.

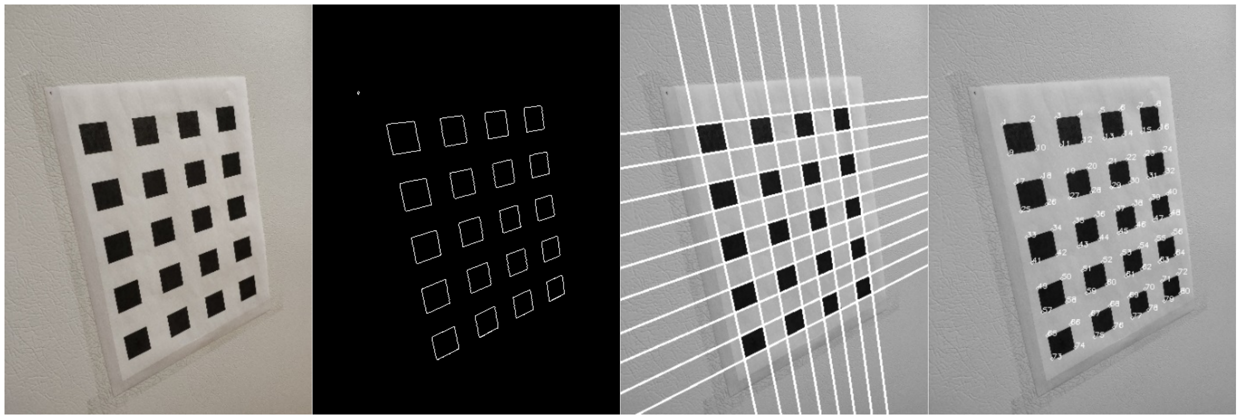


Figure 8: Pre-processing results of Pic_1.jpg - (a) original image taken from a camera (b) detected edges using canny Edge detector (c) detected lines using Hough line detector (d) 80 intersection corners with 1-80 labels.

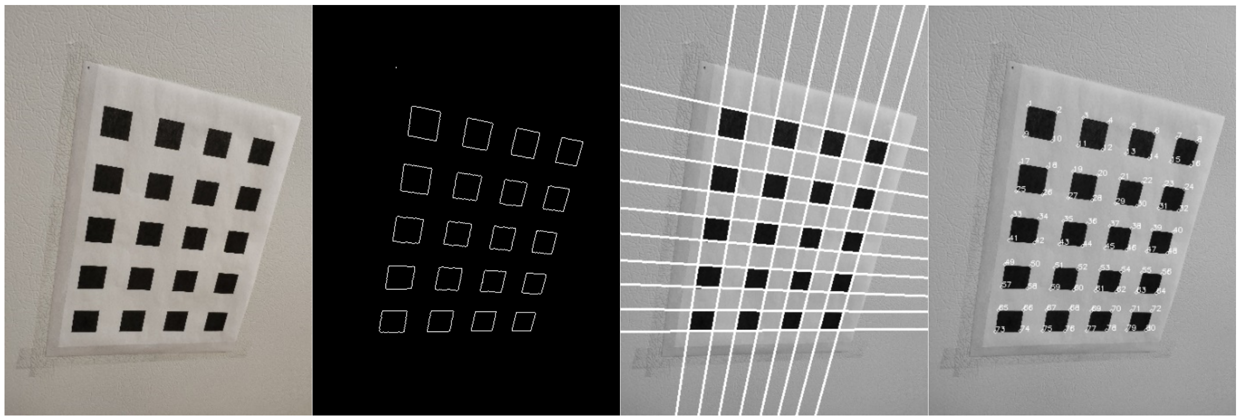

(a) (b) (c) (d)

Figure 9: Pre-processing results of Pic_2.jpg - (a) original image taken from a camera (b) detected edges using canny Edge detector (c) detected lines using Hough line detector (d) 80 intersection corners with 1-80 labels.

As shown in Fig.8 and 9, the Canny edge detector, Hough line detector, MergeLine module and corner detection module work well. All 80 corners are detected and labelled in row-by-row order.

Measuring the ground truth corners in the calibration pattern

This part is same as previous dataset. **Note that I uses millimeter as unit length in physical coordinate and thus by default any unit physical coordinate represents 1 millimeter.**

3.2 Reprojection of the corners

In this step, I re-project the 80 corners of an image to the "Fixed Image" (Pic_21.jpg) using the two projection equations in Sec 1.4. The purpose of this step is to visually examine the precision of the intrinsic parameters and external parameters after LM optimizations.

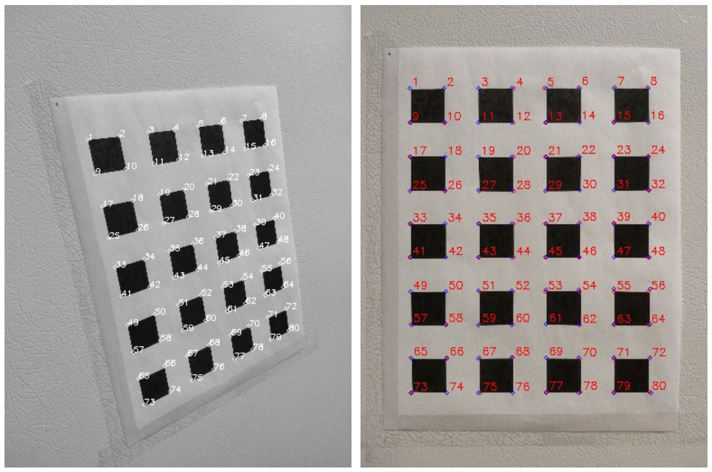
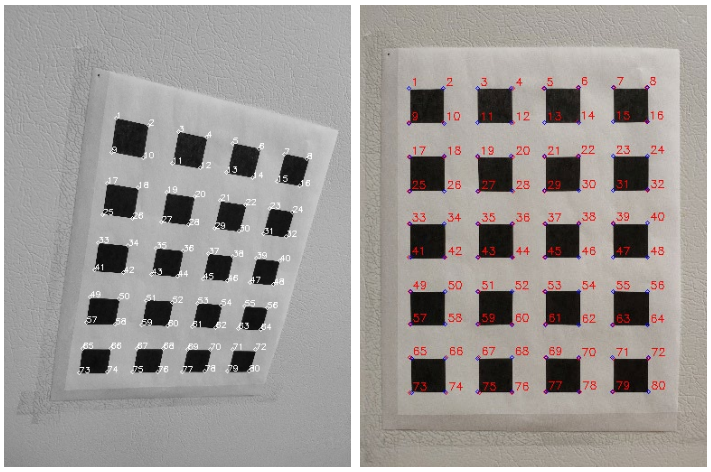
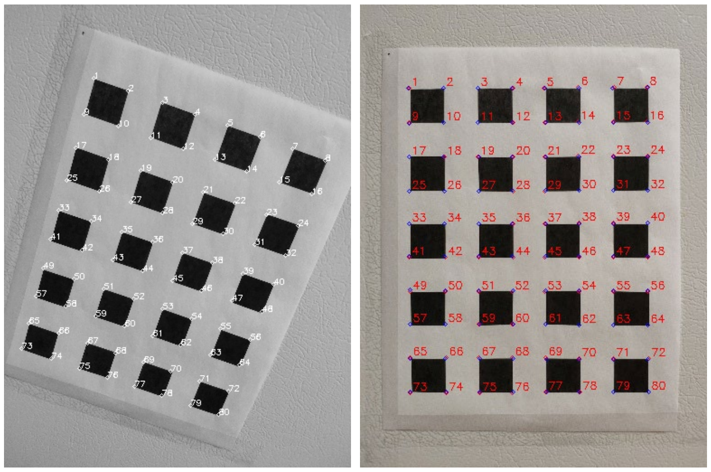

(a) (b)

Figure 10: (a) original image (Pic_1.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.02 and 0.34***



(a) *(b)*

Figure 11: (a) original image (Pic_2.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.78 and 0.19***



(a) *(b)*

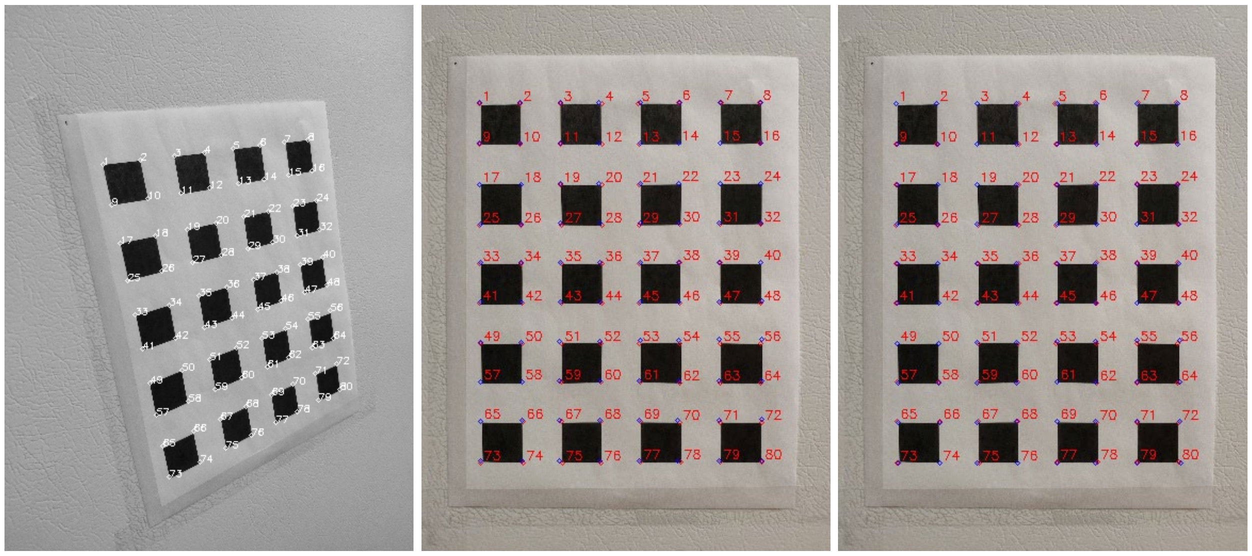
Figure 12: (a) original image (Pic_3.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red). ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.70 and 0.12***

As shown in Fig. 10, 11, and 12, all re-projected corners almost coincide with the true corners on the "Fixed Image". **The mean Euclidean Distance is within 2 pixels in all three images, which is almost negligible, and the variance is also very small (within 1). This shows that the overall calibration algorithm has achieved good accuracy.**

The mean and variance of Euclidean Distance between all re-projected corners and true corners in all 40 given images are 0.98 and 0.75. This aggregated result again proves the good performance of the camera calibration algorithm.

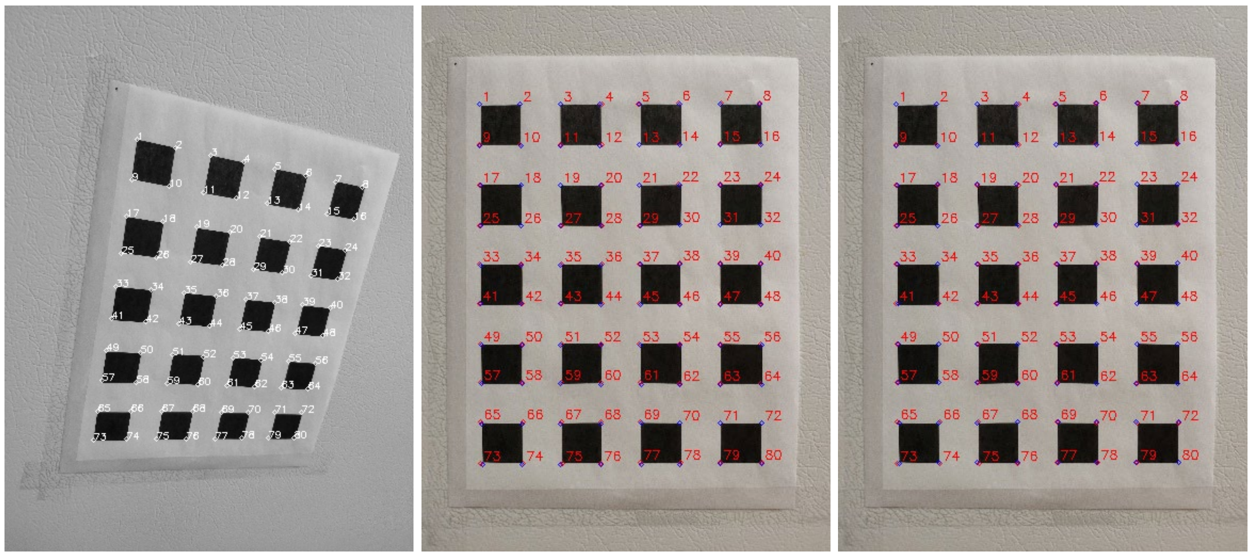
3.3 Marginal gains using LM optimization (with bonus requirement)

In this step, I re-project the 80 corners of an image to the "Fixed Image" (Pic_21.jpg) using the two projection equations in Sec 1.4, while I use the intrinsic parameters and external parameters before and after LM optimizations for comparison to show the marginal gains using LM optimizations.



(a) *(b)* *(c)*

Figure 13: (a) original image (Pic_1.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters before LM optimization. ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.50 and 0.67*** *(c) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters after LM optimization.* ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.02 and 0.34***



(a) *(b)* *(c)*

Figure 14: (a) original image (Pic_2.jpg) and its 80 corners (b) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters before LM optimization. ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 1.11 and 0.25*** *(c) re-projection on the "Fixed Image" (labelled in blue) and the 80 corners in the "Fixed Image" (labelled in red) using parameters after LM optimization.* ***The mean and variance of Euclidean Distance between re-projected corners and true corners are 0.78 and 0.19***

As shown in Fig. 13 and 14, visually we can see some small re-projection errors in some point correspondences using the parameter before LM optimization, while these errors are smaller after LM optimizations. The mean and variance of Euclidean Distance between re-projected corners and true corners are reduced as shown in the captions of each figure. **Considering all the corners in all 40 images, the mean and variance drop from 1.42 and 1.90 to 0.98 and 0.75.**

3.4 Intrinsic camera matrix and external camera calibration matrix

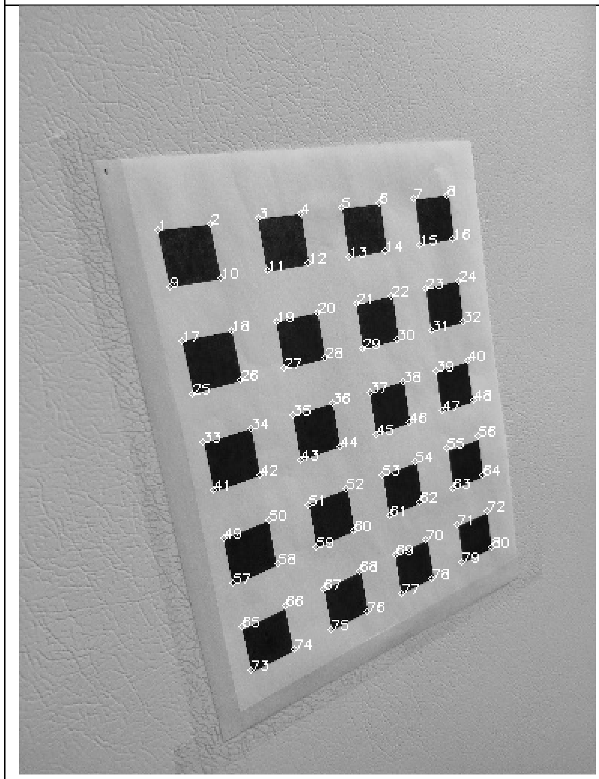
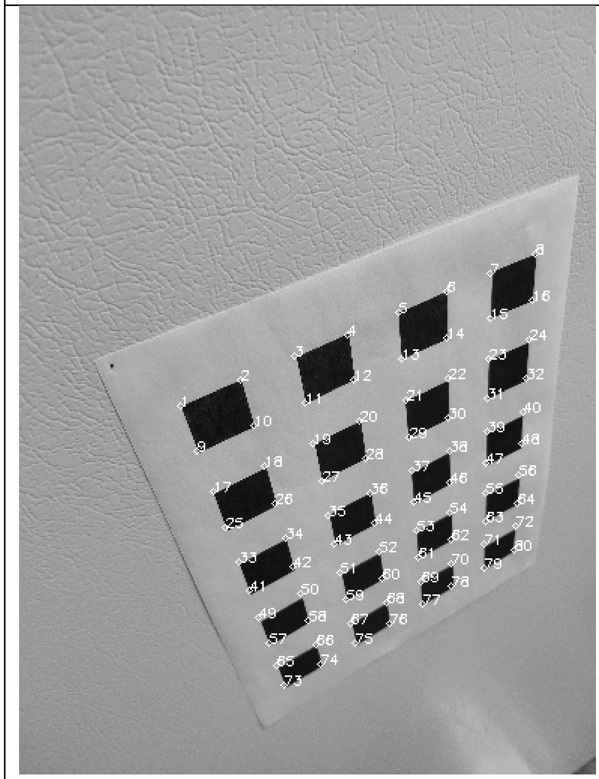
Finally, my estimated intrinsic camera matrix after LM optimization (without considering radial distortion) is,

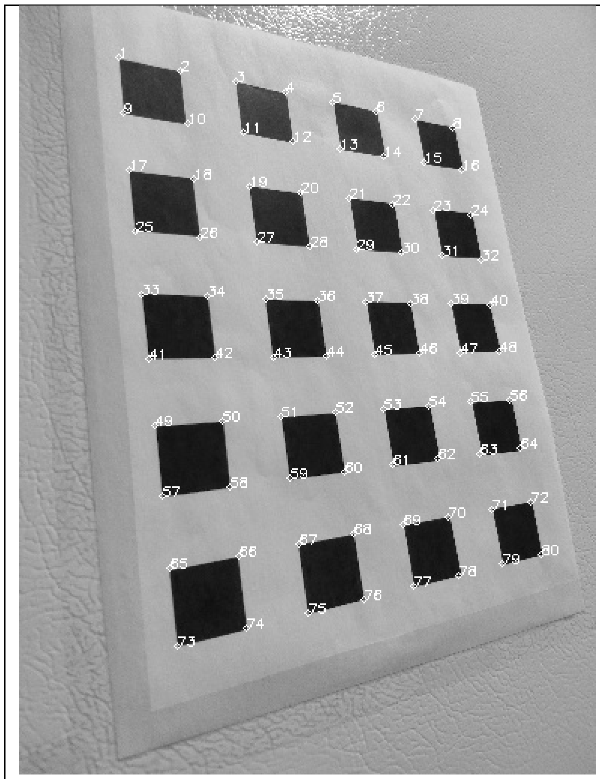
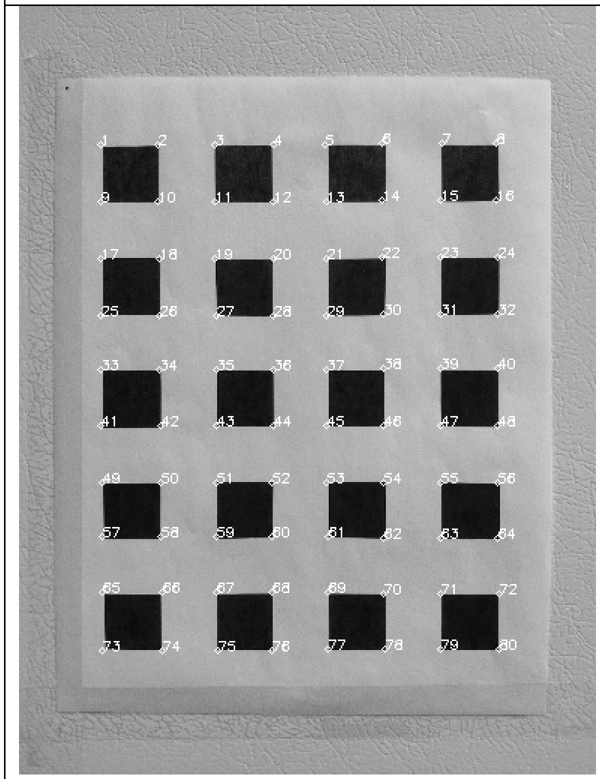
$$K = \begin{bmatrix} 418 & 0.048 & 196 \\ 0 & 416 & 258 \\ 0 & 0 & 1 \end{bmatrix}$$

This shows that the principle point is at (196, 258) which is almost the center of a 384x512 image. The 45° ray right of the principle axis (y=0, x=z) pass through the image plane 418 pixels right of the center and the 45° ray below the principle axis (x=0, y=z) pass through the image plane 416 pixels below the center. So, the camera is apparently not able to capture 90° angle in either horizontal or vertical directions.

Considering the down-sampling ratio of 7.875, the original intrinsic parameter of my camera is

$$K_{org} = \begin{bmatrix} 7.875 & 0 & 0 \\ 0 & 7.875 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 418 & 0.048 & 196 \\ 0 & 416 & 258 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3291.75 & 0.378 & 1543.5 \\ 0 & 3276 & 2031.75 \\ 0 & 0 & 1 \end{bmatrix}$$

Captured Image	External Camera Matrix [R|t] Another representation $R[I_{3x3} \mid -C]$
	$\begin{bmatrix} 0.841 & 0.166 & -0.515 & -91.9 \\ -0.246 & 0.965 & -0.090 & -93.2 \\ 0.482 & 0.202 & 0.852 & 257 \end{bmatrix}$ $\begin{bmatrix} 0.841 & 0.166 & -0.515 \\ -0.246 & 0.965 & -0.090 \\ 0.482 & 0.202 & 0.852 \end{bmatrix} \left[I_{3x3} \mid - \begin{pmatrix} -69 \\ 53 \\ -274 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.069 meters left and 0.053 meters down from the left top corner of the calibration sheet and 0.274 meters from the wall (-z direction). **Measured Ground Truth = [-40, 50, -260], it matches well.**
	$\begin{bmatrix} 0.874 & 0.151 & -0.462 & -74.35 \\ -0.397 & 0.769 & -0.500 & -8.923 \\ 0.280 & 0.621 & 0.733 & 209.2 \end{bmatrix}$ $\begin{bmatrix} 0.874 & 0.151 & -0.462 \\ -0.397 & 0.769 & -0.500 \\ 0.280 & 0.621 & 0.733 \end{bmatrix} \left[I_{3x3} \mid - \begin{pmatrix} 3 \\ -112 \\ -192 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.003 meters right, and 0.112 meters up from the left top corner of the calibration sheet and 0.192 meters from the wall (-z direction). **This can be verified from the image because the center of the camera is almost along y-axis and thus no x-axis transition. The camera is half the height of the paper above the top left corner and thus the y-axis transition is 0.112 meters.** **Measured Ground Truth = [-10, -60, -210], it matches well.**

	$\begin{bmatrix} 0.873 & 0.144 & -0.467 & -100.2 \\ -0.045 & 0.975 & 0.216 & -158.4 \\ 0.486 & -0.167 & 0.858 & 237.3 \end{bmatrix}$ $\begin{bmatrix} 0.873 & 0.144 & -0.467 \\ -0.045 & 0.975 & 0.216 \\ 0.486 & -0.167 & 0.858 \end{bmatrix} \left[\mathrm{I}_{3x3} \middle| - \begin{pmatrix} -35 \\ 209 \\ -216 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.035 meters left, and 0.209 meters down from the left top corner of the calibration sheet and 0.216 meters from the wall (-z direction). **This can be verified from the image because the camera center is truly on the left and down side of the top left corner of the image.** **Measured Ground Truth = [-35, 200, -230], it matches well.**
	$\begin{bmatrix} 0.993 & -0.001 & -0.123 & -83.73 \\ -0.006 & 0.999 & -0.052 & -122.9 \\ 0.123 & 0.053 & 0.991 & 535.7.6 \end{bmatrix}$ $\begin{bmatrix} 1.000 & 0.002 & 0.004 \\ -0.002 & 1.000 & -0.011 \\ -0.004 & 0.011 & 1.000 \end{bmatrix} \left[\mathrm{I}_{3x3} \middle| - \begin{pmatrix} 115 \\ 134 \\ -271 \end{pmatrix} \right]$ From the physical world-to-camera translation, we know that the camera is positioned 0.115 meters right, and 0.134 meters down from the left top corner of the calibration sheet and 0.271 meters from the wall (-z direction). **This can be verified from the image because there is no rotation between the camera coordinates and the world coordinates, and thus the *R* matrix is almost 3x3 identity matrix. The center of the camera along x-axis is about half the width of the paper and half of the height of the paper along y-axis. This matches human's impression on the camera center.** **Measured Ground Truth = [105, 135, -230], it matches well.**

3.5 Radial distortion parameters (bonus requirement)

The estimated two radial distortion parameters are -7.95×10^{-7} and -1.04×10^{-11}, where all physical coordinates are all in millimeter scale.
Note that the equations in Sec 1.3 only defines more accurate projection from world coordinates to the camera coordinates. However, it is hard to remove the distortion from a camera image and transfer back to the world coordinates and thus we can no longer use reprojection error metric. Instead, we use the summed square error between the projected true corners and real true corners in the LM optimizations defined in Sec 1.3 as error metric. The summed square error is 1348.88 before the LM optimization, 689.72 after the LM optimization without considering radial distortion, and 576.01 after the LM optimization considering radial distortion. So, the summed square projection error drops by 16% with considering radial distortion.

4. Source code

The assumption is that the images are placed in "Files/Dataset%d/Pic%d.jpg". All these result directory exist in the root directory: CannyResults1, CannyResults2, HoughLines1, HoughLines2, HoughPoints1, HoughPoints2, ReProjLinear1, ReProjLinear2, ReProjLM1, ReProjLM2

4.1 Preprocessing

```
# ECE 661 HW8 - Camera calibration
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import time

def Mergelines(lines, threshold): # 2D R_Sorted (R, theta)
    Merged, LastR, Group = ([], lines[0,0], [])
    for idx, line in enumerate(lines):
        R, theta = line
        if R-LastR > threshold: # Reach threshold or last line
            LastR = R
            Merged.append(np.mean(Group, axis=0)) # Average rows
            Group = [[R, theta]]
        else:
            Group.append([R, theta])
    Merged.append(np.mean(Group, axis=0))
    return Merged

def KP_Extraction(im, Th_canny1, Th_canny2, Prec_R, Prec_theta, Min_Edge, Th_mergelin
e):
    # Canny edge detection, 0=non-edge, 255=edge
    edges = cv.Canny(im, Th_canny1, Th_canny2, apertureSize=3)

    # Hough line transformation, 8 vertical lines, 10 horizontal lines
    lines = cv.HoughLines(edges,Prec_R,Prec_theta*np.pi/180,Min_Edge)
    lines = lines.reshape(lines.shape[0], lines.shape[-1])

    theta_from_horizontal = np.abs(lines[:,1]-np.pi/2) # pi/2 is horizonal
    lines_horizontal = lines[theta_from_horizontal<np.pi/4] # Horizontal, if theta < p
i/4
    slines_horizontal = lines_horizontal[lines_horizontal[:,0].argsort(kind='mergesort
')]
    mslines_horizontal = Mergelines(slines_horizontal, Th_mergeline)

    lines_vertical = lines[theta_from_horizontal>np.pi/4] # Vertical, if theta > pi/4
    # Fix the negative R where theta is close to pi (vertical lines)
    for idx, (rho, theta) in enumerate(lines_vertical):
        if theta > np.pi/2:
            lines_vertical[idx, 0] = -rho
            lines_vertical[idx, 1] = theta - np.pi # Make theta a little negative

    slines_vertical = lines_vertical[lines_vertical[:,0].argsort(kind='mergesort')]
    mslines_vertical = Mergelines(slines_vertical, Th_mergeline)
    mslines = np.vstack([mslines_horizontal,mslines_vertical])

    # Find the 80 intersecting points in row-first order
    KP = np.zeros((2, 80))
    Cnt = 0
    for msline_horizontal in mslines_horizontal:
        R_hori, theta_hori = msline_horizontal
        HC_hori = [np.cos(theta_hori), np.sin(theta_hori), -R_hori]
        for msline_vertical in mslines_vertical:
            R_vert, theta_vert = msline_vertical
            HC_vert = [np.cos(theta_vert), np.sin(theta_vert), -R_vert]

            HC_inters = np.cross(HC_hori, HC_vert)
            x_inters, y_inters = (HC_inters[0]/HC_inters[2], HC_inters[1]/HC_inters
[2])
            KP[0, Cnt], KP[1, Cnt] = (x_inters, y_inters)
            Cnt = Cnt + 1
    return KP, edges, mslines

def KP_Print(im, idx_img, KP, edges, mslines, idx_dataset):
    # Draw Hough lines
    img = im.copy()
    for x in range(len(mslines)):
        rho, theta = tuple(mslines[x])
        x0 = rho * np.cos(theta) # Anchor point
        y0 = rho * np.sin(theta)
        x1 = int(x0 + 3000*np.cos(theta+np.pi/2)) # 3000 pixels away
        y1 = int(y0 + 3000*np.sin(theta+np.pi/2))
        x2 = int(x0 - 3000*np.cos(theta+np.pi/2))
        y2 = int(y0 - 3000*np.sin(theta+np.pi/2))
        cv.line(img,(x1,y1),(x2,y2),255,2)

    # Draw intersecting points
    img2 = im.copy()
    for x in range(KP.shape[1]):
        x_inters, y_inters = (KP[0,x], KP[1,x])
        cv.circle(img2, center = (int(x_inters), int(y_inters)), radius = 2, color = 2
55)
        cv.putText(img2, "%d" %(x+1), (int(x_inters), int(y_inters)),
                   cv.FONT_HERSHEY_SIMPLEX, 0.3, (255, 255, 255), 1)

    # Write to file
    cv.imwrite("CannyResults%d/%d.png" %(idx_dataset, idx_img+1), edges)
    cv.imwrite("HoughLines%d/%d.png" %(idx_dataset, idx_img+1), img)
    cv.imwrite("HoughPoints%d/%d.png" %(idx_dataset, idx_img+1), img2)

def loadKPCal():
    x = [23, 47.2, 71.7, 96, 120, 144.7, 169, 193.2] # in mm
    y = [29.7, 54, 78.1, 102.7, 126.9, 151.1, 175.5, 200, 224.2, 248.8]
    x80 = x * 10
    y80 = []
    for n in range(10):
        y80 = y80 + [y[n]]*8
    KP_Cal = np.array([x80, y80])
    return KP_Cal
```

4.2 Linear Steps to solve camera parameters

```
def R2r(R): # (3,3) --> (3,1)
    phi = np.arccos((np.trace(R)-1)/2)
    u = np.array([[R[2,1]-R[1,2]],
                  [R[0,2]-R[2,0]],
                  [R[1,0]-R[0,1]]])
    r = phi/2/np.sin(phi)* u
    return r

def r2R(r): # (3,1) --> (3,3)
    phi = np.sqrt(np.sum(r**2))
    wx = np.array([[0, -r[2], r[1]],
                   [r[2],0,-r[0]],
                   [-r[1],r[0],0]])
    R = np.identity(3) + np.sin(phi)/phi*wx+(1-np.cos(phi))/phi/phi*np.dot(wx,wx)
    return R

def SolveH(hw):  # n rows x 4 colomns -- (h1,w1,h2,w2)
    Npairs = hw.shape[0]
    Projection = np.zeros((Npairs*2, 1))    # 12 x 1
    Coefficient = np.zeros((Npairs*2, 8))  # 12 x 8
    for index in range(Npairs):
        Projection[index*2, 0] = hw[index,2]    # x2
        Projection[index*2+1, 0] = hw[index,3]  # y2
        Coefficient[index*2, 0] = hw[index,0]  # x1
        Coefficient[index*2, 1] = hw[index,1]  # y1
        Coefficient[index*2, 2] = 1
        Coefficient[index*2, 6] = - hw[index,0] * hw[index,2]  # -x1*x2
        Coefficient[index*2, 7] = - hw[index,1] * hw[index,2]  # -y1*x2
        Coefficient[index*2+1, 3] = hw[index,0]  # x1
        Coefficient[index*2+1, 4] = hw[index,1]  # y1
        Coefficient[index*2+1, 5] = 1
        Coefficient[index*2+1, 6] = - hw[index,0] * hw[index,3]   # -x1*y2
        Coefficient[index*2+1, 7] = - hw[index,1] * hw[index,3]   # -y1*y2
    h = np.matmul(np.linalg.pinv(Coefficient), Projection)
    Homograhy = np.array(([h[0][0], h[1][0], h[2][0]],
                          [h[3][0], h[4][0], h[5][0]],
                          [h[6][0], h[7][0], 1]))
    return Homograhy

def Solve_K_Rs_Ts(KP_Cal, KPs):
    # From KP_Cal, KPs to Hs, using 8-dimentional method by assuming H_33 = 1
    Hs = [SolveH(np.hstack([np.transpose(KP_Cal), np.transpose(KP)])) for KP in KPs]

    # Construct V
    V = np.zeros((80, 6)) # 40 images, each contributing to 2 equations
    for idx_H, H in enumerate(Hs):
        h11, h12, h13 = (H[0,0], H[1,0], H[2,0])
        h21, h22, h23 = (H[0,1], H[1,1], H[2,1])
        V[idx_H*2, 0] = h11**2-h21**2
        V[idx_H*2, 1] = 2*h11*h12-2*h21*h22
        V[idx_H*2, 2] = h12**2-h22**2
        V[idx_H*2, 3] = 2*h11*h13-2*h21*h23
        V[idx_H*2, 4] = 2*h12*h13-2*h22*h23
        V[idx_H*2, 5] = h13**2-h23**2
        V[idx_H*2+1, 0] = h11*h21
        V[idx_H*2+1, 1] = h11*h22+h12*h21
        V[idx_H*2+1, 2] = h12*h22
        V[idx_H*2+1, 3] = h13*h21+h11*h23
        V[idx_H*2+1, 4] = h13*h22+h12*h23
        V[idx_H*2+1, 5] = h13*h23

    # Solve Vb=0 and w
    svd_u, svd_s, svd_vh = np.linalg.svd(V) # s in decending order
    svd_vh = np.transpose(svd_vh)
    b = svd_vh[:, -1] # Last colomn is the eigenvector to the smallest eigenvalue
    w = np.array([[b[0],b[1],b[3]], [b[1],b[2],b[4]], [b[3],b[4],b[5]]]) # K-T*K-1

    # Solve K, k
    y0 = (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+y0*(w[0,1]*w[0,2]-w[0,0]*w[1,2]))/w[0,0]
    alpha_x = np.sqrt(lamda/w[0,0])
    alpha_y = np.sqrt(lamda*w[0,0]/(w[0,0]*w[1,1]-w[0,1]**2))
    s = -w[0,1]*alpha_x**2*alpha_y/lamda
    x0 = s*y0/alpha_y-w[0,2]*alpha_x**2/lamda
    K = np.array([[alpha_x, s, x0], [0,alpha_y,y0], [0,0,1]])
    k = np.array([alpha_x, s, x0, alpha_y, y0])

    # Solve Rs, rs, ts -- list of (3,3), (3,1), (3,1)
    Rs, rs, ts = ([], [], [])
    for H in Hs:
        # Get Q, t
        K_inv = np.linalg.inv(K)
        h1, h2, h3 = (H[:,0], H[:,1], H[:,2])
        tmp_r1 = np.dot(K_inv, h1)
        xi = 1/np.sqrt(np.sum(tmp_r1**2))
        r1 = (tmp_r1*xi).reshape(3,1)
        r2 = (np.dot(K_inv, h2)*xi).reshape(3,1)
        t = (np.dot(K_inv, h3)*xi).reshape(3,1)
        r3 = (np.cross(r1,r2,axis=0)).reshape(3,1)
        ts.append(t)

        # From Q to R
        Q = np.hstack([r1,r2,r3])
        svd_u, svd_s, svd_vh = np.linalg.svd(Q)
        R = np.dot(svd_u, svd_vh)
        Rs.append(R)

        # From R to r
        r = R2r(R)
        rs.append(r)
    return K, k, Rs, rs, ts

def Mapping2D(H, Pts): # Pts is 2D numpy array (2,n)
    Pts3D = np.vstack([Pts, np.ones((1, Pts.shape[1]))])
    MappedPts3D = np.matmul(H, Pts3D)
    MappedPts2D = MappedPts3D[0:2, :] / MappedPts3D[2, :]
    return MappedPts2D

def Test_K_Rs_Ts(FixImgID, KPs, k, rs, ts): # No radial distortion is considered
    # Shape: int, list (2,80), (5,1), list (3,1), list (3,1)
    ReProjKPs = [] # Reprojected Key Points, list of (2,80)
    EDs = [] # Eur. Distances, list of (80,)
    MeanED_Images, VarED_Images = ([], []) # list of float, list of float
    MeanED_Dataset, VarED_Dataset = (0, 0) # float, float
    K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])

    # Mapping from Cal. Pattern to Fixed Image
    R_Fix = r2R(rs[FixImgID])
    H_Fix = R_Fix.copy()
    H_Fix[:,2] = ts[FixImgID].reshape(3,)
    H_Fix = np.matmul(K, H_Fix)

    # Mapping from Any Image to Cal. Pattern
    for r, t, KP in zip(rs,ts,KPs):
        R = r2R(r)
        H = R.copy()
        H[:,2] = t.reshape(3,)
        H = np.matmul(K, H)
        OriKP = Mapping2D(np.linalg.pinv(H), KP)
        ReProjKP = Mapping2D(H_Fix, OriKP)
        ReProjKPs.append(ReProjKP)
        ED = np.sqrt(np.sum((ReProjKP - KPs[FixImgID])**2, axis = 0)) # (80,)
        EDs.append(ED)
        MeanED_Images.append(np.mean(ED))
        VarED_Images.append(np.var(ED))
    MeanED_Dataset = np.mean(EDs)
    VarED_Dataset = np.var(EDs)
    return ReProjKPs, EDs, MeanED_Images, VarED_Images, MeanED_Dataset, VarED_Dataset
```

4.3 LM optimizations

```
import scipy.optimize
def RadialDis(KP, k1, k2, x0, y0): # KP: (2,n)
    # Remove radial distortions
    xs = KP[0,:]
    ys = KP[1,:]
    r2 = (xs-x0)**2 + (ys-y0)**2
    re_xs = xs + (xs-x0)*(k1*r2+k2*(r2**2))
    re_ys = ys + (ys-y0)*(k1*r2+k2*(r2**2))
    return np.vstack([re_xs, re_ys])

def Loss_Func_woRadDis(h, KP_Cal, KPs):  # Shapes: (5+6n,), (2,80), (2,80)*n
    Nimg = int((h.shape[0]-5)/6)
    k = h[0:5]
    # Reconstruct K, Rs, ts
    K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])
    rs = [h[5+x*3: 5+x*3+3].reshape(3,1) for x in range(Nimg)]
    Rs = [r2R(r) for r in rs]
    ts = [h[5+Nimg*3+x*3: 5+Nimg*3+x*3+3].reshape(3,1) for x in range(Nimg)]
    ReProjHs = [np.dot(K, np.hstack([R[:,0:1], R[:,1:2], t]))
                for R, t in zip(Rs, ts)]
    ReProjKPs = [Mapping2D(ReProjH, KP_Cal) for ReProjH in ReProjHs]
    Diff = np.array(ReProjKPs).flatten() - np.array(KPs).flatten()
    return Diff

def Loss_Func_RadDis(h, KP_Cal, KPs):  # Shapes: (5+6n+2,), (2,80), (2,80)*n
    Nimg = len(KPs)
    # Reconstruct K, Rs, ts, k1, k2
    k = h[0:5]
    (x0, y0) = (k[2], k[4])
    K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])
    rs = [h[5+x*3: 5+x*3+3].reshape(3,1) for x in range(Nimg)]
    Rs = [r2R(r) for r in rs]
    ts = [h[5+Nimg*3+x*3: 5+Nimg*3+x*3+3].reshape(3,1) for x in range(Nimg)]
    k1 = h[5+Nimg*6]
    k2 = h[5+Nimg*6+1]

    ReProjHs = [np.dot(K, np.hstack([R[:,0:1], R[:,1:2], t]))
                for R, t in zip(Rs, ts)]
    ReProjRawKPs = [Mapping2D(ReProjH, KP_Cal) for ReProjH in ReProjHs]
    ReProjKPs = [RadialDis(KP, k1, k2, x0, y0) for KP in ReProjRawKPs]
    Diff = np.array(ReProjKPs).flatten() - np.array(KPs).flatten()
    return Diff

def LM(KP_Cal, KPs, k, rs, ts, k1 = None, k2 = None):
    h_init = np.hstack([k.flatten(), np.array(rs).flatten(),
                        np.array(ts).flatten()])
    Nimg = len(rs)
    if k1 is not None: # LM with radial distortion
        h_init = np.hstack([h_init, k1, k2])
        Diff = Loss_Func_RadDis(h_init, KP_Cal, KPs)
        Cost = np.sum(Diff**2)/2
        print("LM with radial distortion starts! Initial Cost = %.2f" %Cost)
        sol = scipy.optimize.least_squares(Loss_Func_RadDis, h_init, method = 'lm',
                                           args = [KP_Cal, KPs])
        k_lmrd = sol.x[0:5]
        rs_lmrd = [sol.x[5+x*3: 5+x*3+3] for x in range(Nimg)]
        ts_lmrd = [sol.x[5+Nimg*3+x*3: 5+Nimg*3+x*3+3] for x in range(Nimg)]
        k1 = sol.x[5+Nimg*6]
        k2 = sol.x[5+Nimg*6+1]
        print("LM with radial distortion finishes! Final Cost = %.2f" %sol.cost)
        return k_lmrd, rs_lmrd, ts_lmrd, k1, k2
    else: # LM without radial distortion
        Diff = Loss_Func_woRadDis(h_init, KP_Cal, KPs)
        Cost = np.sum(Diff**2)/2
        print("LM without radial distortion starts! Initial Cost = %.2f" %Cost)

        sol = scipy.optimize.least_squares(Loss_Func_woRadDis, h_init,
                                           method = 'lm', args = [KP_Cal, KPs])
        k_lm = sol.x[0:5]
        rs_lm = [sol.x[5+x*3: 5+x*3+3] for x in range(Nimg)]
        ts_lm = [sol.x[5+Nimg*3+x*3: 5+Nimg*3+x*3+3] for x in range(Nimg)]
        print("LM without radial distortion finishes! Final Cost = %.2f" %sol.cost)
        return k_lm, rs_lm, ts_lm

def Reproj_Print(ReProjKP, KP, Cam_Img, OutDir, Idx):
    # Original Red, Reprojection points blue
    im = cv.imread(Cam_Img)
    for x in range(KP.shape[1]):
        x_inters, y_inters = (KP[0,x], KP[1,x])
        cv.circle(im, center = (int(x_inters), int(y_inters)), radius = 2, color = (0,
0,255))
        cv.putText(im, "%d" %(x+1), (int(x_inters+3), int(y_inters-3)),
                   cv.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
    for x in range(ReProjKP.shape[1]):
        x_inters, y_inters = (ReProjKP[0,x], ReProjKP[1,x])
        cv.circle(im, center = (int(x_inters), int(y_inters)), radius = 2, color = (25
5,0,0))
    Outname = "%sPic_%d.png" %(OutDir, Idx)
    cv.imwrite(Outname, im)

    EDs = np.sqrt(np.sum(((ReProjKP-KP)**2), axis = 0))
    MeanED, VarED = (np.mean(EDs), np.var(EDs))
    print("(Mean, Var) of ED in this image = (%.2f, %.2f)" %(MeanED, VarED))

def Parameter_Print(k, rs, ts, idx = [0,1,2,3]):
    # Print out k, rs, and ts
    K = np.array([[k[0], k[1], k[2]], [0, k[3], k[4]], [0, 0, 1]])
    rs_in = [rs[index] for index in idx]
    ts_in = [ts[index] for index in idx]
    Rs = [r2R(r) for r in rs_in]
    Ps = [np.hstack([R, t.reshape(3,1)]) for R, t in zip(Rs, ts_in)]
    MinusCs = [np.dot(np.linalg.pinv(R), t)*(-1) for R, t in zip(Rs, ts_in)]
    print("K = ", K)
    for P, MinusC in zip(Ps, MinusCs):
        print("P = ", P)
        np.set_printoptions(precision=3)
        print("MinusC (in meter) = ", MinusC/1000)
        print("Distance = %.3f meter" %(np.sqrt(np.sum(MinusC**2))/1000))
```

4.4 Main Function of given dataset

```
time1 = time.time()
Th_canny1, Th_canny2 = (200, 400)
Prec_R, Prec_theta, Min_Edge = (1, 0.5, 55)
Th_mergeline = 12
```

```
GT_Img = "Files/CalibrationPattern.png"
Cam_Imgs = ["Files/Dataset1/Pic_%d.jpg" %x for x in range(1,41)]  # (480H, 640W)
KP_Cal, KPs = (loadKPCal(), [])
FixImgID = 12

for idx_img in range(40):
    im = cv.imread(Cam_Imgs[idx_img], cv.IMREAD_GRAYSCALE)
    KP, edges, mslines = KP_Extraction(im, Th_canny1, Th_canny2,
                                       Prec_R, Prec_theta, Min_Edge, Th_mergeline)
    KPs.append(KP)
    KP_Print(im, idx_img, KP, edges, mslines, idx_dataset=1)

# Linear step & reprojection
print("Linear method")
K, k, Rs, rs, ts = Solve_K_Rs_Ts(KP_Cal, KPs)
np.set_printoptions(precision=2)
ReProjKPs, EDs, MeanED_Images, VarED_Images, MeanED_Dataset, VarED_Dataset = \
    Test_K_Rs_Ts(FixImgID=12, KPs=KPs, k=k, rs=rs, ts=ts)
print("(Mean, Var) of ED on the whole dataset = (%.2f, %.2f)" \
      %(MeanED_Dataset, VarED_Dataset))
Reproj_Print(ReProjKPs[0], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLinear1/
", Idx = 0)
Reproj_Print(ReProjKPs[1], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLinear1/
", Idx = 1)
print()

# LM optimization w/o radial distortion & reprojection
# Jointly optimize K,rs,ts, # parameters: 5, 3*40, 3*40
print("LM w/o radial distortion")
k_lm, rs_lm, ts_lm = LM(KP_Cal, KPs, k, rs, ts)
ReProjKPs, EDs, MeanED_Images, VarED_Images, MeanED_Dataset, VarED_Dataset = \
    Test_K_Rs_Ts(FixImgID=12, KPs=KPs, k=k_lm, rs=rs_lm, ts=ts_lm)
print("(Mean, Var) of ED on the whole dataset using LM w/o radial distortion "
      "= (%.2f, %.2f)" %(MeanED_Dataset, VarED_Dataset))
Reproj_Print(ReProjKPs[0], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM1/", I
dx = 0)
Reproj_Print(ReProjKPs[1], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM1/", I
dx = 1)
Reproj_Print(ReProjKPs[2], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM1/", I
dx = 2)
Parameter_Print(k_lm, rs_lm, ts_lm, idx = [0,6,12,36])
print()

# LM optimization w/ radial distortion
# Jointly optimize K,rs,ts,k1,k2, # parameters: 5, 3*40, 3*40, 1, 1
print("LM w/ radial distortion")
k_lmrd, rs_lmrd, ts_lmrd, k1_lmrd, k2_lmrd = LM(KP_Cal, KPs, k, rs, ts, k1=0, k2=0)
print("(k1, k2) = (%.4e, %.4e)" %(k1_lmrd, k2_lmrd))
print()

print("Time: %.2f seconds" %(time.time() - time1))
```

4.4 Main Function of given dataset

```
time1 = time.time()
Th_canny1, Th_canny2 = (200, 400)
Prec_R, Prec_theta, Min_Edge = (1, 0.5, 50)
Th_mergeline = 12
GT_Img = "Files/CalibrationPattern.png"
Cam_Imgs = ["Files/Dataset2/Pic_%d.jpg" %x
            for x in range(1,22)]  # (480H, 640W)
FixImgID = 20
```

```
KP_Cal, KPs = (loadKPCal(), [])
for idx_img in range(21):
    im = cv.imread(Cam_Imgs[idx_img], cv.IMREAD_GRAYSCALE)
    KP, edges, mslines = KP_Extraction(im, Th_canny1, Th_canny2,
                                        Prec_R, Prec_theta, Min_Edge, Th_mergeline)
    KPs.append(KP)
    KP_Print(im, idx_img, KP, edges, mslines, idx_dataset=2)

# Linear step & reprojection
print("Linear method")
K, k, Rs, rs, ts = Solve_K_Rs_Ts(KP_Cal, KPs)
np.set_printoptions(precision=2)
ReProjKPs, EDs, MeanED_Images, VarED_Images, MeanED_Dataset, VarED_Dataset = \
    Test_K_Rs_Ts(FixImgID=20, KPs=KPs, k=k, rs=rs, ts=ts)
print("(Mean, Var) of ED on the whole dataset = (%.2f, %.2f)" \
      %(MeanED_Dataset, VarED_Dataset))
Reproj_Print(ReProjKPs[0], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLinear2/
", Idx = 0)
Reproj_Print(ReProjKPs[1], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLinear2/
", Idx = 1)
print()

# LM optimization w/o radial distortion & reprojection
# Jointly optimize K,rs,ts, # parameters: 5, 3*40, 3*40
print("LM w/o radial distortion")
k_lm, rs_lm, ts_lm = LM(KP_Cal, KPs, k, rs, ts)
ReProjKPs, EDs, MeanED_Images, VarED_Images, MeanED_Dataset, VarED_Dataset = \
    Test_K_Rs_Ts(FixImgID=20, KPs=KPs, k=k_lm, rs=rs_lm, ts=ts_lm)
print("(Mean, Var) of ED on the whole dataset using LM w/o radial distortion "
      "= (%.2f, %.2f)" %(MeanED_Dataset, VarED_Dataset))
Reproj_Print(ReProjKPs[0], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM2/", I
dx = 0)
Reproj_Print(ReProjKPs[1], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM2/", I
dx = 1)
Reproj_Print(ReProjKPs[2], KPs[FixImgID], Cam_Imgs[FixImgID], OutDir = "ReProjLM2/", I
dx = 2)
Parameter_Print(k_lm, rs_lm, ts_lm, idx = [0,5,10,20])
print()

# LM optimization w/ radial distortion
# Jointly optimize K,rs,ts,k1,k2, # parameters: 5, 3*40, 3*40, 1, 1
print("LM w/ radial distortion")
k_lmrd, rs_lmrd, ts_lmrd, k1_lmrd, k2_lmrd = LM(KP_Cal, KPs, k, rs, ts, k1=0, k2=0)
print("(k1, k2) = (%.4e, %.4e)" %(k1_lmrd, k2_lmrd))
print()

print("Time: %.2f seconds" %(time.time() - time1))
```