Homework 8

Alexandre Olivé Pellicer
aolivepe@purdue.edu

1 Theory Questions

1.1 Question 1

Why is the following theoretical observation fundamental to Zhang's algorithm for camera calibration?

The observation that the calibration pattern samples the Absolute Conic Ω_∞ at two Circular Points is fundamental to Zhang's algorithm because it allows the extraction of intrinsic camera parameters. The images of these two points fall on the conic ω (the camera image of the Absolute Conic Ω_∞) in the camera image plane. Each of these two points must obey the conic constraint $\mathbf{x}^T \omega \mathbf{x} = 0$. When plugging the coordinates of the two image points in the conic constraint equations, we get $\mathbf{h}_1^T \omega \mathbf{h}_1 = \mathbf{h}_2^T \omega \mathbf{h}_2$ and $\mathbf{h}_1^T \omega \mathbf{h}_2 = 0$. Therefore, given $\mathbf{h}_1$ and $\mathbf{h}_2$ for several positions of the camera, we can estimate ω and from there estimating $\mathbf{K}$, the intrinsic camera parameters. Furthermore, the Absolute Conic Ω_∞ exists independently of the camera's orientation or position. Its Circular Points are invariant under Euclidean transformations, making them essential for calibration.

To sum up, in Zhang's algorithm, this property is used to compute the homography between the camera image plane and the calibration plane. By leveraging the relationship between Ω_∞ and the homography, intrinsic parameters of the camera, such as focal length and principal point, can be derived without knowing the exact 3D coordinates of the pattern, only requiring its 2D structure.

1.2 Question 2

How would you derive the algebraic form of ω from Ω_∞?

The image of the Absolute Conic Ω_∞ on the camera plane is denoted as ω. We can derive its algebraic form doing:

$$\omega = \mathbf{K}^{-T} \Omega_\infty \mathbf{K}^{-1} = \mathbf{K}^{-T} \mathbf{K}^{-1}$$

where:

- Ω_∞ is the 3×3 identity matrix
- ω is the projection of the Absolute Conic in 3D space (Ω_∞) onto the camera image plane.
- $\mathbf{K}$ is the camera's intrinsic matrix that maps points from the world coordinates to the camera image plane. (More explanation about this matrix is added later in the report)

This would be a long version of the answer following the notes from Lecture 20:

The Absolute Conic Ω_∞ is defined by the direction vectors $\mathbf{x}_d$ that obey $\mathbf{x}_d^T I_{3\times3} \mathbf{x}_d = 0$. We know that under a homography H^d, a conic C transforms as $C' = H^{-T} C H^{-1}$. Since the image formation from the direction vectors $\mathbf{x}_d$ to the pixels $\mathbf{x}$ is the homography $H = KR$, ω is given by:

$$\begin{aligned}\omega &= H^{-T} \Omega_\infty H^{-1} = H^{-T} I_{3\times3} H^{-1} = (KR)^{-T} (KR)^{-1} \\ &= ((KR)^T)^{-1} (KR)^{-1} = (R^T K^T)^{-1} (KR)^{-1} \\ &= K^{-T} R^{-T} R^{-1} K^{-1} = K^{-T} (RR^{-T})^{-1} K^{-1} \\ &= K^{-T} K^{-1} \end{aligned} \tag{1}$$

The actual pixels on the image conic ω would be $\mathbf{x}^T \omega \mathbf{x} = 0$.

Can you prove that ω does not contain any real pixel locations?

Any point $\mathbf{x}$ in the conic must satisfy $\mathbf{x}^T\omega\mathbf{x} = 0$. Since ω is derived from the expression $\omega = \mathbf{K}^{-T}\Omega_\infty\mathbf{K}^{-1}$, ω is positive definite.

For any real point $\mathbf{x}$, the equation $\mathbf{x}^T\omega\mathbf{x} = 0$ can only have imaginary solutions because ω is positive definite, meaning it cannot be zero for any real-valued vector $\mathbf{x}$. Thus, ω does not intersect with the real image plane and does not correspond to any actual pixel locations. Therefore, ω does not contain any real pixel locations. This is a critical theoretical result because it shows that while ω is not directly observable in real images, its properties can still be used to estimate the camera's intrinsic parameters through multiple views of the calibration pattern.

2 Implementation Details

2.1 Corner Detection

The steps described in this section are applied in all the images of the dataset. This is the prepocessing of the input images that we do in order to obtain the corners of each of the squares of the calibration pattern:

- **Canny Edge Detection:** First we convert the input image to gray scale and use the cv2.Canny() function from OpenCV to get the edges of the black squares of the calibration pattern. We experimentally found out that the parameters that performed better where when using as minimum threshold 300 and maximum threshold 400
- **Hough Transform:** We use the cv2.HoughLines() function from OpenCV to get the vertical and horizontal lines that composes the calibration pattern. We set the threshold parameter to 50. Since the Canny edge detector approach is not perfect at pixel level, after using the Hough Transform to get the lines, we will get multiple lines for each border of the squares. This is not the desired behavior since there is only one true line per side. Therefore, we implement an approach to group lines that should correspond to a unique true line and get the final line from that group as the average. We first separate vertical and horizontal. We classify a line as horizontal or as vertical depending on the value of θ given by the Hough Transform. The lines corresponding to the same group will have a similar ρ which is given by the Hough Transform. Therefore, we group vertical and horizontal lines according to how similar is their ρ. Finally, we average the grouped lines to get a final true line. We end up getting 10 horizontal true lines and 8 vertical true lines.
- **Corner Correspondences:** We get the corners of the calibration pattern as the intersection between horizontal and vertical lines. Therefore, we will get 80 intersections, 4 corners for each of the 20 squares of the pattern. In order to generate the world coordinates, we consider that the calibration pattern is in the $Z = 0$ plane, the first corner is at (0, 0) and that the distance between corners is 10.

2.2 Zhang's Algorithm

In this homework we have used Zhang's algorithm for camera calibration. We have assumed that we have been using a pin-hole camera (i.e. we will estimate all the 5 intrinsic parameters and the 6 extrincsic parameters that determine the position and orientation of the camera with respect to a reference world coordinate system). In this section we do an explanation of this algorithm.

We use the calibration pattern provided in the instructions. It is assumed to be in the $Z = 0$ plane of the world frame. The homogeneous representation of a pixel coordinates $\mathbf{x} = (x, y, w)^T$ and the homogeneous representation of the corresponding world coordinates $\mathbf{x}_M = (x, y, z, w)$ are related by the following equation

$$\mathbf{x} = K\begin{bmatrix}R|t\end{bmatrix}\begin{bmatrix}x\\y\\0\\w\end{bmatrix} = H\mathbf{x}_M \tag{2}$$

where:

- K is the camera intrinsic parameter
- R is the world-to-camera rotation matrix
- t is the world-to-camera translation vector
- H is the homography
- $\mathbf{x}_M = [x, y, w]^T$

Note that the homography H is estimated using the corners estimated from the corner detection approach that we have used in this homework explained in the previous section. We can write homography H as $H = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$

The image of the Absolute Conic Ω_∞ is given by $\omega = K^{-T}K^{-1}$. And the two circular points on the image conic ω give us two equations

$$\mathbf{h}_1^T \omega \mathbf{h}_1 = \mathbf{h}_2^T \omega \mathbf{h}_2 \tag{3}$$

$$\mathbf{h}_1^T \omega \mathbf{h}_2 = 0 \tag{4}$$

ω is a 3×3 symmetric matrix which can be written as:

$$\omega = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{12} & \omega_{22} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix}, \tag{5}$$

See that there are only 6 unknowns in ω.

Given N images of the calibration pattern from different angles, we can calculate the set of homographies that relates the world coordinates with the coordinates of the calibration pattern of each of the images taken from different angles and positions. We end up getting N homographies that they are obtained using Singular Value Decomposition taking the right column vector of V.

Given an homography H:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{6}$$

we can rewrite Equations 3 and 4 as:

$$\begin{bmatrix} h_{11}^2 - h_{12}^2 \\ 2h_{11}h_{21} - 2h_{12}h_{22} \\ 2h_{11}h_{31} - 2h_{12}h_{32} \\ h_{21}^2 - h_{22}^2 \\ 2h_{21}h_{31} - 2h_{22}h_{32} \\ h_{31}^2 - h_{32}^2 \end{bmatrix}^T \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{22} \\ w_{23} \\ w_{33} \end{bmatrix} = 0 \tag{7}$$

$$\begin{bmatrix} h_{11}h_{12} \\ h_{11}h_{22} + h_{12}h_{21} \\ h_{11}h_{32} + h_{12}h_{31} \\ h_{21}h_{22} \\ h_{21}h_{32} + h_{22}h_{31} \\ h_{31}h_{32} \end{bmatrix}^T \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{22} \\ w_{23} \\ w_{33} \end{bmatrix} = 0 \tag{8}$$

Using SVD, we can solve this set of homogeneous equations and end up getting ω.

2.3 Estimating the intrinsic parameters of the camera

The intrinsic parameters of the camera are contained in the matrix K which can be written as:

$$\boldsymbol{K} = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Each of these intrinsic parameters can be calculated as:

$$y_0 = \frac{-w_{11}w_{23} + w_{12}w_{13}}{w_{11}w_{22} - w_{12}^2} \tag{10}$$

$$\lambda = w_{33} - \frac{w_{13}^2 + y_0(-w_{11}w_{23} + w_{12}w_{13})}{w_{11}} \tag{11}$$

$$a_x = \sqrt{\frac{\lambda}{w_{11}}} \tag{12}$$

$$a_y = \sqrt{\frac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}} \tag{13}$$

$$s = \frac{a_x^2 a_y w_{12}}{\lambda} \tag{14}$$

$$x_0 = \frac{-a_x^2 w_{13}}{\lambda} + \frac{s y_0}{a_y} \tag{15}$$

2.4 Estimating the extrinsic parameters of the camera

R and t are the extrinsic parameters.

Given an homography $H = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$ we can estimate $R = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3]$ and t as follows (ξ is a scale factor):

$$\xi = \frac{1}{\|K^{-1}h_1\|} \tag{16}$$

$$\mathbf{r}_1 = \xi K^{-1} h_1 \tag{17}$$

$$\mathbf{r}_2 = \xi K^{-1} h_2 \tag{18}$$

$$\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2 \tag{19}$$

$$t = \xi K^{-1} h_3 \tag{20}$$

To ensure that R is orthogonal, we perform SVD such that $R = UDV^T$, and then redefine R as $R = UV^T$.

2.5 Refining the Calibration Parameters

The estimations K, R and t will give us some good result. Nevertheless, this result can be improved by refining K, R and t using a non-linear least squares optimization approach.

We project the points from world coordinated to image coordinates using the actual K, R and t for different images. We compute the Euclidean distance between the projected points and the actual points. We sum all the distances. This is the cost function that we use for the non-linear least squares optimization approach. We can write the cost function as:

$$d^2 = \sum_i \sum_j \|x_{ij} - \hat{x}_{ij}\|^2 = \sum_i \sum_j \|x_{ij} - K \begin{bmatrix} r_{i1} & r_{i2} & t_i \end{bmatrix} x_{ij}\|^2 \tag{21}$$

where:

- x_{ij} is each of the actual points
- $\hat{x}_{ij}$ is each of the projected points

Before applying the LM optimization algorithm, it is important to modify the representation of the rotation matrix R. Following the theory from Lecture 21, in any optimization algorithm, the number of variables used to represent an entity must equal the DoF of the entity. The rotation matrix has 9 elements but only 3 degrees of freedom (DoF). We need a 3-parameter representation of the rotation matrix. We use the Rodrigues Representation in which a rotation in 3D is expressed as a vector $\tilde{\mathbf{w}}$, which is computed as:

$$\tilde{\mathbf{w}} = \frac{\varphi}{2\sin\varphi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{22}$$

where

$$\varphi = \cos^{-1}\left(\frac{\text{trace}(R) - 1}{2}\right) \tag{23}$$

In order to go from $\tilde{\mathbf{w}}$ back to R we can do the following operations:

$$W = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix} \tag{24}$$

$$R = e^W = I_{3\times 3} + \frac{\sin\varphi}{\varphi}W + \frac{1-\cos\varphi}{\varphi^2}W^2 \tag{25}$$

where $\varphi = \|\mathbf{w}\|$.

2.6 Radial Distortion (Extra Credit)

In practical applications, real-world cameras often exhibit a phenomenon known as radial distortion, where straight lines in the scene appear curved in the captured image. This effect arises due to the inherent imperfections in the lens design, which cause light rays to deviate from their ideal pinhole model trajectory as they pass through the lens. This distortion can be corrected using:

$$\hat{x}_{\text{rad}} = \hat{x} + (\hat{x} - x_0)\left(k_1 r^2 + k_2 r^4\right) \tag{26}$$

$$\hat{y}_{\text{rad}} = \hat{y} + (\hat{y} - y_0)\left(k_1 r^2 + k_2 r^4\right) \tag{27}$$

where:

- $(\hat{x}, \hat{y})$ are the projected pixel coordinates before radial distortion correction
- $(\hat{x}_{\text{rad}}, \hat{y}_{\text{rad}})$ are the projected pixel coordinates after radial distortion correction

Note that the values of k_1 and k_2 are calculated using the LM algorithm.

The parameters k_1 and k_2, which characterize the radial distortion, are refined together with K, R and t also following the approach explained in Section 2.5.

2.7 Creating Our Dataset

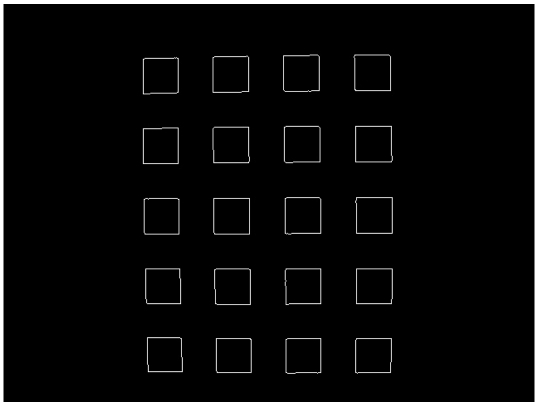
We have built a dataset with a total of 21 images of the calibration pattern provided in the instructions which was printed. We have used an iPhone 12. The focal length was set to 26 mm. The distance between the camera and the "Fixed Image" was 32.4 cm approximately. With a ruler we measure that the side of the squares is 2.2 cm. We have set this distance to be 10 in digital. Therefore, the digital distance from the center of projection to the "Fixed Image" in digital would be:

$$\frac{(2.6 + 32.4)10}{2.2} = 159.1 \tag{28}$$

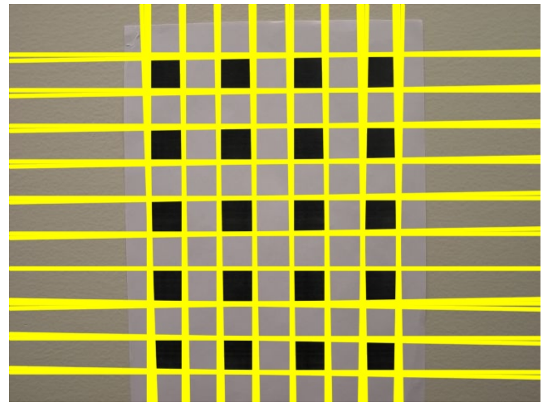
3 Obtained results

3.1 Given Dataset

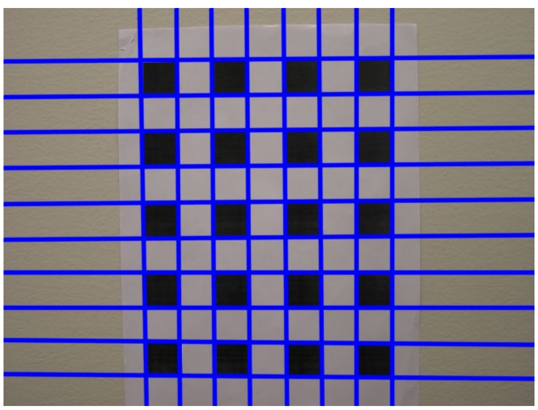
Figure 1 and 2 show the 4 images resulting from the preprocessing of the images in the dataset. It contains the edges after using the Canny edge detector, the multiple lines resulting from the Hough Transform, the final selected lines and the final intersection points.



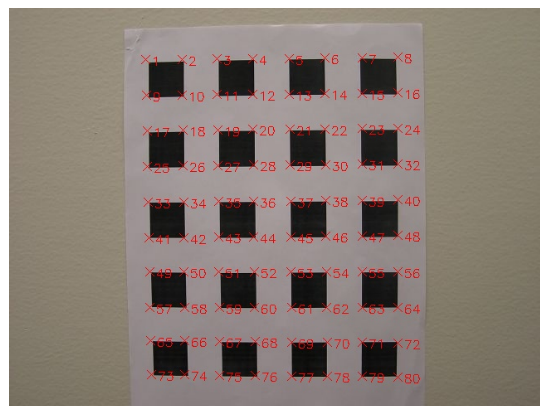
(a) Edges after using the Canny edge detector



(b) Lines resulting from the Hough Transform

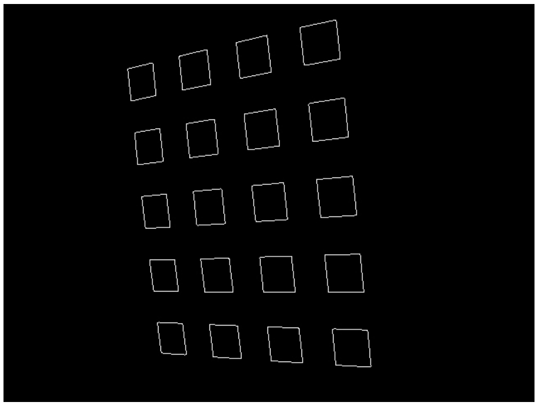


(c) Final selected lines

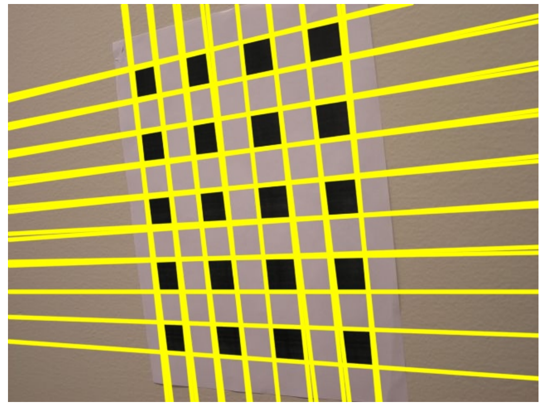


(d) Final intersection points

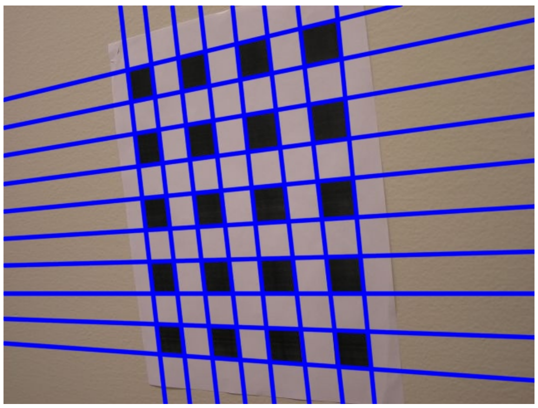
Figure 1: Preprocessing of image 4 from the given dataset.



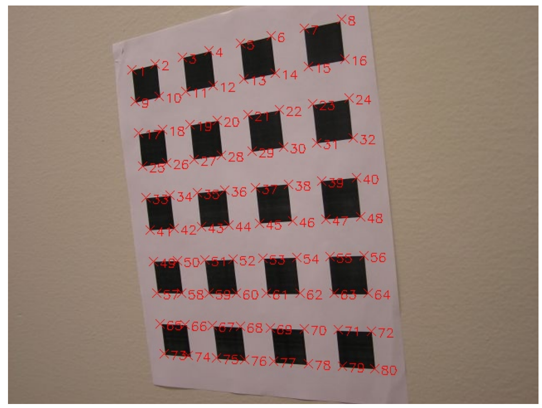
(a) Edges after using the Canny edge detector



(b) Lines resulting from the Hough Transform



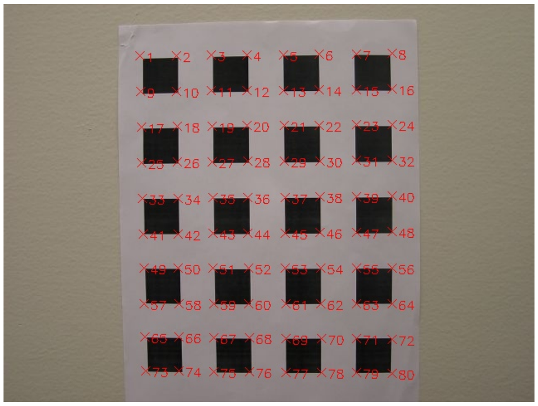
(c) Final selected lines



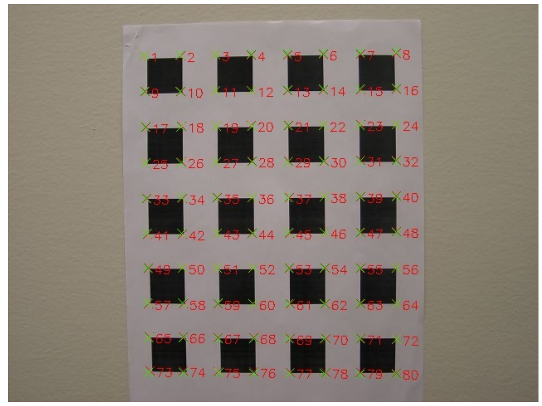
(d) Final intersection points

Figure 2: Preprocessing of image 10 from the given dataset.

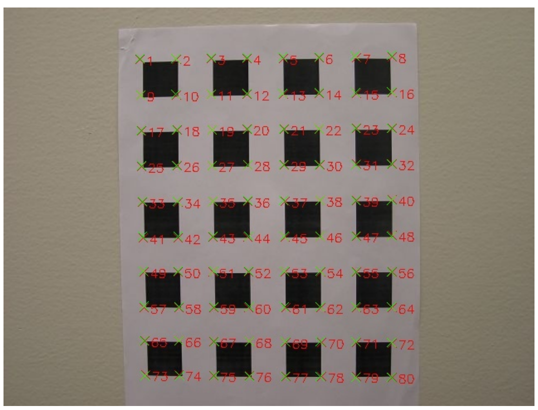
Figures 3 and 4 show the projection results at the beginning and after the refinements. See that reprojected corners are printed in green color. Red corners correspond to ground truth corners. Reprojected corners are drawn above ground truth corners. Therefore, not seeing the ground truth corner means almost perfect reprojection.



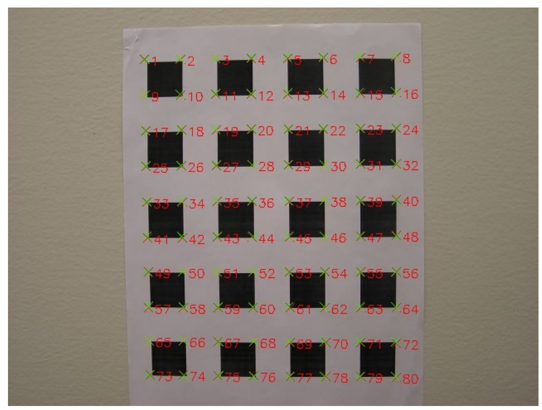
(a) Original corners



(b) Initial projection



(c) Projection after LM



(d) Projection after LM with radial distortion

Figure 3: Comparison of the projection of the world coordinates onto the pattern from image 4 in the given dataset

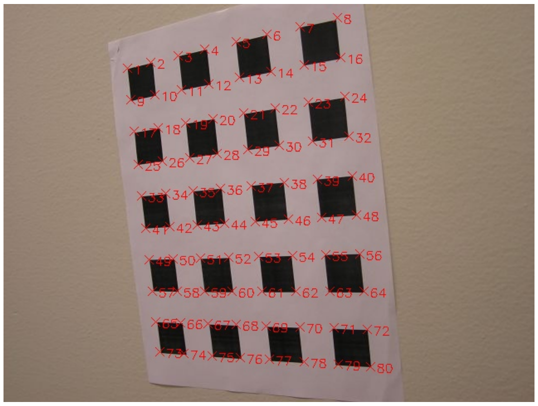
These are the camera matrix, rotation matrix and translation matrix at the beginning and after the refinements:

$$K_{init} = \begin{bmatrix} 717.46 & 0.58 & 317.77 \\ 0 & 714.16 & 237.40 \\ 0 & 0 & 1 \end{bmatrix} K_{LM} = \begin{bmatrix} 722.42 & 1.73 & 321.43 \\ 0 & 719.71 & 238.28 \\ 0 & 0 & 1 \end{bmatrix} K_{radial} = \begin{bmatrix} 728.05 & 1.72 & 319.50 \\ 0 & 725.66 & 238.89 \\ 0 & 0 & 1 \end{bmatrix} \tag{29}$$

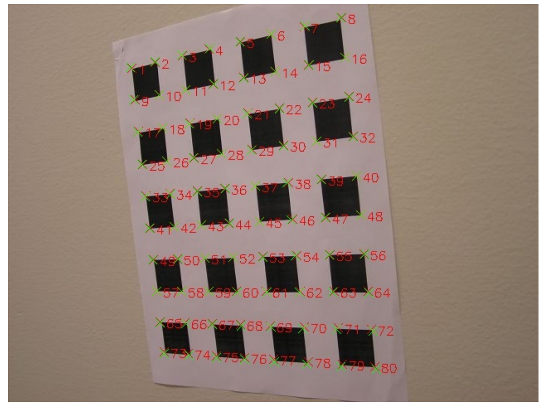
$$R_{init} = \begin{bmatrix} 0.999 & 0.006 & 0.041 \\ -0.004 & 0.999 & -0.035 \\ -0.041 & 0.035 & 0.998 \end{bmatrix} R_{LM} = \begin{bmatrix} 0.999 & 0.005 & 0.037 \\ -0.003 & 0.999 & -0.038 \\ -0.037 & 0.038 & 0.998 \end{bmatrix} R_{radial} = \begin{bmatrix} 0.999 & 0.005 & 0.037 \\ -0.004 & 0.999 & -0.034 \\ -0.037 & 0.034 & 0.998 \end{bmatrix} \tag{30}$$

$$t_{init} = \begin{bmatrix} -35.57 & -40.76 & 166.52 \end{bmatrix} t_{LM} = \begin{bmatrix} -36.36 & -41.12 & 167.82 \end{bmatrix} t_{radial} = \begin{bmatrix} -35.91 & -41.27 & 168.08 \end{bmatrix} \tag{31}$$

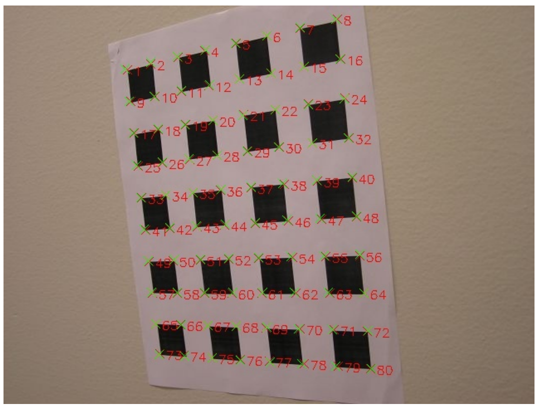
$$k_1 = -2.939e-7 \qquad k_2 = 1.912e-12 \tag{32}$$



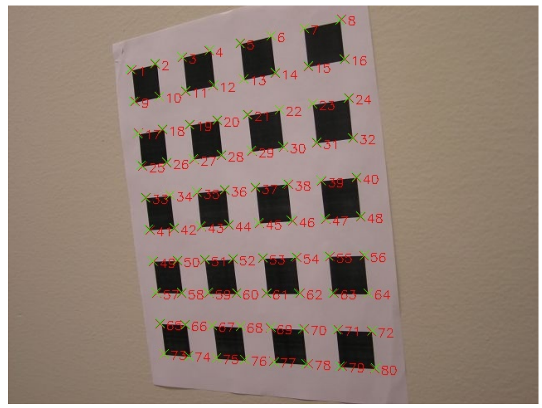
(a) Original corners



(b) Initial projection



(c) Projection after LM



(d) Projection after LM with radial distortion

Figure 4: Comparison of the projection of the world coordinates onto the pattern from image 10 in the given dataset

These are the camera matrix, rotation matrix and translation matrix at the beginning and after the refinements (matrices Ks are the same as stated for image 4):

$$R_{init} = \begin{bmatrix} 0.868 & 0.106 & 0.484 \\ -0.069 & 0.993 & -0.094 \\ -0.491 & 0.048 & 0.869 \end{bmatrix} R_{LM} = \begin{bmatrix} 0.874 & 0.105 & 0.474 \\ -0.069 & 0.993 & -0.091 \\ -0.480 & 0.047 & 0.875 \end{bmatrix} R_{radial} = \begin{bmatrix} 0.872 & 0.105 & 0.476 \\ -0.069 & 0.993 & -0.093 \\ -0.482 & 0.048 & 0.874 \end{bmatrix} \tag{33}$$

$$t_{init} = \begin{bmatrix} -43.432 & -41.090 & 182.717 \end{bmatrix} t_{LM} = \begin{bmatrix} -44.486 & -41.589 & 184.471 \end{bmatrix} t_{radial} = \begin{bmatrix} -44.004 & -41.746 & 184.640 \end{bmatrix} \tag{34}$$

$$k_1 = -2.939e-7 \qquad k_2 = 1.912e-12 \tag{35}$$

Table 1 shows the quantitative evaluation of the projection error

Metric	Image 4	Image 10
Initial error mean	1.01774	1.5646
Initial error variance	0.3092	0.7657
Error mean after LM	0.8693	1.0625
Error variance after LM	0.1785	0.2874
Error mean after LM + radial	0.7890	0.9558
Error variance after LM + radial	0.1652	0.2662

Table 1: Error mean and variance for images 4 and 10 of the given dataset

Figure 5 shows the camera poses that has been used in order to create the given dataset in the instructions. The black box simulates the position of the calibration pattern.

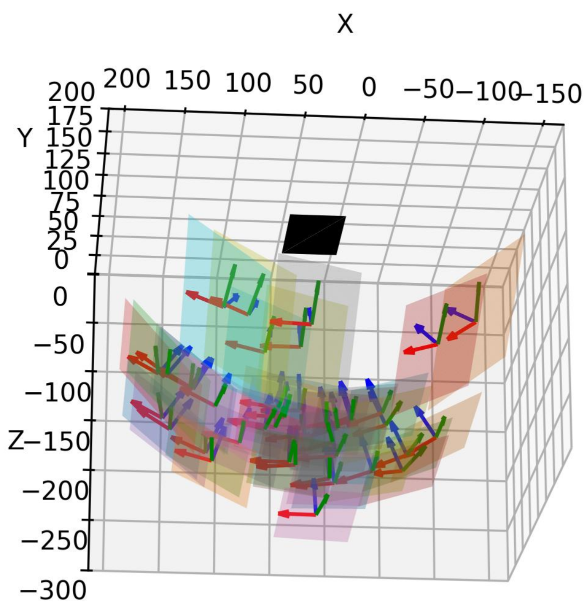
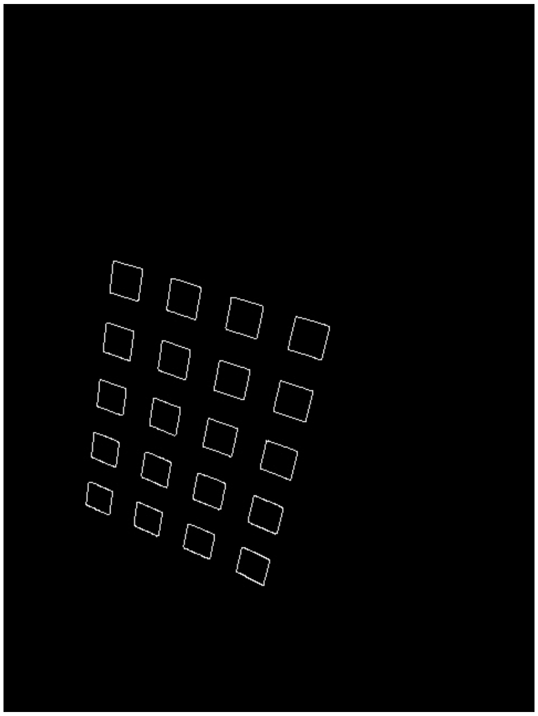


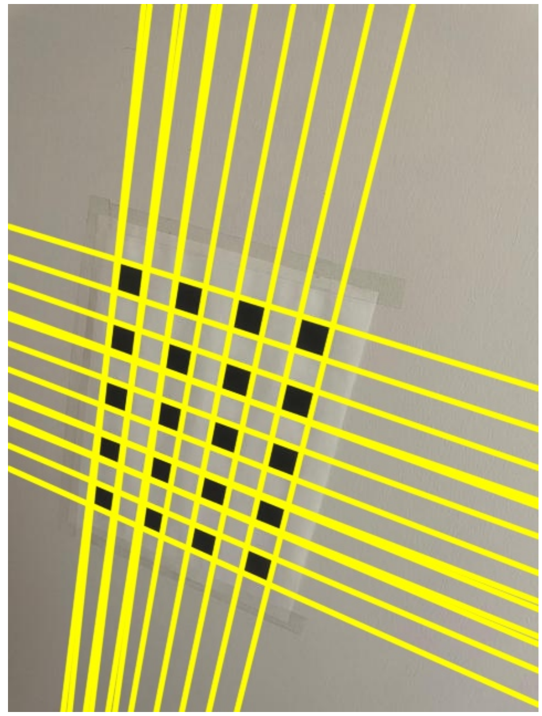
Figure 5: Camera poses to create the images of the calibration pattern in the given dataset

3.2 My own Dataset

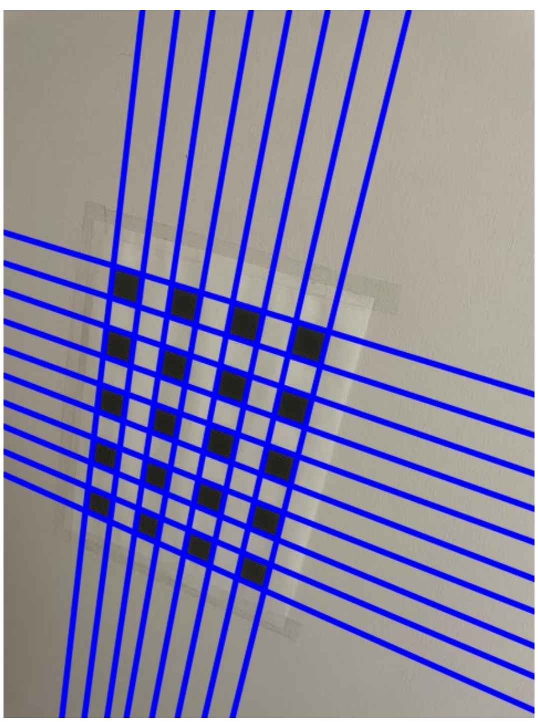
Figure 6 and 7 show the 4 images resulting from the preprocessing of the images in the dataset. It contains the edges after using the Canny edge detector, the multiple lines resulting from the Hough Transform, the final selected lines and the final intersection points.



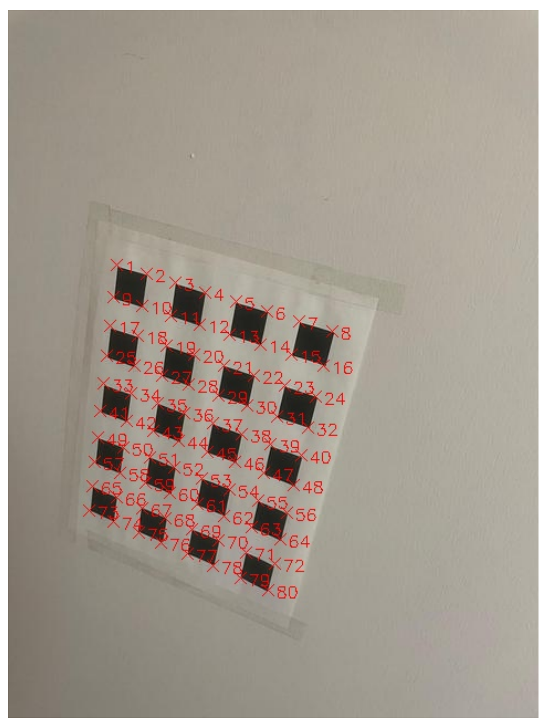
(a) Edges after using the Canny edge detector



(b) Lines resulting from the Hough Transform

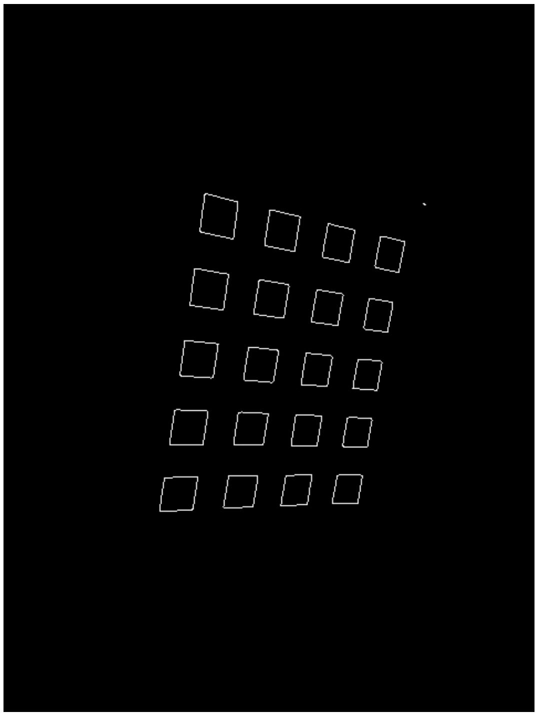


(c) Final selected lines

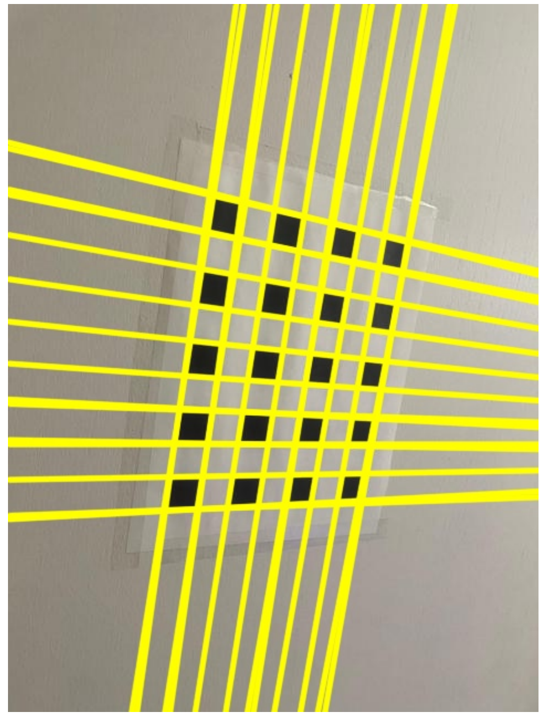


(d) Final intersection points

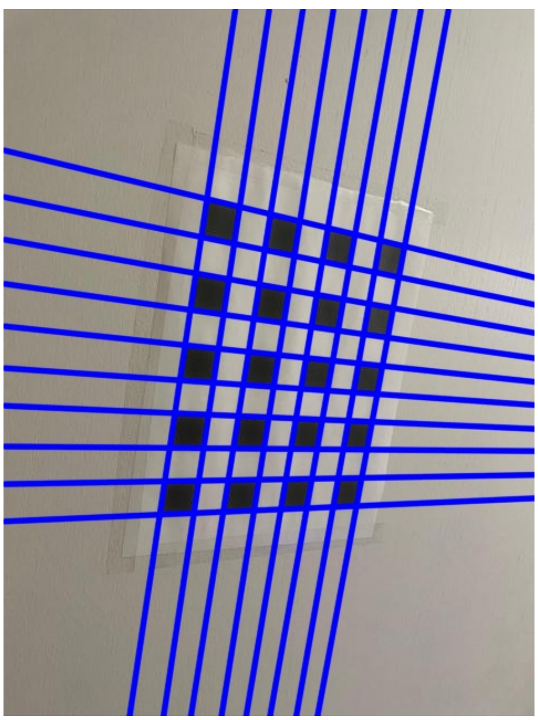
Figure 6: Preprocessing of image 4 from my dataset.



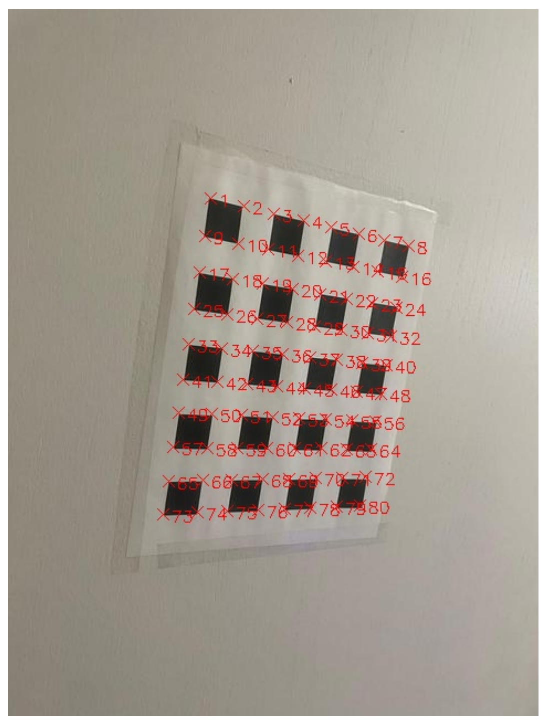
(a) Edges after using the Canny edge detector



(b) Lines resulting from the Hough Transform



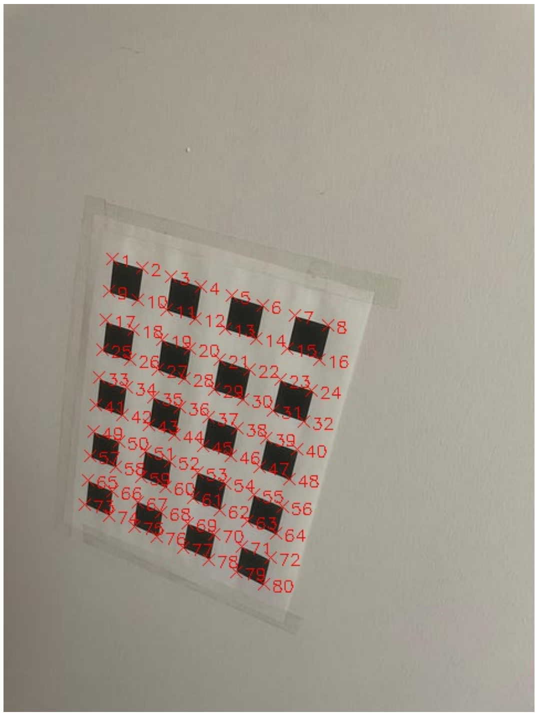
(c) Final selected lines



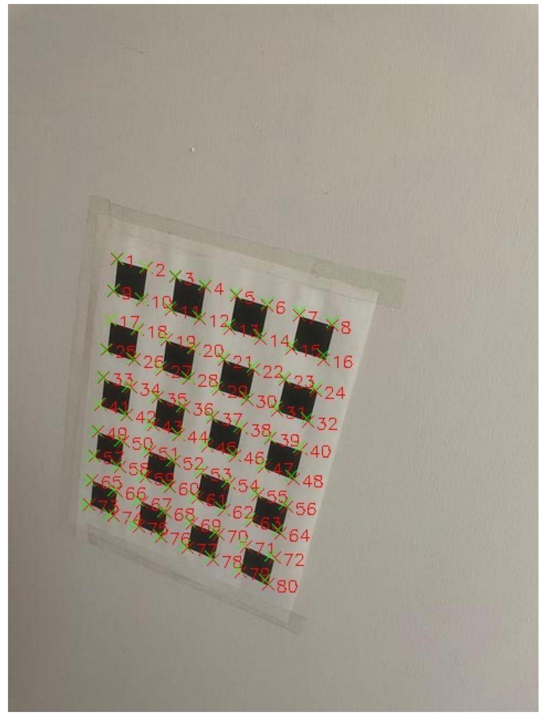
(d) Final intersection points

Figure 7: Preprocessing of image 10 from my dataset.

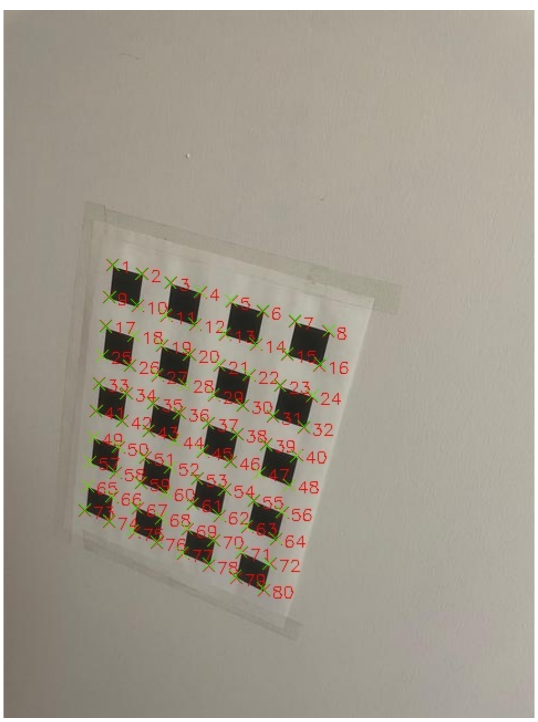
Figures 8 and 9 show the projection results at the beginning and after the refinements. See that reprojected corners are printed in green color. Red corners correspond to ground truth corners. Reprojected corners are drawn above ground truth corners. Therefore, not seeing the ground truth corner means almost perfect reprojection.



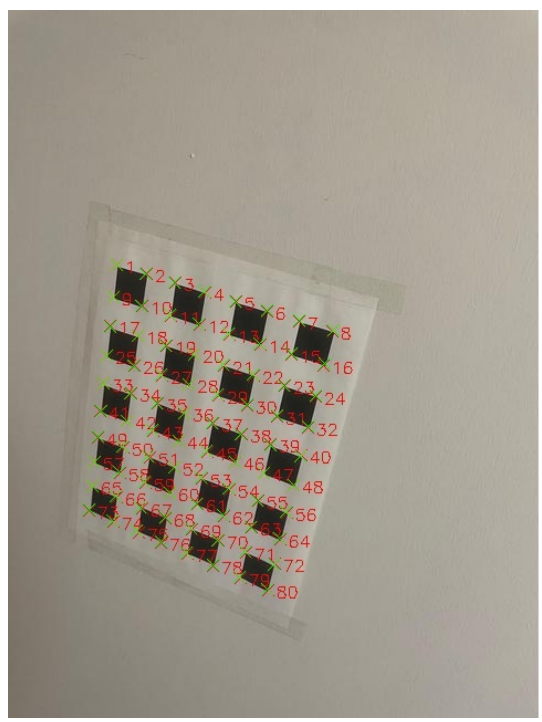
(a) Original corners



(b) Initial projection



(c) Projection after LM



(d) Projection after LM with radial distortion

Figure 8: Comparison of the projection of the world coordinates onto the pattern from image 4 in my dataset

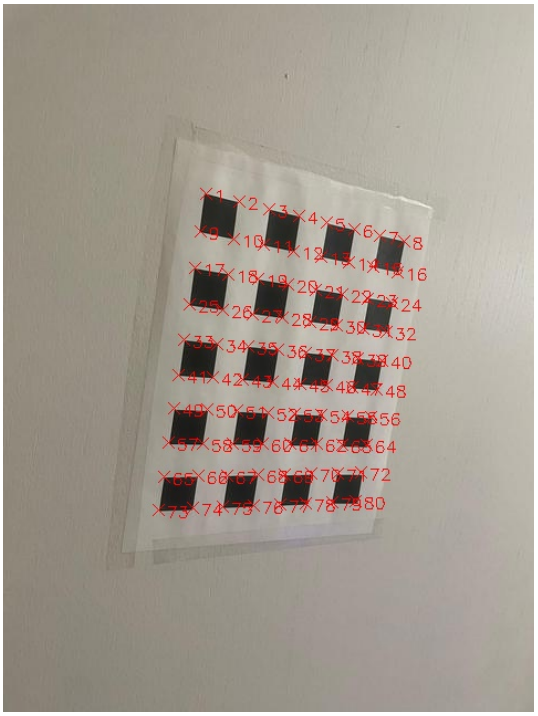
These are the camera matrix, rotation matrix and translation matrix at the beginning and after the refinements:

$$K_{init} = \begin{bmatrix} 489.356 & -0.433 & 238.076 \\ 0 & 493.202 & 316.913 \\ 0 & 0 & 1 \end{bmatrix} K_{LM} = \begin{bmatrix} 492.779 & -0.253 & 239.687 \\ 0 & 496.544 & 317.690 \\ 0 & 0 & 1 \end{bmatrix} K_{radial} = \begin{bmatrix} 478.630 & -0.240 & 238.052 \\ 0 & 481.911 & 318.075 \\ 0 & 0 & 1 \end{bmatrix} \tag{36}$$

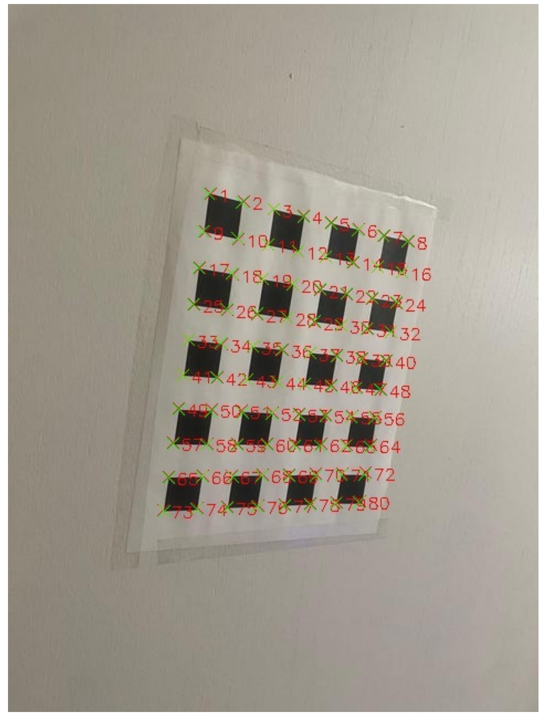
$$R_{init} = \begin{bmatrix} 0.928 & -0.215 & 0.302 \\ 0.292 & 0.926 & -0.237 \\ -0.229 & 0.309 & 0.923 \end{bmatrix} R_{LM} = \begin{bmatrix} 0.921 & -0.207 & 0.329 \\ 0.297 & 0.920 & -0.253 \\ -0.250 & 0.331 & 0.909 \end{bmatrix} R_{radial} = \begin{bmatrix} 0.920 & -0.205 & 0.332 \\ 0.298 & 0.919 & -0.256 \\ -0.253 & 0.335 & 0.907 \end{bmatrix} \tag{37}$$

$$t_{init} = \begin{bmatrix} -47.68 & -29.517 & 168.201 \end{bmatrix} t_{LM} = \begin{bmatrix} -47.803 & -29.392 & 167.734 \end{bmatrix} t_{radial} = \begin{bmatrix} -47.266 & -29.508 & 164.223 \end{bmatrix} \tag{38}$$

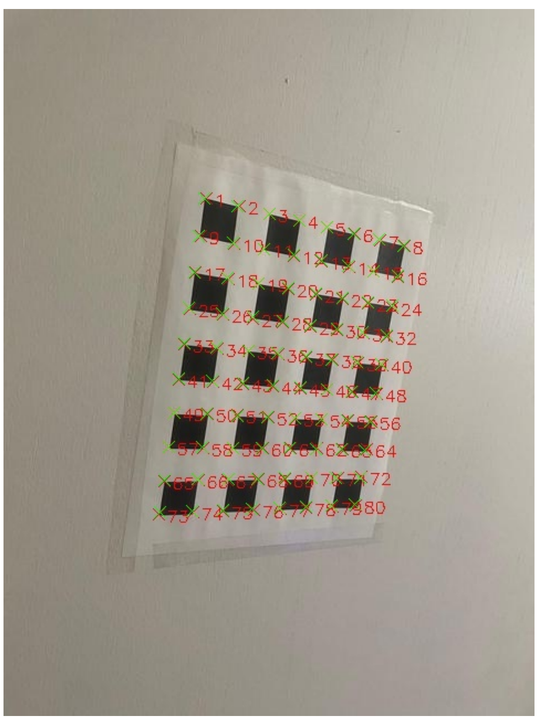
$$k_1 = 6.267e - 7 \qquad k_2 = -6.473e - 12 \tag{39}$$



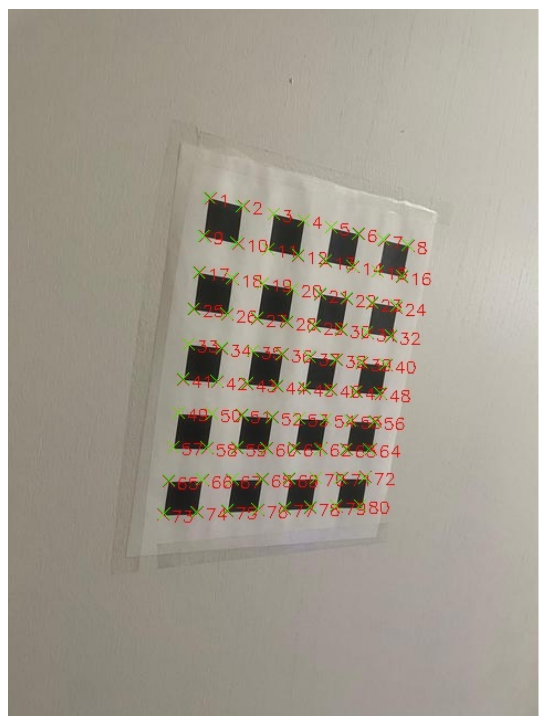
(a) Original corners



(b) Initial projection



(c) Projection after LM



(d) Projection after LM with radial distortion

Figure 9: Comparison of the projection of the world coordinates onto the pattern from image 10 in my dataset

These are the camera matrix, rotation matrix and translation matrix at the beginning and after the refinements (matrices Ks and coefficients k_1 and k_2 are the same as stated for image 4):

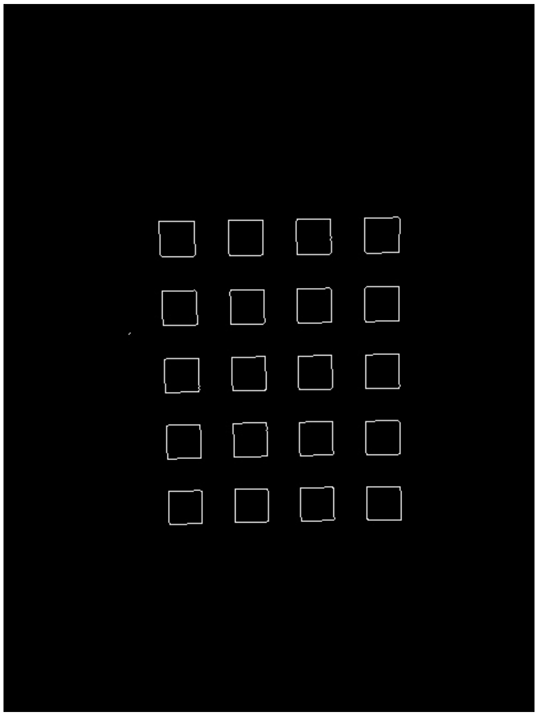
$$R_{init} = \begin{bmatrix} 0.885 & -0.156 & -0.436 \\ 0.088 & 0.981 & -0.170 \\ 0.455 & 0.112 & 0.883 \end{bmatrix} R_{LM} = \begin{bmatrix} 0.886 & -0.158 & -0.433 \\ 0.087 & 0.980 & -0.178 \\ 0.453 & 0.120 & 0.883 \end{bmatrix} R_{radial} = \begin{bmatrix} 0.889 & -0.158 & -0.429 \\ 0.087 & 0.979 & -0.179 \\ 0.449 & 0.121 & 0.885 \end{bmatrix} \tag{40}$$

$$t_{init} = \begin{bmatrix} -17.015 & -43.290 & 147.223 \end{bmatrix} t_{LM} = \begin{bmatrix} -17.543 & -43.748 & 148.756 \end{bmatrix} t_{radial} = \begin{bmatrix} -17.038 & -43.870 & 145.225 \end{bmatrix} \tag{41}$$

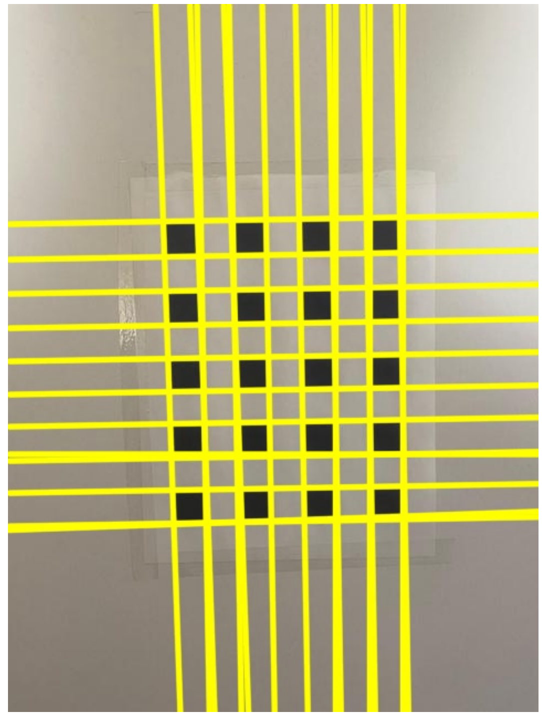
$$k_1 = 6.267e - 7 \qquad k_2 = -6.473e - 12 \tag{42}$$

Finally, we also show the performance with the "Fixed Image" so that we can validate the obtained results with the metrics calculated in Section 2.7

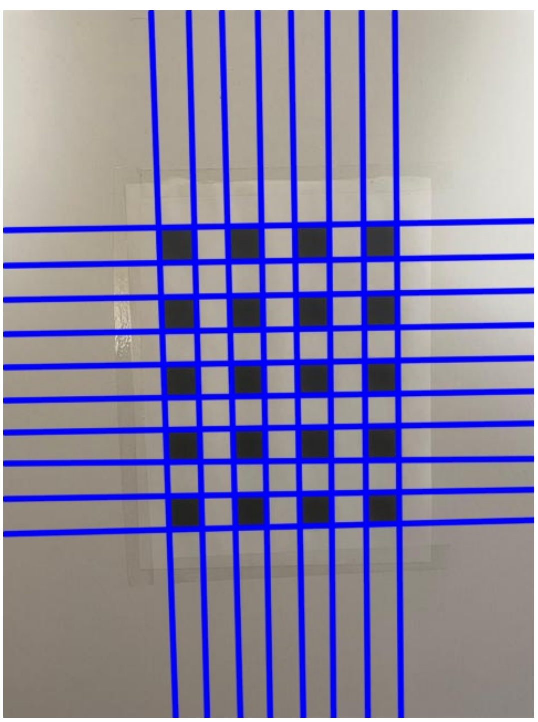
Figure 10 shows the 4 images resulting from the preprocessing of the Fixed Image in the dataset. It contains the edges after using the Canny edge detector, the multiple lines resulting from the Hough Transform, the final selected lines and the final intersection points.



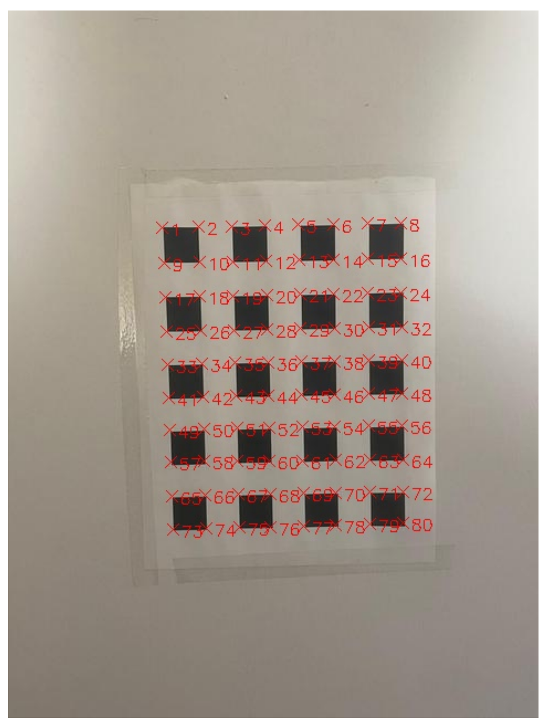
(a) Edges after using the Canny edge detector



(b) Lines resulting from the Hough Transform



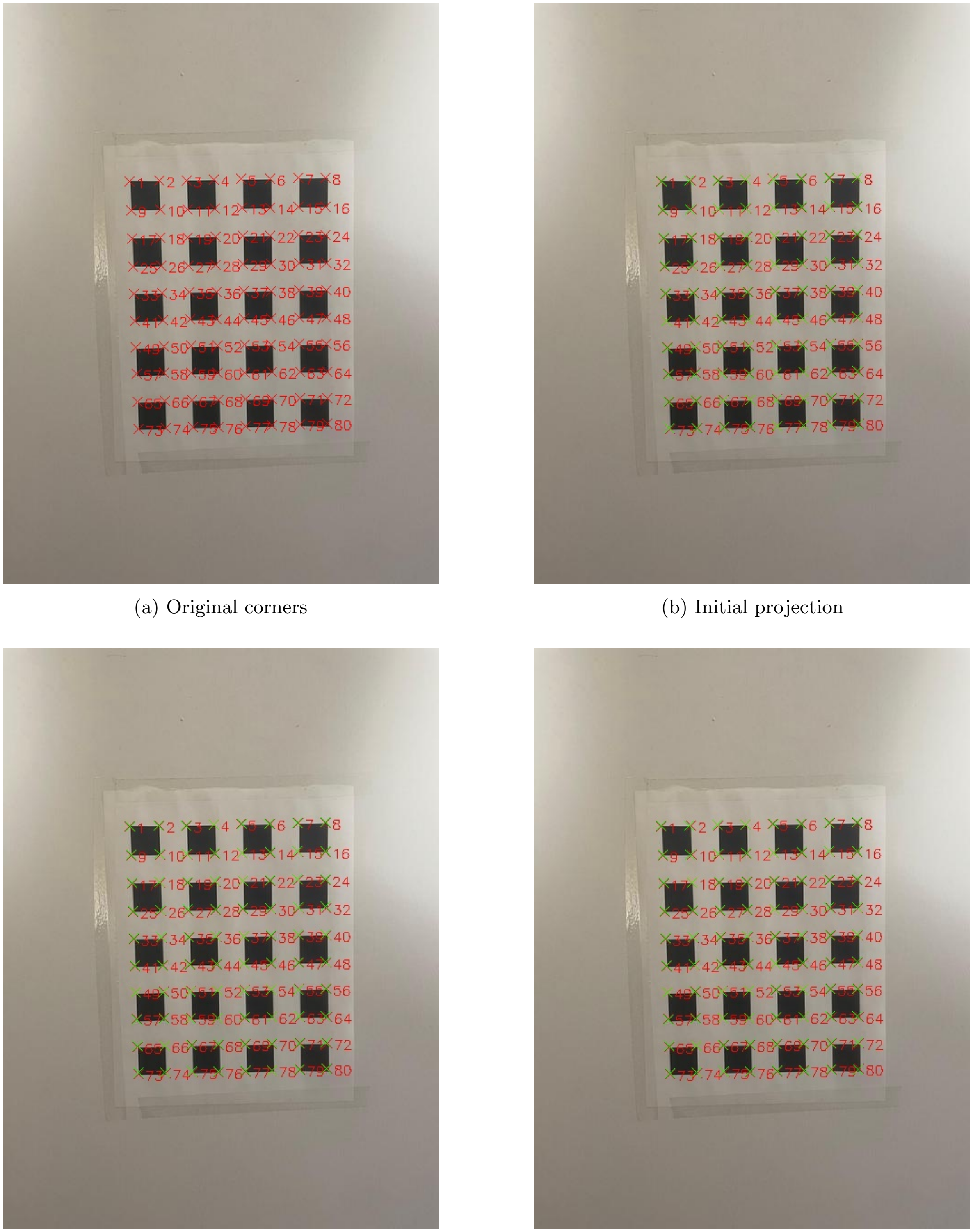
(c) Final selected lines



(d) Final intersection points

Figure 10: Preprocessing of Fixed Image from my dataset.

Figure 11 shows the projection results at the beginning and after the refinements. See that reprojected corners are printed in green color. Red corners correspond to ground truth corners. Reprojected corners are drawn above ground truth corners. Therefore, not seeing the ground truth corner means almost perfect reprojection.



(a) Original corners

(b) Initial projection

(c) Projection after LM

(d) Projection after LM with radial distortion

Figure 11: Comparison of the projection of the world coordinates onto Fixed Image in my dataset

These are the camera matrix, rotation matrix and translation matrix at the beginning and after the refinements (matrices Ks and coefficients k_1 and k_2 are the same as stated for image 4):

$$R_{init} = \begin{bmatrix} 0.999 & 0.020 & 0.003 \\ -0.019 & 0.997 & -0.065 \\ -0.004 & 0.065 & 0.997 \end{bmatrix} R_{LM} = \begin{bmatrix} 0.999 & 0.020 & 0.004 \\ -0.019 & 0.997 & -0.069 \\ -0.005 & 0.069 & 0.997 \end{bmatrix} R_{radial} = \begin{bmatrix} 0.999 & 0.019 & -0.0009 \\ -0.019 & 0.997 & -0.072 \\ -0.0005 & 0.072 & 0.997 \end{bmatrix} \tag{43}$$

$$t_{init} = \begin{bmatrix} -31.899 & -38.917 & 159.520 \end{bmatrix} t_{LM} = \begin{bmatrix} -32.392 & -39.276 & 160.829 \end{bmatrix} t_{radial} = \begin{bmatrix} -31.819 & -39.389 & 157.063 \end{bmatrix} \tag{44}$$

$$k_1 = 6.267e-7 \qquad k_2 = -6.473e-12 \tag{45}$$

Table 2 shows the quantitative evaluation of the projection error

Metric	Image 4	Image 10	Fixed Image
Initial error mean	2.1072	1.2747	0.8405
Initial error variance	0.8061	0.3647	0.2528
Error mean after LM	0.7849	0.7801	0.7452
Error variance after LM	0.1935	0.1366	0.1353
Error mean after LM + radial	0.6531	0.7128	0.6620
Error variance after LM + radial	0.1417	0.1251	0.1281

Table 2: Error mean and variance for images 4 and 10 and Fixed Image of my dataset

Figure 12 shows the camera poses that have been used in order to create my dataset. The black box simulates the position of the calibration pattern.

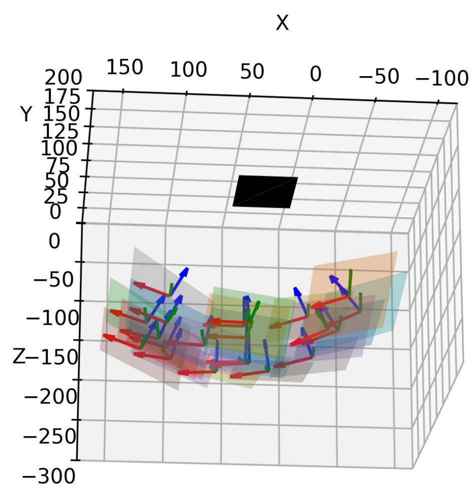


Figure 12: Camera poses to create the images of the calibration pattern in the given dataset

4 Observations

For both, the given dataset and the dataset that we have created we see the same behavior. When reprojecting the corners from the world coordinated onto the selected images we can qualitatively see that the error is reduced after refining the parameters using the LM optimization algorithm and it is even more reduced after incorporating the radial distortion parameters from the "Extra Credit" section of the instructions. This is the desired behavior. Since it is difficult to perceive this improvement visually, we have also shown in tables how the mean and variance error of the reprojection is reduced after refining and even more reduced when refining using the radial distortion parameters. Thus, these quantitative metrics support our qualitative evaluation.

Finally, for the "Fixed Image" in the dataset that we have created, we can also see how the 3rd component of the translation vector t is vary close to the digital distance that we computed in Section 2.7: 159.1. This can serve as a confirmation that our implementation of the tasks asked in this assignment were correctly accomplished.

5 Code

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

# function to find the intersection of 2 lines given 2 points to define each line
def find_intersection(hline, vline):
    x1, y1 = hline[0][0], hline[0][1]
    x2, y2 = hline[1][0], hline[1][1]
    x3, y3 = vline[0][0], vline[0][1]
    x4, y4 = vline[1][0], vline[1][1]

    # Create first line
    A1 = y2 - y1
    B1 = x1 - x2
    C1 = A1 * x1 + B1 * y1

    # Create second line
    A2 = y4 - y3
    B2 = x3 - x4
    C2 = A2 * x3 + B2 * y3

    # Find intersections
    D = A1 * B2 - A2 * B1
    x = (C1 * B2 - C2 * B1) / D
    y = (A1 * C2 - A2 * C1) / D

    return (int(x), int(y))

# Group lines that correspond to the same true line
def group_lines(lines, part):
    clusters = []
    temp_cluster = [lines[0]]

    # Store rho distances between lines and set threshold in those places where distance between lines is higher (should correspond to different groups)
    dist = []
    for k in range(len(lines) - 1):
        dist.append(lines[k + 1][0] - lines[k][0])
    threshold = np.partition(dist, -part)[-part]

    # Group lines depending on the threshold and distance
    for line in lines[1:]:
        rho = line[0]
        prevrho = temp_cluster[-1][0]
        if rho - prevrho < threshold:
            temp_cluster.append(line)
        else:
            clusters.append(temp_cluster)
            temp_cluster = [line]
    if temp_cluster:
        clusters.append(temp_cluster)

    #return found groups
```

```
    return clusters

# Function to given a group of lines find the true line
def get_line(line_form, img, tipo):
    final_lines = []
    for lines in line_form:
        if tipo == "h":
            for i, line in enumerate(lines):
                if line[0] > 0:
                    lines[i] = [line[0], line[1]]
                else:
                    lines[i] = [-line[0], line[1] - np.pi]

        if tipo == "v":
            for i, line in enumerate(lines):
                if line[2] == 1:
                    lines[i] = [line[0], line[1]]
                else:
                    lines[i] = [line[0], line[1] - np.pi]

        rho_val = np.array([line[0] for line in lines])
        theta_val = np.array([line[1] for line in lines])
        # Compute the new rho and theta as the average of the given lines
        new_rho = gmean(rho_val)
        new_theta = np.mean(theta_val)
        new_rho, new_theta = (-new_rho, new_theta + np.pi) if new_theta < 0 else (new_rho, new_theta)

        pt1 = (int(math.cos(new_theta) * new_rho + 5000 * (-math.sin(new_theta))), int(math.sin(new_theta) * new_rho + 5000 * (math.cos(new_theta))))
        pt2 = (int(math.cos(new_theta) * new_rho - 5000 * (-math.sin(new_theta))), int(math.sin(new_theta) * new_rho - 5000 * (math.cos(new_theta))))

        # Get the points for that line and store it
        final_lines.append([pt1, pt2])
        # Draw line
        cv2.line(img, pt1, pt2, (255, 0, 0), 3, cv2.LINE_AA)

    return final_lines

# Main function to find the corners of the calibration pattern for each image of the dataset
def get_corners(path, name):
    img = cv2.imread(path)
    # Apply canny to gray image
    edges = cv2.Canny(cv2.cvtColor(img, cv2.COLOR_RGB2GRAY), 400, 300)
    cv2.imwrite(f'/home/aolivepe/Computer-Vision/HW8/output/{name}_edges.jpg', edges)

    # Use Hough transform to get the lines given the edge image. Classify images in vertical or horizontal depending on the value of rho and theta
    lines = cv2.HoughLines(edges, 1, np.pi / 180, 50, None, 0, 0)
    vlines = []
    hlines = []
    img_lines = np.copy(img)
    if lines is not None:
        for i in range(len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
```

```
            if (rho < 0 and theta > 3 * np.pi / 4) or (rho > 0 and theta < np.pi / 4):
                vlines.append((np.abs(lines[i][0][0]), lines[i][0][1], np.sign(rho)))
            else:
                hlines.append((lines[i][0][0], lines[i][0][1]))

            pt1 = (int(math.cos(theta) * rho + 5000 * (-math.sin(theta))), int(math.sin(theta) * rho + 5000 * (math.cos(theta))))
            pt2 = (int(math.cos(theta) * rho - 5000 * (-math.sin(theta))), int(math.sin(theta) * rho - 5000 * (math.cos(theta))))
            cv2.line(img_lines, pt1, pt2, (0, 255, 255), 4, cv2.LINE_AA)

    cv2.imwrite(f'/home/aolivepe/Computer-Vision/HW8/output/{name}_lines.jpg', img_lines)

    hlines = np.sort(np.array(hlines, dtype=[('', np.float32), ('', np.float32)]), axis=0)
    vlines = np.sort(np.array(vlines, dtype=[('', np.float32), ('', np.float32), ('', int)]), axis=0)

    # Create clusters of the found lines
    real_hlines = group_lines(hlines, part = 9)
    real_vlines = group_lines(vlines, part = 7)

    # Get the true line for each cluster of lines
    img_final_lines = np.copy(img)
    assert len(real_hlines) == 10
    assert len(real_vlines) == 8
    hoz_lines = get_line(real_hlines, img_final_lines, "h")
    ver_lines = get_line(real_vlines, img_final_lines, "v")
    cv2.imwrite(f'/home/aolivepe/Computer-Vision/HW8/output/{name}_all_lines.jpg', img_final_lines)

    # Find the intersection of lines and plot it in the original image
    intersect = []
    img_intersec = np.copy(img)
    for hoz_line in hoz_lines:
        for ver_line in ver_lines:
            pt = find_intersection(hoz_line, ver_line)
            intersect.append(pt)
            x, y = pt
            color = (0, 0, 255)
            thickness = 1
            cv2.line(img_intersec, (x - 5, y - 5), (x + 5, y + 5), color, thickness)
            cv2.line(img_intersec, (x - 5, y + 5), (x + 5, y - 5), color, thickness)
            number = str(len(intersect))
            font = cv2.FONT_HERSHEY_SIMPLEX
            font_scale = 0.5
            text_thickness = 1
            text_position = (x + 7, y + 7)
            cv2.putText(img_intersec, number, text_position, font, font_scale, color, text_thickness)

    cv2.imwrite(f'/home/aolivepe/Computer-Vision/HW8/output/{name}_final_intersec.jpg', img_intersec)
    return intersect

# Find homography from domain and range points
def get_homography(d_pts, r_pts):
```

```python
    mat_A = []
    for i in range(len(r_pts)):
        mat_A.append([0, 0, 0, -d_pts[i][0], -d_pts[i][1], -1, r_pts[i][1] *
            d_pts[i][0], r_pts[i][1] * d_pts[i][1], r_pts[i][1]])
        mat_A.append([d_pts[i][0], d_pts[i][1], 1, 0, 0, 0, -r_pts[i][0] *
            d_pts[i][0], -r_pts[i][0] * d_pts[i][1], -r_pts[i][0]])
    mat_A = np.array(mat_A)
    # Homography given by the last column vector of the matrix V after doing
       SVD decomposition
    _, _, v = np.linalg.svd(mat_A.T @ mat_A)
    return np.reshape(v[-1], (3, 3))

# Function to get the homographies and intersection points of all the images
    in the given dataset
def get_homographies(data_path, world_coord):
    jpg_files = [f for f in os.listdir(data_path) if
       f.lower().endswith('.jpg')]
    print("Num images in dataset: ", len(jpg_files))
    jpg_files.sort()

    homographies = []
    intersecs_total = []
    for i, file in enumerate(jpg_files, start=1):
        path = os.path.join(data_path, file)
        intersec_points = get_corners(path, name=f'Pic_{i}')
        intersecs_total.append(intersec_points)
        homographies.append(get_homography(world_coord, intersec_points))
    return homographies, intersecs_total

# Function to define the matrices needed to estimate w
def calculate_w_matrix_coefficients(h):
    h1, h2 = h[:, 0], h[:, 1]

    # Matrices written following the equations explained in the report
    eq1_coeffs = [
        h1[0]**2 - h2[0]**2,
        2 * (h1[0] * h1[1] - h2[0] * h2[1]),
        2 * (h1[0] * h1[2] - h2[0] * h2[2]),
        h1[1]**2 - h2[1]**2,
        2 * (h1[1] * h1[2] - h2[1] * h2[2]),
        h1[2]**2 - h2[2]**2
    ]

    eq2_coeffs = [
        h1[0] * h2[0],
        h1[0] * h2[1] + h1[1] * h2[0],
        h1[0] * h2[2] + h1[2] * h2[0],
        h1[1] * h2[1],
        h1[1] * h2[2] + h1[2] * h2[1],
        h1[2] * h2[2]
    ]

    return np.array([eq1_coeffs, eq2_coeffs])

# Estimate w given all the homographies
def estimate_w(homographies):
    lhs = []
    for h in homographies:
        lhs.append(calculate_w_matrix_coefficients(h)[0])
        lhs.append(calculate_w_matrix_coefficients(h)[1])

    lhs = np.asarray(lhs, dtype=np.float64)
```

```

    # Use last vector of V in SVD to find w
    _, _, v = np.linalg.svd(lhs)
    w_solution = v[-1, :]
    return w_solution

# Estimate K given w. First calculate all the coefficients following the
    equations from the report and then form matrix K
def estimate_k(w):
    w11, w12, w13, w22, w23, w33 = w

    y0 = (w12 * w13 - w11 * w23) / (w11 * w22 - w12 ** 2)
    lam = w33 - (w13 ** 2 + y0 * (w12 * w13 - w11 * w23)) / w11
    alphax = np.sqrt(lam / w11)
    alphay = np.sqrt(lam * w11 / (w11 * w22 - w12 ** 2))
    s = -(w12 * alphax ** 2 * alphay) / lam
    x0 = s * y0 / alphay - (w13 * alphax ** 2) / lam

    K = np.array([[alphax, s, x0],
                  [0, alphay, y0],
                  [0, 0, 1]])
    return K

# Estimate the extrinsic parameters for each image given the homography and
    K. Compute parameters following equations from the report
def estimate_extrinsic_param(homographies, K):
    rot = []
    trans = []
    K_inv = np.linalg.inv(K)

    for H in homographies:
        h1, h2, h3 = H[:, 0], H[:, 1], H[:, 2]
        r1 = K_inv @ h1 / np.linalg.norm(K_inv @ h1)
        r2 = K_inv @ h2 / np.linalg.norm(K_inv @ h1)
        r3 = np.cross(r1, r2)
        t = K_inv @ h3 / np.linalg.norm(K_inv @ h1)
        R = np.stack([r1,r2,r3], axis=1)

        # Enforce orthogonality
        u, _, v = np.linalg.svd(R)
        R = u @ v

        rot.append(R)
        trans.append(t)

    return rot, trans

# Create vector with all the parameters for each image in the dataset. This
    is needed for the optimization algorithm
def param_cam(K, rots, trans):
    p = [K[0, 0], K[0, 1], K[0, 2], K[1, 1], K[1, 2]]
    # Use Rodrigues Representation for R
    for R, t in zip(rots, trans):
        p.extend(np.hstack(((np.arccos((np.trace(R) - 1) / 2) / (2 *
            np.sin(np.arccos((np.trace(R) - 1) / 2)))) * np.array([R[2, 1] -
            R[1, 2], R[0, 2] - R[2, 0], R[1, 0] - R[0, 1]]), t)))
    return p

# Given the flettened vector p, reconstruct the parameters K, R and t
def reconstruct_p(p):
    K = np.array([[p[0], p[1], p[2]],
                  [0, p[3], p[4]],
```

```
                       [0, 0, 1]])

    rotation_matrices, translation_vectors = [], []
    step_size = 6
    for idx in range(5, len(p), step_size):
        rot_vec = p[idx:idx+3]
        trans_vec = p[idx+3:idx+6]

        # Undo Rodrigues Representation for R
        rot_angle = np.linalg.norm(rot_vec)
        skew_matrix = np.array([
            [0, -rot_vec[2], rot_vec[1]],
            [rot_vec[2], 0, -rot_vec[0]],
            [-rot_vec[1], rot_vec[0], 0]
        ])
        identity_matrix = np.identity(3)
        R_matrix = identity_matrix + (np.sin(rot_angle) / rot_angle) * skew_matrix + ((1 - np.cos(rot_angle)) / (rot_angle ** 2)) * (skew_matrix @ skew_matrix)

        rotation_matrices.append(R_matrix)
        translation_vectors.append(trans_vec)

    return K, rotation_matrices, translation_vectors

# Get the error mean and variance of the projected corners for a specific image
def error(diff, idx):
    diff = diff.reshape((-1 , 2))
    start = idx * 80
    end = start + 80
    diff_norm = np.linalg.norm(diff[start:end], axis =1)
    return np.average(diff_norm), np.var(diff_norm)

# Cost function for the optimization algorithm. Also used to get quantitative evaluation of the refinements done
def cost(p, full_corners, world_coord, radial=False, img_idx=-1, name=None):
    K, Rs, ts = reconstruct_p(p[:-2] if radial else p)

    all_projected_points = []
    for k, (R, t) in enumerate(zip(Rs, ts)):
        # Project points
        H = np.matmul(K, np.column_stack((R[:, 0], R[:, 1], t)))
        pts_h = np.hstack([world_coord, np.ones((len(world_coord), 1))])  # Convert to homogeneous coordinates
        transf_pts = H @ pts_h.T  # Apply homography
        transf_pts = transf_pts.T
        projected_points = transf_pts[:, :2] / transf_pts[:, 2, np.newaxis]  # Normalize by the last row

        # Use radial distortion parameters if desired
        if radial:
            x, y = projected_points[:, 0], projected_points[:, 1]
            k1, k2, x0, y0 = p[-2], p[-1], p[2], p[4]
            r_squared = (x - x0)**2 + (y - y0)**2
            x_corrected = x + (x - x0) * (k1 * r_squared + k2 * r_squared**2)
            y_corrected = y + (y - y0) * (k1 * r_squared + k2 * r_squared**2)
            projected_points = np.vstack((x_corrected, y_corrected)).T

        all_projected_points.append(projected_points)
        # Draw reprojected corners
        if k == img_idx:
```

```python
            reproject(all_projected_points, img_idx, name)

    # Calculated the distance between projected corners and ground truth corners
    all_projected_points = np.concatenate(all_projected_points, axis=0)
    full_corners = np.concatenate(full_corners, axis=0)
    diff = full_corners - all_projected_points
    return diff.flatten()

# Function to draw the projected images onto an image in whic ground truth corners are already drawn
def reproject(all_projected_points, img_idx, name):
    path = f"/home/aolivepe/Computer-Vision/HW8/output/Pic_{img_idx + 1}_final_intersec.jpg"
    img = cv2.imread(path)
    for point in all_projected_points[img_idx]:
        x, y = int(point[0]), int(point[1])
        color = (0, 255, 0)
        thickness = 1
        cv2.line(img, (x - 5, y - 5), (x + 5, y + 5), color, thickness)
        cv2.line(img, (x - 5, y + 5), (x + 5, y - 5), color, thickness)
    cv2.imwrite(f'/home/aolivepe/Computer-Vision/HW8/output/Pic_{img_idx + 1}{name}reproject.jpg', img)

# Function to plot the camera poses for each image of the dataset
def camera_poses(Rs, ts):
    # Calculate the camera centers based on rotations and translations
    camera_centers = [-R.T @ t for R, t in zip(Rs, ts)]

    # Define the axes for each camera
    axis_x = [R.T @ np.array([1, 0, 0]) + center for R, center in zip(Rs, camera_centers)]
    axis_y = [R.T @ np.array([0, 1, 0]) + center for R, center in zip(Rs, camera_centers)]
    axis_z = [R.T @ np.array([0, 0, 1]) + center for R, center in zip(Rs, camera_centers)]

    # Set up the 3D plot
    vector_length = 35
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Plot each camera's x, y, z axes with color-coded quivers
    for center, x, y, z in zip(camera_centers, axis_x, axis_y, axis_z):
        ax.quiver(center[0], center[1], center[2], x[0]-center[0], x[1]-center[1], x[2]-center[2], color="r", length=vector_length, normalize=True)
        ax.quiver(center[0], center[1], center[2], y[0]-center[0], y[1]-center[1], y[2]-center[2], color="g", length=vector_length, normalize=True)
        ax.quiver(center[0], center[1], center[2], z[0]-center[0], z[1]-center[1], z[2]-center[2], color="b", length=vector_length, normalize=True)

    # Plot planes based on camera orientation
    for center, z_axis in zip(camera_centers, axis_z):
        x_vals, y_vals = np.meshgrid(range(int(center[0] - vector_length), int(center[0] + vector_length)), range(int(center[1] - vector_length), int(center[1] + vector_length)))
        z_vals = -((x_vals - center[0]) * z_axis[0] + (y_vals - center[1]) * z_axis[1]) / z_axis[2] + center[2]
        ax.plot_surface(x_vals, y_vals, z_vals, alpha=0.3)
```

```python

    # Plot calibration pattern as a black square
    center_x, center_y = 20, 60
    size = 50
    x_square = [center_x - size / 2, center_x + size / 2, center_x + size /
        2, center_x - size / 2]
    y_square = [center_y - size / 2, center_y - size / 2, center_y + size /
        2, center_y + size / 2]
    z_square = [0, 0, 0, 0]
    ax.plot_trisurf(x_square, y_square, z_square, color='black')

    ax.set_ylim([-1, 200])
    ax.set_zlim([-300, 0])
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")

    # Set orientation of the plot
    elev = -20
    azim = 85
    ax.view_init(elev=elev, azim=azim)

    plt.savefig("3d_vectors_plot.jpg", format="jpg", dpi=300)

##############################################
#                    MAIN                    #
##############################################

index_img_1 = 0
index_img_2 = 10

dataset_path = "/home/aolivepe/Computer-Vision/HW8/Dataset2"
# dataset_path = "/home/aolivepe/Computer-Vision/HW8/HW8-Files/Dataset1"

# Get world coordinates
x_coords = 10 * np.arange(8)
y_coords = 10 * np.arange(10)
y_grid, x_grid = np.meshgrid(y_coords, x_coords, indexing='ij')
world_coord = np.stack([x_grid.ravel(), y_grid.ravel()], axis=-1)

# Get homogrphies and ground truth corners
Hs, full_corners = get_homographies(dataset_path, world_coord)

# Get parameters
w = estimate_w(Hs)
K = estimate_k(w)
print("K: ", K)
Rs, ts = estimate_extrinsic_param(Hs, K)
print("Rs[index_img_1]: ", Rs[index_img_1])
print("ts[index_img_1]: ", ts[index_img_1])
print("Rs[index_img_2]: ", Rs[index_img_2])
print("ts[index_img_2]: ", ts[index_img_2])

# Prepare parameters for refinement
p = param_cam(K, Rs, ts)
# Quantitative metrics for evaluation
mean_init_1, var_init_1 = error(cost(np.array(p), full_corners, world_coord,
    img_idx=index_img_1, name="_original"), idx=index_img_1)
mean_init_2, var_init_2 = error(cost(np.array(p), full_corners, world_coord,
    img_idx=index_img_2, name="_original"), idx=index_img_2)

# Refine and project and quantitative metrics of projection after refinement
```

```python
p_lm = least_squares(cost, np.array(p), method='lm', args=[full_corners,
    world_coord])
mean_refined_1, var_refined_1 = error(cost(p_lm.x, full_corners, world_coord,
    img_idx=index_img_1, name="_lm"), idx=index_img_1)
mean_refined_2, var_refined_2 = error(cost(p_lm.x, full_corners, world_coord,
    img_idx=index_img_2, name="_lm"), idx=index_img_2)

K_refined, Rs_refined, ts_refined = reconstruct_p(p_lm.x)
print("K_refined: ", K_refined)
print("Rs_refined[index_img_1]: ", Rs_refined[index_img_1])
print("ts_refined[index_img_1]: ", ts_refined[index_img_1])
print("Rs_refined[index_img_2]: ", Rs_refined[index_img_2])
print("ts_refined[index_img_2]: ", ts_refined[index_img_2])

# Incorporate radial distortion parameters, refine and quantitative metrics
    of projection after refinement
p_rad = param_cam(K, Rs, ts)
p_rad.extend([0, 0])
p_lm_rad = least_squares(cost, np.array(p_rad), method='lm',
    args=[full_corners, world_coord, True])
mean_radial_1, var_radial_1 = error(cost(p_lm_rad.x, full_corners,
    world_coord, radial=True, img_idx=index_img_1, name="lm_w_rad"),
    idx=index_img_1)
mean_radial_2, var_radial_2 = error(cost(p_lm_rad.x, full_corners,
    world_coord, radial=True, img_idx=index_img_2, name="lm_w_rad"),
    idx=index_img_2)

K_refined_rad, Rs_refined_rad, ts_refined_rad = reconstruct_p(p_lm_rad.x[:-2])
print("K_refined_rad: ", K_refined_rad)
print("Rs_refined_rad[index_img_1]: ", Rs_refined_rad[index_img_1])
print("ts_refined_rad[index_img_1]: ", ts_refined_rad[index_img_1])
print("Rs_refined_rad[index_img_2]: ", Rs_refined_rad[index_img_2])
print("ts_refined_rad[index_img_2]: ", ts_refined_rad[index_img_2])

print(f"--------------Image {index_img_1}----------------")
print(f"|Init mean: {mean_init_1} Init var: {var_init_1}")
print(f"|Refined mean: {mean_refined_1} Refined var: {var_refined_1}")
print(f"|Radial mean: {mean_radial_1} Radial var: {var_radial_1}")
print("[k1 k2] ", p_lm_rad.x[-2:])
print("-------------------------------------")

print(f"--------------Image {index_img_2}----------------")
print(f"|Init mean: {mean_init_2} Init var: {var_init_2}")
print(f"|Refined mean: {mean_refined_2} Refined var: {var_refined_2}")
print(f"|Radial mean: {mean_radial_2} Radial var: {var_radial_2}")
print("[k1 k2] ", p_lm_rad.x[-2:])
print("-------------------------------------")

# Get the camera poses plot
camera_poses(Rs, ts)
```