

ECE66100 Homework #8

Ziyue “Alan” Xiang

November 14, 2022

Contents

I. Theory Questions	3
II. Zhang’s Algorithm	3
III. Experiments and Results	6
IV. Figures	11
V. Tables	26
A. Source Code	27
A.1. Source Listing of code/common.py	27
A.2. Source Listing of code/helper.py	27
A.3. Source Listing of code/func_impl.py	28
A.4. Source Listing of code/environment.yaml	31
A.5. Source Listing of code/08.ipynb	33

List of Figures

1. Examples of radial distortion.	11
2. Vertices and their numbers in the calibration pattern. The image is Pic_13 from the provided dataset.	12
3. The output of corner detection step 1.	13
4. The output of corner detection step 2.	14
5. The output of corner detection step 3.	15
6. The output of corner detection step 4.	16
7. The output of corner detection step 5.	17

8.	The output of corner detection step 6. The lines have been separated into two groups marked with red and blue, respectively; within each group, the spatial order (index) of each line is encoded using color gradient from white to red (blue).	18
9.	The reprojection results for the <code>Pic_1</code> image. Here, the original calibration pattern is projected to the input image.	19
10.	The reprojection results for the <code>Pic_15</code> image. Here, the original calibration pattern is projected to the input image.	20
11.	The reprojection of <code>Pic_1</code> (source image) to <code>Pic_13</code> (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.	21
12.	The reprojection of <code>Pic_15</code> (source image) to <code>Pic_13</code> (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.	22
13.	The output of corner detection on our custom dataset. The lines have been separated into two groups marked with red and blue, respectively; within each group, the spatial order (index) of each line is encoded using color gradient from white to red (blue).	23
14.	The reprojection of <code>IMG_8327.JPG</code> (source image) to <code>IMG_8318.JPG</code> (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.	24
15.	The reprojection of <code>IMG_8328.JPG</code> (source image) to <code>IMG_8318.JPG</code> (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.	25

List of Tables

1.	The reprojection error ϵ of various camera calibration methods. Here, we reproject the original calibration pattern to the input image to compute the reprojection error.	26
2.	The reprojection error ϵ of various camera calibration methods on our custom dataset. Here, we reproject the original calibration pattern to the input image to compute the reprojection error.	26

I. Theory Questions

No, we will not be able to see the image of $\mathbf{w}$ in the camera image. Any point $\mathbf{x}$ on the conic must satisfy $\mathbf{x}^T \mathbf{w} \mathbf{x} = 0$. However, because $\mathbf{w} = \mathbf{K}^{-T} \mathbf{K}^{-1}$, where $\mathbf{K}$ stores intrinsic parameters, $\mathbf{w}$ is positive definite. Therefore, $\mathbf{x}^T \mathbf{w} \mathbf{x} = 0$ only have imaginary solutions. That is, we cannot see $\mathbf{w}$ in the camera image.

The plane where the calibration pattern is located samples $\mathbf{w}$ at two points. With the camera image of the calibration pattern, we can estimate the homography $\mathbf{H} \in \mathbb{R}^{3 \times 3}$ between the image of the calibration pattern and the original calibration pattern. Suppose we write $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$, the following equations hold:

$$\mathbf{h}_1^T \mathbf{w} \mathbf{h}_1 = \mathbf{h}_1^T \mathbf{w} \mathbf{h}_1 \quad (1)$$

$$\mathbf{h}_1^T \mathbf{w} \mathbf{h}_2 = 0. \quad (2)$$

Given the camera image of the calibration pattern from different angles, we can acquire a set of $\mathbf{H}$. Using the equations above, we will be able to estimate $\mathbf{w}$, which can lead us to the intrinsic parameters $\mathbf{K}$.

II. Zhang's Algorithm

Zhang's algorithm refers to the camera calibration algorithm proposed in Zhang's paper (Z. Zhang, "A flexible new technique for camera calibration," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, Nov. 2000).

- Estimation of $\mathbf{w}$

Given N ($N \geq 3$) images of the calibration pattern, we can acquire a set of homographies $\mathbf{H}^{(1)}, \dots, \mathbf{H}^{(N)}$. Since $\mathbf{w}$ is symmetric, we can write it as

$$\mathbf{w} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{12} & w_{22} & w_{23} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}, \quad (3)$$

which contains 6 variables.

For a particular $\mathbf{H}$ such that

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \quad (4)$$

we can rewrite Eqs. (1) and (2) as

$$\begin{bmatrix} h_{11}^2 - h_{12}^2 \\ 2h_{11}h_{21} - 2h_{12}h_{22} \\ 2h_{11}h_{31} - 2h_{12}h_{32} \\ h_{21}^2 - h_{22}^2 \\ 2h_{21}h_{31} - 2h_{22}h_{32} \\ h_{31}^2 - h_{32}^2 \end{bmatrix}^T \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{22} \\ w_{23} \\ w_{33} \end{bmatrix} = 0 \quad (5)$$

$$\begin{bmatrix} h_{11}h_{12} \\ h_{11}h_{22} + h_{12}h_{21} \\ h_{11}h_{32} + h_{12}h_{31} \\ h_{21}h_{22} \\ h_{21}h_{32} + h_{22}h_{31} \\ h_{31}h_{32} \end{bmatrix}^T \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \\ w_{22} \\ w_{23} \\ w_{33} \end{bmatrix} = 0. \quad (6)$$

This allows us to build a set of homogeneous equations to solve for $\mathbf{w}$. The solution can be acquired using SVD.

- The estimation of $\mathbf{K}$

The intrinsic parameters matrix $\mathbf{K}$ is given by

$$\mathbf{K} = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (7)$$

According to Zhang's paper, we can estimate the entries in $\mathbf{K}$ as follows:

$$y_0 = \frac{-w_{11}w_{23} + w_{12}w_{13}}{w_{11}w_{22} - w_{12}^2} \quad (8)$$

$$\lambda = w_{33} - \frac{w_{13}^2 + y_0(-w_{11}w_{23} + w_{12}w_{13})}{w_{11}} \quad (9)$$

$$\alpha_x = \sqrt{\frac{\lambda}{w_{11}}} \quad (10)$$

$$\alpha_y = \sqrt{\frac{\lambda w_{11}}{w_{11}w_{22} - w_{12}^2}} \quad (11)$$

$$s = -\frac{\alpha_x^2 \alpha_y w_{12}}{\lambda} \quad (12)$$

$$x_0 = -\frac{\alpha_x^2 w_{13}}{\lambda} + \frac{sy_0}{\alpha_y} \quad (13)$$

- The linear estimation of $\mathbf{R}$ and $\mathbf{t}$

Given an estimated homography $\mathbf{H} = [\mathbf{h}_1 \ \mathbf{h}_2 \ \mathbf{h}_3]$, according to Zhang's paper, we can estimate $\mathbf{R} = [\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{r}_3]$ and $\mathbf{t}$ as follows:

$$\xi = \frac{1}{\|\mathbf{K}^{-1}\mathbf{h}_1\|} \quad (14)$$

$$\mathbf{r}_1 = \xi \mathbf{K}^{-1} \mathbf{h}_1 \quad (15)$$

$$\mathbf{r}_2 = \xi \mathbf{K}^{-1} \mathbf{h}_2 \quad (16)$$

$$\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2 \quad (17)$$

$$\mathbf{t} = \xi \mathbf{K}^{-1} \mathbf{h}_3. \quad (18)$$

To ensure the orthogonality of $\mathbf{R}$, we apply SVD to it so that $\mathbf{R} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, and then we set $\mathbf{R} = \mathbf{U}\mathbf{V}^T$.

- The LM (nonlinear) refinement of $\mathbf{K}$, $\mathbf{R}$, and $\mathbf{t}$

The estimation of $\mathbf{K}$, $\mathbf{R}$, and $\mathbf{t}$ can be further refined using nonlinear least squares algorithms (e.g., the Levenberg-Marquardt algorithm). Suppose the world coordinates of the calibration pattern are $\mathbf{x}_1, \dots, \mathbf{x}_M$, and there corresponding pixel coordinates in the camera image are $\mathbf{z}_1, \dots, \mathbf{z}_M$. Notice that $\mathbf{x}_i \in \mathbb{R}^4$ is homogeneous coordinates, whereas $\mathbf{z}_i \in \mathbb{R}^2$ is physical coordinates. The function $g(\cdot)$ converts a point from homogeneous coordinates to physical coordinates. We define the geometric loss function as

$$\epsilon = \sum_{i=1}^M \|\mathbf{z}_i - g(\mathbf{K}[\mathbf{R} \mid \mathbf{t}]\mathbf{x}_i)\|^2. \quad (19)$$

This is also known as the reprojection error. We use the LM algorithm with this loss function to solve for better estimations of $\mathbf{K}$, $\mathbf{R}$, and $\mathbf{t}$.

It is worth noting that $\mathbf{R}$ only has three degrees of freedom. Therefore, $\mathbf{R}$ should be associated with 3 variables in the LM algorithm. Any 3D rotation matrix can be represented using 3 Euler angles ϕ , θ , and ψ , which denotes rotation with respect to the z , y , and x axis, respectively. Given ϕ , θ , and ψ , the matrix of the 3D rotation is given by

$$\begin{bmatrix} \cos \phi \cos \theta & \cos \phi \sin \theta \sin \psi - \sin \phi \cos \psi & \cos \phi \sin \theta \cos \psi + \sin \phi \sin \psi \\ \sin \phi \cos \theta & \sin \phi \sin \theta \sin \psi + \cos \phi \cos \psi & \sin \phi \sin \theta \cos \psi - \cos \phi \sin \psi \\ -\sin \theta & \cos \theta \sin \psi & \cos \theta \cos \psi \end{bmatrix}. \quad (20)$$

Contrarily, given any 3D rotation matrix $\mathbf{R}$ such that

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad (21)$$

a solution for ϕ , θ , and ψ is given by (Slabaugh, G.G., 1999. *Computing Euler angles from a rotation matrix*):

$$\theta = -\arcsin r_{31} \quad (22)$$

$$\psi = \arctan2\left(\frac{r_{32}}{\cos \theta}, \frac{r_{33}}{\cos \theta}\right) \quad (23)$$

$$\phi = \arctan2\left(\frac{r_{21}}{\cos \theta}, \frac{r_{11}}{\cos \theta}\right). \quad (24)$$

We can estimate ϕ , θ , and ψ from $\mathbf{R}$ and use them as initial values in the LM algorithm.

- Radial distortion correction

In real-world cameras, straight lines imaged far away from the center of the image plane may occur curved. This is known as radial distortion. Examples of radial distortion is shown in Fig. 1. In Zhang's paper, the author proposed that such distortion can be corrected by

$$r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2 \quad (25)$$

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

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

where $(\hat{x}, \hat{y})$ is the image of an object before radial distortion correction, and $(\hat{x}_{\text{rad}}, \hat{y}_{\text{rad}})$ is the image of an object after radial distortion correction. We can use the LM algorithm to estimate the values of k_1 and k_2 .

III. Experiments and Results

- Corner detection

We observed that directly applying edge detection to the images in the dataset will lead to noisy edge detection results, especially around the border of the calibration pattern paper. Therefore, we employ the following steps to reduce noise in the input image and determine the vertices in the calibration pattern:

1. In step 1, we apply Otsu's method to threshold the image; the true pixels in the output binary image indicates the location of black pixels in the calibration pattern (Fig. 3); the output of this step still contains some noise (some pixels from the outside of the calibration pattern are also labeled as true)
2. In step 2, we apply connected component labeling to the output of step 1 and eliminate small connected components (Fig. 4); this effectively removes the noise in step 1
3. In step 3, we apply dilation to the output of step 2, and then use the output to mask the input image (Fig. 5); that is, only pixels inside and near the black calibration patterns are retained, while other pixels are filled with white

4. In step 4, we apply Canny edge detection to the masked input image, and then enhance the edge detection result with dilation (Fig. 6)
5. In step 5, we apply Hough transform to detect lines in the output of step 4 (Fig. 7); we use the `skimage.transform.hough_line_peaks` function to identify the most prominent lines in the Hough transform result and remove noisy lines; despite the use the aforementioned function, the output of this step is noisy as lines can be missing in the detection result; unwanted lines can also occur
6. In step 6, we need to 1) remove noise in the line detection result; 2) classify lines into two groups; 3) number the lines in each group by there spatial order in the calibration pattern.

The noisy line removal and line classification can be done by applying clustering methods to the angles. Note that we cannot use k -means or Gaussian mixture model to cluster the angles. For example, if we are detecting lines with angle in the range $[-90^\circ, 90^\circ)$, then the two lines with angle -90° and 89.9° will appear to be very similar. However, this characteristic cannot be captured by k -means or Gaussian mixture model. Given two angles α and β in degrees, we first define a similarity measure between α and β as

$$\text{sim}(\alpha, \beta) = 1 - \frac{\min(|\alpha - \beta|, 180 - |\alpha - \beta|)}{180}, \quad (28)$$

and then use this similarity measure with spectral clustering. For number of clusters $\gamma = 2, 3, \dots, 6$, we apply spectral clustering to the angles of the detected lines until we can find one cluster with 10 lines and another cluster with 8 lines. If we cannot find such two clusters, then this step fails.

If clustering succeeds, we have a cluster with 10 lines (the horizontal lines) and another cluster with 8 lines (the vertical lines). We can sort the order of these lines within their respective group by sorting their intersection with the y and x axis, respectively. The final output of this step is shown in Fig. 8. We use gradient colors to represent the order of lines in their respective groups. It can be seen that this step failed for `Pic_8`, and the grouping result is incorrect for `Pic_10`. The results for other images in the dataset are correct.

Once the lines are classified into two groups and as they are sorted by their spatial order, it is straightforward to create a grid of pixel locations for homography estimation.

- Camera calibration

- The vertices in the calibration pattern are numbered according to Fig. 2
- Dataset 1 (the provided dataset)
 - * We only consider the images where corner detection succeeded (a total of 39 images)

- * The calibration pattern are 1in sided squares with 1in stride; we convert the units into centimeters (i.e., 2.54cm sided squares with 2.54cm stride)
 - * We first estimate homographies from the image of the calibration pattern and the original calibration pattern
 - * We use the 39 estimated homographies to estimate $\mathbf{K}$. The estimated value of $\mathbf{K}$ is shown as $\mathbf{K}_{\text{linear}}$ in Fig. 9.
 - * For each image, we then use $\mathbf{K}$ and the estimated homography to estimate $\mathbf{R}$ and $\mathbf{t}$; we also use the estimated $\mathbf{R}$ and $\mathbf{t}$ as initial values for LM-refinement and radial distortion correction; in radial distortion correction, the initial values for k_1 and k_2 are $k_1 = k_2 = 0$
 - * The reprojection error ϵ of all images and all camera calibration methods are shown in Table 1. Note that the reprojection error here is computed by projecting the original calibration pattern back to input images. It can be seen that LM refinement can reduce the reprojection error. Radial distortion correction can further reduce the reprojection error compared to LM refinement.
 - * In Figs. 9 and 10, we reproject the original calibration pattern to images Pic_1 and Pic_15 using different approaches and compute the reprojection loss ϵ . The green crosses in the two figures are locations of the corners extracted in corner detection step. The red crosses in the two figures are the projected locations of the corners in the original calibration pattern.
 - * To further verify the reliability of camera calibration results, we use the camera calibration results to project the corners from one image to another. For this dataset, we select Pic_13 as the "fixed image". We selected two images Pic_1 and Pic_15 and reproject the corners from them to Pic_13. When applying the reprojection with radial distortion correction, we are faced with the problem of estimating $\hat{x}$ and $\hat{y}$ given $\hat{x}_{\text{rad}}$, $\hat{y}_{\text{rad}}$, k_1 , and k_2 . We use numerical optimization technique to solve for $\hat{x}$ and $\hat{y}$. The results are shown in Figs. 11 and 12. Despite radial distortion refinement resulted in reduced reprojection error from the original calibration pattern to one input image (Table 1), it can sometimes result in greater reprojection error in this step compared to LM refinement. Note that even in Table 1, the improvement brought by radial distortion correction is very limited. In this experiment, the improvement may have been overshadowed by the numerical error introduced in the estimation of $\hat{x}$ and $\hat{y}$ as well as other computation steps.
- Dataset 2 (custom dataset)
- * We create our custom dataset using a Canon EOS 60D DSLR camera with an 18-135mm Zoom lens. During image collection, we fixed the focal length of the lense to 18mm and also manually focus the lense at infinity. We took 12 images in total. Our corner detection method succeed in 11 out of the 12 images,

as shown in Fig. 13. The image appears to be blurry because the calibration pattern is out of focus. However, if we increase the distance between the calibration pattern paper and the camera, it will reduce the size of the calibration pattern in the image plane, which also introduces noise.

- * The camera sensor size is $2.23\text{cm} \times 1.49\text{cm}$, the pixel dimensions are 5184×3456 (however, images are downsampled to 1920×1280 to save space); it can be seen that (theoretically)

$$\frac{\alpha_x}{x_0} = \frac{m_x f}{m_x p_x} = \frac{f}{p_x} = \frac{1.8}{2.23/2} \approx 1.614 \quad (29)$$

$$\frac{\alpha_y}{y_0} = \frac{m_y f}{m_y p_y} = \frac{f}{p_y} = \frac{1.8}{1.49/2} \approx 2.416 \quad (30)$$

- * The reprojection error ϵ of all images and all camera calibration methods are shown in Table 2. Note that the reprojection error here is computed by projecting the original calibration pattern back to input images.
- * For this dataset, we select IMG_8318 as the "fixed image". We selected two images IMG_8327 and IMG_8328 and reproject the corners from them to IMG_8318. The results are shown in Figs. 14 and 15. Unlike dataset 1, radial distortion correction results in better reprojection error for this dataset.
- * From Fig. 14, it can be seen that the rotation matrix of the fixed image is almost identity (if we ignore flipping).
- * From Fig. 14, if we use the results from $\mathbf{K}_{\text{linear}}$, we have

$$\frac{\alpha_x}{x_0} \approx \frac{1574.456}{957.127} \approx 1.645 \quad (31)$$

$$\frac{\alpha_y}{y_0} \approx \frac{1554.261}{650.419} \approx 2.390; \quad (32)$$

if we use the results from $\mathbf{K}_{\text{LM}}$, we have

$$\frac{\alpha_x}{x_0} \approx \frac{1577.303}{952.202} \approx 1.656 \quad (33)$$

$$\frac{\alpha_y}{y_0} \approx \frac{1551.250}{649.020} \approx 2.390; \quad (34)$$

if we use the results from $\mathbf{K}_{\text{radial}}$, we have

$$\frac{\alpha_x}{x_0} \approx \frac{1595.561}{898.598} \approx 1.776 \quad (35)$$

$$\frac{\alpha_y}{y_0} \approx \frac{1565.511}{833.581} \approx 1.878. \quad (36)$$

It can be seen that the ratios from $\mathbf{K}_{\text{linear}}$ and $\mathbf{K}_{\text{LM}}$ are very close to the theoretical values; whereas the values from $\mathbf{K}_{\text{radial}}$ is as reliable.

IV. Figures

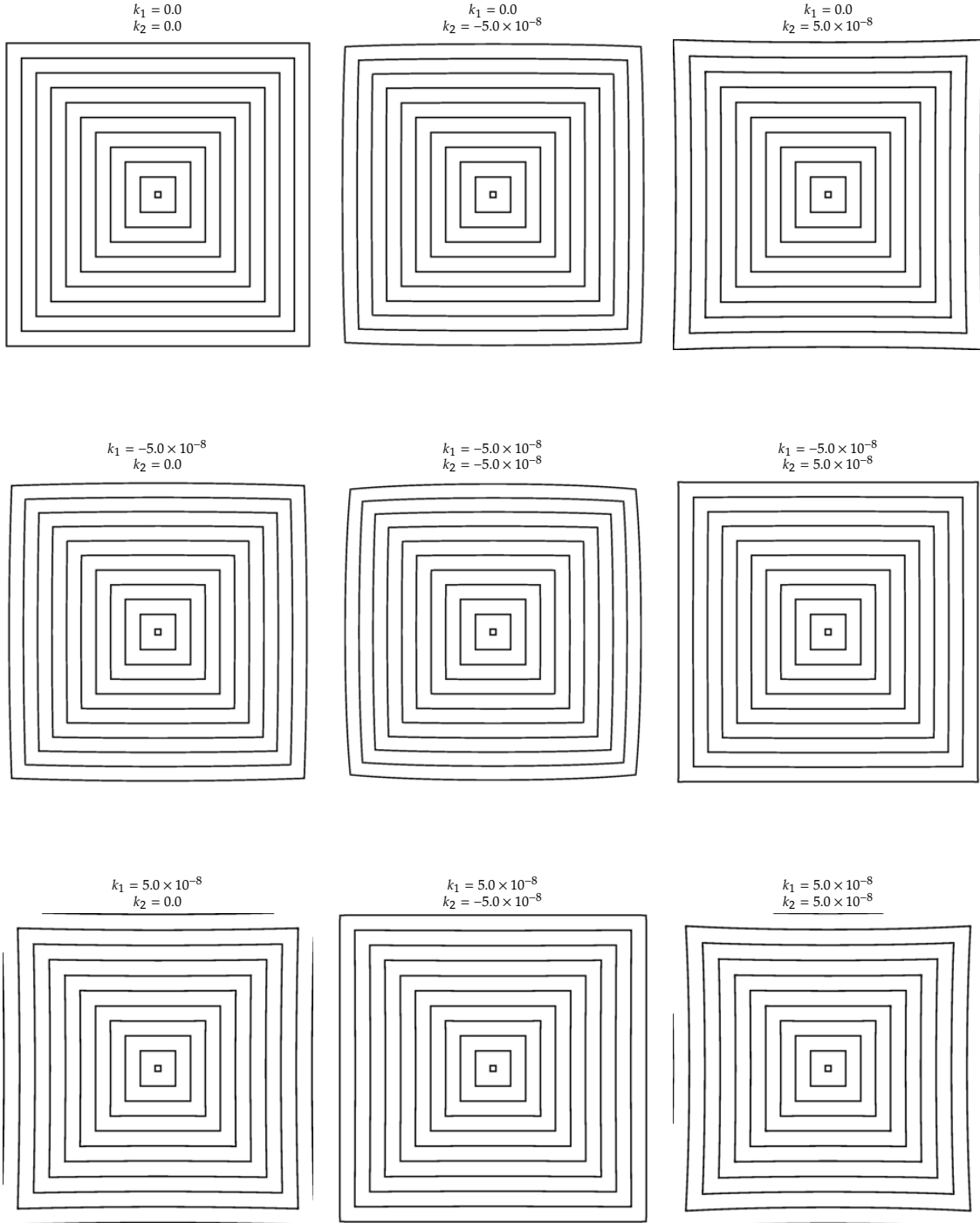


Figure 1: Examples of radial distortion.

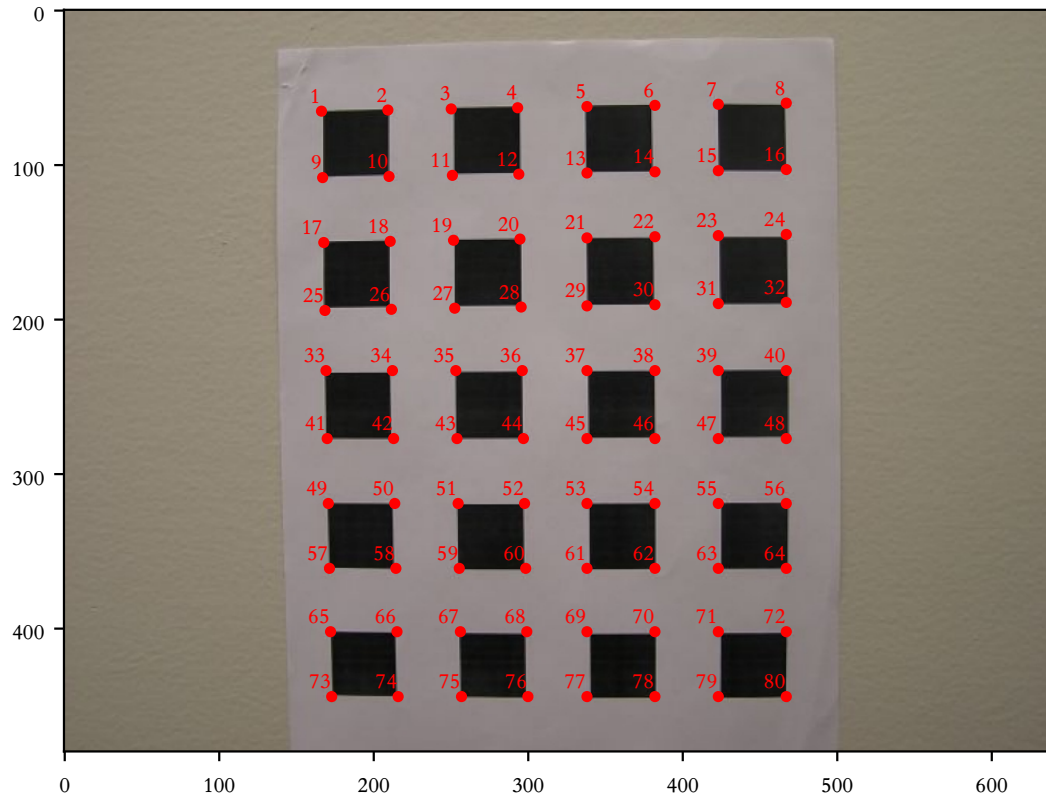


Figure 2: Vertices and their numbers in the calibration pattern. The image is Pic_13 from the provided dataset.

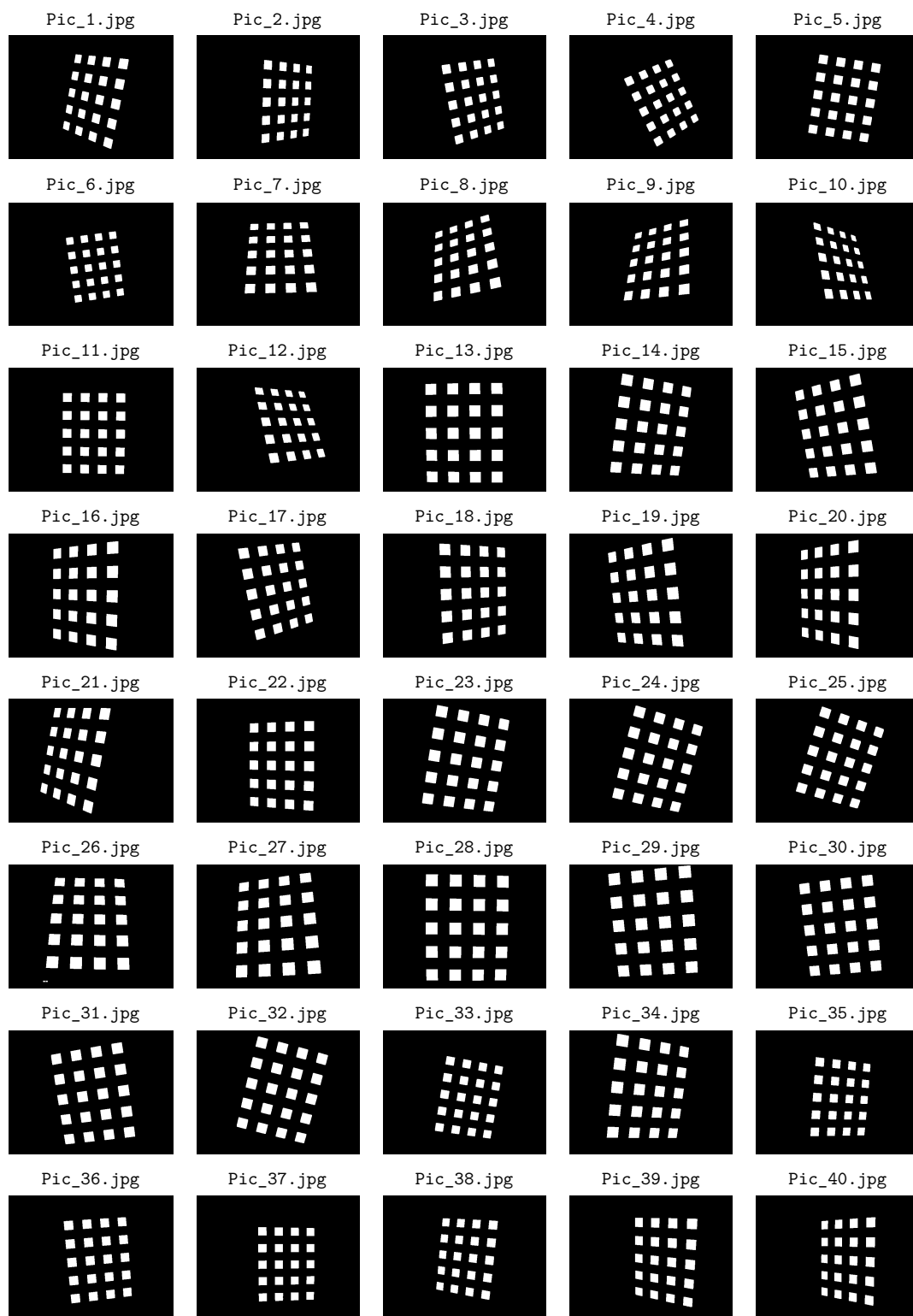


Figure 3: The output of corner detection step 1.

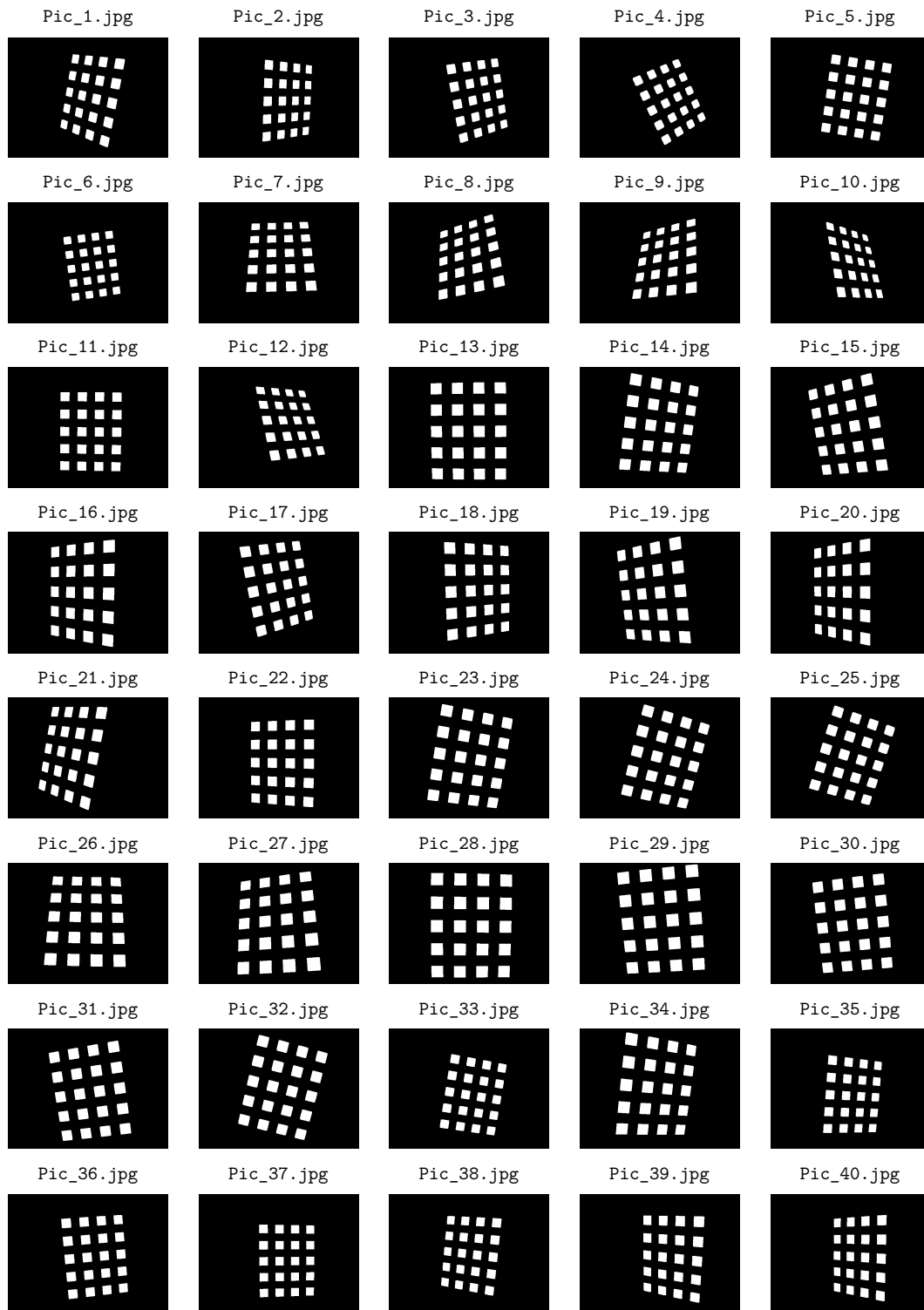


Figure 4: The output of corner detection step 2.

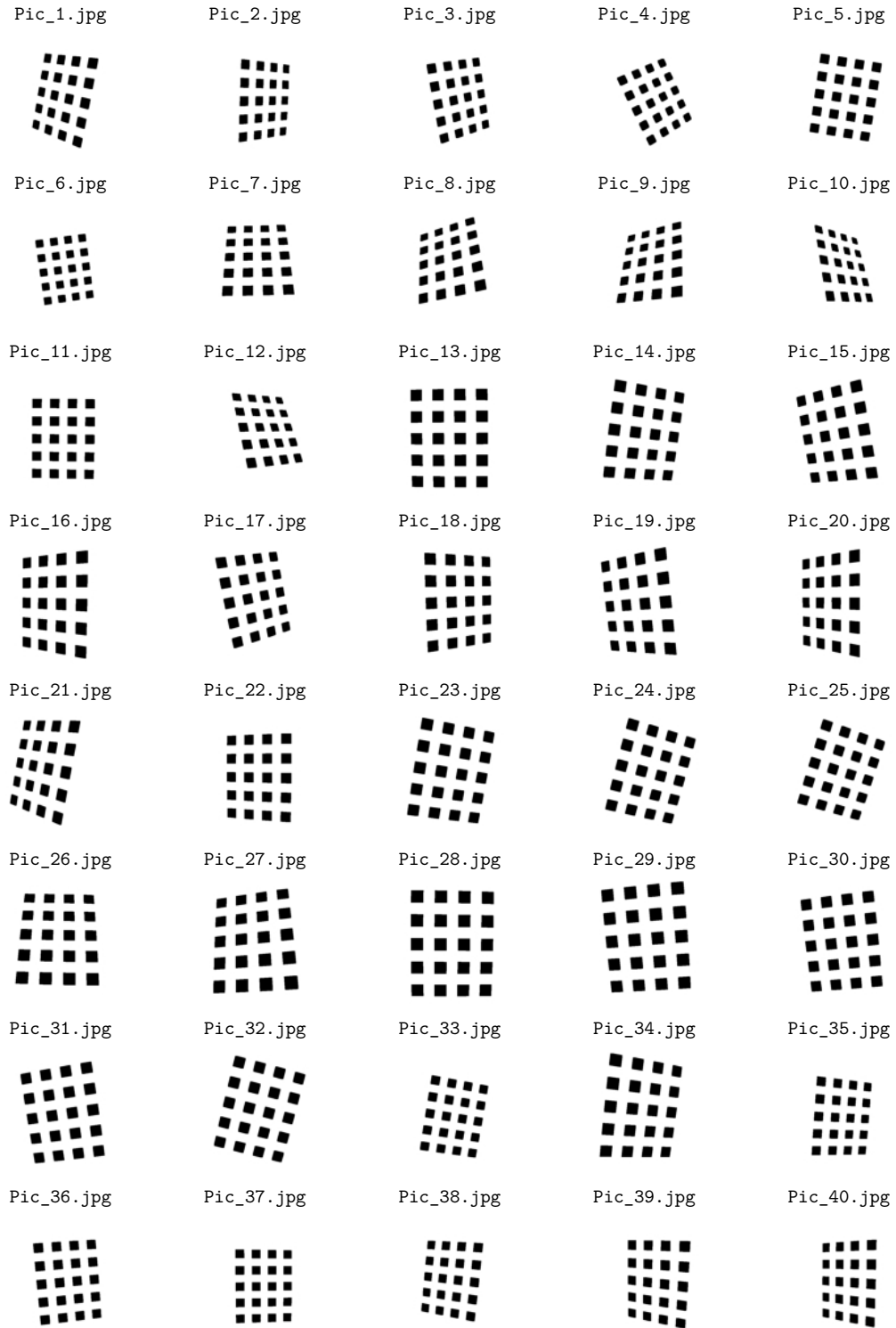


Figure 5: The output of corner detection step 3.

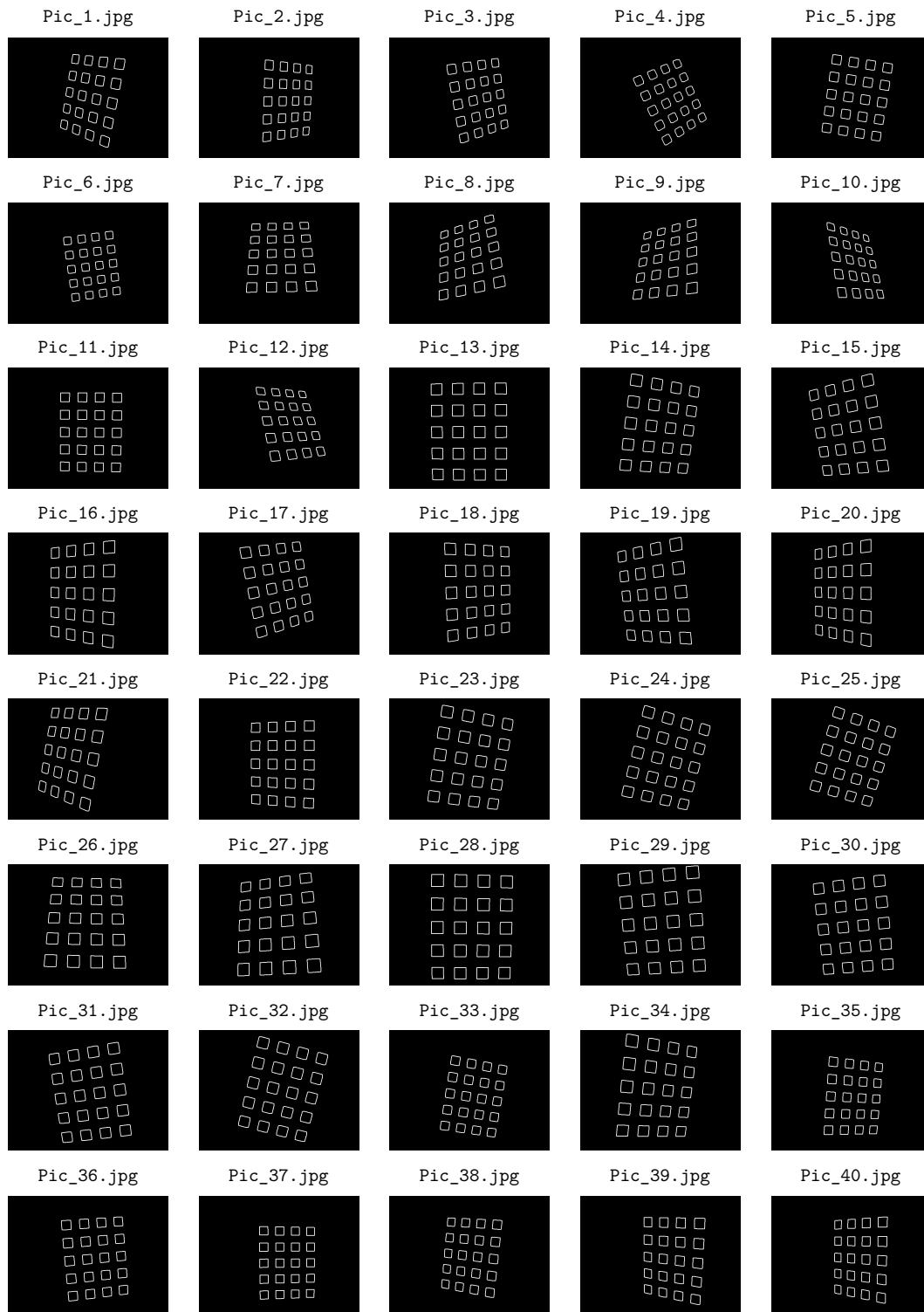


Figure 6: The output of corner detection step 4.

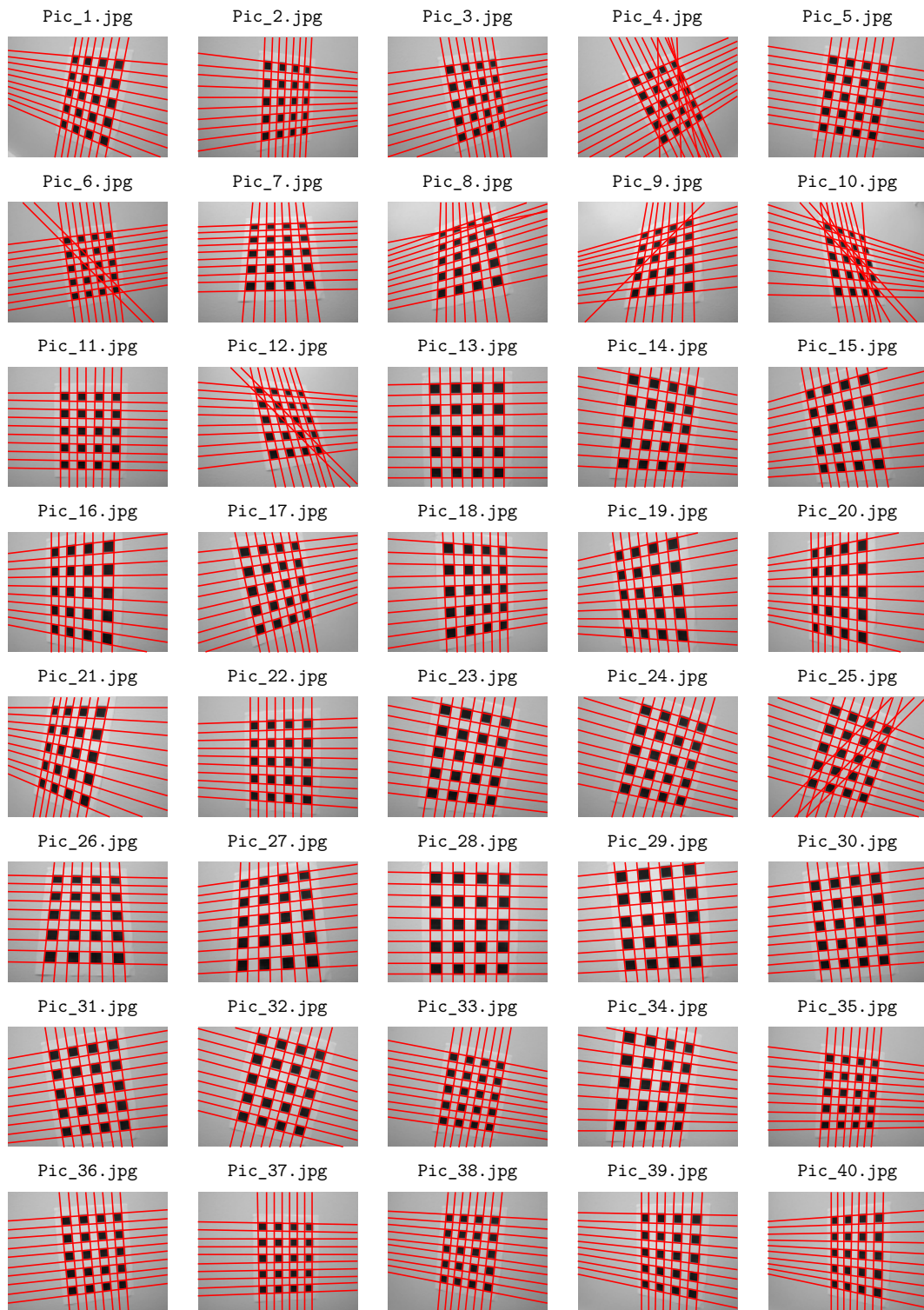


Figure 7: The output of corner detection step 5.

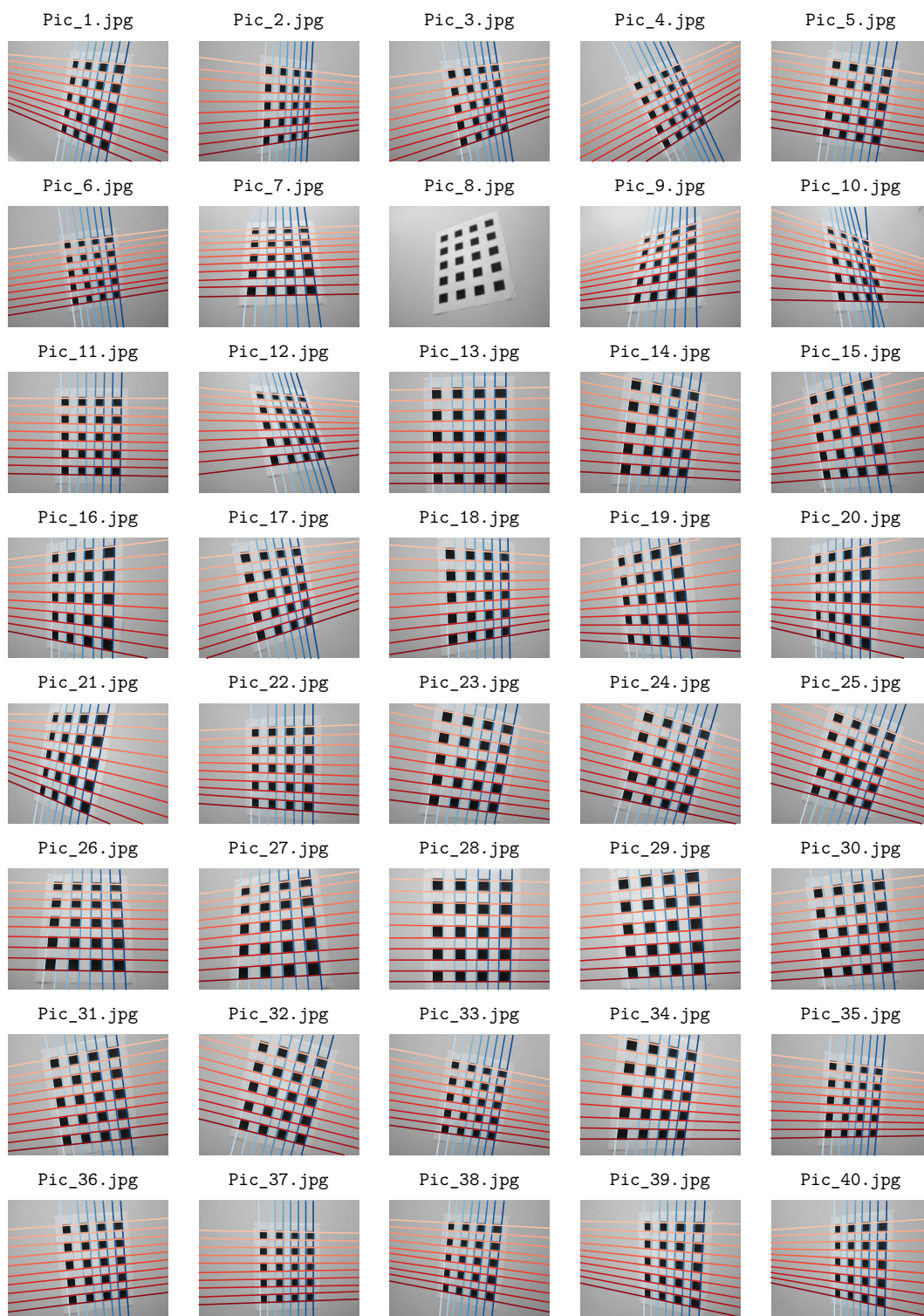


Figure 8: The output of corner detection step 6. The lines have been separated into two groups marked with red and blue, respectively; within each group, the spatial order (index) of each line is encoded using color gradient from white to red (blue).

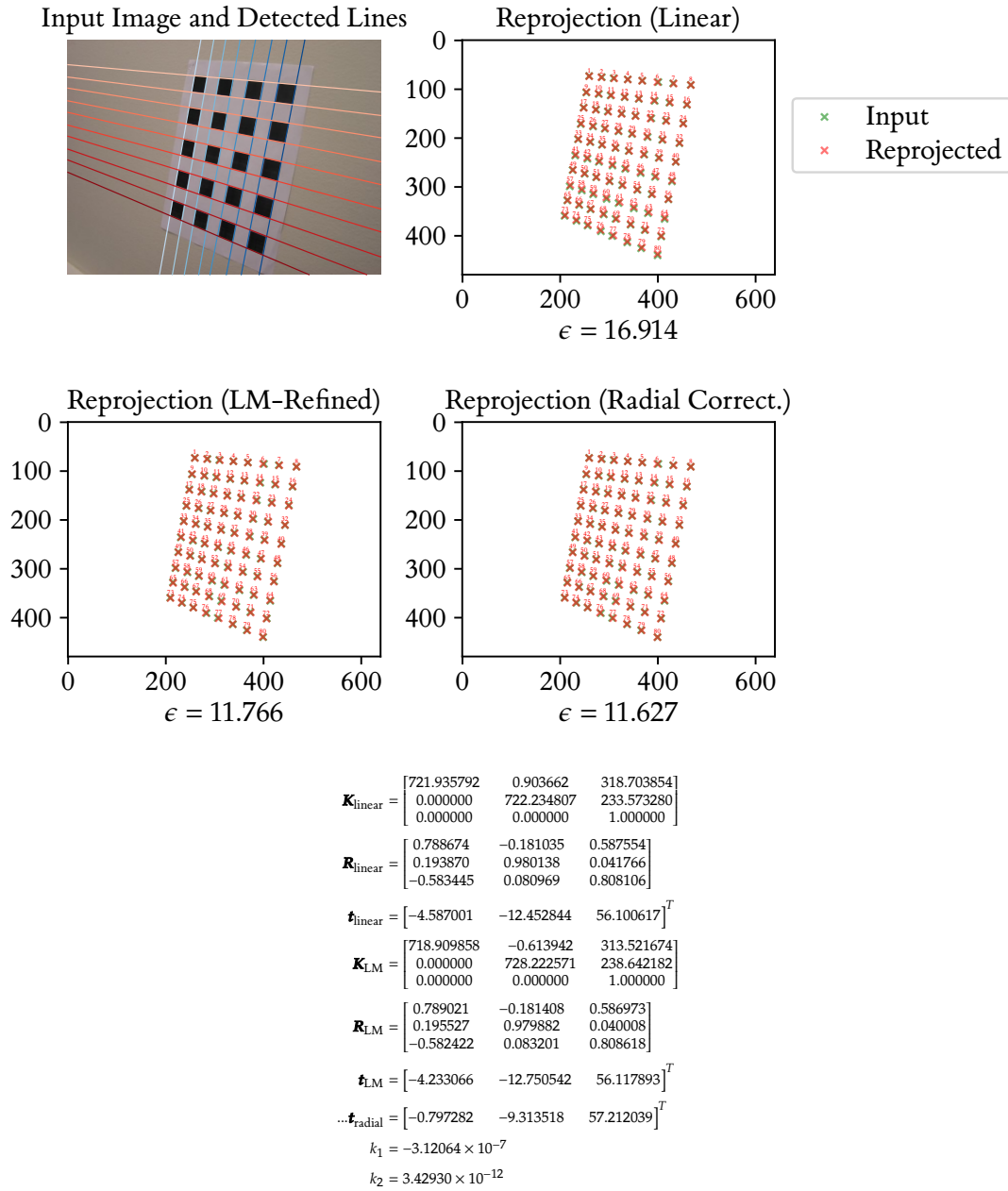


Figure 9: The reprojection results for the `PIC_1` image. Here, the original calibration pattern is projected to the input image.

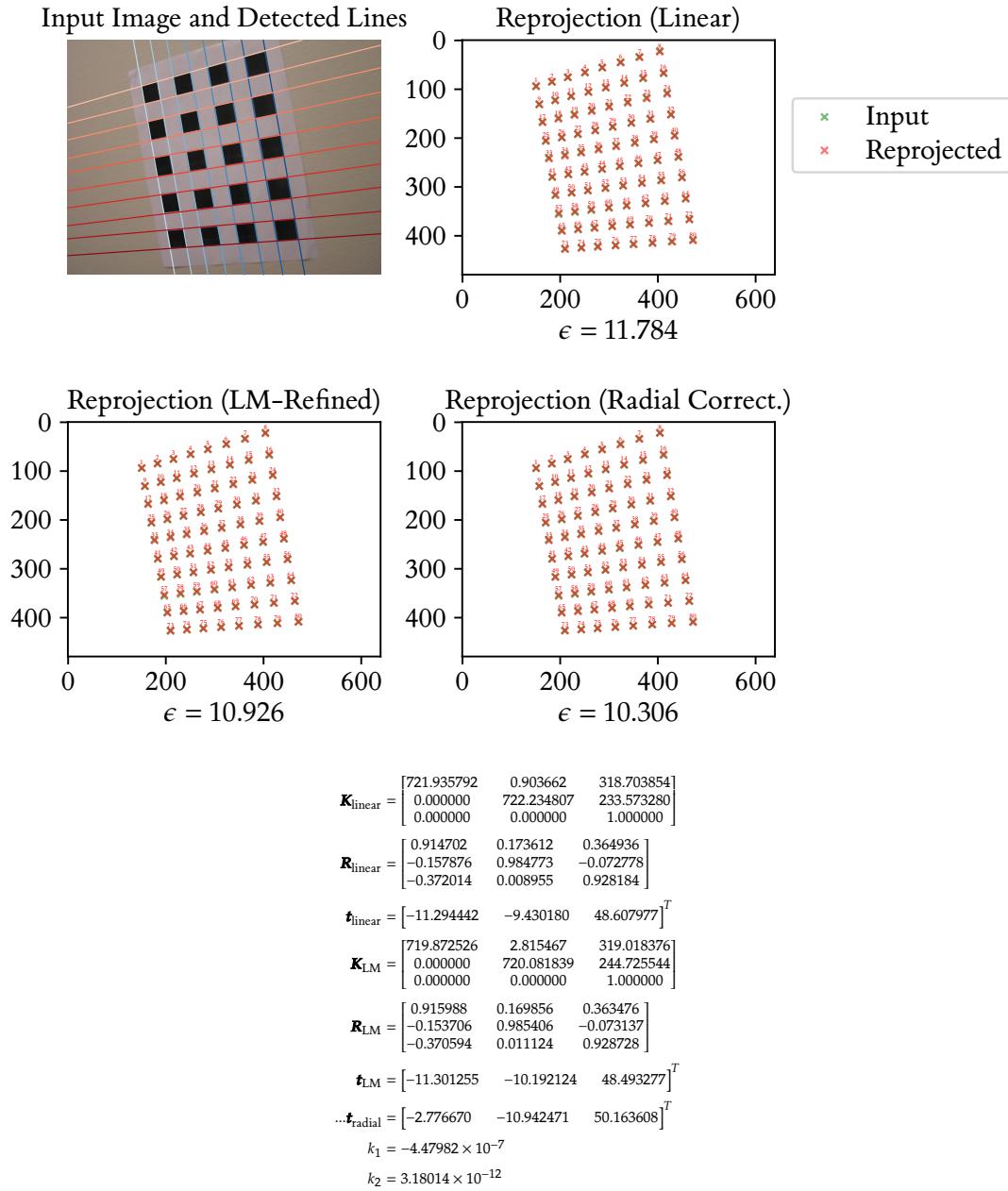


Figure 10: The reprojection results for the `Pic_15` image. Here, the original calibration pattern is projected to the input image.

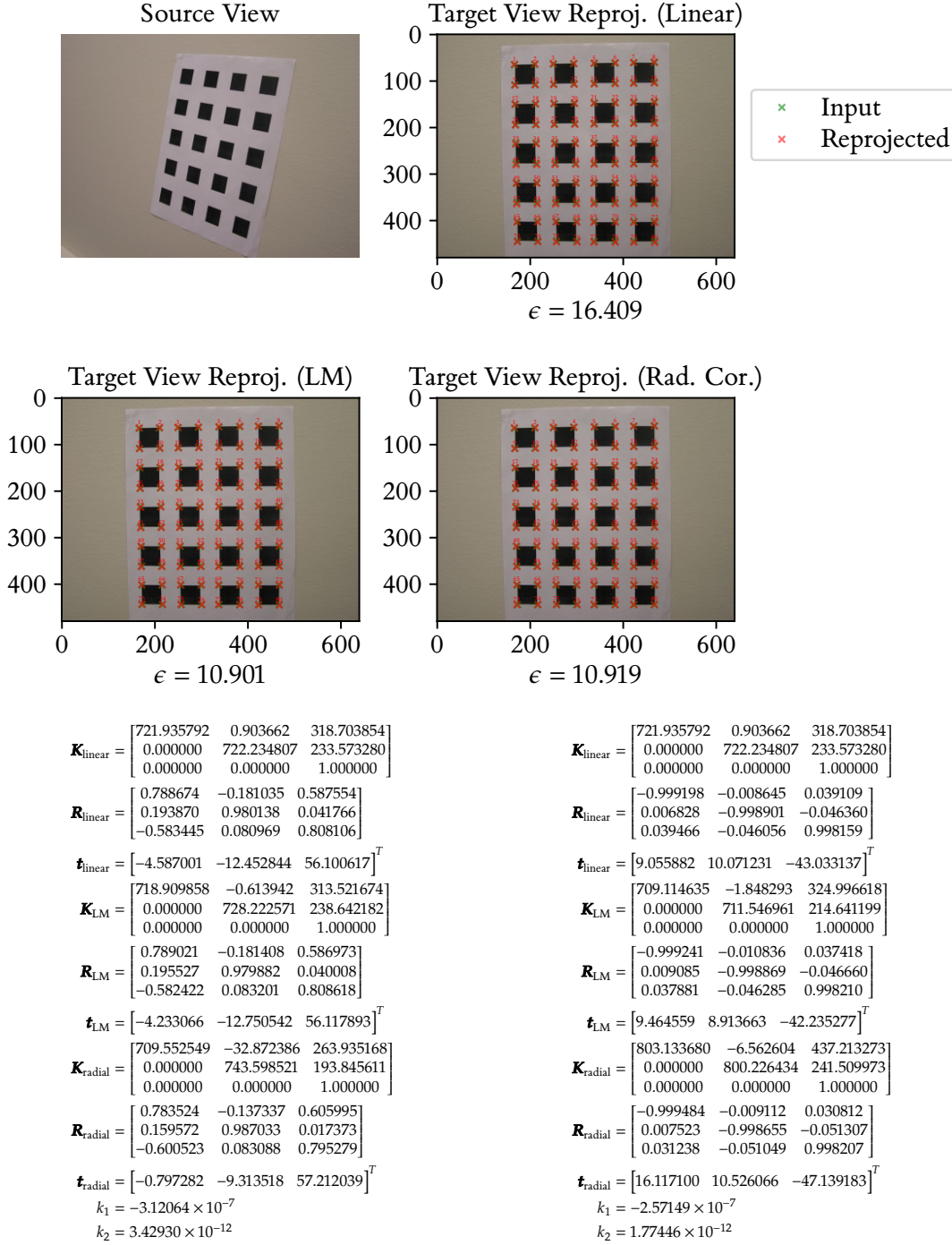


Figure 11: The reprojection of Pic₁ (source image) to Pic₁₃ (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.

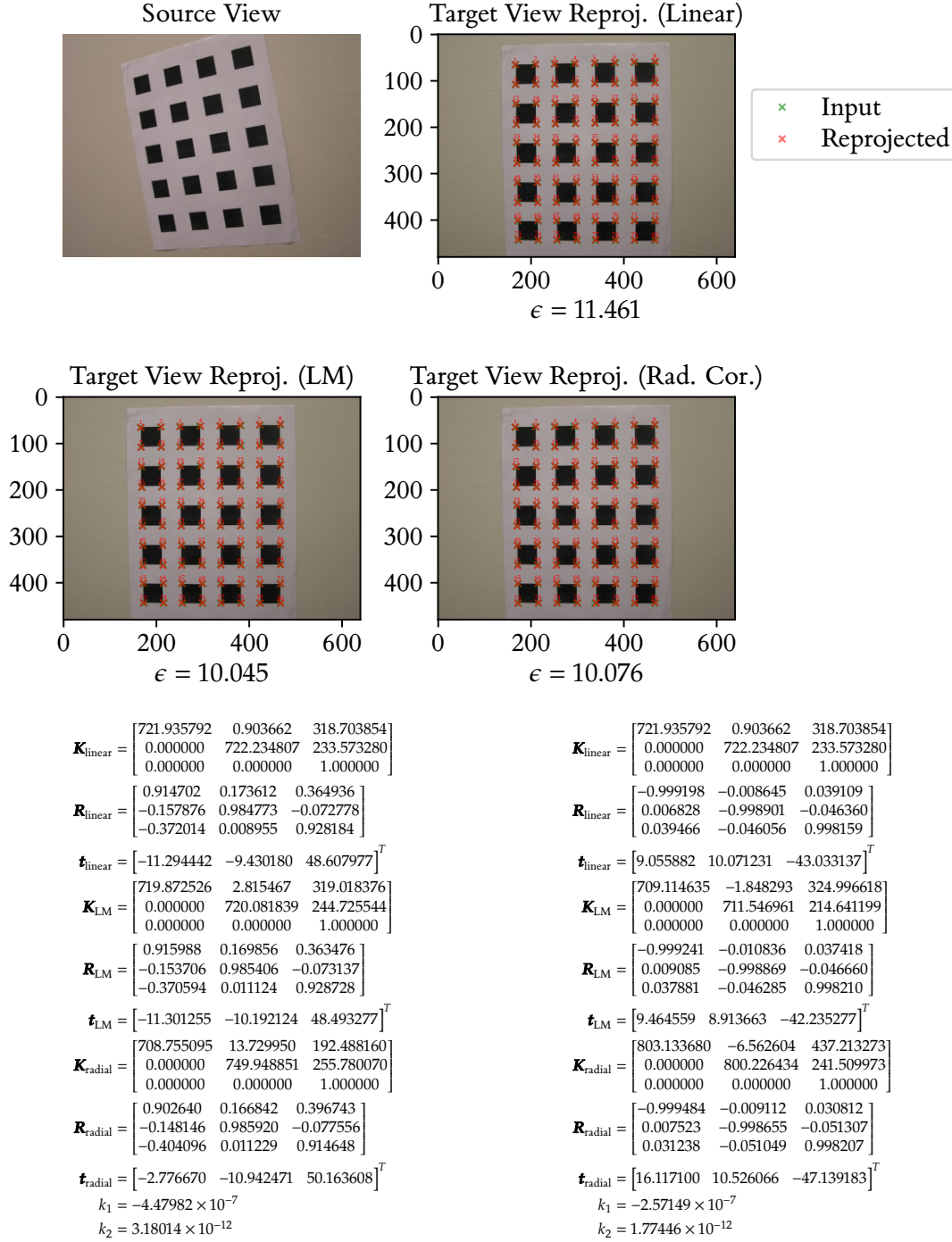


Figure 12: The reprojection of Pic_{15} (source image) to Pic_{13} (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.

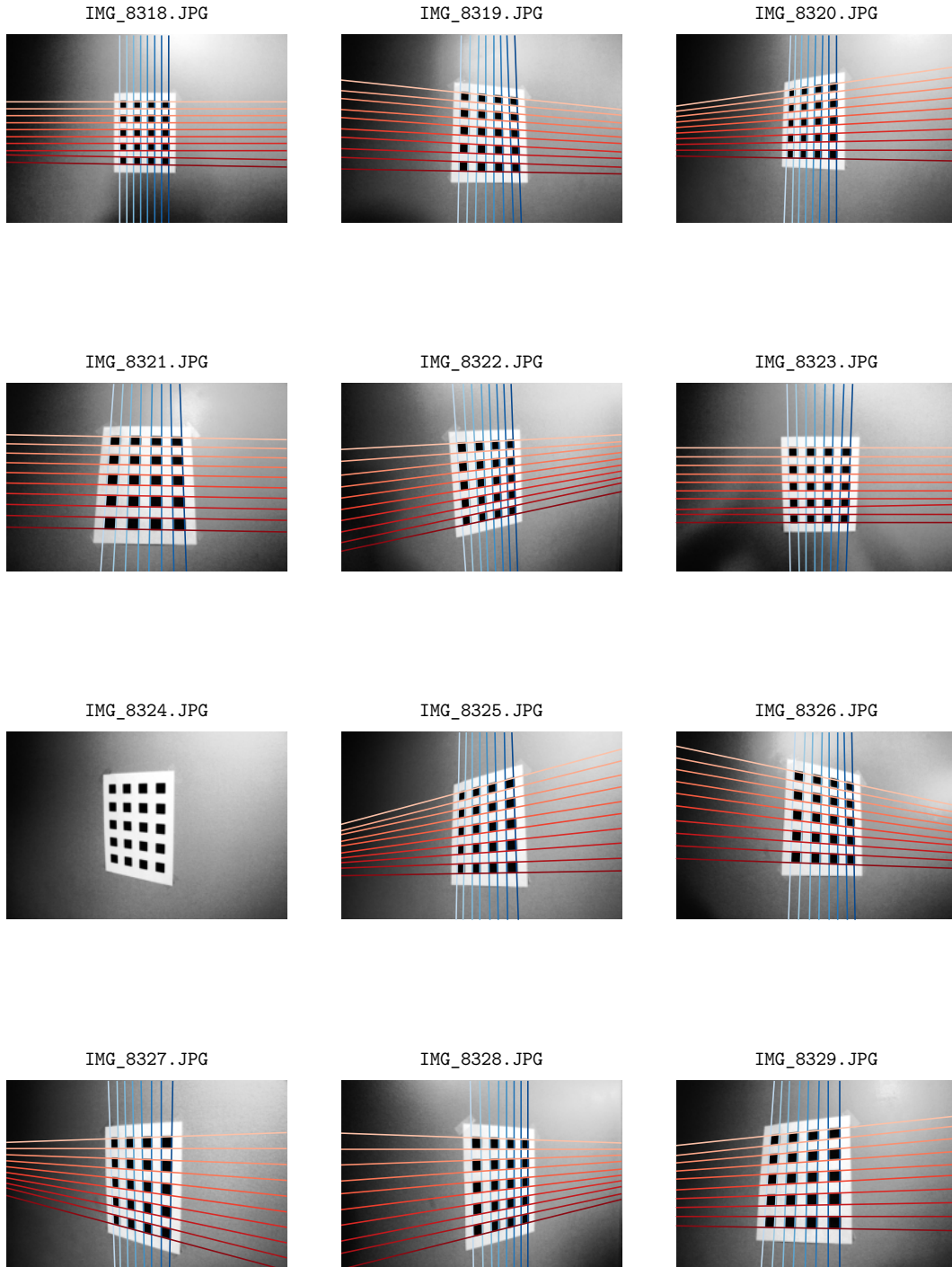


Figure 13: The output of corner detection on our custom dataset. The lines have been separated into two groups marked with red and blue, respectively; within each group, the spatial order (index) of each line is encoded using color gradient from white to red (blue).

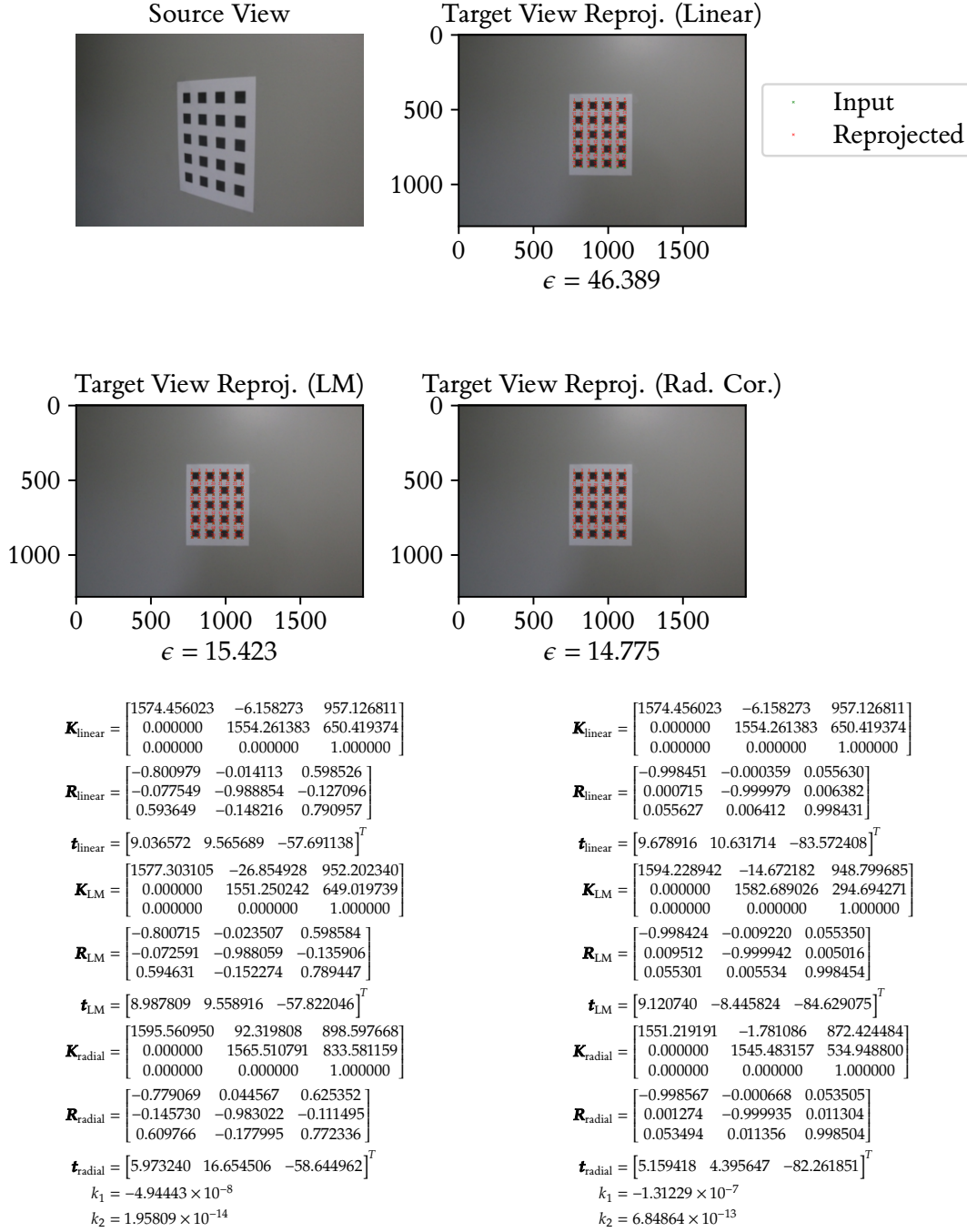


Figure 14: The reprojection of IMG_8327.JPG (source image) to IMG_8318.JPG (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.

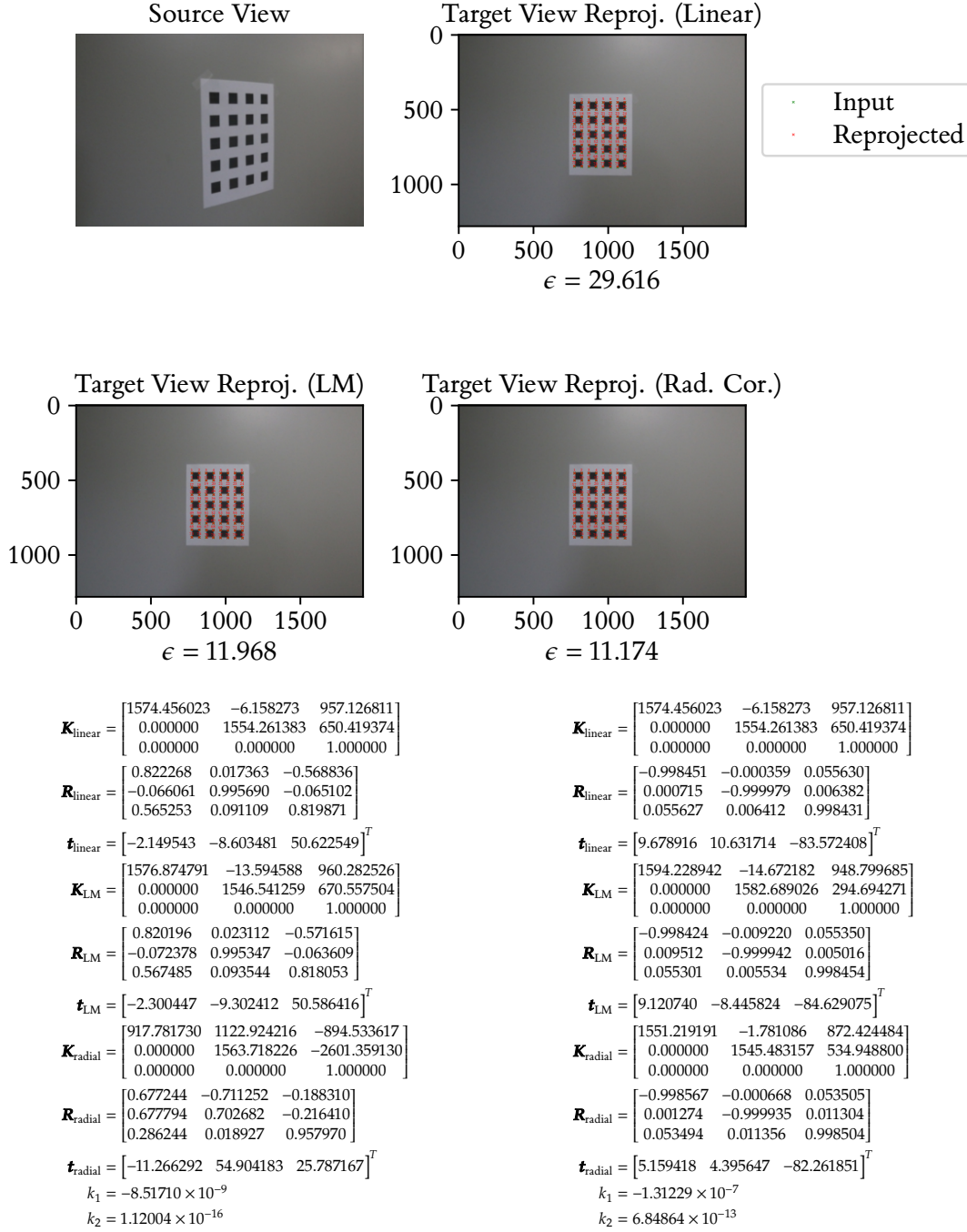


Figure 15: The reprojection of IMG_8328.JPG (source image) to IMG_8318.JPG (target image). The green crosses indicate corners in the target image; the red crosses indicate corners reprojected from the source image. The calibration parameters shown on the left and right hand side are of the source and target image, respectively.

V. Tables

Image	Linear	LM-Refined	Radial Distortion Corrected
Pic_1.jpg	16.914	11.766	11.627
Pic_2.jpg	17.721	8.124	7.995
Pic_3.jpg	17.031	11.150	10.788
Pic_4.jpg	16.518	13.847	13.401
Pic_5.jpg	11.405	10.580	10.435
Pic_6.jpg	10.020	8.594	8.406
Pic_7.jpg	11.752	7.454	6.407
Pic_9.jpg	11.986	9.368	9.287
Pic_10.jpg	98.293	65.873	48.685
Pic_11.jpg	9.568	7.966	6.910
Pic_12.jpg	31.851	8.851	8.154
Pic_13.jpg	10.392	9.589	9.275
Pic_14.jpg	19.263	13.154	12.740
Pic_15.jpg	11.784	10.926	10.306
Pic_16.jpg	15.001	12.155	11.705
Pic_17.jpg	11.572	11.468	11.169
Pic_18.jpg	15.980	9.436	9.088
Pic_19.jpg	12.008	10.887	10.136
Pic_20.jpg	13.073	10.110	9.612
Pic_21.jpg	20.226	12.012	11.388
Pic_22.jpg	11.170	9.213	8.867
Pic_23.jpg	13.752	13.523	13.089
... Pic_38.jpg	14.336	10.093	9.595
Pic_39.jpg	39.842	9.338	9.229
Pic_40.jpg	28.273	8.784	8.297

Table 1: The reprojection error ϵ of various camera calibration methods. Here, we reproject the original calibration pattern to the input image to compute the reprojection error.

Image	Linear	LM-Refined	Radial Distortion Corrected
IMG_8318.JPG	23.925	10.400	9.984
IMG_8319.JPG	24.003	13.180	12.650
IMG_8320.JPG	65.362	9.765	9.260
IMG_8321.JPG	21.965	13.370	12.753
IMG_8322.JPG	23.892	16.143	14.293
IMG_8323.JPG	23.038	15.022	14.813
IMG_8325.JPG	36.390	16.138	15.740
IMG_8326.JPG	16.029	13.905	13.313
IMG_8327.JPG	37.961	14.846	14.288
IMG_8328.JPG	19.230	14.872	13.623
IMG_8329.JPG	20.620	17.743	17.035

Table 2: The reprojection error ϵ of various camera calibration methods on our custom dataset. Here, we reproject the original calibration pattern to the input image to compute the reprojection error.

A. Source Code

A.1. Source Listing of code/common.py

```
1 import matplotlib.pyplot as plt
2 plt.rcParams['svg.fonttype'] = 'none'
3
4 from IPython.display import set_matplotlib_formats
5 set_matplotlib_formats('svg')
6
7 import os
8 import lxml.html
9 import lxml.etree
10 from PIL import Image
11 import io
12 import base64
13
14 notebook_path = os.path.abspath(os.getcwd())
15 tex_src_path = os.path.split(notebook_path)[0]
16
17 print(f'notebook path is {notebook_path}')
18 print(f'TeX source path is {tex_src_path}')
19
20 # use jpeg compression
21 def save_jpeg_svg(out_fn, **savefig_kwargs):
22     bio = io.BytesIO()
23     plt.savefig(bio, format='svg', **savefig_kwargs)
24     bio.seek(0)
25     svg = lxml.html.parse(bio)
26     for ele in svg.findall('//image'):
27         image_str = ele.get('xlink:href')
28         if image_str.startswith('data:image/'):
29             image_format, _, image_b64 = image_str.partition(',')
30             image_bytes = base64.b64decode(image_b64.encode())
31             imgbio = io.BytesIO()
32             imgbio.write(image_bytes)
33             imgbio.seek(0)
34             image_data = Image.open(imgbio)
35             if image_data.mode == 'RGBA':
36                 image_data = image_data.convert('RGB')
37             # save the image as jpeg
38             imgbio = io.BytesIO()
39             image_data.save(imgbio, format='jpeg', quality=100)
40             # encode as b64
41             image_jpeg_b64 = 'data:image/jpeg;base64,' + base64.b64encode(imgbio.getvalue()).decode()
42             ele.set('xlink:href', image_jpeg_b64)
43     print('successfully encoded one image element using jpeg')
44     svg_root = svg.find('//svg')
45     new_svg = lxml.etree.tostring(svg_root, pretty_print=True)
46     with open(out_fn, 'wb') as outfile:
47         outfile.write(new_svg)
```

A.2. Source Listing of code/helper.py

```
1 import numpy as np
2
3 def table_to_latex(table, table_style='booktabs'):
4     nrows = len(table)
5     ncols = len(table[0])
6     for row in table:
7         assert len(row) == ncols
8
9     if table_style == 'booktabs':
10         tabular_spec = 'c' * ncols
11     else:
12         tabular_spec = '|c|' * 'c|' * (ncols-1)
13
14     # find length
15     str_array = np.asarray(table, dtype=np.object)
16     str_array_len = np.vectorize(len)(str_array)
17     col_len = np.max(str_array_len, axis=0)
18     def row_formatter(*cols):
19         res = []
20         for i, col_data in enumerate(cols):
21             string_fmt = '{:%ds}' % col_len[i]
22             res.append(string_fmt.format(col_data))
23         return res
24
25     latex_rows = [' & '.join(row_formatter(*row)) for row in table]
26     if table_style == 'booktabs':
27         latex_content = latex_rows[0] + '\\\\midrule \n'
28         end_of_line = '\\\\ \n'
29         latex_content += end_of_line.join(latex_rows[1:]) + '\\\\bottomrule \n'
30     else:
31         end_of_line = '\\\\ \\hline\n'
32         latex_content = end_of_line.join(latex_rows) + end_of_line
33
34     if table_style == 'booktabs':
35         latex_string = '\\begin{tabular}{%s}\n\\toprule\n%s\\end{tabular}' % (tabular_spec, latex_content)
36     else:
37         latex_string = '\\begin{tabular}{%s}\n\\hline\n%s\\end{tabular}' % (tabular_spec, latex_content)
38
39     return latex_string
40
41 def latex_e_notation(s):
42     if 'e' in s:
43         front, e, exp = s.partition('e')
44         exp = int(exp)
45         if exp != 0:
46             s = r'{}\times 10^{{{}}}'.format(front, exp)
47         else:
48             s = front
49     return s
50
51 def latex_2d_array(arr, fmt=':06f'):
52     arr = np.asarray(arr)
53     assert arr.ndim == 2
54
55     fmt_str = '{%s}' % fmt
56
57     row_strs = []
58     for row in arr.tolist():
59         col_str = [fmt_str.format(col) for col in row]
60         for i in range(len(col_str)):
```

```

61         col_str[i] = latex_e_notation(col_str[i])
62
63     row_strs.append(
64         ' & '.join(col_str)
65     )
66
67     return '\\\\n'.join(row_strs)

```

A.3. Source Listing of code/func_impl.py

```

1  import numpy as np
2  import inspect
3  from inspect import signature
4  from sklearn.cluster import SpectralClustering
5  from collections import Counter
6  import copy
7  import scipy.spatial.distance
8  import skimage, skimage.io, skimage.filters, skimage.transform, \
9      skimage.color, skimage.feature, skimage.measure, skimage.morphology, \
10     skimage.exposure
11
12
13  # convert to homogeneous coordinates
14  def p_to_hc(points):
15      points = np.asarray(points)
16      assert points.ndim == 2
17      assert points.shape[1] == 2
18      ones = np.ones((points.shape[0], 1), points.dtype)
19      return np.concatenate([points, ones], axis=1)
20
21  # convert to physical coordinates
22  def p_to_phy(points):
23      points = np.asarray(points)
24      assert points.ndim == 2
25      assert points.shape[1] == 3
26      points = points.astype(np.float64)
27      points = points / points[:, -1, np.newaxis]
28      assert np.max(np.abs(points[:, -1] - 1)) < 1.0e-6
29      return points[:, :-1]
30
31
32  # returns the transformation matrix from points1 to points2 (homogeneous version)
33  def solve_linear_correspondence(points1, points2):
34      points1 = np.asarray(points1)
35      points2 = np.asarray(points2)
36      assert points1.shape == points2.shape
37
38      # construct the matrix based on points
39      A = []
40      for i in range(points1.shape[0]):
41          x = points1[i]
42          xp = points2[i]
43
44          row = []
45          row.append([0] * 3)
46          row.append(-xp[2] * x)
47          # row.append(-x) # because w' is one
48          # row.append([-1])
49          row.append(xp[1] * x)
50          # row.append([xp[1]])
51          A.append(np.concatenate(row))
52
53          row = []
54          row.append(xp[2] * x)
55          # row.append(x) # because w' is one
56          # row.append([1])
57          row.append([0] * 3)
58          row.append(-xp[0] * x)
59          # row.append([-xp[0]])
60          A.append(np.concatenate(row))
61
62      A = np.asarray(A, dtype=np.float64)
63      # print(A.shape)
64      u,d,vh = np.linalg.svd(A)
65      h = vh[-1, :] # use the last column of the V matrix as the solution
66
67      return h.reshape((3,3))
68
69
70  def angle_dist_func(a, b):
71      a, b = sorted((a, b))
72      return min(b-a, 180 + a - b)
73
74  def angle_sim_func(a, b):
75      ret = 1 - angle_dist_func(a,b) / 180
76      assert 0 <= ret <= 1
77      return ret
78
79  # get the point grid of the calibration pattern
80  # we use centimeter as the unit
81  def get_default_point_grid():
82      in_to_cm = 2.54
83      point_grid = np.zeros((10, 8, 2), np.float64)
84
85      for hind in range(point_grid.shape[0]):
86          for vind in range(point_grid.shape[1]):
87              point_grid[hind, vind, 0] = in_to_cm * vind
88              point_grid[hind, vind, 1] = in_to_cm * hind
89
90      return point_grid
91
92  # corner detection class
93  class CornerDetection:
94
95      def __init__(self,
96          manual_threshold=None, # use a manual threshold instead of otsu
97          eq_hist=False, # if equalize the hist of gray image
98          thres1_edge_crop=None, # if crop the true values in the binary image
99          thres1_opening_width=5, # binary opening after Otsu
100         label_hist_nbins=10, # number of bins for the histogram of connected component sizes
101         label_hist_top_index=8, # number of top bins to consider as markers
102         thres2_dilation_width=3, # dilation width after extracting the marker mask
103         gaussian_std=1, # the std for Gaussian blur before canny
104         canny_dilation_width=1, # the dilation width of canny result
105         hough_angles=(-np.pi/2, np.pi/2, 180), # angles of the hough transform
106         hough_min_distance=15, # parameter for finding peaks in hough transform
107         hough_min_angle=15, # parameter for finding peaks in hough transform

```

```

108     hough_threshold=0.4, # parameter for finding peaks in hough transform
109     canny_edge_cancel=4, # cancel edges in the canny result (this determines the width of cancellation)
110     max_n_line_clusters=6, # maximum number of clusters when doing spectral clustering
111     line_in_group_first_n_values=5 # number of smallest in group intersection distance to inspect when doing line grouping
112 ):
113
114     # save the parameters
115     func_signature = signature(CornerDetection.__init__)
116     for param_name, param in func_signature.parameters.items():
117         if param_name != 'self':
118             if param.default != inspect._empty:
119                 setattr(self, param_name, locals()[param_name])
120
121
122     # determine a marker mask to mask the image before applying canny edge detection
123     # and hough line transform
124     # this can eliminate noise in the image
125     def _marker_mask(self, state):
126         if self.manual_threshold is not None:
127             thres = self.manual_threshold
128         else:
129             # apply otsu
130             thres = skimage.filters.threshold_otsu(state['img_gray'])
131         # threshold the image
132         img_thres1 = state['img_gray'] < thres
133
134
135         if self.thres1_edge_crop is not None:
136             img_thres1[:,self.thres1_edge_crop, :] = False
137             img_thres1[-self.thres1_edge_crop:, :] = False
138             img_thres1[:, :self.thres1_edge_crop] = False
139             img_thres1[:, -self.thres1_edge_crop:] = False
140
141         # do morphology
142         img_thres1 = skimage.morphology.binary_opening(img_thres1, skimage.morphology.square(self.thres1_opening_width))
143
144         # do connected component labeling
145         labels = skimage.measure.label(img_thres1)
146         label_sizes = [np.count_nonzero(labels == i) for i in range(1, labels.max() + 1)]
147         label_size_hist = np.histogram(label_sizes, bins=np.linspace(0, np.max(label_sizes), self.label_hist_nbins + 1))
148         label_size_thres = label_size_hist[1][-self.label_hist_top_index]
149         good_labels = []
150         for i, size in enumerate(label_sizes):
151             if size >= label_size_thres:
152                 good_labels.append(i+1)
153
154         img_thres2 = np.zeros_like(img_thres1)
155         for label in good_labels:
156             img_thres2[labels == label] = True
157
158         # do morphology
159         img_thres2 = skimage.morphology.binary_dilation(img_thres2, skimage.morphology.square(self.thres2_dilation_width))
160
161         # use img_thres2 to mask the original input image
162         img_gray_masked = state['img_gray'].copy()
163         img_gray_masked[np.bitwise_not(img_thres2)] = 255
164
165         state['img_thres1'] = img_thres1
166         state['label_size_hist'] = label_size_hist
167         state['img_thres2'] = img_thres2
168         state['img_gray_masked'] = img_gray_masked
169
170     # use canny edge detection and hough line transform to detect lines (in the masked image)
171     def _line_detection(self, state):
172         # now we operate on the masked image
173         # apply canny edge detection
174         img_canny = skimage.feature.canny(state['img_gray_masked'], self.gaussian_std)
175         # enhance the canny edge results with dilation
176         img_canny = skimage.morphology.binary_dilation(img_canny, skimage.morphology.disk(self.canny_dilation_width))
177
178         # somehow canny introduces blank regions on the edge... we need to get rid of those
179         img_canny[:,self.canny_edge_cancel, :] = False
180         img_canny[-self.canny_edge_cancel:, :] = False
181         img_canny[:, :self.canny_edge_cancel] = False
182         img_canny[:, -self.canny_edge_cancel:] = False
183
184         # hough transform
185         tested_angles = np.linspace(*self.hough_angles, endpoint=False)
186         hough_result = skimage.transform.hough_line(img_canny, theta=tested_angles)
187         hough_peaks = skimage.transform.hough_line_peaks(*hough_result,
188             min_distance=self.hough_min_distance,
189             min_angle=self.hough_min_angle,
190             threshold=self.hough_threshold * np.max(hough_result[0])
191         )
192
193         state['img_canny'] = img_canny
194         state['hough_peaks'] = hough_peaks
195
196     # organize a line group by distance
197     # acquire their corresponding homogeneous representation
198     def _organize_line_group(self, hough_peaks, group_indices, criterion='v'):
199         assert criterion in ('h', 'v')
200         all_dist = []
201         all_hc = []
202
203         x_axis = np.asarray([0,1,0], dtype=np.float64)
204         y_axis = np.asarray([1,0,0], dtype=np.float64)
205
206         for ind in group_indices:
207             angle = hough_peaks[1][ind]
208             dist = hough_peaks[2][ind]
209
210             x0, y0 = dist * np.asarray([np.cos(angle), np.sin(angle)])
211
212             slope = np.tan(angle + np.pi / 2)
213             if abs(slope) > 1.0e7:
214                 all_hc.append([1, 0, -x0])
215             else:
216                 all_hc.append([slope, -1, y0 - slope * x0])
217
218             if criterion == 'v':
219                 itsc = np.cross(all_hc[-1], y_axis)
220                 itsc = itsc / itsc[2]
221                 all_dist.append(itsc[1])
222             else:
223                 itsc = np.cross(all_hc[-1], x_axis)
224                 itsc = itsc / itsc[2]
225                 all_dist.append(itsc[0])
226
227         dist_sort = np.argsort(all_dist)
228         all_hc = np.asarray(all_hc)
229         all_hc = all_hc[dist_sort]

```

```

230         return all_hc
231
232     # classify the detected lines into two groups (8,10 lines)
233     # try cancelling the noise in line detection
234     def _line_grouping(self, state):
235         angle_samples = np.rad2deg(np.asarray(state['hough_peaks'])[1]).reshape(-1, 1)
236         state['angle_samples'] = angle_samples
237
238         angle_groups = dict()
239         cluster_results = dict()
240         if angle_samples.size < 18:
241             raise RuntimeError('less than 18 lines detected')
242         else:
243
244             for j in range(2, self.max_n_line_clusters + 1):
245                 cluster_results[j] = dict()
246                 clustering = SpectralClustering(n_clusters=j, affinity=angle_sim_func)
247                 cluster_pred = clustering.fit_predict(angle_samples)
248                 cluster_pred_counter = Counter(cluster_pred)
249                 _8_found = None
250                 _10_found = None
251                 for key, val in cluster_pred_counter.items():
252                     if val == 8:
253                         _8_found = (key, val)
254                     if val == 10:
255                         _10_found = (key, val)
256
257                 cluster_results[j]['cluster_pred'] = cluster_pred
258                 cluster_results[j][8] = _8_found
259                 cluster_results[j][10] = _10_found
260
261                 if _8_found is not None and _10_found is not None:
262                     angle_groups[8] = np.where(cluster_pred == _8_found[0])[0]
263                     angle_groups[10] = np.where(cluster_pred == _10_found[0])[0]
264                     break
265
266             if len(angle_groups) == 0:
267                 raise RuntimeError('unable to find angle groups')
268
269             # # in this case, just spectral clustering is not enough
270             # if len(angle_groups) == 0:
271             #     # we look at the clustering results of 2 clusters
272             #     # and see if we can at least get the 8 or 10 group
273             #     good_group = None
274             #     if cluster_results[2][8] is not None:
275             #         good_group = 8
276             #     if cluster_results[2][10] is not None:
277             #         good_group = 10
278             #     if good_group is None:
279             #         raise RuntimeError('unable to determine line groups')
280
281             #     assert good_group in (8, 10)
282             #     other_group = 10 if good_group == 8 else 8
283             #     good_group_label = cluster_results[2][good_group][0]
284             #     other_group_label = 1 - good_group_label
285             #     other_group_indices = np.where(cluster_results[2]['cluster_pred'] == other_group_label)[0]
286             #     # organize lines in other group
287             #     other_group_lines = self._organize_line_group(state['hough_peaks'], other_group_indices)
288
289             #     assert other_group_lines.shape[0] > self.line_in_group_first_n_values
290
291             #     # compute mutual intersection and test its distance with respect to the image center
292             #     img_center = np.flip(np.asarray(state['img_gray'].shape) / 2)
293             #     line_intersection_dist = np.zeros((other_group_lines.shape[0], other_group_lines.shape[0]))
294             #     line_intersection_dist[np.diag_indices(line_intersection_dist.shape[0])] = np.inf
295             #     for i in range(other_group_lines.shape[0]):
296             #         for j in range(i):
297             #             itsc = np.cross(other_group_lines[j], other_group_lines[i])
298             #             if abs(itsc[2]) < 1.0e-7:
299             #                 line_intersection_dist[i, j] = np.inf
300             #             else:
301             #                 itsc = itsc / itsc[2]
302             #                 line_intersection_dist[i, j] = np.linalg.norm(itsc[:2] - img_center)
303             #                 line_intersection_dist[j, i] = line_intersection_dist[i, j]
304
305             #     # for each row, look at the smallest values and compare
306             #     line_goodness = [np.mean(np.sort(line_intersection_dist[i][:self.line_in_group_first_n_values]))\
307             #                       for i in range(other_group_lines.shape[0])]
308
309             #     # select the lines with highest line goodness
310             #     line_goodness_sort_index = np.argsort(line_goodness)
311             #     angle_groups[good_group] = np.where(cluster_results[2]['cluster_pred'] == good_group_label)[0]
312             #     angle_groups[other_group] = other_group_indices[line_goodness_sort_index[-other_group:]]
313
314             # print(angle_groups)
315             assert len(angle_groups[8]) == 8
316             assert len(angle_groups[10]) == 10
317
318             state['line_group_8'] = self._organize_line_group(state['hough_peaks'], angle_groups[8], 'h') # sort the 8 line group w.r.t. to their intersections with the x axis
319             state['line_group_10'] = self._organize_line_group(state['hough_peaks'], angle_groups[10], 'v') # sort the 8 line group w.r.t. to their intersections with the y axis
320
321     def _generate_point_grid(self, state):
322         point_grid = np.zeros((10, 8, 2), np.float64)
323         for hind, hline in enumerate(state['line_group_10']):
324             for vind, vline in enumerate(state['line_group_8']):
325                 itsc = np.cross(hline, vline)
326                 itsc = itsc / itsc[2]
327                 point_grid[hind, vind] = itsc[:2]
328
329         state['point_grid'] = point_grid
330
331     def _estimate_pattern_homography(self, state):
332         pattern_point_grid = get_default_point_grid()
333         imgae_point_grid = state['point_grid']
334
335         pattern_point_grid = p_to_hc(pattern_point_grid.reshape(-1, 2))
336         imgae_point_grid = p_to_hc(imgae_point_grid.reshape(-1, 2))
337         homography = solve_linear_correspondence(pattern_point_grid, imgae_point_grid)
338         state['pattern_homography'] = homography
339
340     def __call__(self, img_fn, do_line_grouping=False):
341         img = skimage.io.imread(img_fn)
342         assert img.ndim == 3
343         assert img.shape[2] == 3
344         assert img.dtype == np.uint8
345
346         img_gray = skimage.img_as_float32(skimage.color.rgb2gray(img))
347
348         state = dict()
349         state['img'] = img
350         state['img_gray'] = img_gray

```

```

352         if self.eq_hist:
353             state['img_gray'] = skimage.exposure.equalize_hist(img_gray)
354
355         self._marker_mask(state)
356         self._line_detection(state)
357         if do_line_grouping:
358             try:
359                 self._line_grouping(state)
360                 state['line_grouping_successful'] = True
361             except RuntimeError:
362                 state['line_grouping_successful'] = False
363
364         if state['line_grouping_successful']:
365             self._generate_point_grid(state)
366             self._estimate_pattern_homography(state)
367
368
369         return state

```

A.4. Source Listing of code/environment.yaml

```

1  name: ece66100-cpu
2  channels:
3    - defaults
4  dependencies:
5    - _libgcc_mutex=0.1=main
6    - _openmp_mutex=5.1=1_gnu
7    - anyio=3.5.0=py39h06a4308_0
8    - argon2-cffi=21.3.0=pyhd3eb1b0_0
9    - argon2-cffi-bindings=21.2.0=py39h7f8727e_0
10   - asttokens=2.0.5=pyhd3eb1b0_0
11   - attrs=21.4.0=pyhd3eb1b0_0
12   - babel=2.9.1=pyhd3eb1b0_0
13   - backcall=0.2.0=pyhd3eb1b0_0
14   - beautifulsoup4=4.11.1=py39h06a4308_0
15   - blas=1.0=mkl
16   - bleach=4.1.0=pyhd3eb1b0_0
17   - blosc=1.21.0=h4ff587b_1
18   - brotli=1.0.9=h5eee18b_7
19   - brotli-bin=1.0.9=h5eee18b_7
20   - brotlipy=0.7.0=py39h27cfd23_1003
21   - brunsli=0.1=h2531618_0
22   - bzip2=1.0.8=h7b6447c_0
23   - c-ares=1.18.1=h7f8727e_0
24   - ca-certificates=2022.07.19=h06a4308_0
25   - certifi=2022.9.14=py39h06a4308_0
26   - cffi=1.15.1=py39h74dc2b5_0
27   - cfitsio=3.470=h5893167_7
28   - charls=2.2.0=h2531618_0
29   - charset-normalizer=2.0.4=pyhd3eb1b0_0
30   - click=8.0.4=py39h06a4308_0
31   - cloudpickle=2.0.0=pyhd3eb1b0_0
32   - cryptography=37.0.1=py39h9ce1e76_0
33   - cycler=0.11.0=pyhd3eb1b0_0
34   - cytoolz=0.11.0=py39h27cfd23_0
35   - dask-core=2022.7.0=py39h06a4308_0
36   - dbus=1.13.18=hb2f20db_0
37   - debugpy=1.5.1=py39h295c915_0
38   - decorator=5.1.1=pyhd3eb1b0_0
39   - defusedxml=0.7.1=pyhd3eb1b0_0
40   - entrypoints=0.4=py39h06a4308_0
41   - executing=0.8.3=pyhd3eb1b0_0
42   - expat=2.4.4=h295c915_0
43   - fftw=3.3.9=h27cfd23_1
44   - fontconfig=2.13.1=h6c09931_0
45   - fonttools=4.25.0=pyhd3eb1b0_0
46   - freetype=2.11.0=h70c0345_0
47   - fsspec=2022.7.1=py39h06a4308_0
48   - giflib=5.2.1=h7b6447c_0
49   - glib=2.69.1=h4ff587b_1
50   - gmp=6.2.1=h295c915_3
51   - gmpy2=2.1.2=py39heeb90bb_0
52   - gst-plugins-base=1.14.0=h8213a91_2
53   - gstreamer=1.14.0=h28cd5cc_2
54   - icu=58.2=he6710b0_3
55   - idna=3.3=pyhd3eb1b0_0
56   - imagecodecs=2021.8.26=py39hf0132c2_1
57   - imageio=2.19.3=py39h06a4308_0
58   - intel-openmp=2021.4.0=h06a4308_3561
59   - ipykernel=6.15.2=py39h06a4308_0
60   - ipython=8.4.0=py39h06a4308_0
61   - ipython_genutils=0.2.0=pyhd3eb1b0_1
62   - jedi=0.18.1=py39h06a4308_1
63   - jinja2=3.0.3=pyhd3eb1b0_0
64   - joblib=1.1.0=pyhd3eb1b0_0
65   - jpeg=9e=h7f8727e_0
66   - json5=0.9.6=pyhd3eb1b0_0
67   - jsonschema=4.4.0=py39h06a4308_0
68   - jupyter_client=7.3.5=py39h06a4308_0
69   - jupyter_core=4.10.0=py39h06a4308_0
70   - jupyter_server=1.18.1=py39h06a4308_0
71   - jupyterlab=3.4.4=py39h06a4308_0
72   - jupyterlab_pygments=0.1.2=py_0
73   - jupyterlab_server=2.12.0=py39h06a4308_0
74   - jxrllib=1.1=h7b6447c_2
75   - kiwisolver=1.4.2=py39h295c915_0
76   - krb5=1.19.2=hac12032_0
77   - lcms2=2.12=h3be6417_0
78   - ld_impl_linux-64=2.38=h1181459_1
79   - lerc=3.0=h295c915_0
80   - libaec=1.0.4=he6710b0_1
81   - libbrotlicommon=1.0.9=h5eee18b_7
82   - libbrotlidec=1.0.9=h5eee18b_7
83   - libbrotlienc=1.0.9=h5eee18b_7
84   - libclang=10.0.1=default_hb85057a_2
85   - libcurl=7.84.0=h91b91d3_0
86   - libdeflate=1.8=h7f8727e_5
87   - libedit=3.1.20210910=h7f8727e_0
88   - libev=4.33=h7f8727e_1
89   - libevent=2.1.12=h8f2d780_0
90   - libffi=3.3=he6710b0_2
91   - libgcc-ng=11.2.0=h1234567_1
92   - libgfortran-ng=11.2.0=h00389a5_1
93   - libgfortran5=11.2.0=h1234567_1
94   - libgomp=11.2.0=h1234567_1
95   - libllvm10=10.0.1=hbc73fb_5
96   - libnghttp2=1.46.0=hce63b2e_0

```

```
97 - libpng=1.6.37=hbc83047_0
98 - libpq=12.9=h16c4e8d_3
99 - libsodium=1.0.18=h7b6447c_0
100 - libssh2=1.10.0=h8f2d780_0
101 - libstdcxx-ng=11.2.0=h1234567_1
102 - libtiff=4.4.0=hecacb30_0
103 - libuuid=1.0.3=h7f8727e_2
104 - libwebp=1.2.2=h55f646e_0
105 - libwebp-base=1.2.2=h7f8727e_0
106 - libxcb=1.15=h7f8727e_0
107 - libxkbcommon=1.0.1=hfa300c1_0
108 - libxml2=2.9.14=h74e7548_0
109 - libxslt=1.1.35=h4e12654_0
110 - libzopfli=1.0.3=he6710b0_0
111 - locket=1.0.0=py39h06a4308_0
112 - lxml=4.9.1=py39h1edc446_0
113 - lz4-c=1.9.3=h295c915_1
114 - markupsafe=2.1.1=py39h7f8727e_0
115 - matplotlib=3.4.3=py39h06a4308_0
116 - matplotlib-base=3.4.3=py39hbbc1b5f_0
117 - matplotlib-inline=0.1.6=py39h06a4308_0
118 - mistune=0.8.4=py39h27cfd23_1000
119 - mkl=2021.4.0=h06a4308_640
120 - mkl-service=2.4.0=py39h7f8727e_0
121 - mkl_fft=1.3.1=py39hd3c417c_0
122 - mkl_random=1.2.2=py39h51133e4_0
123 - mpc=1.1.0=h10f8cd9_1
124 - mpfr=4.0.2=hb69a4c5_1
125 - mpmath=1.2.1=py39h06a4308_0
126 - munkres=1.1.4=py_0
127 - nbclassic=0.3.5=pyhd3eb1b0_0
128 - nbclient=0.5.13=py39h06a4308_0
129 - nbconvert=6.4.4=py39h06a4308_0
130 - nbformat=5.5.0=py39h06a4308_0
131 - ncurses=6.3=h5eee18b_3
132 - nest-asyncio=1.5.5=py39h06a4308_0
133 - networkx=2.8.4=py39h06a4308_0
134 - notebook=6.4.12=py39h06a4308_0
135 - nspr=4.33=h295c915_0
136 - nss=3.74=h0370c37_0
137 - numpy-base=1.22.3=py39hf524024_0
138 - openjpeg=2.4.0=h3ad879b_0
139 - openssl=1.1.1q=h7f8727e_0
140 - packaging=21.3=pyhd3eb1b0_0
141 - pandocfilters=1.5.0=pyhd3eb1b0_0
142 - parso=0.8.3=pyhd3eb1b0_0
143 - partd=1.2.0=pyhd3eb1b0_1
144 - pcre=8.45=h295c915_0
145 - pexpect=4.8.0=pyhd3eb1b0_3
146 - pickleshare=0.7.5=pyhd3eb1b0_1003
147 - pillow=9.2.0=py39hace64e9_1
148 - pip=22.1.2=py39h06a4308_0
149 - ply=3.11=py39h06a4308_0
150 - prometheus_client=0.14.1=py39h06a4308_0
151 - prompt-toolkit=3.0.20=pyhd3eb1b0_0
152 - psutil=5.9.0=py39h5eee18b_0
153 - ptyprocess=0.7.0=pyhd3eb1b0_2
154 - pure_eval=0.2.2=pyhd3eb1b0_0
155 - pycparser=2.21=pyhd3eb1b0_0
156 - pygments=2.11.2=pyhd3eb1b0_0
157 - pyopenssl=22.0.0=pyhd3eb1b0_0
158 - pyparsing=3.0.9=py39h06a4308_0
159 - pyqt=5.15.7=py39h6a678d5_1
160 - pyqt5-sip=12.11.0=py39h6a678d5_1
161 - pyrsistent=0.18.0=py39heee7806_0
162 - pysocks=1.7.1=py39h06a4308_0
163 - python=3.9.12=h12debd9_1
164 - python-dateutil=2.8.2=pyhd3eb1b0_0
165 - python-fastjsonschema=2.16.2=py39h06a4308_0
166 - pytz=2022.1=py39h06a4308_0
167 - pywavelets=1.3.0=py39h7f8727e_0
168 - pyyaml=6.0=py39h7f8727e_1
169 - pyzmq=23.2.0=py39h6a678d5_0
170 - qt-main=5.15.2=h327a75a_7
171 - qt-webengine=5.15.9=hd2b0992_4
172 - qtwebkit=5.212=h4eab89a_4
173 - readline=8.1.2=h7f8727e_1
174 - requests=2.28.1=py39h06a4308_0
175 - scikit-image=0.19.2=py39h51133e4_0
176 - scikit-learn=1.0.2=py39h51133e4_1
177 - scipy=1.7.3=py39h6c91a56_2
178 - send2trash=1.8.0=pyhd3eb1b0_1
179 - setuptools=63.4.1=py39h06a4308_0
180 - sip=6.6.2=py39h6a678d5_0
181 - six=1.16.0=pyhd3eb1b0_1
182 - snappy=1.1.9=h295c915_0
183 - sniffio=1.2.0=py39h06a4308_1
184 - soupsieve=2.3.1=pyhd3eb1b0_0
185 - sqlite=3.39.2=h5082296_0
186 - stack_data=0.2.0=pyhd3eb1b0_0
187 - sympy=1.10.1=py39h06a4308_0
188 - termcolor=1.1.0=py39h06a4308_1
189 - terminado=0.13.1=py39h06a4308_0
190 - testpath=0.6.0=py39h06a4308_0
191 - threadpoolctl=2.2.0=pyh0d69192_0
192 - tifffile=2021.7.2=pyhd3eb1b0_2
193 - tk=8.6.12=h1ccaba5_0
194 - toml=0.10.2=pyhd3eb1b0_0
195 - toolz=0.11.2=pyhd3eb1b0_0
196 - tornado=6.2=py39h5eee18b_0
197 - tqdm=4.64.0=py39h06a4308_0
198 - traitlets=5.1.1=pyhd3eb1b0_0
199 - typing-extensions=4.3.0=py39h06a4308_0
200 - typing_extensions=4.3.0=py39h06a4308_0
201 - tzdata=2022c=h04d1e81_0
202 - urllib3=1.26.11=py39h06a4308_0
203 - wcwidth=0.2.5=pyhd3eb1b0_0
204 - webencodings=0.5.1=py39h06a4308_1
205 - websocket-client=0.58.0=py39h06a4308_4
206 - wheel=0.37.1=pyhd3eb1b0_0
207 - xz=5.2.6=h5eee18b_0
208 - yaml=0.2.5=h7b6447c_0
209 - zeromq=4.3.4=h2531618_0
210 - zfp=0.5.5=h295c915_6
211 - zlib=1.2.12=h5eee18b_3
212 - zstd=1.5.2=ha4553b6_0
213 - pip:
214 -   importlib-metadata==4.12.0
215 -   markdown==3.4.1
216 -   numpy==1.21.4
217 -   opencv-contrib-python-headless==4.6.0.66
218 -   opencv-python-headless==4.6.0.66
```

```
219         - pyclustering==0.10.1.2
220         - tabulate==0.8.10
221         - zipp==3.8.1
222     prefix: /home/xiang71/miniconda3/envs/ece66100-cpu
```

A.5. Source Listing of code/08.ipynb

```
In 1: %load_ext autoreload
      %autoreload 2
      # %import common
      from common import *
      plt.rcParams['figure.dpi'] = 180
      # from common import save_jpeg_svg
      %import helper
      # from helper import *
```

```
In 2: import os
      from tabulate import tabulate
      from pathlib import Path
      import sympy as sp
      from scipy.optimize import minimize
      import numpy as np
      import scipy.signal
      import skimage, skimage.io, skimage.filters, skimage.transform, skimage.color, skimage.feature, skimage.measure, skimage.morphology
      import subprocess
      import sympy as sp

      figure_output_dir = os.path.abspath(os.path.join(os.getcwd(), '..', 'figures'))
      figure_output_dir = Path(figure_output_dir)
      data_dir = os.path.abspath(os.path.join(os.getcwd(), '..', 'data'))
      data_dir = Path(data_dir)
      print(figure_output_dir, data_dir)
```

Estimating the homographies from dataset

```
In 3: dataset1_dir = os.path.join(data_dir, 'Dataset1')
      dataset1_images = []
      for fn in os.listdir(dataset1_dir):
          if fn.endswith('.jpg'):
              dataset1_images.append(os.path.join(dataset1_dir, fn))
      dataset1_images.sort(key=lambda x: int(os.path.splitext(os.path.split(x)[1])[0].split('_')[1]))
```

```
In 4: %import func_impl
      from func_impl import p_to_hc, p_to_phy, solve_linear_correspondence
      from func_impl import CornerDetection, get_default_point_grid
      from helper import table_to_latex, latex_2d_array, latex_e_notation
```

```
In 5: key_to_show = 'img_thres1'
      kwargs = dict(interpolation='none', cmap='gray')
      nrows, ncols = 8, 5
      fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

      for i in range(nrows):
          for j in range(ncols):
              ax = axs[i, j]
              img_fn = dataset1_images[i * ncols + j]
              ret = CornerDetection()(img_fn)
              ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
              ax.imshow(ret[key_to_show], **kwargs)
              ax.axis('off')

      plt.tight_layout()
      save_jpeg_svg(figure_output_dir/'corner_detection_thres1.svg', bbox_inches='tight')
```

```
In 6: key_to_show = 'img_thres2'
      kwargs = dict(interpolation='none', cmap='gray')
      nrows, ncols = 8, 5
      fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

      for i in range(nrows):
          for j in range(ncols):
              ax = axs[i, j]
              img_fn = dataset1_images[i * ncols + j]
              ret = CornerDetection()(img_fn)
              ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
              ax.imshow(ret[key_to_show], **kwargs)
              ax.axis('off')

      save_jpeg_svg(figure_output_dir/'corner_detection_thres2.svg', bbox_inches='tight')
```

```
In 7: key_to_show = 'img_gray_masked'
      kwargs = dict(cmap='gray')
      nrows, ncols = 8, 5
      fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

      for i in range(nrows):
          for j in range(ncols):
              ax = axs[i, j]
              img_fn = dataset1_images[i * ncols + j]
              ret = CornerDetection()(img_fn)
              ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
              ax.imshow(ret[key_to_show], **kwargs)
              ax.axis('off')

      save_jpeg_svg(figure_output_dir/'corner_detection_masked.svg', bbox_inches='tight')
```

```
In 8: key_to_show = 'img_canny'
      kwargs = dict(interpolation='none', cmap='gray')
      nrows, ncols = 8, 5
      fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))
```

```
for i in range(nrows):
    for j in range(ncols):
        ax = axs[i, j]
        img_fn = dataset1_images[i * ncols + j]
        ret = CornerDetection()(img_fn)
        ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
        ax.imshow(ret[key_to_show], **kwargs)
        ax.axis('off')

save_jpeg_svg(figure_output_dir/'corner_detection_canny.svg', bbox_inches='tight')
```

In :

```
nrows, ncols = 8, 5
fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

for i in range(nrows):
    for j in range(ncols):
        ax = axs[i, j]
        img_fn = dataset1_images[i * ncols + j]
        ret = CornerDetection()(img_fn)
        ax.imshow(ret['img_gray'], cmap='gray')
        ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
        ax.axis('off')
        for _, angle, dist in zip(*ret['hough_peaks']):
            x0, y0 = dist * np.asarray([np.cos(angle), np.sin(angle)])
            ax.axline((x0, y0), slope=np.tan(angle + np.pi / 2), color='red', lw=0.5)
        ax.set_xlim(0, ret['img_gray'].shape[1])
        ax.set_ylim(ret['img_gray'].shape[0], 0)

save_jpeg_svg(figure_output_dir/'corner_detection_hough_lines.svg', bbox_inches='tight')
```

In :

```
import matplotlib as mpl
import matplotlib.cm as cm

nrows, ncols = 8, 5
fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

for i in range(nrows):
    for j in range(ncols):
        ax = axs[i, j]
        img_fn = dataset1_images[i * ncols + j]
        ret = CornerDetection()(img_fn, True)
        ax.imshow(ret['img_gray'], cmap='gray')
        ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
        ax.axis('off')
        if ret['line_grouping_successful']:
            for lg_id, lg in enumerate([ret['line_group_8'], ret['line_group_10']]):
                for ln_id, ln in enumerate(lg):
                    color = cmaps[lg_id](cmap_norms[lg_id](ln_id))
                    if abs(ln[0]) < 1.0e-8:
                        # this is a h line
                        ax.axhline(-ln[2]/ln[1], color=color, lw=0.5)
                    elif abs(ln[1]) < 1.0e-8:
                        # this is a v line
                        ax.axvline(-ln[2]/ln[0], color=color, lw=0.5)
                    else:
                        x0 = -ln[2]/ln[0]
                        slope = -ln[0]/ln[1]
                        ax.axline((x0, 0), slope=slope, color=color, lw=0.5)
        ax.set_xlim(0, ret['img_gray'].shape[1])
        ax.set_ylim(ret['img_gray'].shape[0], 0)

save_jpeg_svg(figure_output_dir/'corner_detection_hough_lines_grouped.svg', bbox_inches='tight')
```

In :

```
nrows, ncols = 8, 5
for i in range(nrows):
    for j in range(ncols):
        print(i, j)
        img_fn = dataset1_images[i * ncols + j]
        ret = CornerDetection()(img_fn, True)
        if ret['line_grouping_successful']:
            print(ret['pattern_homography'])
            #print(i, j, ret['found_line_groups'])
```

Estimating the image of the absolute conic

In :

```
from IPython.display import Latex, display

def disp(expr, latex=False):
    l = '${}$'.format(sp.latex(expr))
    display(Latex(l))
    if latex:
        print(sp.latex(expr))
```

In :

```
# build a symmetric version of w with 6 dof
w_elem = sp.symbols('w_{11} w_{12} w_{13} w_{22} w_{23} w_{33}')
w_mat = sp.Matrix([
    [w_elem[0], w_elem[1], w_elem[2]],
    [w_elem[1], w_elem[3], w_elem[4]],
    [w_elem[2], w_elem[4], w_elem[5]]
])

w_mat_lam = sp.lambdify(w_elem, w_mat, dummify=True)

print(w_elem)
disp(sp.Matrix(w_elem), True)
disp(w_mat, True)
```

```

# build the matrix H
h_mat = [[] for _ in range(3)]
for i in range(3):
    for j in range(3):
        h_mat[i].append(sp.symbols('h_{%d%d}' % (i+1, j+1)))
h_mat = sp.Matrix(h_mat)
h_symbols = h_mat.reshape(9, 1)
h_symbols = [h_symbols[i] for i in range(9)]

disp(h_mat)
print(h_symbols)

# we need to compute the equation
h1_vec = h_mat[:, 0]
h2_vec = h_mat[:, 1]

eqn1 = (h1_vec.T * w_mat * h1_vec - h2_vec.T * w_mat * h2_vec)[0]
eqn2 = (h1_vec.T * w_mat * h2_vec)[0]

# extract coefficients with respect to w's
w_terms = []
for eqn in [eqn1, eqn2]:
    terms = []
    for w_term in w_elem:
        coef = eqn.expand().coeff(w_term)
        terms.append(coef)
    w_terms.append(terms)

w_left_v1 = sp.Matrix(w_terms[0])
w_left_v2 = sp.Matrix(w_terms[1])

# use SymPy to generate numeric expressions
w_left_v1_lam = sp.lambdify(h_symbols, w_left_v1, dummify=True)
w_left_v2_lam = sp.lambdify(h_symbols, w_left_v2, dummify=True)

disp(w_left_v1, True)
disp(w_left_v2, True)

def estimate_w(pattern_homographies):
    lhs = []
    for homography in pattern_homographies:
        hlist = homography.flatten().tolist()
        lhs.append(np.asarray(w_left_v1_lam+hlist))
        lhs.append(np.asarray(w_left_v2_lam+hlist))

    lhs = np.asarray(lhs, dtype=np.float64).squeeze()
    u,d,vh = np.linalg.svd(lhs)
    w = vh[-1, :] # use the last column of the V matrix as the solution
    return w

```

Estimating intrinsic parameters

In [□](#):

```

# w_elem = sp.symbols('w_{11} w_{12} w_{13} w_{22} w_{23} w_{33}')
w11, w12, w13, w22, w23, w33 = w_elem
x0, y0, alphax, alphay, s, lam = sp.symbols('x_0 y_0 \alpha_x \alpha_y s \lambda')

eqns = dict()
eqns[y0] = (w12 * w13 - w11 * w23)/(w11*w22 - w12**2)
eqns[lam] = w33 - (w13**2 + y0 * (w12*w13 - w11*w23))/(w11)
eqns[alphax] = sp.sqrt(lam/w11)
eqns[alphay] = sp.sqrt(lam * w11/(w11*w22 - w12**2))
eqns[s] = -(w12 * alphax**2 * alphay)/lam
eqns[x0] = s*y0/alphay - w13 * alphax**2 / lam

for key, val in eqns.items():
    #display(Latex('${}={}$'.format(sp.latex(key), sp.latex(val))))
    print('${}={}$'.format(sp.latex(key), sp.latex(val)))

def estimate_k(w):
    # estimate x0
    subs = list(zip([w11, w12, w13, w22, w23, w33], w.tolist()))
    estimated_values = dict()
    variable_seq = [y0, lam, alphax, alphay, s, x0]
    for var in variable_seq:
        val = eqns[var].subs(subs)
        subs.append((var, val))
        estimated_values[var] = float(val)

    # build the k matrix
    k = np.zeros((3,3), np.float64)
    k[0,0] = estimated_values[alphax]
    k[0,1] = estimated_values[s]
    k[0,2] = estimated_values[x0]
    k[1,1] = estimated_values[alphay]
    k[1,2] = estimated_values[y0]
    k[2,2] = 1.0

    return k

```

Estimating extrinsic parameters

In [□](#):

```

def estimate_extrinsic(k, h):
    kinv = np.linalg.inv(k)
    r1 = kinv @ h[:, 0]
    xi = 1 / np.linalg.norm(r1)

    r1 = xi * kinv @ h[:, 0]
    r2 = xi * kinv @ h[:, 1]
    r3 = np.cross(r1, r2)
    t = xi * kinv @ h[:, 2]

    R = np.stack([r1,r2,r3], axis=1)
    #print(R, '\n', r1, r2, r3)

    u, d, v = np.linalg.svd(R)
    R = u @ v
    return R, t

```

Reprojection and LM refinement

```
In □:
from scipy.spatial.transform.rotation import Rotation
from scipy.optimize import least_squares

def get_camera_projection_mat(K, R, t):
    Rt = np.concatenate([R, np.expand_dims(t, -1)], axis=1)
    # print(R, t, Rt)
    return K @ Rt

# rz - phi
# ry - theta
# rx - psi
def get_euler_angles_from_r(R):
    phi = theta = psi = None
    if 1 - abs(R[2, 0]) < 1.0e-6:
        phi = 0.0
        if R[2, 0] < 0:
            theta = np.pi / 2
            psi = phi + np.arctan2(R[0, 1], R[0, 2])
        else:
            theta = -np.pi / 2
            psi = -phi + np.arctan2(-R[0, 1], -R[0, 2])
    else:
        theta = -np.arcsin(R[2, 0])
        psi = np.arctan2(R[2, 1]/np.cos(theta), R[2, 2]/np.cos(theta))
        phi = np.arctan2(R[1, 0]/np.cos(theta), R[0, 0]/np.cos(theta))

    rotation_mat = get_r_from_euler_angles(phi, theta, psi)
    diff_mat = np.abs(R - rotation_mat)
    assert(np.all(np.max(R - rotation_mat) < 1.0e-4))

    return phi, theta, psi

def get_r_from_euler_angles(phi, theta, psi):
    return Rotation.from_euler('ZYX', (phi, theta, psi)).as_matrix()

def reproject_pattern(cam_mat):
    default_pattern = get_default_point_grid()
    default_pattern_flat = default_pattern.reshape((-1, 2))
    # add z and w
    zeros = np.zeros((default_pattern_flat.shape[0], 1), np.float64)
    ones = np.ones((default_pattern_flat.shape[0], 1), np.float64)
    default_pattern_flat_hc = np.concatenate([default_pattern_flat, zeros, ones], axis=1)

    projected_pattern_flat_hc = (cam_mat @ default_pattern_flat_hc.T).transpose()
    projected_pattern = p_to_phy(projected_pattern_flat_hc)
    return projected_pattern.reshape(default_pattern.shape)

def reproject_pattern_z(pg, h):
    assert h.shape == (3,3)
    default_pattern = pg
    default_pattern_flat_hc = p_to_hc(default_pattern.reshape((-1, 2)))

    projected_pattern_flat_hc = (h @ default_pattern_flat_hc.T).transpose()
    projected_pattern = p_to_phy(projected_pattern_flat_hc)
    return projected_pattern.reshape(default_pattern.shape)

def lm_build_k(alphax, alphay, x0, y0, s):
    k = np.zeros((3,3), np.float64)
    k[0,0] = alphax
    k[0,1] = s
    k[0,2] = x0
    k[1,1] = alphay
    k[1,2] = y0
    k[2,2] = 1.0
    return k

def lm_get_camera_projection_mat(
    alphax, alphay, x0, y0, s, # parameters in K (intrinsic)
    phi, theta, psi, # parameters in R (extrinsic)
    t1, t2, t3 # parameters in t (extrinsic)
):
    k = lm_build_k(alphax, alphay, x0, y0, s)
    R = get_r_from_euler_angles(phi, theta, psi)
    t = np.asarray([t1,t2,t3])
    return get_camera_projection_mat(k, R, t)

def compute_reprojection_error(pg1, pg2):
    return np.linalg.norm((pg1-pg2).flatten())

def lm_refine(k, R, t, image_point_grid):
    # get Euler angles from R

    alphax = k[0,0]
    alphay = k[1,1]
    x0 = k[0,2]
    y0 = k[1,2]
    s = k[0,1]
    phi, theta, psi = get_euler_angles_from_r(R)
    t1,t2,t3 = t

    def cost_func(args):
        cam_mat = lm_get_camera_projection_mat(*args)
        reprojected = reproject_pattern(cam_mat)
        reproj_error = reprojected - image_point_grid
        return reproj_error.reshape((-1, 2)).flatten()

    result = least_squares(cost_func,
        (alphax, alphay, x0, y0, s, phi, theta, psi, t1, t2, t3),
        method='lm'
    )

    return lm_get_camera_projection_mat(*result.x), result.x
```

Taking radial distortion into consideration

```
In □:
from scipy.optimize import minimize
```

```

def reproject_pattern_radial(k, R, t, k1, k2):
    x0, y0 = k[0,2], k[1,2]

    cam_mat = get_camera_projection_mat(k, R, t)

    default_pattern = get_default_point_grid()
    default_pattern_flat = default_pattern.reshape((-1, 2))
    # add z and w
    zeros = np.zeros((default_pattern_flat.shape[0], 1), np.float64)
    ones = np.ones((default_pattern_flat.shape[0], 1), np.float64)
    default_pattern_flat_hc = np.concatenate([default_pattern_flat, zeros, ones], axis=1)

    projected_pattern_flat_hc = (cam_mat @ default_pattern_flat_hc.T).transpose()
    projected_pattern = p_to_phy(projected_pattern_flat_hc)

    # compute r for each position
    pp_diff = projected_pattern - np.asarray([x0,y0])
    r = np.sum(np.power(pp_diff, 2), axis=1)
    scale = np.expand_dims(k1 * r + k2 * r ** 2, -1)
    pp_radial = projected_pattern + pp_diff * scale

    return pp_radial.reshape(default_pattern.shape)

def reproject_pattern_radial_z(pg_src, k_src, k_tgt, k1_src, k2_src, k1_tgt, k2_tgt, cam_src, cam_tgt):
    h_src = np.stack([cam_src[:, 0], cam_src[:, 1], cam_src[:, 3]], axis=1)
    h_tgt = np.stack([cam_tgt[:, 0], cam_tgt[:, 1], cam_tgt[:, 3]], axis=1)

    pg_src_flat = pg_src.reshape((-1, 2))
    # 1st step: remove radial distortion

    # compute r for each position
    x0, y0 = k_src[0,2], k_src[1,2]

    # use optimization to compute the pixel location before radial distortion
    def xy_rad(x, y):
        r2 = (x - x0) ** 2 + (y - y0) ** 2
        scale = k1_src * r2 + k2_src * r2 ** 2
        x_rad = x + (x - x0) * scale
        y_rad = y + (y - y0) * scale
        return np.asarray([x_rad, y_rad])

    pg_linear = []
    for i in range(pg_src_flat.shape[0]):
        p0 = pg_src_flat[i]
        # define a loss function
        def loss(args):
            x, y = args
            p_rad = xy_rad(x, y)
            return np.linalg.norm(p_rad - p0)

        res = minimize(loss, p0)
        pg_linear.append(res.x)

    pg_linear = p_to_hc(np.asarray(pg_linear))

    # pg_diff = pg_src_flat - np.asarray([x0,y0])
    # r = np.sum(np.power(pg_diff, 2), axis=1)
    # scale = np.expand_dims(k1_src * r + k2_src * r ** 2, -1)
    # use subtraction to correct the distortion
    # pg_linear = pg_src_flat - pg_diff * scale

    # 2nd step: project back to the target pattern
    h_combined = h_tgt @ np.linalg.inv(h_src)
    projected_pattern_flat_hc = (h_combined @ pg_linear.T).transpose()
    projected_pattern = p_to_phy(projected_pattern_flat_hc)

    # 3rd step: remove radial distortion
    # compute r for each position
    x0, y0 = k_tgt[0,2], k_tgt[1,2]
    pp_diff = projected_pattern - np.asarray([x0,y0])
    r = np.sum(np.power(pp_diff, 2), axis=1)
    scale = np.expand_dims(k1_tgt * r + k2_tgt * r ** 2, -1)
    pp_radial = projected_pattern + pp_diff * scale

    return pp_radial.reshape(pg_src.shape)

def lm_get_camera_projection_mat_radial(
    alphax, alphay, x0, y0, s, # parameters in K (intrinsic)
    phi, theta, psi, # parameters in R (extrinsic)
    t1, t2, t3 # parameters in t (extrinsic)
):
    k = lm_build_k(alphax, alphay, x0, y0, s)
    R = get_r_from_euler_angles(phi, theta, psi)
    t = np.asarray([t1,t2,t3])
    return k, R, t

def lm_refine_radial(k, R, t, image_point_grid):
    # get Euler angles from R

    alphax = k[0,0]
    alphay = k[1,1]
    x0 = k[0,2]
    y0 = k[1,2]
    s = k[0,1]
    phi, theta, psi = get_euler_angles_from_r(R)
    t1,t2,t3 = t
    k1 = k2 = 0.0

    def cost_func(args):
        k, R, t = lm_get_camera_projection_mat_radial(*args[:-2])
        reprojected = reproject_pattern_radial(k, R, t, args[-2], args[-1])
        reproj_error = reprojected - image_point_grid
        return reproj_error.reshape((-1, 2)).flatten()

    result = least_squares(cost_func,
        (alphax, alphay, x0, y0, s, phi, theta, psi, t1, t2, t3, k1, k2),
        method='lm'
    )

    return lm_build_k(*result.x[:5]), \
        get_r_from_euler_angles(*result.x[5:8]), \
        result.x[8:11], \
        result.x[11:]

```

Homework dataset analysis

```
In □:
pattern_homographies = []

for i in range(len(dataset1_images)):
    img_fn = dataset1_images[i]
    ret = CornerDetection()(img_fn, True)
    if ret['line_grouping_successful']:
        pattern_homographies.append(ret['pattern_homography'])

w = estimate_w(pattern_homographies)
k = estimate_k(w)

In □:
# generate statistics for the entire dataset
table_rows = [
    ['Image', 'Linear', 'LM-Refined', 'Radial Distortion Corrected']
]
for img_fn in dataset1_images:
    ret = CornerDetection()(img_fn, True)
    if ret['line_grouping_successful']:
        row = []
        R, t = estimate_extrinsic(k, ret['pattern_homography'])
        cam_mat = get_camera_projection_mat(k, R, t)
        refined_cam_mat, _ = lm_refine(k, R, t, ret['point_grid'])
        rk, rR, rt, (k1, k2) = lm_refine_radial(k, R, t, ret['point_grid'])
        row.append('\texttt{%s}' % os.path.split(img_fn)[1])
        row.append(compute_reprojection_error(reproject_pattern(cam_mat), ret['point_grid']))
        row.append(compute_reprojection_error(reproject_pattern(refined_cam_mat), ret['point_grid']))
        row.append(compute_reprojection_error(reproject_pattern_radial(rk, rR, rt, k1, k2), ret['point_grid']))

        row[0] = row[0].replace('_', '\\_')
        for i in range(len(row)):
            if isinstance(row[i], float):
                row[i] = '{:.3f}'.format(row[i])

        table_rows.append(row)

In □:
print(table_to_latex(table_rows))

In □:
# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

img_fn = dataset1_images[0]
print(img_fn)

fig, axs = plt.subplots(2, 2)

latex_strings = []

# now, plot the image and lines
ax = axs[0,0]
ret = CornerDetection()(img_fn, True)
ax.imshow(ret['img'])
ax.set_title('Input Image and Detected Lines')
ax.axis('off')
if ret['line_grouping_successful']:
    for lg_id, lg in enumerate([ret['line_group_8'], ret['line_group_10']]):
        for ln_id, ln in enumerate(lg):
            color = cmaps[lg_id](cmap_norms[lg_id](ln_id))
            if abs(ln[0]) < 1.0e-8:
                # this is a h line
                ax.axhline(-ln[2]/ln[1], color=color, lw=0.5)
            elif abs(ln[1]) < 1.0e-8:
                # this is a v line
                ax.axvline(-ln[2]/ln[0], color=color, lw=0.5)
            else:
                x0 = -ln[2]/ln[0]
                slope = -ln[0]/ln[1]
                ax.axline(x0, 0, slope=slope, color=color, lw=0.5)
ax.set_xlim(0, ret['img'].shape[1])
ax.set_ylim(ret['img'].shape[0], 0)

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
empty_image = np.ones((ret['img'].shape[0], ret['img'].shape[1], 3), np.float32)
ax.imshow(empty_image)
ax.set_title('Reprojection (Linear)')
#ax.axis('off')
if ret['line_grouping_successful']:
    pg = ret['point_grid']
    for i in range(pg.shape[0]):
        for j in range(pg.shape[1]):
            ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

# reproject the points
R, t = estimate_extrinsic(k, ret['pattern_homography'])
cam_mat = get_camera_projection_mat(k, R, t)

pg = reproject_pattern(cam_mat)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5,
            label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{}}}}'.format(i + pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\\epsilon={:.3f}\$'.format(compute_reprojection_error(pg, ret['point_grid'])))

# do refinement
```

```

refined_cam_mat, lm_x = lm_refine(k, R, t, ret['point_grid'])

ax = axs[1,0]
ax.imshow(empty_image)
ax.set_title('Reprojection (LM-Refined)')
pg = ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5, label='Input')

pg = reproject_pattern(refined_cam_mat)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{\\{\\}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon=\\{:.3f\\}$'.format(compute_reprojection_error(pg, ret['point_grid'])))

rk, rR, rt, (k1, k2) = lm_refine_radial(k, R, t, ret['point_grid'])

ax = axs[1,1]
ax.imshow(empty_image)
ax.set_title('Reprojection (Radial Correct.)')
pg = ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5, label='Input')

pg = reproject_pattern_radial(rk, rR, rt, k1, k2)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{\\{\\}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon=\\{:.3f\\}$'.format(compute_reprojection_error(pg, ret['point_grid'])))

latex_strings.append('\\symbfit{K}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\\symbfit{R}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(R))
latex_strings.append('\\symbfit{t}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(t.reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_x[:5])))
latex_strings.append('\\symbfit{R}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_x[5:8])))
latex_strings.append('\\symbfit{t}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(lm_x[8:11].reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rk))
latex_strings.append('\\symbfit{R}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rR))
latex_strings.append('\\symbfit{t}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(rt.reshape(1, 3)))
latex_strings.append('k_16=%s' % latex_e_notation('{:.5e}'.format(k1)))
latex_strings.append('k_26=%s' % latex_e_notation('{:.5e}'.format(k2)))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

save_jpeg_svg(figure_output_dir/'sample1.svg', bbox_inches='tight')
print('\\\\\\n'.join(latex_strings))

```

In :

```

# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

img_fn = dataset1_images[14]
print(img_fn)

fig, axs = plt.subplots(2, 2)

latex_strings = []

# now, plot the image and lines
ax = axs[0,0]
ret = CornerDetection()(img_fn, True)
ax.imshow(ret['img'])
ax.set_title('Input Image and Detected Lines')
ax.axis('off')
if ret['line_grouping_successful']:
    for lg_id, lg in enumerate([ret['line_group_8'], ret['line_group_10']]):
        for ln_id, ln in enumerate(lg):
            color = cmaps[lg_id](cmap_norms[lg_id](ln_id))
            if abs(ln[0]) < 1.0e-8:
                # this is a h line
                ax.axhline(-ln[2]/ln[1], color=color, lw=0.5)
            elif abs(ln[1]) < 1.0e-8:
                # this is a v line
                ax.axvline(-ln[2]/ln[0], color=color, lw=0.5)
            else:
                x0 = -ln[2]/ln[0]
                slope = -ln[0]/ln[1]
                ax.axline((x0, 0), slope=slope, color=color, lw=0.5)
ax.set_xlim(0, ret['img'].shape[1])
ax.set_ylim(ret['img'].shape[0], 0)

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
empty_image = np.ones((ret['img'].shape[0], ret['img'].shape[1], 3), np.float32)
ax.imshow(empty_image)
ax.set_title('Reprojection (Linear)')
#ax.axis('off')
if ret['line_grouping_successful']:
    pg = ret['point_grid']
    for i in range(pg.shape[0]):
        for j in range(pg.shape[1]):
            ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                    label='Input' if i + j == 0 else None)

# reproject the points
R, t = estimate_extrinsic(k, ret['pattern_homography'])
cam_mat = get_camera_projection_mat(k, R, t)

```

```

pg = reproject_pattern(cam_mat)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5,
                label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, ret['point_grid'])))

# do refinement
refined_cam_mat, lm_x = lm_refine(k, R, t, ret['point_grid'])

ax = axs[1,0]
ax.imshow(empty_image)
ax.set_title('Reprojection (LM-Refined)')
pg = ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5, label='Input')

pg = reproject_pattern(refined_cam_mat)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, ret['point_grid'])))

rk, rR, rt, (k1, k2) = lm_refine_radial(k, R, t, ret['point_grid'])

ax = axs[1,1]
ax.imshow(empty_image)
ax.set_title('Reprojection (Radial Correct.)')
pg = ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5, label='Input')

pg = reproject_pattern_radial(rk, rR, rt, k1, k2)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, ret['point_grid'])))

latex_strings.append('\sympfit{K}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\sympfit{R}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(R))
latex_strings.append('\sympfit{t}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(t.reshape(1, 3)))
latex_strings.append('\sympfit{K}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_x[:5])))
latex_strings.append('\sympfit{R}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_x[5:8])))
latex_strings.append('\sympfit{t}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(lm_x[8:11].reshape(1, 3)))
latex_strings.append('\sympfit{K}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rk))
latex_strings.append('\sympfit{R}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rR))
latex_strings.append('\sympfit{t}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(rt.reshape(1, 3)))
latex_strings.append('k_{16}=%s' % latex_e_notation('{:.5e}'.format(k1)))
latex_strings.append('k_{26}=%s' % latex_e_notation('{:.5e}'.format(k2)))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

save_jpeg_svg(f'figure_output_dir/sample2.svg', bbox_inches='tight')
print('\\\\n'.join(latex_strings))

```

Sample-to-sample reprojection

In :

```

# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

source_img_fn = dataset1_images[0]
target_img_fn = dataset1_images[12]
print(source_img_fn, target_img_fn)

fig, axs = plt.subplots(2, 2)

source_ret = CornerDetection()(source_img_fn, True)
target_ret = CornerDetection()(target_img_fn, True)

axs[0,0].imshow(source_ret['img'])
axs[0,0].axis('off')
axs[0,0].set_title('Source View')

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
#empty_image = np.ones((target_ret['img'].shape[0], target_ret['img'].shape[1], 3), np.float32)
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Linear)')
#ax.axis('off')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

# reproject the points
R_src, t_src = estimate_extrinsic(k, source_ret['pattern_homography'])
cam_mat_src = get_camera_projection_mat(k, R_src, t_src)
R_tgt, t_tgt = estimate_extrinsic(k, target_ret['pattern_homography'])
cam_mat_tgt = get_camera_projection_mat(k, R_tgt, t_tgt)

```

```

h_src = np.stack([cam_mat_src[:, 0], cam_mat_src[:, 1], cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([cam_mat_tgt[:, 0], cam_mat_tgt[:, 1], cam_mat_tgt[:, 3]], axis=1)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5,
                label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon={:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

# do refinement
refined_cam_mat_src, lm_src = lm_refine(k, R_src, t_src, source_ret['point_grid'])
refined_cam_mat_tgt, lm_tgt = lm_refine(k, R_tgt, t_tgt, target_ret['point_grid'])
h_src = np.stack([refined_cam_mat_src[:, 0], refined_cam_mat_src[:, 1], refined_cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([refined_cam_mat_tgt[:, 0], refined_cam_mat_tgt[:, 1], refined_cam_mat_tgt[:, 3]], axis=1)

ax = axs[1,0]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (LM)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon={:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

rk_src, rR_src, rt_src, (k1_src, k2_src) = lm_refine_radial(k, R_src, t_src, source_ret['point_grid'])
rk_tgt, rR_tgt, rt_tgt, (k1_tgt, k2_tgt) = lm_refine_radial(k, R_tgt, t_tgt, target_ret['point_grid'])
rcam_src = get_camera_projection_mat(rk_src, rR_src, rt_src)
rcam_tgt = get_camera_projection_mat(rk_tgt, rR_tgt, rt_tgt)

ax = axs[1,1]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Rad. Cor.)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_radial_z(source_ret['point_grid'],
                                rk_src,
                                rk_tgt,
                                k1_src, k2_src,
                                k1_tgt, k2_tgt,
                                rcam_src,
                                rcam_tgt)
)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon={:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

prefix = 'target1'
save_jpeg_svg(figure_output_dir/f'{prefix}.svg', bbox_inches='tight')

latex_strings = []
latex_strings.append('\\symbfit{K}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\\symbfit{R}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(R_src))
latex_strings.append('\\symbfit{t}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(t_src.reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_src[:5])))
latex_strings.append('\\symbfit{R}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_src[5:8])))
latex_strings.append('\\symbfit{t}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(lm_src[8:11].reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rk_src))
latex_strings.append('\\symbfit{R}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rR_src))
latex_strings.append('\\symbfit{t}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(rt_src.reshape(1, 3)))
latex_strings.append('k_1\\delta=\\s' % latex_e_notation('{:.5e}'.format(k1_src)))
latex_strings.append('k_2\\delta=\\s' % latex_e_notation('{:.5e}'.format(k2_src)))
latex_str = '\\\\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_source_param.tex'), 'w') as outfile:
    outfile.write('\\begin{align*}\\n\\s\\n\\end{align*}' % latex_str)

latex_strings = []
latex_strings.append('\\symbfit{K}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\\symbfit{R}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(R_tgt))
latex_strings.append('\\symbfit{t}_{\\mathrm{linear}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(t_tgt.reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_tgt[:5])))
latex_strings.append('\\symbfit{R}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(get_r_from_euler_angles(*lm_tgt[5:8])))
latex_strings.append('\\symbfit{t}_{\\mathrm{LM}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(lm_tgt[8:11].reshape(1, 3)))
latex_strings.append('\\symbfit{K}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rk_tgt))
latex_strings.append('\\symbfit{R}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rR_tgt))
latex_strings.append('\\symbfit{t}_{\\mathrm{radial}}\\delta\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(rt_tgt.reshape(1, 3)))
latex_strings.append('k_1\\delta=\\s' % latex_e_notation('{:.5e}'.format(k1_tgt)))
latex_strings.append('k_2\\delta=\\s' % latex_e_notation('{:.5e}'.format(k2_tgt)))
latex_str = '\\\\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_target_param.tex'), 'w') as outfile:
    outfile.write('\\begin{align*}\\n\\s\\n\\end{align*}' % latex_str)

```

In :

```

# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

```

```

source_img_fn = dataset1_images[14]
target_img_fn = dataset1_images[12]
print(source_img_fn, target_img_fn)

fig, axs = plt.subplots(2, 2)

source_ret = CornerDetection()(source_img_fn, True)
target_ret = CornerDetection()(target_img_fn, True)

axs[0,0].imshow(source_ret['img'])
axs[0,0].axis('off')
axs[0,0].set_title('Source View')

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
#empty_image = np.ones((target_ret['img'].shape[0], target_ret['img'].shape[1], 3), np.float32)
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Linear)')
#ax.axis('off')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

# reproject the points
R_src, t_src = estimate_extrinsic(k, source_ret['pattern_homography'])
cam_mat_src = get_camera_projection_mat(k, R_src, t_src)
R_tgt, t_tgt = estimate_extrinsic(k, target_ret['pattern_homography'])
cam_mat_tgt = get_camera_projection_mat(k, R_tgt, t_tgt)
h_src = np.stack([cam_mat_src[:, 0], cam_mat_src[:, 1], cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([cam_mat_tgt[:, 0], cam_mat_tgt[:, 1], cam_mat_tgt[:, 3]], axis=1)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5,
                label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

# do refinement
refined_cam_mat_src, lm_src = lm_refine(k, R_src, t_src, source_ret['point_grid'])
refined_cam_mat_tgt, lm_tgt = lm_refine(k, R_tgt, t_tgt, target_ret['point_grid'])
h_src = np.stack([refined_cam_mat_src[:, 0], refined_cam_mat_src[:, 1], refined_cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([refined_cam_mat_tgt[:, 0], refined_cam_mat_tgt[:, 1], refined_cam_mat_tgt[:, 3]], axis=1)

ax = axs[1,0]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (LM)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

rk_src, rR_src, rt_src, (k1_src, k2_src) = lm_refine_radial(k, R_src, t_src, source_ret['point_grid'])
rk_tgt, rR_tgt, rt_tgt, (k1_tgt, k2_tgt) = lm_refine_radial(k, R_tgt, t_tgt, target_ret['point_grid'])
rcam_src = get_camera_projection_mat(rk_src, rR_src, rt_src)
rcam_tgt = get_camera_projection_mat(rk_tgt, rR_tgt, rt_tgt)

ax = axs[1,1]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Rad. Cor.)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='green', alpha=0.5,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_radial_z(source_ret['point_grid'],
                                rk_src,
                                rk_tgt,
                                k1_src, k2_src,
                                k1_tgt, k2_tgt,
                                rcam_src,
                                rcam_tgt)
)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=3, color='red', alpha=0.5, label='Reprojected')
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\$\epsilon= {:.3f}\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

prefix = 'target2'
save_jpeg_svg(figure_output_dir/f'{prefix}.svg', bbox_inches='tight')

latex_strings = []
latex_strings.append('\sympfit{K}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\sympfit{R}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(R_src))
latex_strings.append('\sympfit{t}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(t_src.reshape(1, 3)))
latex_strings.append('\sympfit{K}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_src[:5])))
latex_strings.append('\sympfit{R}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_src[5:8])))
latex_strings.append('\sympfit{t}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(lm_src[8:11].reshape(1, 3)))
latex_strings.append('\sympfit{K}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rk_src))
latex_strings.append('\sympfit{R}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rR_src))
latex_strings.append('\sympfit{t}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(rt_src.reshape(1, 3)))
latex_strings.append('k_1\delta=\s' % latex_e_notation('{:.5e}'.format(k1_src)))
latex_strings.append('k_2\delta=\s' % latex_e_notation('{:.5e}'.format(k2_src)))
latex_str = '\\\\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_source_param.tex'), 'w') as outfile:

```

```

        outfile.write('\begin{align*}\n%s\n\end{align*}' % latex_str)
latex_strings = []
latex_strings.append('\symbfit{K}_{\mathrm{linear}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\symbfit{R}_{\mathrm{linear}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(R_tgt))
latex_strings.append('\symbfit{t}_{\mathrm{linear}}\delta=\begin{bmatrix}\n%s\end{bmatrix}^T' % latex_2d_array(t_tgt.reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{LM}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_tgt[:5])))
latex_strings.append('\symbfit{R}_{\mathrm{LM}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_tgt[5:8])))
latex_strings.append('\symbfit{t}_{\mathrm{LM}}\delta=\begin{bmatrix}\n%s\end{bmatrix}^T' % latex_2d_array(lm_tgt[8:11].reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{radial}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(rk_tgt))
latex_strings.append('\symbfit{R}_{\mathrm{radial}}\delta=\begin{bmatrix}\n%s\end{bmatrix}' % latex_2d_array(rR_tgt))
latex_strings.append('\symbfit{t}_{\mathrm{radial}}\delta=\begin{bmatrix}\n%s\end{bmatrix}^T' % latex_2d_array(rt_tgt.reshape(1, 3)))
latex_strings.append('k_1\delta=%s' % latex_e_notation('{:.5e}'.format(k1_tgt)))
latex_strings.append('k_2\delta=%s' % latex_e_notation('{:.5e}'.format(k2_tgt)))
latex_str = '%%\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_target_param.tex'), 'w') as outfile:
    outfile.write('\begin{align*}\n%s\n\end{align*}' % latex_str)
```

Radial distortion example

In :

```
import scipy.ndimage

img = np.zeros((1200, 1200), dtype=np.bool_)

# draw lines in the image

img_center = np.asarray(img.shape) // 2
max_side = np.min(img.shape) - 100

for half_side in np.linspace(10, max_side, 20):
    start = img_center - (half_side, half_side)
    end = img_center + (half_side, half_side)
    rr, cc = skimage.draw.rectangle_perimeter(start=start, end=end, shape=img.shape)
    img[rr,cc] = True

img = skimage.morphology.binary_dilation(img, skimage.morphology.square(4))

# plt.imshow(img, cmap='gray')

def radial_distort(img, k1, k2):
    pix_locations = np.stack(np.where(img), axis=1)

    # compute r
    d = pix_locations - img_center
    d = d.astype(np.float128)
    r = np.sum(np.power(d, 2), axis=1)
    r = k1 * r + k2 * np.power(r, 2)
    pix_locations_new = pix_locations + d * r[:, np.newaxis]
    # pix_locations_new = np.round(pix_locations_new).astype(np.int32)

    new_img = np.zeros_like(img, np.float32)
    for i in range(pix_locations_new.shape[0]):
        loc = pix_locations_new[i]
        if 0 <= loc[0] < img.shape[0] and 0 <= loc[1] < img.shape[1]:
            rounded_loc = np.round(loc).astype(np.int32)
            new_img[rounded_loc[0] - 1 : rounded_loc[0] + 2,
                    rounded_loc[1] - 1 : rounded_loc[1] + 2] = 1.0

    new_img = np.clip(new_img, 0, 1)

    return 1 - new_img

k1_choices = [0.0, -5.0e-8, 5.0e-8]
k2_choices = [0.0, -5.0e-14, 5.0e-14]
k1_strings = [latex_e_notation('{:.1e}'.format(x)) for x in k1_choices]
k2_strings = [latex_e_notation('{:.1e}'.format(x)) for x in k2_choices]
print(', '.join(k1_strings))
print(', '.join(k2_strings))

fig, axs = plt.subplots(3, 3, figsize=(6, 8))
ax_ind = 0
for k1 in k1_choices:
    for k2 in k2_choices:
        ax = axs[divmod(ax_ind, axs.shape[1])]
        ax.imshow(radial_distort(img, k1, k2), cmap='gray')

        ax.axis('off')
        ax.set_title('\x%d\x%d' % (ax_ind // axs.shape[1] + 1, ax_ind % axs.shape[1] + 1))
        ax_ind += 1

plt.tight_layout()
save_jpeg_svg(figure_output_dir/'radial_distortion_example.svg', bbox_inches='tight')
```

Plot point numbering

In :

```
img_fn = dataset1_images[12]
print(img_fn)

ret = CornerDetection()(img_fn, True)

plt.imshow(ret['img'])
pg = ret['point_grid']

p_index = 1
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        plt.plot(pg[i,j,0], pg[i,j,1], 'o', color='red', markersize=3)
        plt.text(pg[i,j,0]-0.5, pg[i,j,1]-5.5, str(p_index), color='red', ha='right')
        p_index += 1

save_jpeg_svg(figure_output_dir/'grid_points.svg', bbox_inches='tight')
```

Custom dataset

In :

```
dataset2_images = []
```

```
for fn in os.listdir(figure_output_dir/'mydataset'):
    if fn.endswith('.JPG'):
        dataset2_images.append(os.path.join(figure_output_dir/'mydataset', fn))
dataset2_images.sort(key=lambda x: int(os.path.splitext(os.path.split(x)[1])[0].split('_')[1]))
```

In :

```
import matplotlib as mpl
import matplotlib.cm as cm

nrows, ncols = 4, 3
fig, axs = plt.subplots(nrows, ncols, figsize=(6, 8.5))

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

for i in range(nrows):
    for j in range(ncols):
        ax = axs[i, j]
        img_fn = dataset2_images[i * ncols + j]
        ret = CornerDetection(eq_hist=True,
                              manual_threshold=12/255,
                              thres1_edge_crop=200,
                              thres2_dilation_width=6,
                              canny_dilation_width=1,
                              hough_min_distance=25
                              )(img_fn, True)
        ax.imshow(ret['img_gray'], cmap='gray')
        ax.set_title(r'\texttt{{{}}}'.format(os.path.split(img_fn)[1]).replace('_', '\\_'), fontsize=4)
        ax.axis('off')
        if ret['line_grouping_successful']:
            for lg_id, lg in enumerate([ret['line_group_8'], ret['line_group_10']]):
                for ln_id, ln in enumerate(lg):
                    color = cmaps[lg_id](cmap_norms[lg_id](ln_id))
                    if abs(ln[0]) < 1.0e-8:
                        # this is a h line
                        ax.axhline(-ln[2]/ln[1], color=color, lw=0.5)
                    elif abs(ln[1]) < 1.0e-8:
                        # this is a v line
                        ax.axvline(-ln[2]/ln[0], color=color, lw=0.5)
                    else:
                        x0 = -ln[2]/ln[0]
                        slope = -ln[0]/ln[1]
                        ax.axline((x0, 0), slope=slope, color=color, lw=0.5)
        ax.set_xlim(0, ret['img_gray'].shape[1])
        ax.set_ylim(ret['img_gray'].shape[0], 0)

save_jpeg_svg(figure_output_dir/'my_dataset_hough_lines_grouped.svg', bbox_inches='tight')
```

In :

```
# estimate intrinsic
pattern_homographies = []

for i in range(len(dataset2_images)):
    img_fn = dataset2_images[i]
    ret = CornerDetection(eq_hist=True,
                          manual_threshold=12/255,
                          thres1_edge_crop=200,
                          thres2_dilation_width=6,
                          canny_dilation_width=1,
                          hough_min_distance=25
                          )(img_fn, True)
    if ret['line_grouping_successful']:
        pattern_homographies.append(ret['pattern_homography'])

w = estimate_w(pattern_homographies)
k = estimate_k(w)
```

In :

```
# generate statistics for the entire dataset
table_rows = [
    ['Image', 'Linear', 'LM-Refined', 'Radial Distortion Corrected']
]
for img_fn in dataset2_images:
    ret = CornerDetection(eq_hist=True,
                          manual_threshold=12/255,
                          thres1_edge_crop=200,
                          thres2_dilation_width=6,
                          canny_dilation_width=1,
                          hough_min_distance=25
                          )(img_fn, True)
    if ret['line_grouping_successful']:
        row = []
        R, t = estimate_extrinsic(k, ret['pattern_homography'])
        cam_mat = get_camera_projection_mat(k, R, t)
        refined_cam_mat, _ = lm_refine(k, R, t, ret['point_grid'])
        rk, rR, rt, (k1, k2) = lm_refine_radial(k, R, t, ret['point_grid'])
        row.append('\texttt{%s}' % os.path.split(img_fn)[1])
        row.append(compute_reprojection_error(reproject_pattern(cam_mat), ret['point_grid']))
        row.append(compute_reprojection_error(reproject_pattern(refined_cam_mat), ret['point_grid']))
        row.append(compute_reprojection_error(reproject_pattern_radial(rk, rR, rt, k1, k2), ret['point_grid']))

        row[0] = row[0].replace('_', '\\_')
        for i in range(len(row)):
            if isinstance(row[i], float):
                row[i] = '{:.3f}'.format(row[i])

        table_rows.append(row)
```

In :

```
print(table_to_latex(table_rows))
```

In :

```
# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]
```

```

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

source_img_fn = dataset2_images[-3]
target_img_fn = dataset2_images[0]
print(source_img_fn, target_img_fn)

fig, axs = plt.subplots(2, 2)

source_ret = CornerDetection(eq_hist=True,
                             manual_threshold=12/255,
                             thres1_edge_crop=200,
                             thres2_dilation_width=6,
                             canny_dilation_width=1,
                             hough_min_distance=25
                             )(source_img_fn, True)
target_ret = CornerDetection(eq_hist=True,
                             manual_threshold=12/255,
                             thres1_edge_crop=200,
                             thres2_dilation_width=6,
                             canny_dilation_width=1,
                             hough_min_distance=25
                             )(target_img_fn, True)

axs[0,0].imshow(source_ret['img'])
axs[0,0].axis('off')
axs[0,0].set_title('Source View')

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
#empty_image = np.ones((target_ret['img'].shape[0], target_ret['img'].shape[1], 3), np.float32)
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Linear)')
#ax.axis('off')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                label='Input' if i + j == 0 else None)

# reproject the points
R_src, t_src = estimate_extrinsic(k, source_ret['pattern_homography'])
cam_mat_src = get_camera_projection_mat(k, R_src, t_src)
R_tgt, t_tgt = estimate_extrinsic(k, target_ret['pattern_homography'])
cam_mat_tgt = get_camera_projection_mat(k, R_tgt, t_tgt)
h_src = np.stack([cam_mat_src[:, 0], cam_mat_src[:, 1], cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([cam_mat_tgt[:, 0], cam_mat_tgt[:, 1], cam_mat_tgt[:, 3]], axis=1)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, markeredgewidth=0.1,
                label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon= {:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

# do refinement
refined_cam_mat_src, lm_src = lm_refine(k, R_src, t_src, source_ret['point_grid'])
refined_cam_mat_tgt, lm_tgt = lm_refine(k, R_tgt, t_tgt, target_ret['point_grid'])
h_src = np.stack([refined_cam_mat_src[:, 0], refined_cam_mat_src[:, 1], refined_cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([refined_cam_mat_tgt[:, 0], refined_cam_mat_tgt[:, 1], refined_cam_mat_tgt[:, 3]], axis=1)

ax = axs[1,0]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (LM)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, label='Reprojected', markeredgewidth=0.1)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon= {:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

rk_src, rR_src, rt_src, (k1_src, k2_src) = lm_refine_radial(k, R_src, t_src, source_ret['point_grid'])
rk_tgt, rR_tgt, rt_tgt, (k1_tgt, k2_tgt) = lm_refine_radial(k, R_tgt, t_tgt, target_ret['point_grid'])
rcam_src = get_camera_projection_mat(rk_src, rR_src, rt_src)
rcam_tgt = get_camera_projection_mat(rk_tgt, rR_tgt, rt_tgt)

ax = axs[1,1]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Rad. Cor.)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_radial_z(source_ret['point_grid'],
                                rk_src,
                                rk_tgt,
                                k1_src, k2_src,
                                k1_tgt, k2_tgt,
                                rcam_src,
                                rcam_tgt
                                )
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, label='Reprojected', markeredgewidth=0.1)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon= {:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

```

```

prefix = 'target3'
save_peg_svg(figure_output_dir/f'{prefix}.svg', bbox_inches='tight')

latex_strings = []
latex_strings.append('\symbfit{K}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\symbfit{R}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(R_src))
latex_strings.append('\symbfit{t}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(t_src.reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_src[:5])))
latex_strings.append('\symbfit{R}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_src[5:8])))
latex_strings.append('\symbfit{t}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(lm_src[8:11].reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rk_src))
latex_strings.append('\symbfit{R}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rR_src))
latex_strings.append('\symbfit{t}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(rt_src.reshape(1, 3)))
latex_strings.append('k_1\delta=%s' % latex_e_notation('{:.5e}'.format(k1_src)))
latex_strings.append('k_2\delta=%s' % latex_e_notation('{:.5e}'.format(k2_src)))
latex_str = '~~~~~\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_source_param.tex'), 'w') as outfile:
    outfile.write('\begin{align*}\n\end{align*}' % latex_str)
latex_strings = []
latex_strings.append('\symbfit{K}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\symbfit{R}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(R_tgt))
latex_strings.append('\symbfit{t}_{\mathrm{linear}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(t_tgt.reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_tgt[:5])))
latex_strings.append('\symbfit{R}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_tgt[5:8])))
latex_strings.append('\symbfit{t}_{\mathrm{LM}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(lm_tgt[8:11].reshape(1, 3)))
latex_strings.append('\symbfit{K}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rk_tgt))
latex_strings.append('\symbfit{R}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}' % latex_2d_array(rR_tgt))
latex_strings.append('\symbfit{t}_{\mathrm{radial}}\delta=\begin{bmatrix}\n\end{bmatrix}^T' % latex_2d_array(rt_tgt.reshape(1, 3)))
latex_strings.append('k_1\delta=%s' % latex_e_notation('{:.5e}'.format(k1_tgt)))
latex_strings.append('k_2\delta=%s' % latex_e_notation('{:.5e}'.format(k2_tgt)))
latex_str = '~~~~~\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_target_param.tex'), 'w') as outfile:
    outfile.write('\begin{align*}\n\end{align*}' % latex_str)

```

In [□](#):

```

# plot samples from the dataset
import matplotlib as mpl
import matplotlib.cm as cm

cmap_norms = [
    mpl.colors.Normalize(-3, 8),
    mpl.colors.Normalize(-3, 10)
]

cmaps = [
    cm.get_cmap('Blues'),
    cm.get_cmap('Reds')
]

source_img_fn = dataset2_images[-2]
target_img_fn = dataset2_images[0]
print(source_img_fn, target_img_fn)

fig, axs = plt.subplots(2, 2)

source_ret = CornerDetection(eq_hist=True,
                             manual_threshold=12/255,
                             thres1_edge_crop=200,
                             thres2_dilation_width=6,
                             canny_dilation_width=1,
                             hough_min_distance=25
                             )(source_img_fn, True)
target_ret = CornerDetection(eq_hist=True,
                             manual_threshold=12/255,
                             thres1_edge_crop=200,
                             thres2_dilation_width=6,
                             canny_dilation_width=1,
                             hough_min_distance=25
                             )(target_img_fn, True)

axs[0,0].imshow(source_ret['img'])
axs[0,0].axis('off')
axs[0,0].set_title('Source View')

# now, plot the location of the points from the image and the reprojected points
ax = axs[0,1]
#empty_image = np.ones((target_ret['img'].shape[0], target_ret['img'].shape[1], 3), np.float32)
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Linear)')
#ax.axis('off')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                label='Input' if i + j == 0 else None)

# reproject the points
R_src, t_src = estimate_extrinsic(k, source_ret['pattern_homography'])
cam_mat_src = get_camera_projection_mat(k, R_src, t_src)
R_tgt, t_tgt = estimate_extrinsic(k, target_ret['pattern_homography'])
cam_mat_tgt = get_camera_projection_mat(k, R_tgt, t_tgt)
h_src = np.stack([cam_mat_src[:, 0], cam_mat_src[:, 1], cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([cam_mat_tgt[:, 0], cam_mat_tgt[:, 1], cam_mat_tgt[:, 3]], axis=1)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, markeredgewidth=0.1,
                label='Reprojected' if i + j == 0 else None)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\s{4}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\epsilon={:3f}\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

# do refinement
refined_cam_mat_src, lm_src = lm_refine(k, R_src, t_src, source_ret['point_grid'])
refined_cam_mat_tgt, lm_tgt = lm_refine(k, R_tgt, t_tgt, target_ret['point_grid'])
h_src = np.stack([refined_cam_mat_src[:, 0], refined_cam_mat_src[:, 1], refined_cam_mat_src[:, 3]], axis=1)
h_tgt = np.stack([refined_cam_mat_tgt[:, 0], refined_cam_mat_tgt[:, 1], refined_cam_mat_tgt[:, 3]], axis=1)

ax = axs[1,0]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (LM)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):

```

```

        for j in range(pg.shape[1]):
            ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                    label='Input' if i + j == 0 else None)

pg = reproject_pattern_z(source_ret['point_grid'], h_tgt @ np.linalg.inv(h_src))
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, label='Reprojected', markeredgewidth=0.1)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon={:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

rk_src, rR_src, rt_src, (k1_src, k2_src) = lm_refine_radial(k, R_src, t_src, source_ret['point_grid'])
rk_tgt, rR_tgt, rt_tgt, (k1_tgt, k2_tgt) = lm_refine_radial(k, R_tgt, t_tgt, target_ret['point_grid'])
rcam_src = get_camera_projection_mat(rk_src, rR_src, rt_src)
rcam_tgt = get_camera_projection_mat(rk_tgt, rR_tgt, rt_tgt)

ax = axs[1,1]
ax.imshow(target_ret['img'])
ax.set_title('Target View Reproj. (Rad. Cor.)')

pg = target_ret['point_grid']
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='green', alpha=0.5, markeredgewidth=0.1,
                label='Input' if i + j == 0 else None)

pg = reproject_pattern_radial_z(source_ret['point_grid'],
                                rk_src,
                                rk_tgt,
                                k1_src, k2_src,
                                k1_tgt, k2_tgt,
                                rcam_src,
                                rcam_tgt
)
for i in range(pg.shape[0]):
    for j in range(pg.shape[1]):
        ax.plot(pg[i,j,0], pg[i,j,1], 'x', markersize=1, color='red', alpha=0.5, label='Reprojected', markeredgewidth=0.1)
        ax.text(pg[i,j,0], pg[i,j,1]-5, '\\s{{{}}}'.format(i * pg.shape[1] + j + 1), fontsize=3, color='red', alpha=0.5, ha='center')

ax.set_xlabel(r'\\$\\epsilon={:.3f}\\$'.format(compute_reprojection_error(pg, target_ret['point_grid'])))

axs[0,1].legend(bbox_to_anchor=(1.02, 0.8))
plt.tight_layout()

prefix = 'target4'
save_jpeg_svg(f'{figure_output_dir}/f'{prefix}.svg', bbox_inches='tight')

latex_strings = []
latex_strings.append('\\sympfit{K}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\\sympfit{R}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(R_src))
latex_strings.append('\\sympfit{t}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(t_src.reshape(1, 3)))
latex_strings.append('\\sympfit{K}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_src[:5])))
latex_strings.append('\\sympfit{R}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_src[5:8])))
latex_strings.append('\\sympfit{t}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(lm_src[8:11].reshape(1, 3)))
latex_strings.append('\\sympfit{K}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rk_src))
latex_strings.append('\\sympfit{R}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rR_src))
latex_strings.append('\\sympfit{t}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(rt_src.reshape(1, 3)))
latex_strings.append('k_1\\delta=%s' % latex_e_notation('{:.5e}'.format(k1_src)))
latex_strings.append('k_2\\delta=%s' % latex_e_notation('{:.5e}'.format(k2_src)))
latex_str = '\\\\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_source_param.tex'), 'w') as outfile:
    outfile.write('\\begin{align*}\\n\\s\\n\\end{align*}' % latex_str)

latex_strings = []
latex_strings.append('\\sympfit{K}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(k))
latex_strings.append('\\sympfit{R}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(R_tgt))
latex_strings.append('\\sympfit{t}_{\\mathrm{linear}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(t_tgt.reshape(1, 3)))
latex_strings.append('\\sympfit{K}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(lm_build_k(*lm_tgt[:5])))
latex_strings.append('\\sympfit{R}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(get_r_from_euler_angles(*lm_tgt[5:8])))
latex_strings.append('\\sympfit{t}_{\\mathrm{LM}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(lm_tgt[8:11].reshape(1, 3)))
latex_strings.append('\\sympfit{K}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rk_tgt))
latex_strings.append('\\sympfit{R}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}' % latex_2d_array(rR_tgt))
latex_strings.append('\\sympfit{t}_{\\mathrm{radial}}\\delta=\\begin{bmatrix}\\n\\s\\end{bmatrix}^T' % latex_2d_array(rt_tgt.reshape(1, 3)))
latex_strings.append('k_1\\delta=%s' % latex_e_notation('{:.5e}'.format(k1_tgt)))
latex_strings.append('k_2\\delta=%s' % latex_e_notation('{:.5e}'.format(k2_tgt)))
latex_str = '\\\\n'.join(latex_strings)
with open(os.path.join(figure_output_dir, f'{prefix}_target_param.tex'), 'w') as outfile:
    outfile.write('\\begin{align*}\\n\\s\\n\\end{align*}' % latex_str)

```

In :