ECE 661: Homework 8

Christina Eberhardt (eberharc@purdue.edu)

Part 1: Theory

In Lecture 18, we showed that the image of the Absolute Conic Ω_∞ is given by $\omega = K^{-T}K^{-1}$. As you know, the Absolute Conic resides in the plane π_∞ at infinity. Does the derivation we went through in Lecture 18 mean that you can actually see ω in a camera image? Give reasons for both "yes" and "no" answers. Also, explain in your own words the role played by this result in camera calibration.

The answer to this question depends on how we define "actually seeing". All pixels in the image conic $\boldsymbol{\omega}$ are imaginary and images only plot the real pixels.

Hence, in the traditional sense of seeing things we cannot see $\boldsymbol{\omega}$. The answer would be no. If we define seeing in a broader sense (analogously to how we can "see" the complex resistors) then the answer would be yes.

We always know the coordinates of the circular points **I** and **J**. This result makes it easy to obtain an efficient algorithm for determining the matrix **K** of the camera using Zhang's algorithm.

Part 2: Creating my own dataset

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)
(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)
(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)
(0,8)	(1,8)	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)	(7,8)
(0,9)	(1,9)	(2,9)	(3,9)	(4,9)	(5,9)	(6,9)	(7,9)

Table 1: World coordinates in inch when choosing top most left corner as world origin (0,0)

1	11	21	31	41	51	61	71
2	12	22	32	42	52	62	72
3	13	23	33	43	53	63	73
4	14	24	34	44	54	64	74
5	15	25	35	45	55	65	75
6	16	26	36	46	56	66	76
7	17	27	37	47	57	67	77
8	18	28	38	48	58	68	78
9	19	29	39	49	59	69	79
10	20	30	40	50	60	70	80

Table 2: Labels of the corners

Part 3: Zhang's algorithm

3.1: Corner Detection

Use the canny edge detector from OpenCV to determine the edges in the image. Use the Hough transform from OpenCV to fit straight lines to the edges. We calculate the slope of each line and divide the lines into 2 subgroups: horizontal and vertical.

These operations yield too many lines in the image. Therefore, check if the lines intersect in the image frame and if they do put them into the same group. This way we get multiple groups of lines out of which we only want to keep one line. If the remaining lines are too many we consider the distance between the lines to decide on which lines to discard.

Once we have 10 horizontal and 8 vertical lines, we sort them from top to bottom and left to right respectively. Once they are sorted a nested loop can be used to create the intersection points with fixed labels. We can use these intersection points for the camera calibration.

3.2: Camera Calibration

We establish the correspondence between the extracted corners in each image and their world coordinates by getting their homography. We use the scaling factor 2.54*10 for the world coordinates. This way we get the correspondence 1mm = 1px. A typical choice would be 1cm = 1px but in this case this would yield a very small result. To get the homography we use the Code from Homework 5.

We assume that the calibration pattern is in the z=0 plane of the world frame.

Then we have $\boldsymbol{x}_{image} = \boldsymbol{H} x_{world} = \boldsymbol{H} \begin{pmatrix} x_{world} \\ y_{world} \\ w_{world} \end{pmatrix} = \boldsymbol{K}[\boldsymbol{R}|\boldsymbol{t}] \begin{pmatrix} x \\ y \\ 0 \\ w \end{pmatrix} = \begin{pmatrix} x \\ y \\ w \end{pmatrix}$.

We have already determined $\boldsymbol{H} = [\boldsymbol{h}_1 \quad \boldsymbol{h}_2 \quad \boldsymbol{h}_3]$ by establishing the correspondences between the extracted corners and the world coordinates.

The camera image of the Absolute Conic $\boldsymbol{\Omega}_\infty$ is independent of **R** and **t** and given by $\boldsymbol{\omega} = \boldsymbol{K}^{-T}\boldsymbol{K}^{-1}$. Hence, we can use the image of the Absolute Conic to determine **K**. Any plane in the world frame samples $\boldsymbol{\Omega}_\infty$ at exactly two points. The images of those points fall onto $\boldsymbol{\omega}$ in the camera image plane. This yields the two following equations:

$$\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_1 = \boldsymbol{h}_2^T \boldsymbol{\omega} \boldsymbol{h}_2$$

$$\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_2 = 0$$

We know that $\boldsymbol{\omega} = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix}$ is symmetric. Therefore, if we flatten the matrix and remove duplicate values, we can express it as

$$\boldsymbol{b} = (\omega_{11} \quad \omega_{12} \quad \omega_{22} \quad \omega_{13} \quad \omega_{23} \quad \omega_{33})^T$$

Express $\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_1 - \boldsymbol{h}_2^T \boldsymbol{\omega} \boldsymbol{h}_2 = 0$ and $\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_2 = 0$ in terms of **b**.

Let $\boldsymbol{h}_i^T = (h_{i1} \quad h_{i2} \quad h_{i3})$ and define a vector

$$\boldsymbol{v}_{ij} = \begin{pmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{pmatrix}$$

Where i and j can take on the values 1 and 2. Then,

$$\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_2 = \boldsymbol{v}_{12}^T \boldsymbol{b} = 0$$

$$\boldsymbol{h}_1^T \boldsymbol{\omega} \boldsymbol{h}_1 - \boldsymbol{h}_2^T \boldsymbol{\omega} \boldsymbol{h}_2 = (\boldsymbol{v}_{11} - \boldsymbol{v}_{22})^T \boldsymbol{b} = 0$$

Which we can combine into one equation for one image:

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

This can then be generalized by stacking n such equations for n images.

$$\begin{bmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \\ \vdots \\ \boldsymbol{V}_n \end{bmatrix} \boldsymbol{b} = 0$$

This can be solved using Linear least-squares. In a second step we will apply LM optimization. We can obtain $\boldsymbol{\omega}$ from **b**.

In the next step we calculate the 5 intrinsic parameters of the camera. We have

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

We can calculate the parameters using the following formulas:

$$x_0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2}$$

$$\lambda = \omega_{33} - \frac{\omega_{13}^2 + x_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}}$$

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}}$$

$$\alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}}$$

$$s = -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda}$$

$$y_0 = \frac{sx_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

Now that we have the intrinsic parameters, we can calculate the extrinsic parameters R and t of the camera for a certain position of the camera.

We know that $\boldsymbol{K}^{-1}[\boldsymbol{h}_1 \quad \boldsymbol{h}_2 \quad \boldsymbol{h}_3]$=$[\boldsymbol{r}_1 \quad \boldsymbol{r}_2 \quad \boldsymbol{t}]$. We get the following equations:

$$\boldsymbol{r}_1 = \boldsymbol{K}^{-1}\boldsymbol{h}_1$$

$$\boldsymbol{r}_2 = \boldsymbol{K}^{-1}\boldsymbol{h}_2$$

$$\boldsymbol{r}_3 = \boldsymbol{r}_1 X \boldsymbol{r}_2$$

$$\boldsymbol{t} = \boldsymbol{K}^{-1}\boldsymbol{h}_3$$

With those we are not guaranteed to have an orthonormal **R**. Hence, we introduce a scaling factor

$$\xi = \frac{1}{||\boldsymbol{K}^{-1}\boldsymbol{h}_1||}$$

And obtain the following changed parameters:

$$\boldsymbol{r}_1 = \xi\boldsymbol{K}^{-1}\boldsymbol{h}_1$$

$$\boldsymbol{r}_2 = \xi\boldsymbol{K}^{-1}\boldsymbol{h}_2$$

$$\boldsymbol{r}_3 = \boldsymbol{r}_1 X \boldsymbol{r}_2$$

$$\boldsymbol{t} = \xi \boldsymbol{K}^{-1} \boldsymbol{h}_3$$

We can make sure that R is orthogonal by calculating the SVD of **R** and setting $\boldsymbol{R} = UV^T$ as the new **R**.

We can refine **K**, **R** and **t** using Levenberg-Marquardt optimization.

We use the cost function

$$\begin{aligned} d_{geom}^2 &= \sum_i \sum_j \left\| \boldsymbol{x}_{ij} - \widehat{\boldsymbol{x}}_{ij} \right\|^2 \\ &= \sum_i \sum_j \left\| \boldsymbol{x}_{ij} - \boldsymbol{K}[\boldsymbol{R}_i | \boldsymbol{t}_i] x_{mj} \right\|^2 \\ &= \sum_i \sum_j \left\| \boldsymbol{x}_{ij} - \boldsymbol{K}[\boldsymbol{r}_{i1} \boldsymbol{r}_{i2} \boldsymbol{t}_i] x_{mj} \right\|^2 \\ &= \| \boldsymbol{X} - \boldsymbol{f}(\boldsymbol{p}) \|^2 \end{aligned}$$

Where $\boldsymbol{p} = (\boldsymbol{K}, \boldsymbol{R}_i, \boldsymbol{t}_i | i = 1,2, \dots)^T$

Where $\boldsymbol{x}_{ij}$ is the actual image point for the jth salient point on the calibration pattern $\boldsymbol{x}_{mj}$ and $\widehat{\boldsymbol{x}}_{ij}$ is the projected image point for that salient point using P for the ith camera position. These are the physical coordinates.

R has 3 degrees of freedom but is represented by 9 parameters. For the LM algorithm to be successful we need R to be represented in 3 parameters. We can use the Rodrigues representation to enforce this. The Rodrigues representation expresses a 3D rotation in a vector $\boldsymbol{w} = \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix}$. We can construct **w** from $\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$ with the following formulas:

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

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

Once we get the improved values of **w,** we need to transform them back into the matrix **R**. This can be done with the following equations:

$$\varphi = \|\boldsymbol{w}\|$$

$$[\boldsymbol{w}]_X = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$$

$$\boldsymbol{R} = e^{[\boldsymbol{w}]_X} = \boldsymbol{I}_{3x3} + \frac{\sin(\varphi)}{\varphi}[\boldsymbol{w}]_X + \frac{1-\cos(\varphi)}{\varphi^2}[\boldsymbol{w}]_X{}^2$$

This conversion ensures that **R** is orthonormal.

With the Rodrigues representation **p** changes to

$$\boldsymbol{p} = (\boldsymbol{K}, \boldsymbol{w}_i, \boldsymbol{t}_i | i = 1,2,\ldots)^T$$

With this choice of **p** we do not have more variables than degrees of freedom and LM can succeed.

We can now obtain $\boldsymbol{H}_i = \boldsymbol{K}[\boldsymbol{R}_i|\boldsymbol{t}_i]$ for each image i. Then we use these homographies to project the intersection points of each image into the fixed image and calculate mean and variance w.r.t. the intersection points in the fixed image.

3.3: Extra credit task: radial distortion parameters

When taking into account radial distortion we can do everything the way we did before concerning the preliminary determination of **K** and **[R|t]**. We change p to have two more parameters - k_1 and k_2. These are the parameters that we want to determine from the formulas for radial distortion.

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

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

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

$(\hat{x}, \hat{y})$ is the projection of the intersection point to the world coordinates. x_0 and y_0 are the parameters within **K**.

Refining the projection onto the world coordinates with these formulas in the cost function improves out final estimate for the camera parameters.

Part 4: Observations

Part 4.1: Dataset 1

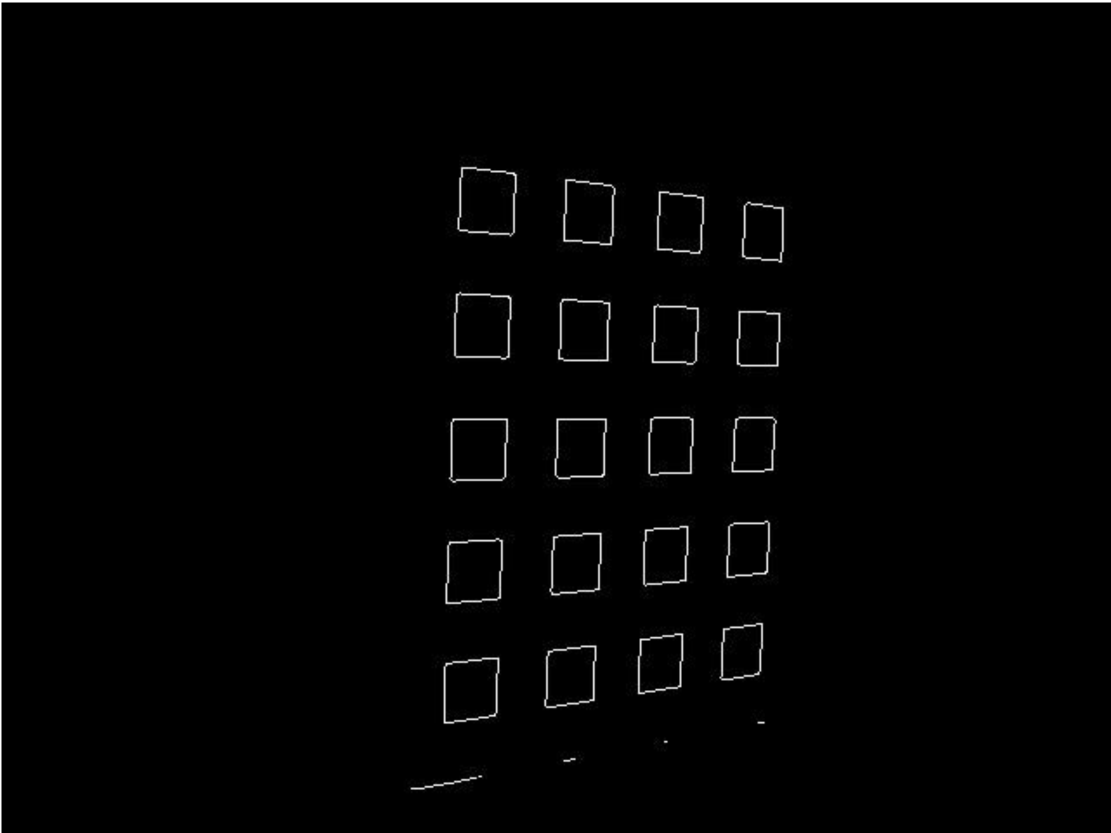

Figure 1: Canny detector for Pic_2

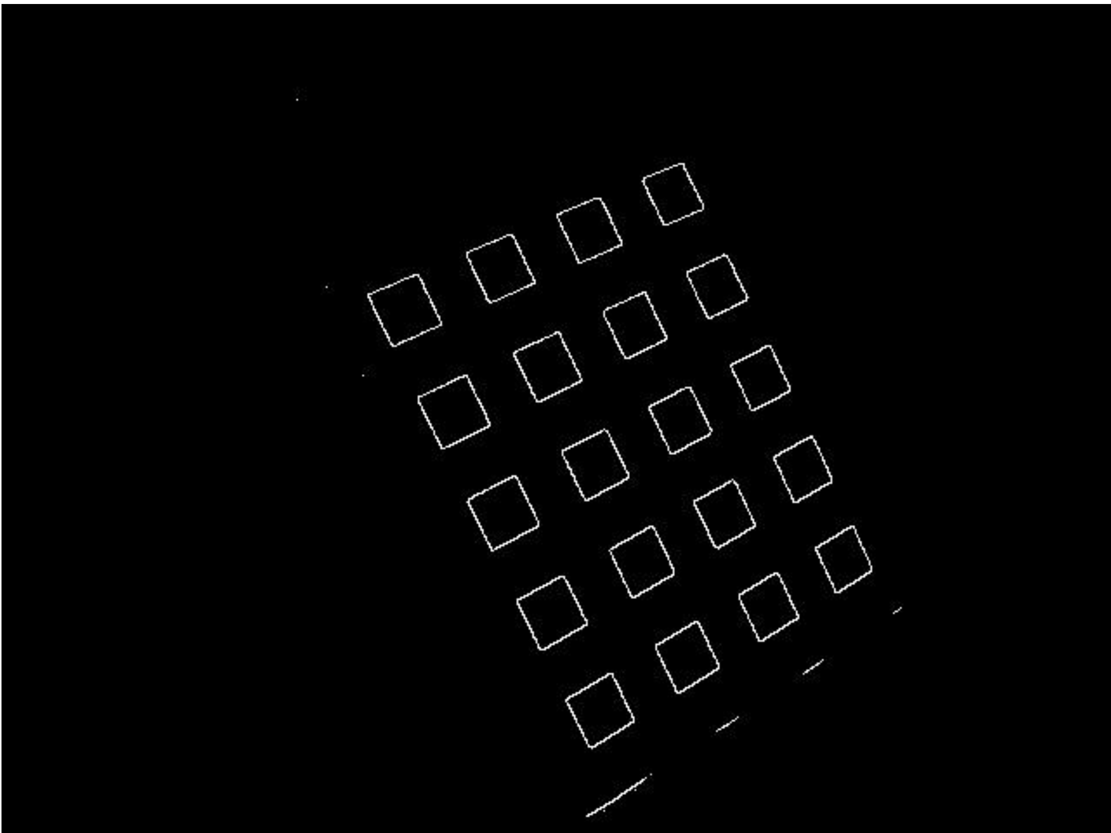

Figure 2: Canny detector for Pic_4

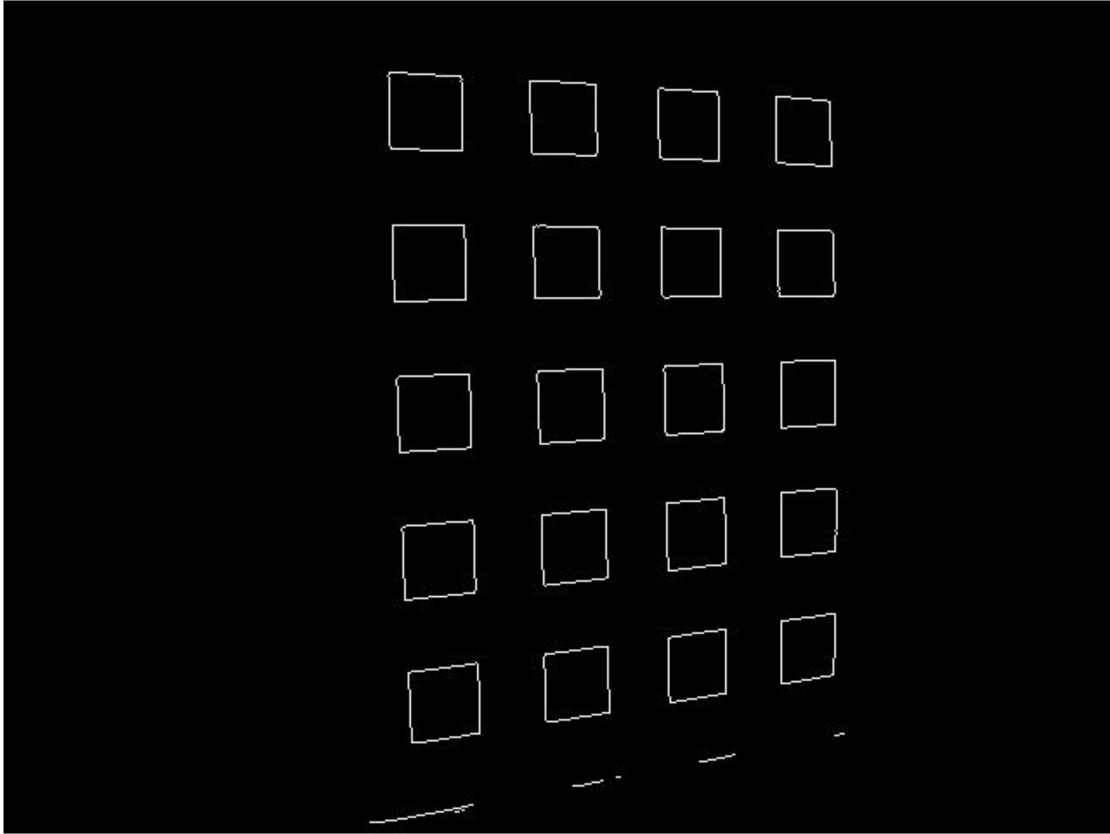

Figure 3: Canny detector for Pic_18

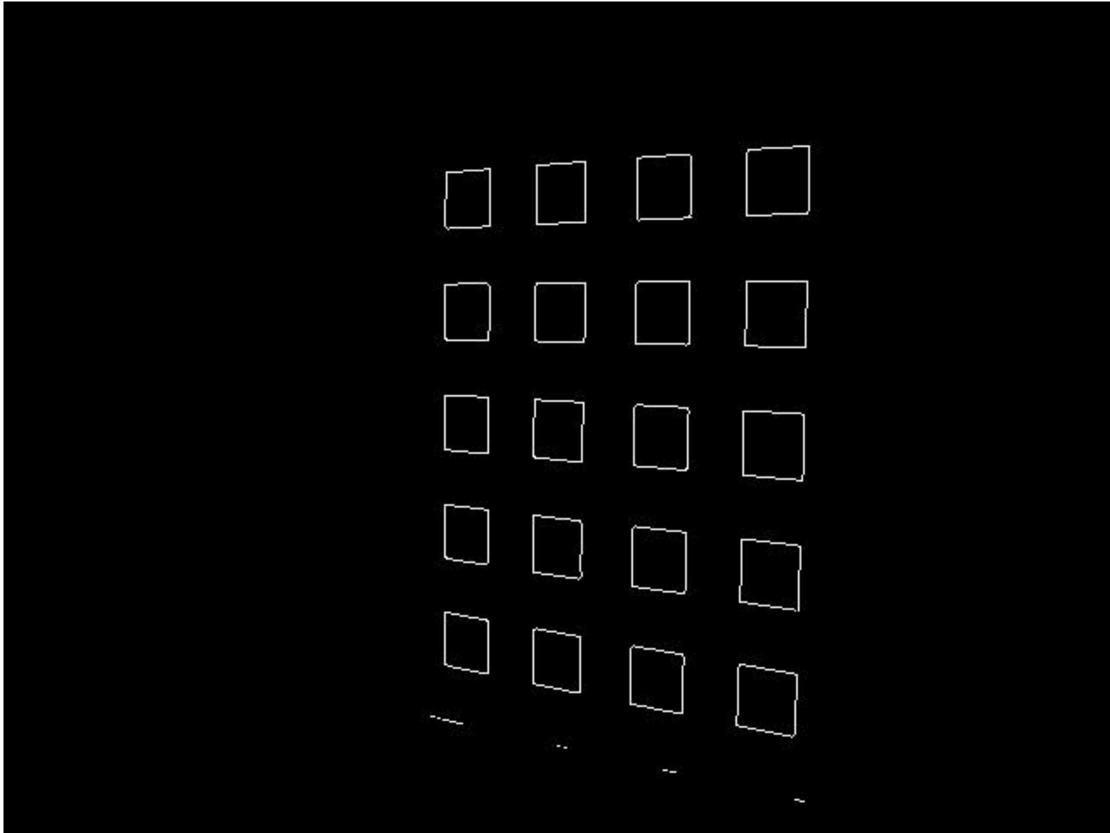

Figure 4: Canny detector for Pic_40

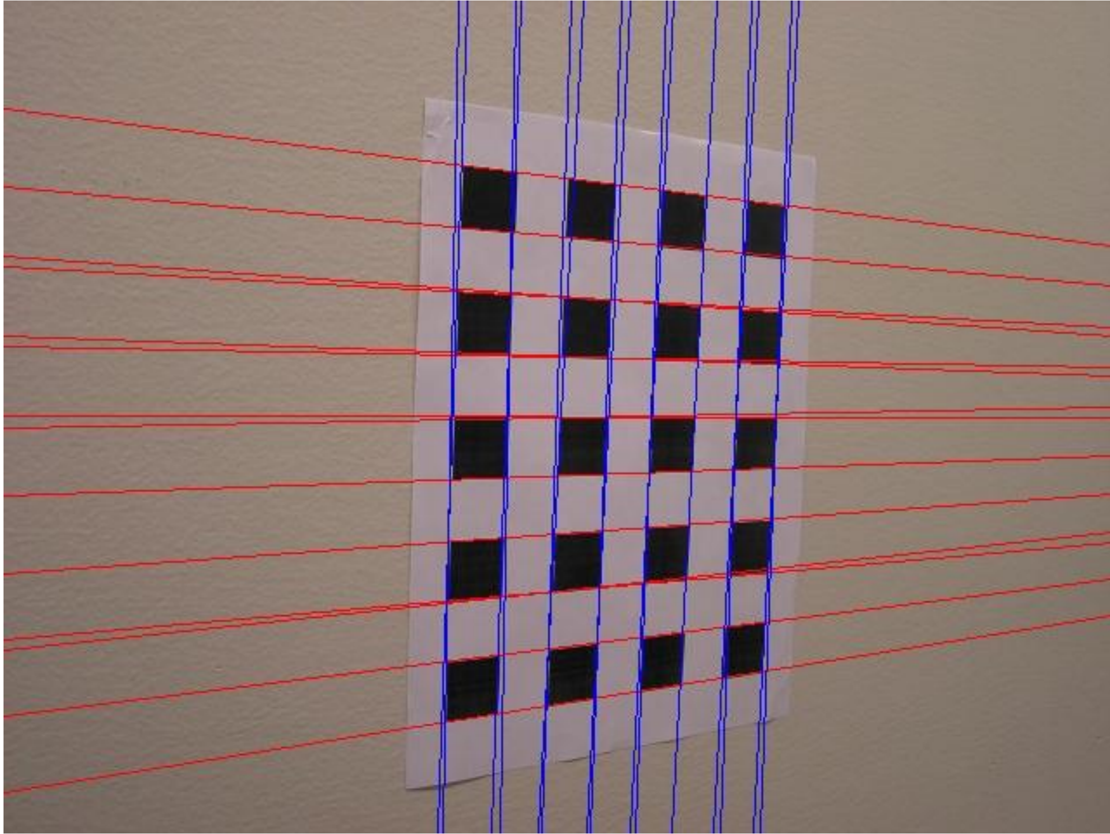

Figure 5: All Hough lines for Pic_2

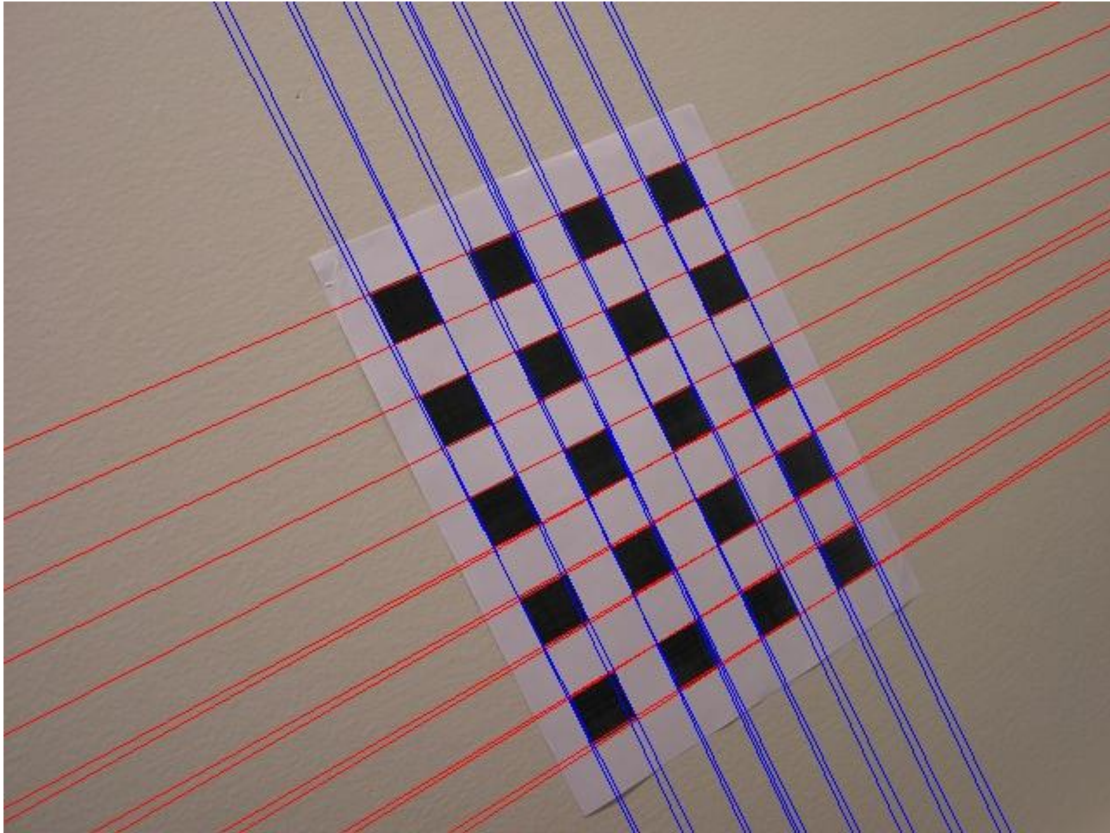

Figure 6: All Hough lines for Pic_4

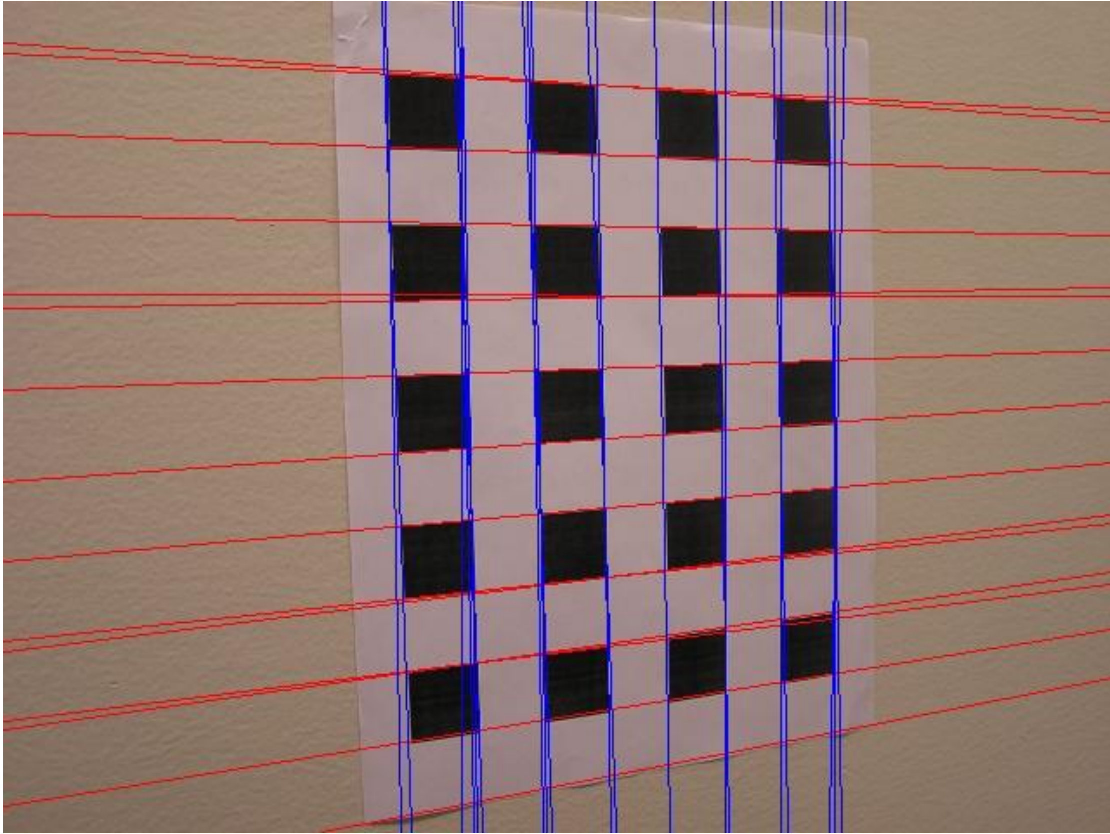

Figure 7: All Hough lines for Pic_18

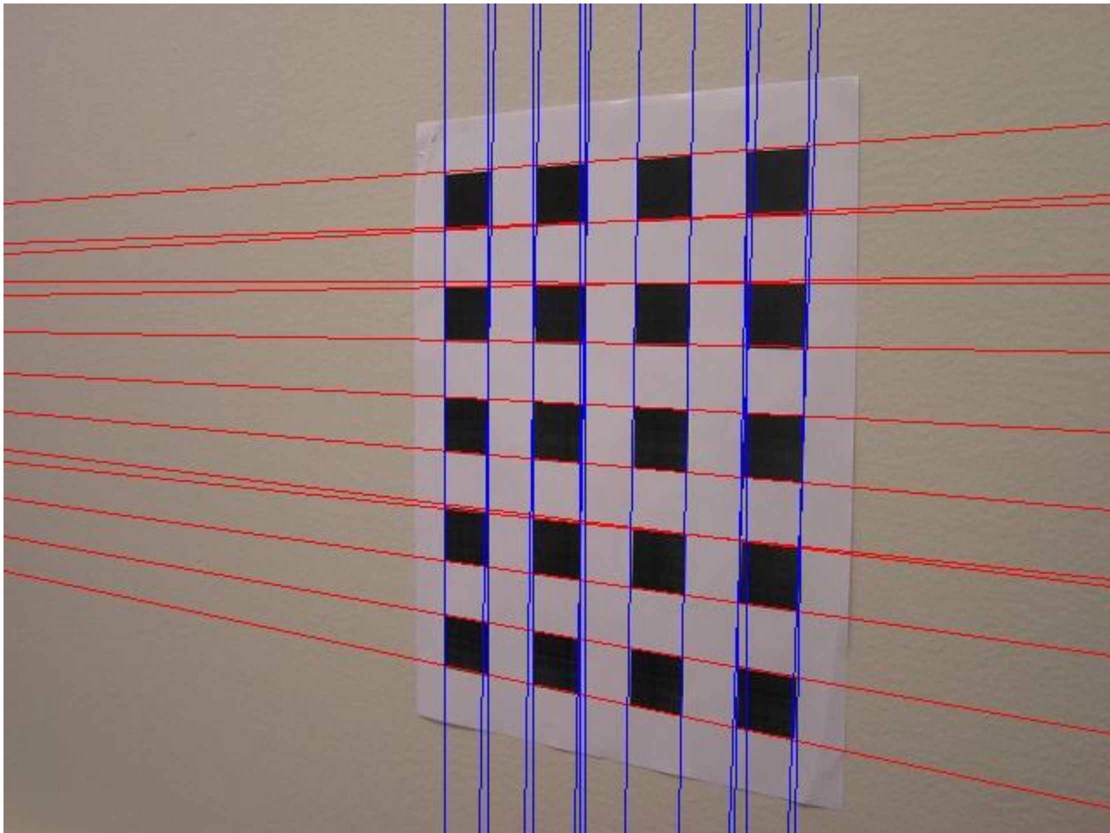

Figure 8: All Hough lines for Pic_40

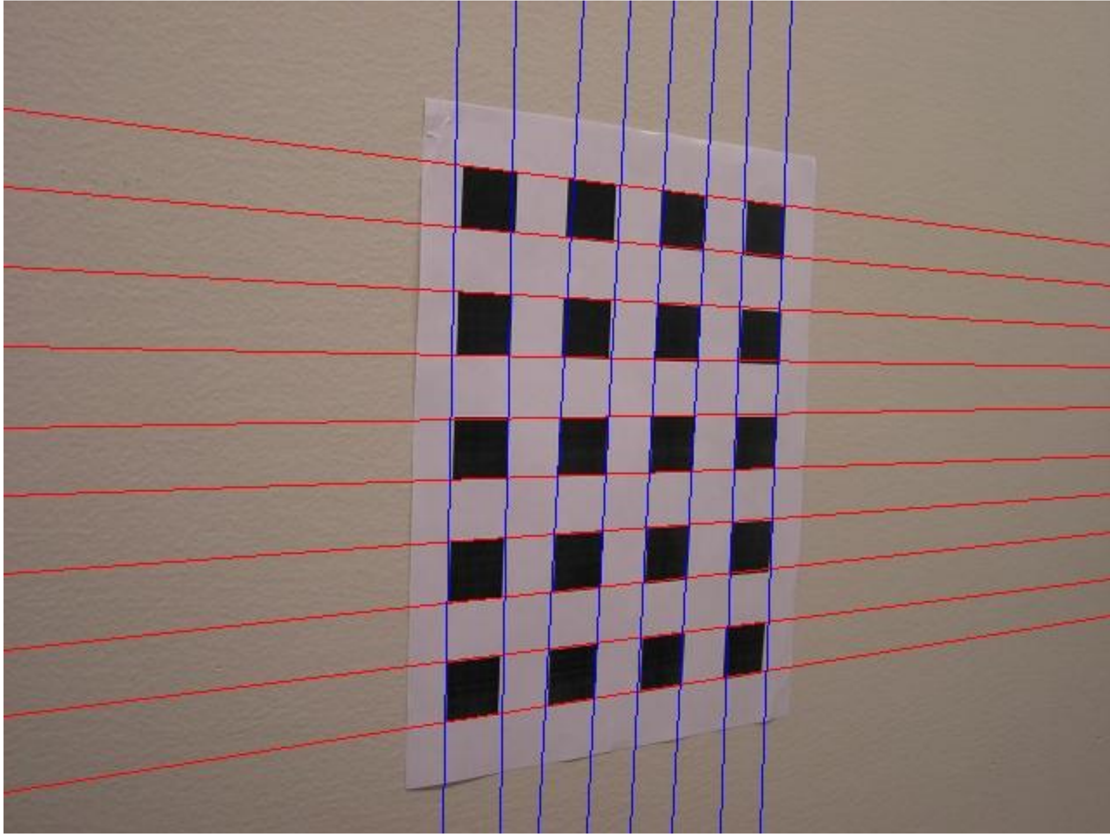

Figure 9: Final Hough lines for Pic_2

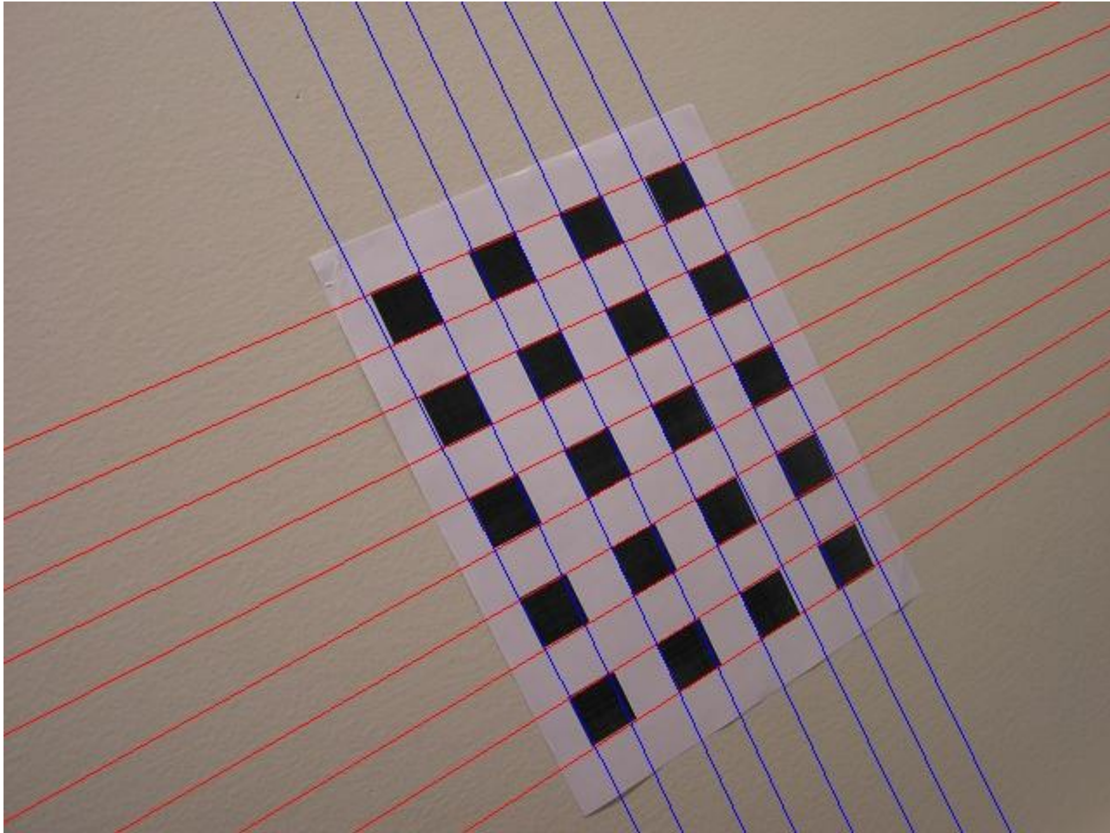

Figure 10: Final Hough lines for Pic_4

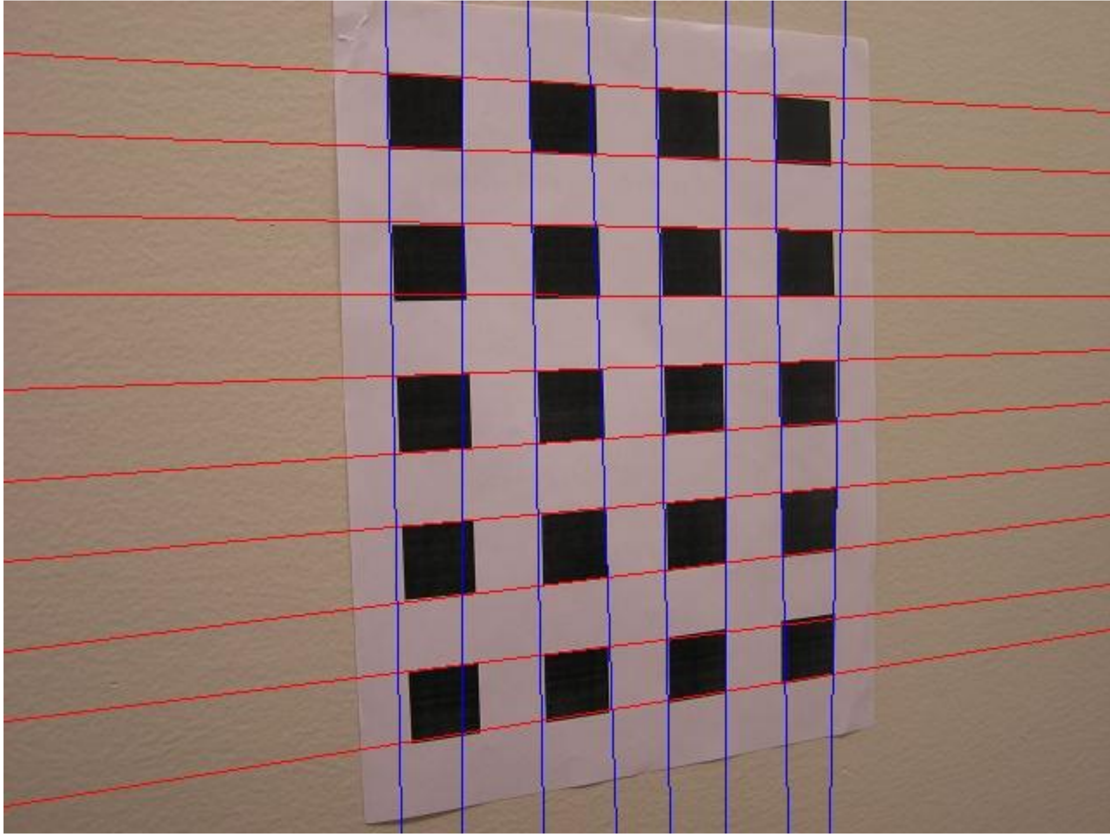

Figure 11: Final Hough lines for Pic_18

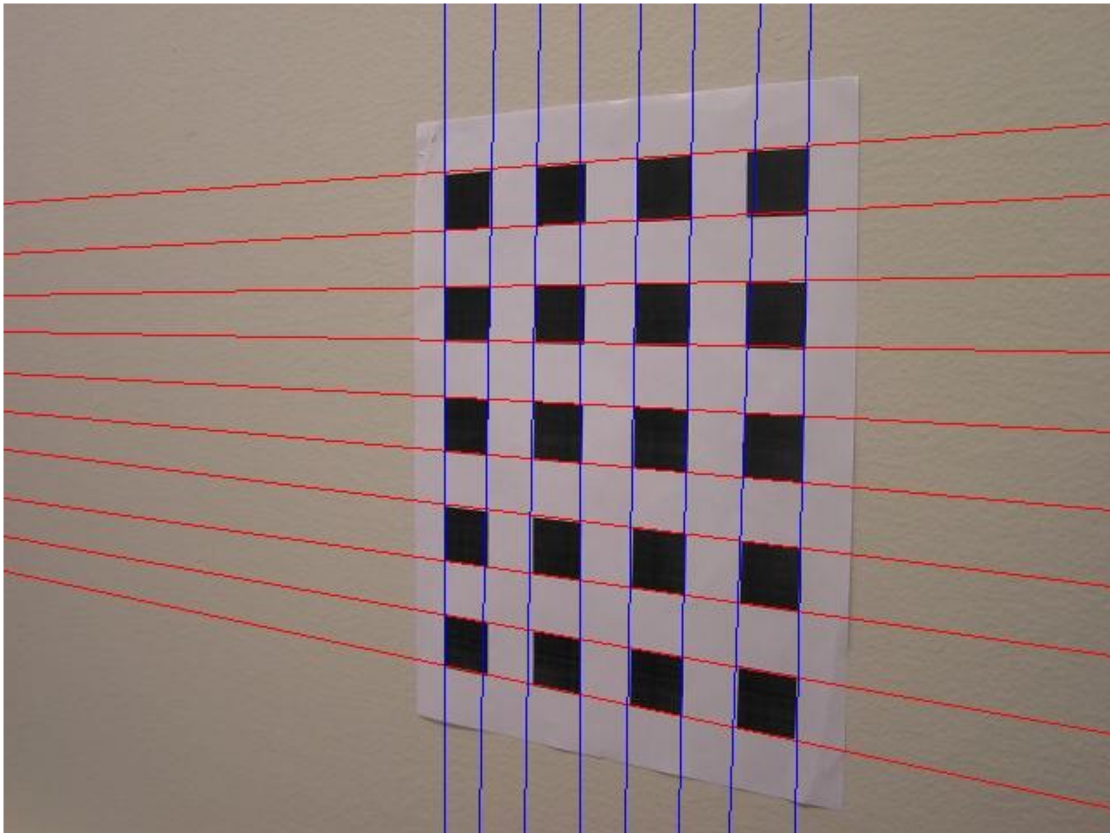

Figure 12: Final Hough lines for Pic_40

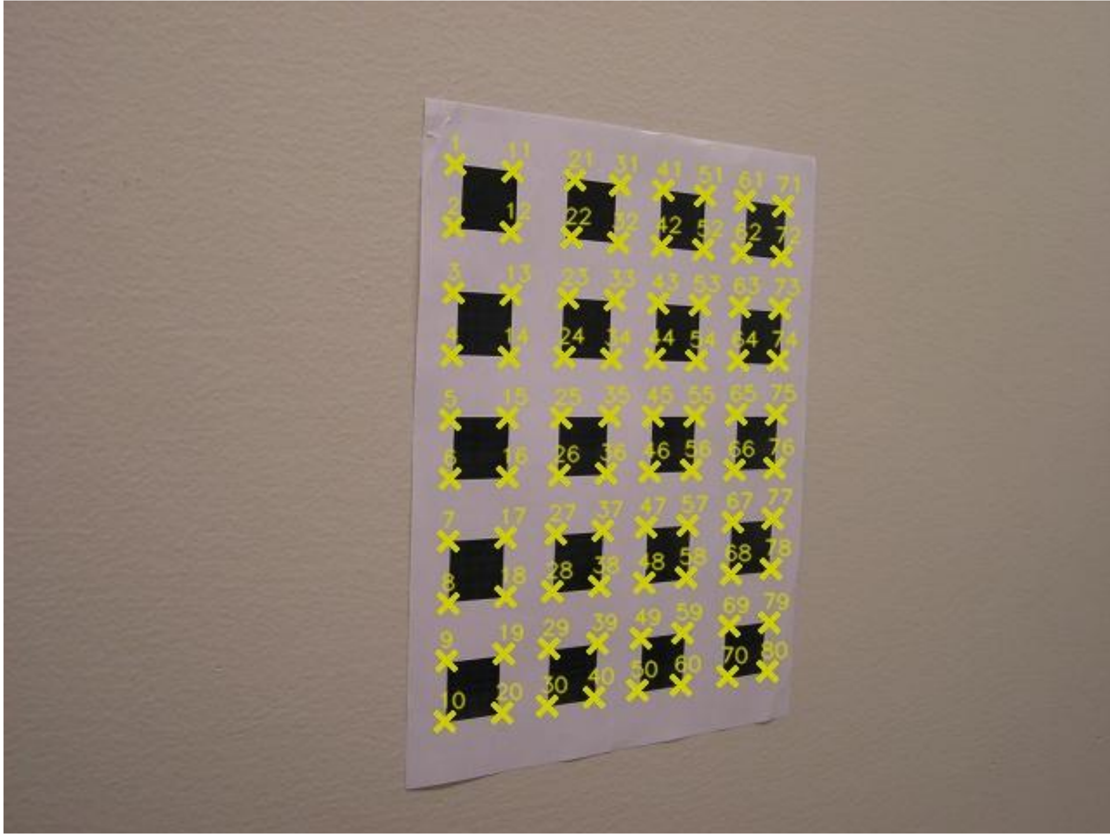

Figure 13: Detected Corners for Pic_2

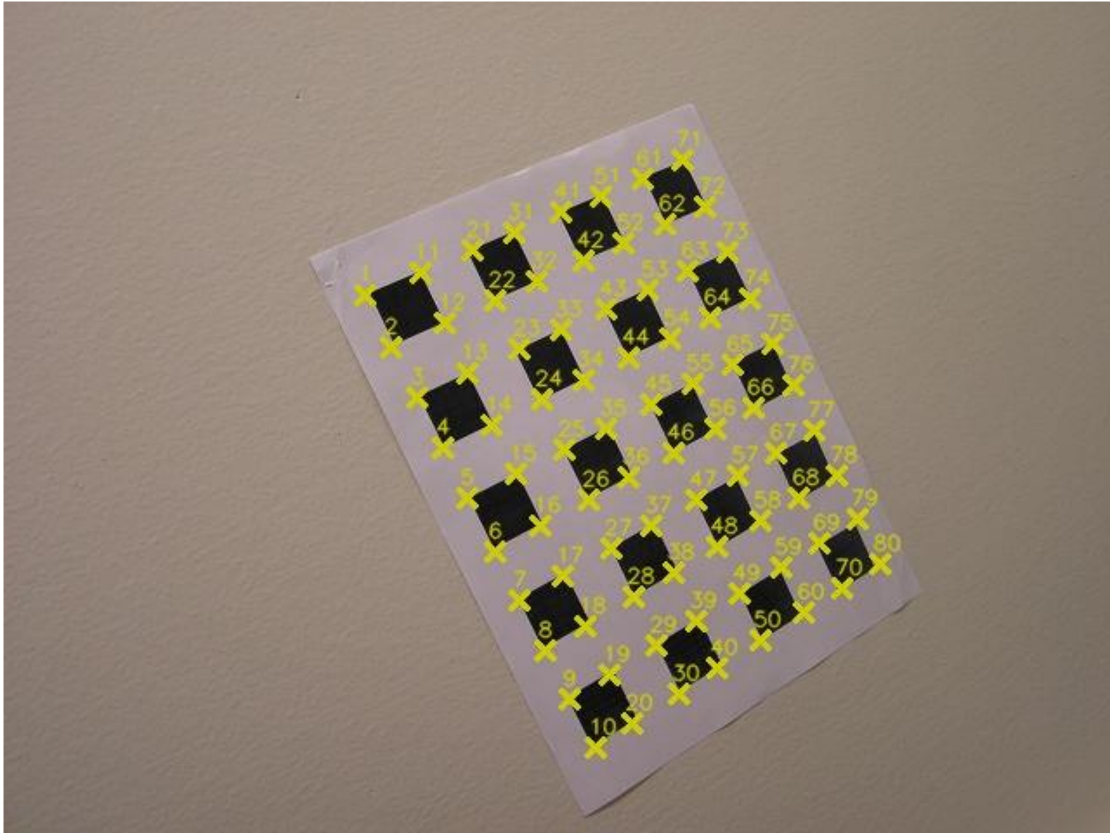

Figure 14: Detected Corners for Pic_4

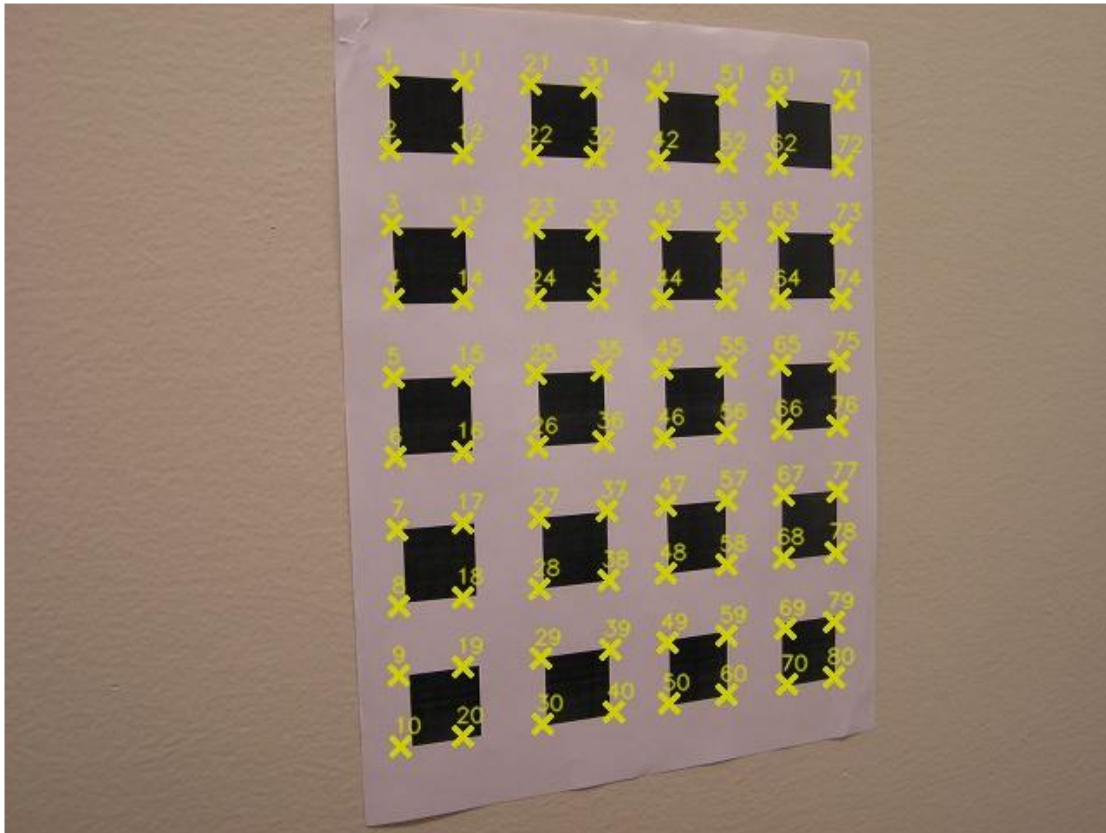

Figure 15: Detected Corners for Pic_18

Figure 16: Detected Corners for Pic_40

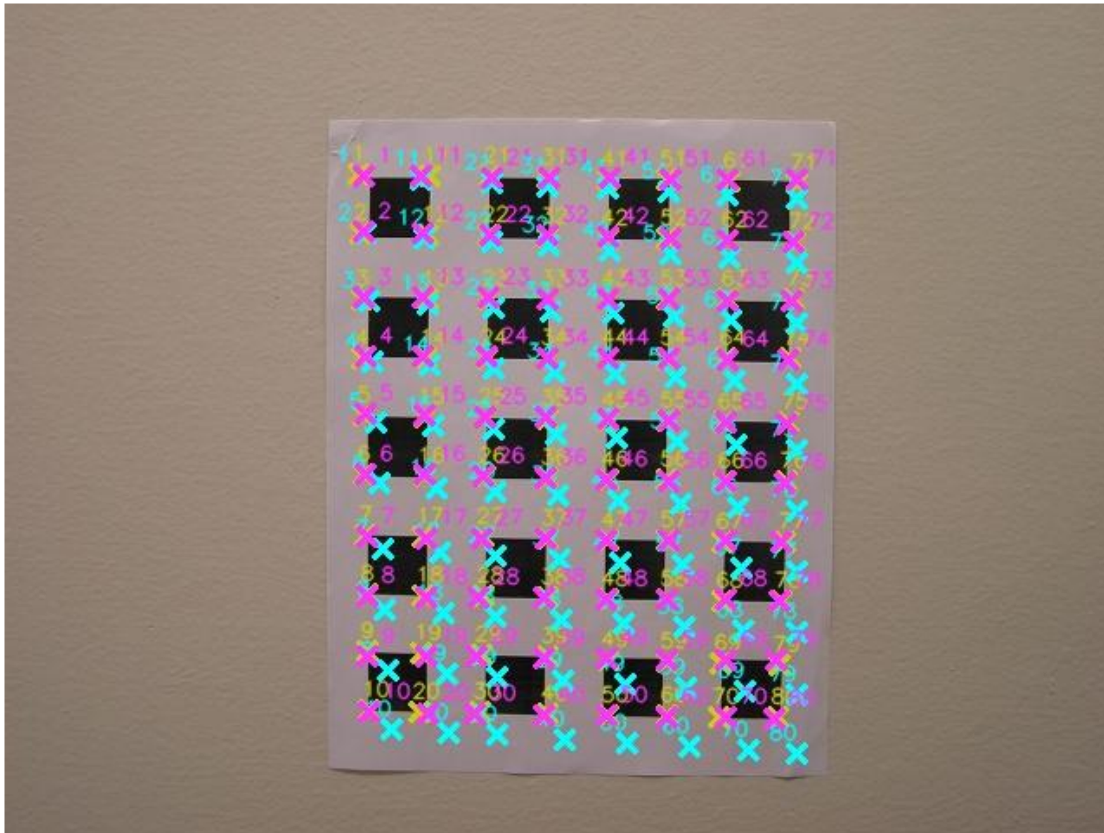

Figure 17: Labels and points for Pic_2 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

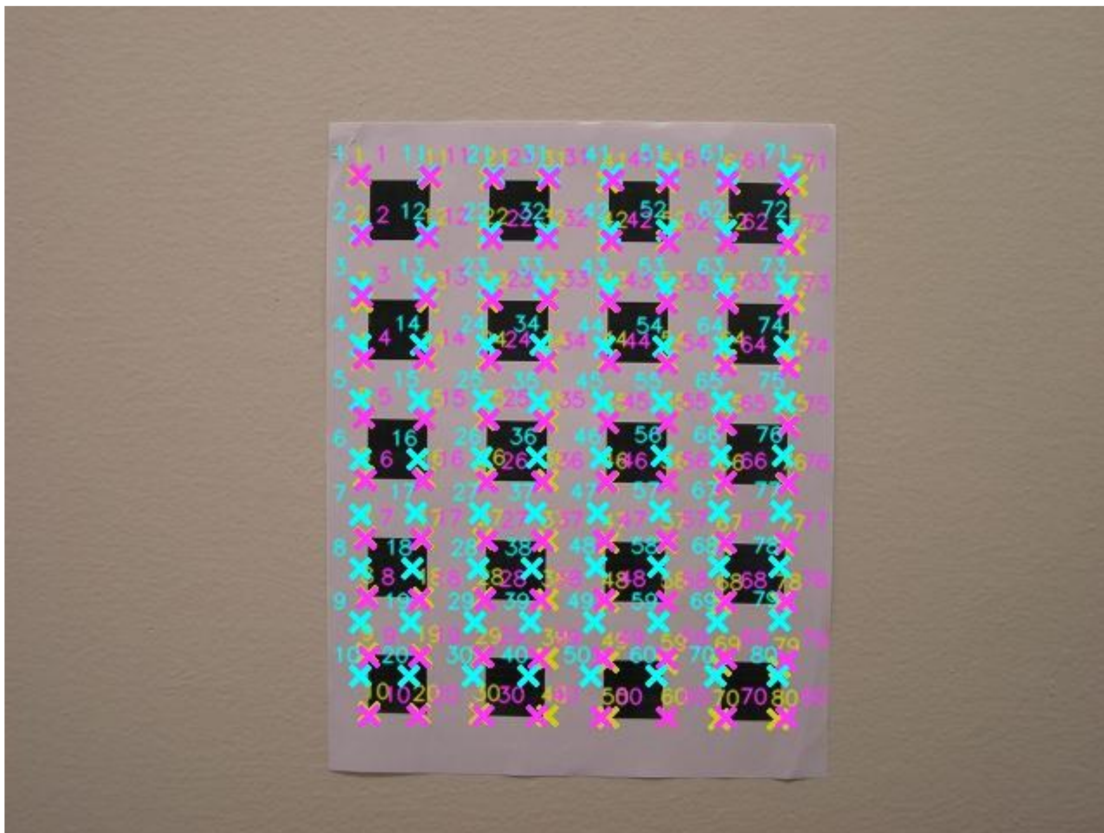

Figure 18: Labels and points for Pic_4 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

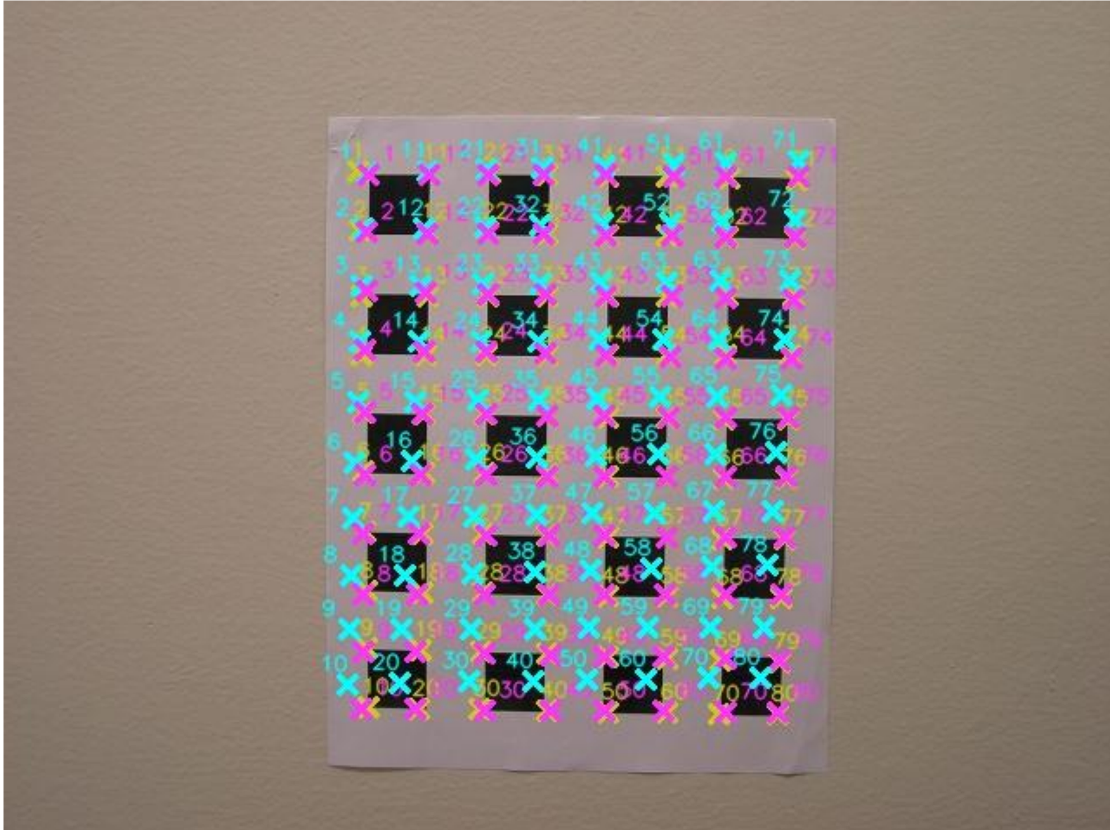

Figure 19: Labels and points for Pic_18 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

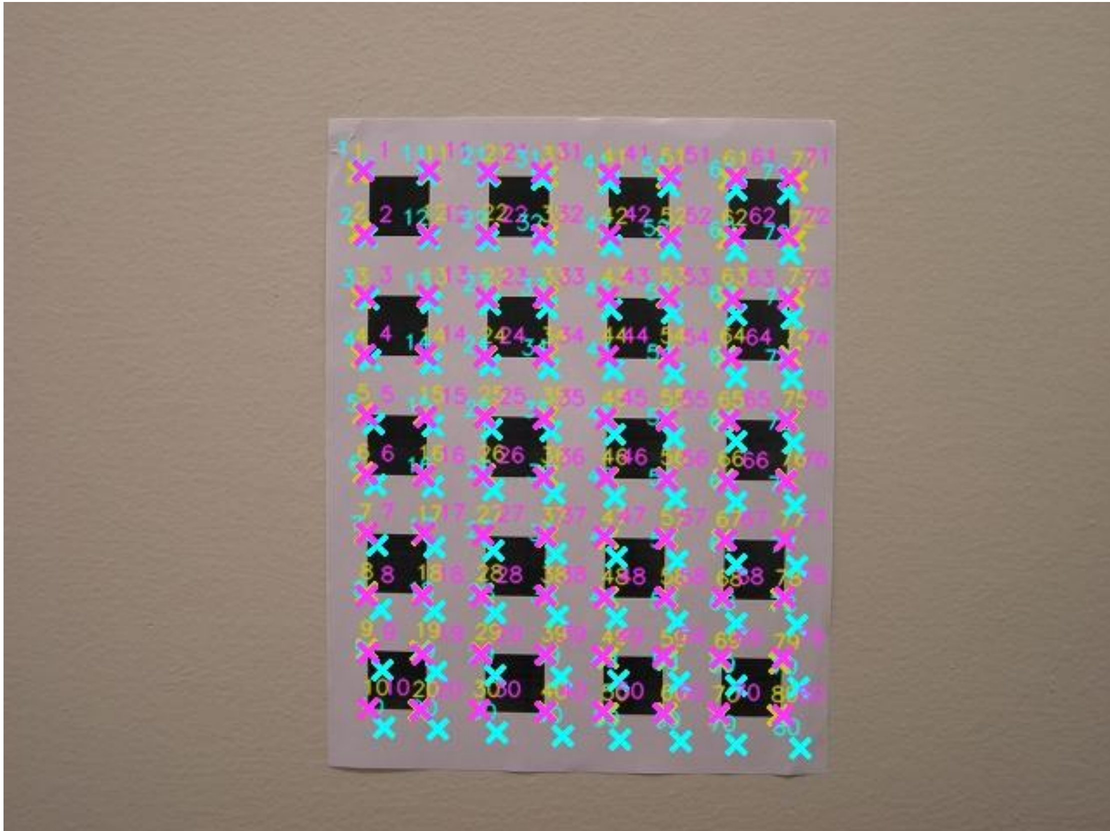

Figure 20: Labels and points for Pic_40 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

We can see from the above four images that using LM optimization improves the performance significantly. Some of the LM projected points (pink) are more accurate than the original estimated points (yellow). This is the case because we use the real-world measurements for LM.

If we look at the improvement in terms of mean and variance of the distance between the actual point (yellow) and the original or refined projection respectively we observe the following values:

	Pic_2	**Pic_4**	**Pic_18**	**Pic_40**
mean original	12.7805	13.4414	12.1607	10.7896
mean refined	2.3811	2.0079	1.9480	1.7185
variance original	34.7540	50.9309	29.4631	21.5682
variance refined	1.9027	1.7366	0.9626	1.1319

Table 3: mean and variance before and after LM optimization

And we can observe the following intrinsic camera parameters:

$$\boldsymbol{K} = \begin{bmatrix} 718.811 & 5.03085 & 263.156 \\ 0 & 713.608 & 324.925 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{K}_{refined} = \begin{bmatrix} 720.536 & 2.14895 & 320.338 \\ 0 & 717.347 & 235.569 \\ 0 & 0 & 1 \end{bmatrix}$$

For the respective images we get the following extrinsic camera parameters:

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_2} = \begin{bmatrix} -0.836101 & 0.0111407 & -0.548462 & -1.11032 \\ 0.0441387 & -0.995186 & -0.0875017 & 152.628 \\ -0.546797 & -0.0973687 & 0.831584 & -471.409 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_2,refined} = \begin{bmatrix} -0.832207 & 0.0388003 & -0.553105 & 38.9754 \\ 0.00476626 & -0.997011 & -0.0771116 & 98.244 \\ -0.554444 & -0.0668091 & 0.829535 & -500.315 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_4} = \begin{bmatrix} 0.814494 & 0.444632 & -0.372694 & -37.0911 \\ -0.426108 & 0.894417 & 0.135832 & -112.393 \\ 0.393739 & 0.0481738 & 0.917959 & 512.82 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_4,refined} = \begin{bmatrix} 0.826725 & 0.430115 & -0.362666 & -84.3509 \\ -0.418766 & 0.900926 & 0.113872 & -51.6182 \\ 0.375714 & 0.0577314 & 0.924936 & 554.333 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_18} = \begin{bmatrix} 0.921936 & 0.040576 & -0.385212 & -22.6886 \\ -0.0635958 & 0.996859 & -0.0472018 & -157.774 \\ 0.382087 & 0.0680149 & 0.92162 & 394.982 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_18,refined} = \begin{bmatrix} 0.924644 & 0.0138298 & -0.380583 & -57.4718 \\ -0.0446701 & 0.996381 & -0.0723212 & -112.702 \\ 0.378205 & 0.083872 & 0.921915 & 411.246 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_40} = \begin{bmatrix} -0.804331 & 0.0393246 & 0.592878 & 6.08788 \\ -0.0885824 & -0.994593 & -0.0542061 & 186.384 \\ 0.587541 & -0.0961182 & 0.803466 & 0.803466 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_40,refined} = \begin{bmatrix} -0.821167 & 0.0149849 & 0.570492 & 51.8059 \\ -0.0482401 & -0.9979 & -0.0432254 & 109.145 \\ 0.568646 & -0.0630158 & 0.820165 & -568.705 \end{bmatrix}$$

Pic_11 is the fixed image. Look at its matrix as well.

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_11} = \begin{bmatrix} 0.999901 & -0.0132451 & 0.00467088 & -38.9977 \\ 0.0134912 & 0.998264 & -0.0573241 & -165.075 \\ -0.0039035 & 0.0573815 & 0.998345 & 518.092 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_11,refined} = \begin{bmatrix} 0.999852 & -0.0124638 & 0.0118183 & -80.9059 \\ 0.0131443 & 0.998149 & -0.0593719 & -99.8428 \\ -0.0110564 & 0.0595185 & 0.998166 & 520.186 \end{bmatrix}$$

R is very close to the identity matrix. Therefore, we can say that the fixed image is indeed the fixed image.

Part 4.2: Dataset 2

Note: all lines are without interruption. Due to the lines being very thin the images do not scale well.

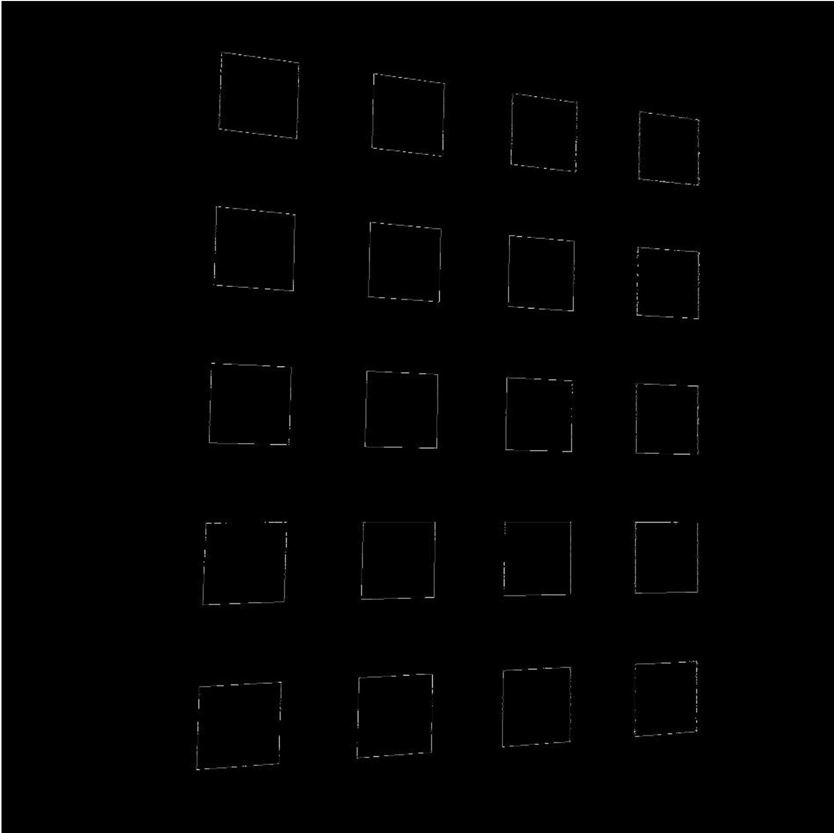

Figure 21: Canny detector for Pic_3

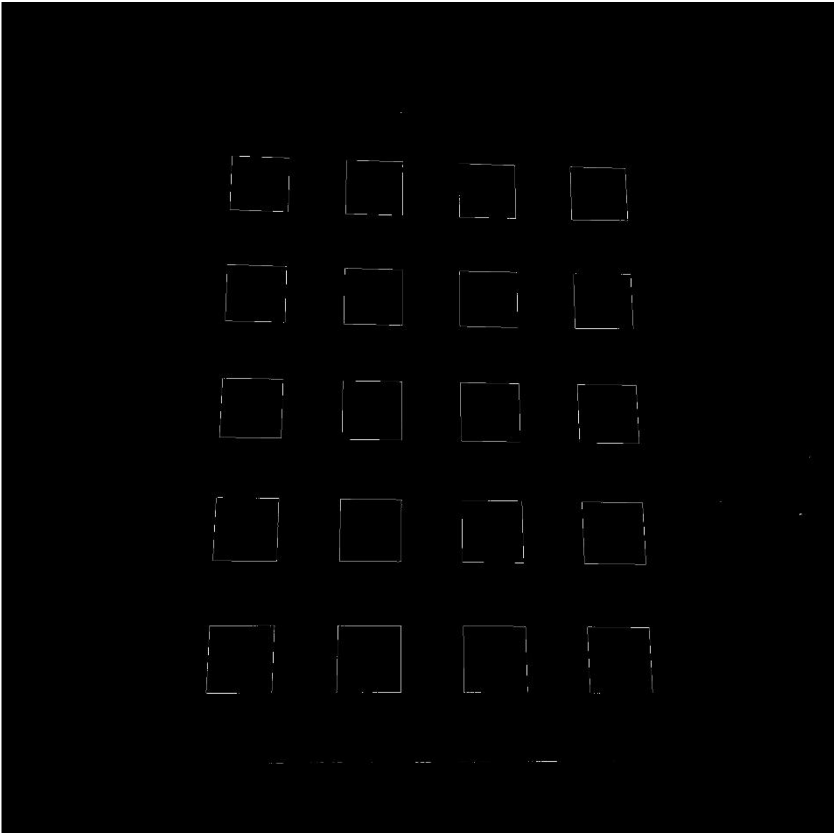

Figure 22: Canny detector for Pic_11

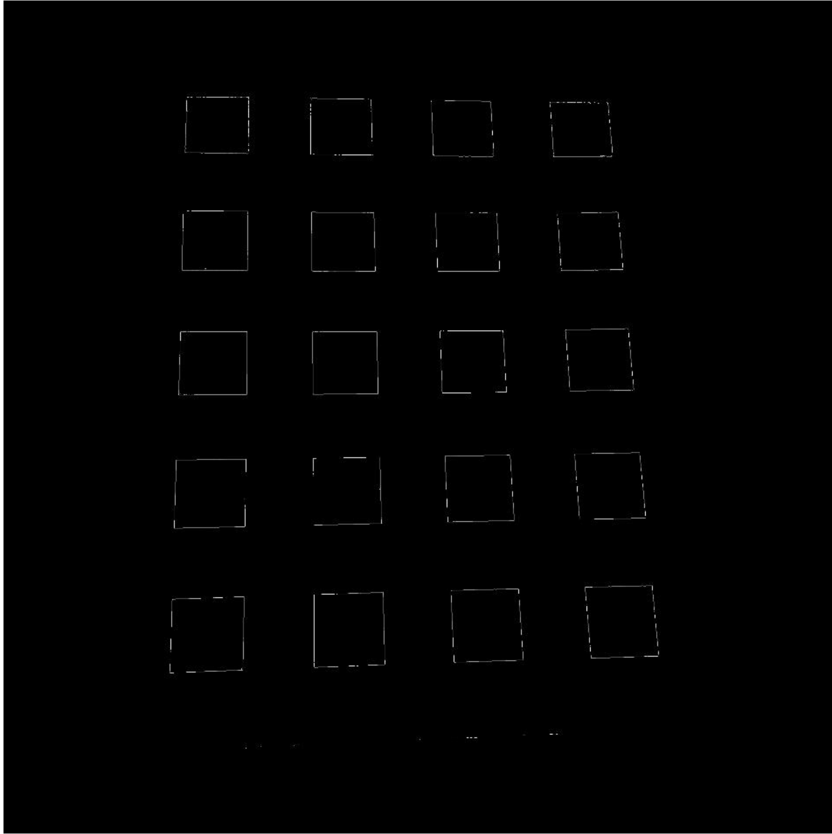

Figure 23: Canny detector for Pic_22

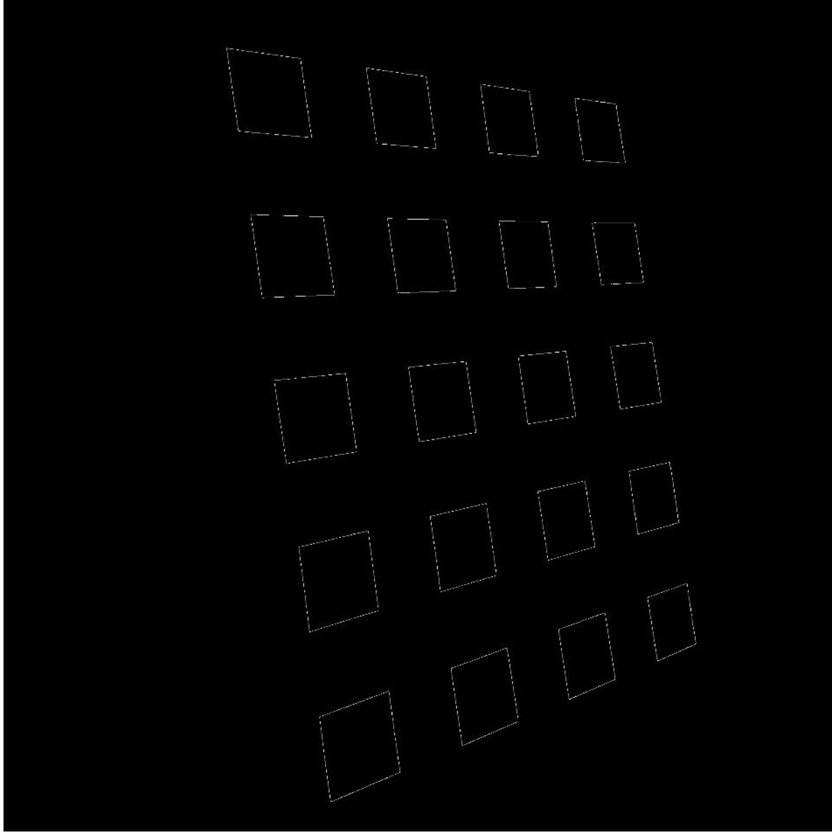

Figure 24: Canny detector for Pic_34

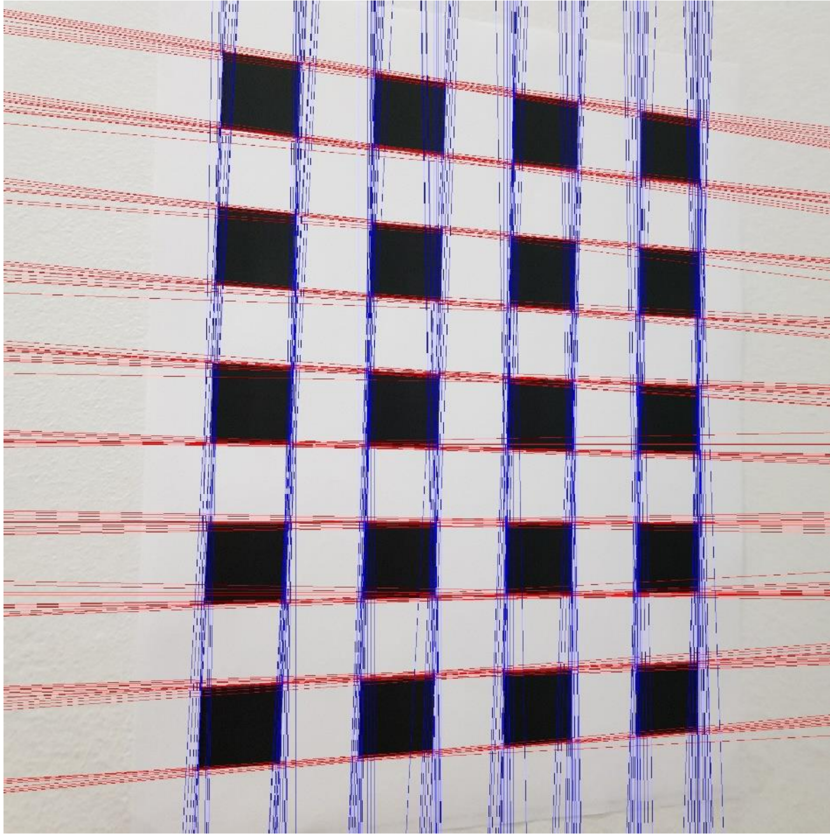

Figure 25: All Hough lines for Pic_3

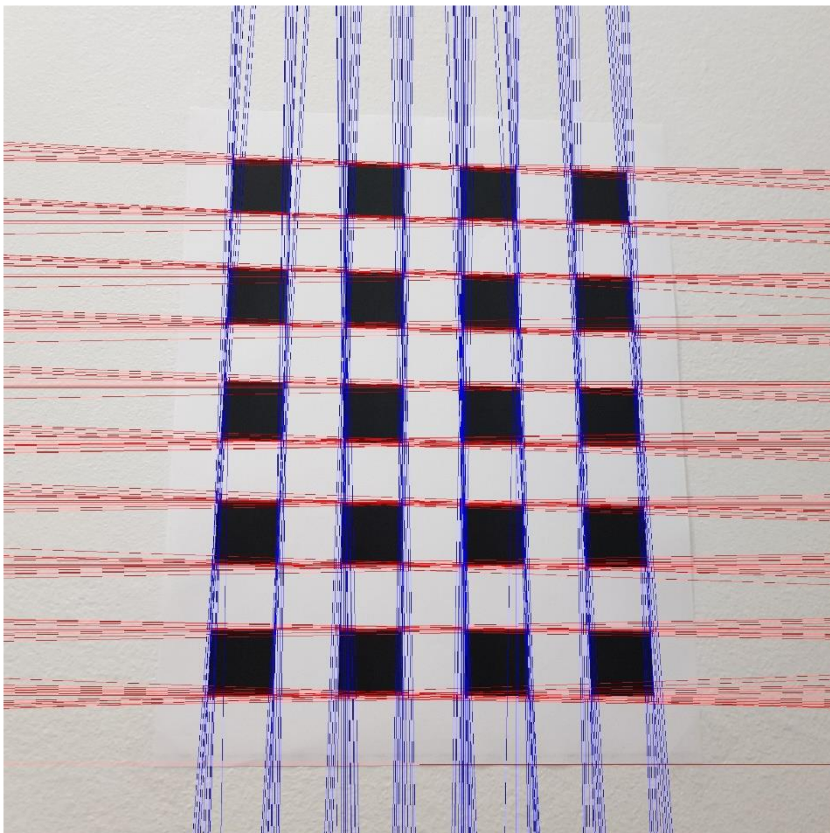

Figure 26: All Hough lines for Pic_11

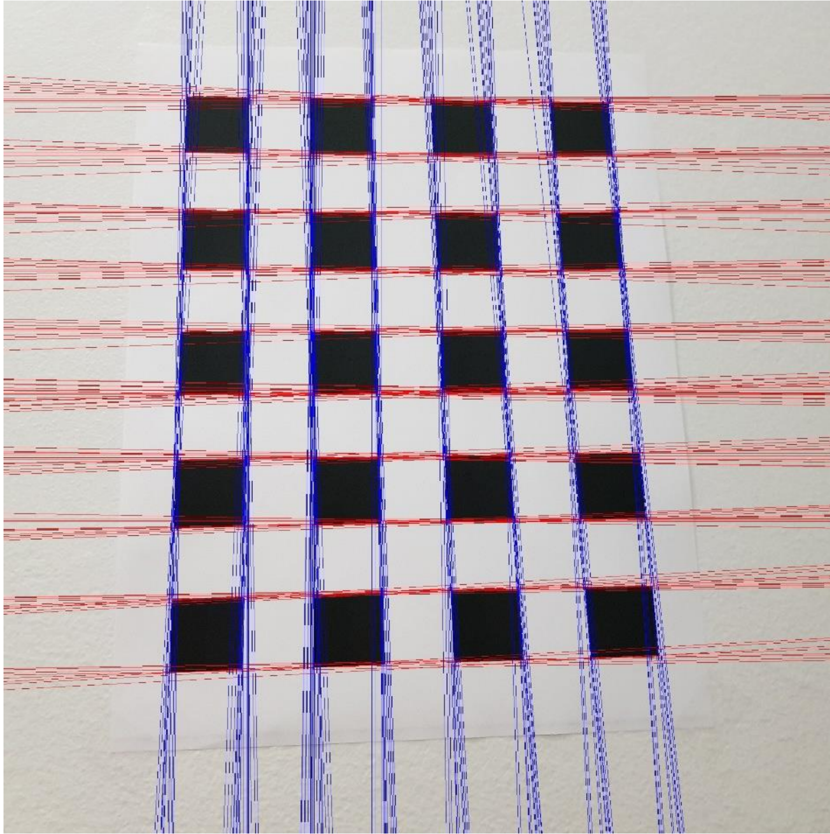

Figure 27: All Hough lines for Pic_22

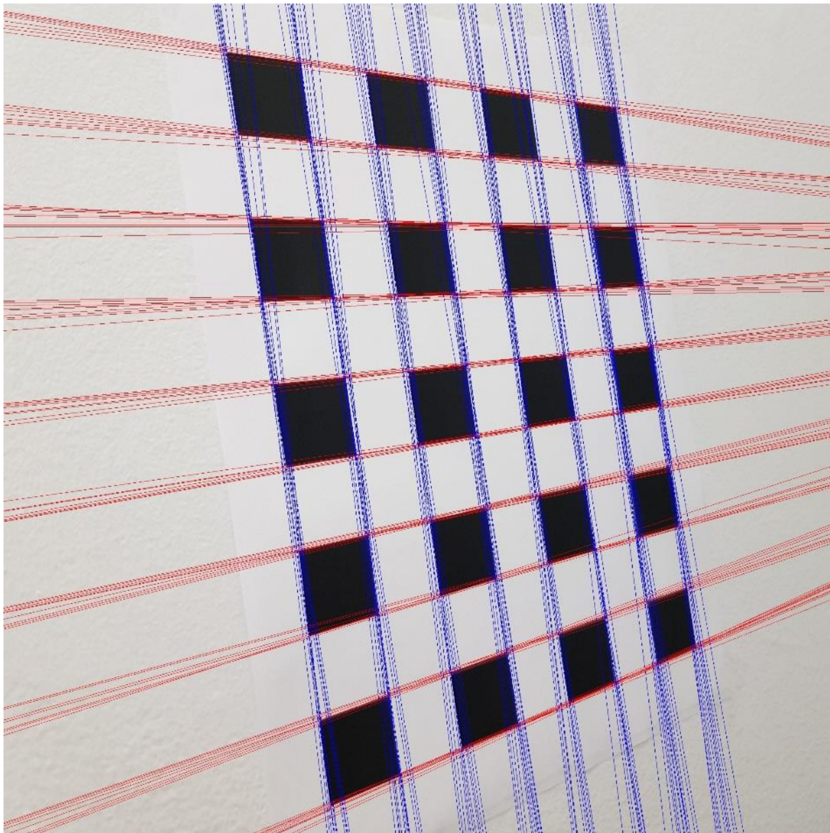

Figure 28: All Hough lines for Pic_34

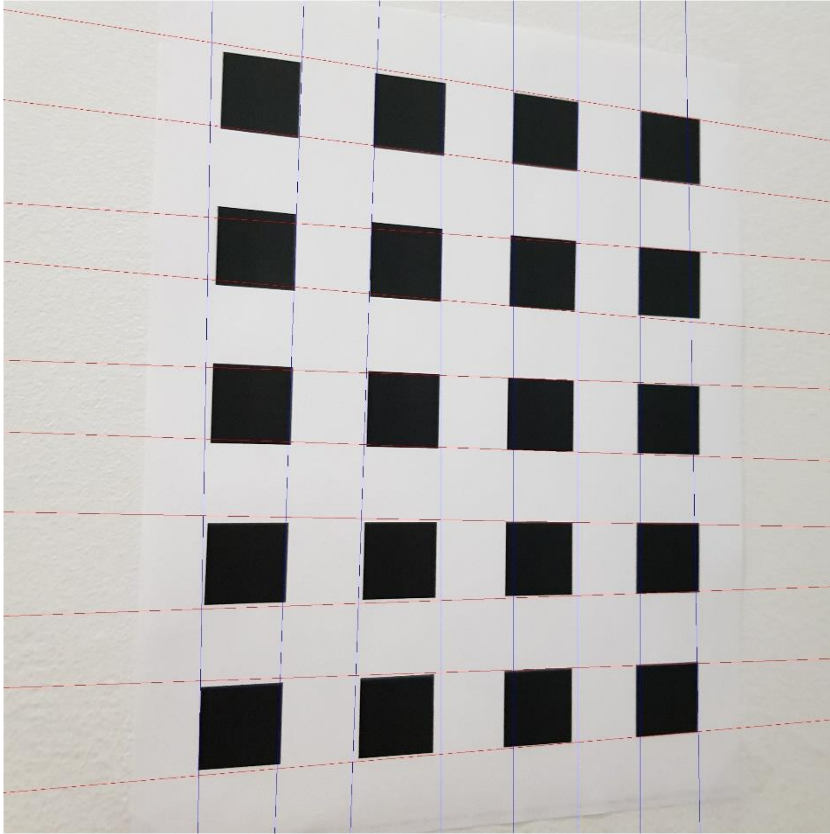

Figure 29: Final Hough lines for Pic_3

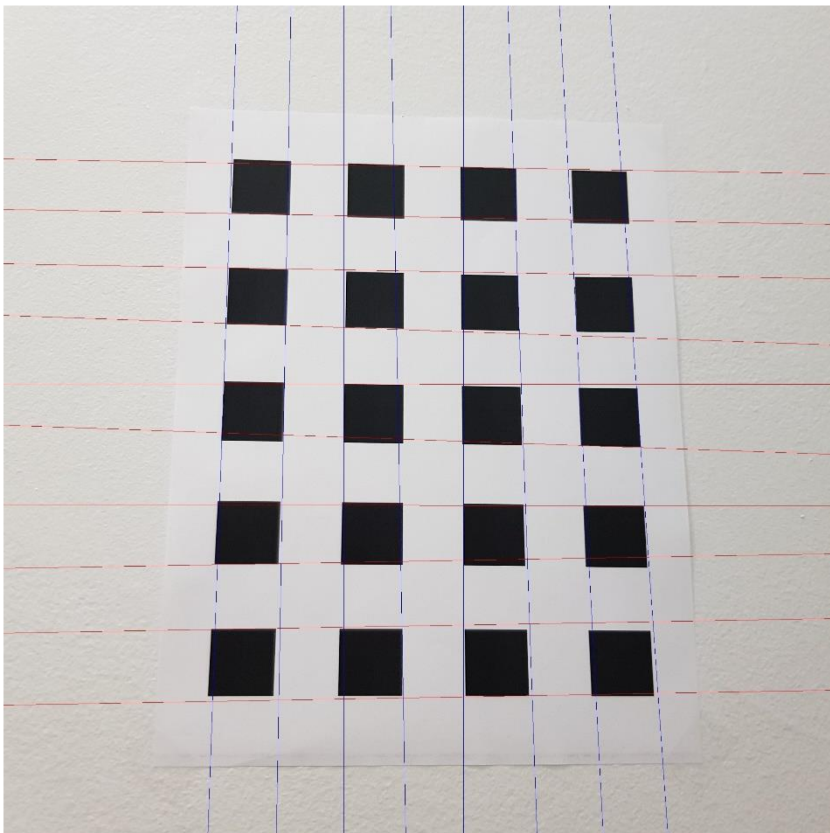

Figure 30: Final Hough lines for Pic_11

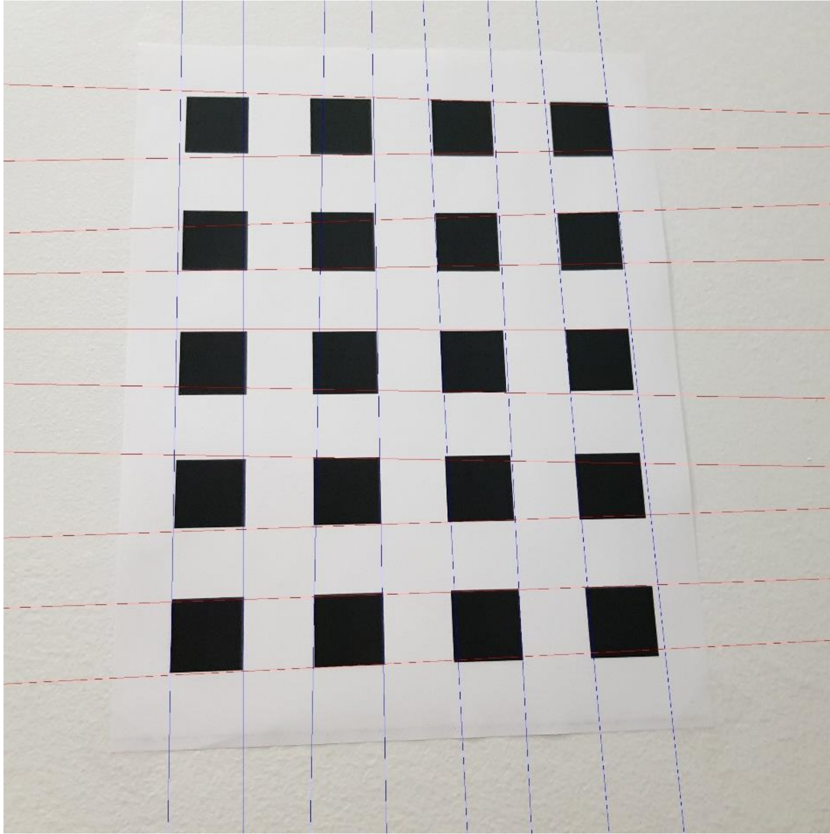

Figure 31: Final Hough lines for Pic_22

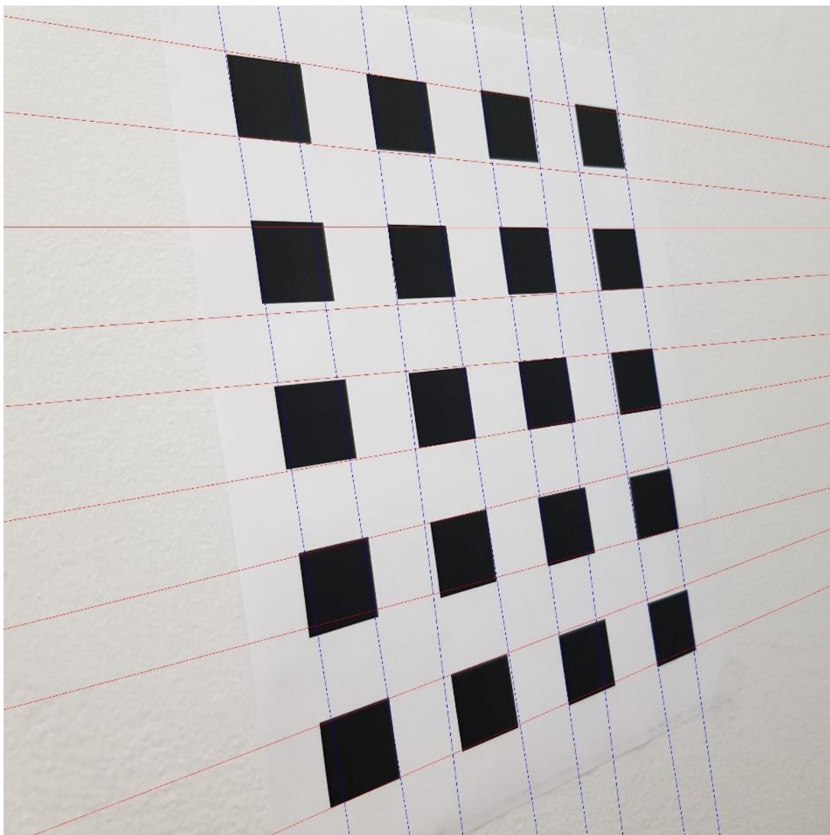

Figure 32: Final Hough lines for Pic_34

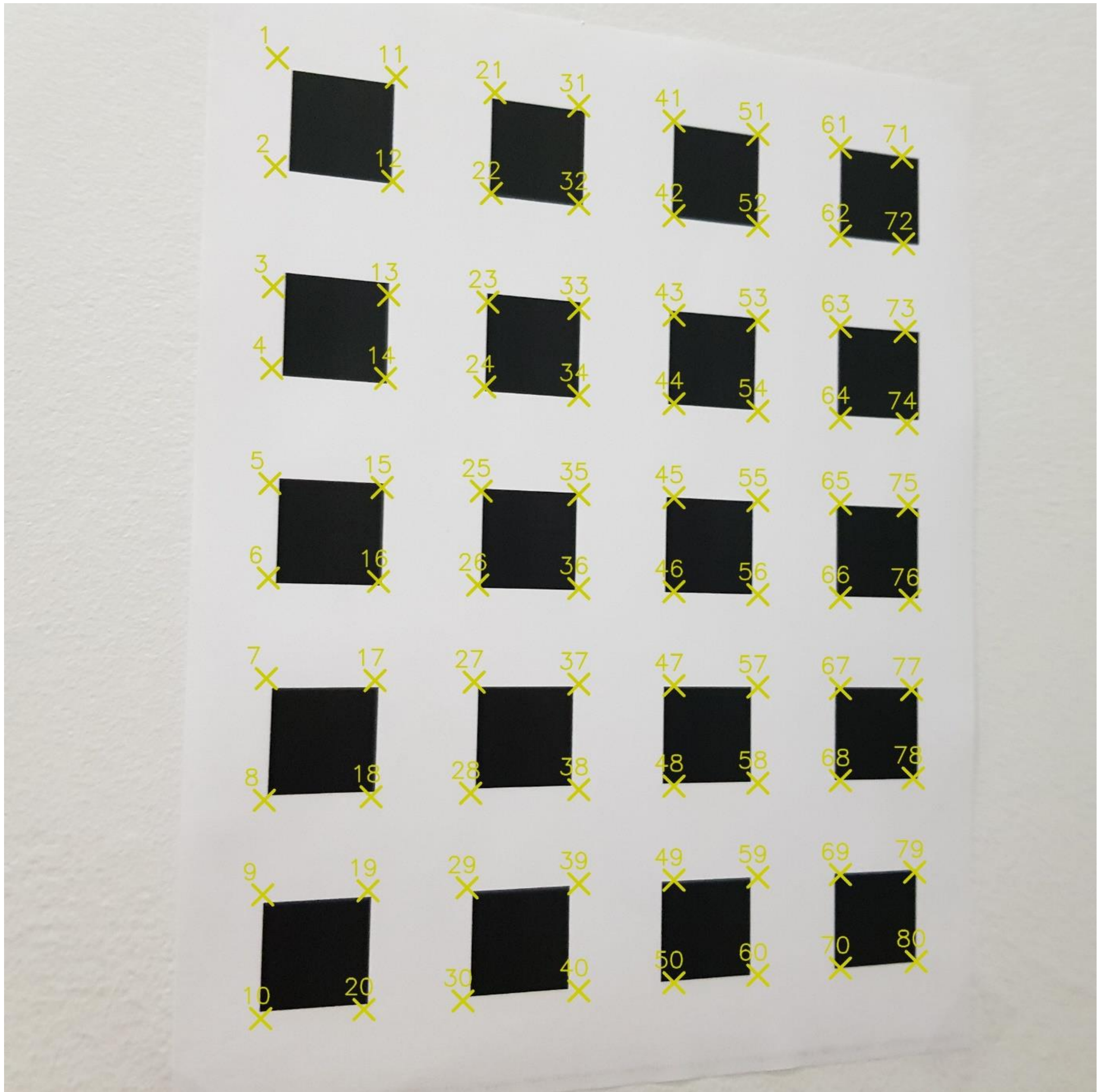

Figure 33: Detected Corners for Pic_3

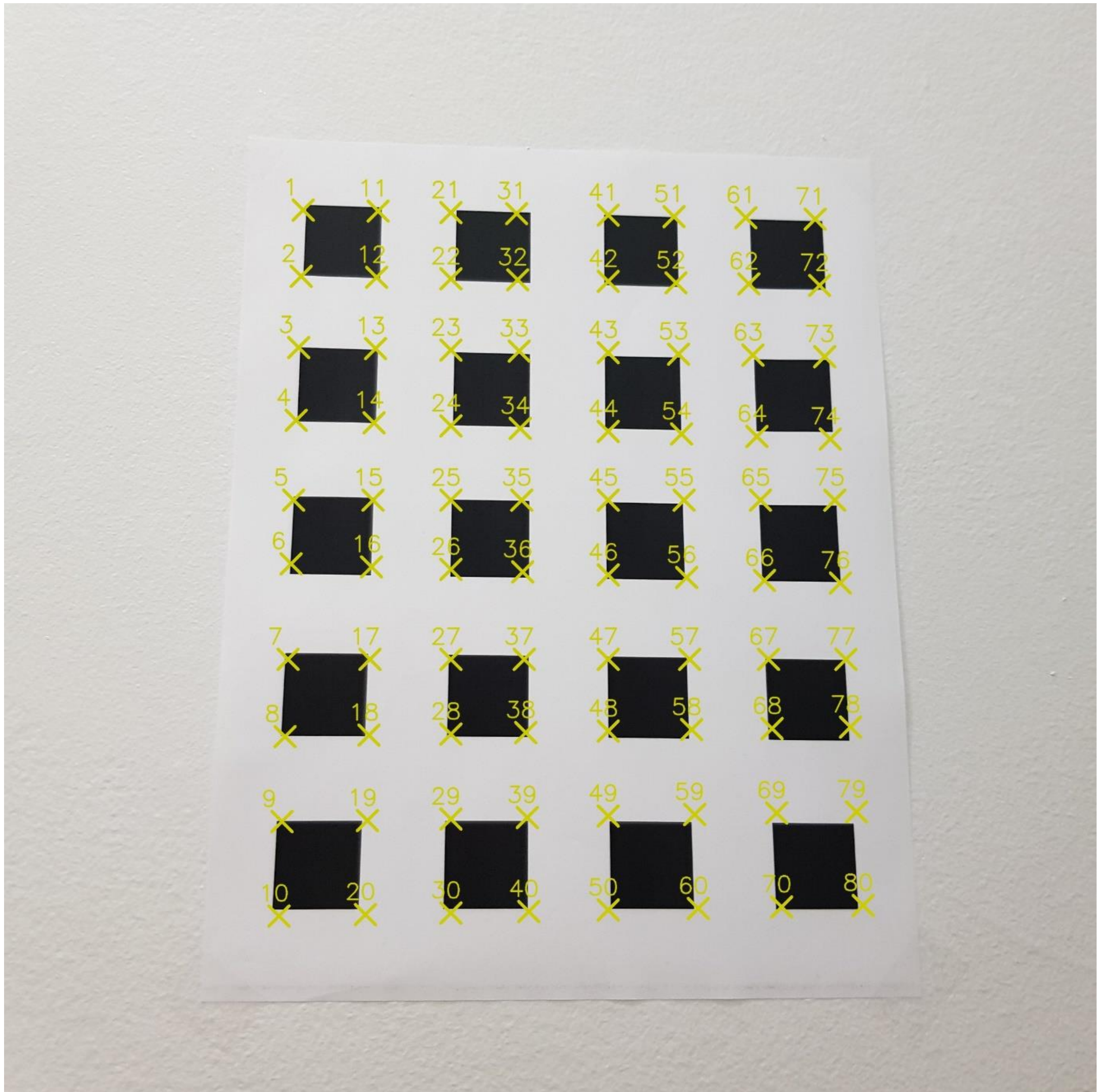

Figure 34: Detected Corners for Pic_11

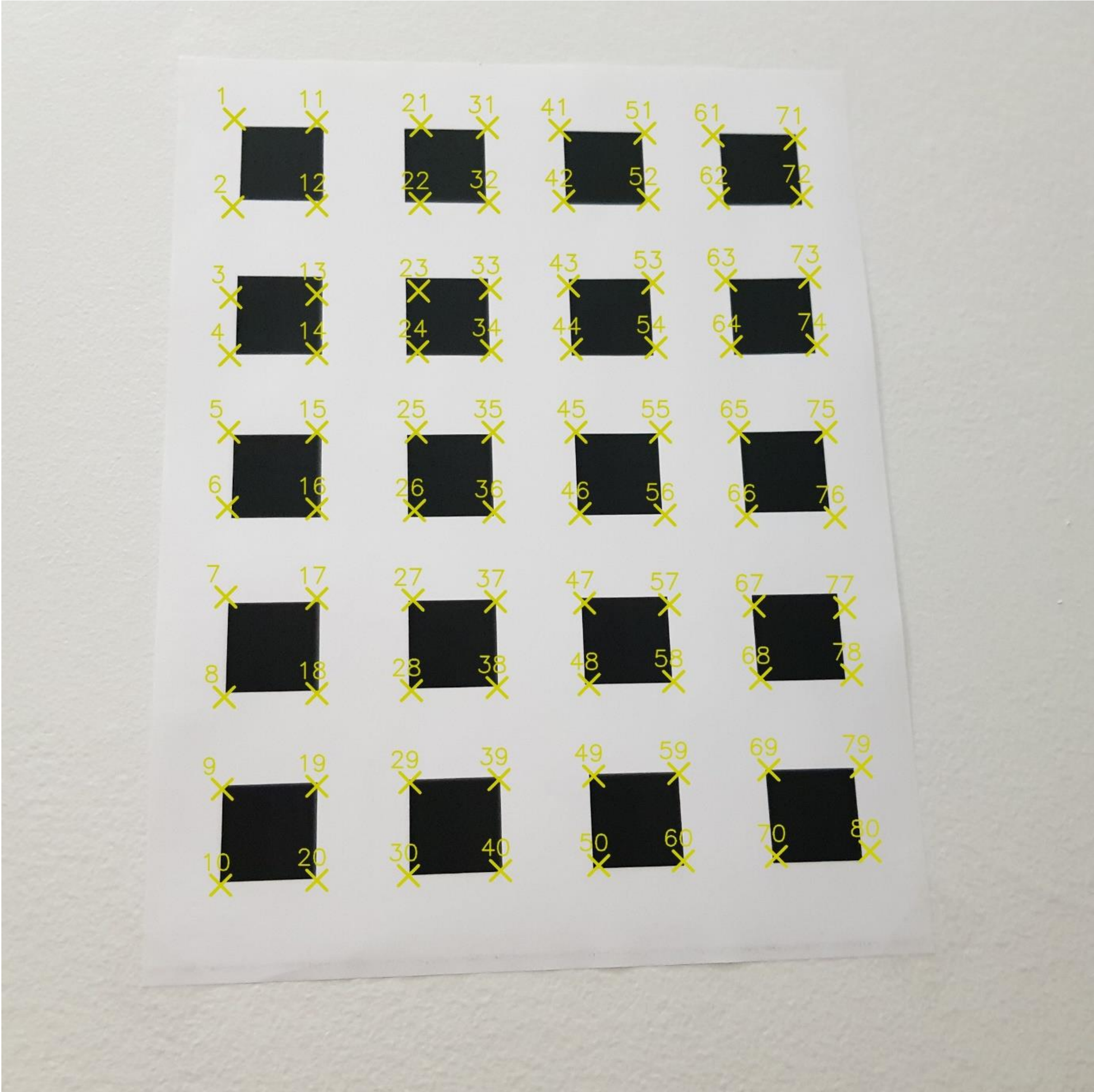

Figure 35: Detected Corners for Pic_22

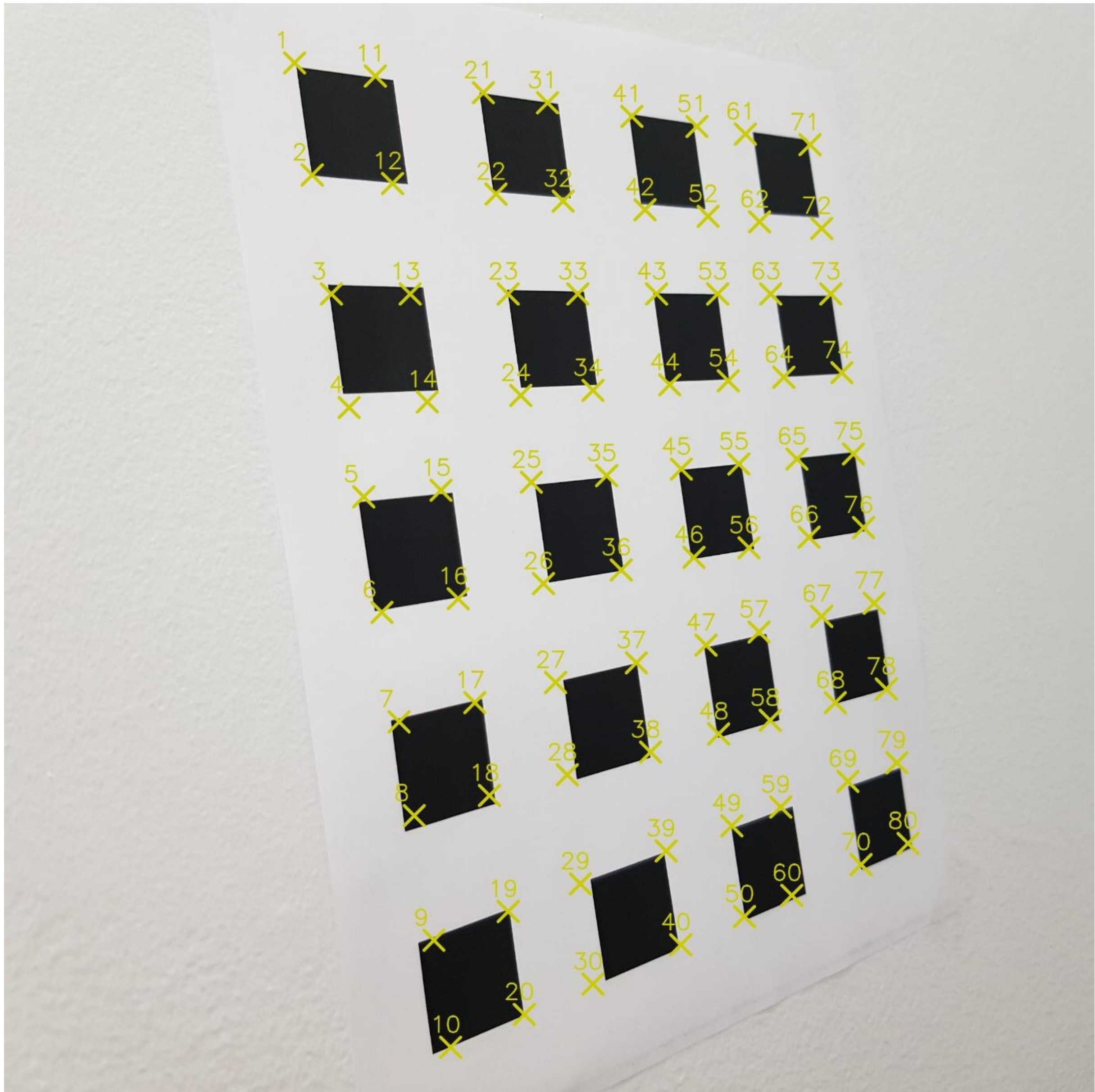

Figure 36: Detected Corners for Pic_34

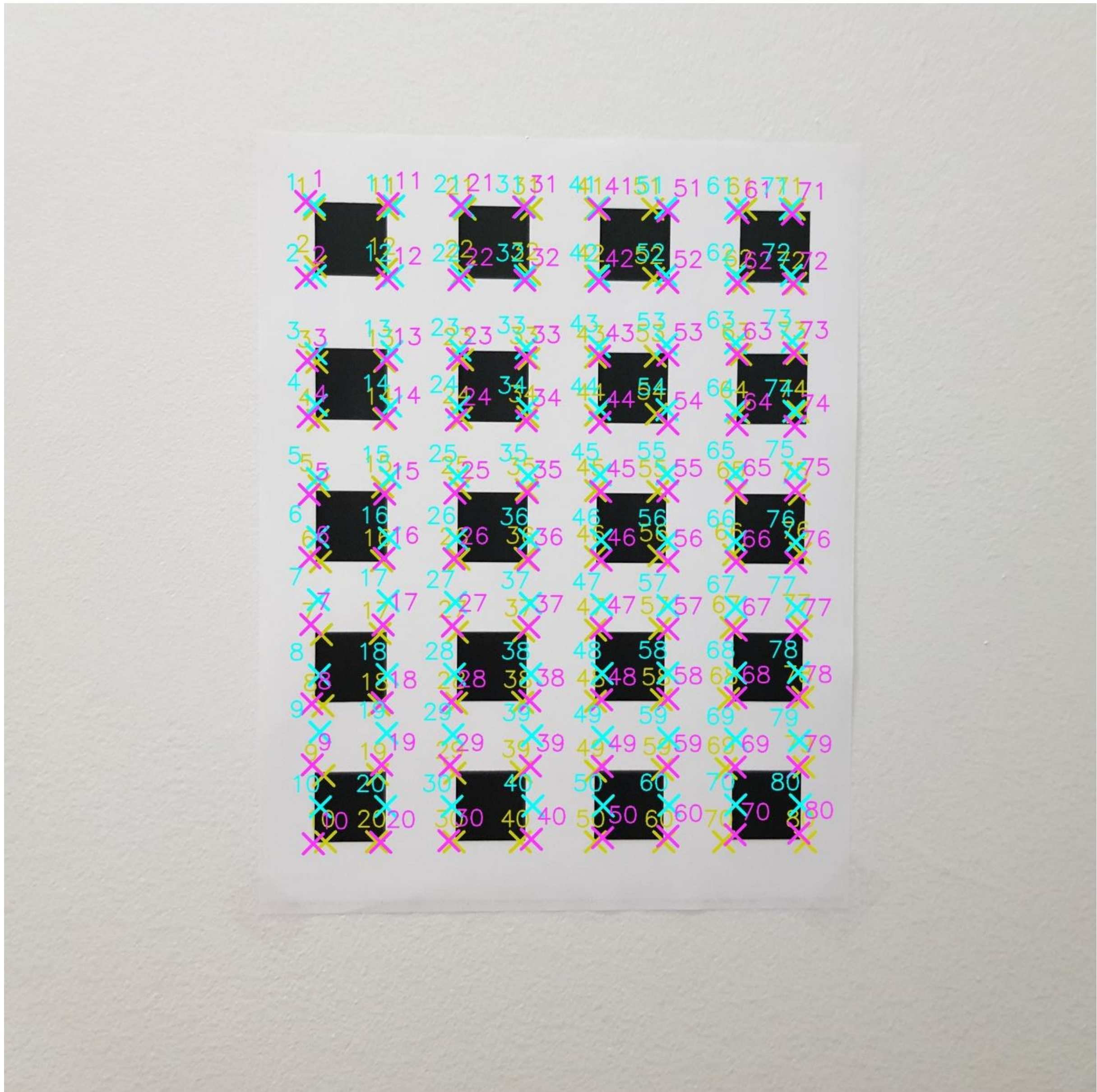

Figure 37: Labels and points for Pic_3 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

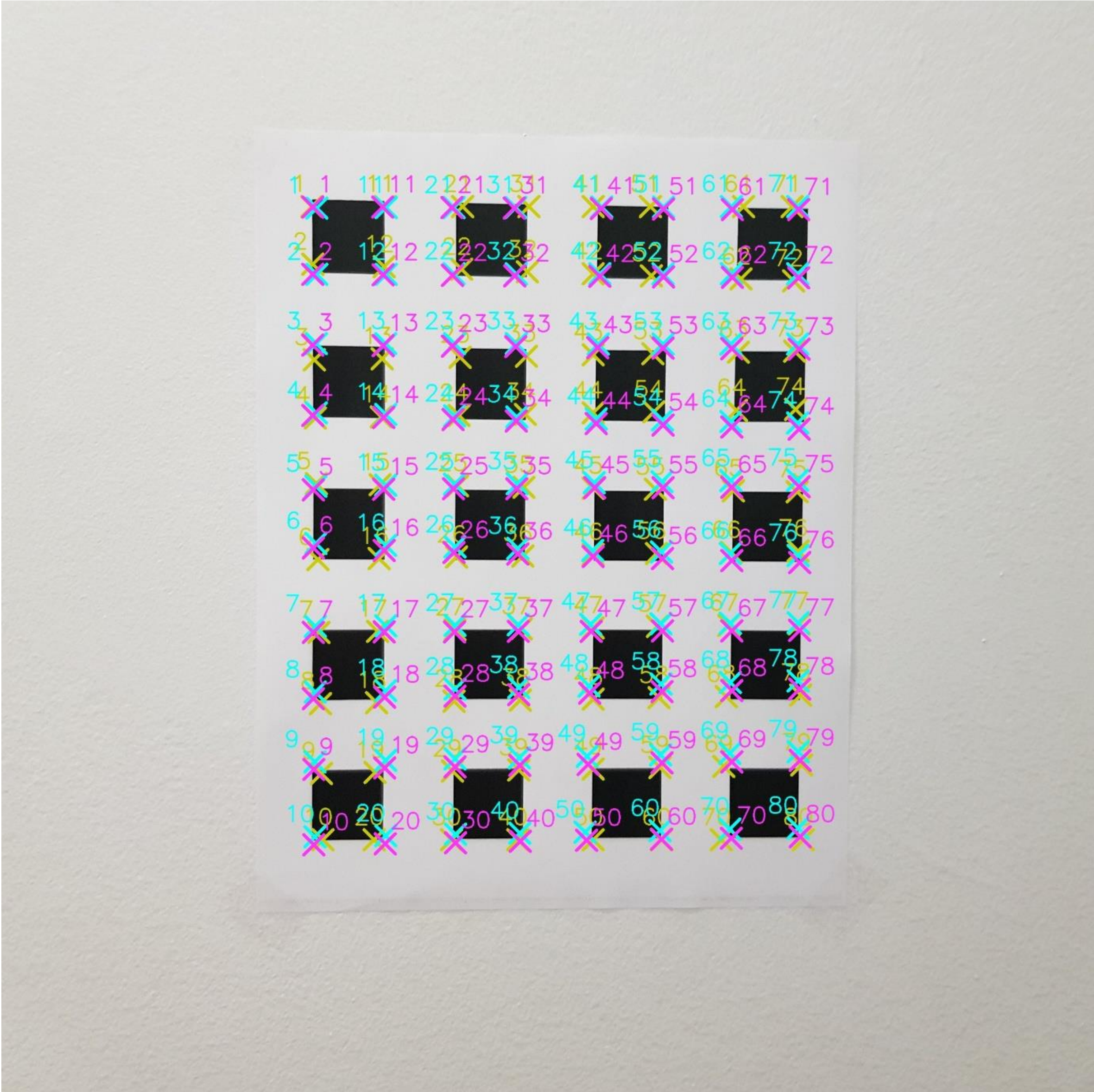

Figure 38: Labels and points for Pic_11 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

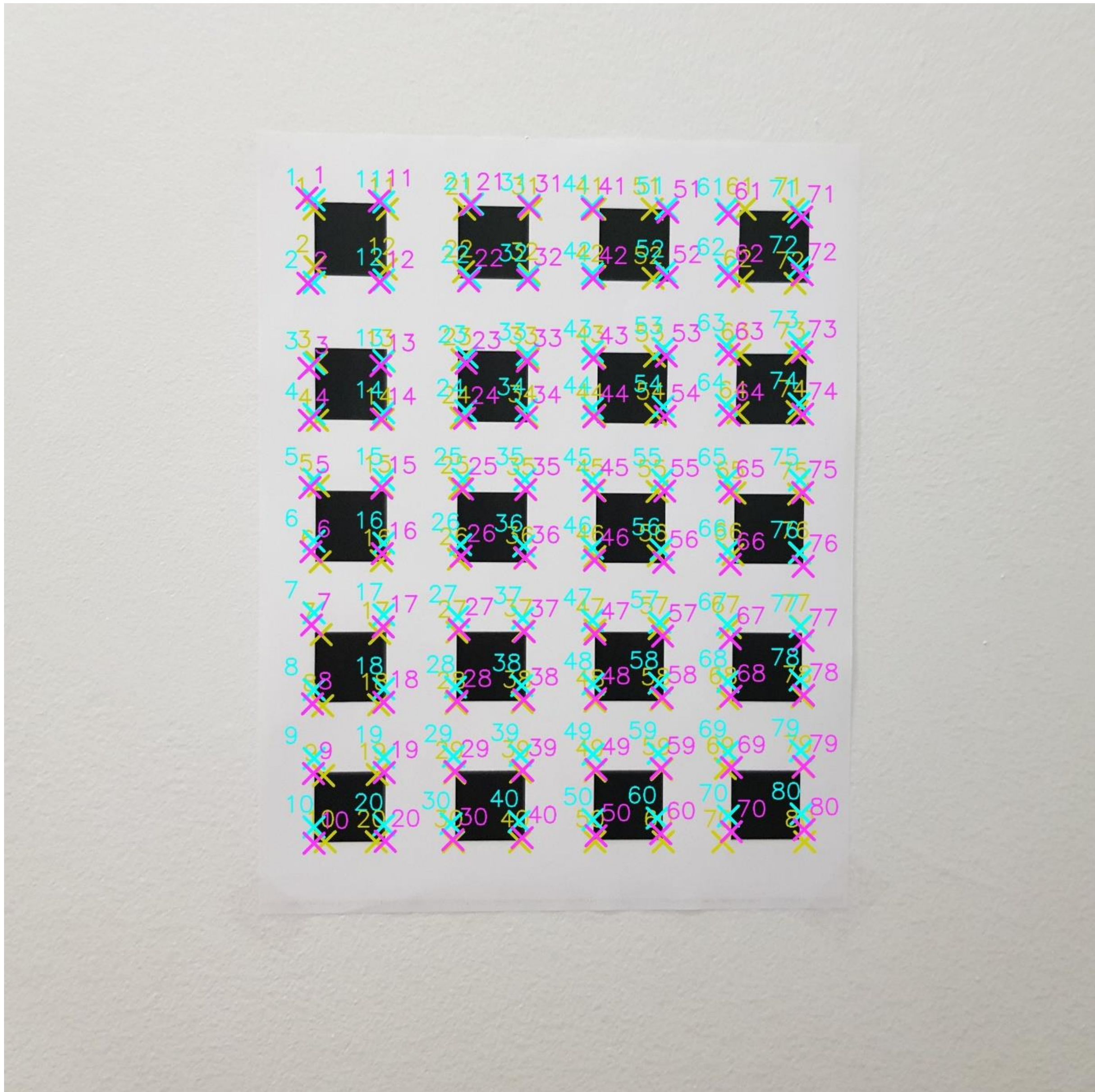

Figure 39: Labels and points for Pic_22 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

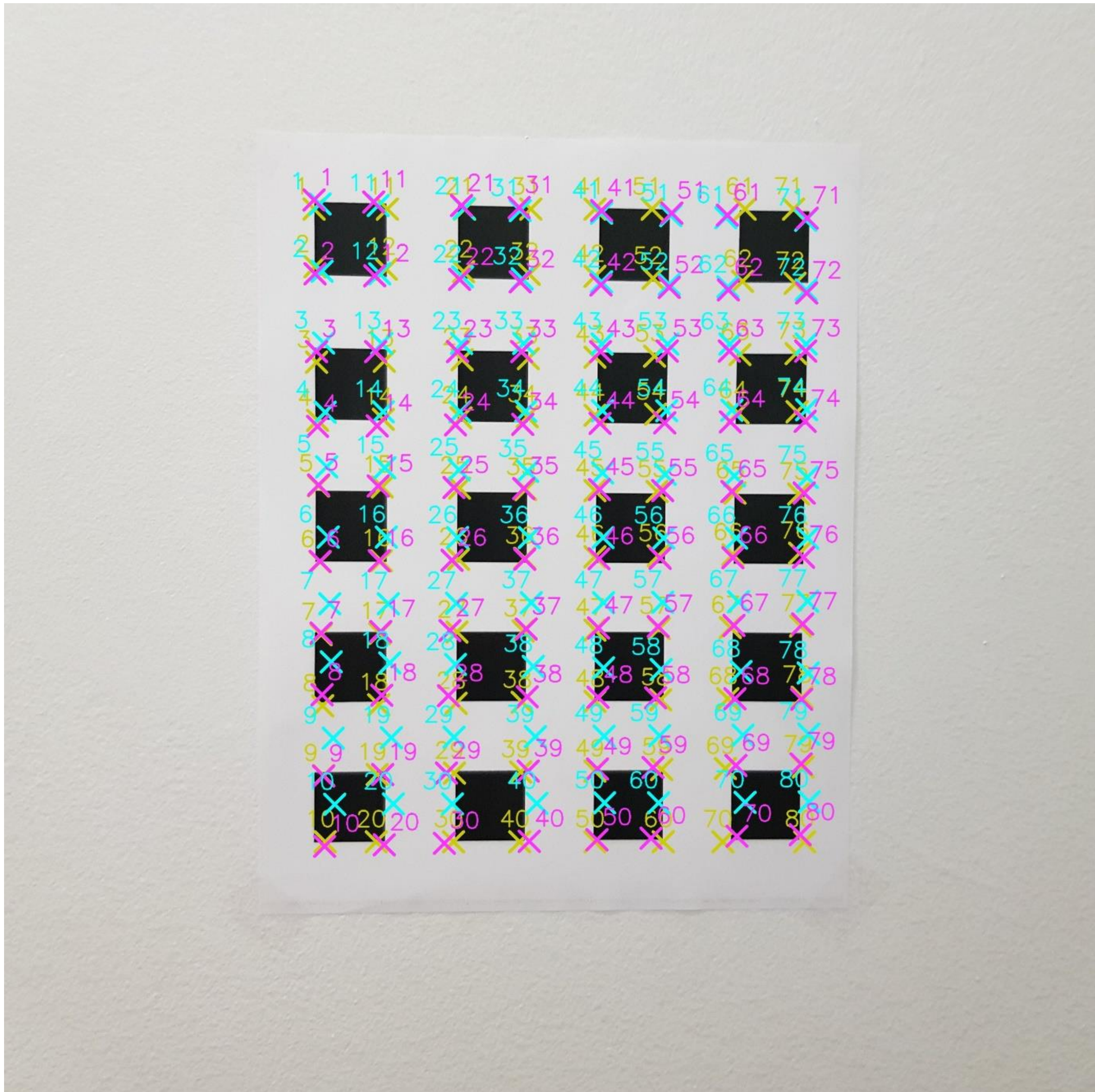

Figure 40: Labels and points for Pic_34 projected on the fixed image. Yellow is the determined actual point in the fixed image, blue is the projection without LM and pink is with LM.

We can see from the above four images that using LM optimization improves the performance significantly. Some of the LM projected points (pink) are more accurate than the original estimated points (yellow). This is the case because we use the real-world measurements for LM.

If we look at the improvement in terms of mean and variance of the distance between the actual point (yellow) and the original or refined projection respectively we observe the following values:

	Pic_3	**Pic_11**	**Pic_22**	**Pic_34**
mean original	39.1883	19.2907	25.1325	45.1577
mean refined	15.4940	15.7037	13.8100	15.6594
variance original	523.5310	64.7211	153.3002	535.6311
variance refined	49.0974	56.0464	83.6291	59.0080

Table 4: mean and variance before and after LM optimization

Those results are not as good as for Dataset1. This can be attributed to the less precise determination of the actual corners (yellow). In many cases the LM optimized projection finds the actual corner of the calibration pattern while the "ground truth" corner (yellow) is not at the actual corner.

We can observe the following intrinsic camera parameters:

$$\boldsymbol{K} = \begin{bmatrix} 2535.71 & -21.1311 & 932.697 \\ 0 & 2529.42 & 1020.42 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{K}_{refined} = \begin{bmatrix} 2346.61 & -2.30911 & 1018.32 \\ 0 & 2357.38 & 1063.45 \\ 0 & 0 & 1 \end{bmatrix}$$

For the respective images we get the following extrinsic camera parameters:

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_3} = \begin{bmatrix} -0.943117 & 0.0131554 & -0.332199 & 45.8117 \\ -0.0345071 & -0.997693 & 0.0584562 & 106.958 \\ -0.330664 & 0.0665943 & 0.941396 & -300.511 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_3,refined} = \begin{bmatrix} -0.959891 & 0.0145018 & -0.279997 & 56.7705 \\ -0.0334633 & -0.997449 & 0.0630593 & 116.783 \\ -0.278368 & 0.0698997 & 0.957928 & -294.63 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_11} = \begin{bmatrix} -0.99799 & -0.00665256 & -0.0630181 & 59.6954 \\ -0.00832636 & -0.972085 & 0.23448 & 106.09 \\ -0.0628189 & 0.234533 & 0.970076 & -435.968 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_11,refined} = \begin{bmatrix} -0.998212 & -0.00680424 & -0.0593895 & 73.7515 \\ -0.00648218 & -0.975322 & 0.220694 & 113.067 \\ -0.0594255 & 0.220684 & 0.973533 & -404.841 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_22} = \begin{bmatrix} 0.995623 & 0.0335912 & -0.0872168 & -73.5515 \\ -0.0108181 & 0.968326 & 0.249453 & -121.055 \\ 0.0928338 & -0.247418 & 0.964451 & 398.129 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_22,refined} = \begin{bmatrix} 0.997128 & 0.0319915 & -0.0686398 & -86.7317 \\ -0.0141493 & 0.96913 & 0.246143 & -129.044 \\ 0.0743954 & -0.244465 & 0.9668 & 375.189 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_34} = \begin{bmatrix} 0.783845 & 0.140589 & -0.604832 & -40.518 \\ -0.108921 & 0.99006 & 0.0889749 & -98.2121 \\ 0.611329 & -0.00386377 & 0.791367 & 279.856 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_34,refined} = \begin{bmatrix} 0.808116 & 0.144112 & -0.571121 & -51.89 \\ -0.111714 & 0.989508 & 0.0916137 & -108.782 \\ 0.578332 & -0.0102326 & 0.815737 & 278.037 \end{bmatrix}$$

Pic_25 is the fixed image. Look at its matrix as well.

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_25} = \begin{bmatrix} 0.999462 & 0.00437817 & -0.032513 & -58.3794 \\ -0.00415633 & 0.999968 & 0.00688779 & -114.027 \\ 0.0325421 & -0.00674895 & 0.999448 & 469.209 \end{bmatrix}$$

$$[\boldsymbol{R}|\boldsymbol{t}]_{Pic_25,refined} = \begin{bmatrix} 0.999051 & 0.000124016 & -0.0435554 & -73.8912 \\ -0.000204078 & 0.999998 & -0.00183372 & -119.962 \\ 0.0435551 & 0.00184087 & 0.999049 & 428.639 \end{bmatrix}$$

R is very close to the identity matrix. Therefore, we can say that the fixed image is indeed the fixed image.

Now look at the optimization we can get by considering the radial distortion.

For Dataset 1 we get

$$k_1 = -2.824743111562335e - 07$$

$$k_2 = 1.5362098950755817e - 12$$

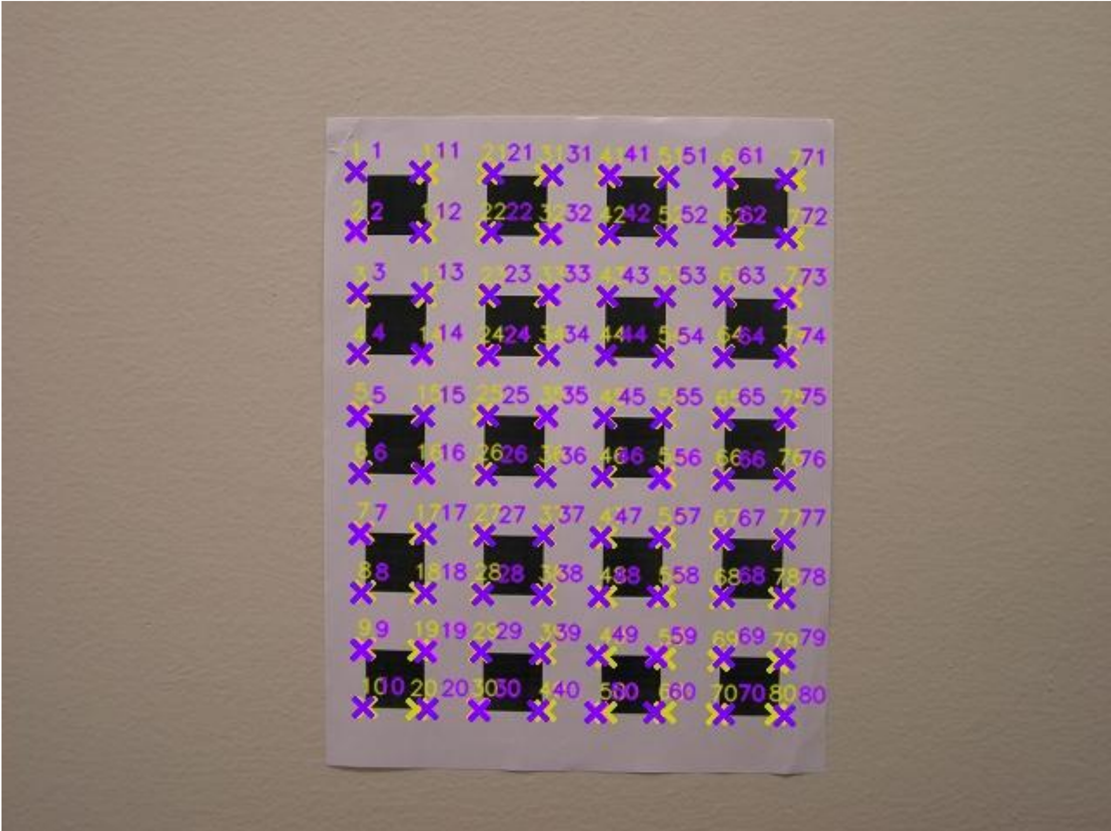

Figure 41: Pic_2 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

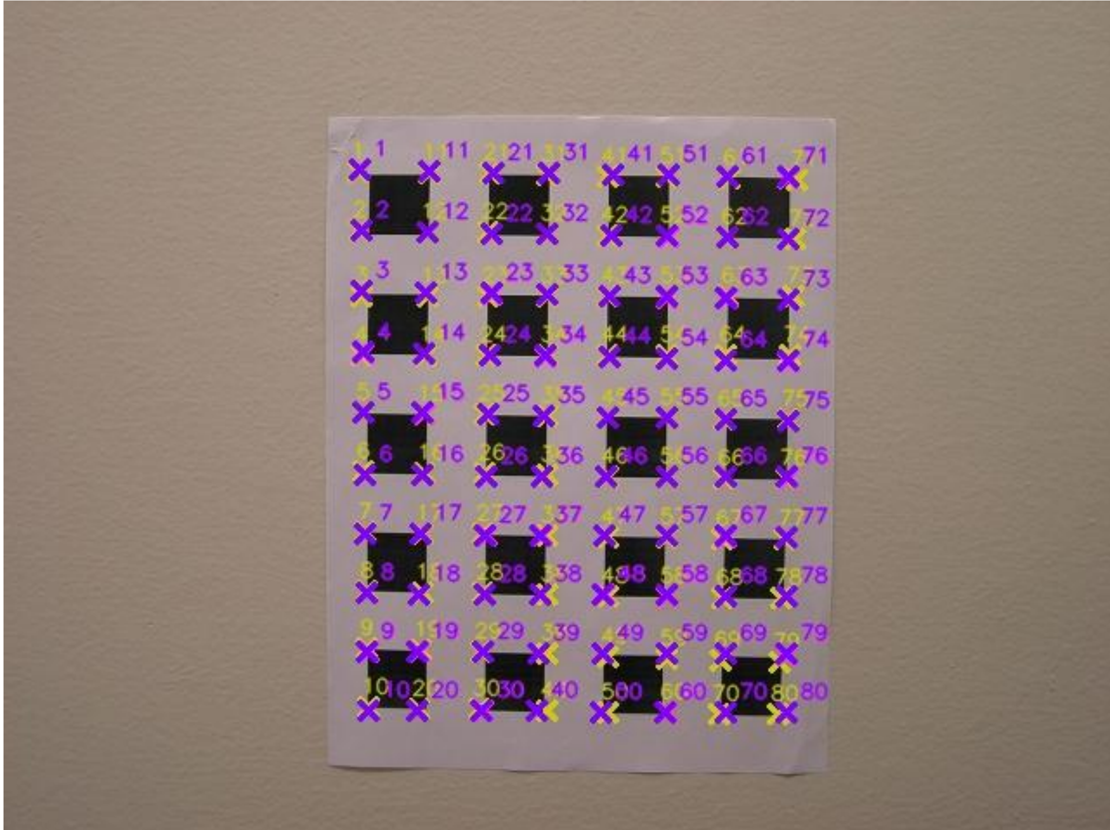

Figure 42: Pic_4 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

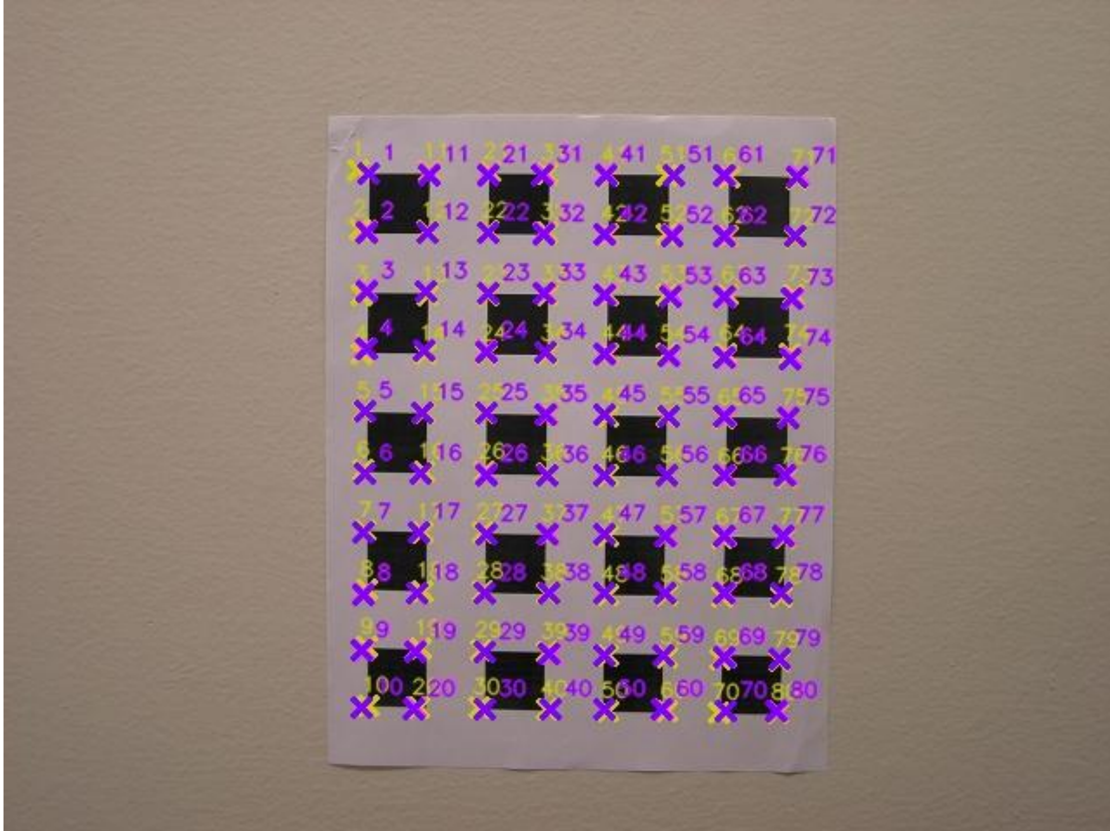

Figure 43: Pic_18 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

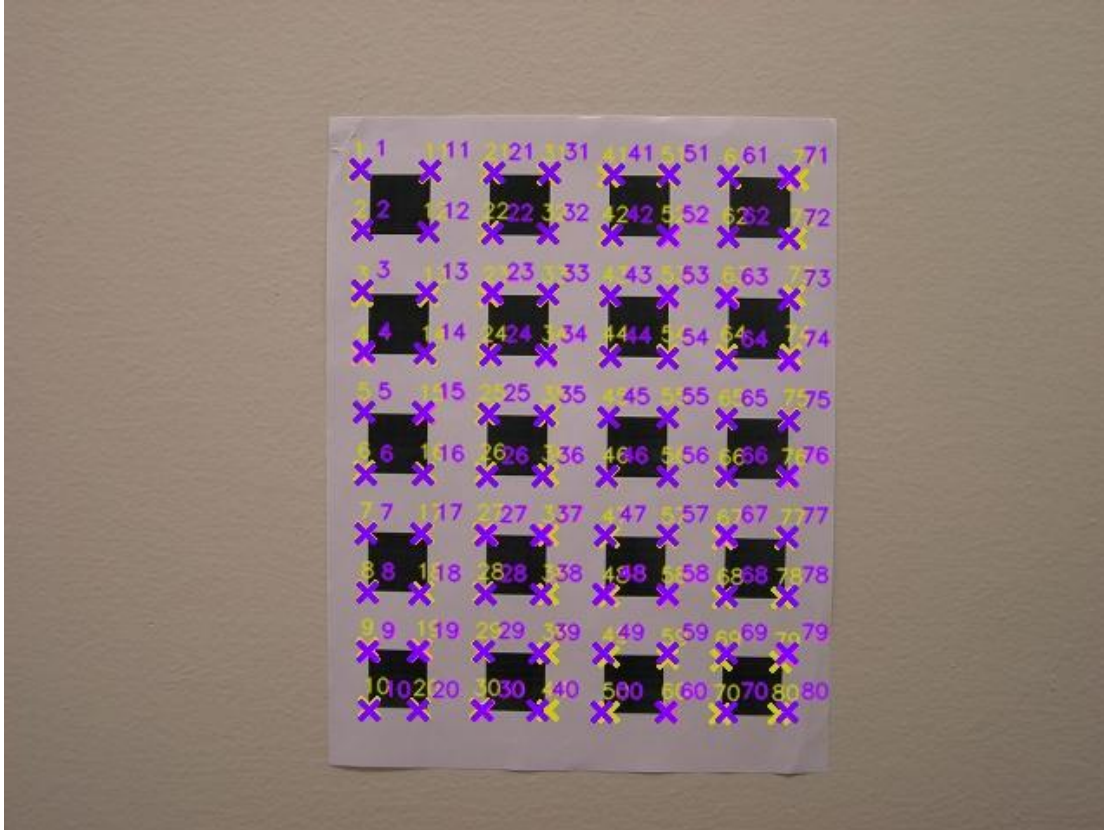

Figure 44: Pic_4 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

	Pic_2	Pic_4	Pic_18	Pic_40
mean refined	2.3811	2.0079	1.9480	1.7185
mean rad. dist.	2.3849	2.0265	1.9940	1.7181
variance refined	1.9027	1.7366	0.9626	1.1319
variance rad. dist.	1.9656	1.8167	1.0280	1.1091

Table 5: mean and variance after LM optimization with and without considering radial distortion

Visually the results are better when considering radial distortion. The increase in mean and variance for most images is related to the ground truth being estimated with the corner detection algorithm. It can be seen easily that the yellow markers are slightly off their actual position. For most images the change due to radial distortion is very small.

For Dataset 2 we get

$$k_1 = 2.4953070414347955e - 08$$

$$k_2 = -1.1199186587846307e - 14$$

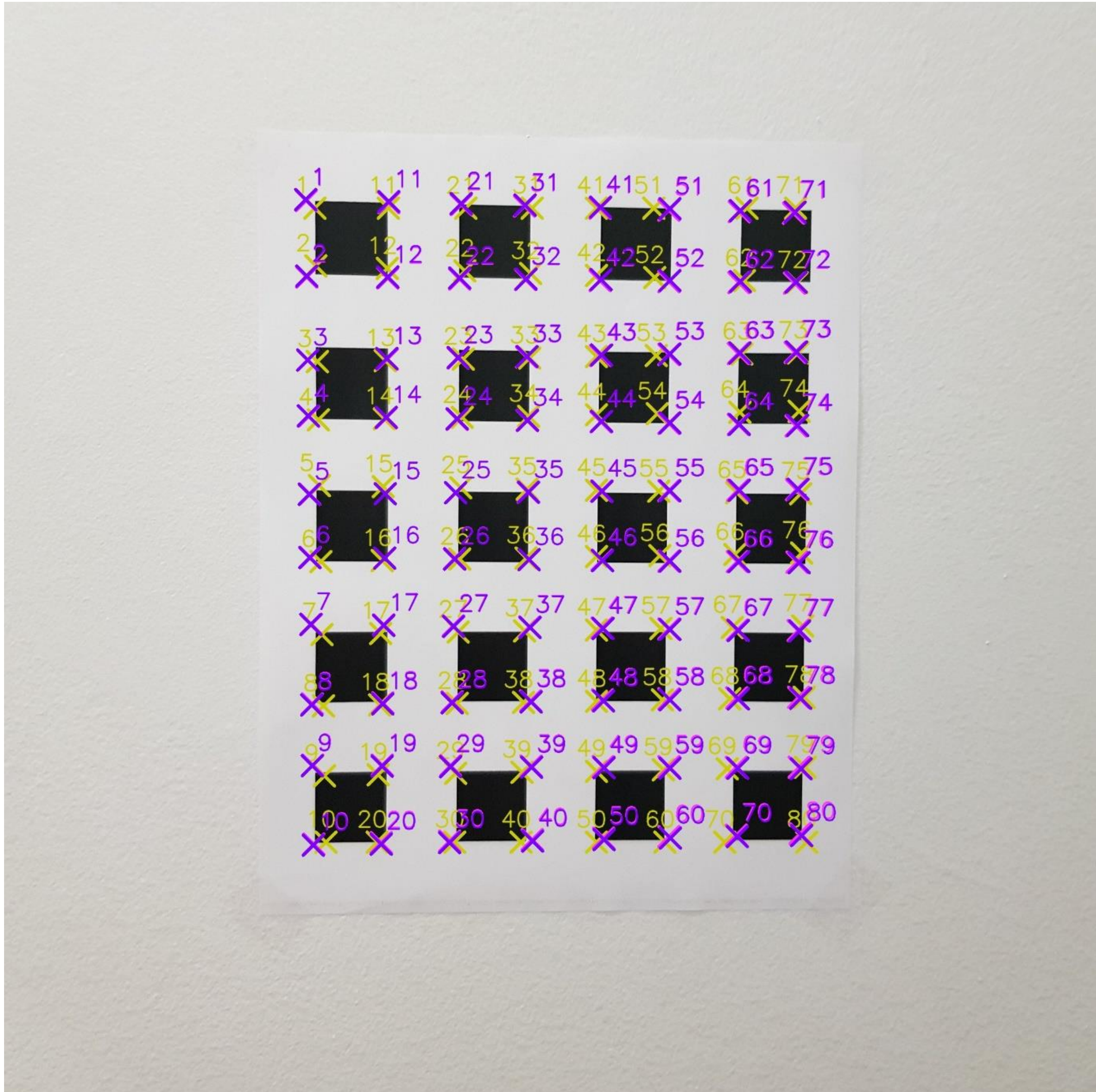

Figure 45: Pic_3 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

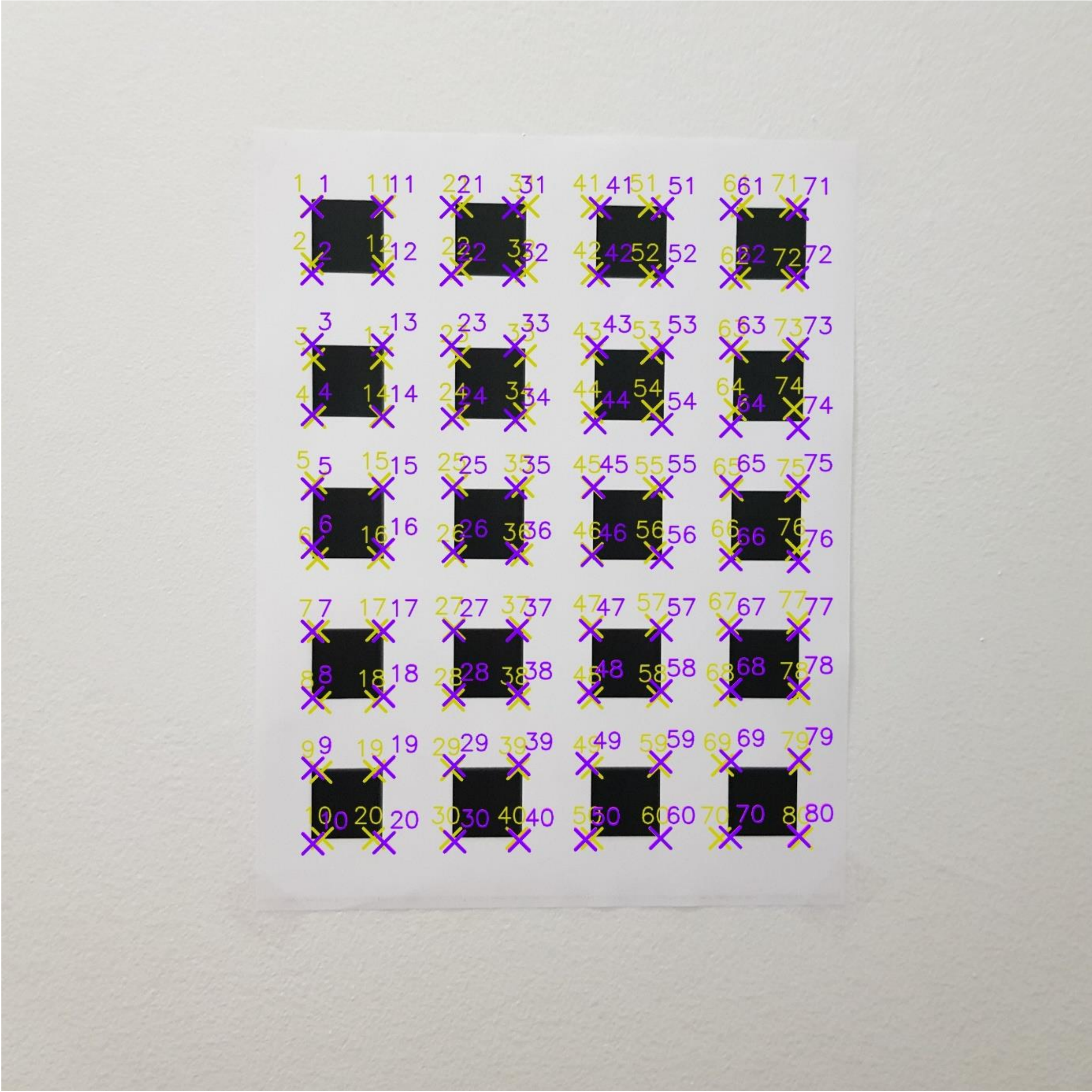

Figure 46: Pic_11 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

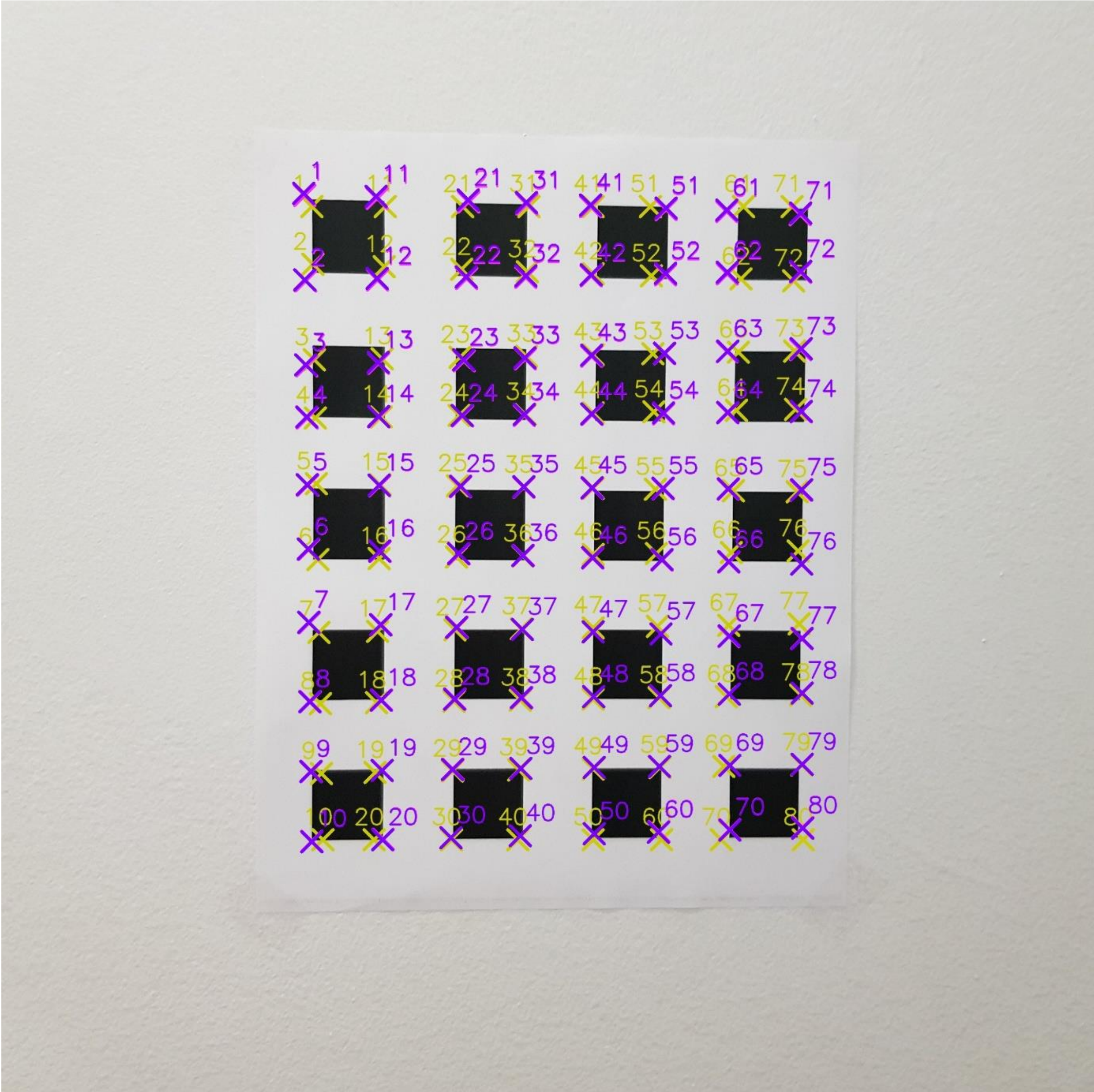

Figure 47: Pic_22 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

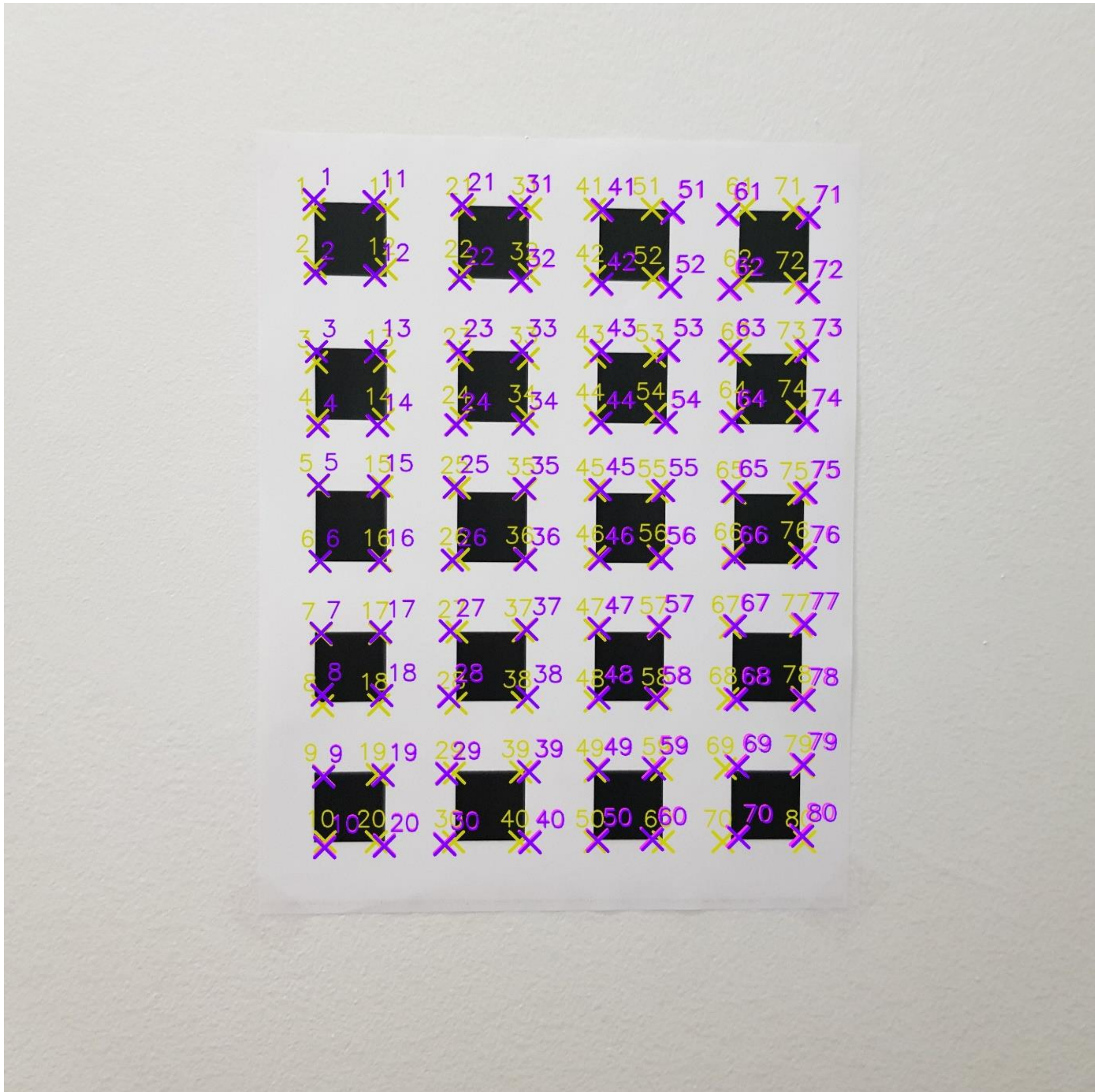

Figure 48: Pic_34 with ground truth in yellow, original LM in pink and the LM improved with radial distortion in purple

	Pic_3	**Pic_11**	**Pic_22**	**Pic_34**
mean refined	15.4940	15.7037	13.8100	15.6594
mean rad. dist.	15.4858	15.7260	13.4600	14.3863
variance refined	49.0974	56.0464	83.6291	59.0080
variance rad. dist.	54.1140	59.6482	86.2602	63.7995

Table 6: mean and variance after LM optimization with and without considering radial distortion

In this dataset the ground truth data is more inaccurate than in the first dataset. It can be confirmed visually that considering radial distortion causes slight improvements.

Part 5: Code

```
import cv2

import numpy as np

from scipy.optimize import root,least_squares

from collections import defaultdict


def load_images(dataset_no):

    """

        Load the 40 images of the dataset of number dataset_no.

    """

    images = []

    for k in range(1,41):

        images.append(cv2.imread("./Files/Dataset%s/Pic_%s.jpg"%(dataset_no, k)))

    return images


def run_corner_detection(dataset_no, stretch_lines, hough_parameter):

    """

        Returns a list of the corner points of the calibration pattern determined for each image using Canny Edge Detector and the Hough
transform for the first dataset.

    """

    dataset = load_images(dataset_no)

    y_max, x_max, channels = np.shape(dataset[0])

    x_val = np.floor(x_max/2).astype(np.int)

    y_val = np.floor(y_max/2).astype(np.int)

    k = 1

    debug1 = []

    debug2 = []

    intersection_points_images = []

    for image in dataset:

#       line_template = image.copy()

#       line_template = np.ones(np.shape(image))*255

        edges = cv2.Canny(image, 245, 260)

        cv2.imwrite("./Files/Dataset%s/Canny_edges/Pic_%s.jpg"%(dataset_no,k),edges)

        hough = cv2.HoughLines(edges, 1, np.pi/180, hough_parameter)


        in_xy_coordinates_vert = []

        in_xy_coordinates_hor = []

        homogenous_lines_vert = []

        homogenous_lines_hor = []
```

```
    for line in hough:
        for rho,theta in line:
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a*rho
            y0 = b*rho
            x1 = int(x0 + stretch_lines*(-b))
            y1 = int(y0 + stretch_lines*(a))
            x2 = int(x0 - stretch_lines*(-b))
            y2 = int(y0 - stretch_lines*(a))
            if (x2-x1) == 0:
                m = np.inf
            else:
                m = (y2-y1)/(x2-x1)
            point1_hom = np.array((x1,y1,1))
            point2_hom = np.array((x2,y2,1))
            line_homogenous = np.cross(point1_hom, point2_hom)
            line_homogenous = line_homogenous/line_homogenous[2]
            xy_coords = [x1,y1,x2,y2]
            if np.abs(m) > 1:
                in_xy_coordinates_vert.append(xy_coords)
                homogenous_lines_vert.append(line_homogenous)
#                cv2.line(line_template,(x1,y1),(x2,y2),(0,0,255),1)
            else:
                in_xy_coordinates_hor.append(xy_coords)
                homogenous_lines_hor.append(line_homogenous)
#                cv2.line(line_template,(x1,y1),(x2,y2),(255,0,0),1)

    line_template2 = image.copy()
    for x1,y1,x2,y2 in in_xy_coordinates_hor:
        cv2.line(line_template2,(x1,y1),(x2,y2),(0,0,255),1)

    for x1,y1,x2,y2 in in_xy_coordinates_vert:
        cv2.line(line_template2,(x1,y1),(x2,y2),(255,0,0),1)

    cv2.imwrite("./Files/Dataset%s/All_lines/Pic_%s.jpg"%(dataset_no,k),line_template2)

    if len(in_xy_coordinates_hor) > 10:
        list_of_overlapping = []
```

```
for i in range(len(homogenous_lines_hor)-1):
    for j in range(i+1, len(homogenous_lines_hor)):
        intersection = np.cross(homogenous_lines_hor[i], homogenous_lines_hor[j])
        intersection = intersection/intersection[2]
        if intersection[1] <= y_max and intersection[1] >=0 and intersection[0]< x_max and intersection[0] >=0 or intersection[2]== 0 :
            list_of_overlapping.append(i)
list_of_overlapping = sorted(list(dict.fromkeys(list_of_overlapping)), reverse = True)
for index in list_of_overlapping:
    in_xy_coordinates_hor.pop(index)
    homogenous_lines_hor.pop(index)


while len(in_xy_coordinates_hor) > 10:
    max_distances = []
    min_distances = []
    indices = []
    for i in range(len(homogenous_lines_hor)-1):
        distances = []
        ij_comb = []
        for j in range(i+1, len(homogenous_lines_hor)):
            # Pick a point on the jth line (Take point 1)
            a1, b1, c1 = homogenous_lines_hor[j]
            x1 = x_val
            y1 = (-c1-a1*x1)/b1
            # Pick the line i
            a, b, c = homogenous_lines_hor[i]
            d = np.abs(a*x1 + b*y1 + c)/np.sqrt(np.square(a)+ np.square(b))
            distances.append(d)
            ij_comb.append((i,j))
        max_distances.append(np.max(distances))
        min_distances.append(np.min(distances))
        ij = ij_comb[np.argmax(distances)]
        indices.append(ij)
    if np.min(min_distances) < 10:
        index = np.argmin(min_distances)
        max_distances.pop(index)
        in_xy_coordinates_hor.pop(index)
        homogenous_lines_hor.pop(index)
    else:
        temp = np.argmax(max_distances)
```

```
            temp_indices = indices[temp]
            max1 = max(in_xy_coordinates_hor[temp_indices[0]][1],in_xy_coordinates_hor[temp_indices[0]][3])
            max2 = max(in_xy_coordinates_hor[temp_indices[1]][1],in_xy_coordinates_hor[temp_indices[1]][3])
            if max1 < max2:
                index = temp_indices[1]
            else:
                index = temp_indices[0]
            max_distances.pop(index)
            in_xy_coordinates_hor.pop(index)
            homogenous_lines_hor.pop(index)

    if len(in_xy_coordinates_vert) > 8:
        list_of_overlapping = []
        for i in range(len(homogenous_lines_vert)-1):
            for j in range(i+1, len(homogenous_lines_vert)):
                intersection = np.cross(homogenous_lines_vert[i], homogenous_lines_vert[j])
                intersection = intersection/intersection[2]
                if intersection[1] <= y_max and intersection[1] >=0 and intersection[0]< x_max and intersection[0] >=0 or intersection[2] == 0:
                    list_of_overlapping.append(i)
        list_of_overlapping = sorted(list(dict.fromkeys(list_of_overlapping)), reverse = True)
        for index in list_of_overlapping:
            in_xy_coordinates_vert.pop(index)
            homogenous_lines_vert.pop(index)


        while len(in_xy_coordinates_vert) > 8:
            max_distances = []
            min_distances = []
            indices = []
            for i in range(len(homogenous_lines_vert)-1):
                distances = []
                ij_comb = []
                for j in range(i+1, len(homogenous_lines_vert)):
                    # Pick a point on the jth line (Take point 1)
                    a1, b1, c1 = homogenous_lines_vert[j]
                    y1 = y_val
                    x1 = (-c1-b1*y1)/a1
                    # Pick the line i
                    a, b, c = homogenous_lines_vert[i]
                    d = np.abs(a*x1 + b*y1 + c)/np.sqrt(np.square(a)+ np.square(b))
```

```
            distances.append(d)
            ij_comb.append((i,j))
        max_distances.append(np.max(distances))
        min_distances.append(np.min(distances))
        ij = ij_comb[np.argmax(distances)]
        indices.append(ij)
    if np.min(min_distances) < 10:
        index = np.argmin(min_distances)
        max_distances.pop(index)
        in_xy_coordinates_vert.pop(index)
        homogenous_lines_vert.pop(index)
    else:
        temp = np.argmax(max_distances)
        temp_indices = indices[temp]
        max1 = max(in_xy_coordinates_vert[temp_indices[0]][0],in_xy_coordinates_vert[temp_indices[0]][2])
        max2 = max(in_xy_coordinates_vert[temp_indices[1]][0],in_xy_coordinates_vert[temp_indices[1]][2])
        if max1 < max2:
            index = temp_indices[1]
        else:
            index = temp_indices[0]
        max_distances.pop(index)
        in_xy_coordinates_vert.pop(index)
        homogenous_lines_vert.pop(index)


line_template = image.copy()
for x1,y1,x2,y2 in in_xy_coordinates_hor:
    cv2.line(line_template,(x1,y1),(x2,y2),(0,0,255),1)

for x1,y1,x2,y2 in in_xy_coordinates_vert:
    cv2.line(line_template,(x1,y1),(x2,y2),(255,0,0),1)

debug1.append(len(in_xy_coordinates_hor))
debug2.append(len(in_xy_coordinates_vert))

cv2.imwrite("./Files/Dataset%s/Lines_Hough/Pic_%s.jpg"%(dataset_no,k),line_template)

point_template = image.copy()
point_template2 = point_template.copy()
```

```
intersection_points = []

ys = []
for l1 in homogenous_lines_hor:
    a1, b1, c1 = l1
    x1 = x_val
    y1 = (-c1-a1*x1)/b1
    ys.append(y1)
ys = np.array(ys).reshape(-1,1)
array_homogenous_lines_hor = np.array(homogenous_lines_hor)
temp = np.append(array_homogenous_lines_hor,ys, axis = 1)
temp_sorted = temp[temp[:,3].argsort()]
ys_sorted = temp_sorted[:,3]
homogenous_lines_hor_sorted = list(temp_sorted[:,:3])

xs = []
for l2 in homogenous_lines_vert:
    a2, b2, c2 = l2
    y2 = y_val
    x2 = (-c2-b2*y2)/a2
    xs.append(x2)
xs = np.array(xs).reshape(-1,1)
array_homogenous_lines_vert = np.array(homogenous_lines_vert)
temp = np.append(array_homogenous_lines_vert,xs, axis = 1)
temp_sorted = temp[temp[:,3].argsort()]
xs_sorted = temp_sorted[:,3]
homogenous_lines_vert_sorted = list(temp_sorted[:,:3])

colornumber = 0
for l1 in homogenous_lines_vert_sorted:
    # inner loop correct, only outer has issues
    for l2 in homogenous_lines_hor_sorted:
        intersect = np.cross(l1,l2)
        intersect = intersect/intersect[2]
        intersection_points.append(intersect[:2])
        cv2.drawMarker(point_template,(np.round(intersect[0]).astype(np.int),np.round(intersect[1]).astype(np.int)), (0,204,204),1, 10,
2)
        cv2.putText(point_template, "%s"%(colornumber+1), (np.round(intersect[0]-4).astype(np.int),np.round(intersect[1]-
8).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  0.4, (0,204,204),  1, cv2.LINE_AA)
```

```
            colornumber = colornumber+1
        cv2.imwrite("./Files/Dataset%s/Corners/Pic_%s.jpg"%(dataset_no,k),point_template)
        k = k + 1
        intersection_points_images.append(intersection_points)
    return intersection_points_images


def run_corner_detection_2(dataset_no, stretch_lines, hough_parameter):
    """
        Returns a list of the corner points of the calibration pattern determined for each image using Canny Edge Detector and the Hough
transform for the second dataset.
    """
    dataset = load_images(dataset_no)
    y_max, x_max, channels = np.shape(dataset[0])
    x_val = np.floor(x_max/2).astype(np.int)
    y_val = np.floor(y_max/2).astype(np.int)
    k = 1
    debug1 = []
    debug2 = []
    intersection_points_images = []
    for image in dataset:
#     line_template = image.copy()
#     line_template = np.ones(np.shape(image))*255
        edges = cv2.Canny(image, 245, 260)
        cv2.imwrite("./Files/Dataset%s/Canny_edges/Pic_%s.jpg"%(dataset_no,k),edges)
        hough = cv2.HoughLines(edges, 1, np.pi/180, hough_parameter)


        in_xy_coordinates_vert = []
        in_xy_coordinates_hor = []
        homogenous_lines_vert = []
        homogenous_lines_hor = []
        slope_vert = []
        slope_hor = []
        for line in hough:
            for rho,theta in line:
                a = np.cos(theta)
                b = np.sin(theta)
                x0 = a*rho
                y0 = b*rho
                x1 = int(x0 + stretch_lines*(-b))
```

```
            y1 = int(y0 + stretch_lines*(a))
            x2 = int(x0 - stretch_lines*(-b))
            y2 = int(y0 - stretch_lines*(a))
            if (x2-x1) == 0:
                m = np.inf
            else:
                m = (y2-y1)/(x2-x1)
            point1_hom = np.array((x1,y1,1))
            point2_hom = np.array((x2,y2,1))
            line_homogenous = np.cross(point1_hom, point2_hom)
            line_homogenous = line_homogenous/line_homogenous[2]
            xy_coords = [x1,y1,x2,y2]
            if np.abs(m) > 1:
                in_xy_coordinates_vert.append(xy_coords)
                homogenous_lines_vert.append(line_homogenous)
                slope_vert.append(m)
#               cv2.line(line_template,(x1,y1),(x2,y2),(0,0,255),1)
            else:
                in_xy_coordinates_hor.append(xy_coords)
                homogenous_lines_hor.append(line_homogenous)
                slope_hor.append(m)
#               cv2.line(line_template,(x1,y1),(x2,y2),(255,0,0),1)

    line_template2 = image.copy()
    for x1,y1,x2,y2 in in_xy_coordinates_hor:
        cv2.line(line_template2,(x1,y1),(x2,y2),(0,0,255),1)

    for x1,y1,x2,y2 in in_xy_coordinates_vert:
        cv2.line(line_template2,(x1,y1),(x2,y2),(255,0,0),1)

    cv2.imwrite("./Files/Dataset%s/All_lines/Pic_%s.jpg"%(dataset_no,k),line_template2)
    if len(in_xy_coordinates_hor) > 10:
        all_overlaps = []
        for i in range(len(homogenous_lines_hor)-1):
            list_of_overlapping = [i]
            for j in range(i, len(homogenous_lines_hor)):
                intersection = np.cross(homogenous_lines_hor[i], homogenous_lines_hor[j])
                intersection = intersection/intersection[2]
                if intersection[1] <= y_max and intersection[1] >=0 and intersection[0]< x_max and intersection[0] >=0 or intersection[2]== 0 :
```

```
                list_of_overlapping.append(j)
            if len(list_of_overlapping) > 1:
                all_overlaps.append(list_of_overlapping)

        overlaps_wo_duplicates = list(merge_common(all_overlaps))

        indices_of_median = []
        for index in range(len(overlaps_wo_duplicates)):
            sublist_overlapping = overlaps_wo_duplicates[index]
            if len(sublist_overlapping)%2 == 0:
                median_of_slope_index = sublist_overlapping[np.floor(len(sublist_overlapping)/2).astype(np.int)-1]
            else:
                median_of_slope_index = sublist_overlapping[np.floor(len(sublist_overlapping)/2).astype(np.int)]
            indices_of_median.append(median_of_slope_index)

        final_lines_hor= []
        final_xy_coords_hor = []
        for overlap_ind in range(len(overlaps_wo_duplicates)):
            overlap = overlaps_wo_duplicates[overlap_ind]
            index = indices_of_median[overlap_ind]
            line = homogenous_lines_hor[index]
            final_lines_hor.append(line)
            xy_coord = in_xy_coordinates_hor[index]
            final_xy_coords_hor.append(xy_coord)

        # if bottom edge of paper detected
        if len(final_lines_hor)>10:
            ymax = -np.inf
            for index in range(len(final_lines_hor)):
                a1, b1, c1 = final_lines_hor[index]
                x1 = x_val
                y1 = (-c1-a1*x1)/b1
                if y1 > ymax:
                    ymax = y1
                    pos = index
            final_lines_hor.pop(pos)
            final_xy_coords_hor.pop(pos)

    if len(in_xy_coordinates_vert) > 8:
```

```
        all_overlaps = []
        for i in range(len(homogenous_lines_vert)-1):
            list_of_overlapping = [i]
            for j in range(i, len(homogenous_lines_vert)):
                intersection = np.cross(homogenous_lines_vert[i], homogenous_lines_vert[j])
                intersection = intersection/intersection[2]
                if intersection[1] <= y_max and intersection[1] >=0 and intersection[0]< x_max and intersection[0] >=0 or intersection[2]== 0 :
                    list_of_overlapping.append(j)
            if len(list_of_overlapping) > 1:
                all_overlaps.append(list_of_overlapping)

        overlaps_wo_duplicates = list(merge_common(all_overlaps))

        indices_of_median = []
        for index in range(len(overlaps_wo_duplicates)):
            sublist_overlapping = overlaps_wo_duplicates[index]
            if len(sublist_overlapping)%2 == 0:
                median_of_slope_index = sublist_overlapping[np.floor(len(sublist_overlapping)/2).astype(np.int)-1]
            else:
                median_of_slope_index = sublist_overlapping[np.floor(len(sublist_overlapping)/2).astype(np.int)]
            indices_of_median.append(median_of_slope_index)

        final_lines_vert= []
        final_xy_coords_vert = []
        for overlap_ind in range(len(overlaps_wo_duplicates)):
            index = indices_of_median[overlap_ind]
            line = homogenous_lines_vert[index]
            final_lines_vert.append(line)
            xy_coord = in_xy_coordinates_vert[index]
            final_xy_coords_vert.append(xy_coord)




    line_template = image.copy()
    for x1,y1,x2,y2 in final_xy_coords_hor:
        cv2.line(line_template,(x1,y1),(x2,y2),(0,0,255),1)
#
    for x1,y1,x2,y2 in final_xy_coords_vert:
```

```
        cv2.line(line_template,(x1,y1),(x2,y2),(255,0,0),1)
#
    debug1.append(len(final_xy_coords_hor))
    debug2.append(len(final_xy_coords_vert))
#
    cv2.imwrite("./Files/Dataset%s/Lines_Hough/Pic_%s.jpg"%(dataset_no,k),line_template)
#
    point_template = image.copy()
    intersection_points = []

    ys = []
    for l1 in final_lines_hor:
        a1, b1, c1 = l1
        x1 = x_val
        y1 = (-c1-a1*x1)/b1
        ys.append(y1)
    ys = np.array(ys).reshape(-1,1)
    array_homogenous_lines_hor = np.array(final_lines_hor)
    temp = np.append(array_homogenous_lines_hor,ys, axis = 1)
    temp_sorted = temp[temp[:,3].argsort()]
    ys_sorted = temp_sorted[:,3]
    homogenous_lines_hor_sorted = list(temp_sorted[:,:3])

    xs = []
    for l2 in final_lines_vert:
        a2, b2, c2 = l2
        y2 = y_val
        x2 = (-c2-b2*y2)/a2
        xs.append(x2)
    xs = np.array(xs).reshape(-1,1)
    array_homogenous_lines_vert = np.array(final_lines_vert)
    temp = np.append(array_homogenous_lines_vert,xs, axis = 1)
    temp_sorted = temp[temp[:,3].argsort()]
    xs_sorted = temp_sorted[:,3]
    homogenous_lines_vert_sorted = list(temp_sorted[:,:3])

    colornumber = 0
    for l1 in homogenous_lines_vert_sorted:
        # inner loop correct, only outer has issues
```

```
            for l2 in homogenous_lines_hor_sorted:

                intersect = np.cross(l1,l2)

                intersect = intersect/intersect[2]

                intersection_points.append(intersect[:2])

                cv2.drawMarker(point_template,(np.round(intersect[0]).astype(np.int),np.round(intersect[1]).astype(np.int)), (0,204,204),1, 40, 
5)

                cv2.putText(point_template, "%s"%(colornumber+1), (np.round(intersect[0]-40).astype(np.int),np.round(intersect[1]-
30).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  1.5, (0,204,204),  2, cv2.LINE_AA)

                colornumber = colornumber+1

        cv2.imwrite("./Files/Dataset%s/Corners/Pic_%s.jpg"%(dataset_no,k),point_template)

        intersection_points_images.append(intersection_points)

        k = k+1

    return intersection_points_images




def linear_least_squares_homography(list_of_intersection_points, list_of_world_coordinates):

    """

    This function calculates the homography H from an images A to an image B.


    The function is passed eight correspondences for A and B.

    It returns the 3x3 homography matrix H.

    """

    A = []

    parameters_LM = []

    for index in range(len(list_of_intersection_points)):

        x_prime = list_of_intersection_points[index][0]

        y_prime = list_of_intersection_points[index][1]


        x = list_of_world_coordinates[index][0]

        y = list_of_world_coordinates[index][1]


        A.append([0,0,0, -x, -y, -1, y_prime*x, y_prime*y, y_prime])

        A.append([ x, y, 1,0,0,0, -x_prime*x, -x_prime*y, -x_prime])


        parameters_LM.append([x,y,x_prime,y_prime])

    arr_parameters_LM = np.array(parameters_LM)

    A = np.array(A)

#    b = np.zeros(9)

    u,d,vt = np.linalg.svd(A)
```

```
    v = np.transpose(vt)
    h = v[:,8]
    H = np.reshape(h, (3,3))
    H = np.reshape(h, (3,3))
    H = run_Levenberg_Marquardt(H, arr_parameters_LM)
    return H

def run_Levenberg_Marquardt(homography, best_inliers):
    h = homography.flatten()
    solution = root(cost_function, h, args=best_inliers, jac = True, method = "lm")
    h = solution.x
    H = np.reshape(h, (3,3))
    return H

def cost_function(homography, best_inliers):
    """
    cost function for LM for the original homography.
    """
    homography = np.reshape(homography,(3,3))
    X_unsorted = best_inliers[:,2:4]
    original_points =  best_inliers[:,0:2]
    z = np.ones((original_points.shape[0],1), dtype = np.int64)

    points_3_coords = np.append(original_points, z, axis = 1)
    f_unsorted = points_3_coords.dot(homography)

    f_unsorted = np.transpose(f_unsorted)

    f_unsorted = f_unsorted/f_unsorted[2]
    f_unsorted = np.transpose(f_unsorted)

    h_1 = np.array((homography[0,0], homography[0,1], homography[0,2]))
    h_2 = np.array((homography[1,0], homography[1,1], homography[1,2]))
    h_3 = np.array((homography[2,0], homography[2,1], homography[2,2]))
    f_1_num = np.dot(points_3_coords,h_1)

    f_2_num = np.dot(points_3_coords,h_2)
    f_denom = np.dot(points_3_coords,h_3)
    f_1 = np.divide(f_1_num, f_denom)
```

```
    f_2 = np.divide(f_2_num, f_denom)

    f = []
    X = []
    J = []
    for i in range(f_unsorted.shape[0]):
        f.append(f_1[i])
        f.append(f_2[i])
        X.append(X_unsorted[i,0])
        X.append(X_unsorted[i,1])
        x, y, z = np.divide(points_3_coords[i],f_denom[i])
        z = np.divide(1, f_denom[i])

        J.append([x, y, z, 0,0,0, -np.multiply(x, f_1[i]), -np.multiply(y, f_1[i]), -np.multiply(z, f_1[i]) ])
        J.append([0, 0, 0, x,y,z, -np.multiply(x, f_2[i]), -np.multiply(y, f_2[i]), -np.multiply(z, f_2[i]) ])

    f = np.asarray(f)
    X = np.asarray(X)
    J = np.asarray(J)

    error = X-f
    C = error
    error = error.reshape(-1,1)
    J_error = -J
    return C, J_error

def apply_homography(image, target, homography, output_filename, change_size, fixed_height = 1000):
    """
        Use a homography to change the perspective of an image into the perspective of the target.
    """
    image = np.transpose(image, (1,0,2))
    height, width, z = np.shape(image)

    target = np.transpose(target, (1,0,2))
    frame_width, frame_height, frame_z = np.shape(target)

    x_values_in_target = np.arange(frame_width)
    y_values_in_target = np.arange(frame_height)
    target_grid = np.array(np.meshgrid(x_values_in_target, y_values_in_target, np.arange(1,2)))
```

```
combinations_in_target = np.transpose(target_grid.T.reshape(-1, 3))
target_new_homogenous_coord = homography.dot(combinations_in_target)
target_new_homogenous_coord = target_new_homogenous_coord/target_new_homogenous_coord[2]

x_coordinates = target_new_homogenous_coord[0]
y_coordinates = target_new_homogenous_coord[1]

x_coordinates_rounded = np.round(target_new_homogenous_coord[0]).astype(int)
y_coordinates_rounded = np.round(target_new_homogenous_coord[1]).astype(int)
# Check conditions on x
x_coordinates_greater_zero = x_coordinates_rounded > 0
x_coordinates_smaller_x_shape = x_coordinates_rounded < height-1
x_coordinates_valid = x_coordinates_greater_zero*x_coordinates_smaller_x_shape
# Check conditions on y
y_coordinates_greater_zero = y_coordinates_rounded > 0
y_coordinates_smaller_y_shape = y_coordinates_rounded < width-1
y_coordinates_valid = y_coordinates_greater_zero*y_coordinates_smaller_y_shape

valid_coordinates = x_coordinates_valid*y_coordinates_valid

target_valid_x = combinations_in_target[0][valid_coordinates == True]
target_valid_y = combinations_in_target[1][valid_coordinates == True]

list_of_x_values_target = list(target_valid_x)
list_of_y_values_target = list(target_valid_y)
list_of_valid_coordinate_pairs = list()

for i in range(len(list_of_x_values_target)):
    list_of_valid_coordinate_pairs.append([list_of_x_values_target[i], list_of_y_values_target[i]])
valid_x = x_coordinates[valid_coordinates == True]
valid_y = y_coordinates[valid_coordinates == True]

list_of_x_values = list(valid_x)
list_of_y_values = list(valid_y)

list_of_original_coords_for_mapping = list()
for i in range(len(list_of_x_values)):
    list_of_original_coords_for_mapping.append([list_of_x_values[i], list_of_y_values[i]])
```

```
    j = 0
    for pair in list_of_valid_coordinate_pairs:

        target[pair[0], pair[1], : ] = getPixel(list_of_original_coords_for_mapping[j][0],list_of_original_coords_for_mapping[j][1], image)
        j = j+1
    target = np.transpose(target, (1,0,2))
    cv2.imwrite(output_filename, target)
    return target

def calculate_distance(d1, d2):
    return np.sqrt(np.add(np.square(d1),np.square(d2)))

def getPixel(x, y, image):
    xl = np.floor(x.copy()).astype(int)
    xu = np.ceil(x.copy()).astype(int)
    yl = np.floor(y.copy()).astype(int)
    yu = np.ceil(y.copy()).astype(int)

    dx_l = x - xl
    dx_u = x - xu
    dy_l = y - yl
    dy_u = y - yu

    d_ll = calculate_distance(dx_l,dy_l)
    d_uu = calculate_distance(dx_u,dy_u)
    d_lu = calculate_distance(dx_l,dy_u)
    d_ul = calculate_distance(dx_u,dy_l)

    if(d_ll + d_uu + d_ul + d_lu) != 0:
        pixel = d_ll * image[xl,yl, :] + d_uu* image[xu,yu, :] + d_lu*image[xl,yu, :] + d_ul*image[xu,yl, :]
        pixel = pixel/(d_ll + d_uu + d_ul + d_lu)
    else:
        pixel = image[xu,yu, :]
    return pixel

def get_v_vectors(H):
    """
        Returns all V_ij as described by Zhang.
    """
```

```
    h1 = H[:,0]
    h2 = H[:,1]

    v11 = get_v_vector(h1,h1)
    v22 = get_v_vector(h2,h2)
    v12 = get_v_vector(h1,h2)
    return v11, v22, v12

def get_v_vector(h_i, h_j):
    """
        Returns a certain V_ij as described by Zhang.
    """
    v = np.zeros(6)
    v[0] = h_i[0]*h_j[0]
    v[1] = h_i[0]*h_j[1] + h_i[1]*h_j[0]
    v[2] = h_i[1]*h_j[1]
    v[3] = h_i[2]*h_j[0] + h_i[0]*h_j[2]
    v[4] = h_i[2]*h_j[1] + h_i[1]*h_j[2]
    v[5] = h_i[2]*h_j[2]
    return v

def get_intrinsic_parameters(b):
    """
        Return the matrix K of the camera based
    """
    w11 = b[0]
    w12 = b[1]
    w22 = b[2]
    w13 = b[3]
    w23 = b[4]
    w33 = b[5]

    x0 = (w12*w13 - w11*w23)/(w11*w22-np.square(w12))
    lambd = w33 - (np.square(w13)+x0*(w12*w13 - w11*w23))/w11
    alpha_x = np.sqrt(lambd/w11)
    alpha_y = np.sqrt(lambd*w11/(w11*w22-np.square(w12)))
    s = -1*(w12*np.square(alpha_x)*alpha_y)/lambd
    y0 = s*x0/alpha_y - (w13*np.square(alpha_x)/lambd)
```

```
    K = np.array(((alpha_x, s, x0),(0,alpha_y, y0), (0,0,1)))
    return K


def get_homographies_and_b(intersection_points):
    """
        Returns the homographies w.r.t. the world measurements.
    """
    world_coordinates = []
    for i in range(8):
        for j in range(10):
            # Use 1px = 1mm and have world coordinates in inch
            world_coordinates.append((i*2.54*10,j*2.54*10))
            # Use 1px = 1cm and have world coordinates in inch
#            world_coordinates.append((i*2.54,j*2.54))
    list_of_homographies = []
    V = np.array([])
    for intersection_list in intersection_points:
        H = linear_least_squares_homography(intersection_list, world_coordinates)
        list_of_homographies.append(H)

        v11, v22, v12 = get_v_vectors(H)

        V_single = np.zeros((2,6))
        V_single[0,:] = np.transpose(v12)
        V_single[1,:] = np.transpose(v11-v22)

        V = np.append(V,V_single)
    V_arr = np.reshape(V,(-1,6))
    u,d,vt = np.linalg.svd(V_arr)
    v = np.transpose(vt)
    b = v[:,5]
    return b, list_of_homographies, world_coordinates


def get_extrinsic_parameters(list_of_homographies, K):
    """
        Get the list of [R|t] and t from the homographies and intrinsic parameters.
    """
    list_of_Rt = []
    list_of_t = []
```

```
    for k in range(len(list_of_homographies)):
        H = list_of_homographies[k]
        K_inv = np.linalg.pinv(K)
        h1 = H[:,0]
        h2 = H[:,1]
        h3 = H[:,2]

        r1 = np.dot(K_inv, h1)
        chi = 1/(np.linalg.norm(r1))
        r1 = r1*chi
        r2 = np.dot(K_inv, h2)*chi
        r3 = np.cross(r1, r2)
        t = np.dot(K_inv, h3)*chi

        R = np.transpose(np.append(np.append(r1,r2),r3).reshape((3,3)))
        u, d, vt = np.linalg.svd(R)
        R = np.dot(u,vt)
        r1 = R[:,0]
        r2 = R[:,1]
        r3 = R[:,2]

        Rt = np.transpose(np.append(np.append(np.append(r1,r2),r3),t).reshape((4,3)))
        list_of_Rt.append(Rt)
        list_of_t.append(t)
    return list_of_Rt, list_of_t


def convert_to_Rodrigues(list_of_Rt):
    """
        Reduce the 9 variables in R to 3 in w.
    """
    list_of_ws = []
    for Rt in list_of_Rt:
        R = Rt[:,:3]
        phi = np.arccos((np.trace(R)-1)/2)
        # TODO: check if necessary
        if phi == 0:
            scaling = 1
        else:
            scaling = phi/(2*np.sin(phi))
```

```
        w= scaling*np.array(((R[2,1]-R[1,2]),(R[0,2]-R[2,0]),(R[1,0]-R[0,1])))
        list_of_ws.append(w)
    return list_of_ws

def create_p(K,list_of_Rt, list_of_ws, list_of_ts):
    """
        stack all variables for LM.
    """
    p = []
    alpha_x = K[0,0]
    alpha_y = K[1,1]
    s = K[0,1]
    x0 = K[0,2]
    y0 = K[1,2]

    p.append(alpha_x)
    p.append(s)
    p.append(x0)
    p.append(alpha_y)
    p.append(y0)

    for k in range(len(list_of_Rt)):
        w = list_of_ws[k]
        t = list_of_ts[k]
        p.append(w[0])
        p.append(w[1])
        p.append(w[2])
        p.append(t[0])
        p.append(t[1])
        p.append(t[2])
    return p

def create_p_w_rad_dist(K,list_of_Rt, list_of_ws, list_of_ts):
    """
        stack all variables for LM.
    """
    p = []
    alpha_x = K[0,0]
    alpha_y = K[1,1]
```

```
    s = K[0,1]

    x0 = K[0,2]

    y0 = K[1,2]


    p.append(alpha_x)

    p.append(s)

    p.append(x0)

    p.append(alpha_y)

    p.append(y0)


    for k in range(len(list_of_Rt)):

        w = list_of_ws[k]

        t = list_of_ts[k]

        p.append(w[0])

        p.append(w[1])

        p.append(w[2])

        p.append(t[0])

        p.append(t[1])

        p.append(t[2])

    # initialize k1 = 0 and k2=0 --> no radial distortion

    p.append(0)

    p.append(0)

    return p




def merge_common(lists):

    """

        Method to merge sublists with same elements. Taken from https://www.geeksforgeeks.org/python-merge-list-with-common-
elements-in-a-list-of-lists/

    """

    neigh = defaultdict(set)

    visited = set()

    for each in lists:

        for item in each:

            neigh[item].update(each)

    def comp(node, neigh = neigh, visited = visited, vis = visited.add):

        nodes = set([node])

        next_node = nodes.pop

        while nodes:
```

```
            node = next_node()
            vis(node)
            nodes |= neigh[node] - visited
            yield node
    for node in neigh:
        if node not in visited:
            yield sorted(comp(node))


def cost_parameters(p):
    """
        Cost function for the LM optimization for the camera parameters.
    """
    global intersection_points
    global world_coordinates
    global list_of_Rt
    distances = []
    for k in range(len(list_of_Rt)):
        alpha_x = p[0]
        s = p[1]
        x0 = p[2]
        alpha_y = p[3]
        y0 = p[4]

        wx = p[6*k + 5]
        wy = p[6*k + 1 + 5]
        wz = p[6*k + 2 + 5]
        t1 = p[6*k + 3 + 5]
        t2 = p[6*k + 4 + 5]
        t3 = p[6*k + 5 + 5]

        w = np.array([wx,wy,wz])
        phi = np.linalg.norm(w)
        Wx = np.array(((0,-wz,wy),(wz,0,-wx),(-wy,wx,0)))

        term1 = np.identity(3)
        if phi == 0:
            term2 = 0*Wx
            term3 = 0*np.dot(Wx,Wx)
        else:
```

```
            term2 = np.sin(phi)/phi*Wx
            term3 = ((1-np.cos(phi))/np.square(phi))*np.dot(Wx,Wx)
        R = term1 + term2 + term3
        t = np.array([t1,t2,t3]).reshape((3,1))

        K = np.array(((alpha_x, s, x0),(0,alpha_y, y0), (0,0,1)))
        Rt = np.append(R,t, axis = 1)
        Rt = Rt[:,[0,1,3]]
        KRt = np.dot(K,Rt)
        for q in range(len(intersection_points[k])):
            intersect_point = intersection_points[k][q]
            world_point = world_coordinates[q]
            world_point = np.append(world_point,1)

            world_proj = np.dot(KRt,world_point)
            world_proj = world_proj/world_proj[2]
            world_proj_point = world_proj[:2]

            distances.append(intersect_point[0] - world_proj_point[0])
            distances.append(intersect_point[1] - world_proj_point[1])
    return distances

def cost_parameters_rad_dist(p):
    """
        Cost function for the LM optimization for the camera parameters.
    """
    global intersection_points
    global world_coordinates
    global list_of_Rt
    distances = []
    for k in range(len(list_of_Rt)):
        alpha_x = p[0]
        s = p[1]
        x0 = p[2]
        alpha_y = p[3]
        y0 = p[4]

        wx = p[6*k + 5]
        wy = p[6*k + 1 + 5]
```

```
wz = p[6*k + 2 + 5]
t1 = p[6*k + 3 + 5]
t2 = p[6*k + 4 + 5]
t3 = p[6*k + 5 + 5]

k1 = p[-2]
k2 = p[-1]
w = np.array([wx,wy,wz])
phi = np.linalg.norm(w)
Wx = np.array(((0,-wz,wy),(wz,0,-wx),(-wy,wx,0)))

term1 = np.identity(3)
if phi == 0:
    term2 = 0*Wx
    term3 = 0*np.dot(Wx,Wx)
else:
    term2 = np.sin(phi)/phi*Wx
    term3 = ((1-np.cos(phi))/np.square(phi))*np.dot(Wx,Wx)
R = term1 + term2 + term3
t = np.array([t1,t2,t3]).reshape((3,1))

K = np.array(((alpha_x, s, x0),(0,alpha_y, y0), (0,0,1)))
Rt = np.append(R,t, axis = 1)
Rt = Rt[:,[0,1,3]]
KRt = np.dot(K,Rt)
for q in range(len(intersection_points[k])):
    intersect_point = intersection_points[k][q]
    world_point = world_coordinates[q]
    world_point = np.append(world_point,1)

    world_proj = np.dot(KRt,world_point)
    world_proj = world_proj/world_proj[2]
    world_proj_point = world_proj[:2]

    r_square = np.square(world_proj_point[0]-x0)+np.square(world_proj_point[1]-y0)
    rad_dist_x = world_proj_point[0]+ (world_proj_point[0]-x0)*(k1*r_square+k2*np.square(r_square))
    rad_dist_y = world_proj_point[1]+ (world_proj_point[1]-y0)*(k1*r_square+k2*np.square(r_square))
    distances.append(intersect_point[0] - rad_dist_x)
    distances.append(intersect_point[1] - rad_dist_y)
```

```
    return distances


def get_optimized_homographies(list_of_Rt, optimization, K):
    """
        Convert the results of the LM optimization to get the optimized homographies and [R|t]
    """
    list_of_H_optimized = []
    list_of_Rt_optimized = []
    for k in range(len(list_of_Rt)):
        wx = optimization.x[6*k + 5]
        wy = optimization.x[6*k + 1 + 5]
        wz = optimization.x[6*k + 2 + 5]
        t1 = optimization.x[6*k + 3 + 5]
        t2 = optimization.x[6*k + 4 + 5]
        t3 = optimization.x[6*k + 5 + 5]

        w = np.transpose(np.array([wx,wy,wz]))
        Wx = np.array(((0,-wz,wy),(wz,0,-wx),(-wy,wx,0)))

        phi = np.linalg.norm(w)

        term1 = np.identity(3)
        term2 = (np.sin(phi)/phi)*Wx
        term3 = ((1-np.cos(phi))/np.square(phi))*np.dot(Wx,Wx)
        R = term1 + term2 + term3
        t = np.transpose(np.array([t1,t2,t3])).reshape((3,1))
        Rt = np.append(R,t, axis = 1)
        list_of_Rt_optimized.append(Rt)
        H = np.dot(K,Rt[:,[0,1,3]])
        list_of_H_optimized.append(H)
    return list_of_H_optimized, list_of_Rt_optimized


def reproject_points_and_mean_var(list_of_Rt, list_of_H_fromKRt, list_of_H_optimized, intersection_points):
    """
        Project the intersection points of one image into the fixed image using the respective homographies.
    """
    if dataset_no == 1:
        fixed_image_index = 10
        marker_size = 2
```

```
        marker_length = 10
        font_size = 0.4
        font_boldness = 1
    if dataset_no == 2:
        fixed_image_index = 24
        marker_size = 5
        marker_length = 40
        font_size = 1.5
        font_boldness = 2

    list_of_all_projected_points = []
    list_of_all_projected_points_refined = []
    means = []
    means_refined = []
    variances = []
    variances_refined = []
    for k in range(len(list_of_Rt)):
        list_error = []
        list_error_refined = []
        image = cv2.imread("./Files/Dataset%s/Corners/Pic_%s.jpg"%(dataset_no, fixed_image_index+1))
        P1 = np.dot(list_of_H_fromKRt[fixed_image_index], np.linalg.pinv(list_of_H_fromKRt[k]))
        P2 = np.dot(list_of_H_optimized[fixed_image_index], np.linalg.pinv(list_of_H_optimized[k]))
        list_of_projected_points = []
        list_of_projected_points_refined = []
        j = 0
        for point in intersection_points[k]:
            point = np.append(point,1)
            point_proj1 = np.dot(P1,point)
            point_proj1 = point_proj1/point_proj1[2]
            point_proj1 = point_proj1[:2]
            list_of_projected_points.append(point_proj1)
            cv2.drawMarker(image,(np.round(point_proj1[0]).astype(np.int),np.round(point_proj1[1]).astype(np.int)), (255,255,0),1,
marker_length, marker_size)
            if dataset_no == 1:
                cv2.putText(image, "%s"%(j+1),(np.round(point_proj1[0]-16).astype(np.int),np.round(point_proj1[1]-8).astype(np.int)),
cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,255,0),  font_boldness, cv2.LINE_AA)
            elif dataset_no == 2:
                cv2.putText(image, "%s"%(j+1),(np.round(point_proj1[0]-60).astype(np.int),np.round(point_proj1[1]-30).astype(np.int)),
cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,255,0),  font_boldness, cv2.LINE_AA)
            list_error.append(np.linalg.norm(intersection_points[fixed_image_index][j]-point_proj1))
```

```
            point_proj2 = np.dot(P2,point)

            point_proj2 = point_proj2/point_proj2[2]

            point_proj2 = point_proj2[:2]

            list_of_projected_points_refined.append(point_proj2)

            cv2.drawMarker(image,(np.round(point_proj2[0]).astype(np.int),np.round(point_proj2[1]).astype(np.int)), (255,51,255),1, 
marker_length, marker_size)

            if dataset_no == 1:

              cv2.putText(image, "%s"%(j+1),(np.round(point_proj2[0]+8).astype(np.int),np.round(point_proj2[1]-8).astype(np.int)), 
cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,51,255), font_boldness, cv2.LINE_AA)

            elif dataset_no == 2:

              cv2.putText(image, "%s"%(j+1),(np.round(point_proj2[0]+10).astype(np.int),np.round(point_proj2[1]-30).astype(np.int)), 
cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,51,255), font_boldness, cv2.LINE_AA)

            list_error_refined.append(np.linalg.norm(intersection_points[fixed_image_index][j]-point_proj2))

            j = j+1


        list_of_all_projected_points.append(list_of_projected_points)

        list_of_all_projected_points_refined.append(list_of_projected_points)

        cv2.imwrite("./Files/Dataset%s/Projected_Comp/Pic_%s.jpg"%(dataset_no,k+1),image)


        means.append(np.mean(list_error))

        means_refined.append(np.mean(list_error_refined))


        variances.append(np.var(list_error))

        variances_refined.append(np.var(list_error_refined))


    return list_of_all_projected_points, list_of_all_projected_points_refined, means, means_refined, variances, variances_refined


def reproject_points_and_mean_var_rad_dist(list_of_Rt, list_of_H_fromKRt, list_of_H_optimized1, list_of_H_optimized2, 
intersection_points):

    """

        Project the intersection points of one image into the fixed image using the respective homographies.

    """

    if dataset_no == 1:

        fixed_image_index = 10

        marker_size = 2

        marker_length = 10

        font_size = 0.4

        font_boldness = 1
```

```
    if dataset_no == 2:
        fixed_image_index = 24
        marker_size = 5
        marker_length = 40
        font_size = 1.5
        font_boldness = 2

    list_of_all_projected_points = []
    list_of_all_projected_points_refined = []
    list_of_all_projected_points_rad = []
    means = []
    means_refined = []
    means_rad = []
    variances = []
    variances_refined = []
    variances_rad = []
    for k in range(len(list_of_Rt)):
        list_error = []
        list_error_refined = []
        list_error_rad = []
        image = cv2.imread("./Files/Dataset%s/Corners/Pic_%s.jpg"%(dataset_no, fixed_image_index+1))
        P1 = np.dot(list_of_H_fromKRt[fixed_image_index], np.linalg.pinv(list_of_H_fromKRt[k]))
        P2 = np.dot(list_of_H_optimized1[fixed_image_index], np.linalg.pinv(list_of_H_optimized1[k]))
        P3 = np.dot(list_of_H_optimized2[fixed_image_index], np.linalg.pinv(list_of_H_optimized2[k]))
        list_of_projected_points = []
        list_of_projected_points_refined = []
        list_of_projected_points_rad = []
        j = 0
        for point in intersection_points[k]:
            point = np.append(point,1)
            point_proj1 = np.dot(P1,point)
            point_proj1 = point_proj1/point_proj1[2]
            point_proj1 = point_proj1[:2]
            list_of_projected_points.append(point_proj1)
#           cv2.drawMarker(image,(np.round(point_proj1[0]).astype(np.int),np.round(point_proj1[1]).astype(np.int)), (255,255,0),1, marker_length, marker_size)
#           if dataset_no == 1:
#               cv2.putText(image, "%s"%(j+1),(np.round(point_proj1[0]-16).astype(np.int),np.round(point_proj1[1]-8).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,255,0),  font_boldness, cv2.LINE_AA)
```

```
#           elif dataset_no == 2:

#               cv2.putText(image, "%s"%(j+1),(np.round(point_proj1[0]-60).astype(np.int),np.round(point_proj1[1]-30).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,255,0),  font_boldness, cv2.LINE_AA)

            list_error.append(np.linalg.norm(intersection_points[fixed_image_index][j]-point_proj1))




            point_proj2 = np.dot(P2,point)

            point_proj2 = point_proj2/point_proj2[2]

            point_proj2 = point_proj2[:2]

            list_of_projected_points_refined.append(point_proj2)

            cv2.drawMarker(image,(np.round(point_proj2[0]).astype(np.int),np.round(point_proj2[1]).astype(np.int)), (255,51,255),1, marker_length, marker_size)

            if dataset_no == 1:

                cv2.putText(image, "%s"%(j+1),(np.round(point_proj2[0]+8).astype(np.int),np.round(point_proj2[1]-8).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,51,255), font_boldness, cv2.LINE_AA)

            elif dataset_no == 2:

                cv2.putText(image, "%s"%(j+1),(np.round(point_proj2[0]+10).astype(np.int),np.round(point_proj2[1]-30).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,51,255), font_boldness, cv2.LINE_AA)

            list_error_refined.append(np.linalg.norm(intersection_points[fixed_image_index][j]-point_proj2))



            point_proj3 = np.dot(P3,point)

            point_proj3 = point_proj3/point_proj3[2]

            point_proj3 = point_proj3[:2]

            list_of_projected_points_rad.append(point_proj3)

            cv2.drawMarker(image,(np.round(point_proj3[0]).astype(np.int),np.round(point_proj3[1]).astype(np.int)), (255,0,127),1, marker_length, marker_size)

            if dataset_no == 1:

                cv2.putText(image, "%s"%(j+1),(np.round(point_proj3[0]+8).astype(np.int),np.round(point_proj3[1]-8).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,0,127), font_boldness, cv2.LINE_AA)

            elif dataset_no == 2:

                cv2.putText(image, "%s"%(j+1),(np.round(point_proj3[0]+10).astype(np.int),np.round(point_proj3[1]-30).astype(np.int)), cv2.FONT_HERSHEY_SIMPLEX,  font_size, (255,0,127), font_boldness, cv2.LINE_AA)

            list_error_rad.append(np.linalg.norm(intersection_points[fixed_image_index][j]-point_proj3))

            j = j+1


        list_of_all_projected_points.append(list_of_projected_points)

        list_of_all_projected_points_refined.append(list_of_projected_points)

        list_of_all_projected_points_rad.append(list_of_projected_points_rad)

        cv2.imwrite("./Files/Dataset%s/Rad_Dist/Pic_%s.jpg"%(dataset_no,k+1),image)


        means.append(np.mean(list_error))

        means_refined.append(np.mean(list_error_refined))
```

```
        means_rad.append(np.mean(list_error_rad))


        variances.append(np.var(list_error))

        variances_refined.append(np.var(list_error_refined))

        variances_rad.append(np.var(list_error_rad))


    return list_of_all_projected_points, list_of_all_projected_points_refined, list_of_all_projected_points_rad, means, means_refined, means_rad, variances, variances_refined, variances_rad




## Code for Dataset1

dataset_no = 1

stretch_lines = 1000

hough_parameter = 50


intersection_points = run_corner_detection(dataset_no, stretch_lines, hough_parameter)

b, list_of_homographies, world_coordinates = get_homographies_and_b(intersection_points)


K = get_intrinsic_parameters(b)


## Code for Debugging the homography

#for k in range(1,41):

#    image = cv2.imread("./Files/Dataset1/Pic_%s.jpg"%(k))

#    target = np.zeros(np.shape(image))

#    H = list_of_homographies[k-1]

#    target = apply_homography(image, target, H, "./Files/Dataset1/Debug/Pic_%s.jpg"%(k), False)


list_of_Rt, list_of_ts = get_extrinsic_parameters(list_of_homographies, K)


#homography reconstructed from KRt

list_of_H_fromKRt = []

for k in range(len(list_of_Rt)):

    list_of_H_fromKRt.append(np.dot(K,list_of_Rt[k][:,[0,1,3]]))


list_of_ws = convert_to_Rodrigues(list_of_Rt)


p = create_p(K,list_of_Rt, list_of_ws, list_of_ts)


optimization = least_squares(cost_parameters, p, method = "lm", max_nfev=500)
```

```
K_optimized = np.array(((optimization.x[0], optimization.x[1], optimization.x[2]),(0,optimization.x[3], optimization.x[4]), (0,0,1)))

list_of_H_optimized, list_of_Rt_opt = get_optimized_homographies(list_of_Rt, optimization, K_optimized)

list_of_all_projected_points, list_of_all_projected_points_refined, means, means_refined, variances, variances_refined =
reproject_points_and_mean_var(list_of_Rt, list_of_H_fromKRt, list_of_H_optimized, intersection_points)

# Code for Dataset2
dataset_no = 2
stretch_lines = 2500
hough_parameter = 50

intersection_points = run_corner_detection_2(dataset_no, stretch_lines, hough_parameter)
b, list_of_homographies, world_coordinates = get_homographies_and_b(intersection_points)

K = get_intrinsic_parameters(b)

## Code for Debugging the homography
#for k in range(1,41):
#    image = cv2.imread("./Files/Dataset1/Pic_%s.jpg"%(k))
#    target = np.zeros(np.shape(image))
#    H = list_of_homographies[k-1]
#    target = apply_homography(image, target, H, "./Files/Dataset1/Debug/Pic_%s.jpg"%(k), False)

list_of_Rt, list_of_ts = get_extrinsic_parameters(list_of_homographies, K)

#homography reconstructed from KRt
list_of_H_fromKRt = []
for k in range(len(list_of_Rt)):
    list_of_H_fromKRt.append(np.dot(K,list_of_Rt[k][:,[0,1,3]]))

list_of_ws = convert_to_Rodrigues(list_of_Rt)

p = create_p(K,list_of_Rt, list_of_ws, list_of_ts)

optimization = least_squares(cost_parameters, p, method = "lm", max_nfev=500)

K_optimized = np.array(((optimization.x[0], optimization.x[1], optimization.x[2]),(0,optimization.x[3], optimization.x[4]), (0,0,1)))
```

```
list_of_H_optimized, list_of_Rt_opt = get_optimized_homographies(list_of_Rt, optimization, K_optimized)

list_of_all_projected_points, list_of_all_projected_points_refined, means, means_refined, variances, variances_refined = reproject_points_and_mean_var(list_of_Rt, list_of_H_fromKRt, list_of_H_optimized, intersection_points)

## Code for Dataset1 with radial distortion
dataset_no = 1
stretch_lines = 1000
hough_parameter = 50

intersection_points = run_corner_detection(dataset_no, stretch_lines, hough_parameter)
b, list_of_homographies, world_coordinates = get_homographies_and_b(intersection_points)

K = get_intrinsic_parameters(b)

## Code for Debugging the homography
#for k in range(1,41):
#    image = cv2.imread("./Files/Dataset1/Pic_%s.jpg"%(k))
#    target = np.zeros(np.shape(image))
#    H = list_of_homographies[k-1]
#    target = apply_homography(image, target, H, "./Files/Dataset1/Debug/Pic_%s.jpg"%(k), False)

list_of_Rt, list_of_ts = get_extrinsic_parameters(list_of_homographies, K)

#homography reconstructed from KRt
list_of_H_fromKRt = []
for k in range(len(list_of_Rt)):
    list_of_H_fromKRt.append(np.dot(K,list_of_Rt[k][:,[0,1,3]]))

list_of_ws = convert_to_Rodrigues(list_of_Rt)

p1 = create_p(K,list_of_Rt, list_of_ws, list_of_ts)
p2 = create_p_w_rad_dist(K,list_of_Rt, list_of_ws, list_of_ts)

optimization1 = least_squares(cost_parameters, p1, method = "lm", max_nfev=500)
```

```
optimization2 = least_squares(cost_parameters_rad_dist, p2, method = "lm", max_nfev=500)

K_optimized1 = np.array(((optimization1.x[0], optimization1.x[1], optimization1.x[2]),(0,optimization1.x[3], optimization1.x[4]), (0,0,1)))
K_optimized2 = np.array(((optimization2.x[0], optimization2.x[1], optimization2.x[2]),(0,optimization2.x[3], optimization2.x[4]), (0,0,1)))

list_of_H_optimized1, list_of_Rt_opt1 = get_optimized_homographies(list_of_Rt, optimization1, K_optimized1)
list_of_H_optimized2, list_of_Rt_opt2 = get_optimized_homographies(list_of_Rt, optimization2, K_optimized2)
k1 = optimization2.x[-2]
k2 = optimization2.x[-1]

list_of_all_projected_points, list_of_all_projected_points_refined, list_of_all_projected_points_rad, means, means_refined, means_rad,
variances, variances_refined, variances_rad = reproject_points_and_mean_var_rad_dist(list_of_Rt, list_of_H_fromKRt,
list_of_H_optimized1, list_of_H_optimized2, intersection_points)

## Code for Dataset2 with radial distortion
dataset_no = 2
stretch_lines = 2500
hough_parameter = 50

intersection_points = run_corner_detection_2(dataset_no, stretch_lines, hough_parameter)
b, list_of_homographies, world_coordinates = get_homographies_and_b(intersection_points)

K = get_intrinsic_parameters(b)

## Code for Debugging the homography
#for k in range(1,41):
#    image = cv2.imread("./Files/Dataset1/Pic_%s.jpg"%(k))
#    target = np.zeros(np.shape(image))
#    H = list_of_homographies[k-1]
#    target = apply_homography(image, target, H, "./Files/Dataset1/Debug/Pic_%s.jpg"%(k), False)

list_of_Rt, list_of_ts = get_extrinsic_parameters(list_of_homographies, K)

#homography reconstructed from KRt
list_of_H_fromKRt = []
for k in range(len(list_of_Rt)):
    list_of_H_fromKRt.append(np.dot(K,list_of_Rt[k][:,[0,1,3]]))

list_of_ws = convert_to_Rodrigues(list_of_Rt)
```

```
p1 = create_p(K,list_of_Rt, list_of_ws, list_of_ts)

p2 = create_p_w_rad_dist(K,list_of_Rt, list_of_ws, list_of_ts)


optimization1 = least_squares(cost_parameters, p1, method = "lm", max_nfev=500)

optimization2 = least_squares(cost_parameters_rad_dist, p2, method = "lm", max_nfev=500)


K_optimized1 = np.array(((optimization1.x[0], optimization1.x[1], optimization1.x[2]),(0,optimization1.x[3], optimization1.x[4]), (0,0,1)))

K_optimized2 = np.array(((optimization2.x[0], optimization2.x[1], optimization2.x[2]),(0,optimization2.x[3], optimization2.x[4]), (0,0,1)))


list_of_H_optimized1, list_of_Rt_opt1 = get_optimized_homographies(list_of_Rt, optimization1, K_optimized1)

list_of_H_optimized2, list_of_Rt_opt2 = get_optimized_homographies(list_of_Rt, optimization2, K_optimized2)

k1 = optimization2.x[-2]

k2 = optimization2.x[-1]


list_of_all_projected_points, list_of_all_projected_points_refined, list_of_all_projected_points_rad, means, means_refined, means_rad, variances, variances_refined, variances_rad = reproject_points_and_mean_var_rad_dist(list_of_Rt, list_of_H_fromKRt, list_of_H_optimized1, list_of_H_optimized2, intersection_points)
```