COMPUTER VISION
ECE:66100

Homework Assignment 8

Name: Amruthavarshini Talikoti
Email ID: atalikot@purdue.edu
Student ID: 0029949176
Purdue University
Department of Electrical and Computer Engineering

Introduction

The main objective is to determine the intrinsic and extrinsic camera parameters using the Zhang's camera calibration algorithm. The concepts and ideas involved have been briefed herewith.

Corner Detection

All the images in the dataset are read and stored in their grayscale. The inbuilt Canny and HoughLines functions are used to determine the edges and apply the Hough transform to find the strong disjoint edges in each of the checker-board image views. The threshold used for Canny are 255*1.5 and 255.

The next step is to choose the best 10X8 lines that will optimally represent the intersections in the image. For this, the hough lines retrieved are divided into two sets, horizontal comprising of all the lines with angles between $\frac{\pi}{4}$ and $3\frac{\pi}{4}$ and vertical, all the other lines. Now, the aim is to select 10 best lines from the horizontal set and 8 from the vertical. For this lines from either set are chosen such that the distance between them is greater than a given threshold. This threshold is iteratively tuned until 10 lines are selected from horizontal and 8 from vertical. This will give the final hough lines selection. A ready function was used to draw these selected lines.

Each of the lines are represented using ρ and θ factors. Now the horizontal and vertical line sets are each sorted according to their ρ parameters. And a ready function is used to find the intersections of these 10 horizontal with 8 vertical lines returning the nearest integer pixel locations of the 80 intersections for each of the images. Finally these intersections are labelled in each of the images starting from top to bottom and left to right.

Camera Calibration

The first step is to estimate the homographies for each of the images between the intersections computed previously and their respective world points. The world points are determined by measuring the distance between each of the squares in the checkerboard pattern using a ruler. The homographies are computed so as to satisfy the equation $x_{image} = H * x_{world}$. The homographies are found as in the calculation of nullspace of matrix as done previously. The solution is obtained as teh eigen vector corresponding to the smallest eigenvalue in the SVD of the considered matrix. This matrix is rearranged as a 3X3 homography. Thus, we have 40 homographies for each of the 40 images in the first dataset.

The homographies so found are used to find the intrinsic and extrinsic camera parameters using Zhang's algorithm. It is based on the assumption that the calibration pattern is at the Z=0 plane of the world frame and images of this pattern are taken from different viewpoints. The other fact basis for this algorithm is that the camera image of the absolute conic Ω_∞ is independent of the extrinsic parameters R and $\vec{t}$, and it is given by $w = K^{-T}K^{-1}$.The camera image of the two points that lie on any world plane (Z=0) and the absolute conic must lie on the camera image of the absolute conic. So they satisfy $\vec{x}^T w\vec{x} = 0$. This implies the following two equations,

$$\vec{h_1}^T w\vec{h_1} = \vec{h_2}^T w\vec{h_2}$$
$$\vec{h_1}^T w\vec{h_2} = 0$$

These equations can be rewritten as following,

$$(\vec{V_{11}} - \vec{V_{22}})^T \vec{b} = 0$$
$$\vec{V_{12}}^T \vec{b} = 0$$

wherein $\vec{b}$ is a vector of unknowns

$$b = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{22} \\ w_{13} \\ w_{23} \\ w_{33} \end{bmatrix}$$

and

$$\vec{V_{ij}} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix}$$

wherein h_1, h_2 and h_3 are the columns of the homography matrix H. Now these equations can be stacked into a (2nX6) matrix V as follows

$$V = \begin{bmatrix} \vec{V}_{12}^T \\ (\vec{V}_{11} - \vec{V}_{22})^T \end{bmatrix}$$

Further this is equivalent to solving the system of equations $V\vec{b} = \vec{0}$ by using linear least squares method. Finally using this $\vec{b}$ the image conic w can be determined.

Intrinsic Parameters:

The intrinsic parameters are related to the image of the absolute conic as follows,

$$w = K^{-T}K^{-1}$$

Since we are dealing with homogeneous equations, the aim is to scale the w matrix so that the final solution of the K matrix has the following form.

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

wherein the different elements are calculated as follows:

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

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

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

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

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

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

Extrinsic Parameters:

While the intrinsic parameters are the same for all different viewpoints (images) for one

camera, the extrinsic parameters change for every image depending on the view point So the R and $\vec{t}$ are different for each image. These are found by carrying out the following computations.

$$\vec{r}_1 = \varepsilon K^{-1}\vec{h}_1$$
$$\vec{r}_2 = \varepsilon K^{-1}\vec{h}_2$$
$$\vec{r}_3 = \vec{r}_1 X \vec{r}_2$$
$$\vec{t} = \varepsilon K^{-1}\vec{h}_3$$

wherein

$$\varepsilon = \frac{1}{||K^{-1}\vec{h}_1||}$$

$\vec{r}_1, \vec{r}_2$ and $\vec{r}_3$ are the columns of the R matrix and the$\vec{t}$ denote the translation parameters. R and $\vec{t}$ together denote the extrinsic parameters for each image.

Refinement of the Calibration Parameters

Since the obtained K, R and t parameters are coarse and not exact, Levenberg Marquardt algorithm is used to refine them. For this, the inbuild lm function is used with cost function as:

$$d^2_{geom} = \sum_i \sum_j ||\vec{x_{ij}} - K[R_i|t_i]\vec{x_{M,j}}||^2$$

wherein $\vec{x_{M,j}}$ is the j-th salient point on the calibration pattern and $\vec{x_{ij}}$ is the actual image point for $\vec{x_{M,j}}$ in the i-th position of the camera. R_i is the rotation matrix for the i-th position of the camera. $\vec{t_i}$ is the translational vector for the i-th position of the camera and K is the camera calibration matrix for the intrinsic parameters. The parameters are all stacked in a long vector of length (5+6*no.of.images) and sent to the cost function in the form of $[\alpha_x, s, \alpha_y, x_0, y_0, w_{x1}, w_{y1}, w_{z1}, t_{11}, t_{21}, t_{31}.....]$
Also, since the rotation matrix is expected to have only 3 degrees of freedom and the 3X3 R matrix has 9 degrees of freedom, we make use of Rodrigues representation. Using polar

coordinates, matrix R is transformed into Rodrigues representation as,

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

$$with \;\; R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

$$and \;\; \phi = cos^{-1}\frac{trace(R) - 1}{2}$$

Transforming from Rodrigues to R involves the following computations,

$$R = I_{3X3} + \frac{sin\phi}{\phi}[\vec{w}]_x + \frac{1 - cos\phi}{\phi^2}[\vec{w}]_x^2$$

$$wherein \;\; \phi = ||w||$$

$$and \;\; [\vec{w}]_x = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$$

Converting $\vec{w}$ to R ensures orthonormality of R.

Conditioning Rotation Matrix

We have to condition the rotation matrix R while computing the extrinsic parameters so that its orthonormality is maintained. For this given a matrix P we have to minimize the Frobenius norm $||R - P||_F^2$ wherein R is the required final rotation matrix. This is carried out by finding the SVD of $P = UDV^T$ and setting the required $R = UV^T$.

Reprojecting onto fixed image

The Homographies before and after LM are computed as $K[r_1 r_2 t]$ using the old and refined R and t respectively. The fixed image is number 11 for given dataset and 1 for my dataset. The points in any other given image say 18 is reprojected onto the fixed image say 11 as follows,

$$Reprojected.points.in.11 = (H[11] * H[18]^{-1} * (point.in.18)).$$

These are plotted with the intersections in fixed image(11) with two different colors and the mean and variances of the errors measured as euclidean distances is computed for 4 images.

Incorporating Radial Distortion

We have assumed a pin-hole camera model for all the calibration computations. However, this model breaks in the case of short focal-length cameras. Subsequently the pixel coordinates predicted by pinhole model must be radially adjusted to incorporate radial distortion. Let K be the intrinsic camera parameters, $(R_i, \vec{t_i})$ be extrinsic parameters for each position of camera, $(\hat{x}, \hat{y})$ be the predicted position of a pixel using pinhole model and the tentative set of calibration parameters, $(x\hat{_{rad}}, y\hat{_{rad}})$ be the pixel coordinates that would be predicted if radial distortion were included. LM is used to tune the values of k_1 and k_2 by passing it in the vector p to the cost function.

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

where (x_0, y_0) are the currently available principal point coordinates for plane of camera image and

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

Results

Given Dataset Results

Edge Detection

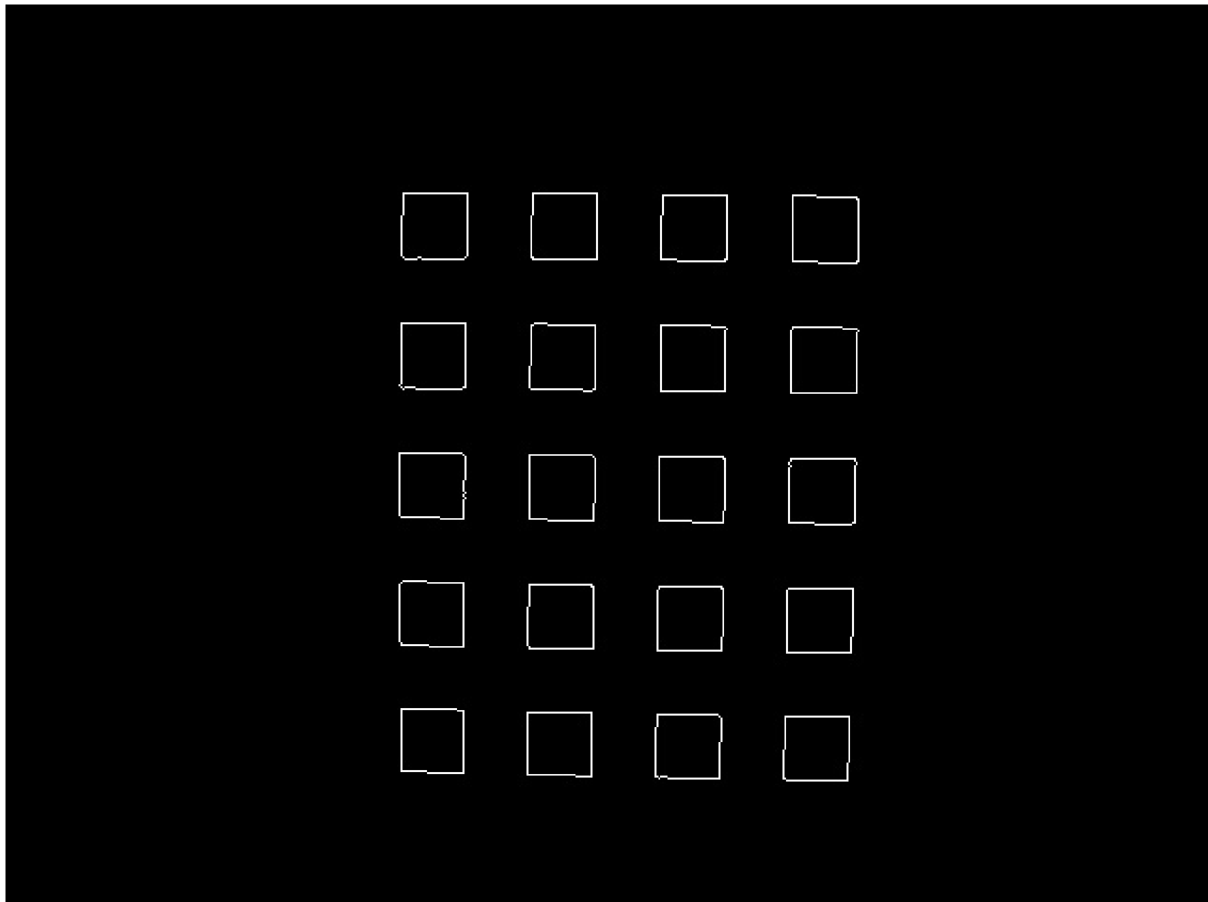

Figure 1: Canny Edge Detector for Image 11

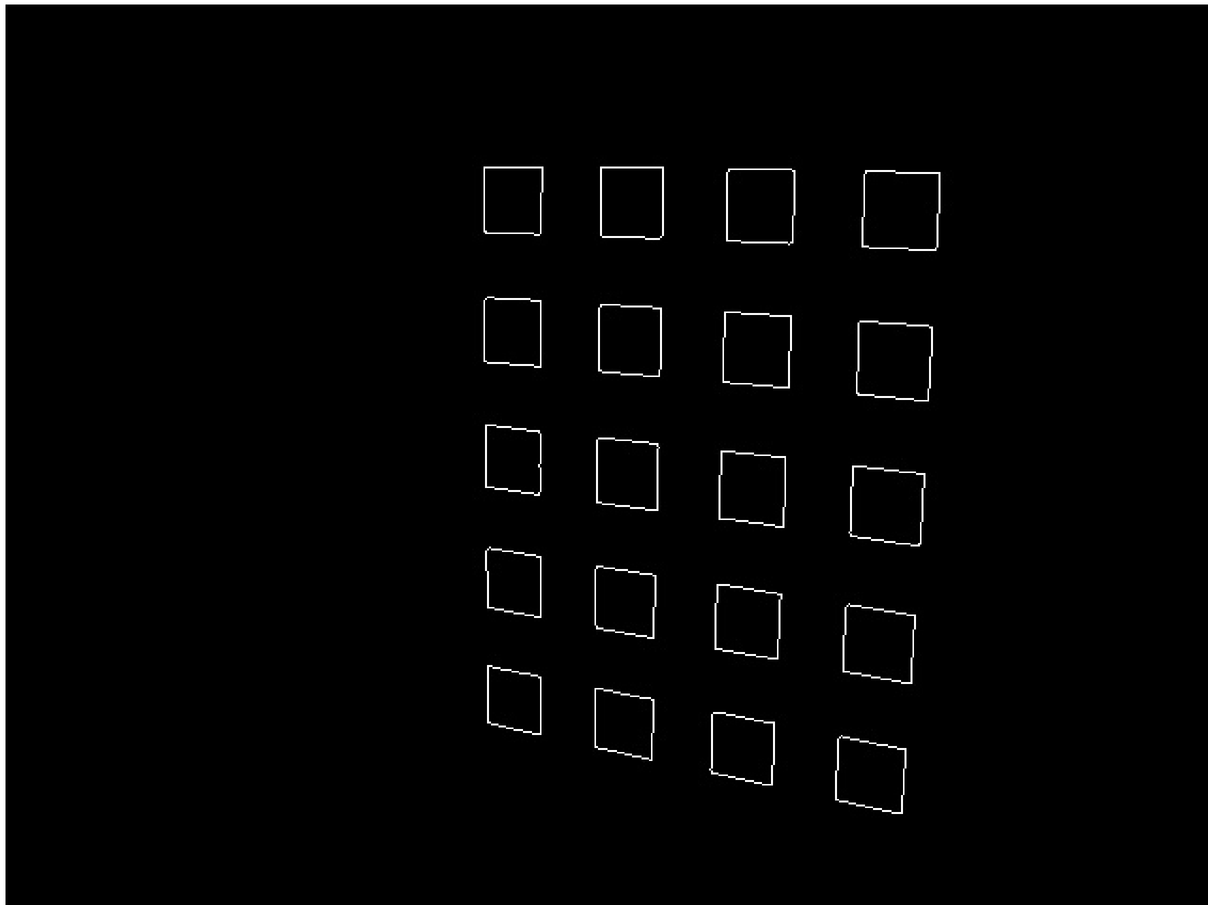

Figure 2: Canny Edge Detector for Image 39

Hough Line Fitting

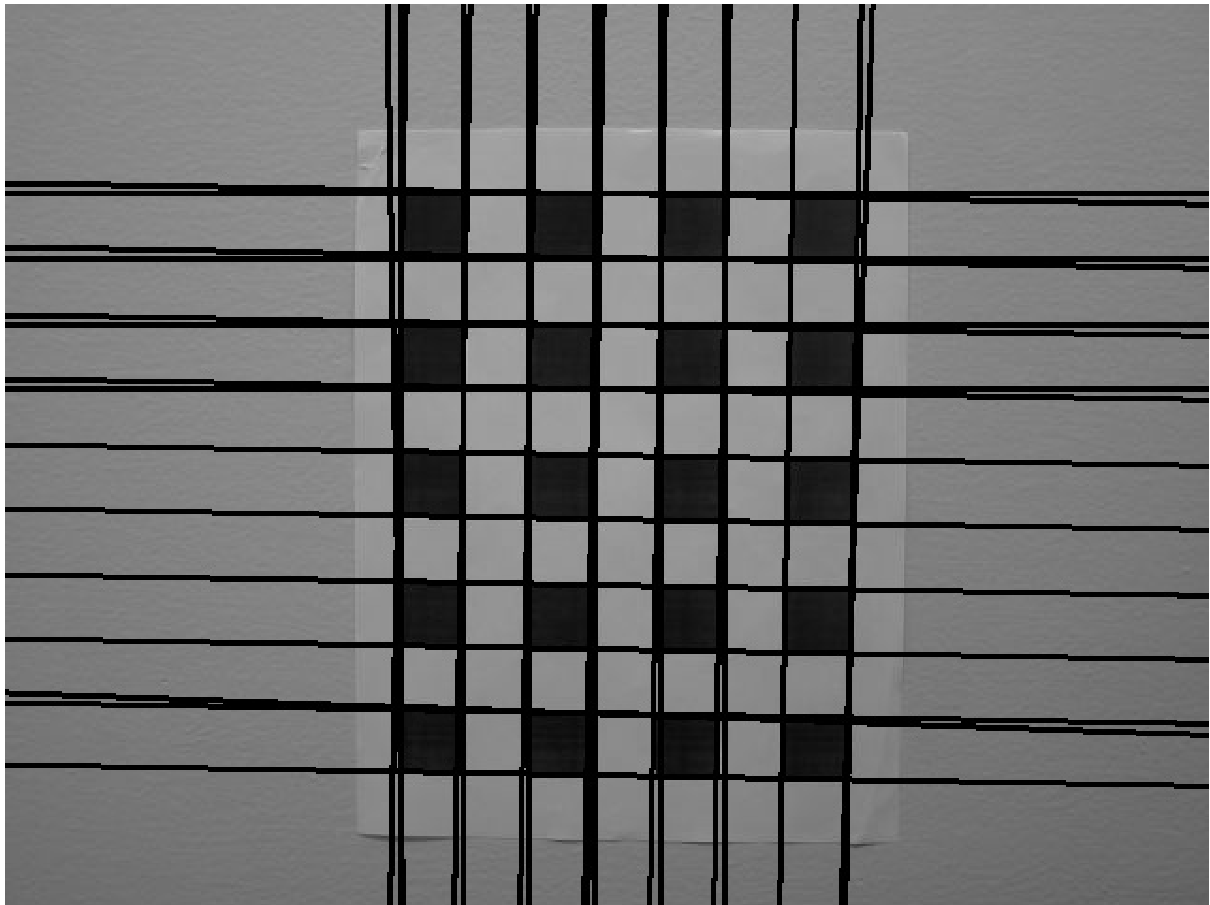

Figure 3: Hough Line Fitting for Image 11

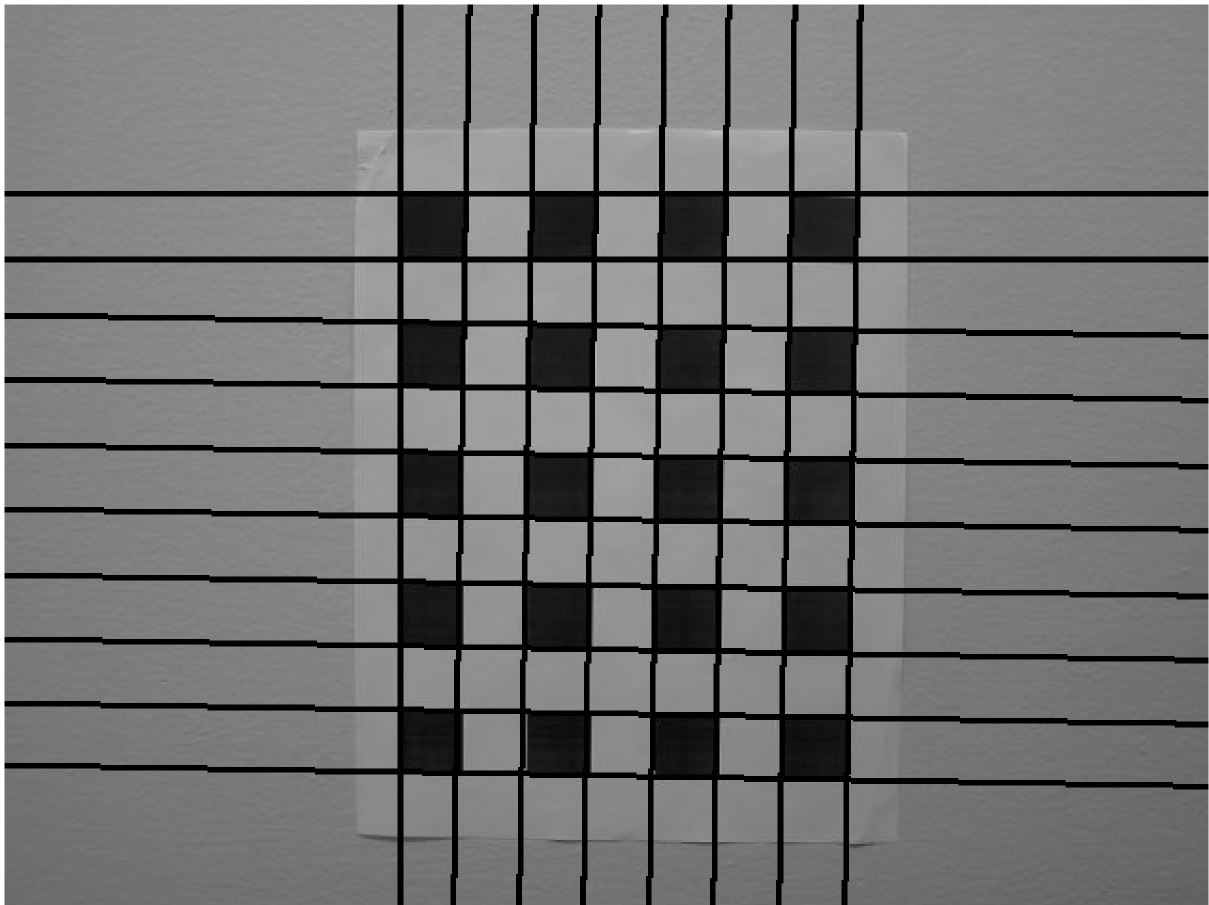

Figure 4: Final Hough Lines Selection for Image 11

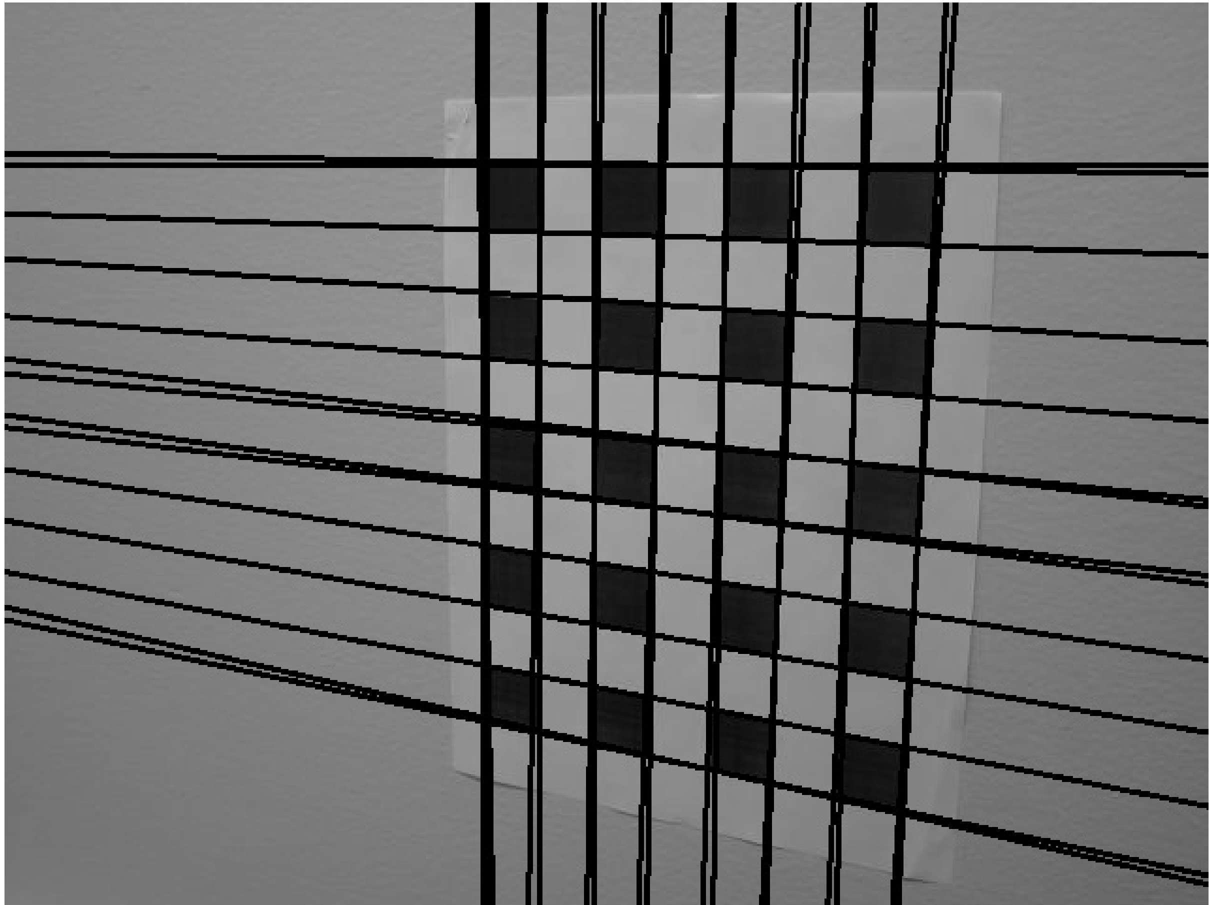

Figure 5: Hough Line Fitting for Image 39

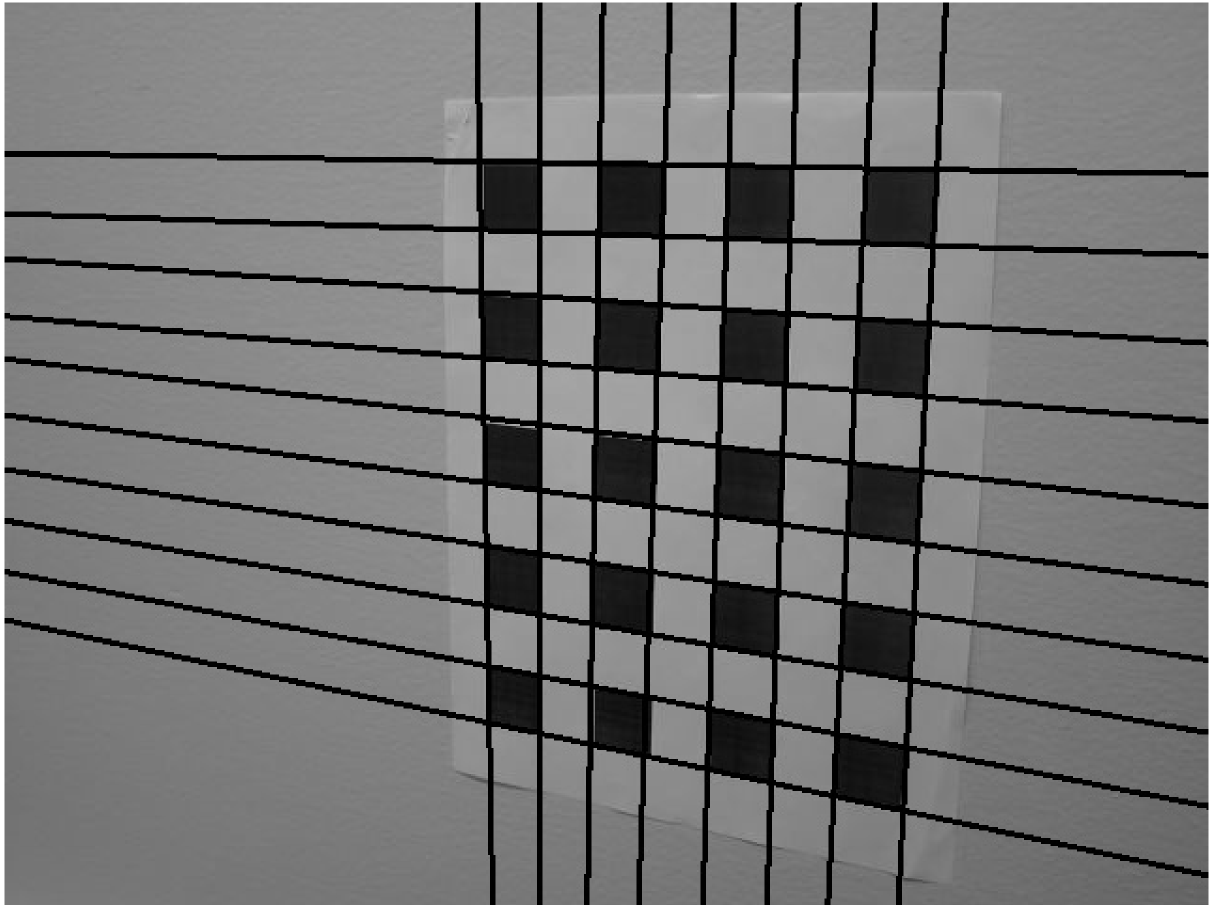

Figure 6: Final Hough Lines Selection for Image 39

Corner Detection

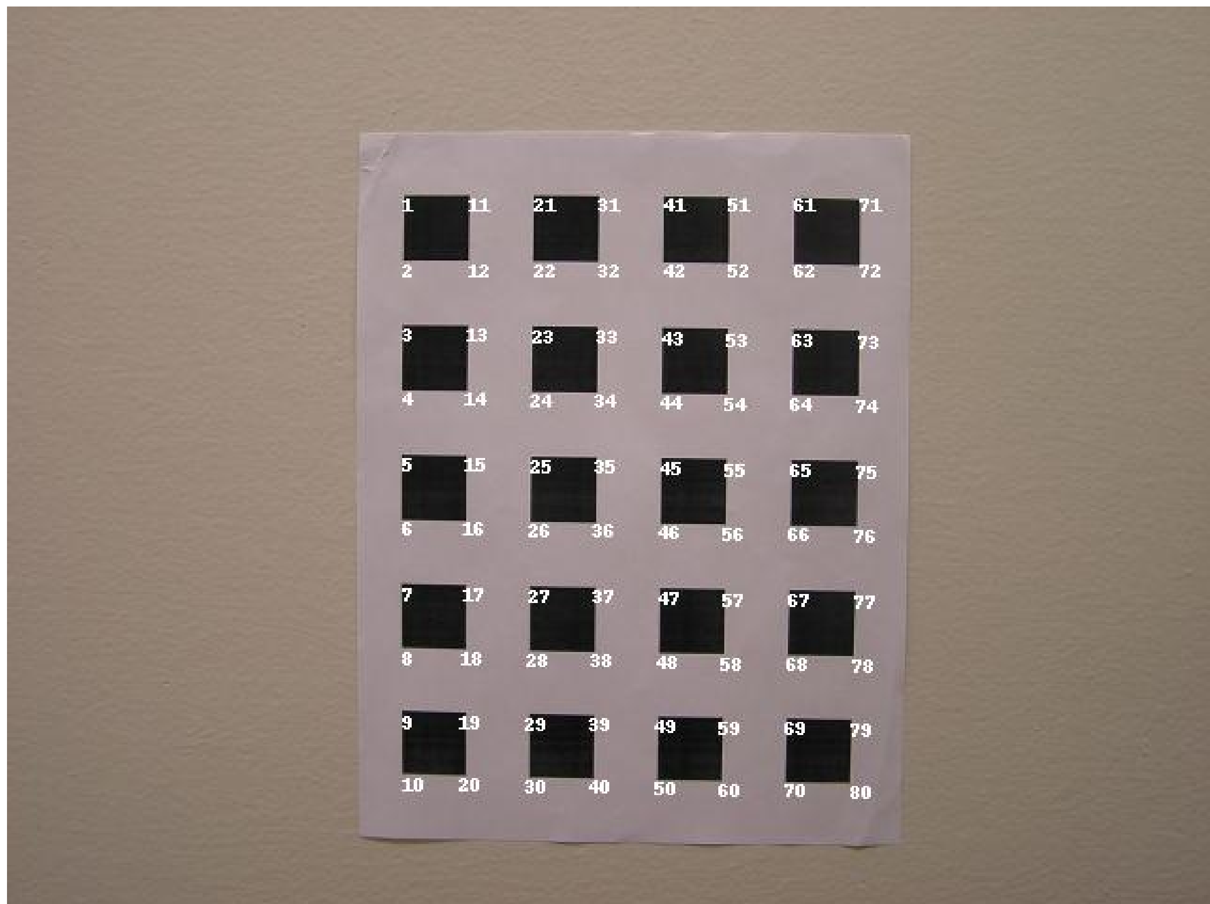

Figure 7: Corner Detection and labelling for Image 11

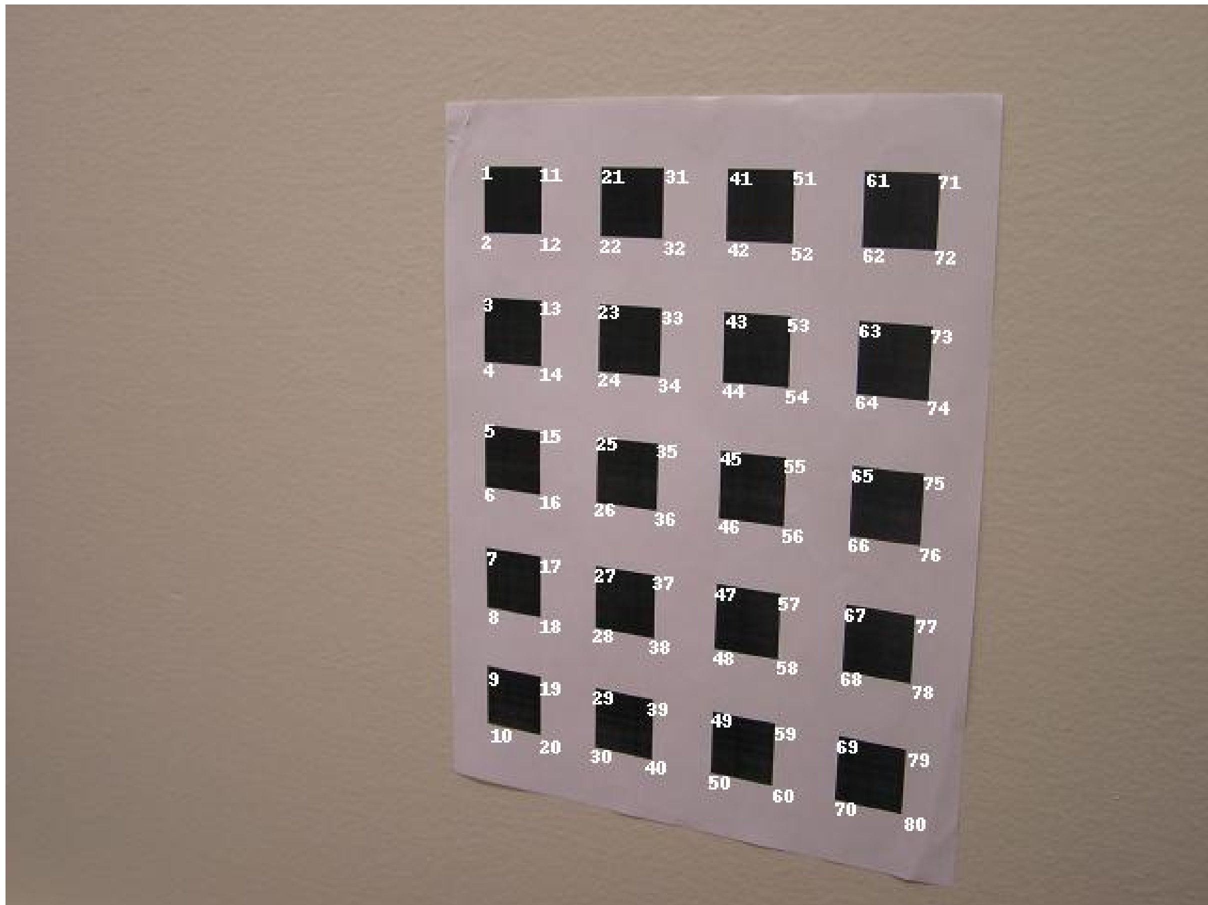

Figure 8: Corner Detection and labelling for Image 39

Reprojecting Corners

Without Radial Distortion

Before LM Refinement

Note: The labels tend to overlap with the reprojected points and reduce readability and clarity, hence have been removed. Please note that the labeling scheme is same as in the previous figure. The red points indicate the original intersections in the fixed image. And the green dots show the reprojected corners from the new images considered. Fixed Image : 11

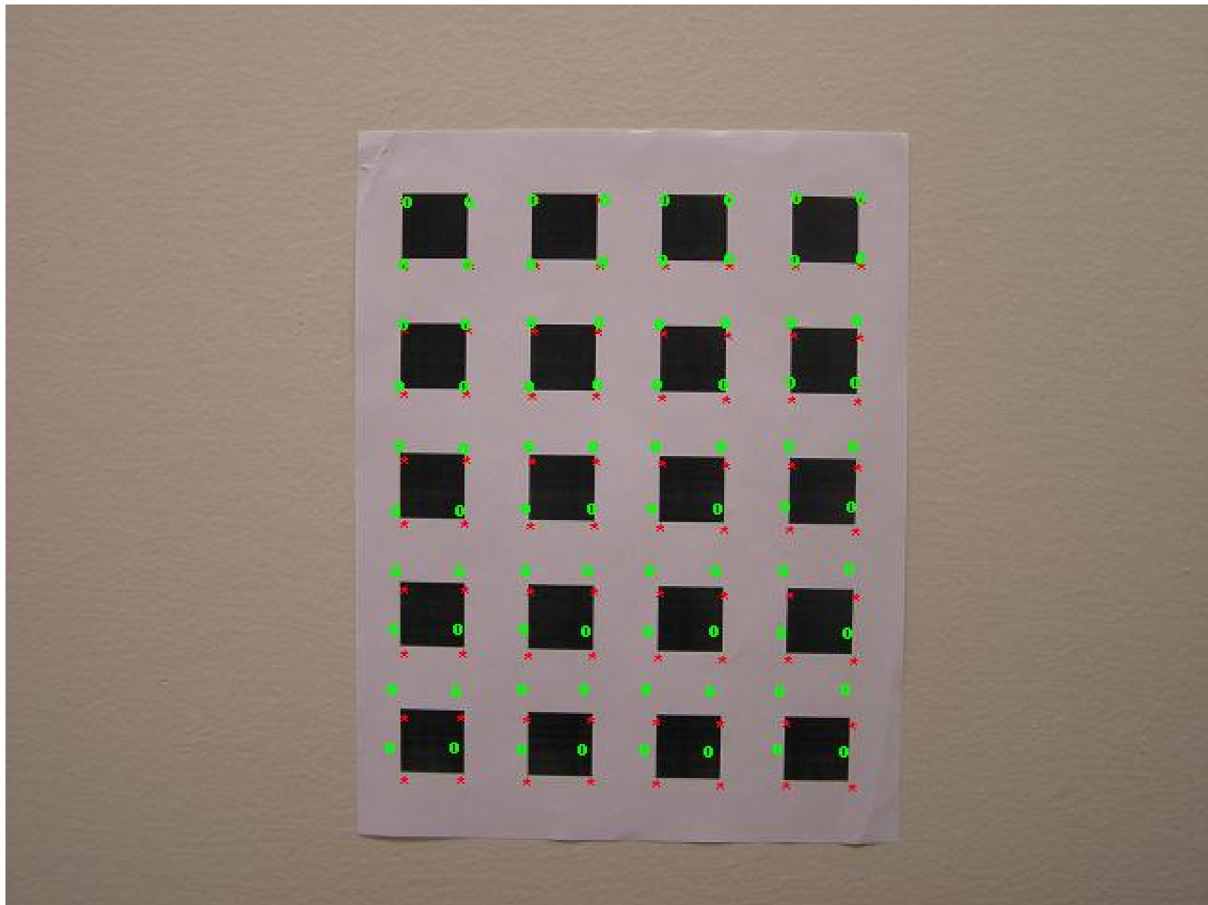

Figure 9: Before LM : Reprojecting Corners from Image 18 onto Image 11

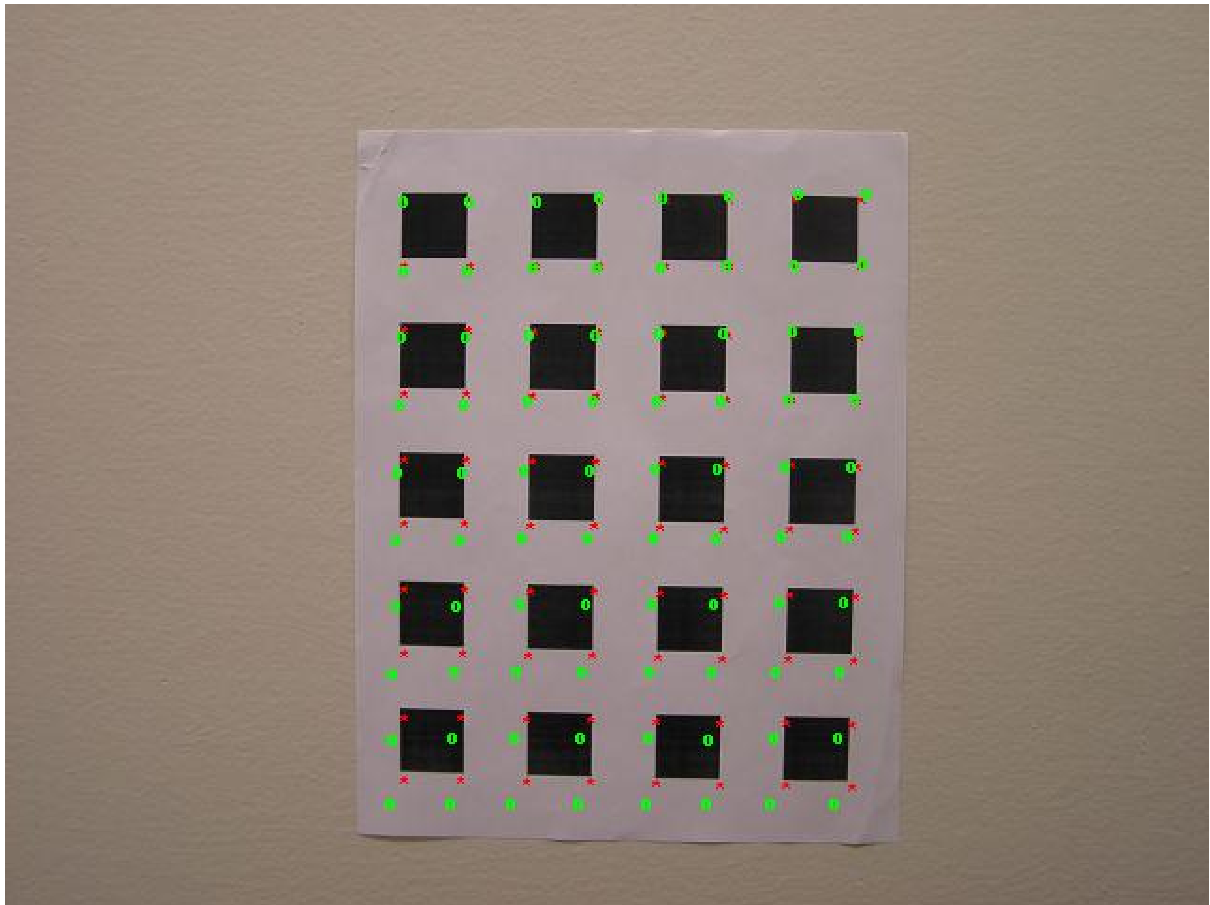

Figure 10: Before LM : Reprojecting Corners from Image 26 onto Image 11

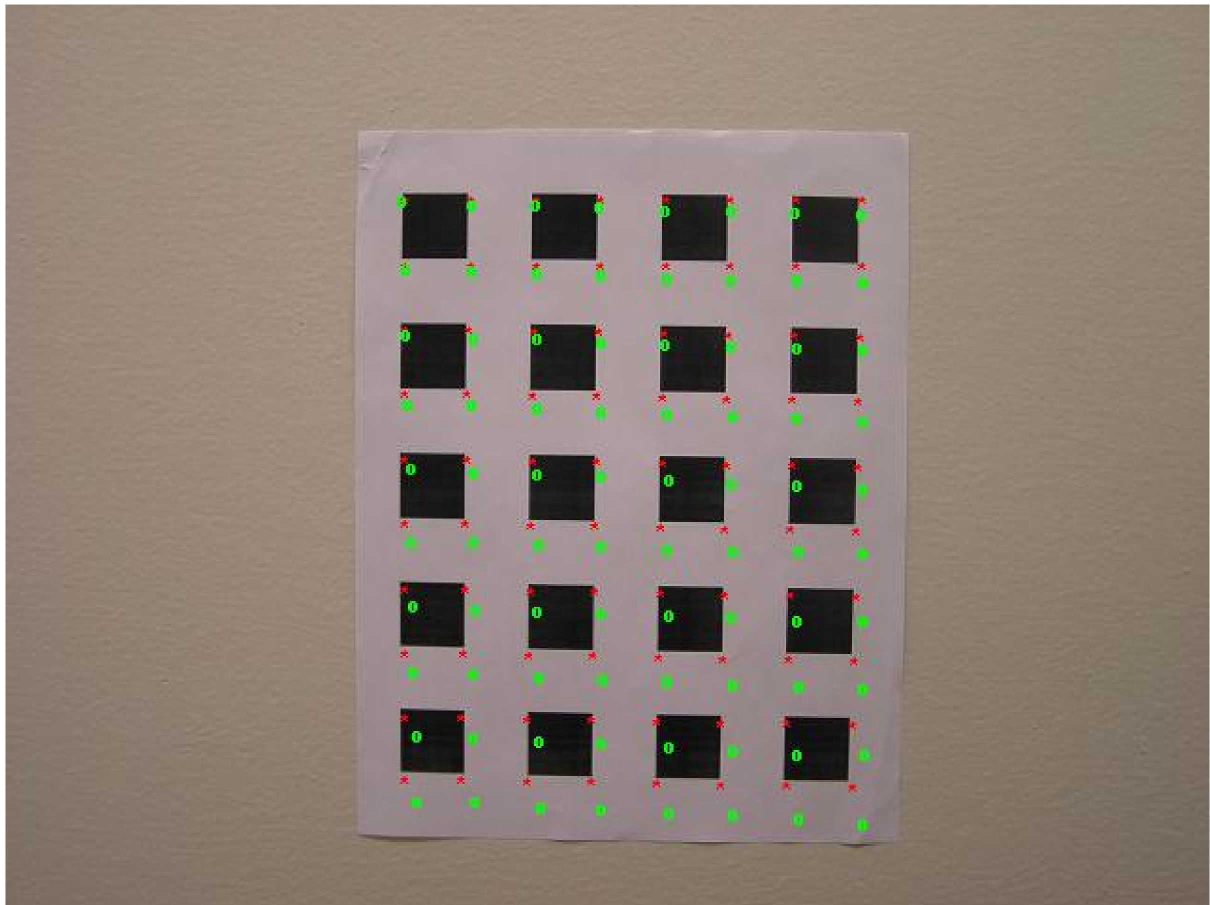

Figure 11: Before LM : Reprojecting Corners from Image 38 onto Image 11

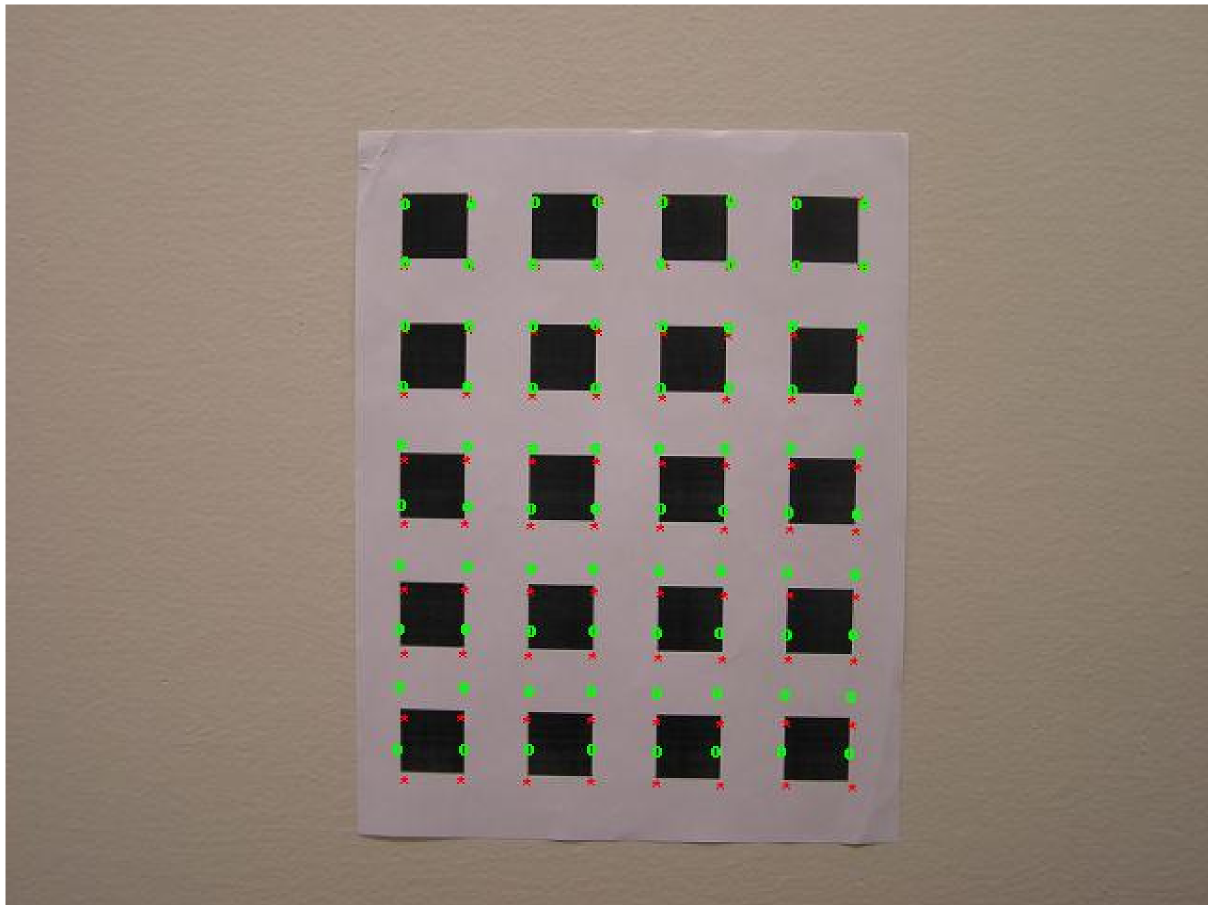

Figure 12: Before LM : Reprojecting Corners from Image 4 onto Image 11

After LM Refinement

Visually the improvement in performance can be seen in the camera calibration matrix.

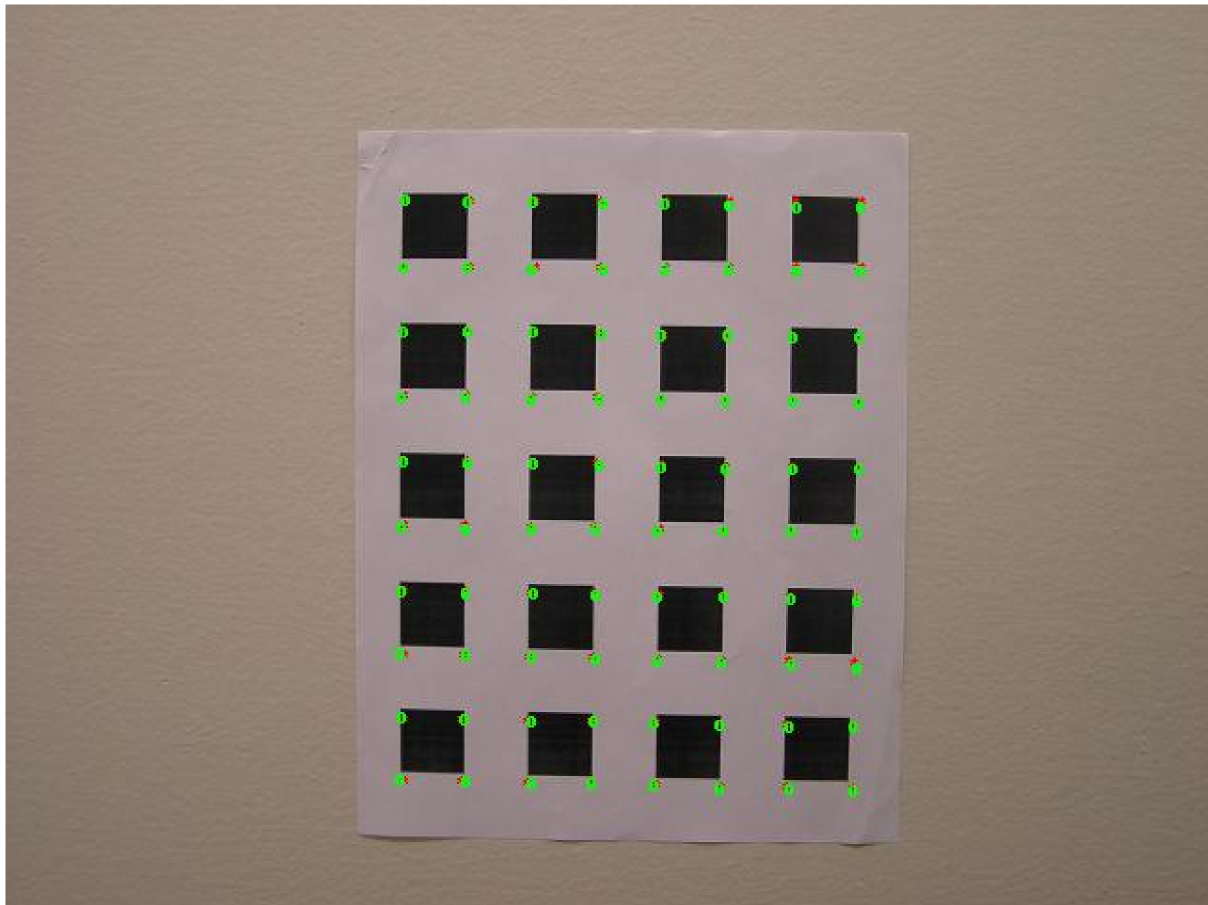

Figure 13: After LM : Reprojecting Corners from Image 18 onto Image 11

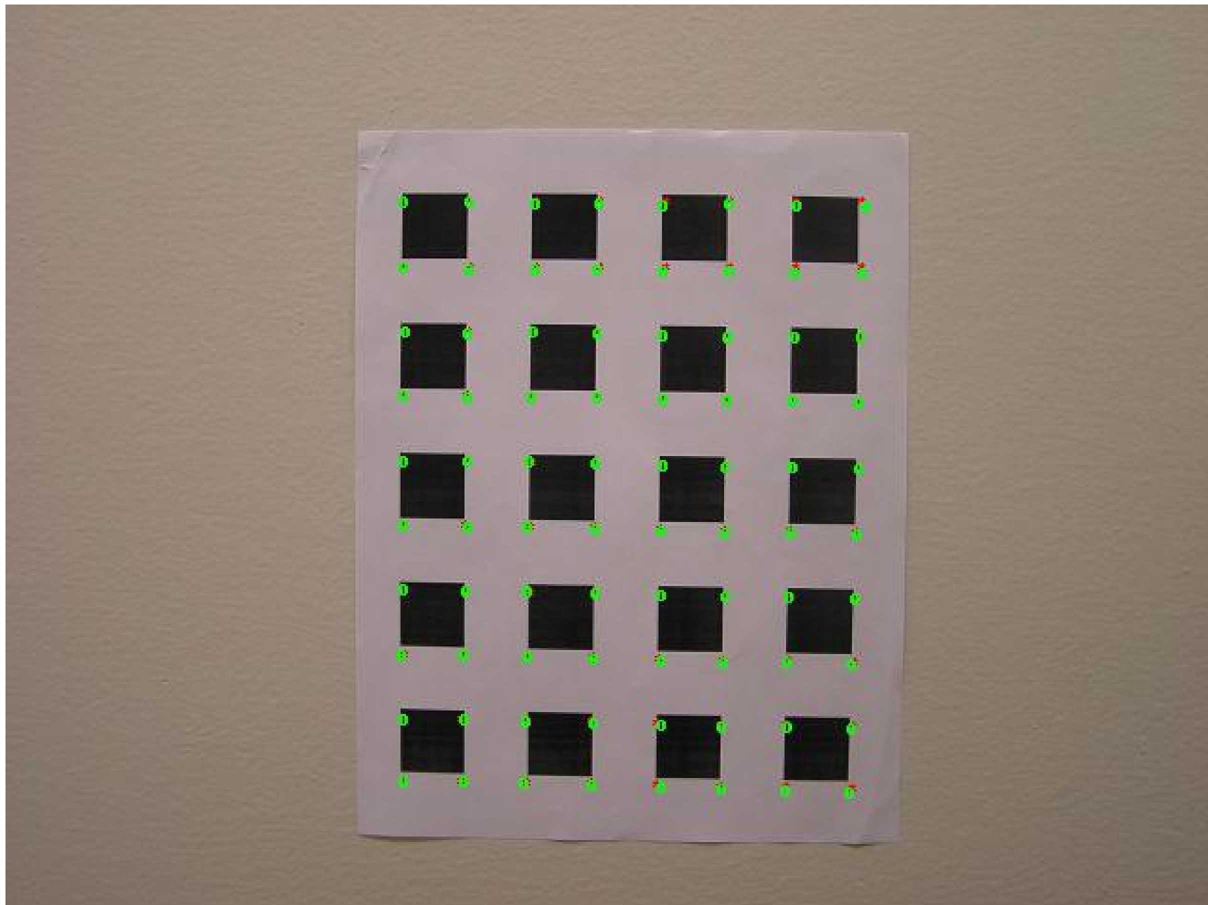

Figure 14: After LM : Reprojecting Corners from Image 26 onto Image 11

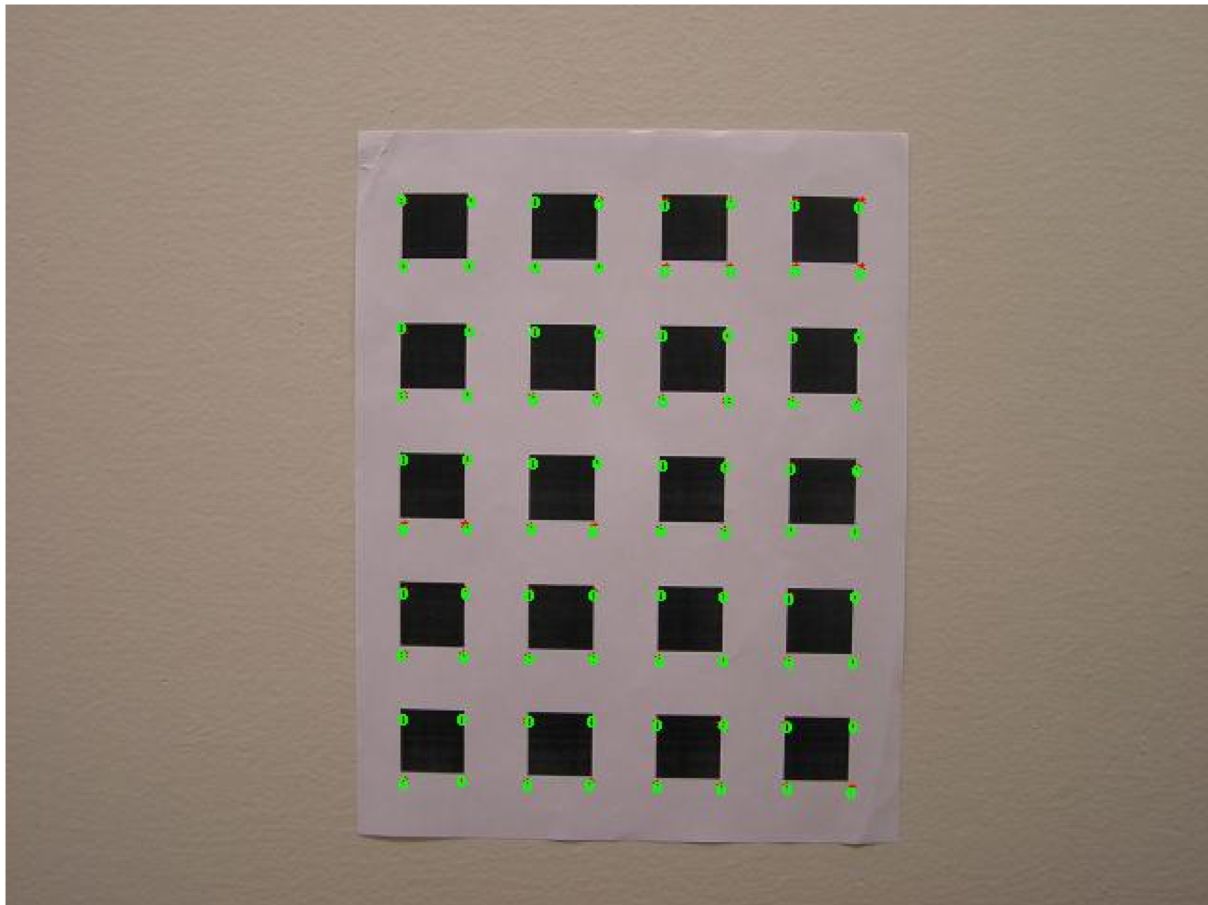

Figure 15: After LM : Reprojecting Corners from Image 38 onto Image 11

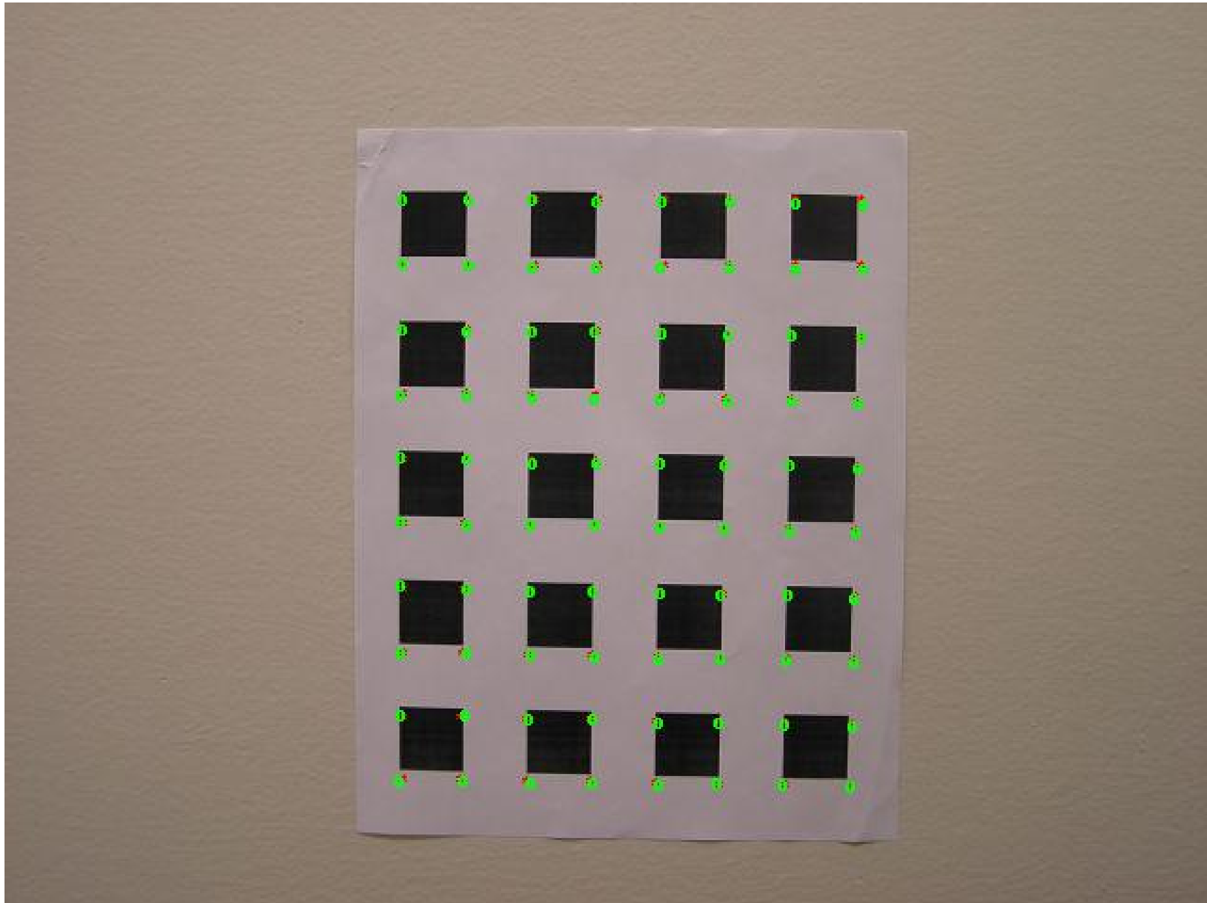

Figure 16: After LM : Reprojecting Corners from Image 4 onto Image 11

Quantitative Improvement in performance with LM

	Mean before LM	Mean after Lm	Variance before LM	Variance after LM
Image 4	9.9725	1.3052	34.0318	0.4088
Image 18	11.1668	1.2582	34.1461	0.4057
Image 26	6.0017	1.0692	13.1052	0.3827
Image 38	8.9710	1.0053	18.7095	0.2904

BEFORE LM:

Intrinsic Camera Parameters

$$K = \begin{bmatrix} 716.24497765 & 1.47782936 & 239.16052624 \\ 0. & 715.74408647 & 321.36183807 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters

Image 1:

$$[R|t] = \begin{bmatrix} 7.62691124e-01 & -2.06201323e-01 & 6.13011634e-01 & 1.61243877e+00 \\ 2.35284957e-01 & 9.71331684e-01 & 3.39963040e-02 & -1.95974898e+01 \\ -6.02447706e-01 & 1.18303737e-01 & 7.89342123e-01 & 5.65492919e+01 \end{bmatrix}$$

Image 9:

$$[R|t] = \begin{bmatrix} 0.88572776 & -0.13694307 & 0.44354586 & 1.55980245 \\ -0.12206312 & 0.85316327 & 0.50716174 & -17.20785084 \\ -0.44786932 & -0.50334782 & 0.73895469 & 62.6524857 \end{bmatrix}$$

Image 23:

$$[R|t] = \begin{bmatrix} 0.97799135 & -0.15641417 & -0.13808519 & -1.50646513 \\ 0.14380969 & 0.98483739 & -0.09702623 & -16.8801649 \\ 0.15116773 & 0.07503282 & 0.98565633 & 40.87504312 \end{bmatrix}$$

Image 36:

$$[R|t] = \begin{bmatrix} 9.86466485e-01 & 9.48183377e-02 & -1.33766049e-01 & -1.73853366e+00 \\ -1.00408311e-01 & 9.94306845e-01 & -3.56660867e-02 & -1.52316262e+01 \\ 1.29622699e-01 & 4.86146222e-02 & 9.90370928e-01 & 4.88057352e+01 \end{bmatrix}$$

AFTER LM:

Intrinsic Camera Parameters

$$K = \begin{bmatrix} 719.40373 & 1.97916453 & 322.79664827 \\ 0. & 718.44500803 & 240.31786387 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters

Image 5:

$$[R|t] = \begin{bmatrix} 0.98737613 & -0.15744477 & 0.01730673 & -5.45902104 \\ 0.15831604 & 0.98439495 & -0.07682845 & -12.09045442 \\ -0.00494042 & 0.07859851 & 0.99689411 & 51.90237641 \end{bmatrix}$$

Image 14:

$$[R|t] = \begin{bmatrix} 0.94906669 & -0.14001759 & -0.28225431 & -6.76112791 \\ 0.11692633 & 0.98837806 & -0.09714441 & -12.47227292 \\ 0.2925759 & 0.05919356 & 0.95440844 & 41.11965781 \end{bmatrix}$$

Image 28:

$$[R|t] = \begin{bmatrix} 0.99905958 & -0.01116927 & -0.04189508 & -8.24390961 \\ 0.0085102 & 0.99796969 & -0.06311952 & -10.78413141 \\ 0.04251502 & 0.06270362 & 0.99712624 & 38.42402536 \end{bmatrix}$$

Image 36:

$$[R|t] = \begin{bmatrix} 0.98710395 & 0.08375135 & -0.13642395 & -7.59006047 \\ -0.0910927 & 0.9946617 & -0.04847904 & -9.90366684 \\ 0.13163549 & 0.06028108 & 0.98946364 & 49.76055356 \end{bmatrix}$$

LM With Radial Distortion Accommodated

Radial Distortion Parameters:

k1=-2.2049e-07

k2=1.2666e-12

Some small improvements can be seen when radial distortion is incorporated.

	Mean before LM	Mean after Lm(RD)	Variance before LM	Variance after LM(RD)
Image 4	9.9725	1.2820	34.0318	0.4161
Image 18	11.1668	1.2880	34.1461	0.4542
Image 26	6.0017	1.0824	13.1052	0.5038
Image 38	8.9710	1.0164	18.7095	0.2790

Intrinsic Camera Parameters with LM with Radial Distortion Incorporated

$$K = \begin{bmatrix} 723.53767828 & 1.96835421 & 321.83508048 \\ 0. & 722.85850844 & 241.01205454 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters with LM with Radial Distortion Incorporated

Image 6:

$$[R|t] = \begin{bmatrix} 0.98905528 & 0.13690583 & -0.05501314 & -8.8631359 \\ -0.13829612 & 0.99013992 & -0.02229616 & -8.81858199 \\ 0.05141824 & 0.02966024 & 0.99823666 & 63.8707372 \end{bmatrix}$$

Measured Ground Truth wrt Fixed Image

The [R|t] matrix for fixed image 11 is shown below. It is seen that R approximately corresponds to I_{3X3} identity matrix and the t vector gives displacement and depth. This verifies the accuracy of the implementation.

$$[R|t] = \begin{bmatrix} 0.99910334 & -0.01517218 & 0.03952631 & -8.12783912 \\ 0.01675176 & 0.99906152 & -0.03994306 & -10.1948547 \\ -0.03888319 & 0.04056938 & 0.99841986 & 51.98057851 \end{bmatrix}$$

My Dataset Results

25 images of the calibration pattern have been taken from different angles

Edge Detection

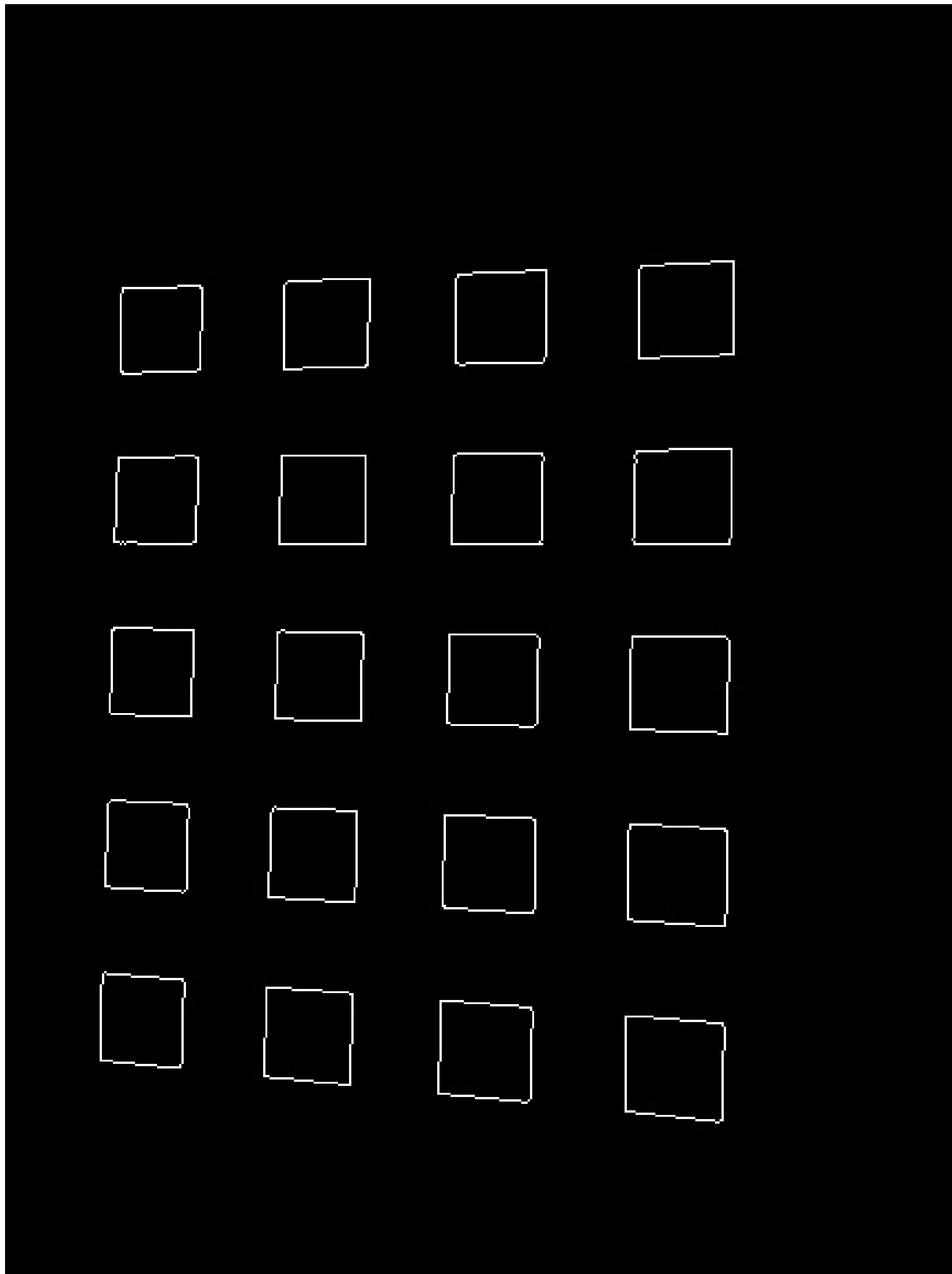

Figure 17: Canny Edge Detector for Image 3

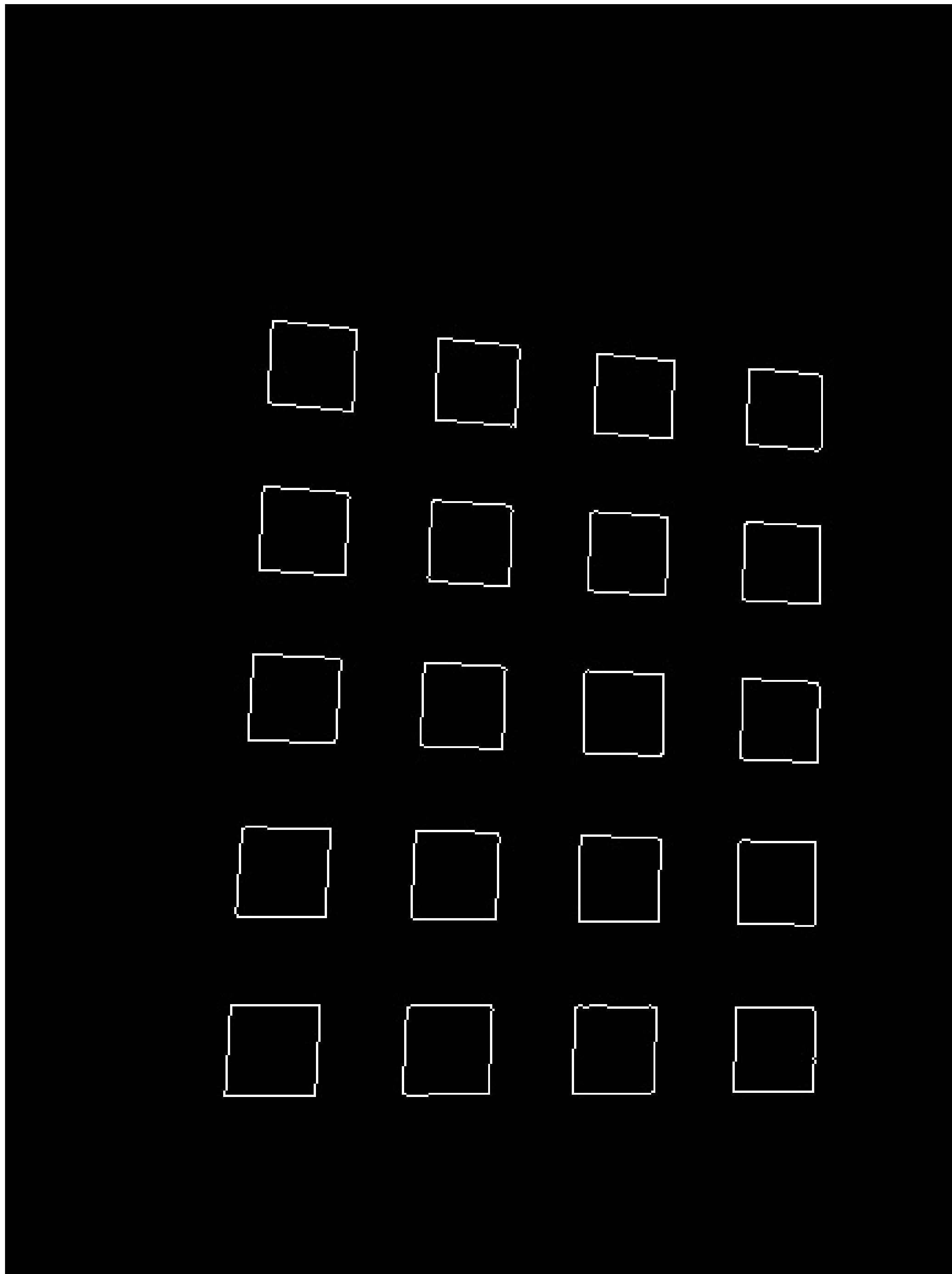

Figure 18: Canny Edge Detector for Image 24

Hough Line Fitting

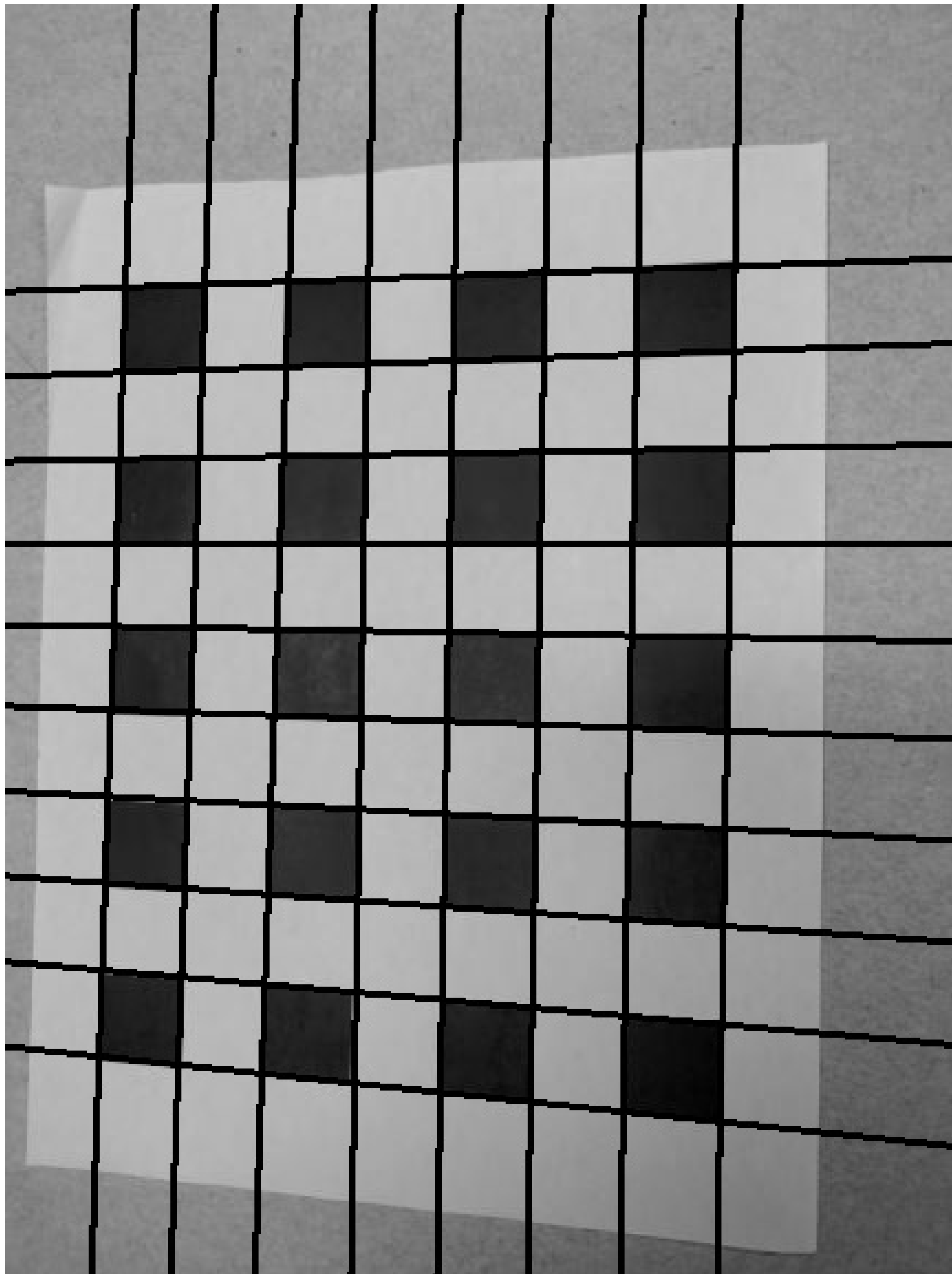

Figure 19: Hough Line Fitting for Image 3

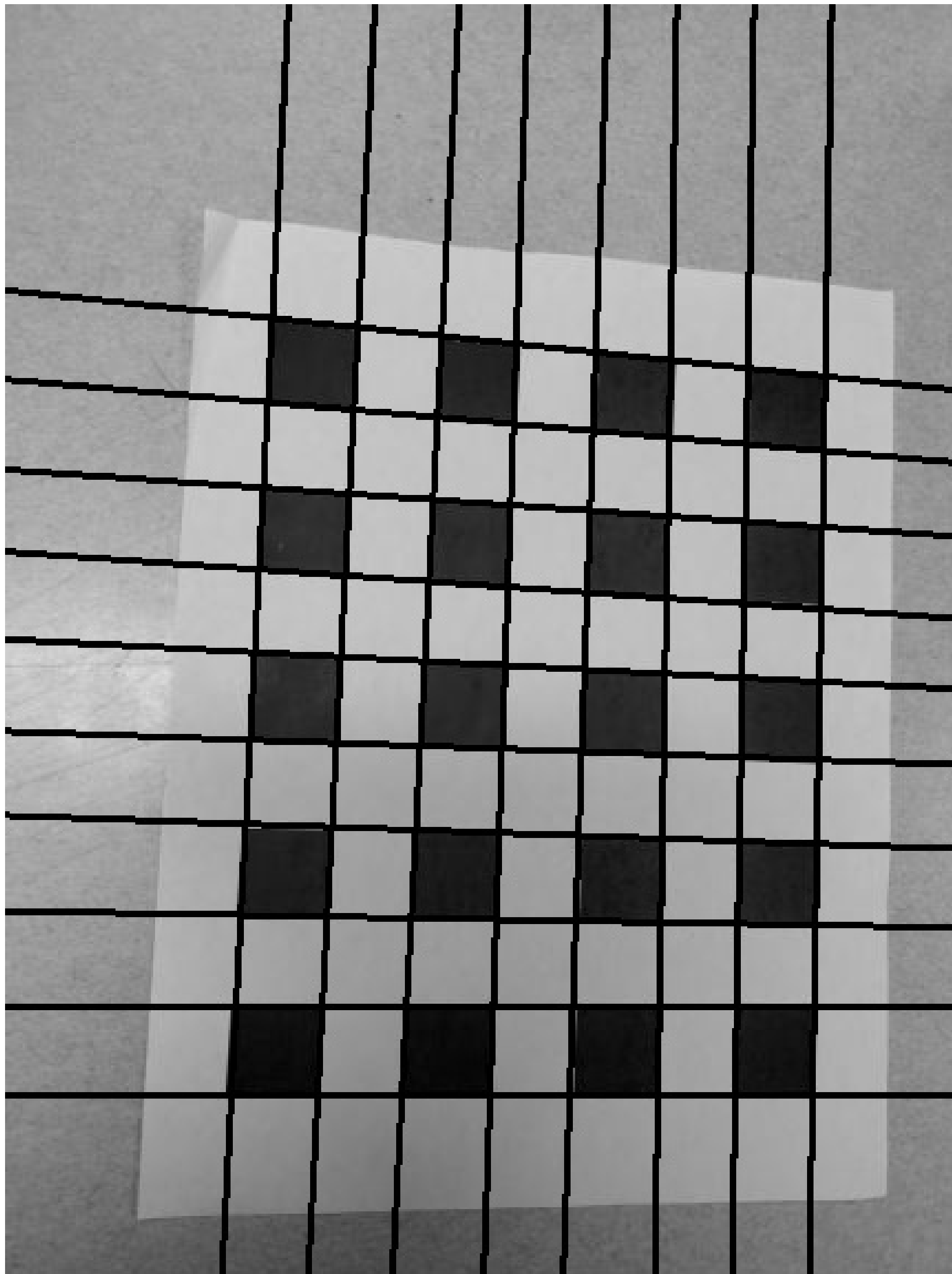

Figure 20: Hough Line Fitting for Image 24

Corner Detection

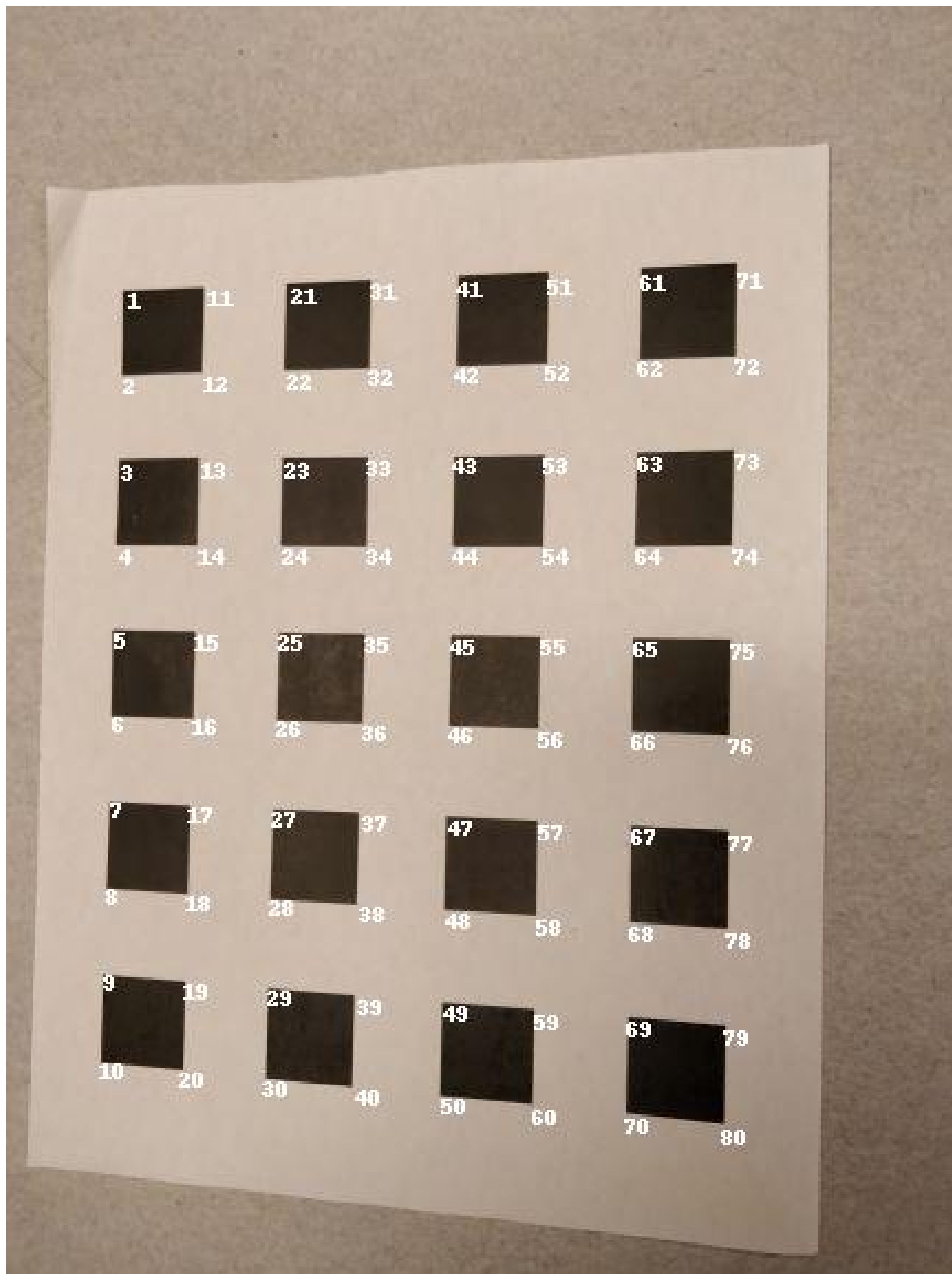

Figure 21: Corner Detection and Labeling for Image 3

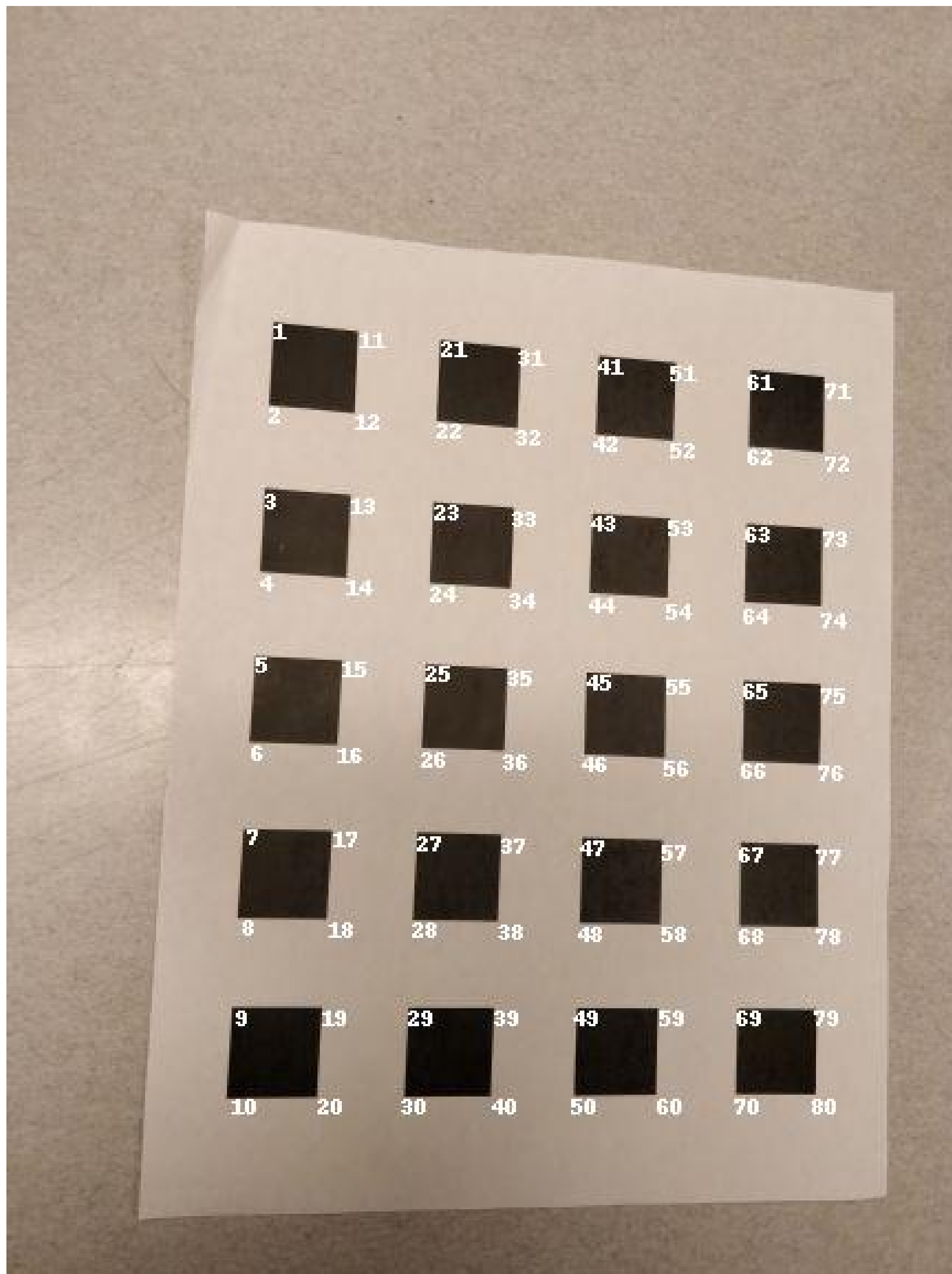

Figure 22: Corner Detection and Labeling for Image 24

Reprojecting Corners

Fixed Image = 1

Without Radial Distortion

Before LM Refinement

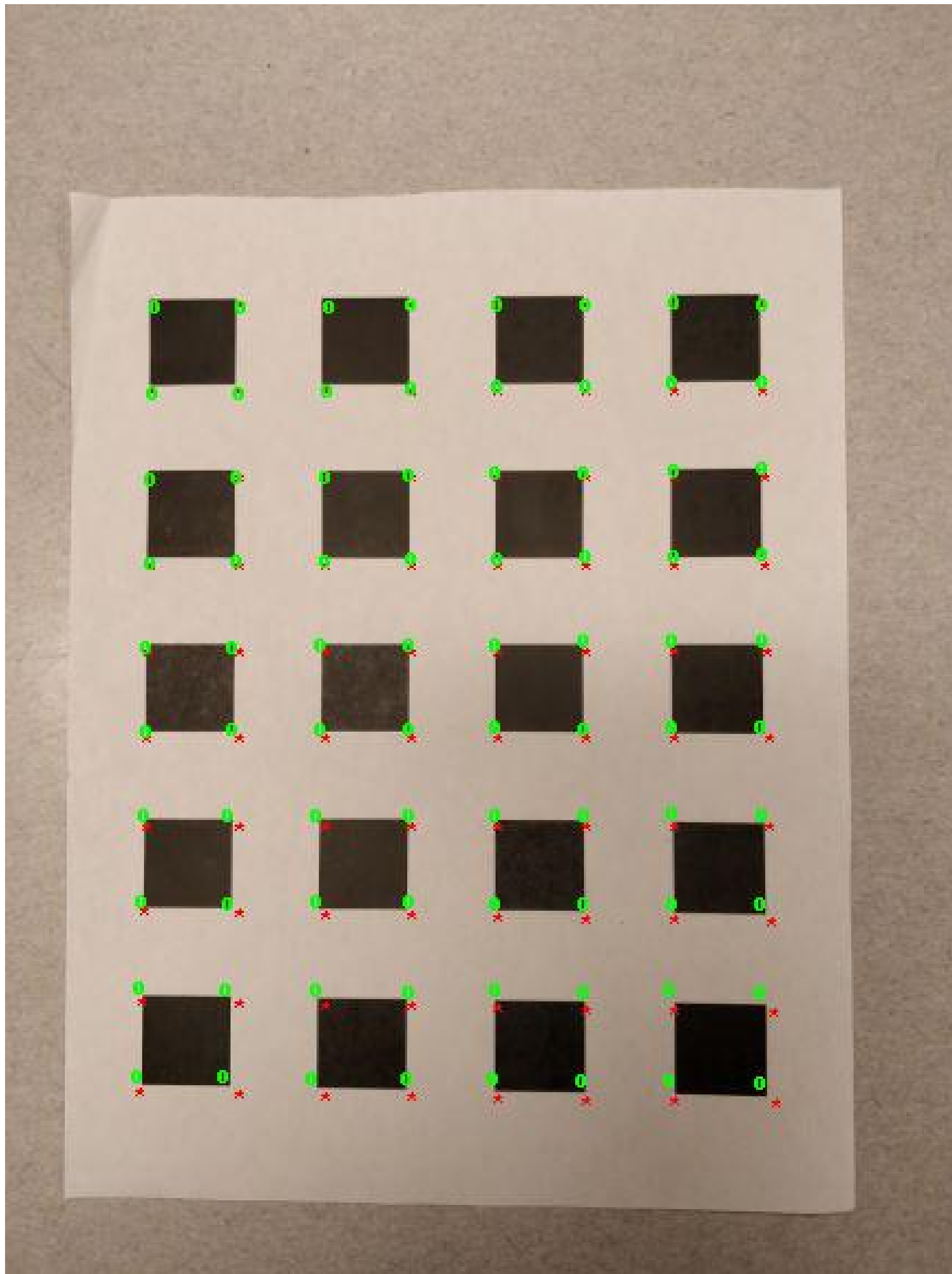

Figure 23: Before LM : Reprojecting Corners from Image 3 onto Image 1

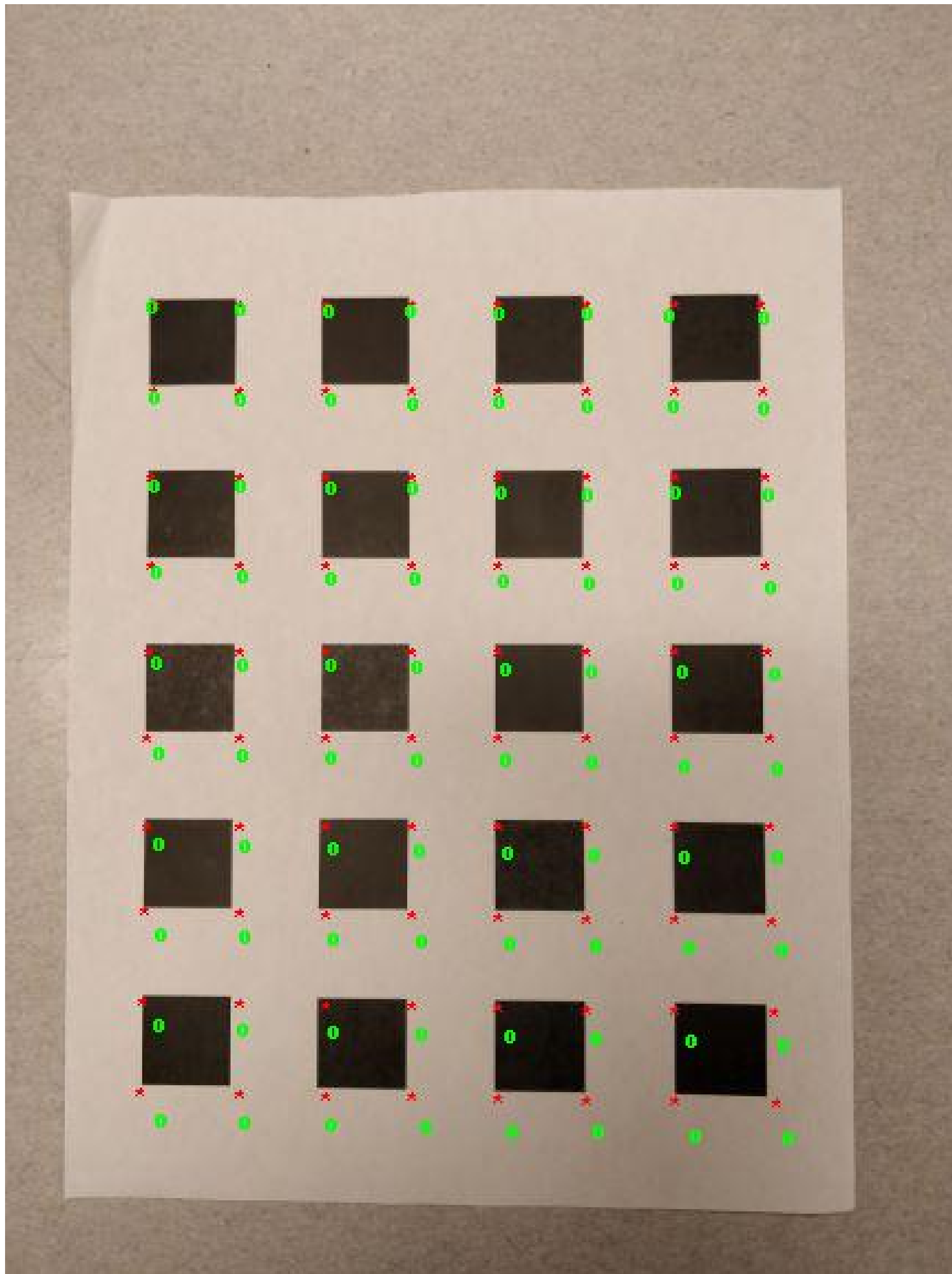

Figure 24: Before LM : Reprojecting Corners from Image 8 onto Image 1

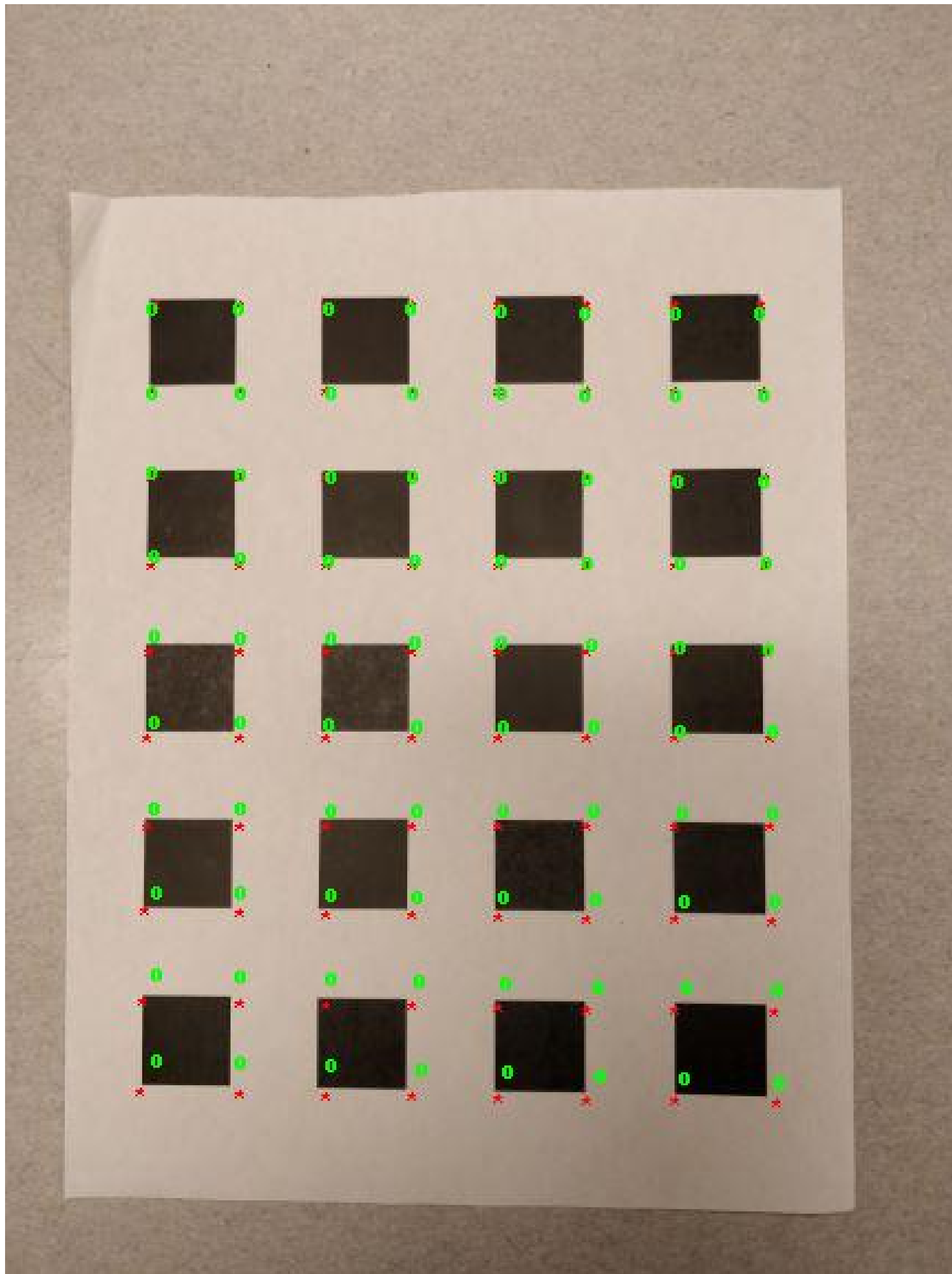

Figure 25: Before LM : Reprojecting Corners from Image 15 onto Image 1

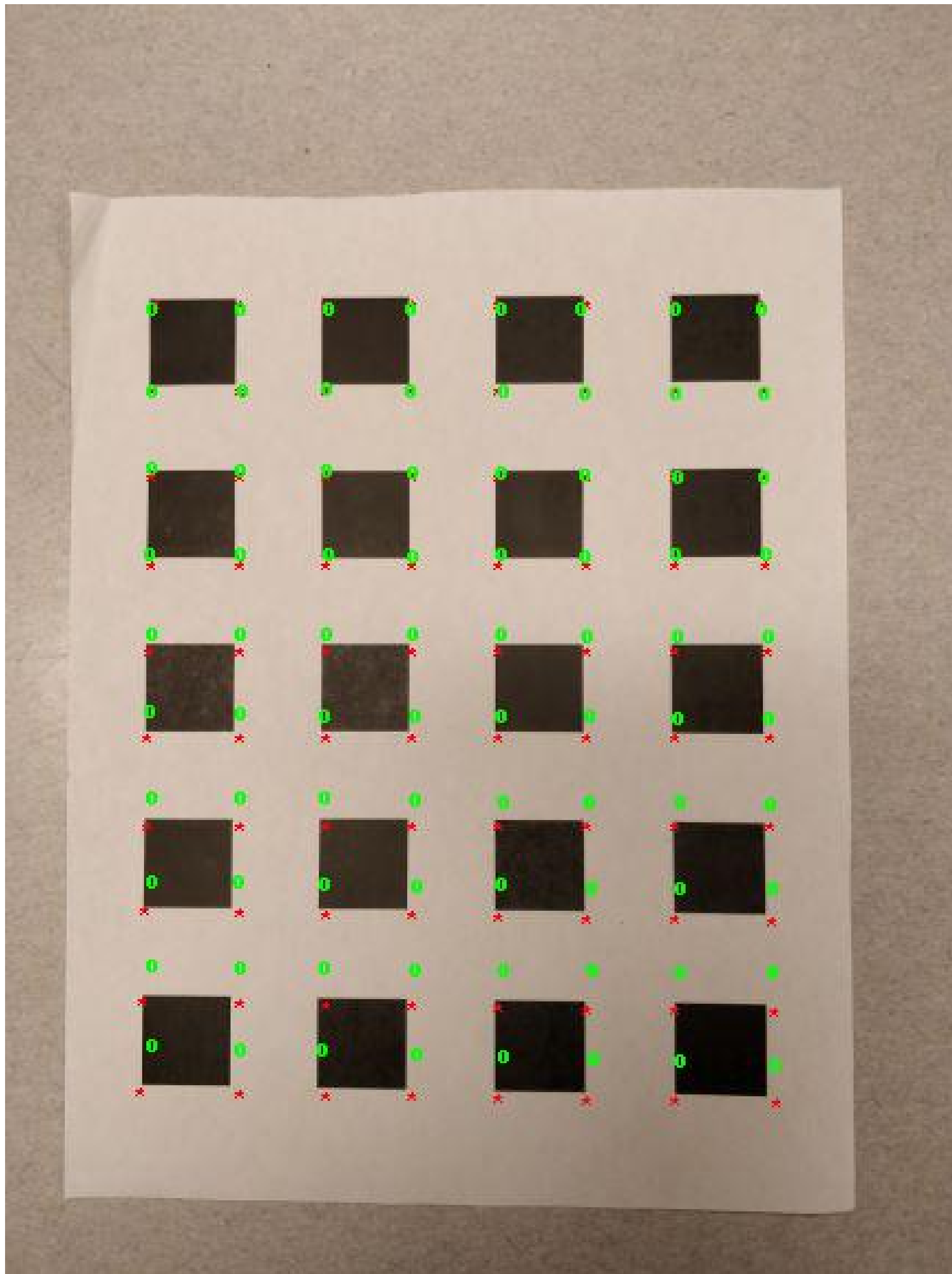

Figure 26: Before LM : Reprojecting Corners from Image 22 onto Image 1

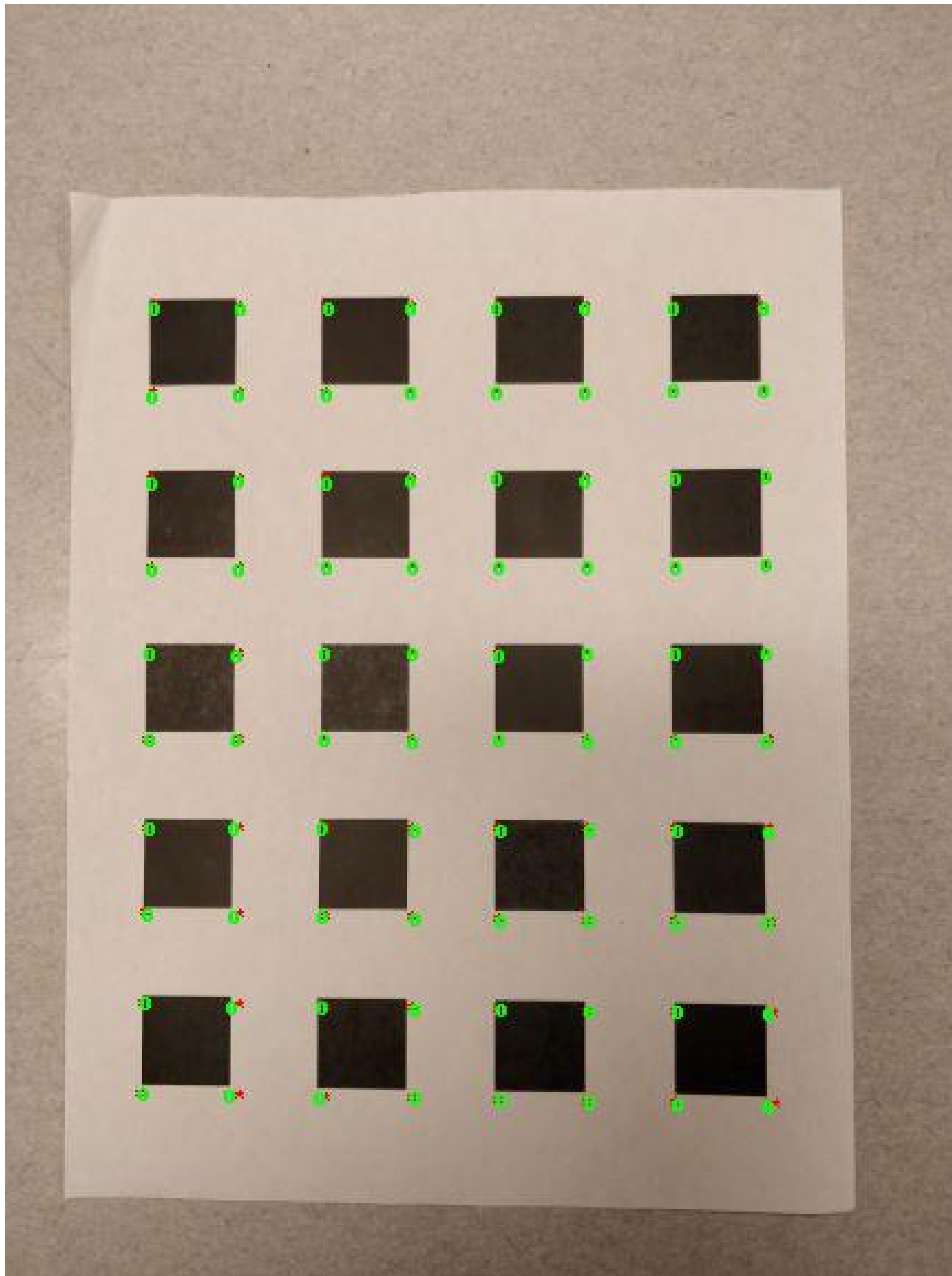

Figure 27: After LM : Reprojecting Corners from Image 3 onto Image 1

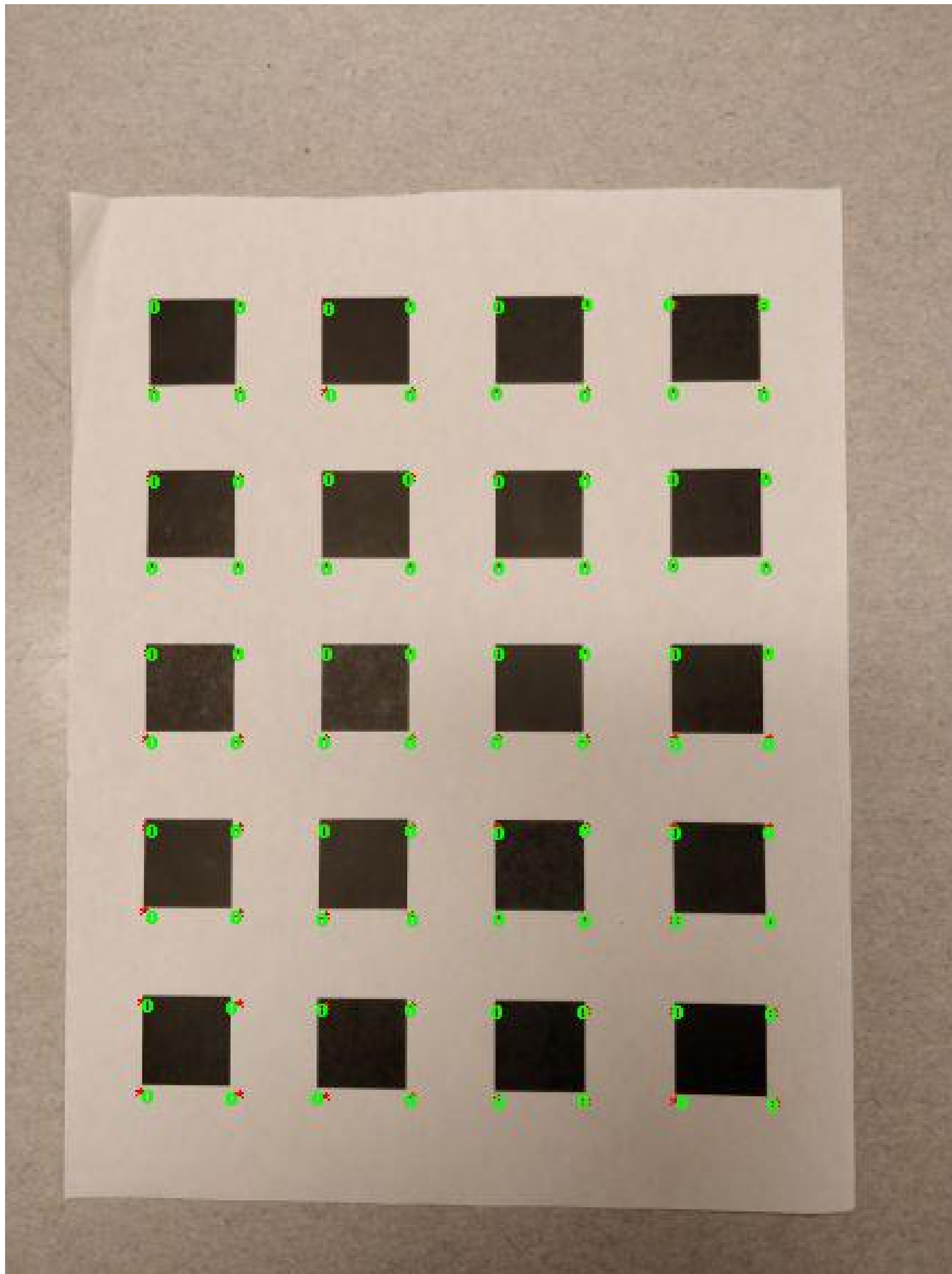

Figure 28: After LM : Reprojecting Corners from Image 8 onto Image 1

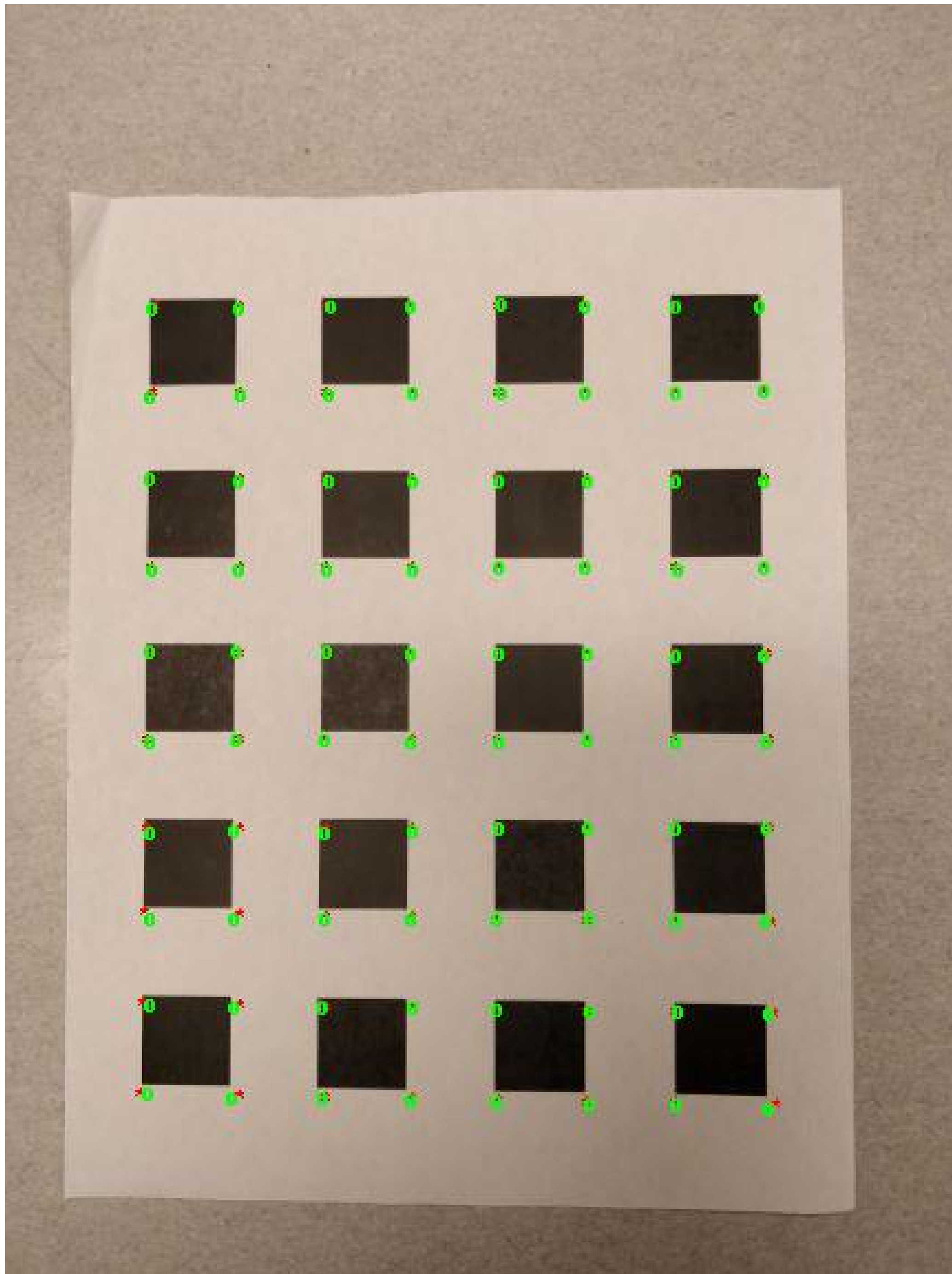

Figure 29: After LM : Reprojecting Corners from Image 15 onto Image 1

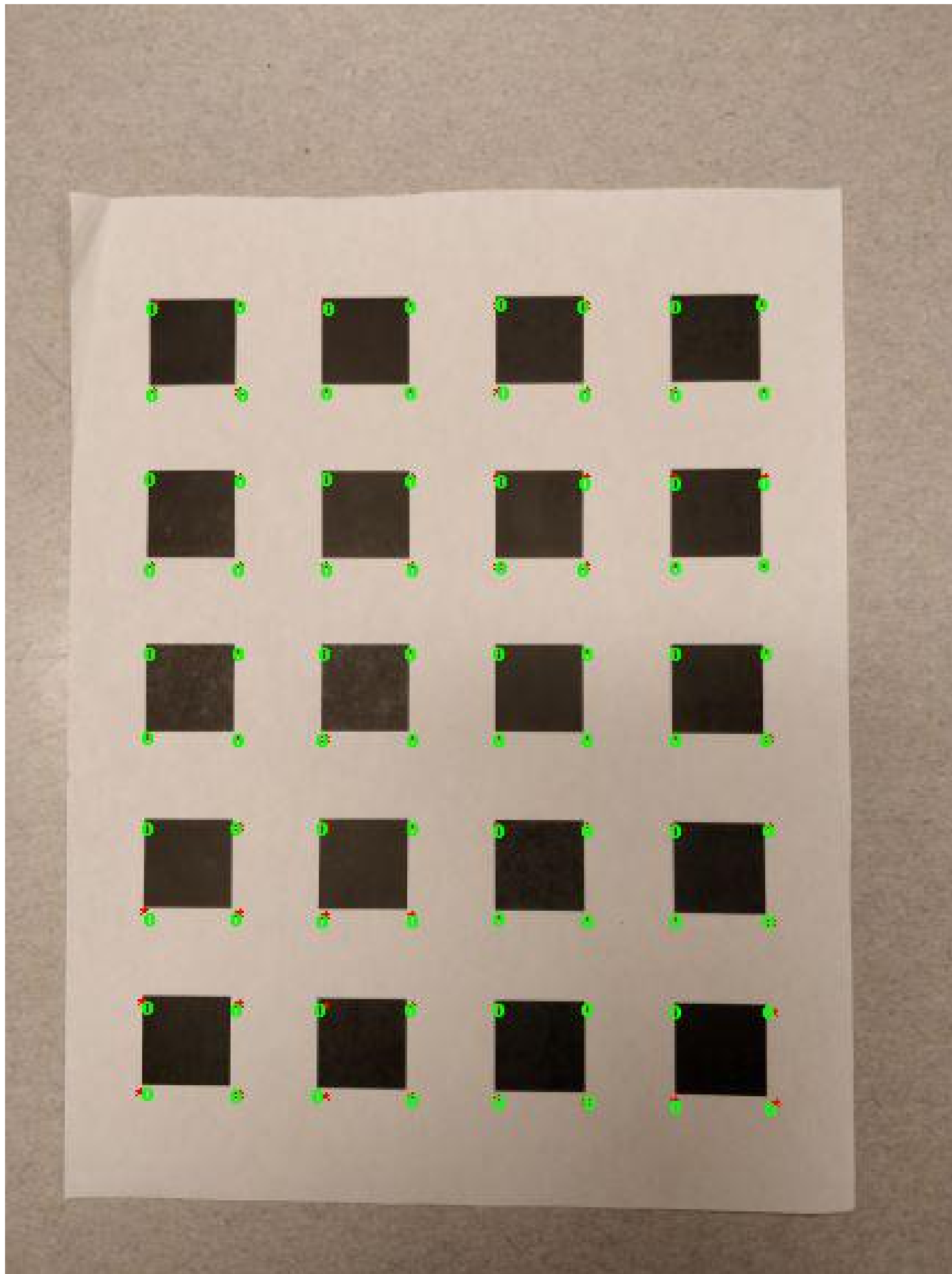

Figure 30: After LM : Reprojecting Corners from Image 22 onto Image 1

Quantitative Improvement in performance with LM

Significant improvements seen as shown below

	Mean before LM	Mean after Lm	Variance before LM	Variance after LM
Image 3	5.9954	1.3396	10.0602	0.6143
Image 8	8.2099	1.2970	17.2305	0.6852
Image 15	6.8268	1.2408	21.5458	0.6698
Image 22	10.2113	1.2489	44.8102	0.5724

BEFORE LM:

Intrinsic Camera Parameters

$$K = \begin{bmatrix} 502.4296083 & 0.61132347 & 278.62494628 \\ 0. & 501.46835879 & 216.71339826 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters

Image 4:

$$[R|t] = \begin{bmatrix} 9.94113602e-01 & -9.15218810e-03 & 1.07955475e-01 & -1.40118666e+01 \\ 7.46803629e-03 & 9.99844191e-01 & 1.59944241e-02 & -5.49575391e+00 \\ -1.08085039e-01 & -1.50940591e-02 & 9.94027059e-01 & 3.14371129e+01 \end{bmatrix}$$

Image 9:

$$[R|t] = \begin{bmatrix} 0.96954786 & -0.03438354 & -0.24247624 & -9.35999228 \\ 0.05562281 & 0.9951366 & 0.08129725 & -4.74431806 \\ 0.23850169 & -0.09230879 & 0.96674507 & 33.47898277 \end{bmatrix}$$

Image 16:

$$[R|t] = \begin{bmatrix} 9.75854507e-01 & -4.78890812e-02 & 2.13107055e-01 & -1.33306434e+01 \\ -3.28138809e-02 & 9.32452361e-01 & 3.59799728e-01 & -4.03134499e+00 \\ -2.15942655e-01 & -3.58105056e-01 & 9.08366412e-01 & 3.86024584e+01 \end{bmatrix}$$

Image 25:

$$[R|t] = \begin{bmatrix} 9.94113602e-01 & -9.15218810e-03 & 1.07955475e-01 & -1.40118666e+01 \\ 7.46803629e-03 & 9.99844191e-01 & 1.59944241e-02 & -5.49575391e+00 \\ -1.08085039e-01 & -1.50940591e-02 & 9.94027059e-01 & 3.14371129e+01 \end{bmatrix}$$

AFTER LM:

Intrinsic Camera Parameters

$$K = \begin{bmatrix} 498.16759372 & 0.71948151 & 216.77236795 \\ 0. & 497.59204734 & 280.67282212 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters

Image 4:

$$[R|t] = \begin{bmatrix} 0.99398037 & -0.01749572 & 0.10815232 & -10.27944023 \\ 0.01569002 & 0.99972332 & 0.01752446 & -9.62916775 \\ -0.10842899 & -0.01572206 & 0.99397986 & 31.58296457 \end{bmatrix}$$

Image 9:

$$[R|t] = \begin{bmatrix} 0.97186542 & -0.02587485 & -0.23411128 & -5.10238879 \\ .04462265 & 0.996174 & 0.07514102 & -8.77540711 \\ 0.2312713 & -0.08347362 & 0.96930168 & 32.26108243 \end{bmatrix}$$

Image 16:

$$[R|t] = \begin{bmatrix} 0.97482213 & -0.080744 & 0.20785143 & -8.80626252 \\ 0.00235537 & 0.93581318 & 0.3524885 & -9.20521445 \\ -0.22297144 & -0.34312402 & 0.91244158 & 39.37597458 \end{bmatrix}$$

Image 25:

$$[R|t] = \begin{bmatrix} 0.99398037 & -0.01749572 & 0.10815232 & -10.27944023 \\ 0.01569002 & 0.99972332 & 0.01752446 & -9.62916775 \\ -0.108429 & -0.01572206 & 0.99397986 & 31.58296457 \end{bmatrix}$$

LM with Radial Distortion Incorporated

Radial Distortion Parameters://

k1=4.5457e-07

k2=-5.0363e-12

Some small improvements can be seen when radial distortion is incorporated.

	Mean before LM	Mean after Lm(RD)	Variance before LM	Variance after LM(RD)
Image 3	5.9954	1.3179	10.0602	0.7177
Image 8	8.2099	1.3278	17.2305	0.7975
Image 15	6.8268	1.2399	21.5458	0.6803
Image 22	10.2113	1.2538	44.8102	0.5822

Intrinsic Camera Parameters with LM with Radial Distortion Incorporated

$$K = \begin{bmatrix} 502.4296083 & 0.61132347 & 278.62494628 \\ 0. & 501.46835879 & 216.71339826 \\ 0. & 0. & 1. \end{bmatrix}$$

Extrinsic Camera Parameters with LM with Radial Distortion Incorporated

Image 9:

$$[R|t] = \begin{bmatrix} 0.97120703 & -0.02583156 & -0.23683249 & -5.16984462 \\ 0.04345778 & 0.99663433 & 0.06950855 & -8.9883172 \\ 0.23423987 & -0.0777994 & 0.96906085 & 31.9862055 \end{bmatrix}$$

Measured Ground Truth wrt Fixed Image

The [R|t] matrix for fixed image 1 is shown below. It is seen that R approximately corresponds to I_{3X3} identity matrix and the t vector gives displacement and depth. This verifies the accuracy of the implementation.

$$[R|t] = \begin{bmatrix} 0.99950066 - 0.002188850.03152216 & -9.73576769 \\ 0.00102430.999317960.03691282 & -9.69792586 \\ -0.03158146 - 0.03686210.9988212 & 31.82310695 \end{bmatrix}$$

Source Code

The Source Code has also been attached for clarity.

```
import math
import numpy as np
import cv2 as cv
from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from scipy.optimize import least_squares
```

```
no_images=40
# Function to mark the original and reprojected points
# and determine the accuracy of the camera projection matrix
# Also computes the mean and variance of the errors between the points.
def mark_points(intersection,Proj):
    path="C:/Users/Amruthavarshini/Desktop/ECE-66100/HW8/Dataset1/Pic_11.jpg"
    img=(Image.open(path))
    draw = ImageDraw.Draw(img)
    dist=list()
    for j in range(len(intersection[10])):
        draw.text((intersection[10][j][0][0], intersection[10][j][0][1]),"*",(255,0
        draw.text((list(Proj[j])[0],list(Proj[j])[1]),"O",(0,255,0))
        #draw.text((intersection[10][j][0][0], intersection[10][j][0][1]),str(j+1),
        dist.append(np.linalg.norm(np.asarray(intersection[10][j][0])-Proj[j]))
    img.save('reproject_38onto11.jpg')
    return np.mean(dist),np.var(dist)
# Function to reproject points based on original and refined
# camera projection matrices.
def reproject_points(H_temp,intersection,k):
    # Reprojecting k=4,18,26,38 onto 11
    H=np.matmul(H_temp[10],np.linalg.pinv(H_temp[k-1]))
    Proj=list()
    for i in range(len(intersection[10])):
        point=list(np.asarray(intersection[k-1][i][0]).copy())
        point.append(1.0)
        proj_point=np.matmul(H,np.asarray(point))
        proj_point=proj_point/proj_point[2]
        Proj.append(proj_point[:-1])
    return Proj
# Function to find H_new(camera projection matrix) from the
# lm modified r. Reconstructing RT and K from r to find H_new
def find_Hnew(r):
    H=list()
    for i in range(no_images):
        K=np.zeros((3,3))
```

```
K[0][0]=r.x[0]
K[0][1]=r.x[1]
K[0][2]=r.x[2]
K[1][1]=r.x[3]
K[1][2]=r.x[4]
K[2][2]=1.0
wx=r.x[5+6*i]
wy=r.x[5+1+6*i]
wz=r.x[5+2+6*i]
W=np.zeros((3,1))
W[0]=wx
W[1]=wy
W[2]=wz
phi=np.linalg.norm(W)
Wx=np.zeros((3,3))
Wx[0][1]=-1*wz
Wx[0][2]=wy
Wx[1][0]=wz
Wx[1][2]=-1*wx
Wx[2][0]=-1*wy
Wx[2][1]=wx
R=np.zeros((3,3))
R_first=np.zeros((3,3))
R_first[0][0]=R_first[1][1]=R_first[2][2]=1.0
R_second=(np.sin(phi)/phi)*Wx
R_third=((1-np.cos(phi))/(phi*phi))*np.matmul(Wx,Wx)
R=R_first+R_second+R_third
T=np.zeros((3))
T[0]=r.x[5+3+6*i]
T[1]=r.x[5+4+6*i]
T[2]=r.x[5+5+6*i]
RT=np.zeros((3,3))
RT[:,0]=R[:,0]
RT[:,1]=R[:,1]
RT[:,2]=T
```

```
        if (i==10):
            print K
            print R
            print T
        H.append(np.matmul(K,RT))
    return H

# Function to refine the camera matrix
def cost(p):
    global image_points
    ss=list()
    for img in range(no_images):
        # Building the K, R , t matrices for each image
        wx=p[6*img+5]
        wy=p[6*img+1+5]
        wz=p[6*img+2+5]
        t1=p[6*img+3+5]
        t2=p[6*img+4+5]
        t3=p[6*img+5+5]
        alpha_x=p[0]
        s=p[1]
        x_0=p[2]
        alpha_y=p[3]
        y_0=p[4]
        W=np.zeros((3,1))
        W[0]=wx
        W[1]=wy
        W[2]=wz
        phi=np.linalg.norm(W)
        Wx=np.zeros((3,3))
        Wx[0][0]=0.0
        Wx[0][1]=-1*wz
        Wx[0][2]=wy
        Wx[1][0]=wz
        Wx[1][1]=0.0
```

```
Wx[1][2]=-1*wx
Wx[2][0]=-1*wy
Wx[2][1]=wx
Wx[2][2]=0.0
R_first=np.zeros((3,3))
R_first[0][0]=R_first[1][1]=R_first[2][2]=1.0
R_second=(np.sin(phi)/phi)*Wx
R_third=((1-np.cos(phi))/(phi*phi))*np.matmul(Wx,Wx)
R=R_first+R_second+R_third
T=np.zeros((3))
T[0]=t1
T[1]=t2
T[2]=t3
K=np.zeros((3,3))
K[0][0]=alpha_x
K[0][1]=s
K[0][2]=x_0
K[1][1]=alpha_y
K[1][2]=y_0
K[2][2]=1.0
RT=np.zeros((3,3))
RT[:,0]=R[:,0]
RT[:,1]=R[:,1]
RT[:,2]=T
# Radial distortion parameters

# Adding the values for each of the points in this image
#print img
for i in range(len(image_points[img])):
    actual_point=np.array(np.asarray(image_points[img][i][0]).copy()).tolis
    actual_point.append(1.0) # making it homogeneous 3-vector
    estimate_point=list()
    estimate_x=(i/10)*2.5
    estimate_y=(i%10)*2.5
    estimate_point.append(estimate_x)
```

```
            estimate_point.append(estimate_y)
            estimate_point.append(1.0)
            KRT=np.matmul(K,RT)
           # print(np.shape(np.asarray(actual_point)))
            #print(np.shape(np.matmul(KRT,np.asarray(estimate_point))))
            temp=np.matmul(KRT,np.asarray(estimate_point))
            temp=temp/temp[2]
            # Radial distortion
#           gamma=np.sqrt(np.square(temp[0]-x_0)+np.square(temp[1]-y_0))
#           new_temp=np.zeros((3))
#           new_temp[2]=1.0
#           new_temp[0]=temp[0]+(temp[0]-x_0)*(p[-2]*np.square(gamma)+p[-1]*np.squ
#           new_temp[1]=temp[1]+(temp[1]-y_0)*(p[-2]*np.square(gamma)+p[-1]*np.squ
#           final_point=(np.asarray(actual_point)-new_temp)
            final_point=(np.asarray(actual_point)-temp)
            ss.append(final_point[0])
            ss.append(final_point[1])
            ss.append(final_point[2])
    return ss


# Function to compute the extrinsic parameters of the camera
def compute_extrinsic(K,homographies):
    R=list()
    for img in range(len(homographies)):
        e=1/np.linalg.norm(np.matmul(np.linalg.pinv(K),homographies[img][:,0]))
        r1=e*np.matmul(np.linalg.pinv(K),homographies[img][:,0])
        r2=e*np.matmul(np.linalg.pinv(K),homographies[img][:,1])
        r3=np.cross(r1,r2)

        # Conditioning the rotational matrix
        Q=np.zeros((3,3))
        Q[:,0]=r1
        Q[:,1]=r2
        Q[:,2]=r3
```

```
        # Computing SVD of V
        u, d, vt = np.linalg.svd(Q)
        Z=np.matmul(u,vt)
        r1=Z[:,0]
        r2=Z[:,1]
        r3=Z[:,2]

        t=e*np.matmul(np.linalg.pinv(K),homographies[img][:,2])
        rt=np.zeros((3,4))
        rt[:,0]=r1
        rt[:,1]=r2
        rt[:,2]=r3
        rt[:,3]=t
        R.append(rt)
    return R
# Function to compute the intrinsic matrix K from the W matrix
def compute_K(w):
    x_0=((w[0][1]*w[0][2])-(w[0][0]*w[1][2]))/((w[0][0]*w[1][1])-(w[0][1]*w[0][1]))
    lamb=w[2][2]-(((w[0][2]*w[0][2])+x_0*((w[0][1]*w[0][2])-(w[0][0]*w[1][2])))/w[0
    alpha_x=abs(np.sqrt(lamb/w[0][0]))
    alpha_y=abs(np.sqrt((lamb*w[0][0])/((w[0][0]*w[1][1])-(w[0][1]*w[0][1]))))
    s=-1*((w[0][1]*alpha_x*alpha_x*alpha_y)/(lamb))
    y_0=((s*x_0)/alpha_y)-((w[0][2]*alpha_x*alpha_x)/lamb)
    K=np.zeros((3,3))
    K[0][0]=alpha_x
    K[0][1]=s
    K[0][2]=x_0
    K[1][0]=0.0
    K[1][1]=alpha_y
    K[1][2]=y_0
    K[2][0]=0.0
    K[2][1]=0.0
    K[2][2]=1.0
    return K,alpha_x,alpha_y,s,x_0,y_0
# Function to calculate vector v given H,i,j
```

```
def compute_V(Ht,i,j):
    H=np.transpose(Ht)
    V=np.zeros((1,6))
    V[0][0]=H[i][0]*H[j][0]
    V[0][1]=(H[i][0]*H[j][1])+(H[i][1]*H[j][0])
    V[0][2]=H[i][1]*H[j][1]
    V[0][3]=(H[i][2]*H[j][0])+(H[i][0]*H[j][2])
    V[0][4]=(H[i][2]*H[j][1])+(H[i][1]*H[j][2])
    V[0][5]=H[i][2]*H[j][2]
    return V
# Function to calculate the transpose of the Vij vectors(V1 and V2)
def compute_Vij(H):
    # computing V1=transpose(v12)
    i=0
    j=1
    V1=compute_V(H,i,j)
    # computing V2=transpose(V11-V22)
    i=0
    j=0
    V11=compute_V(H,i,j)
    i=1
    j=1
    V22=compute_V(H,i,j)
    V2=V11-V22
    return V1[0],V2[0]

# Function to compute the 'w' matrix using all the homographies corresponding
# to 40 images taken with different positions of the camera
def compute_omega(homographies):
    V=np.zeros((2*len(homographies),6))
    for img in range(len(homographies)):
        V1,V2=compute_Vij(homographies[img])
        V[2*img]=V1
        V[2*img+1]=V2
    # Computing SVD of V
```

```
    u, d, vt = np.linalg.svd(V)
    # Choosing the last column (smallest eigen value vector )
    # of vt as solution b
    b=np.transpose(vt)[:,-1]
    # Constructing omega(w) matrix knowing b
    w=np.zeros((3,3))
    w[0][0]=b[0]
    w[0][1]=b[1]
    w[0][2]=b[3]
    w[1][0]=b[1]
    w[1][1]=b[2]
    w[1][2]=b[4]
    w[2][0]=b[3]
    w[2][1]=b[4]
    w[2][2]=b[5]
    return w,V
# Function to compute the homographies for each image to the
# evaluated points of intersection from the respective world coordinates
# using linear least squares
def compute_homography(intersection):
    homographies=list()
    for img in range(len(intersection)):
        A=np.zeros((2*len(intersection[img]),9))
        for i in range(len(intersection[img])):
            world_x=(i/10)*2.5
            world_y=(i%10)*2.5
            image_x=intersection[img][i][0][0]
            image_y=intersection[img][i][0][1]
            A[2*i+0][0]=world_x
            A[2*i+0][1]=world_y
            A[2*i+0][2]=1.0
            A[2*i+0][3]=0.0
            A[2*i+0][4]=0.0
            A[2*i+0][5]=0.0
            A[2*i+0][6]=-1*world_x*image_x
```

```
            A[2*i+0][7]=-1*world_y*image_x
            A[2*i+0][8]=-1*image_x
            A[2*i+1][0]=0.0
            A[2*i+1][1]=0.0
            A[2*i+1][2]=0.0
            A[2*i+1][3]=world_x
            A[2*i+1][4]=world_y
            A[2*i+1][5]=1.0
            A[2*i+1][6]=-1*world_x*image_y
            A[2*i+1][7]=-1*world_y*image_y
            A[2*i+1][8]=-1*image_y
        # Computing SVD of A
        u, d, vt = np.linalg.svd(A)
        # Choosing the last column (smallest eigen value vector )
        # of vt as solution
        H_col=np.transpose(vt)[:,-1]
        # Converting the H to a normalized 3X3 matrix
        H_col=H_col/H_col[8]
        H=np.zeros((3,3))
        H[0][0]=H_col[0]
        H[0][1]=H_col[1]
        H[0][2]=H_col[2]
        H[1][0]=H_col[3]
        H[1][1]=H_col[4]
        H[1][2]=H_col[5]
        H[2][0]=H_col[6]
        H[2][1]=H_col[7]
        H[2][2]=1.0
        homographies.append(H)
    return homographies
# Function to label the intersection points in each image
def number_points(intersection):
    global no_images
    Images=list()
    for i in range(no_images):
```

```
        path="C:/Users/Amruthavarshini/Desktop/ECE-66100/HW8/Dataset1/Pic_"+str(i+1
        img=(Image.open(path))
        draw = ImageDraw.Draw(img)
        for j in range(len(intersection[i])):
            draw.text((intersection[i][j][0][0], intersection[i][j][0][1]),str(j+1)
        img.save('numbered_img'+str(i+1)+'.jpg')
# Function to find intersection of all the horizontal and vertical lines in each im
def find_intersection(hor, ver):
    global no_images
    """Finds the intersection of two lines given in Hesse normal form.
    Returns closest integer pixel locations.
    Reference : https://stackoverflow.com/a/383527/5087436
    """
    final_intersections=list()
    for img in range(no_images):
        intersections=list()
        for i in range(len(ver[img])):
            for j in range(len(hor[img])):
                #print img,i,j
                rho1, theta1 = ver[img][i][0]
                rho2, theta2 = hor[img][j][0]
                A = np.array([
                    [np.cos(theta1), np.sin(theta1)],
                    [np.cos(theta2), np.sin(theta2)]
                ])
                b = np.array([[rho1], [rho2]])
                x0, y0 = np.linalg.solve(A, b)
                x0, y0 = int(np.round(x0)), int(np.round(y0))
                intersections.append([[x0, y0]])
        final_intersections.append(intersections)
    return final_intersections
# Function to choose (10X8) lines for each image
def choose_lines(houghLines):
    finalHough=list()
    for i in range(len(houghLines)):
```

```
# For each image
# Dividing into two lists - vertical and horizontal lines
horizontal=list()
vertical=list()
for j in houghLines[i]:
    theta=j[0][1]
    if np.pi/4<theta<(np.pi*3)/4:
        horizontal.append(j)
    else:
        vertical.append(j)
# Choosing 8 lines in vertical and 10 lines in horizontal set
# to represent final lines
final_horizontal=list()
threshold=100
# Tuning threshold so as to find 10 best horizontal lines
while(len(final_horizontal)<10):
    threshold=threshold-0.05
    final_horizontal=list()
    for x in range(len(horizontal)):
        line=horizontal[x][0][0]
        discard=0
        for k in final_horizontal:
            if abs(abs(k[0][0])-abs(line))<threshold:
                discard=1
        if discard==0:
            final_horizontal.append(horizontal[x])
#print (len(final_horizontal))

# Tuning threshold so as to find 8 best vertical lines
final_vertical=list()
threshold=100
while(len(final_vertical)<8):
    threshold=threshold-0.05
    final_vertical=list()
    for x in range(len(vertical)):
```

```
                line=vertical[x][0][0]
                discard=0
                for k in final_vertical:
                    if abs(abs(k[0][0])-abs(line))<threshold:
                        discard=1
                if discard==0:
                    final_vertical.append(vertical[x])
        #print (len(final_vertical))
        # Adding selected 8 vertical and 10 horizontal lines to final list
        finalHough.append(final_horizontal+final_vertical)
    return finalHough
# Function to draw the found lines
def draw_lines(Images,houghLines):
    for i in range(len(houghLines)):
        for j in houghLines[i]:
            rho=j[0][0]
            theta=j[0][1]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a*rho
            y0 = b*rho
            x1 = int(x0 + 1000*(-b))
            y1 = int(y0 + 1000*(a))
            x2 = int(x0 - 1000*(-b))
            y2 = int(y0 - 1000*(a))
            cv.line(Images[i],(x1,y1),(x2,y2),(0,0,255),2)
        cv.imwrite('houghlinesxxx'+str(i)+'.jpg',Images[i])
# Function to find edge lines using Canny edge detector and Hough transform
def find_lines(Images):
    houghLines=list()
    for i in range(len(Images)):
        edge=cv.Canny(Images[i],255*1.5,255)
        cv.imwrite('houghedges'+str(i)+'.jpg',edge)
        lines=cv.HoughLines(edge,1,np.pi/180,50)
        houghLines.append(lines)
```

```
    return houghLines
# Reading all the images from dataset 1
# Storing them in a list "Images"
def read_images():
    global no_images
    Images=list()
    for i in range(no_images):
        path="C:/Users/Amruthavarshini/Desktop/ECE-66100/HW8/Dataset1/Pic_"+str(i+1
        img=np.asarray(Image.open(path))
        img_g=cv.cvtColor(img,cv.COLOR_BGR2GRAY)
        Images.append(img_g)
    return Images
Images=read_images()
houghLines=find_lines(Images)
Images1=np.asarray(Images).copy()
#draw_lines(Images1,houghLines)
houghFinal=choose_lines(houghLines)
draw_lines(Images1,houghFinal)
# Sorting the lines based on the first component
# However keeping the horizontal and vertical lines separately
hf=np.asarray(houghFinal).copy()
finalHorizontal=list()
finalVertical=list()
for i in range(len(hf)):
    hff=np.array(hf[i]).tolist()
    hff_horizontal=hff[0:10]
    hff_vertical=hff[10:]
    hff_horizontal.sort(key=lambda x: x[0][0])
    #hf_vertical=abs(np.array(np.asarray(hff_vertical))).tolist()
    hff_vertical.sort(key=lambda x: abs(x[0][0]))
    finalHorizontal.append(hff_horizontal)
    finalVertical.append(hff_vertical)
# Finding the points of intersection
intersection=find_intersection(finalHorizontal,finalVertical)
global image_points
```

```
image_points=np.array(np.asarray(intersection).copy()).tolist()
number_points(intersection)
homographies=compute_homography(intersection)
w,V=compute_omega(homographies)
K,alpha_x,alpha_y,s,x_0,y_0=compute_K(w)
RT=compute_extrinsic(K,homographies)
# Computing camera matrix H=K[R|T] before LM refinement
H_old=list()
RT=np.asarray(RT)
for i in range(no_images):
    H_old.append(np.matmul(K,RT[i][:,[0,1,3]]))
# non linear estimation for refinement
RT=list(RT)
p=list()
p.append(alpha_x)
p.append(s)
p.append(x_0)
p.append(alpha_y)
p.append(y_0)
for i in range(len(homographies)):
    trace=(np.trace(RT[i][:,0:3])-1)/2
    if trace>1.0:
        trace=1.0
    phi=np.arccos(trace)
    if phi==0:
        phi=1
    p.append((RT[i][2][1]-RT[i][1][2])*(phi/(2*np.sin(phi))))
    p.append((RT[i][0][2]-RT[i][2][0])*(phi/(2*np.sin(phi))))
    p.append((RT[i][1][0]-RT[i][0][1])*(phi/(2*np.sin(phi))))
    p.append(RT[i][0][3])
    p.append(RT[i][1][3])
    p.append(RT[i][2][3])
# Radial distortion k1=0 and k2=0
#p.append(0)
#p.append(0)
```

```
r=least_squares(cost, p, method='lm',max_nfev=500)
K_lm=np.zeros((3,3))
K_lm[0][0]=r.x[0]
K_lm[0][1]=r.x[1]
K_lm[0][2]=r.x[2]
K_lm[1][1]=r.x[3]
K_lm[1][2]=r.x[4]
K_lm[2][2]=1.0
H_new=find_Hnew(r)

# Reprojecting points
Proj_old=reproject_points(H_old,intersection,18)
mean18old,var18old=mark_points(intersection,Proj_old)
Proj_new=reproject_points(H_new,intersection,18)
mean18new,var18new=mark_points(intersection,Proj_new)

Proj_old=reproject_points(H_old,intersection,26)
mean26old,var26old=mark_points(intersection,Proj_old)
Proj_new=reproject_points(H_new,intersection,26)
mean26new,var26new=mark_points(intersection,Proj_new)

Proj_old=reproject_points(H_old,intersection,4)
mean4old,var4old=mark_points(intersection,Proj_old)
Proj_new=reproject_points(H_new,intersection,4)
mean4new,var4new=mark_points(intersection,Proj_new)

Proj_old=reproject_points(H_old,intersection,38)
mean38old,var38old=mark_points(intersection,Proj_old)
Proj_new=reproject_points(H_new,intersection,38)
mean38new,var38new=mark_points(intersection,Proj_new)
```