School of Electrical and Computer Engineering
Purdue University, WL, IN, USA

Nahian Ibn Hasan
Email: hasan34@purdue.edu
PUID: 0032764564
ECE66100 - Computer Vission
Fall 2022
Homework-8

November 16, 2022

1 Objective

The goal in this homework is to implement the popular Zhang's algorithm for camera calibration. A formal description of the algorithm can be found in the Zhang's technical report [1].

2 Theoretical Questions

In Lecture 20, 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 20 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.

Answer

No, we cannot see the image of an absolute coniic in the image planesince the pixels are imaginary. This image of absolute conic is rotationally and translationally invariant. This metric helps us to identify the intrinsic parameters of the camera.

3 Programming Implementation Details

- For the camera calibration task, we have chosen a checkerboard pattern with 4 blocks along the horizontal directiona nd 5 blocks along the vertical direction.
- First, the edges of the images are identified using the Canny Edge Detector (implemented by OpenCV source code). The image is first converted to gray scale. The high and low threshold of the images are both set to 400.
- OpenCV hough transformation implementation is applied to find the lines from the edges of the canny edge detector. To extract a reasonable amount of lines, the default parameters in the implementation source code is utilized.
- Next, using homographic transformation, the intersection of lines are calculated using the cross product of a pair of lines. The number of intersections is very high and clustered around each corner point. The clustered corners around an original corner point are averaged. Some false corner points at the background of the pattern are discarded by gray level thresholding at those locations. Further, some unnecesary corner points are discarded which are beyond a certain radius from the black checkerboard patterns. In this way, there are 80 points successfully extracted.
- next, the corners are sorted from left to right and top to bottom, so that each corner point are labelled uniquely and in similar fashion in every viewpoint. Sorting and labelling algorithm is implemented from the paper of [2].

4 Results

4.1 Detected Edges and Hough Transformation Lines in Images of Dataset 1

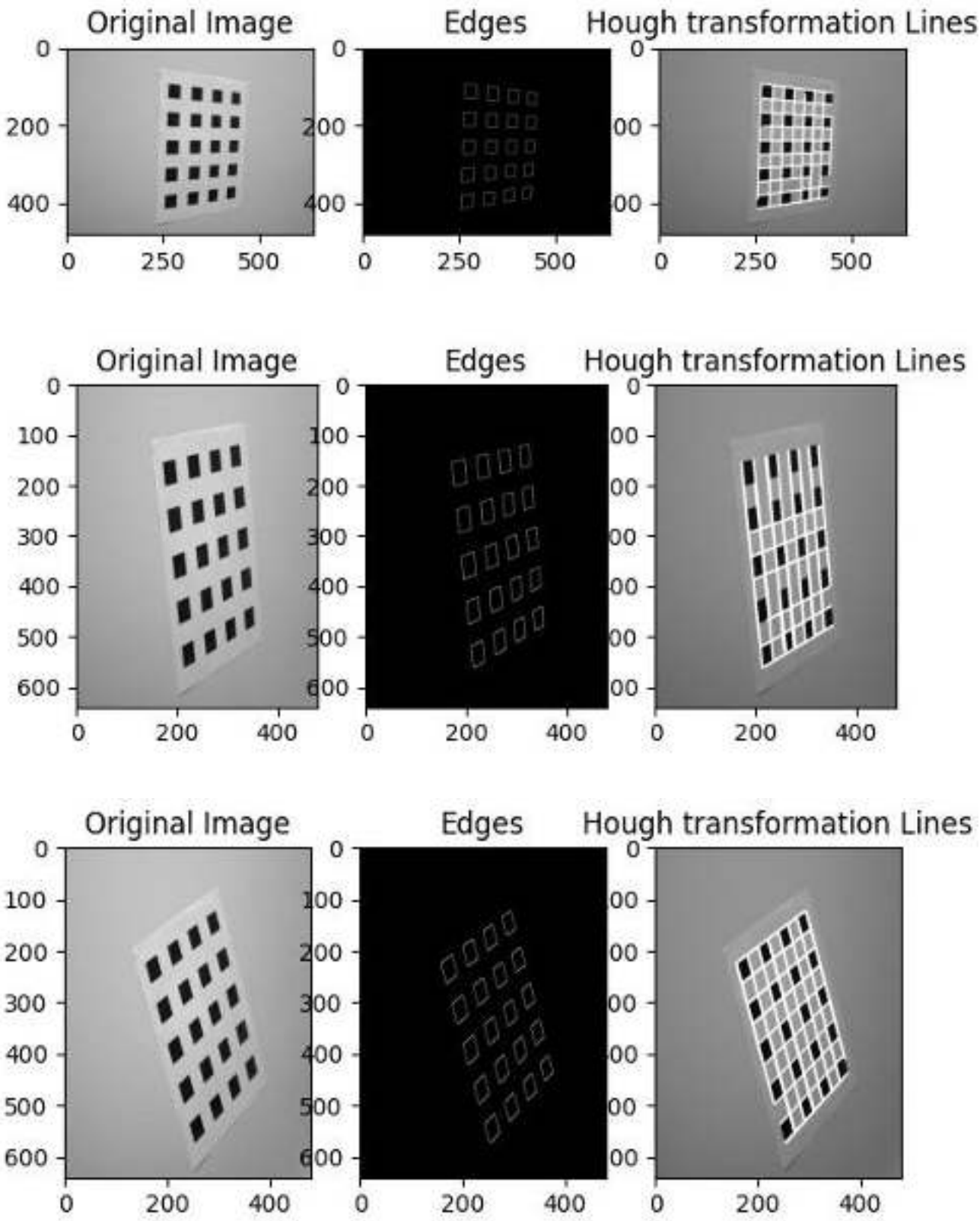

Figure 1: Sample Detected Edges (Canny Edge Detector) and Hough Transformation Lines in Images of Dataset 1

4.2 Detected Edges and Hough Transformation Lines in Images of Dataset 2

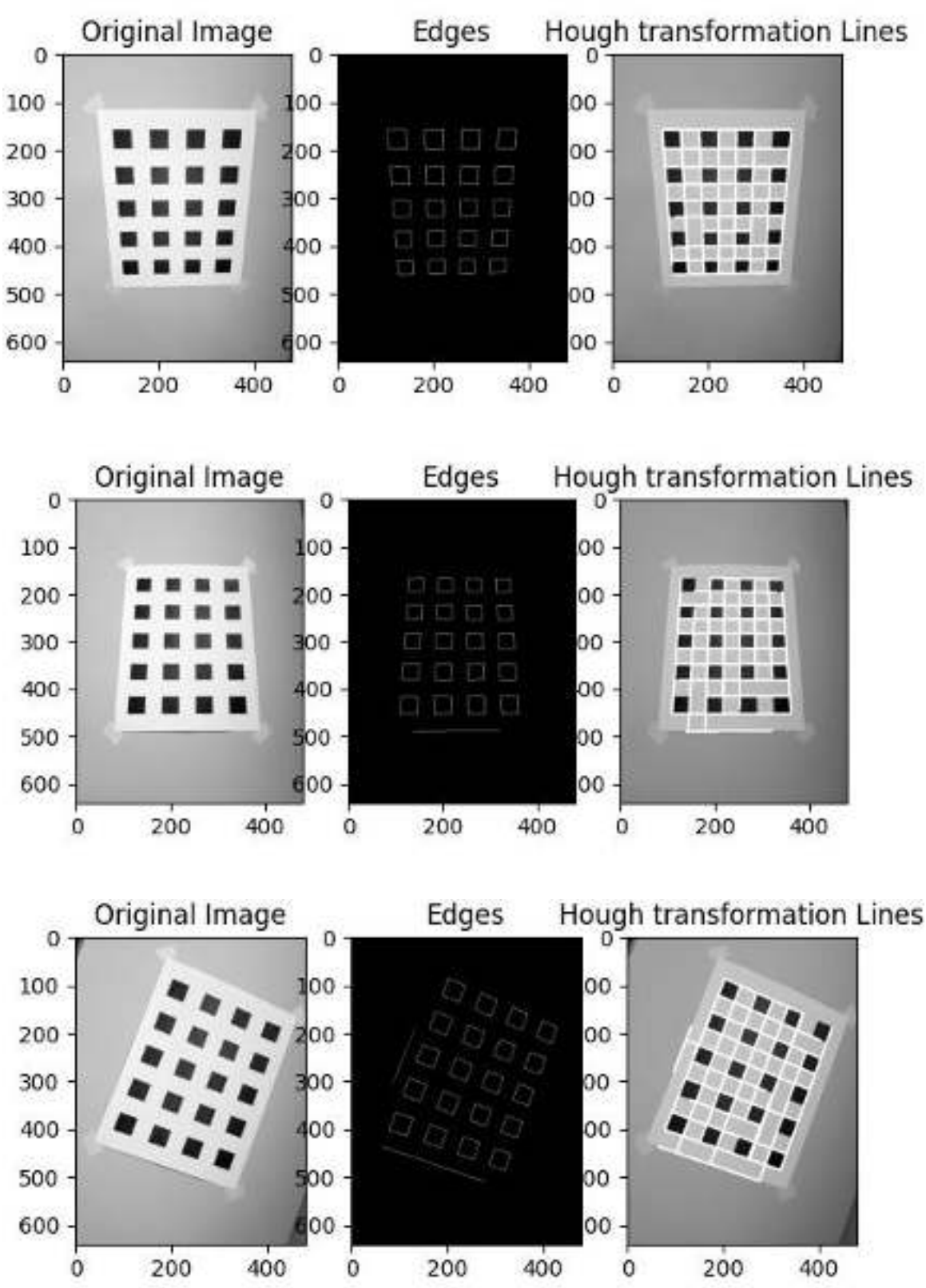

Figure 2: Sample Detected Edges (Canny Edge Detector) and Hough Transformation Lines in Images of Dataset 2

4.3 Detected and Labelled Corners on Images of Database 1

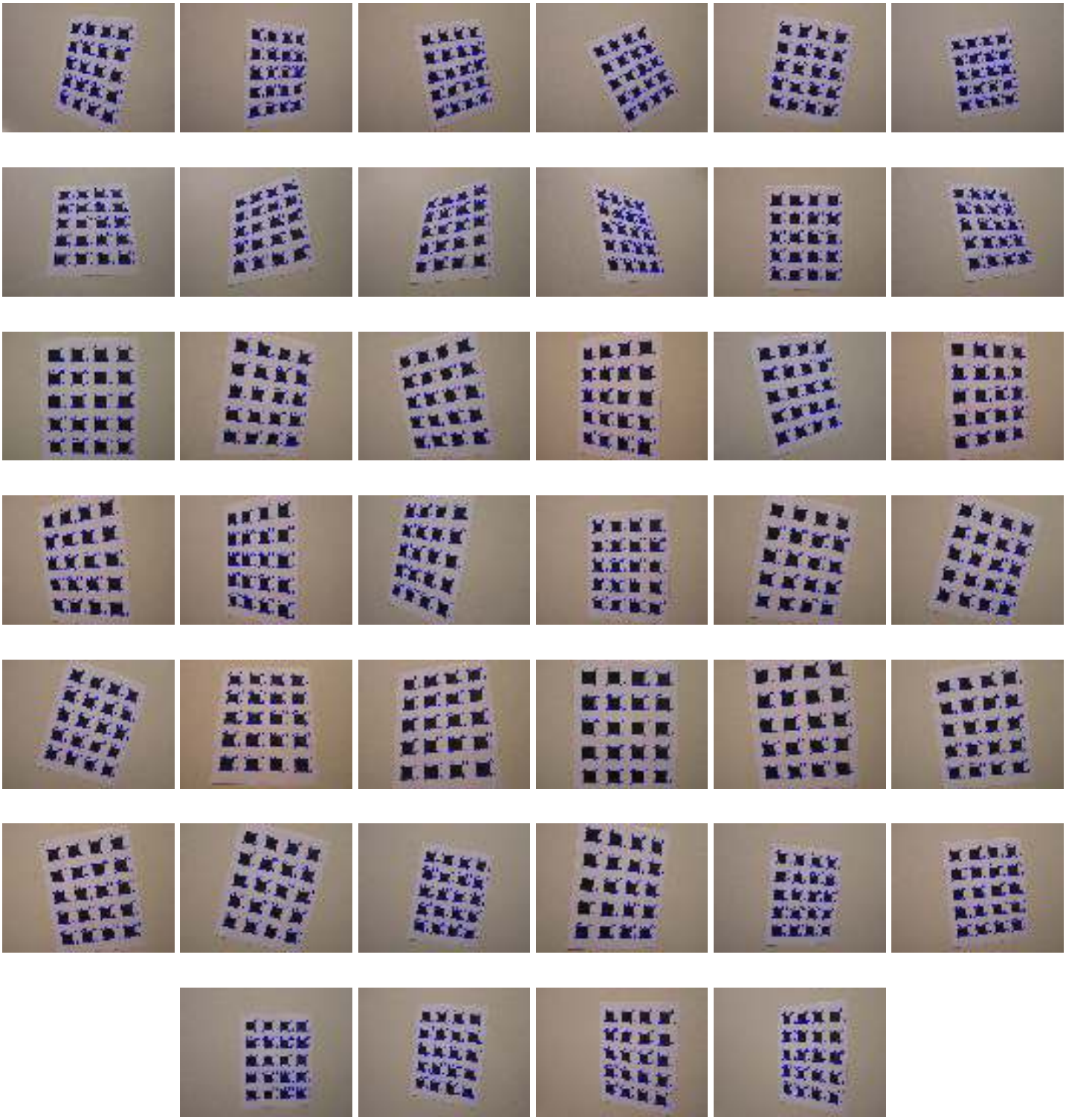

Figure 3: Assigned Corners on every image of database 1

4.4 Detected and Labelled Corners on Images of Database 2

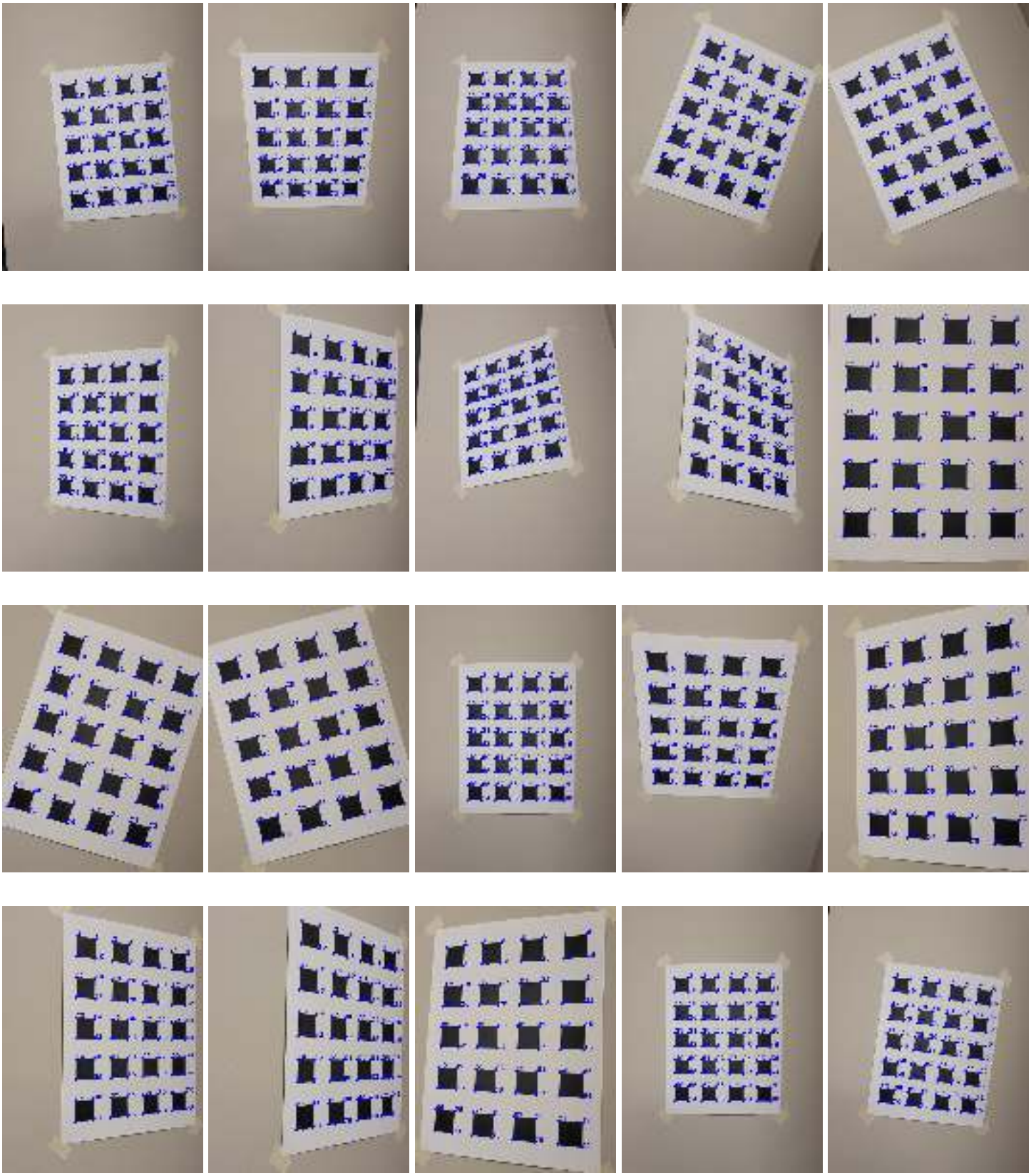

Figure 4: Assigned Corners on every image of database 2 (Custom Dataset)

4.5 Zhang's Algorithm

4.5.1 Theoretical Overview

The Zhang's algorithm starts by extracting intrinsic parameters, followed by extracting extrinsic parameters and the fixing the radial distortion. In our implementation, we apply the Levenberg-Marquadt (LM) optimization technique to further calibrate the parameters.

Extracting Intrinsic Parameters

Let's assume, the pattern lays on the plane $z = 0$. Any point on this pattern is imaged by the following equation-

$$y = Hx = K[R|t] \begin{bmatrix} x \\ y \\ 0 \\ w \end{bmatrix}, \tag{1}$$

here, K = intrinsic camera parameter matrix, R is the rotation matrix and t is the camera translation vector. The image of two circular points in the image of an absolute conic follows the following two criteria-

$$h_1^T w h_1 - h_2^T w h_2 = 0 \tag{2}$$

$$h_1^T w h_2 = 0, \tag{3}$$

here, $H = [h_1, h_2, h_3]$. These two sets of equations can be rearranged to the following format-

$$Vb = \begin{bmatrix} v_{12}^T \\ v_{11}^T - v_{22}^T \end{bmatrix} b = 0 \tag{4}$$

here,

$$v_{ij} = \begin{bmatrix} h_{i1}hj1 \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \quad b = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{22} \\ w_{13} \\ w_{23} \\ w_{33} \end{bmatrix} \tag{5}$$

For each H we get between the ground truth pattern and the corner points pattern (that we calculated), there are two sets of equations for the above system. hence, we need at least three images to solve the equation. However, for bettwe estimation, we have used 40 images in dataset 1 and 20 images in dataset 2.
Next, the intrinsic parameter matrix K is determined from

$$w = K^{-T} K^{-1} \tag{6}$$

The K matrix is determined by-

$$\begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

According to Zhang's paper, these parameters are determined as-

$$y_0 = \frac{w_{12}w_{13} - w_{11}w_{23}}{w_{11}w_{22} - w_{12}^2} \tag{8}$$

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

$$\alpha_x = \sqrt{\frac{\lambda}{w_{11}}} \tag{10}$$

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

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

$$x_0 = -\frac{s y_0}{\alpha_y} - \frac{w_{13}\alpha_x^2}{\lambda} \tag{13}$$

To determine the extrinsic parameters ($R = [r_1, r_2, r_3], t$), we use the following formula-

$$\begin{bmatrix} r_1 & r_2 & t \end{bmatrix} = \psi K^{-1} \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \tag{14}$$

$$r_3 = \psi r_1 \times r_2 \tag{15}$$

here, $\psi = ||K^{-1}h_1||^{-1}$. The matrix R has another representation -

$$w = \frac{\phi}{2sin\phi} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{16}$$

$$\phi = cos^{-1} \frac{trace(R) - 1}{2} \tag{17}$$

To transform from the LM refinement form back to R, we utilize the following sets of equations-

$$W = \begin{bmatrix} 0 & -w_2 & w_1 \\ w_2 & 0 & -w_0 \\ -w_1 & w_0 & 0 \end{bmatrix} \tag{18}$$

$$R = e^W = I_{3\times 3} + \frac{sin\phi}{\phi} W + \frac{1 - cos\phi}{\phi^2} W^2 \tag{19}$$

$$\phi = ||w|| \tag{20}$$

Once, the initial set of parameters are determined, we calibrate those parameters using the LM refinement technique based on the following loss finction-

$$d^2_{geom} = ||X - f(p)||^2 = \sum_i \sum_j ||x_{i,j} - \hat{x_{i,j}}|| \tag{21}$$

$$\hat{x_{i,j}} = K[R_i|t]x_{m,j} \tag{22}$$

Here, all the R_i, t_i and K are considered to be parameters of the LM method.
For fixing radial distortion, we have the following sets of equations-

$$\hat{x_{rad}} = \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4] \tag{23}$$

$$\hat{y_{rad}} = \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4] \tag{24}$$

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

Here, $\hat{x}, \hat{y}$ are the projected pattern without radial distortion, and (x_0, y_0) are the curerntly available principal point. The radial distortion parameters for the database 1 are-

$$k_1 = -6.5489 \times 10^{-12} \tag{26}$$

$$k_2 = 7.292 \times 10^{-22} \tag{27}$$

and for database 2 are -

$$k_1 = 1.5207 \times 10^{-12} \tag{28}$$

$$k_2 = -1.06298 \times 10^{-22} \tag{29}$$

4.6 Zhang Algorithm's Results

4.6.1 Reprojected world corners to each viewpoint before LM refinement (Database 1)

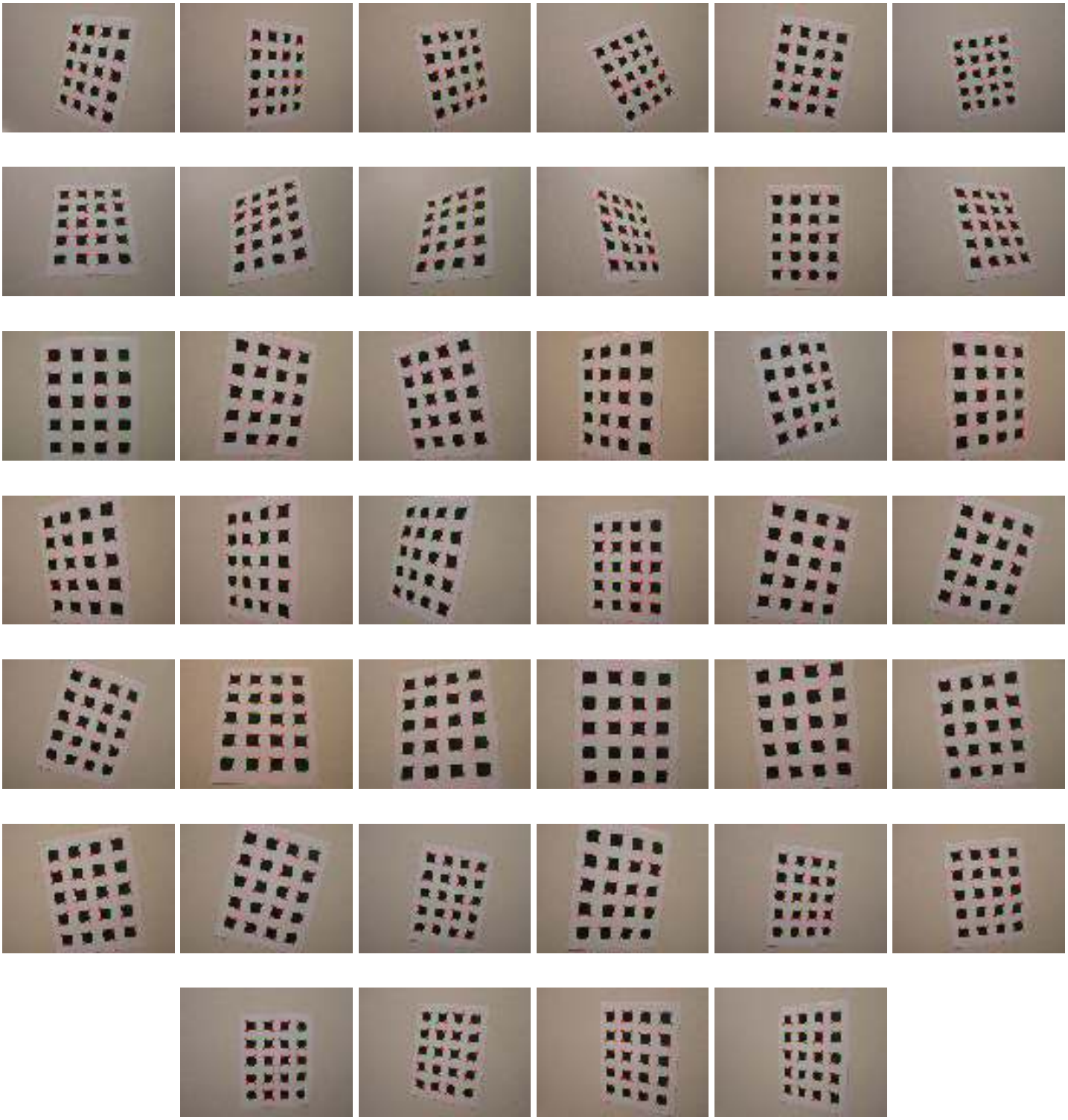

Figure 5: Reprojected world corners to each viewpoint before LM refinement (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.2 Reprojected world corners to each viewpoint before LM refinement (Database 2)

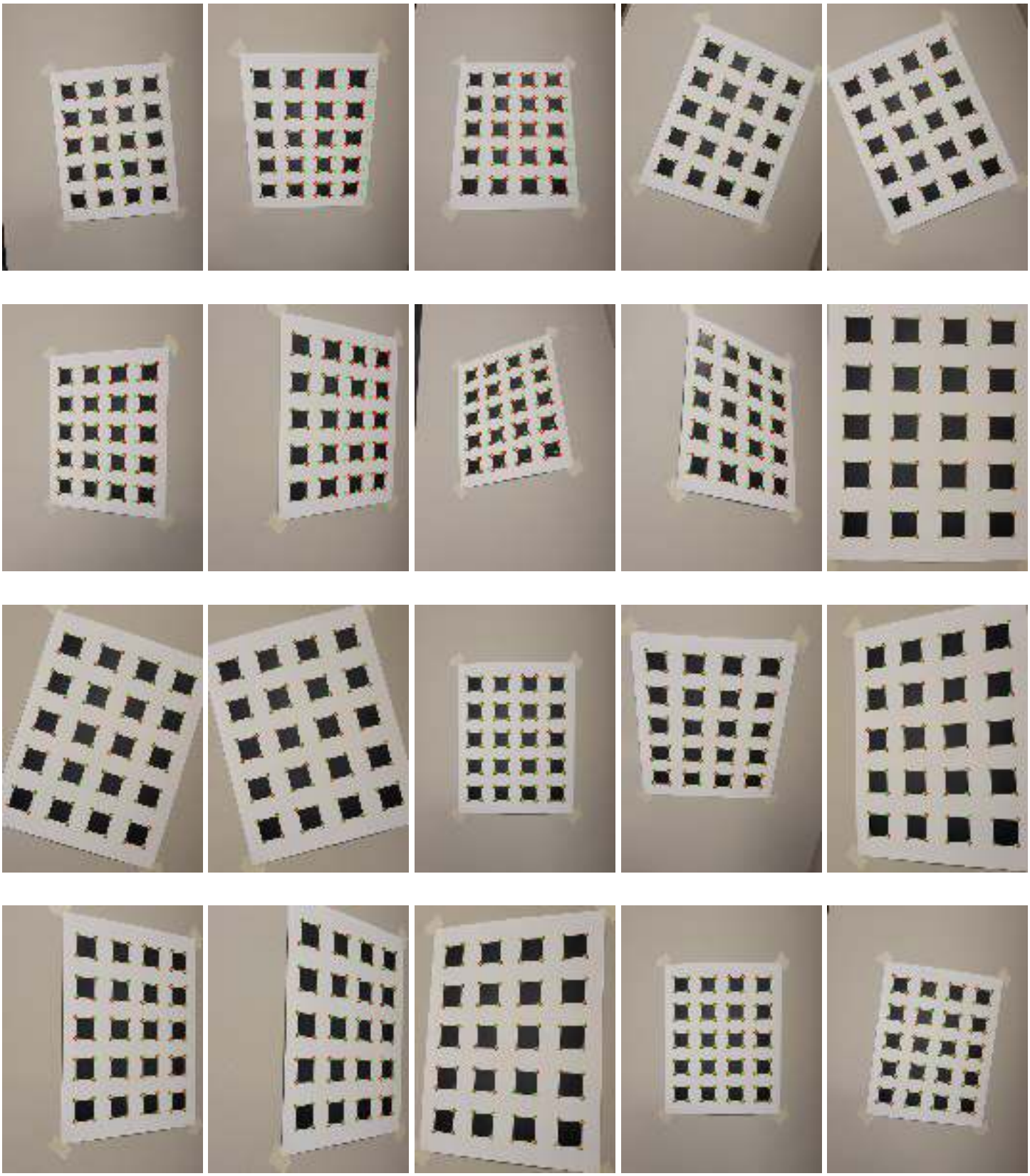

Figure 6: Reprojected world corners to each viewpoint before LM refinement (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.3 Reprojected corners from each viewpoint to the Fixed Image before LM refinement (Database 1)

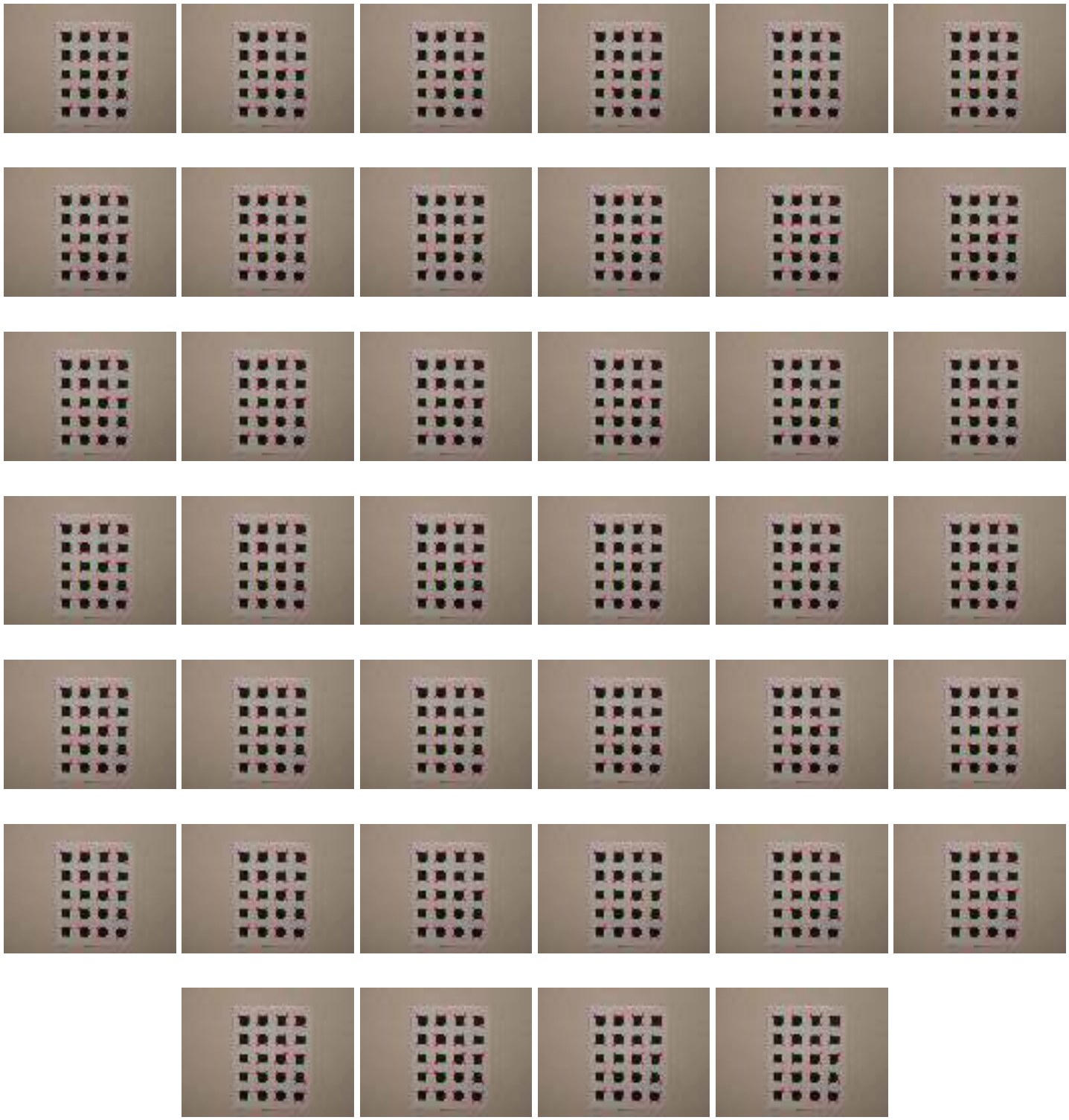

Figure 7: Reprojected corners from each viewpoint to the Fixed Image before LM refinement (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.4 Reprojected corners from each viewpoint to the Fixed Image before LM refinement (Database 2)

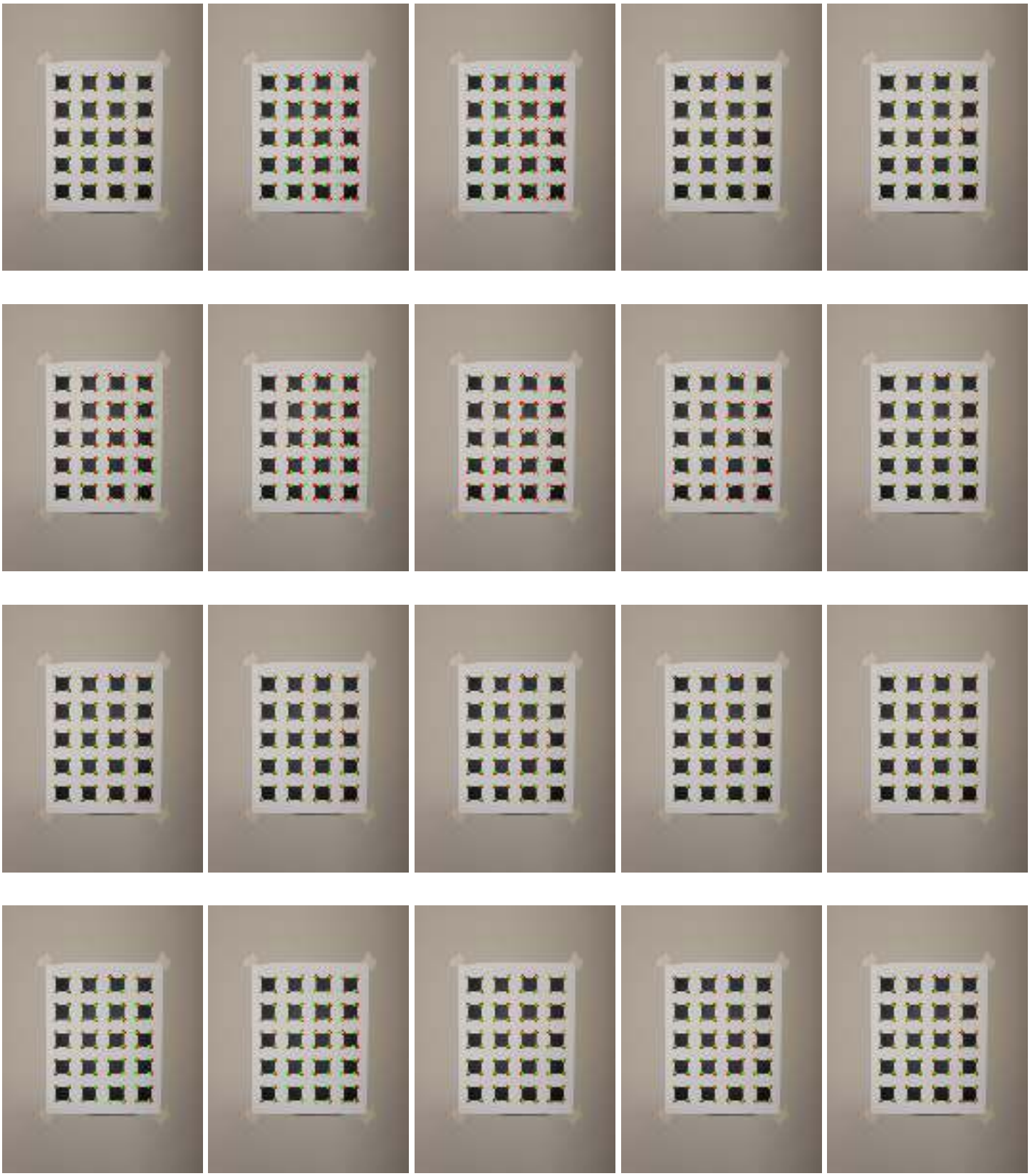

Figure 8: Reprojected corners from each viewpoint to the Fixed Image before LM refinement (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.5 Reprojected world corners to each viewpoint after LM refinement (Database 1)

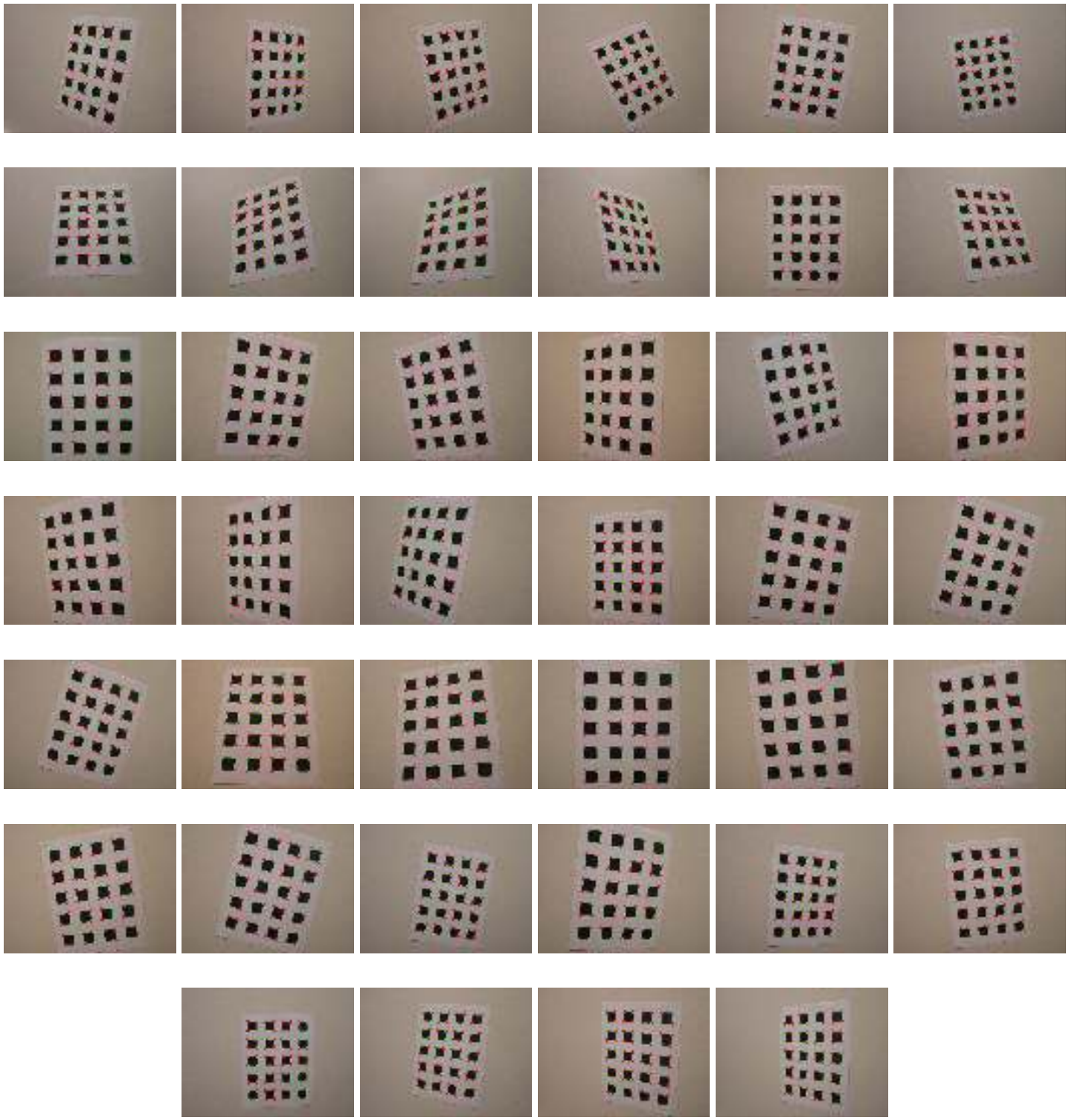

Figure 9: Reprojected world corners to each viewpoint after LM refinement (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.6 Reprojected world corners to each viewpoint after LM refinement (Database 2)

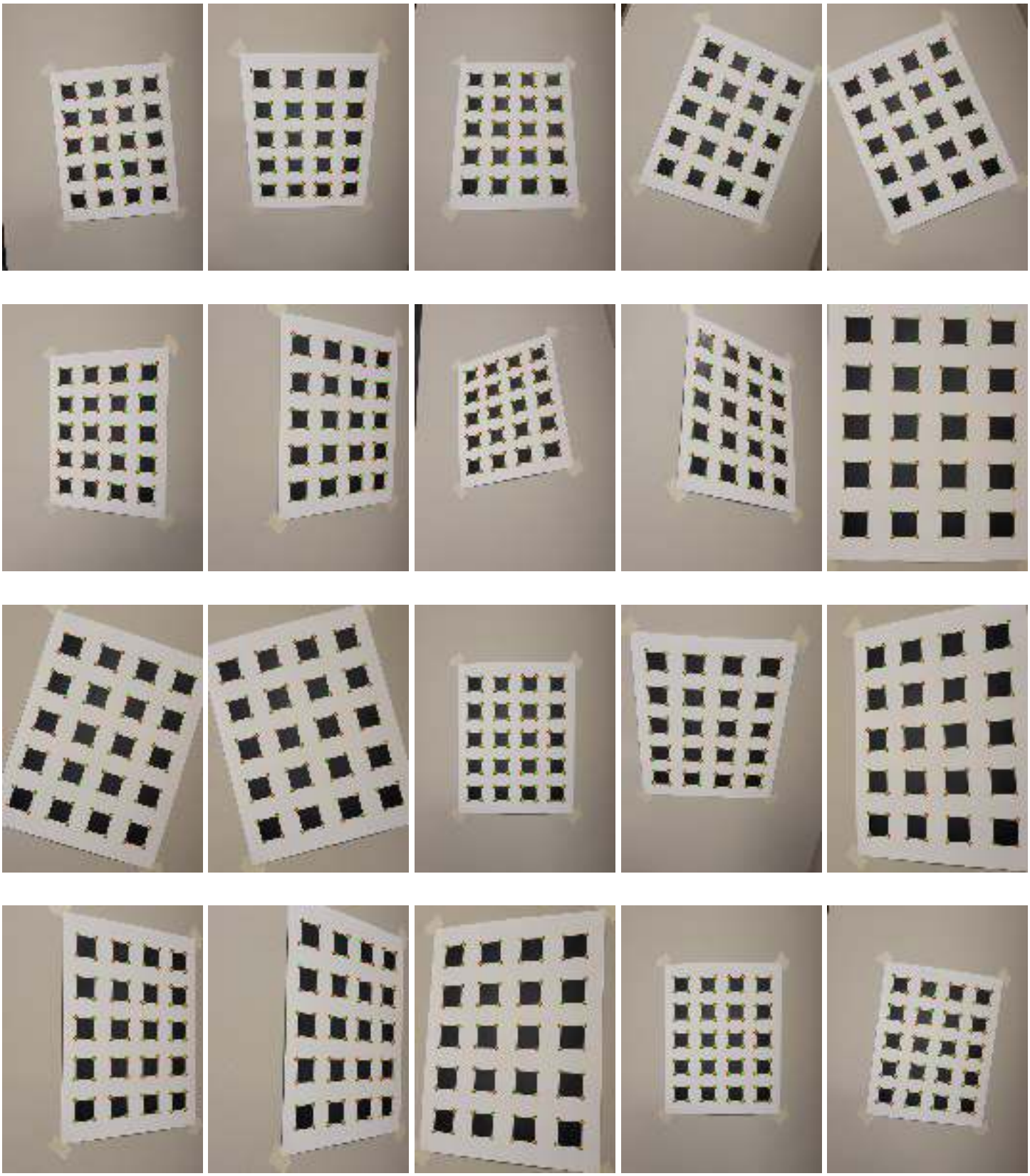

Figure 10: Reprojected world corners to each viewpoint after LM refinement (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.7 Reprojected corners from each viewpoint to the Fixed Image after LM refinement (Database 1)

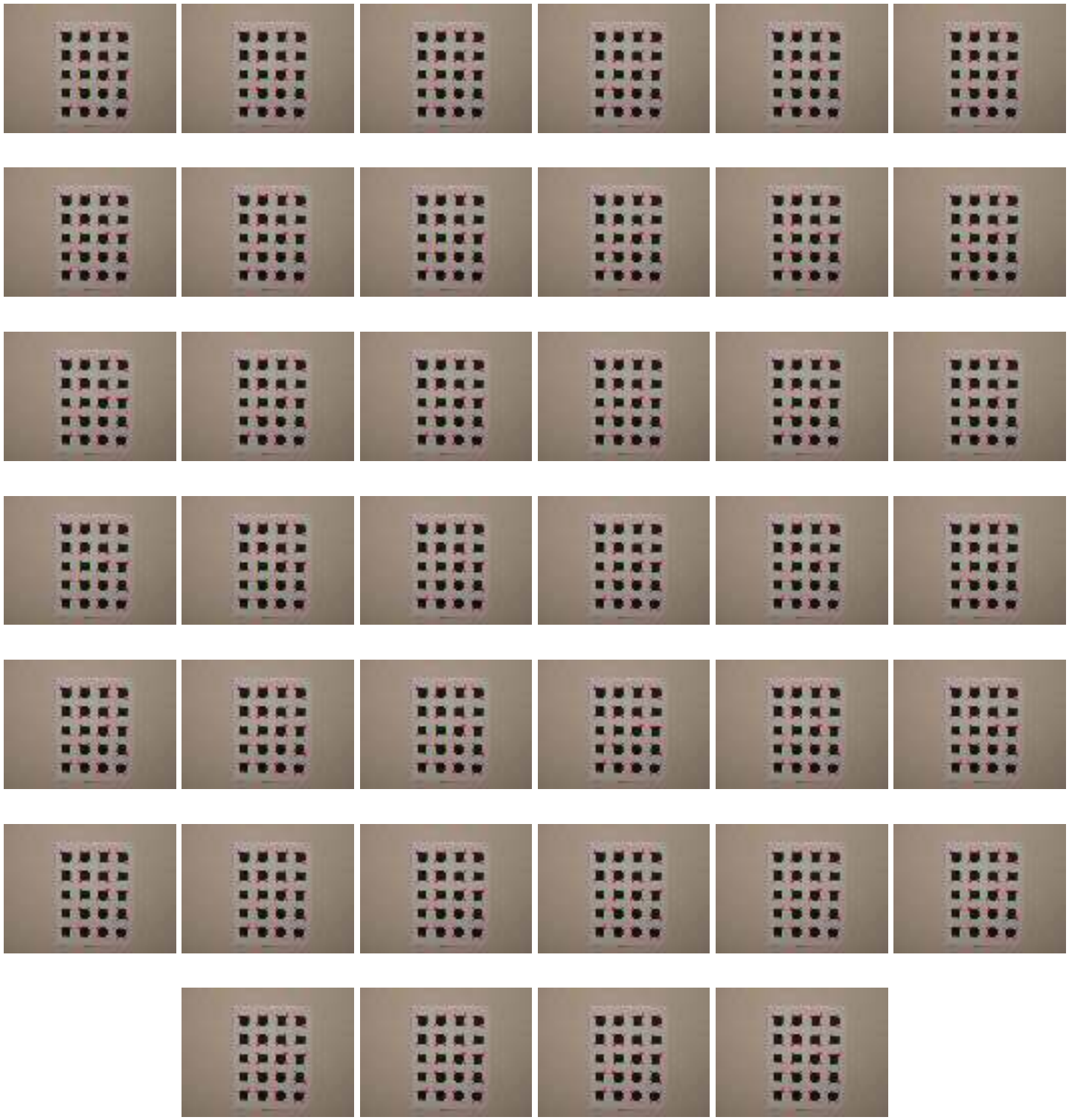

Figure 11: Reprojected corners from each viewpoint to the Fixed Image after LM refinement (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.8 Reprojected corners from each viewpoint to the Fixed Image after LM refinement (Database 2)

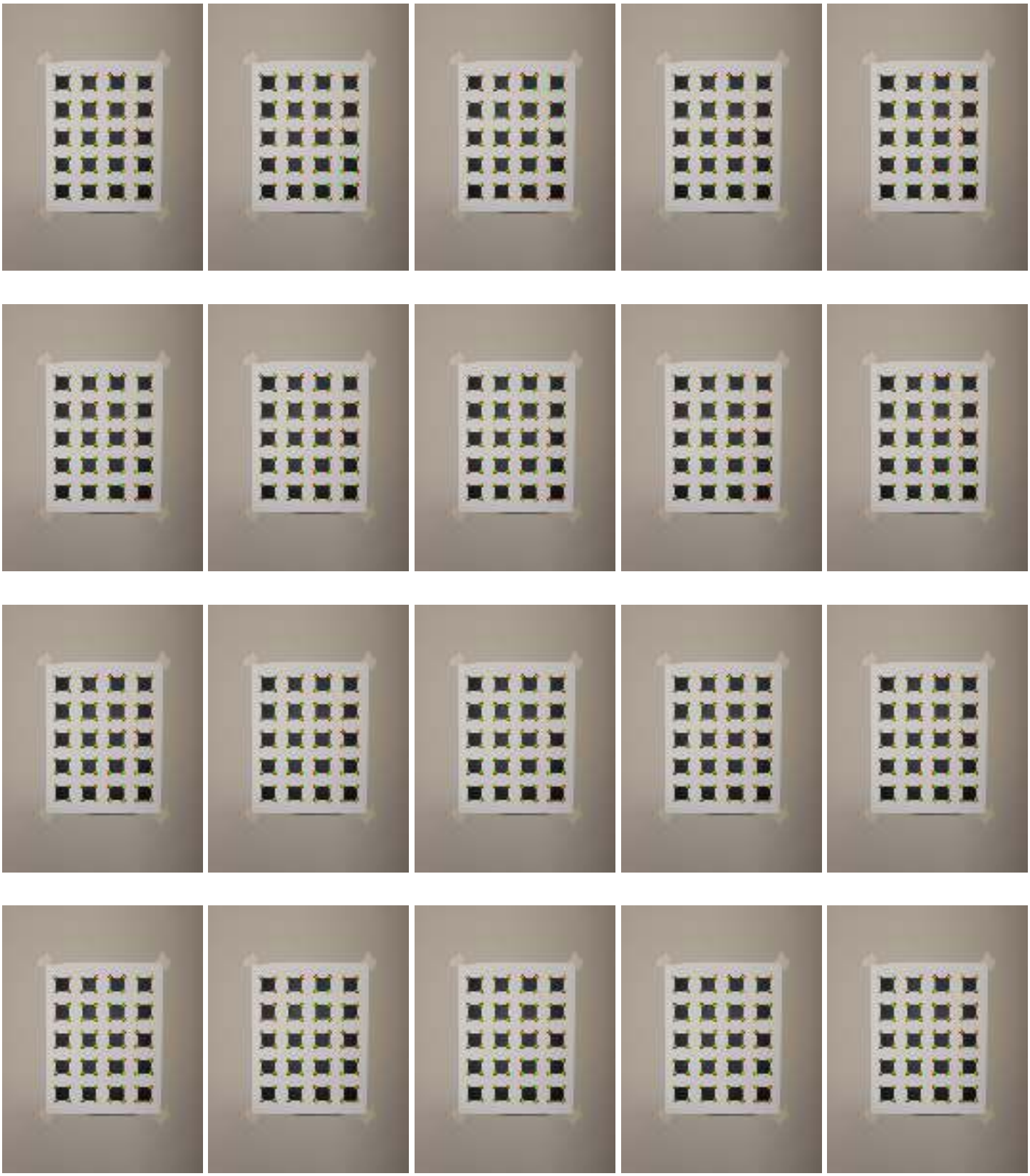

Figure 12: Reprojected corners from each viewpoint to the Fixed Image after LM refinement (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.9 Reprojected world corners to each viewpoint after LM refinement and fixing radial distortion (Database 1)

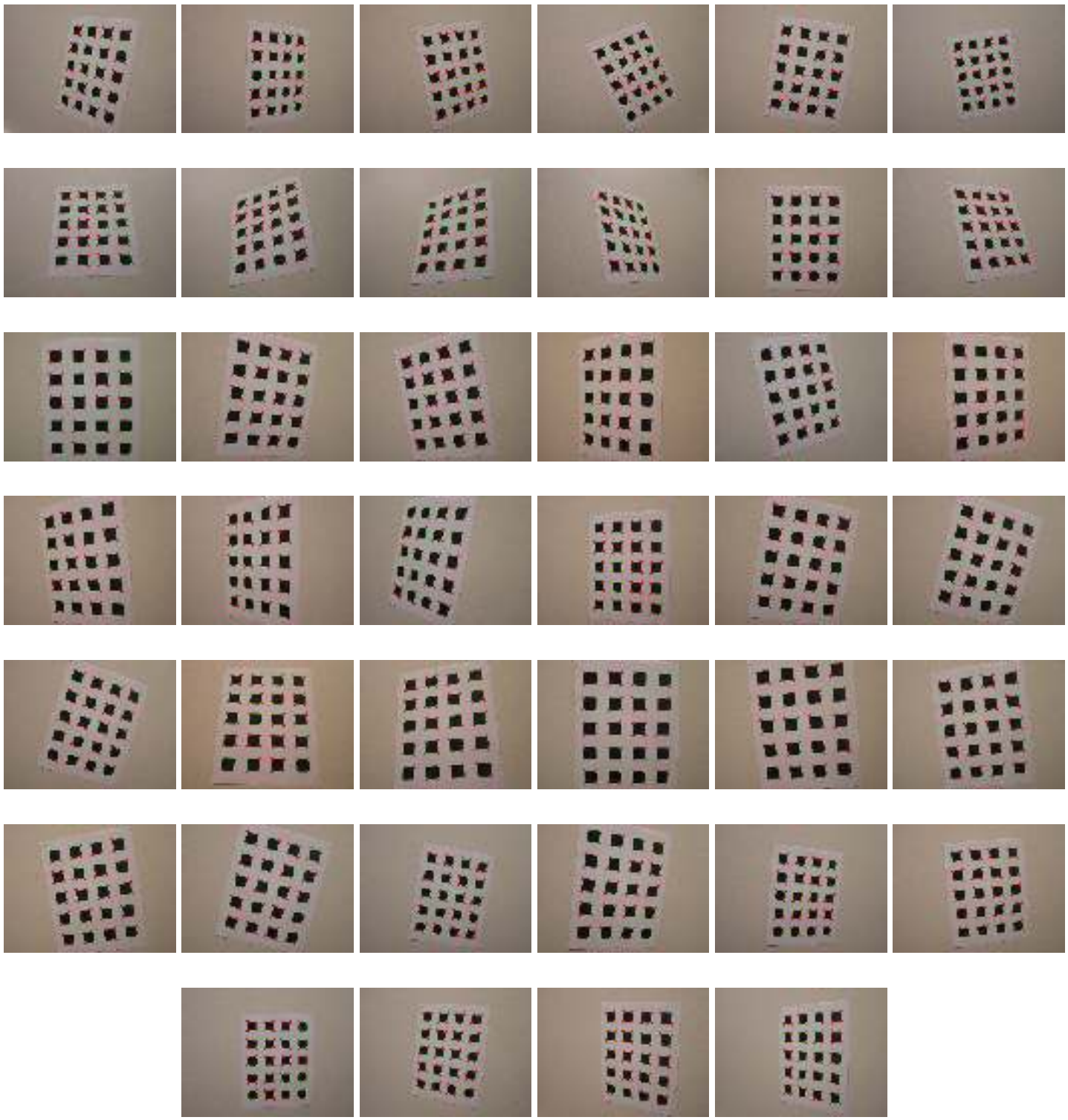

Figure 13: Reprojected world corners to each viewpoint after LM refinement and fixing radial distortion (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.10 Reprojected world corners to each viewpoint after LM refinement and fixing radial distortion(Database 2)

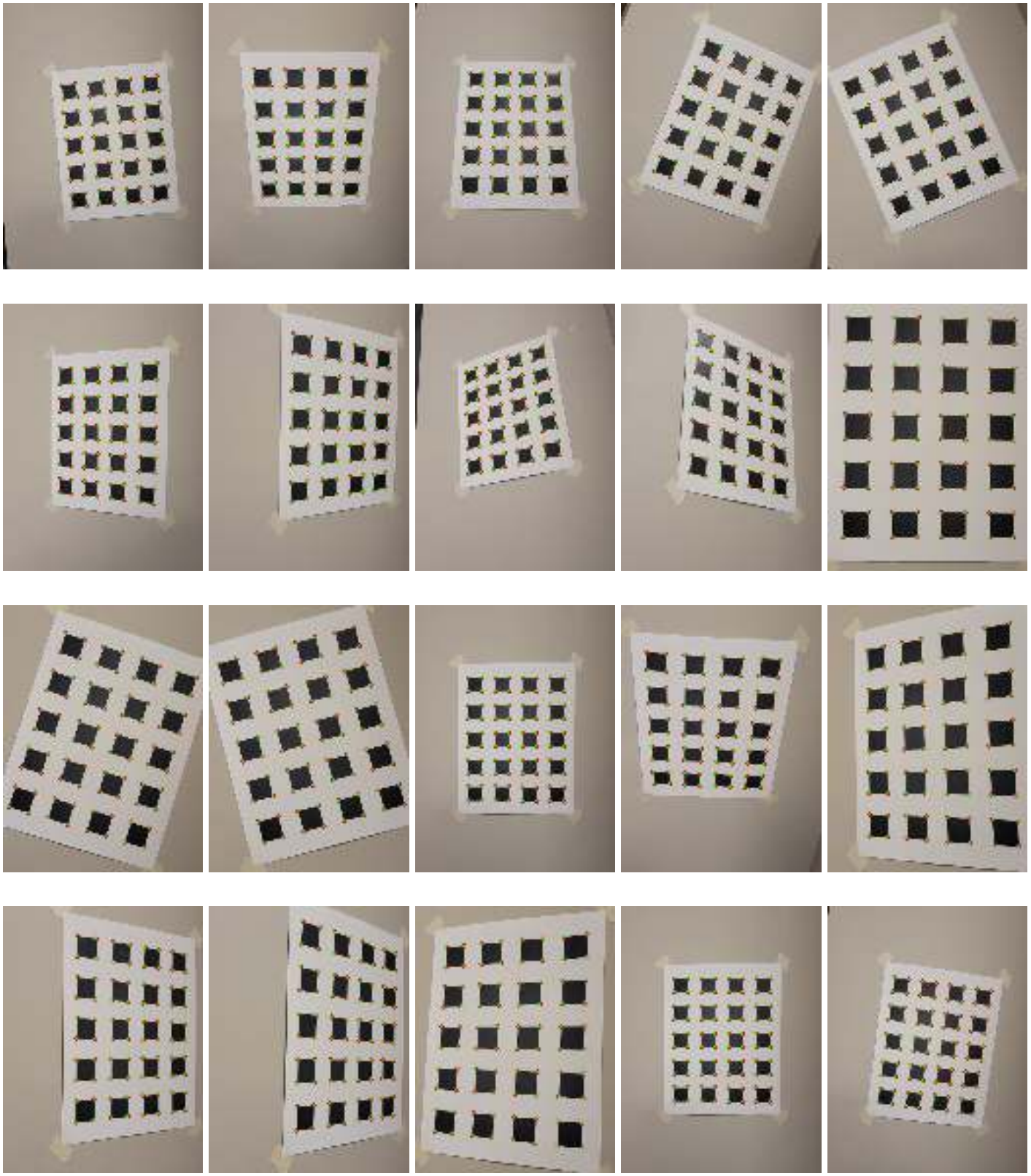

Figure 14: Reprojected world corners to each viewpoint after LM refinement and fixing radial distortion (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.11 Reprojected corners from each viewpoint to the Fixed Image after LM refinement and fixing radial distortion (Database 1)

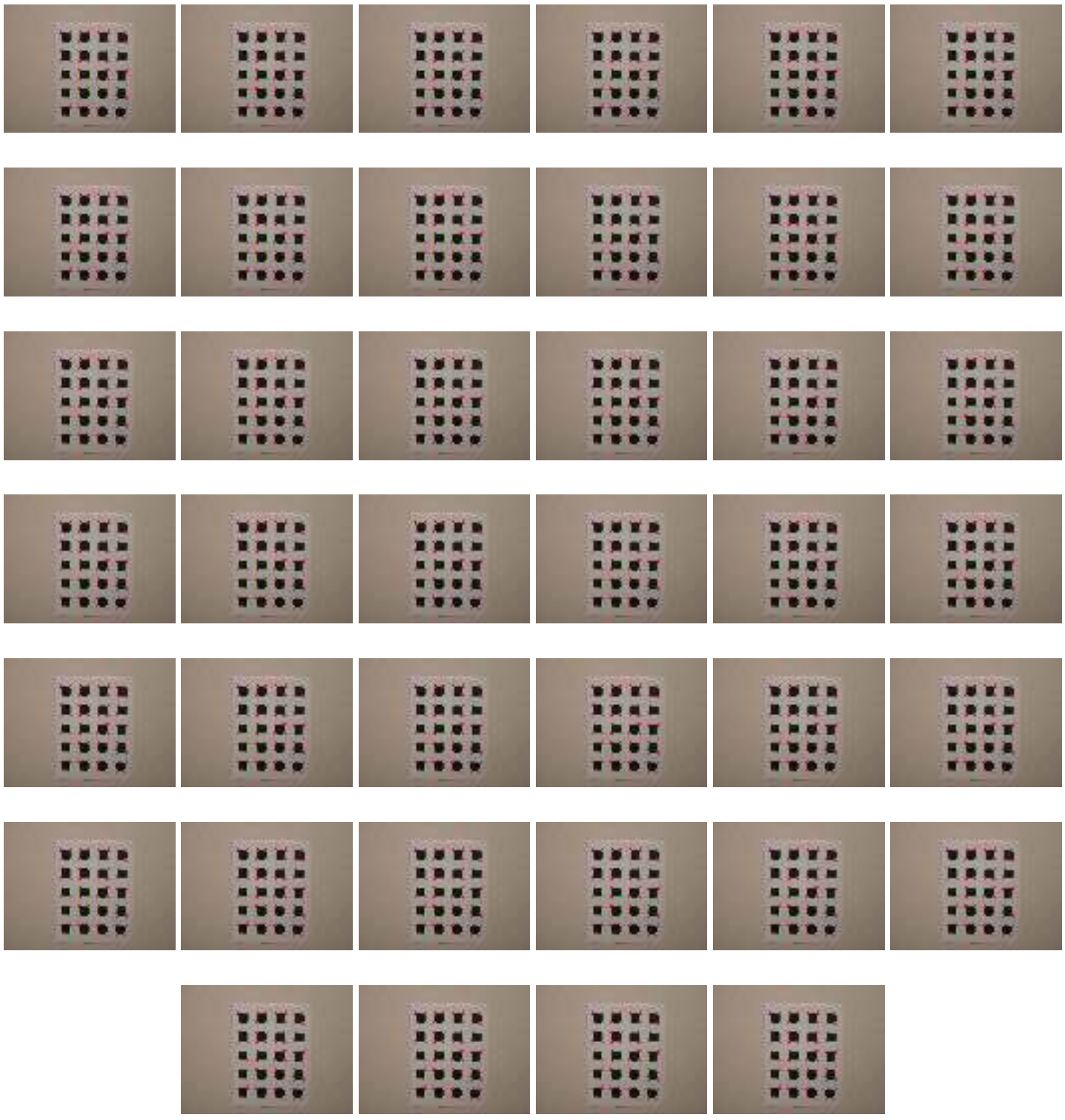

Figure 15: Reprojected corners from each viewpoint to the Fixed Image after LM refinement and fixing radial distortion (Database 1). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.6.12 Reprojected corners from each viewpoint to the Fixed Image after LM refinement and fixing radial distortion(Database 2)

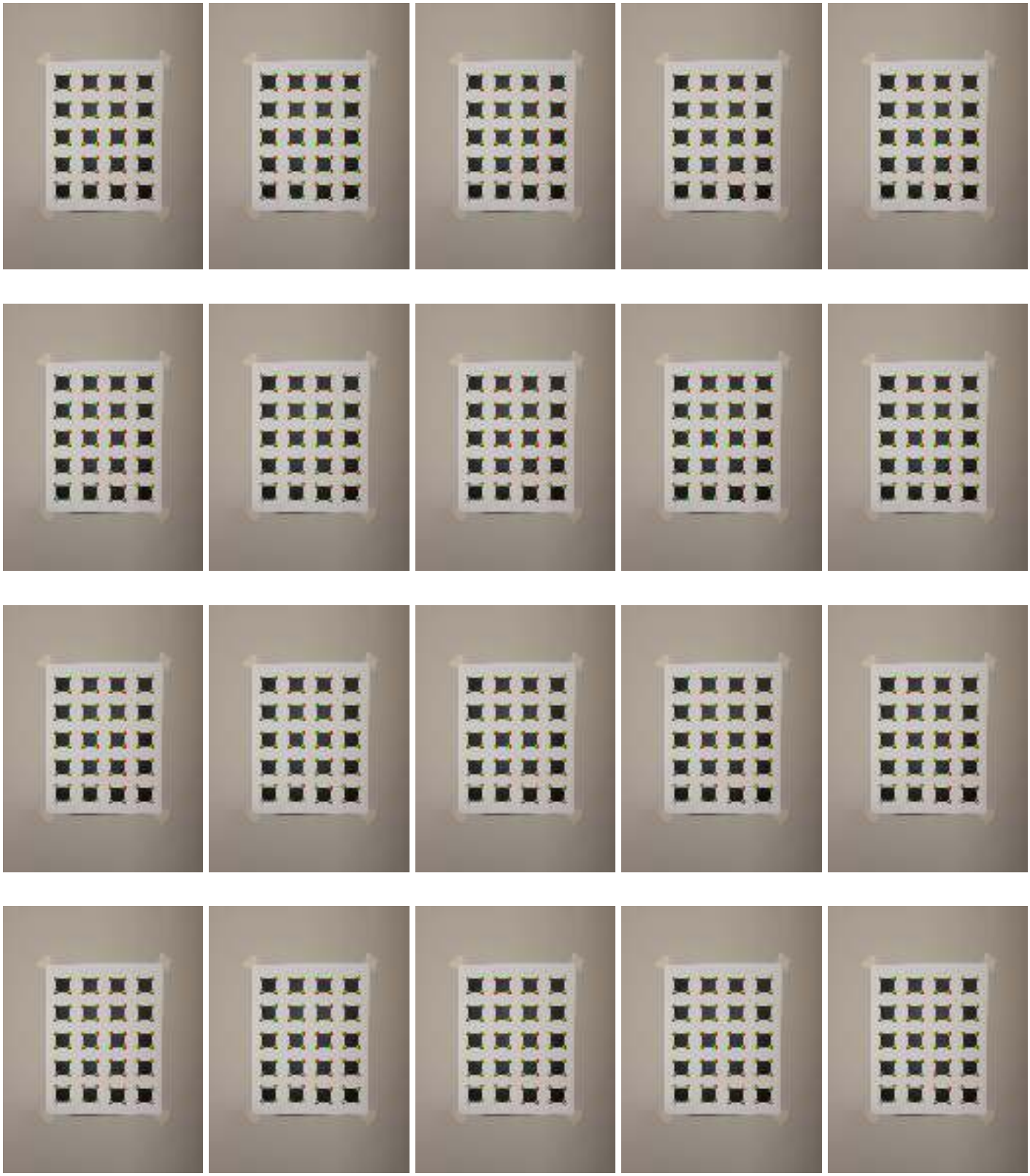

Figure 16: Reprojected corners from each viewpoint to the Fixed Image after LM refinement and fixing radial distortion (Database 2). Red Dots are the original intersection points. Green dots are the reprojected points. The red dots circles are plotted with a larger radius to visually compare the projection accuracy with green dots.

4.7 Mean and Variances of reprojection error for each viewpoint to the Fixed Image

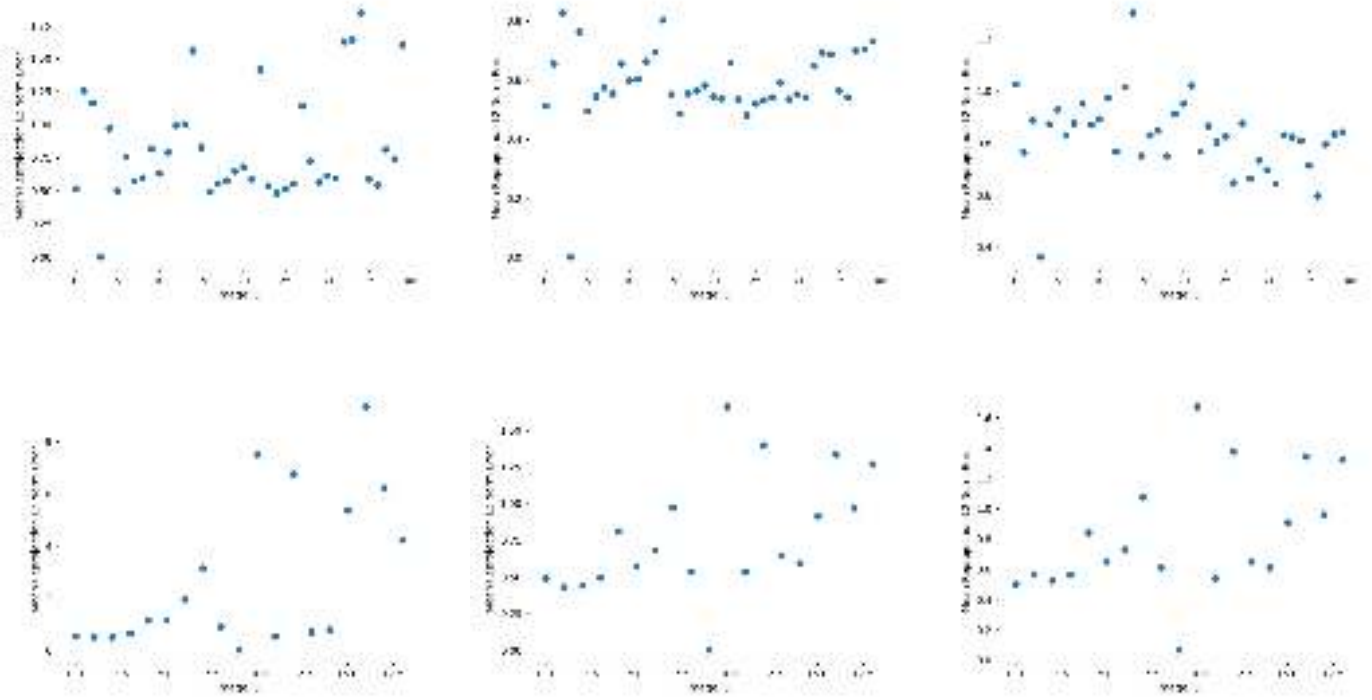

Figure 17: Mean reprojection error for each viewpoint to the Fixed Image (Database 1 (top row) and Database 2 (bottom row)). The left column is with LM refinement, the middle column is with LM refinement and the right column is for LM refinement with fixing radial distortion.

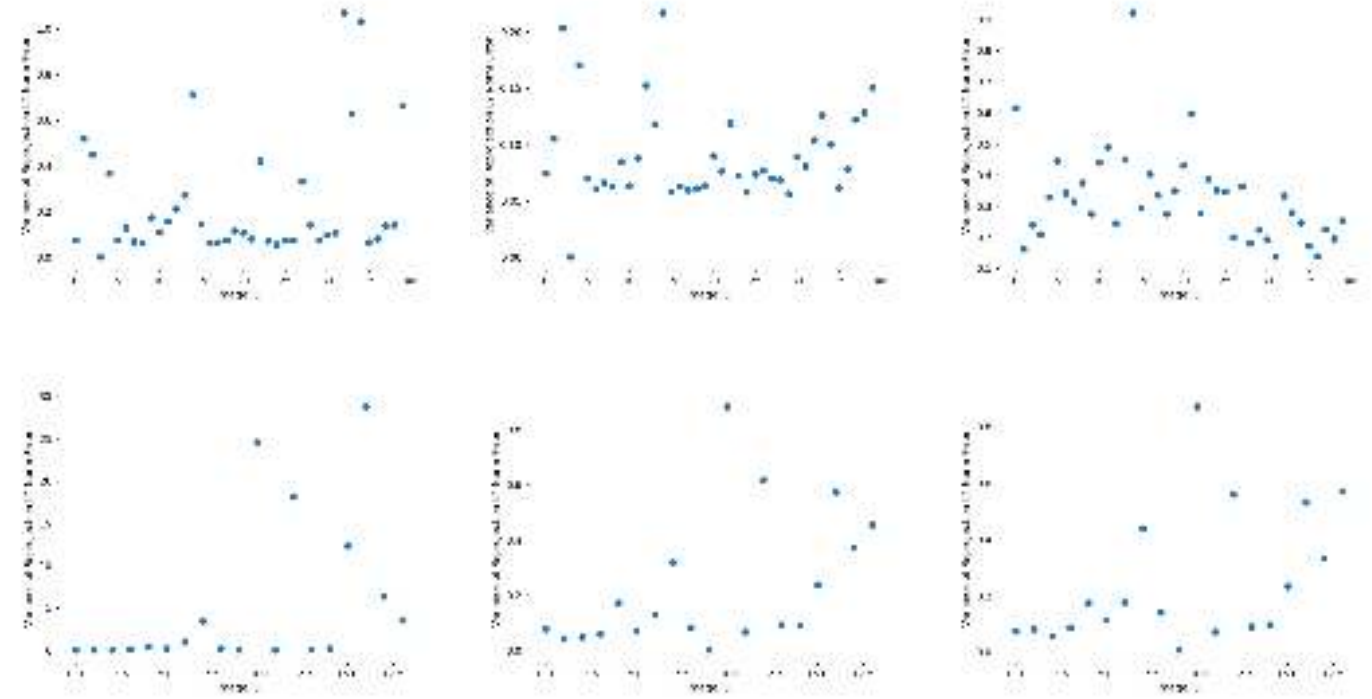

Figure 18: Variances of reprojection error for each viewpoint to the Fixed Image (Database 1 (top row) and Database 2 (bottom row)). The left column is with LM refinement, the middle column is with LM refinement and the right column is for LM refinement with fixing radial distortion.

4.8 Mean and Variances of reprojection error for from ground truth to each viewpoint

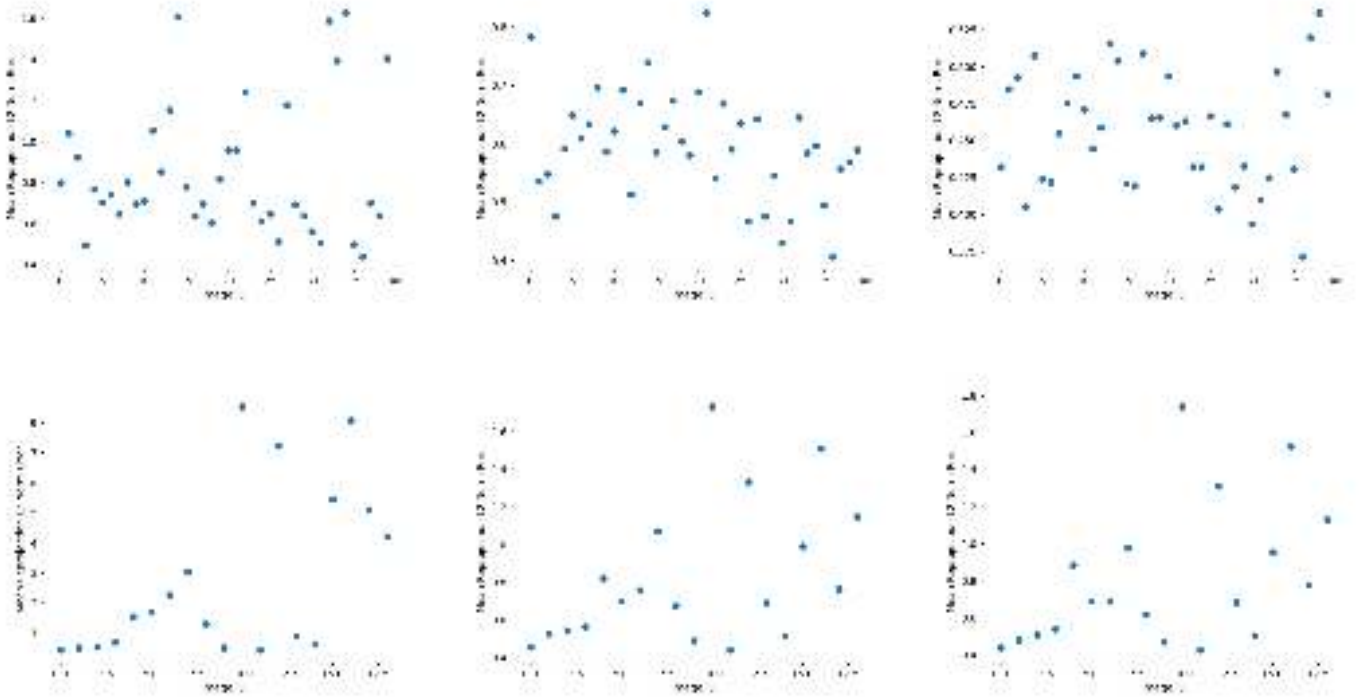

Figure 19: Mean reprojection error from ground truth to each viewpoint (Database 1 (top row) and Database 2 (bottom row)). The left column is with LM refinement, the middle column is with LM refinement and the right column is for LM refinement with fixing radial distortion.

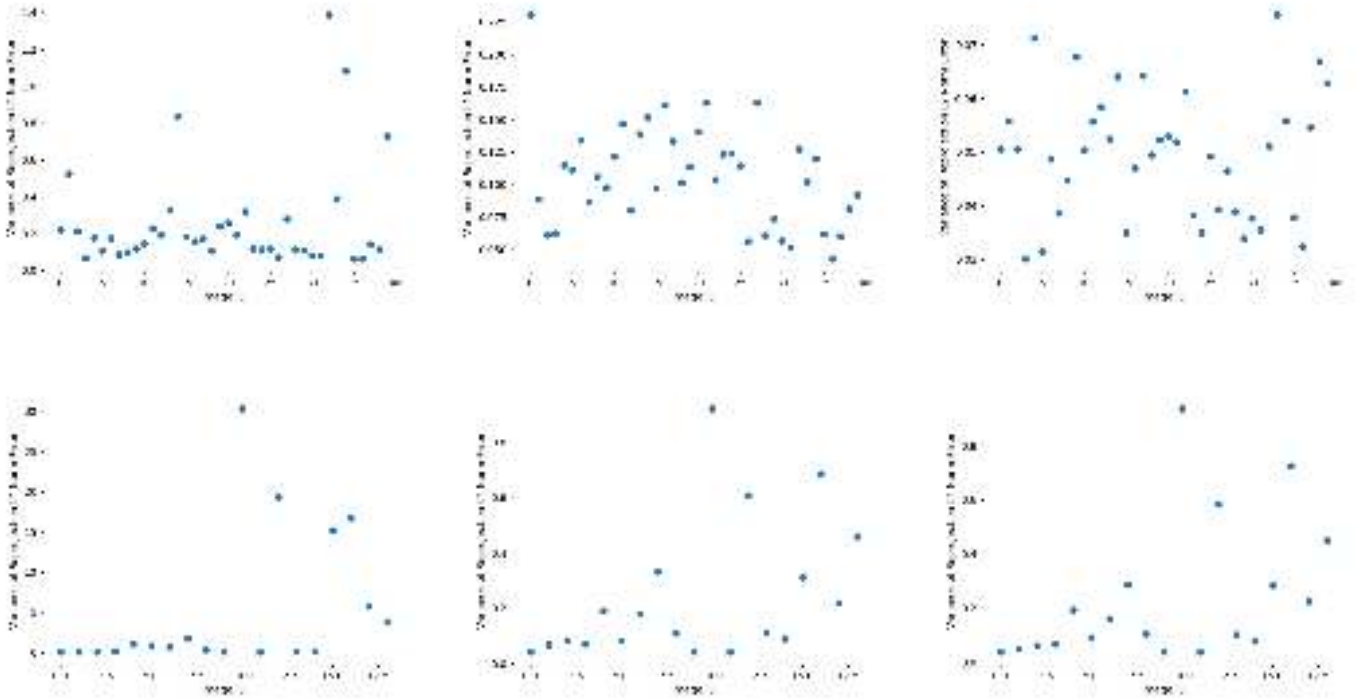

Figure 20: Variances of reprojection error from ground truth to each viewpoint (Database 1 (top row) and Database 2 (bottom row)). The left column is with LM refinement, the middle column is with LM refinement and the right column is for LM refinement with fixing radial distortion.

4.9 Camera Intrinsic and Extrinsic Parameters

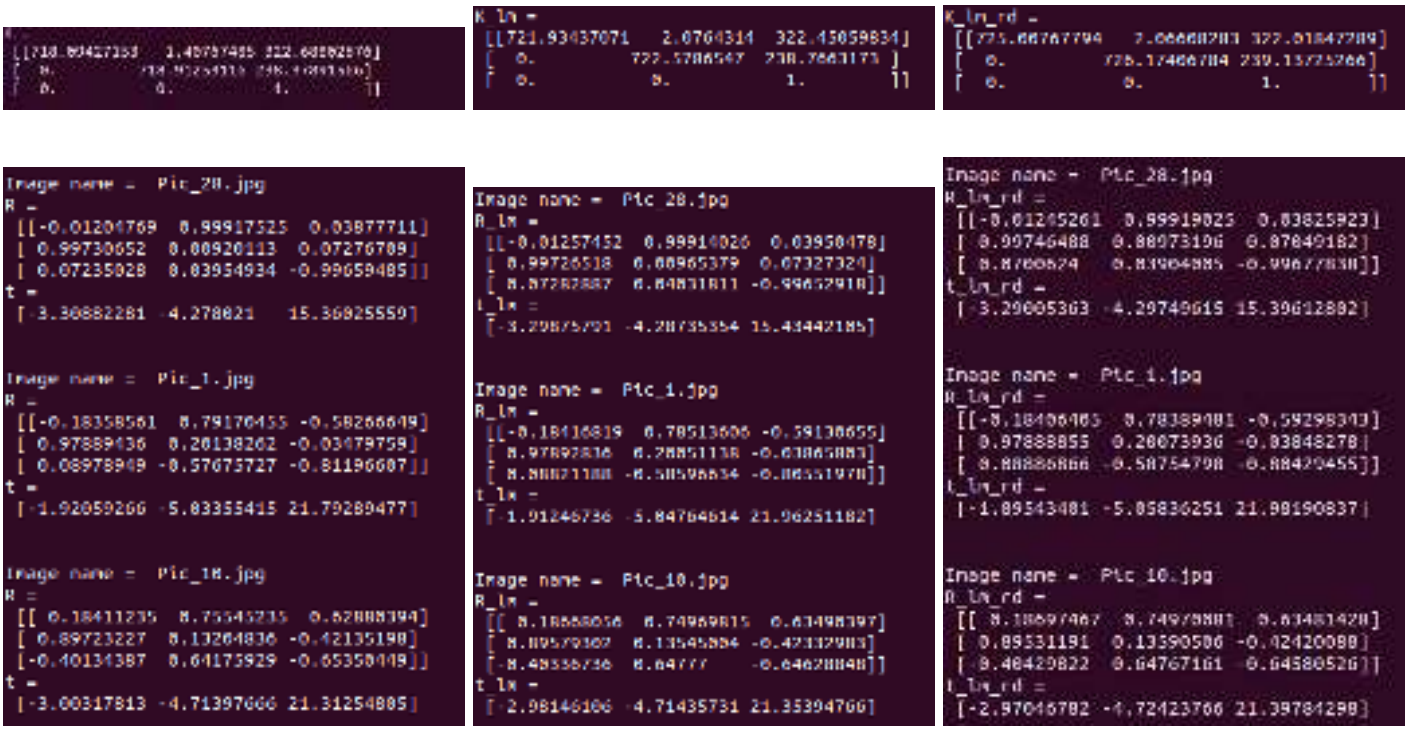

Figure 21: Intrinsic and Extrinsic Parameters of the camera from Database 1. The extrinsic parameters are shown for three sample images. The left column is before LM refinement, the middle column corresponds to after LM refinement and the right column is after LM refinement with fixing radial distortion. The parameters for the rest of the images in the dataset can be checked at the terminal after running the source code.

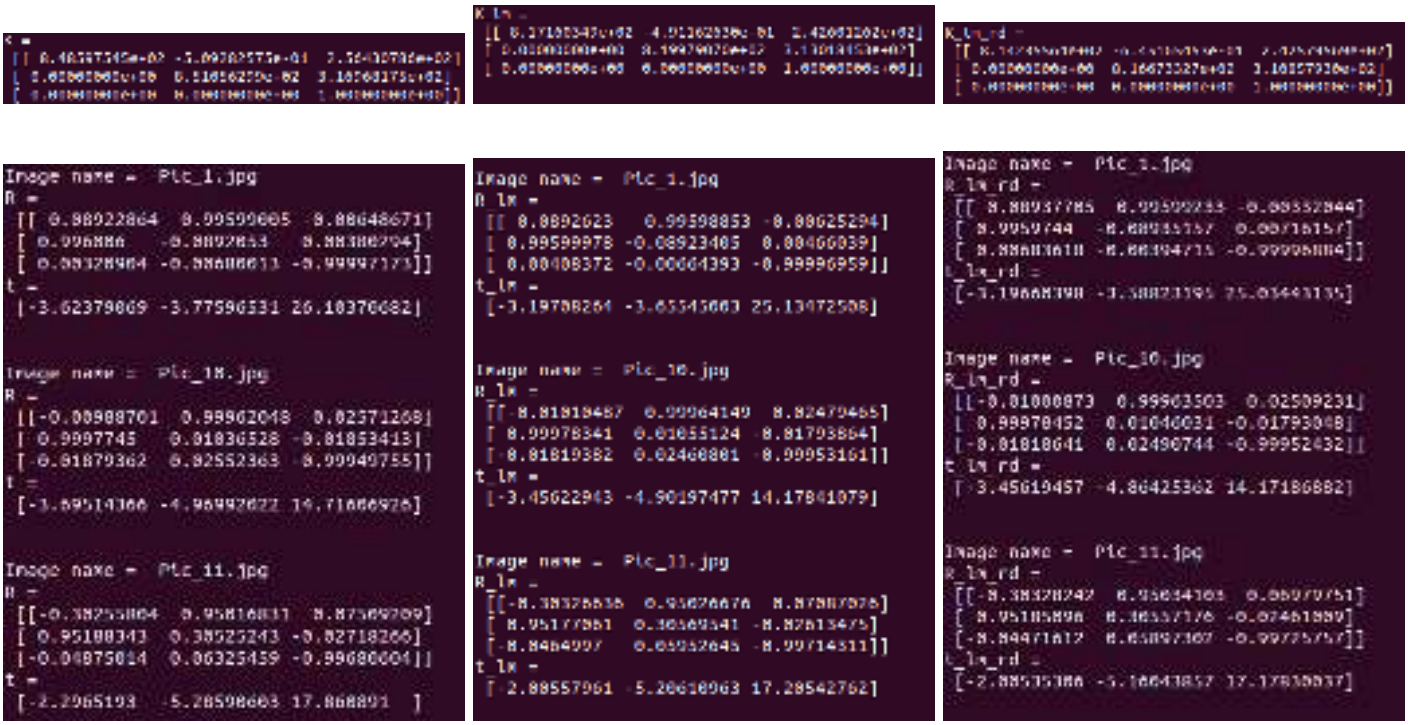

Figure 22: Intrinsic and Extrinsic Parameters of the camera from Database 2. The extrinsic parameters are shown for three sample images. The left column is before LM refinement, the middle column corresponds to after LM refinement and the right column is after LM refinement with fixing radial distortion. The parameters for the rest of the images in the dataset can be checked at the terminal after running the source code.

5 Discussion

6 References

1 Zhengyou Zhang. A flexible new technique for camera calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22:1330-1334, December 2000.

2 Liu, Y., Liu, S., Cao, Y., & Wang, Z. (2016). Automatic chessboard corner detection method. IET Image Processing, 10(1), 16-23.

7 Source Code

```
import cv2
import os
import matplotlib.pyplot as plt
import numpy as np
from scipy import optimize


def form_line(pt1,pt2):
        if len(pt1)<3:
                pt1.append(1)
        if len(pt2)<3:
                pt2.append(1)
        t = np.cross(pt1,pt2)
        if t[2]!= 0:
                line = t/t[2]
        else:
                line = t
        return line
def form_intersection(line_1,line_2):
        t = np.cross(line_1,line_2)
        if t[2]!= 0:
                intersection = t/t[2]
        else:
                intersection = t
        return intersection
def edge_detection(im,save_path):
        edges = cv2.Canny(im,400,400)
        fig = plt.figure()
        plt.subplot(121)
        plt.imshow(im,cmap='gray')
        plt.title('Original Image')
        plt.subplot(122)
        plt.imshow(edges,cmap='gray')
        plt.title('Edges')
        plt.savefig(save_path)
        plt.close()
        return edges
def hough_transformation(im,edges,hough_transformation_threshold,save_path):
        img = im.copy()
        lines = cv2.HoughLinesP(edges,1,np.pi/180,hough_transformation_threshold,minLineLength=1
        fig = plt.figure()
        plt.subplot(131)
        plt.imshow(img,cmap='gray')
        plt.title('Original Image')
        plt.subplot(132)
        plt.imshow(edges,cmap='gray')
        plt.title('Edges')
        plt.subplot(133)
        for L in lines:
                x1,y1,x2,y2 =  L[0]
                cv2.line(img, (x1,y1), (x2,y2), (255,0,0), 3)
        plt.imshow(img,cmap='gray')
        plt.title('Hough transformation Lines')
        plt.tight_layout()
        plt.savefig(save_path)
        plt.close()
        return lines
def plot_intersections(im,edges,lines,intersections,save_path):
```

```
        img1 = im.copy()
        img2 = im.copy()
        fig = plt.figure()
        plt.subplot(221)
        plt.imshow(img1)
        plt.title('Original Image')
        plt.subplot(222)
        plt.imshow(edges,cmap='gray')
        plt.title('Edges')
        plt.subplot(223)
        for L in lines:
                x1,y1,x2,y2 =  L[0]
                cv2.line(img1, (x1,y1), (x2,y2), (255,0,0), 3)
        plt.imshow(img1,cmap='gray')
        plt.title('Hough transformation Lines')
        plt.subplot(224)
        for ix in range(0,len(intersections)):
                img2 = cv2.circle(img2,tuple(intersections[ix]),radius=2,color=(255, 0, 0),thickn
        plt.imshow(img2)
        plt.title('Intersections')
        fig.tight_layout()
        plt.savefig(save_path)
        plt.close()
def unique_corner_detection(im,intersections,database):
        intersections = np.array(intersections).squeeze()
        intersections = np.reshape(intersections,[intersections.shape[0],2])
        if database.upper() == 'DATASET1':
                bg_gray_level = [100,125]
        elif database.upper() == 'DATASET2':
                bg_gray_level = [100,180]
        Intersections_refined_first = list()
        if len(im.shape)>2:
                im_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
        else:
                im_gray = im
        for ix in range(0,intersections.shape[0]):
                if not (im_gray[intersections[ix,1],intersections[ix,0]] >= bg_gray_level[0] and
                        Intersections_refined_first.append(intersections[ix,:])
        intersections = np.reshape(np.array(Intersections_refined_first),[-1,2])
        '''
        Intersections_refined_second = list()
        indices = np.argwhere(im_gray<=100)
        indices = np.reshape(indices,[-1,2])
        for ix in range(0,intersections.shape[0]):
                current_pixel = np.reshape(np.array(intersections[ix,:]),[1,2])
                distance_from_current_pixel = np.linalg.norm(indices-current_pixel,axis=1)
                #print(distance_from_current_pixel)
                indices_2 = np.argwhere(distance_from_current_pixel<=100)
                if len(indices_2 > 0):
                        Intersections_refined_second.append(current_pixel)
        intersections = np.reshape(np.array(Intersections_refined_second),[-1,2])
        print(intersections.shape)
        '''

        Intersections_refined_third = list()
        while intersections.shape[0] > 0:
                distances = np.sqrt(np.sum(np.square(np.subtract(intersections,intersections[0,:]
                indices = np.argwhere(distances <= 10.0)
                avg = (np.sum(intersections[indices,:],axis=0)/len(indices)).astype(int)
                interest_point = avg.squeeze().tolist()
```

```
                #second stage refinement based on neighboring points. Find the distance and reje
                distances = np.sqrt(np.sum(np.square(np.subtract(intersections,interest_point)),
                indices = np.argwhere(distances <= 20.0)
                if len(indices>1):
                        if not (interest_point[0]==0 and interest_point[1]==0):
                                if not (interest_point[0] < 40 or interest_point[0] >530):
                                        Intersections_refined_third.append(interest_point)
                intersections = np.delete(intersections,indices,0)
        intersections =Intersections_refined_third
        return intersections
def label_corners(intersections):
        intersections = np.array(intersections).squeeze()
        iter = 1
        while intersections.shape[0] > 0:
                top_left_point = intersections[np.argmin(np.sum(intersections,axis=1)),:]#find th
                top_right_point = intersections[np.argmax(intersections[:,0]-intersections[:,1])
                distances_from_point_to_line = list()
                for ix in range(0,intersections.shape[0]):
                        distances_from_point_to_line.append(np.abs((top_right_point[0]-top_left_p
                distances_from_point_to_line = np.array(distances_from_point_to_line)
                min_distance = np.min(distances_from_point_to_line)
                if min_distance == 0:
                        threshold = [0,10]
                else:
                        threshold = [min_distance-10,min_distance+10]
                idx = np.where((distances_from_point_to_line >= threshold[0])&(distances_from_po

                point_block = intersections[idx,:].squeeze()
                point_block_sorted = point_block[point_block[:,0].argsort()]
                if iter == 1:
                        intersections_sorted = point_block_sorted
                else:
                        intersections_sorted = np.concatenate((intersections_sorted,point_block_s
                intersections = np.delete(intersections,idx,0)
                iter = iter+1
        return intersections_sorted
def plot_corner_labels(im,intersections,save_path):
        img = im.copy()
        for ix in range(0,len(intersections)):
                img = cv2.circle(img,tuple(intersections[ix]),radius=2,color=(255, 0, 0),thickne
                cv2.putText(img=img, text=str(ix+1), org=tuple(intersections[ix]), fontFace=cv2.F
        cv2.imwrite(save_path, img)
def Homography_Matrix(domain_pt,range_pt):
        #estimate the homography matrix from interest point pairs
        #domain_pt = shape:[N,2]
        #range_pt = shape:[N,2]
        A = np.zeros((2*domain_pt.shape[0],8))
        b = np.zeros((2*domain_pt.shape[0],1))
        for ix in range(0,domain_pt.shape[0]):
                A[2*ix,:] = np.array([domain_pt[ix,0],domain_pt[ix,1],1,0,0,0,-domain_pt[ix,0]*ra
                A[2*ix+1,:] = np.array([0,0,0,domain_pt[ix,0],domain_pt[ix,1],1,-domain_pt[ix,0]*
                b[2*ix,0] = range_pt[ix,0]
                b[2*ix+1,0] = range_pt[ix,1]

        if A.shape[0]==A.shape[1]:
                H = np.matmul(np.inalg.inv(A),b)
        else:
                H = np.matmul(np.linalg.pinv(A),b)
        H = np.reshape(np.append(H,1),(3,3))
        return H
```

```
def calculate_pseudo_inverse(A):
        return np.matmul(np.linalg.inv(np.matmul(A.T,A)),A.T)
def build_pattern(box_size, h_box, v_box):

        world_corners = np.zeros((80,2))
        for ix in range(0,2*v_box):
                for jx in range(0,2*h_box):
                        world_corners[ix*8+jx,:] = [ix,jx]
        return world_corners
def compute_omega(H_matrices):
        def compute_v(H):
                #return np.array([H[0,i]*H[j,0], H[0,i]*H[j,1]+H[1,i]*H[j,0], H[1,i]*H[j,1], H[2
                h_11,h_12,h_13 = H[0,0], H[1,0], H[2,0]
                h_21,h_22,h_23 = H[0,1], H[1,1], H[2,1]
                V_local = np.array([[h_11*h_21, h_11*h_22+h_12*h_21, h_12*h_22, h_13*h_21+h_11*h
                                                [h_11**2-h_21**2, 2*h_11*h_12-2*h_21*h_22, h_12**
                return V_local

        V = list()
        for H in H_matrices:
                V.append(compute_v(H))
                #V.append(compute_v(H,1,0))
                #V.append(np.subtract(compute_v(H,0,0),compute_v(H,1,1)))
        V= np.array(V).squeeze()
        V = np.reshape(V,[len(H_matrices)*2,6])
        _,_,v = np.linalg.svd(V)
        b = v[-1]
        omega = np.array([[b[0],b[1],b[3]],[b[1],b[2],b[4]],[b[3],b[4],b[5]]])
        return omega
def compute_K(omega):
        K = np.zeros((3,3))
        K[1,2] = (omega[0,1]*omega[0,2]-omega[0,0]*omega[1,2])/(omega[0,0]*omega[1,1] - omega[0,1
        lambd = omega[2,2]-((omega[0,2]**2 + K[1,2]*(omega[0,1]*omega[0,2] - omega[0,0]*omega[1,2
        K[0,0] = np.sqrt(lambd/omega[0,0])
        K[1,1] = np.sqrt((lambd*omega[0,0])/(omega[0,0]*omega[1,1] - omega[0,1]**2))
        K[0,1] = -(omega[0,1]*K[0,0]**2*K[1,1])/lambd
        K[0,2] = (K[0,1]*K[1,2])/K[1,1] - (omega[0,2]*K[0,0]**2)/lambd
        K[2,2] = 1
        return K
def compute_extrinsic_parameters(H_matrices,K):
        R_list = list()
        t_list = list()
        for H in H_matrices:
                R12t = np.matmul(np.linalg.inv(K),H)
                factor = 1/np.linalg.norm(R12t[:,0])
                R12t = factor*R12t
                R3 = np.cross(R12t[:,0],R12t[:,1])
                t_list.append(R12t[:,2])
                #Find an orthonormal rotation matrix
                R123 = np.concatenate((R12t[:,0:2],np.reshape(R3,[3,1])),axis=1)
                u,_,v = np.linalg.svd(np.reshape(R123,[3,3]))
                R_list.append(np.dot(u,v))
        return R_list,t_list
def form_parametric_representation(R_list,t_list,K):
        parametric_Rt = list()
        for R,t in zip(R_list,t_list):
                angle = np.arccos((np.trace(R)-1)/2)
                parametric_omega = angle/(2*np.sin(angle))*np.array([R[2,1]-R[1,2], R[0,2]-R[2,0]
                parametric_Rt.append(np.append(parametric_omega,t))
        parametric_K = np.array([K[0,0],K[0,1],K[0,2],K[1,1],K[1,2]])
```

```
        parameters = np.append(np.concatenate(parametric_Rt),parametric_K)
        return parameters
def transform_image(H,input_data):

        input_data = input_data.T
        input_data = np.concatenate((input_data,np.ones((1,input_data.shape[1]))),axis=0)
        transformed_image = np.matmul(H,input_data)
        transformed_image = transformed_image[0:2,:]/transformed_image[2,:]
        return transformed_image[0:2,:].T
def reconstruct_parameters(parameters,radial_distort=False):
        parameters = parameters.squeeze()
        if radial_distort:
                k1 = parameters[-2]
                k2 = parameters[-1]
                x_0 = parameters[-5]
                y_0 = parameters[-3]
                parameters = parameters[:-2]
        k = parameters[-5:]
        K = np.zeros((3,3))
        K[0,0] = k[0]
        K[0,1] = k[1]
        K[0,2] = k[2]
        K[1,1] = k[3]
        K[1,2] = k[4]
        K[2,2] = 1
        R_list = list()
        t_list = list()
        for ix in range(0,len(parameters)-5,6):
                w = parameters[ix:ix+3]
                t = parameters[ix+3:ix+6]
                W = np.array([[0,-w[2],w[1]], [w[2],0,-w[0]], [-w[1],w[0],0]])
                phi = np.linalg.norm(w)
                R = np.eye(3) + np.sin(phi)/phi*W + (1-np.cos(phi))/(phi**2)*np.dot(W,W)
                R_list.append(np.reshape(R,[3,3]).squeeze())
                t_list.append(np.reshape(t,[3,1]).squeeze())
        if radial_distort:
                return K,R_list,t_list,[k1,k2,x_0,y_0]
        else:
                return K,R_list,t_list
def calculate_radial_distortion(corner_points,k1,k2,x_0,y_0):
        r = (corner_points[:,0]-x_0)**2+(corner_points[:,1]-y_0)**2
        x_rd = corner_points[:,0] + (corner_points[:,0]-x_0)*(k1*r**2+k2*r**4)
        y_rd = corner_points[:,1] + (corner_points[:,1]-y_0)*(k1*r**2+k2*r**4)
        new_corners = np.concatenate((np.reshape(x_rd,[-1,1]),np.reshape(y_rd,[-1,1])),axis=1)
        return new_corners
def cost_function(parameters,world_corners_list,intersection_list,radial_distort):
        if radial_distort:
                K,R_list,t_list = reconstruct_parameters(parameters[:-2])
                k1 = parameters[-2]
                k2 = parameters[-1]
                x_0 = parameters[-5]
                y_0 = parameters[-3]
        else:
                K,R_list,t_list = reconstruct_parameters(parameters)
        projected_corners_list = list()
        num_sets = len(R_list)
        loss_metric = list()
        for ix in range(0,num_sets):
                R,t = R_list[ix],t_list[ix]
                Rt = np.concatenate((R[:,0:2],np.reshape(t,[3,1])),axis=1)
```

```
                H = np.matmul(K,Rt)
                projected_corners = transform_image(H, world_corners_list[-1])
                if radial_distort:
                        projected_corners = calculate_radial_distortion(projected_corners,k1,k2,
                loss_metric = loss_metric + np.subtract(projected_corners, intersection_list[ix])
        loss_metric = np.array(loss_metric)
        return loss_metric
def optimize_parameters(parameters, world_corners_list, intersection_list, radial_distort=False):
        if radial_distort:
                parameters = np.append(parameters,np.array([0,0]))
        loss = cost_function(parameters, world_corners_list, intersection_list, radial_distort)
        lm_refined_params = optimize.least_squares(cost_function, parameters, args=[world_corners_
        return lm_refined_params
def reprojection_to_fixed_image(R_list_refined, t_list_refined, K_refined, world_corners_list, inters
        projected_corners_list = list()
        H_list = list()
        num_sets = len(R_list_refined)
        Error_list = list()
        for ix in range(0,num_sets):
                R,t = R_list_refined[ix], t_list_refined[ix]
                Rt = np.concatenate((R[:,0:2],np.reshape(t,[3,1])),axis=1)
                H = np.matmul(K_refined,Rt)
                H_list.append(H)
        for ix in range(0,num_sets-1):
                H = H_list[ix]
                H_fixed_im = np.matmul(H_list[-1],np.linalg.inv(H))
                projected_corners = transform_image(H_fixed_im, intersection_list[ix])
                if rd_parameters:
                        projected_corners = calculate_radial_distortion(projected_corners,rd_para
                projected_corners_list.append(projected_corners)
                error = np.subtract(projected_corners, intersection_list[-1]).flatten()

                error = np.reshape(error,[-1,2])
                error_norm = np.linalg.norm(error, axis=1)
                error_mean = np.mean(error_norm)
                error_var = np.var(error_norm)
                error_max = np.max(error_norm)
                error_min = np.min(error_norm)
                error_metrics = [error_mean, error_var, error_max, error_min]
                Error_list.append(np.array(error_metrics))
        return Error_list, projected_corners_list
def reprojection_error_calculate(R_list_refined, t_list_refined, K_refined, world_corners_list, inte
        projected_corners_list = list()
        H_list = list()
        num_sets = len(R_list_refined)
        Error_list = list()
        for ix in range(0,num_sets):
                R,t = R_list_refined[ix], t_list_refined[ix]
                Rt = np.concatenate((R[:,0:2],np.reshape(t,[3,1])),axis=1)
                H = np.matmul(K_refined,Rt)
                H_list.append(H)
        for ix in range(0,num_sets-1):
                H = H_list[ix]
                projected_corners = transform_image(H, world_corners_list[-1])
                if rd_parameters:
                        projected_corners = calculate_radial_distortion(projected_corners,rd_para
                projected_corners_list.append(projected_corners)
                error = np.subtract(projected_corners, intersection_list[ix]).flatten()

                error = np.reshape(error,[-1,2])
```

```
                error_norm = np.linalg.norm(error,axis=1)
                error_mean = np.mean(error_norm)
                error_var = np.var(error_norm)
                error_max = np.max(error_norm)
                error_min = np.min(error_norm)
                error_metrics = [error_mean,error_var,error_max,error_min]
                Error_list.append(np.array(error_metrics))
        return Error_list,projected_corners_list
def plot_reprojected_corners(projected_corners_list,intersection_list,fnames,directotry,name):
        for imx in range(0,len(fnames)-1):
                save_path = './'+directotry+'/'+name+'/'+name+'_'+fnames[imx]
                im = cv2.imread('../'+directotry+'/'+fnames[imx])
                if directotry.upper() == 'DATASET2':
                        im =  cv2.resize(im, (480,640), interpolation = cv2.INTER_AREA)

                intersections = intersection_list[imx].astype(int)
                projected_corners = projected_corners_list[imx].astype(int)
                #projected_corners[:,0] = projected_corners[:,0]/np.max(projected_corners[:,0])*6
                #projected_corners[:,1] = projected_corners[:,1]/np.max(projected_corners[:,1])*4
                for ix in range(0,len(intersections)):
                        im = cv2.circle(im,tuple(intersections[ix,:]),radius=4,color=(0, 0, 255)
                        im = cv2.circle(im,tuple(projected_corners[ix,:]),radius=2,color=(0, 255
                cv2.imwrite(save_path, im)
def plot_reprojected_corners_fi(projected_corners_list,intersection_list,fnames,directotry,name)
        for imx in range(0,len(fnames)-1):
                save_path = './'+directotry+'/'+name+'/'+name+'_'+fnames[imx]
                im = cv2.imread('../'+directotry+'/'+fnames[-1])
                if directotry.upper() == 'DATASET2':
                        im =  cv2.resize(im, (480,640), interpolation = cv2.INTER_AREA)

                intersections = intersection_list[-1].astype(int)
                projected_corners = projected_corners_list[imx].astype(int)
                #projected_corners[:,0] = (projected_corners[:,0]/np.max(projected_corners[:,0])*
                #projected_corners[:,1] = projected_corners[:,1]/np.max(projected_corners[:,1])*4
                for ix in range(0,len(intersections)):
                        im = cv2.circle(im,tuple(intersections[ix,:]),radius=4,color=(0, 0, 255)
                        im = cv2.circle(im,tuple(projected_corners[ix,:]),radius=2,color=(0, 255
                cv2.imwrite(save_path, im)
def plot_reprojection_errors(Error_list,directotry,name):
        Means = list()
        Variances = list()
        Max = list()
        Min = list()

        for error_metrics in Error_list:
                Means.append(error_metrics[0])
                Variances.append(error_metrics[1])
                Max.append(error_metrics[2])
                Min.append(error_metrics[3])
        image_ids = np.arange(0,len(Means))

        plt.figure()
        plt.scatter(image_ids,Means)
        plt.xlabel('Image ID')
        plt.ylabel('Mean Reprojection L2 Norm Error')
        plt.savefig('./'+directotry+'/'+name+'/Mean_Error.png')
        plt.close()

        plt.figure()
        plt.scatter(image_ids,Variances)
```

```
        plt.xlabel('Image ID')
        plt.ylabel('Variance of Reprojection L2 Norm Error')
        plt.savefig('./'+directotry+'/'+name+'/Variances_Error.png')
        plt.close()

        plt.figure()
        plt.scatter(image_ids,Max)
        plt.xlabel('Image ID')
        plt.ylabel('Maximum Reprojection L2 Norm Error')
        plt.savefig('./'+directotry+'/'+name+'/Max_Error.png')
        plt.close()

        plt.figure()
        plt.scatter(image_ids,Min)
        plt.xlabel('Image ID')
        plt.ylabel('Minimum Reprojection L2 Norm Error')
        plt.savefig('./'+directotry+'/'+name+'/Min_Error.png')
        plt.close()

def main():
        #detect the edges of the images and store them
        directotry = 'Dataset1'
        hough_transformation_threshold = 50
        corner_detecctcion = 'auto'#options='manual','auto'

        world_corners = build_pattern(10, 4, 5)
        world_corners = np.array(world_corners).squeeze().astype(float)
        H_matrices = list()
        intersection_list = list()
        world_corners_list = list()

        fnames = os.listdir('../'+directotry)
        if not os.path.exists('./'+directotry):
                os.mkdir('./'+directotry)
        if not os.path.exists('./'+directotry+'/Edges'):
                os.mkdir('./'+directotry+'/Edges')
        if not os.path.exists('./'+directotry+'/Hough_Lines'):
                os.mkdir('./'+directotry+'/Hough_Lines')
        if not os.path.exists('./'+directotry+'/Intersections'):
                os.mkdir('./'+directotry+'/Intersections')
        if not os.path.exists('./'+directotry+'/Corner_Labels'):
                os.mkdir('./'+directotry+'/Corner_Labels')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners'):
                os.mkdir('./'+directotry+'/Reprojected_Corners')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners_fi'):
                os.mkdir('./'+directotry+'/Reprojected_Corners_fi')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners_lm'):
                os.mkdir('./'+directotry+'/Reprojected_Corners_lm')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners_lm_fi'):
                os.mkdir('./'+directotry+'/Reprojected_Corners_lm_fi')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners_lm_rd'):
                os.mkdir('./'+directotry+'/Reprojected_Corners_lm_rd')
        if not os.path.exists('./'+directotry+'/Reprojected_Corners_lm_fi_rd'):
                os.mkdir('./'+directotry+'/Reprojected_Corners_lm_fi_rd')
        if not os.path.exists('./'+directotry+'/Auto_Corners'):
                os.mkdir('./'+directotry+'/Auto_Corners')

        #relocate the fixed image at the end
        fnames_relocate = list()
        for im_count in range(0,len(fnames)):
```

```
        if fnames[im_count] != 'Fixed_Image.jpg':
                fnames_relocate.append(fnames[im_count])
fnames_relocate.append('Fixed_Image.jpg')
fnames = fnames_relocate

for im_count in range(0,len(fnames)):
        print('Image ',str(im_count+1),'  ',fnames[im_count])
        fname = '../'+directotry+'/'+fnames[im_count]
        original_im = cv2.imread(fname)
        if directotry.upper()=='DATASET2':
                original_im =  cv2.resize(original_im, (480,640), interpolation = cv2.INT
        if len(original_im.shape)>2:
                im = cv2.cvtColor(original_im,cv2.COLOR_BGR2GRAY)
        else:
                im = original_im

        [height,width] = im.shape
        if corner_detecctcion.upper()=='MANUAL':
                save_path = './'+directotry+'/Edges//Edges_'+fnames[im_count]
                edges = edge_detection(im,save_path)
                save_path = './'+directotry+'/Hough_Lines/Hough_Lines_'+fnames[im_count]
                lines = hough_transformation(im,edges,hough_transformation_threshold,save
                LH = list()
                for L in lines:
                        x1,y1,x2,y2 =  L[0]
                        LH.append(form_line([x1,y1],[x2,y2]))
                Intersections = list()
                for ix in range(0,len(LH)):
                        for jx in range(0,len(LH)):
                                if jx != ix:
                                        Intersections.append(form_intersection(LH[ix],LH
                #refine intersections
                Intersections = np.array(Intersections).squeeze()
                Intersections = np.reshape(Intersections,[len(Intersections),3])
                Intersections_refined = list()
                for ix in range(0,Intersections.shape[0]):
                        if not (Intersections[ix,0] < 0 or Intersections[ix,0] > width or
                                Intersections_refined.append(np.array(Intersections[ix,:
                #Further refinement
                Intersections_refined = unique_corner_detection(im,Intersections_refined
                print('   #Intersection Points = ',len(Intersections_refined),end=' ')
                save_path = './'+directotry+'/Intersections/Manual_Intersections_'+fnames
                plot_intersections(original_im,edges,lines,Intersections_refined,save_pat
                intersections_sorted = label_corners(Intersections_refined)
                save_path = './'+directotry+'/Corner_Labels/Manual_Corner_Labels_'+fnames
                plot_corner_labels(original_im,intersections_sorted.tolist(),save_path)
        elif corner_detecctcion.upper() == 'AUTO':
                #Harris corners
                im = np.float32(im)
                dst = cv2.cornerHarris(im,10,3,0.075)
                dst = cv2.dilate(dst,None)
                ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
                dst = np.uint8(dst)
                #centroids
                ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)
                # define the criteria to stop and refine the corners
                criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.0
                Intersections_refined = cv2.cornerSubPix(im,np.float32(centroids),(5,5),(
                Intersections_refined = Intersections_refined[1:,:].astype(int)
                intersections_sorted = label_corners(Intersections_refined)
```

```
                save_path = './'+directotry+'/Corner_Labels/Auto_Corner_Labels_'+fnames[i
                plot_corner_labels(original_im,intersections_sorted.tolist(),save_path)

        print('   #Intersection Points = ',len(intersections_sorted))
        #Estimate correspondence between world coordinates and image plane coordinates
        if intersections_sorted.shape[0] == 80:
                num_points = np.min([world_corners.shape[0],intersections_sorted.shape[0
                H_matrices.append(Homography_Matrix(world_corners[:num_points,:],intersec
                intersection_list.append(intersections_sorted[:num_points,:])
                world_corners_list.append(world_corners[:num_points,:])

omega = compute_omega(H_matrices)
print('Omega = \n',omega)
K = compute_K(omega)
print('K = \n',K)
R_list,t_list = compute_extrinsic_parameters(H_matrices,K)
for ix in range(0,len(R_list)):
        print('\n\nImage name = ',fnames[ix])
        print('R = \n',R_list[ix])
        print('t = \n',t_list[ix])

Error_list,projected_corners_list = reprojection_error_calculate(R_list,t_list,K,world_c
plot_reprojected_corners(projected_corners_list,intersection_list,fnames,directotry,'Rep
plot_reprojection_errors(Error_list,directotry,'Reprojected_Corners')
Error_list_fi,projected_corners_list_fi = reprojection_to_fixed_image(R_list,t_list,K,wo
plot_reprojected_corners_fi(projected_corners_list_fi,intersection_list,fnames,directotry
plot_reprojection_errors(Error_list_fi,directotry,'Reprojected_Corners_fi')

########################## LM refinement ##############################################
parameters = form_parametric_representation(R_list,t_list,K)
#The last 5 params of 'parameters' are intrinsic parameters. The set of each consecutive
#parameters and 3 translation vector parameters respectively. The set of 6 is for a sing
lm_refined_parameters = optimize_parameters(parameters,world_corners_list,intersection_li
np.savez('lm_refined_parameters_'+directotry+'.npz',lm_refined_parameters=lm_refined_para
#Refine the parameter list using LM technique
K_refined,R_list_refined,t_list_refined = reconstruct_parameters(lm_refined_parameters.x)
print('K_lm = \n',K_refined)
for ix in range(0,len(R_list_refined)):
        print('\n\nImage name = ',fnames[ix])
        print('R_lm = \n',R_list_refined[ix])
        print('t_lm = \n',t_list_refined[ix])
Error_list,projected_corners_list = reprojection_error_calculate(R_list_refined,t_list_re
plot_reprojected_corners(projected_corners_list,intersection_list,fnames,directotry,'Rep
plot_reprojection_errors(Error_list,directotry,'Reprojected_Corners_lm')
Error_list_fi,projected_corners_list_fi = reprojection_to_fixed_image(R_list_refined,t_li
plot_reprojected_corners_fi(projected_corners_list_fi,intersection_list,fnames,directotry
plot_reprojection_errors(Error_list_fi,directotry,'Reprojected_Corners_lm_fi')

########################## LM refinement with fixing radial distortion ################
parameters = form_parametric_representation(R_list_refined,t_list_refined,K_refined)
lm_refined_parameters_rd = optimize_parameters(parameters,world_corners_list,intersectio
np.savez('lm_refined_parameters_rd_'+directotry+'.npz',lm_refined_parameters_rd=lm_refine
K_refined_rd,R_list_refined_rd,t_list_refined_rd,rd_parameters = reconstruct_parameters(
print('K_lm_rd = \n',K_refined_rd)
for ix in range(0,len(R_list_refined_rd)):
        print('\n\nImage name = ',fnames[ix])
        print('R_lm_rd = \n',R_list_refined_rd[ix])
        print('t_lm_rd = \n',t_list_refined_rd[ix])
print(rd_parameters)
Error_list_rd,projected_corners_list_rd = reprojection_error_calculate(R_list_refined_rd
```

```
        plot_reprojected_corners(projected_corners_list_rd,intersection_list,fnames,directotry,'
        plot_reprojection_errors(Error_list_rd,directotry,'Reprojected_Corners_lm_rd')
        Error_list_fi_rd,projected_corners_list_fi_rd = reprojection_to_fixed_image(R_list_refine
        plot_reprojected_corners_fi(projected_corners_list_fi_rd,intersection_list,fnames,directo
        plot_reprojection_errors(Error_list_fi_rd,directotry,'Reprojected_Corners_lm_fi_rd')

main()
```