Theory Question

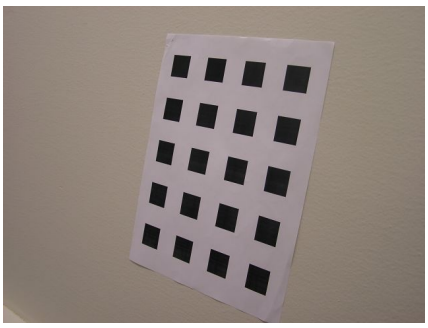
In Lecture 20, we showed that the image of the Absolute Conic Ω_∞ is given by $\omega = K^{-T}K^{-1}$. As you know, the Absolute Conic resides in the plane Π_∞ at infinity. Does the derivation we went through in Lecture 20 mean that you can actually see ω in a camera image? Give reasons for both 'yes' and 'no' answers. Also, explain in your own words the role played by this result in camera calibration.

The Absolute Conic Ω_∞ given by ω is an imaginary point conic with no real points. It resides in the plane at infinity Π_∞. It cannot be seen, but only inferred. As a mathematical device, it can be used to derive the intrinsic parameters of the camera with the relation: $\omega = K^{-T}K^{-1}$ which can then be used to estimate extrinsic parameters.

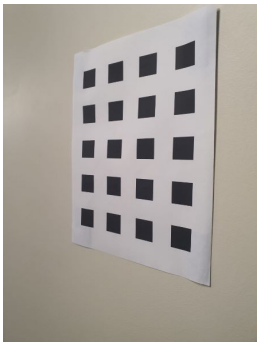
1. Programming Task

The goal in this homework is to implement the popular Zhang's algorithm for camera calibration on a pin-hole camera. A complete calibration procedure will involve estimating all the 5 intrinsic parameters and the 6 extrinsic parameters that determine the position and orientation of the camera with respect to a reference world coordinate system.

Correspondences between image points and their world coordinates are established by using the checkerboard pattern consisting of alternating black and white squares shown below. Two dataset are used for this assignment. Dataset 1 contains 40 images and dataset 2 contains 21 images.



(a) Example from dataset 1.jpg



(b) Example from dataset 2.jpg

Figure 1: Calibration Patterns

1.1 Background and Theory

Brief Summary of the process:

The process is divided into 4 main parts:

- Corner Detection and Homography Calculations:
    1. The edges are extracted from the gray-scale versions of the images using Canny Edge Detector. OpenCV's built-in cv2.Canny module is used. For dataset 2, the gray-scale versions of the images were blurred with cv2.GaussianBlur before using Canny to detect edges.
    2. Using the edges, Hough transform was applied with OpenCV's cv2.HoughLines module to fit straight lines. The parameters for HoughLines were adjusted to give better results.
    3. The cv2.HoughLines fits multiple lines on one possible edge and therefore the lines are refined to pick the best fitting line. This process is detailed further below.
    4. Using the refined lines, the intersections between these lines are computed which gives the coordinates of the corners of the black squares on the calibration pattern.
    5. Next, the images with 80 corners are used to compute homographies that relate the world corner coordinates with the above computed image corner coordinates. The world corner coordinates are a fixed grid of coordinates that assume the distance between each corner is 10 units: (0,0)..(70,0),(0,10)...(70,10)...(70,90)
- Calculate Intrinsic and Extrinsic Parameters:
    1. Using all the homographies, the absolute conic ω is computed with linear least squares. This process is detailed further below.
    2. Once the ω is calculated, the intrinsic parameter matrix K is computed using the relation $\omega = K^{-T}K^{-1}$
    3. Using K and the homographies calculated earlier, the rotation matrices and translation vectors are calculated.
    4. Using these calculated parameters, homographies are reconstructed and applied on the world coordinates to project them onto the images. The distance between the projected corners and the corners computed from the previous step was calculated and used as a metric for performance. Since the parameters are not refined yet, the projection accuracy is not great. The parameters computed so far are only estimates. Further refining is needed to get better projection.
- Refining the Calibration Parameters with LM:
    1. The estimated parameters are refined using Levenberg-Marquadt (LM) non-linear optimization. The parameters are refined by reducing the cost function.
    2. `scipy.optimize.least_squares` module is used to perform LM. Further details of this process is given below.

- Reproject corners onto fixed image using the refined parameters

    1. With the refined parameters, the new homographies are calculated and used to project corners of images onto a "fixed" image. The fixed image is chosen so that its corners are similar to world corner coordinates.
    2. The corners are plotted on the image and their accuracies are calculated.

Refining Lines: The output from the cv2.HoughLines module gives multiple lines that need to be refined to get one line per set of edges. This is done by non-maximum suppression process. The lines were first separated into two groups: horizontal and vertical. Next, for each of the two groups the lines were sorted based on the distance from the origin. If the distance between two consecutive lines was more than a specific threshold (calculated from the ratio of average distance between the lines), the first line was selected to be the "refined" line for a set of edges. The refined lines were then used to find the intersections by calculating the cross product of the homogeneous coordinates of the lines.

The low and high threshold for Canny detector were set 255 and 1.5*255 respectively. The parameters for HoughLines were varied to get optimal results. The resolution of the parameter r is set to 1 pixel. The resolution of the parameter θ in radians is set to 0.5 radians. For dataset 1 the minimum number of intersections to detect a line threshold is set to 60, and for dataset 2 it is set to 50.

Computing ω: The world corners lie on the plan $z = 0$. The image of a world corner $\vec{x}$ is given by:

$$\vec{x}\,' = H\vec{x} = K\left[R|\vec{t}\right]\vec{x}$$

This $H = [\vec{h_1}, \vec{h_2}, \vec{h_3}]$ also relates the two Circular points on world plane $z = 0$ with their images on the image plane with the following equations: $\vec{I}\,' = H\vec{I}$ and $\vec{J}\,' = H\vec{J}$. Since $\vec{I} = [1, i, 0]^T$ and $\vec{J} = [1, -i, 0]^T$, this gives $H\vec{I} = \vec{h}_1 + i\vec{h}_2$ and $H\vec{J} = \vec{h}_1 - i\vec{h}_2$. Since Circular Points lie on the Absolute Conic ω, $(\vec{I}\,')^T\omega(\vec{I}\,') = 0$ and $(\vec{J}\,')^T\omega(\vec{J}\,') = 0$. This gives the following relation :

$$(\vec{h}_1 + i\vec{h}_2)^T\omega(\vec{h}_1 + i\vec{h}_2) = 0$$
$$(\vec{h}_1 - i\vec{h}_2)^T\omega(\vec{h}_1 - i\vec{h}_2) = 0$$

Simplifying the above equations, we get:

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

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

Simplifying further, as Zhang's algorithm shows, the above equations can be formed into:

$$V\vec{b} = \begin{bmatrix} \vec{v}_{1,2}^T \\ \vec{v}_{1,1}^T - \vec{v}_{2,2}^T \end{bmatrix}\vec{b} = 0$$

$$\vec{v}_{i,j} = \begin{bmatrix} \vec{h}_{i1}\vec{h}_{j1} \\ \vec{h}_{i1}\vec{h}_{j2} + \vec{h}_{i2}\vec{h}_{j1} \\ \vec{h}_{i2}\vec{h}_{j2} \\ \vec{h}_{i3}\vec{h}_{j1} + \vec{h}_{i1}\vec{h}_{j3} \\ \vec{h}_{i3}\vec{h}_{j2} + \vec{h}_{i2}\vec{h}_{j3} \\ \vec{h}_{i3}\vec{h}_{j3} \end{bmatrix} \quad \vec{b} = \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}$$

Each H gives two equations and with the H from each image stacked together in V. Vector $\vec{b}$ is defined as the smallest singular values of the computed SVD of V. Rearranging $\vec{b}$ gives ω.

Computing Intrinsic and Extrinsic Parameters With the computed ω, the intrinsic parameter K matrix is computed by the following relation: $\omega = K^{-T}K^{-1}$, where

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

With Zhang's algorithm implementation, the parameters in the K matrix can be computed by the following equations:

$$y_0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2}$$

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

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

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

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

$$x_0 = \frac{sy_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

Since the homography for each image (or camera position) can also be defined as $H = K[R|\vec{t}]$, the matrix $[R|\vec{t}]$ for that position of the camera can be calculated by the following relation:

$$K^{-1}[\vec{h_1}, \vec{h_2}, \vec{h_3}] = [\vec{r_1}, \vec{r_1}, \vec{t}]$$

To make all the columns of the R matrix be of unit magnitude, the scale factor is computed as $\xi = \frac{1}{\|K^{-1}\vec{h_1}\|}$. Therefore the $[R|\vec{t}]$ matrix is computed as:

$$\vec{r_1} = \xi K^{-1}\vec{h_1}$$

$$\vec{r_2} = \xi K^{-1}\vec{h_2}$$

$$\vec{r_3} = \vec{r_1} \times \vec{r_2}$$
$$\vec{t} = \xi K^{-1} \vec{h_3}$$

To represent R with 3 DoF, R is represented as $\vec{w}$:

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

where

$$\varphi = Cos^{-1} \frac{trace(R) - 1}{2}$$

To construct R from a given $\vec{w}$:

$$R = e^{[\vec{w}]_x} = I_{3\times 3} + \frac{Sin\varphi}{\varphi}[\vec{w}]_x + \frac{1 - Cos\varphi}{\varphi^2}[\vec{w}]_x^2, \quad \varphi = \|\vec{w}\|$$

This allows R to be orthonormal. The parameters computed so far are the initial guesses. They are used to reconstruct homographies and the the world corners are reprojected onto the images and compared with the corners found before.

Refining parameters: Once the intrinsic and extrinsic parameters K and $[R|\vec{t}]$ have been computed they need refining to increase accuracy. To refine them, non-linear least squares method is used. The Levenberg-Marquadt (LM) method is used. This process refines parameter by reducing the cost computed by a cost function. For our purpose, the cost that needs to be reduced is the distance between the projected corners $\hat{\vec{x}}_{i,j}$ and the corners found by intersecting lines $\vec{x}_{i,j}$. This is defined below:

$$d_{geom}^2 = \|\vec{X} - \vec{f}(\vec{p})\|^2 = \sum_i \sum_j \|\vec{x}_{i,j} - \hat{\vec{x}}_{i,j}\|$$

where

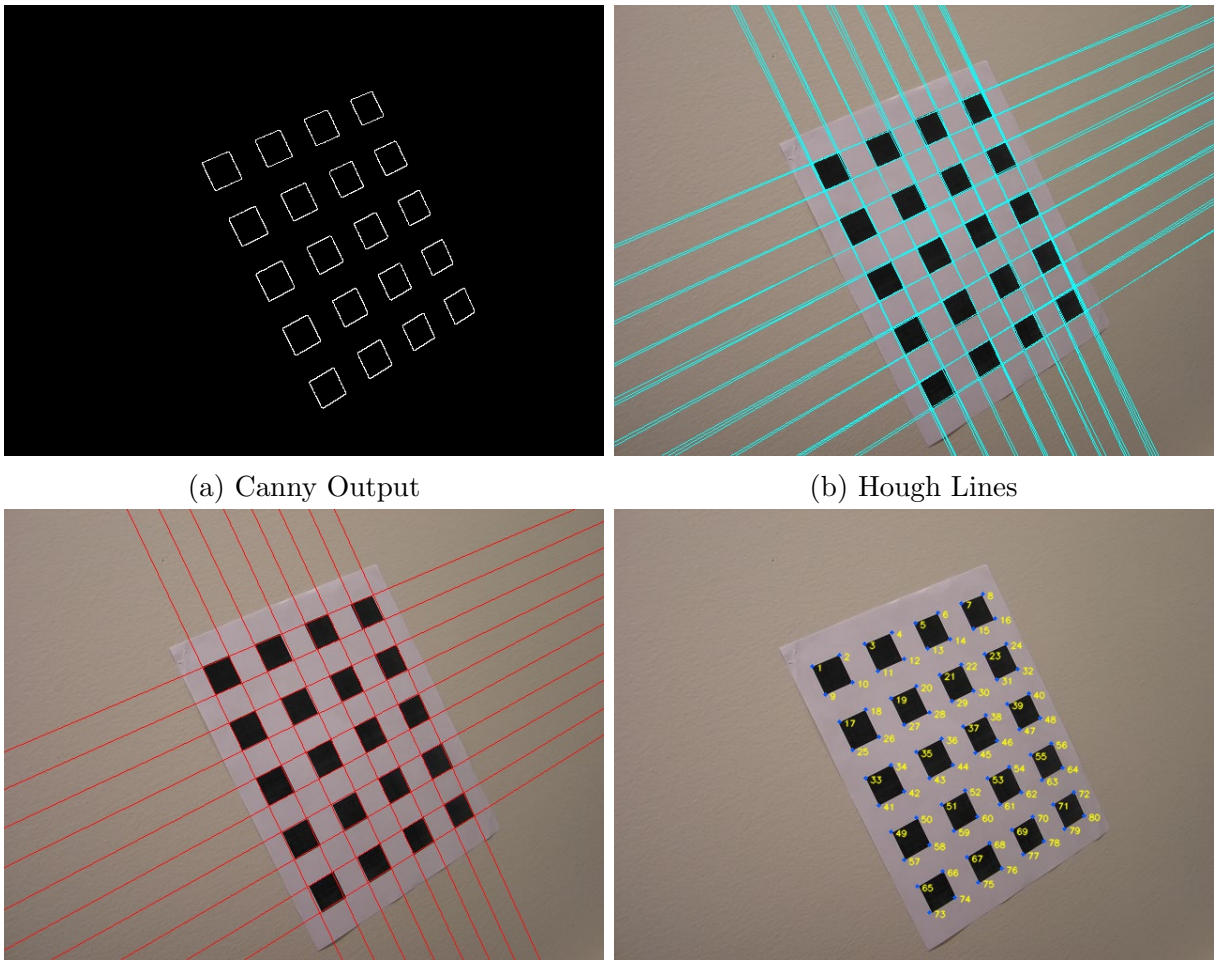
$$\hat{\vec{x}}_{i,j} = K[R_i|\vec{t}_i]\vec{x}_{m,j}$$

1.2 Results and Outputs

1.2.1 Corner Detection Results

The Canny, Hough lines, refined lines and the intersections for two images are shown below.

Dataset 1:



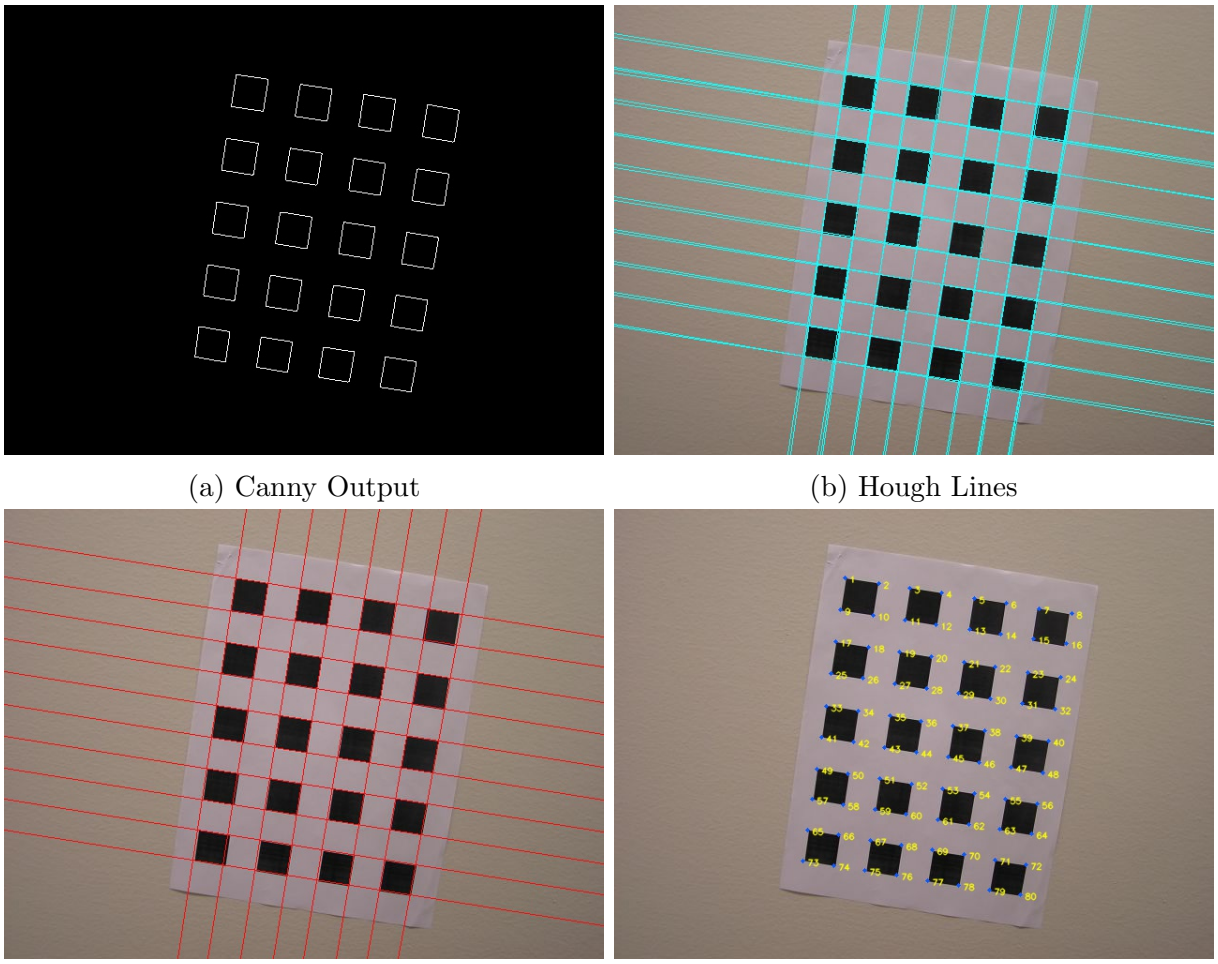
(a) Canny Output

(b) Hough Lines

(c) Refined Lines

(d) Intersections

Figure 2: Canny, Hough Lines, Refined Lines and Intersection Outputs for Pic_04



(a) Canny Output

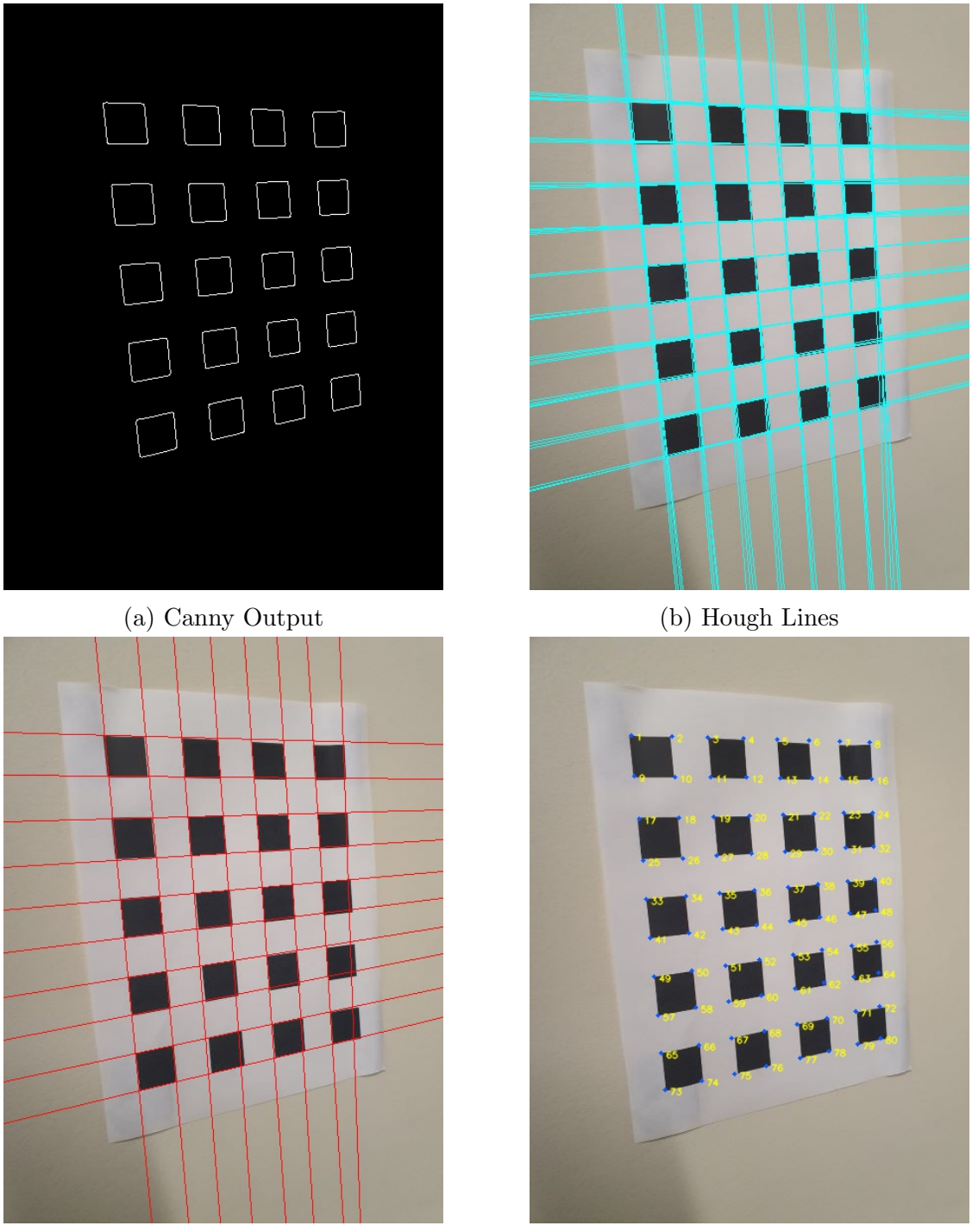
(b) Hough Lines

(c) Refined Lines

(d) Intersections

Figure 3: Canny, Hough Lines, Refined Lines and Intersection Outputs for Pic_05

Dataset 2:



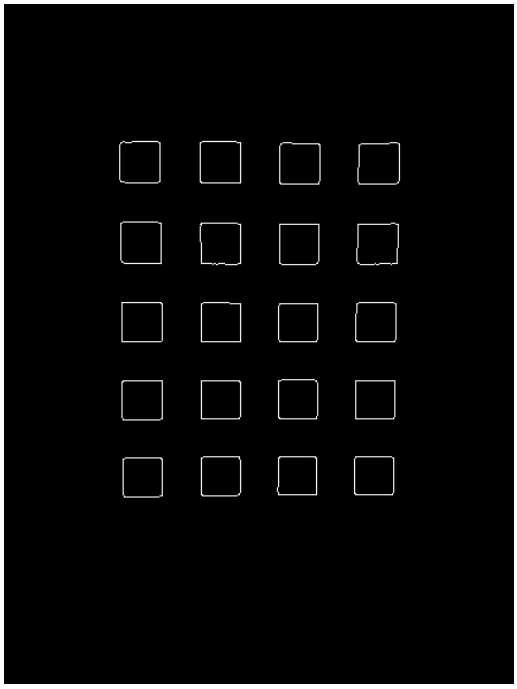
(a) Canny Output

(b) Hough Lines

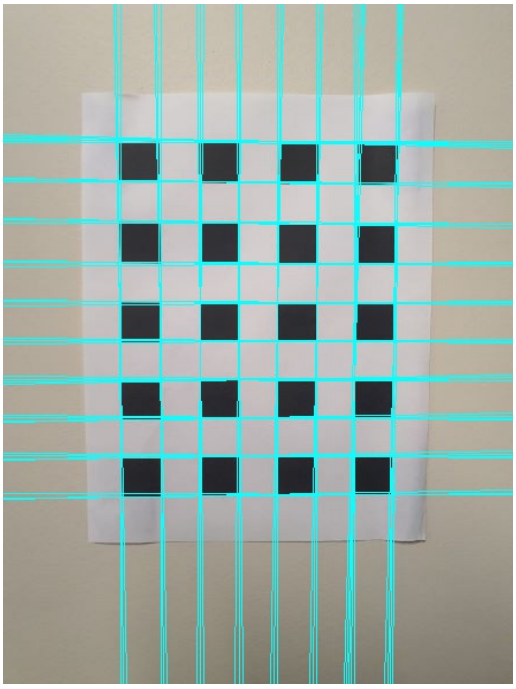
(c) Refined Lines

(d) Intersections

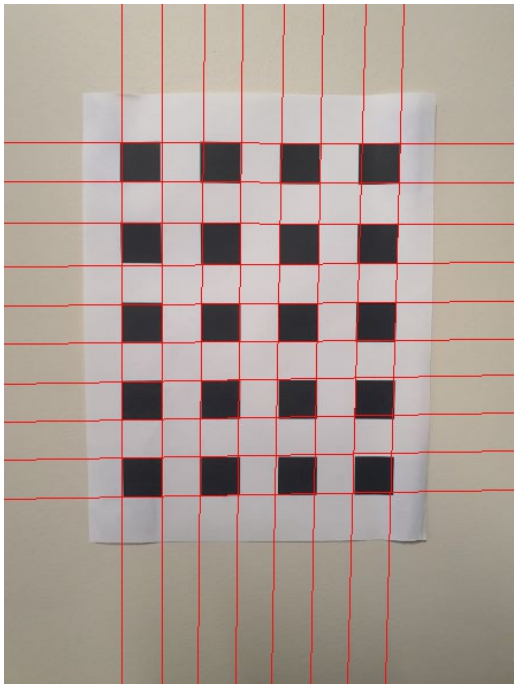
Figure 4: Canny, Hough Lines, Refined Lines and Intersection Outputs for Pic_02



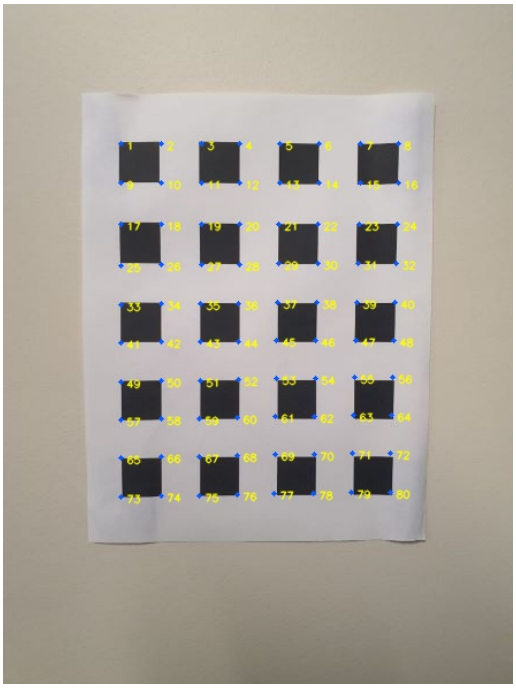
(a) Canny Output



(b) Hough Lines



(c) Refined Lines



(d) Intersections

Figure 5: Canny, Hough Lines, Refined Lines and Intersection Outputs for Pic_10

1.2.2 Initial and Refined Parameter:

Dataset 1:
Initial Parameters:

$$K = \begin{bmatrix} 715.777 & -7.9369 & 324.3792 \\ 0 & 712.3227 & 239.1015 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[R_4|\vec{t_4}] = \left[\begin{array}{ccc|c} 0.8211 & 0.4292 & -0.3762 & -34.4081 \\ -0.4184 & 0.9010 & 0.1148 & -21.9788 \\ 0.3883 & 0.0631 & 0.9194 & 217.1464 \end{array}\right]$$

$$[R_5|\vec{t_5}] = \left[\begin{array}{ccc|c} 0.9882 & -0.1517 & 0.0235 & -22.7107 \\ 0.1531 & 0.9849 & -0.0812 & -47.9170 \\ -0.0108 & 0.0838 & 0.9964 & 205.5939 \end{array}\right]$$

Refined Parameters:

$$K = \begin{bmatrix} 718.0537 & -9.0665 & 327.3779 \\ 0 & 714.4062 & 242.3557 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[R_4|\vec{t_4}] = \left[\begin{array}{ccc|c} 0.8152 & 0.4322 & -0.3855 & -35.0687 \\ -0.4150 & 0.9002 & 0.1317 & -22.7576 \\ 0.4039 & 0.0526 & 0.9133 & 216.0865 \end{array}\right]$$

$$[R_5|\vec{t_5}] = \left[\begin{array}{ccc|c} 0.9881 & -0.1499 & 0.0344 & -23.5735 \\ 0.1523 & 0.9849 & -0.0817 & -48.7497 \\ -0.0216 & 0.0860 & 0.9961 & 206.7149 \end{array}\right]$$

Dataset 2:
Initial Parameters:

$$K = \begin{bmatrix} 468.5008 & -9.6200 & 211.3910 \\ 0 & 469.3135 & 297.6943 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[R_2|\vec{t_2}] = \left[\begin{array}{ccc|c} 0.9272 & 0.0930 & -0.3625 & -27.5795 \\ -0.1209 & 0.9911 & -0.0547 & -48.0864 \\ 0.3542 & 0.0946 & 0.9303 & 113.9867 \end{array}\right]$$

$$[R_{10}|\vec{t_{10}}] = \left[\begin{array}{ccc|c} 1.000 & 0.0009 & -0.0287 & -32.4019 \\ -0.0026 & 0.9982 & -0.0605 & -50.5342 \\ 0.0286 & 0.06116 & 0.9977 & 134.2268 \end{array}\right]$$

Refined Parameters:

$$K = \begin{bmatrix} 463.5047 & -10.1250 & 210.4049 \\ 0 & 464.3107 & 295.2226 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[R_2|\vec{t_2}] = \left[\begin{array}{ccc|c} 0.9265 & 0.0953 & -0.3640 & -27.2194 \\ -0.1191 & 0.9919 & -0.0434 & -47.3063 \\ 0.3569 & 0.0836 & 0.9304 & 121.6909 \end{array}\right]$$

$$[R_{10}|\vec{t_{10}}] = \left[\begin{array}{ccc|c} 0.9996 & 0.0098 & -0.0262 & -32.1557 \\ -0.0114 & 0.9981 & -0.0612 & -50.1097 \\ 0.0256 & 0.0615 & 0.9978 & 133.3660 \end{array}\right]$$

1.2.3 Projections of world corners onto images:

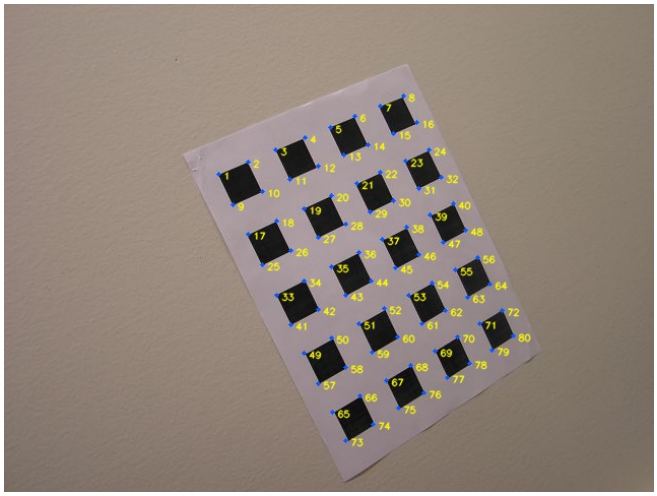
The accuracies of each dataset for a set of images includes average, variance and the maximum distance between the projected world corner and its corresponding corner found from corner detection process.

Dataset 1:

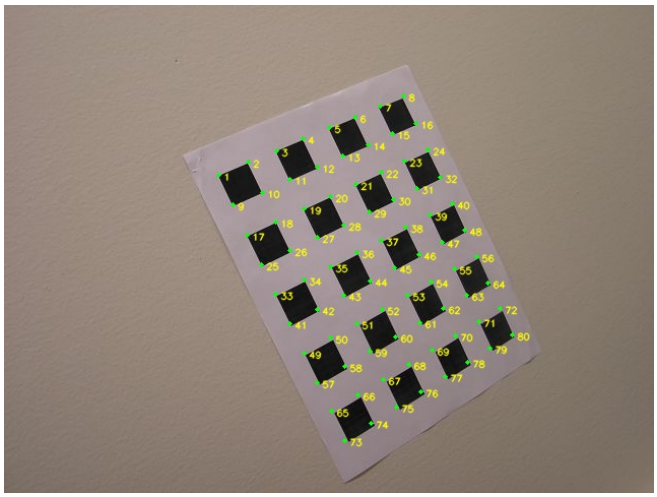
The accuracies for the projections of world corners onto the dataset 1 images are given below.

	Before LM		After LM	
	Image 4	Image 5	Image 4	Image 5
Mean	1.7038	0.7472	0.5814	0.5761
Variance	0.6817	0.1245	0.0948	0.1542
Max Distance	3.4069	2.1029	1.2006	2.0986

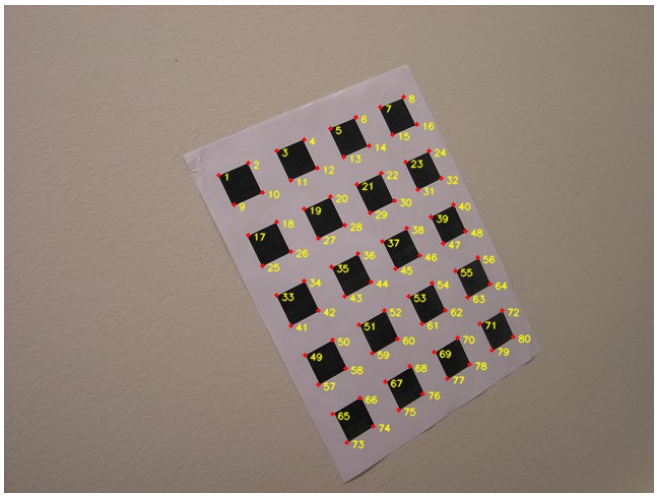
Table 1: Accuracy for Image 4 and Image 5 with initial and refined parameters



(a) Intersection Corners of Image 4

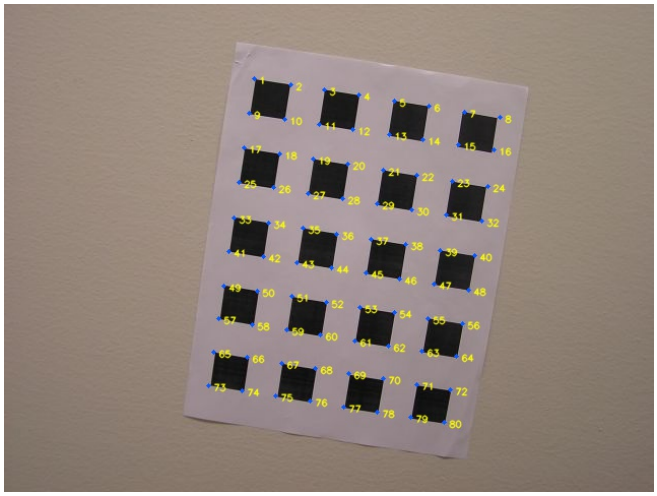


(b) Projection of world corners before LM

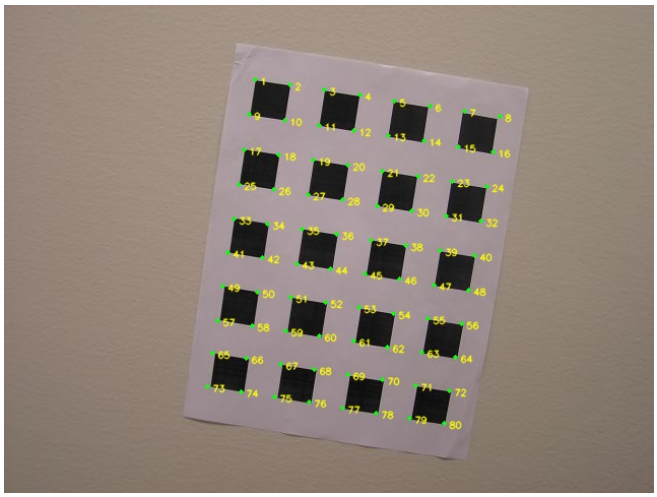


(c) Projection of world corners after LM

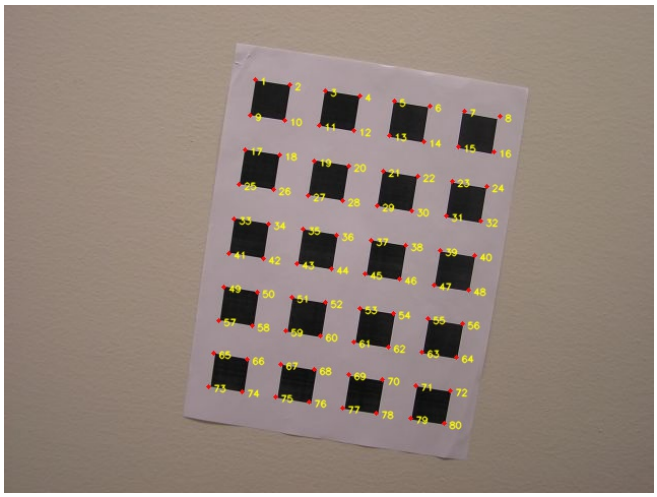
Figure 6: Projection of world corners with initial and refined parameters for image 4



(a) Intersection Corners of Image 5



(b) Projection of world corners before LM



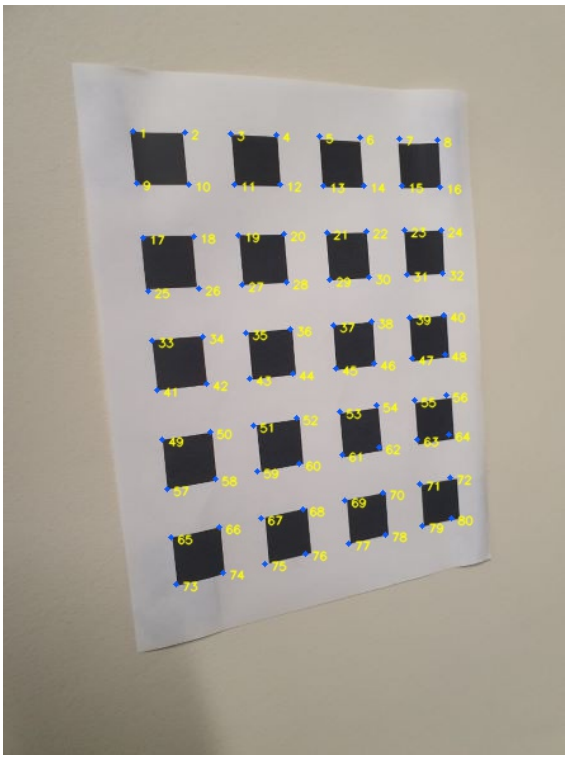
(c) Projection of world corners after LM

Figure 7: Projection of world corners with initial and refined parameters for image 5

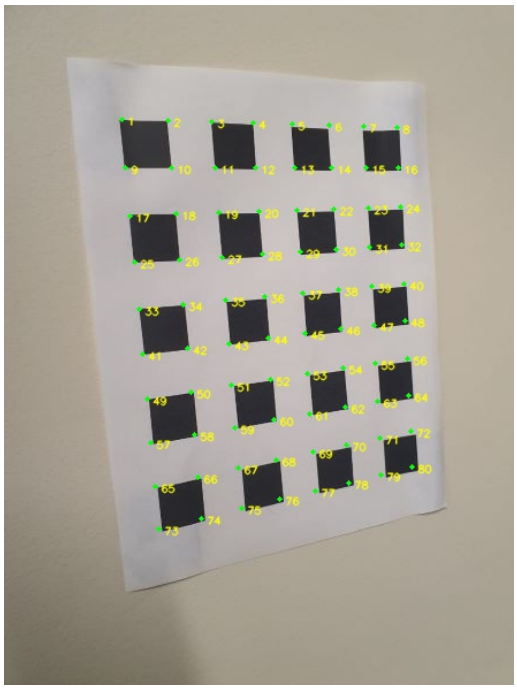
Dataset 2:
The accuracies for the projections of world corners onto the dataset 2 images are given below.

	Before LM		After LM	
	Image 2	Image 10	Image 2	Image 10
Mean	1.5361	1.2949	1.2892	0.8844
Variance	0.5944	0.5568	0.2423	0.0958
Max Distance	3.3775	2.6290	2.9489	1.4078

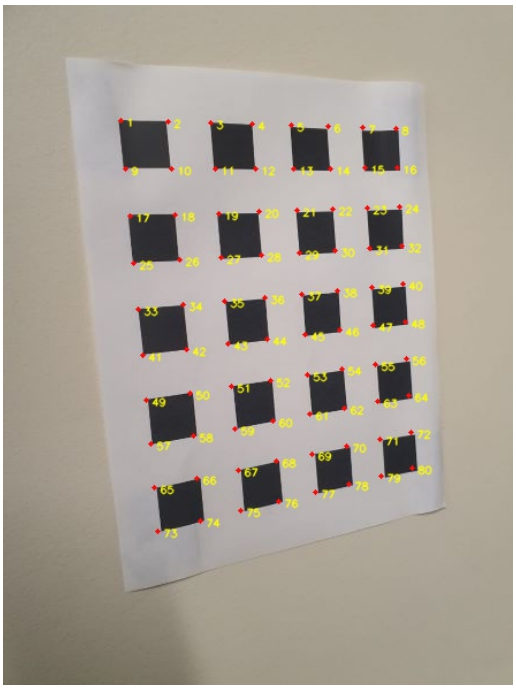
Table 2: Accuracy for Image 2 and Image 10 with initial and refined parameters



(a) Intersection corners of Image 2

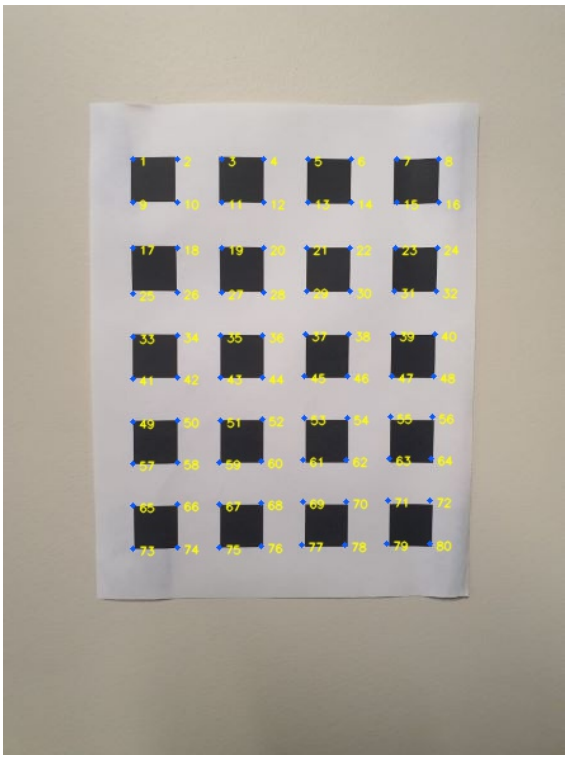


(b) Projection of world corners before LM

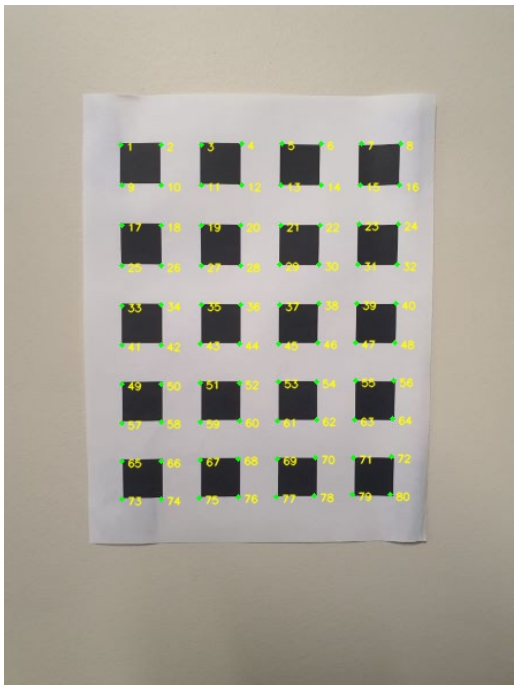


(c) Projection of world corners after LM

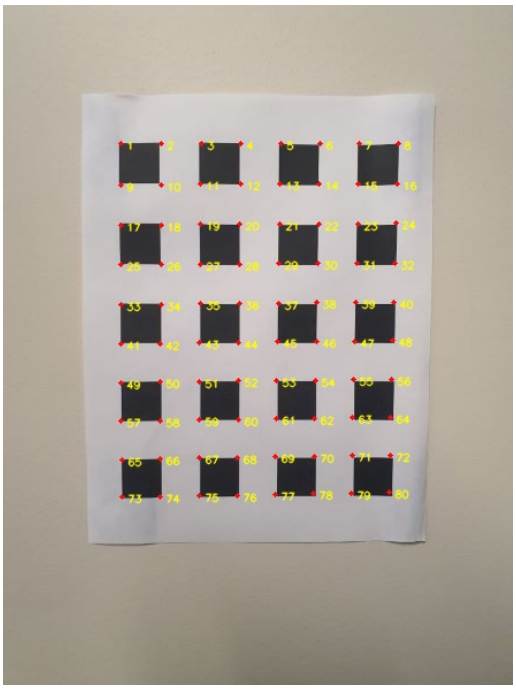
Figure 8: Projection of world corners with initial and refined parameters for image 2



(a) Intersection corners of Image 10



(b) Projection of world corners before LM



(c) Projection of world corners after LM

Figure 9: Projection of world corners with initial and refined parameters for image 10

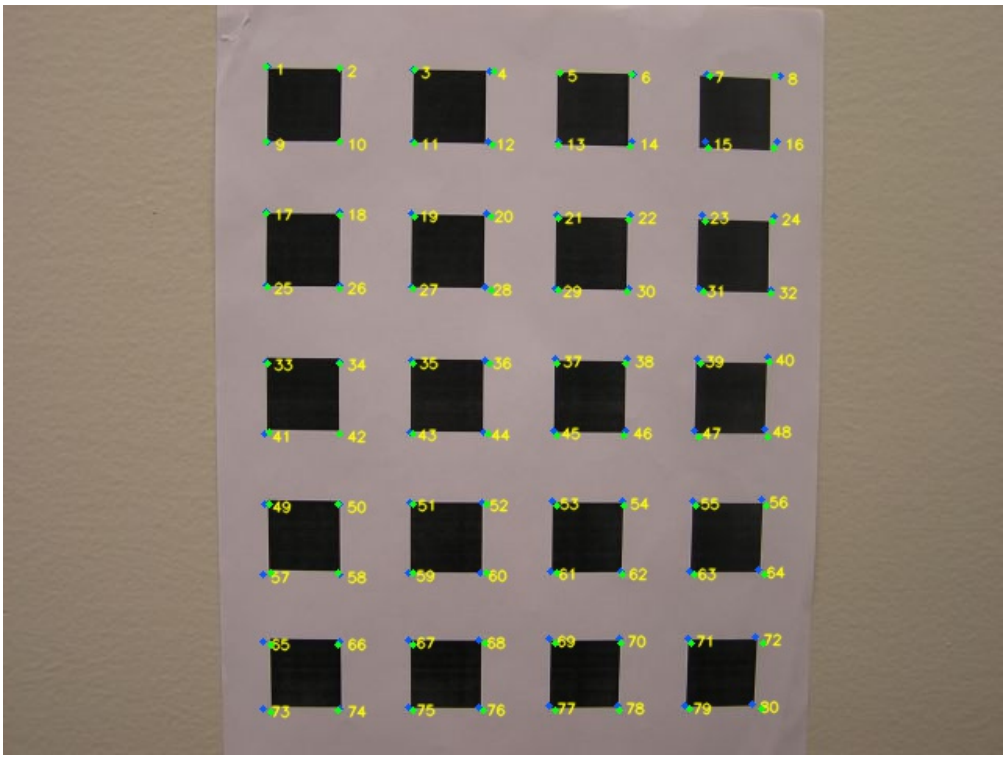
1.2.4 Re-projecting corners on Fixed Image:

To see how well the calibration parameters are refined, the corners of all the images are projected back onto a fixed image. The accuracy of the projection is calculated and presented for two images. For dataset 1, Image 28 was chosen as the fixed image. For dataset 2, image 10 was chosen as the fixed image. For both the dataset, re-projection of corners of Image 2 and Image 6 onto the fixed image is shown below.

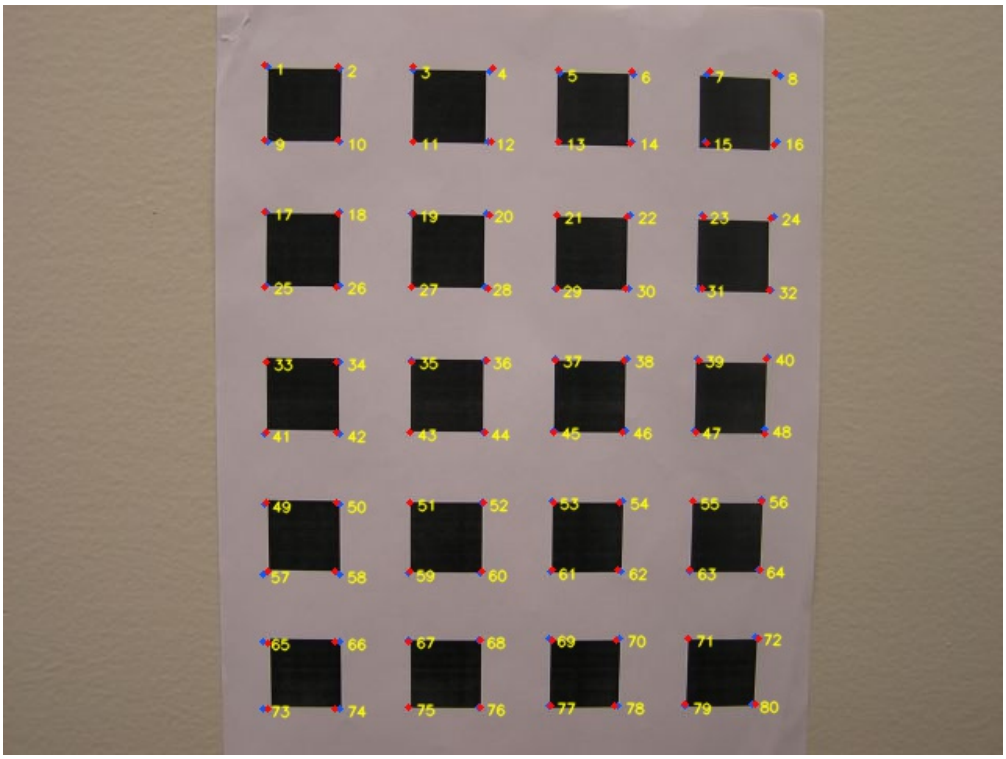
Dataset 1:

	Before LM		After LM	
	Image 2	Image 6	Image 2	Image 6
Mean	2.9585	2.7827	1.6014	1.5098
Variance	1.8584	1.2762	0.7092	0.8087
Max Distance	6.1403	5.5418	3.6831	4.0138

Table 3: Accuracy for Image 2 and Image 6 corners projected onto Image 28

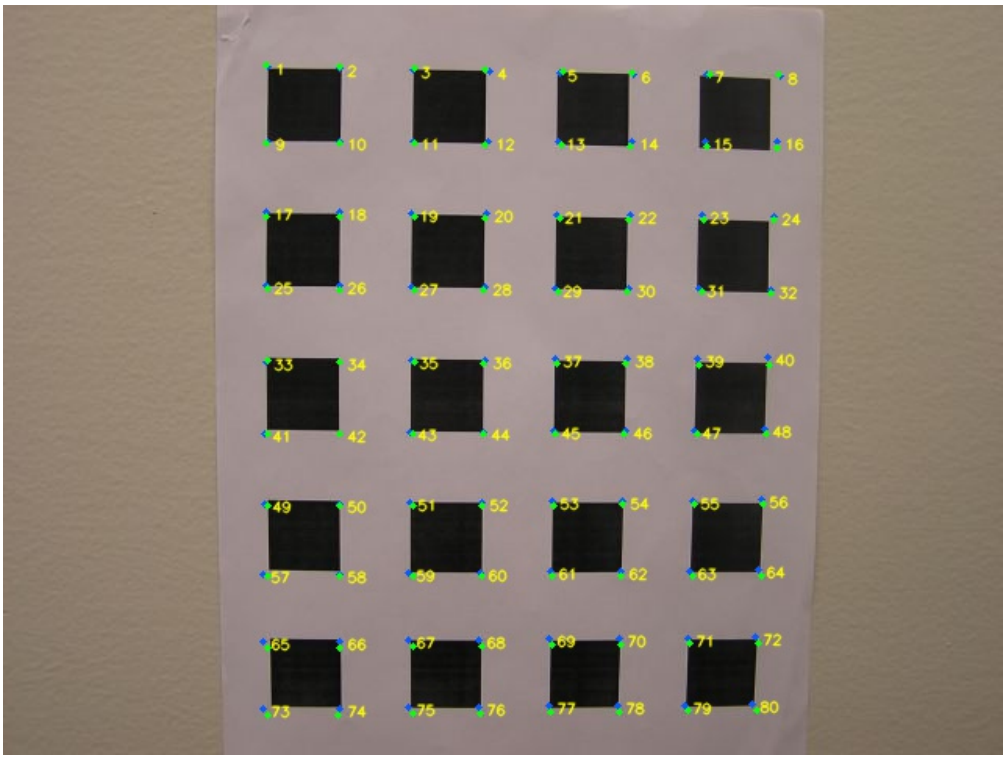


(a) Re-projection before LM (Green)

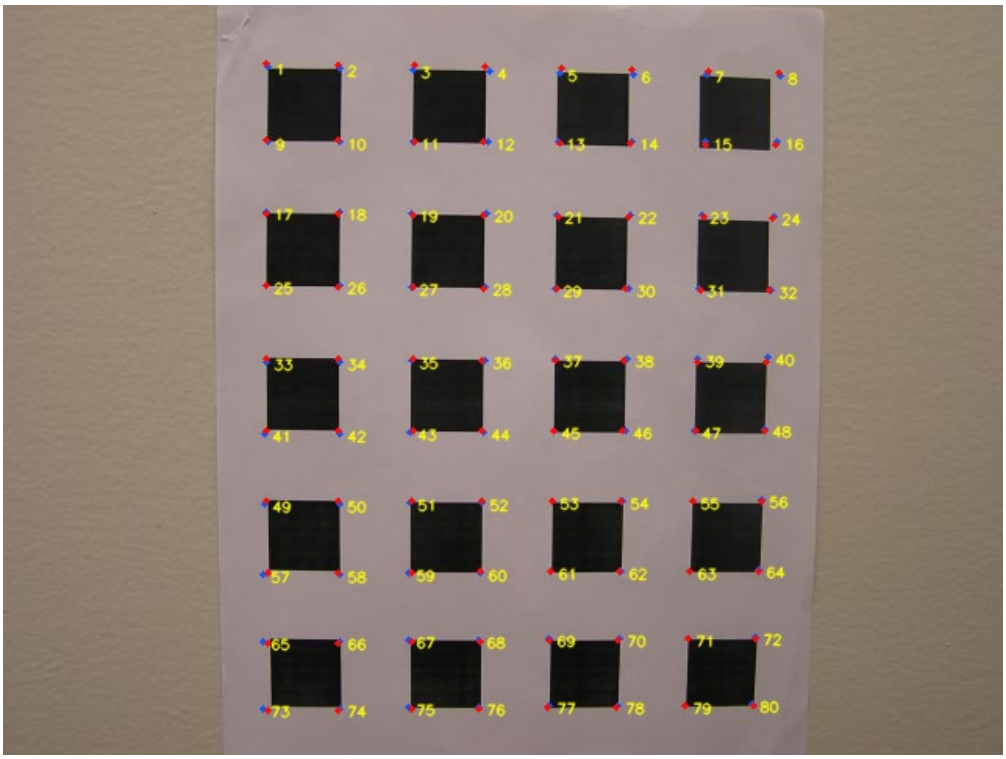


(b) Re-projection after LM (Red)

Figure 10: Image 2 on fixed Image 28



(a) Before LM



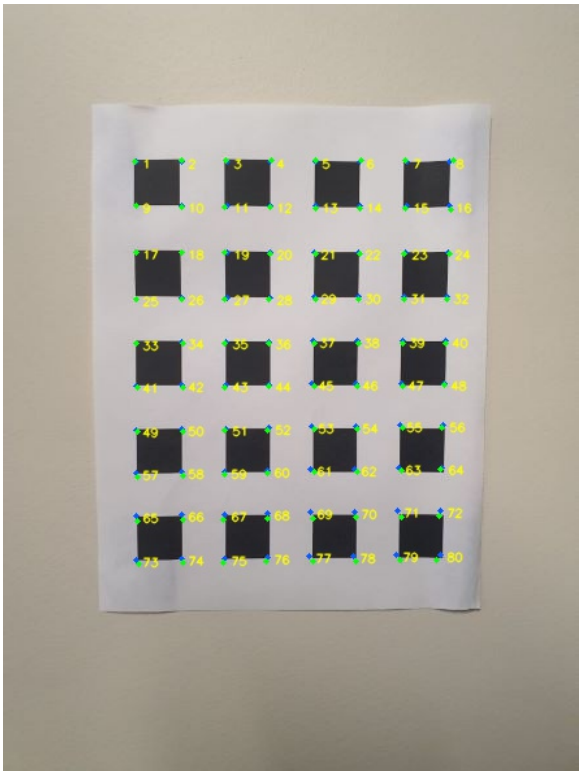
(b) After LM

Figure 11: Image 6 onto fixed Image 28

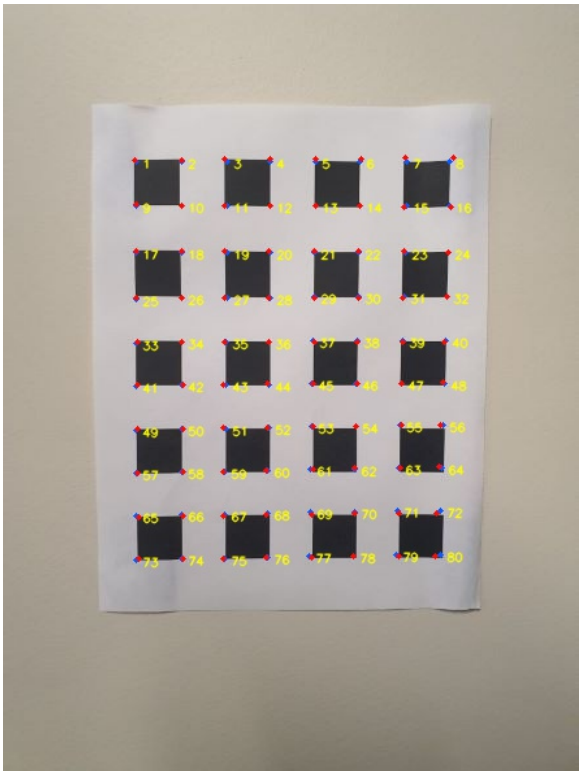
Dataset 2:

	Before LM		After LM	
	Image 2	Image 6	Image 2	Image 6
Mean	2.3954	4.4235	1.2938	1.6899
Variance	1.3509	5.7407	0.5845	1.1491
Max Distance	5.1242	8.6685	3.9461	4.6200

Table 4: Accuracy for Image 2 and Image 6 corners projected onto Image 10

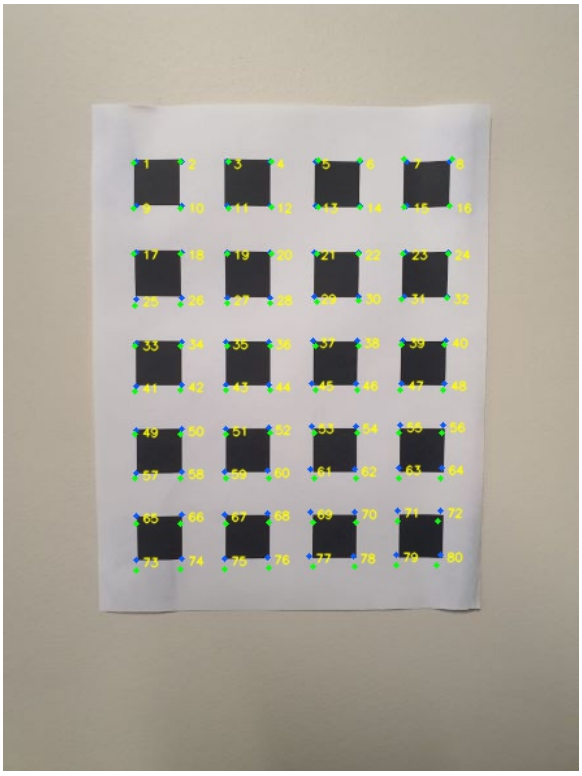


(a) Re-projection before LM (Green)

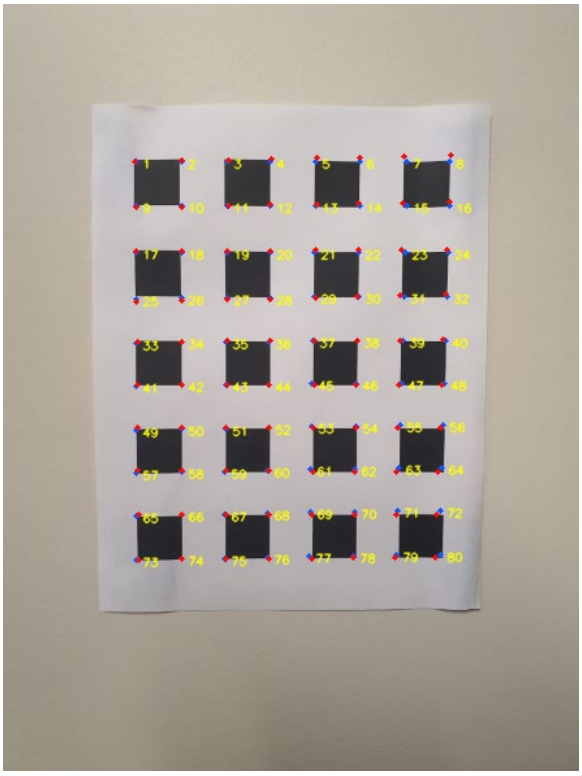


(b) Re-projection after LM (Red)

Figure 12: Image 2 onto fixed image 10



(a) Before LM



(b) After LM

Figure 13: Image 6 onto fixed image 10

1.3 Observations

Refining the parameters with the LM non-linear least square method improves accuracy. As can be seen from the projection of world corners onto each image before and after LM refining. The re-projection of corners of all the images onto the fixed images before and after LM refining visually shows the improvement in projection of corners. Overall accuracy for both datasets is given below:

	Dataset 1		Dataset 2	
	Before LM 1	After LM	Before LM 1	After LM
Mean	1.4517	0.8193	13.2897	1.1684
Variance	1.3681	0.2762	111.9522	0.5721
Max Distance	9.8283	4.0275	64.8083	6.4851

Table 5: Over all Accuracy for Dataset 1 and Dataset 2

For Dataset 2, the fixed image is Image 10. The extrinsic parameter matrix after LM refining in section 1.2.2 shows that R matrix is close to being an identity matrix. This is expected since the fixed image is taken such that the Principal Axis is approximately perpendicular to the calibration pattern, the x-axis of the image lines with the horizontal axis of the pattern and the y-axis of the image lines with the vertical axis of the pattern.

1.4 Source Code

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
import glob
import scipy.optimize
#=======================Functions=======================#
#### Corner Detection Functions
def getFixedPts(gridSize, hlines, vlines):
    x = np.linspace(0, gridSize*(vlines-1), vlines)
    y = np.linspace(0, gridSize*(hlines-1), hlines)
    xmesh, ymesh = np.meshgrid(x,y)
    fixedPts = np.concatenate([xmesh.reshape((-1,1)), ymesh.reshape((-1,1)
                                                )], axis=1)
    return fixedPts

def getPoints(lines):
    '''
    Code referenced from https://docs.opencv.org/3.4/d9/db0/
                                                tutorial_hough_lines.html
    '''
    rho = lines[:,0]
    theta = lines[:,1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
```

```
    y0 = b * rho
    p1 = np.array([x0 + 1000*(-b), y0 + 1000*(a)])
    p2 = np.array([x0 - 1000*(-b), y0 - 1000*(a)])
    return p1.T, p2.T

def drawLines(image, pts1, pts2, color, t=1):
    img = image.copy()
    for i in range(0, pts1.shape[0]):
        pt1 = (int(pts1[i,0]),int(pts1[i,1]))
        pt2 = (int(pts2[i,0]),int(pts2[i,1]))
        cv.line(img, pt1, pt2, color, t)
    return img

def drawPoints(image, pts, r, color, t=1, text = False):
    img = image.copy()
    for i in range(pts.shape[0]):
        pt1 = int(pts[i,0])
        pt2 = int(pts[i,1])
        cv.circle(img, (pt1,pt2), r, color, -1)
        if text:
            cv.putText(img, str(i+1), (pt1+5,pt2+5), cv.
                                                    FONT_HERSHEY_SIMPLEX, 0.3
                                                    , (0,255,255), 1, cv.
                                                    LINE_AA)
    return img

def getLines(imgName, num, r, thres, dataset):
    img = cv.imread(imgName)
    # Canny and Hough Transform to get lines
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    if dataset==2:
        gray = cv.GaussianBlur(gray,(3,3),2)
    edges = cv.Canny(gray, 255*1.5, 255)
    lines = cv.HoughLines(edges, 1, r*np.pi/180, thres)
    lines = np.squeeze(lines)
    if dataset ==1:
        if num<5:
            # Save Canny Image
            cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                    CornersAndLines/Pics_'+
                                                    str(num+1)+'_canny_d1.jpg
                                                    ', edges)
            # Save HoughLines Image
            p1, p2 = getPoints(lines)
            img_HoughLines = drawLines(img, p1, p2, (255,255,10), 1)
            cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                    CornersAndLines/Pics_'+
                                                    str(num+1)+'_HoughL_d1.
                                                    jpg', img_HoughLines)
    if dataset==2:
        if num<12:
            # Save Canny Image
            cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                    CornersAndLines/Pics_'+
```

```
                                                    str(num+1)+'_canny_d2.jpg
                                                    ', edges)
            # Save HoughLines Image
            p1, p2 = getPoints(lines)
            img_HoughLines = drawLines(img, p1, p2, (255,255,10), 1)
            cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                    CornersAndLines/Pics_'+
                                                    str(num+1)+'_HoughL_d2.
                                                    jpg', img_HoughLines)
    return lines

def refineLines(lines_sorted, nms_r, v=0):
    validLines = []
    if v == 1:# if vertical
        d = lines_sorted[:,0]*np.cos(lines_sorted[:,1]) # get distance (
                                                    already sorted from before)
        d_thres = nms_r * (np.max(np.abs(d))-np.min(np.abs(d)))/7 #
                                                    average distance ratio
    else:
        d = lines_sorted[:,0]*np.sin(lines_sorted[:,1])
        d_thres = nms_r * (np.max(np.abs(d))-np.min(np.abs(d)))/9
    for i in range(d.shape[0]-1):
        if d[i+1] - d[i] > d_thres:
            #if the distance between lines is > thresh, take first line
            validLines.append([lines_sorted[i,0], lines_sorted[i,1]])
        if i==d.shape[0]-2:
            validLines.append([lines_sorted[i+1,0], lines_sorted[i+1,1]])
    return np.asarray(validLines)

def getCorners(imgName, num, lines, nms, dataset):
    img = cv.imread(imgName)
    # get vertical and horizontal lines
    vlines = lines[np.where(np.cos(lines[:,1])**2 > 0.5)]
    hlines = lines[np.where(np.cos(lines[:,1])**2 <=0.5)]

    # Sort the lines (hlines-> up to down, vlines -> left to right)
    vlines_sort = vlines[:,0]*np.cos(vlines[:,1])
    hlines_sort = hlines[:,0]*np.sin(hlines[:,1])
    vlines_sorted = vlines[np.argsort(vlines_sort)]
    hlines_sorted = hlines[np.argsort(hlines_sort)]

    # Refine lines
    vlines_refined = refineLines(vlines_sorted, nms, v = 1)
    hlines_refined = refineLines(hlines_sorted, nms)

    vpt1, vpt2 = getPoints(vlines_refined)
    hpt1, hpt2 = getPoints(hlines_refined)
    imgLines = drawLines(img, vpt1, vpt2, (10,10,255), 1)
    imgLines = drawLines(imgLines, hpt1, hpt2, (10,10,255), 1)

    # Save refined lines image
    if dataset==1:
        cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/CornersAndLines/
                                                    Pics_'+str(num+1)+'_lines_d1.
```

```
                                                jpg', imgLines)
    elif dataset==2:
        cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/CornersAndLines/
                                                Pics_'+str(num+1)+'_lines_d2.
                                                jpg', imgLines)

    # Get corners
    vpt1_hc = np.append(vpt1, np.ones((vpt1.shape[0],1)), axis=1)
    vpt2_hc = np.append(vpt2, np.ones((vpt2.shape[0],1)), axis=1)
    hpt1_hc = np.append(hpt1, np.ones((hpt1.shape[0],1)), axis=1)
    hpt2_hc = np.append(hpt2, np.ones((hpt2.shape[0],1)), axis=1)
    vlines_HC = np.cross(vpt1_hc, vpt2_hc)  #valid vertical lines
    hlines_HC = np.cross(hpt1_hc, hpt2_hc)  #valid horizontal lines

    corners = []
    for i in range(hlines_HC.shape[0]):
        # Intersection of 2 lines
        c = np.cross(vlines_HC, hlines_HC[i])
        c = c[:,:2]/c[:,2].reshape((-1,1))
        corners.append(c)
    corners = np.asarray(corners)
    corners = corners.reshape((-1,2))
    imgPoints = drawPoints(img, corners, 2, (255,90,10), 1, text=True)
    if dataset==1:
        cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/CornersAndLines/
                                                Pics_'+str(num+1)+'_points_d1
                                                .jpg', imgPoints)
    elif dataset==2:
        cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/CornersAndLines/
                                                Pics_'+str(num+1)+'_points_d2
                                                .jpg', imgPoints)
    return corners

def getHomography(domainPts, RangePts):
    # rangePts = H*domainPts
    numPts = len(domainPts)
     # A and B Matrix
    A = np.zeros((2*numPts,8))
    b = np.zeros((2*numPts,1))
    for i in range(numPts):
        xp = RangePts[i,0]
        yp = RangePts[i,1]
        x = domainPts[i,0]
        y = domainPts[i,1]
        # A matrix
        A[2*i][0:8] = [x, y, 1 , 0 , 0 , 0 ,-x*xp, -y*xp]
        A[2*i+1][0:8] = [0, 0, 0, x, y, 1, -x*yp, -y*yp]
        # B matrix
        b[2*i] = xp
        b[2*i+1] = yp

    # H matrix
    H = np.zeros((3,3))
    H = np.matmul(np.linalg.pinv(A),b)
```

```
    H = np.append(H,[1])
    H = np.reshape(H, (3,3))
    return H

#### Camera Calibration Functions

def calcVij(hi,hj):
    Vij = np.array([hi[0]*hj[0],
                    hi[0]*hj[1] + hi[1]*hj[0],
                    hi[1]*hj[1],
                    hi[2]*hj[0] + hi[0]*hj[2],
                    hi[2]*hj[1] + hi[1]*hj[2],
                    hi[2]*hj[2]])
    return Vij.T

def getOmega(Hs):
    # vb = 0
    V = []
    for H in Hs:
        h1 = H[:,0]
        h2 = H[:,1]
        h3 = H[:,2]
        v11 = calcVij(h1,h1)
        v12 = calcVij(h1,h2)
        v22 = calcVij(h2,h2)
        V.append(v12.T)
        V.append((v11-v22).T)

    # Compute SVD of v to find b
    u,s,v = np.linalg.svd(V)
    b = v[-1]
    omega = np.zeros((3,3))
    omega[0,0] = b[0]
    omega[0,1] = b[1]
    omega[0,2] = b[3]
    omega[1,0] = b[1]
    omega[1,1] = b[2]
    omega[1,2] = b[4]
    omega[2,0] = b[3]
    omega[2,1] = b[4]
    omega[2,2] = b[5]
    return omega

def getK(w):
    y_0 = (w[0,1] * w[0,2] - w[0,0] * w[1,2])/(w[0,0] * w[1,1] - w[0,1]**2
                                               )
    lamb = w[2,2] - (w[0,2]**2 + y_0 * (w[0,1] * w[0,2] - w[0,0] * w[1,2])
                                                )/w[0,0]
    alpha_x = np.sqrt(lamb/w[0,0])
    alpha_y = np.sqrt(lamb*(w[0,0]/(w[0,0] * w[1,1] - w[0,1]**2)))
    s = -w[0,1] * alpha_x**2 * alpha_y/ lamb
    x_0 = (s * y_0 / alpha_y) - (w[0,2] * alpha_x**2 / lamb)
    K = np.array([[alpha_x, s, x_0], [0, alpha_y, y_0], [0, 0, 1]])
    return K
```

```
def getRt(Hs, K):
    RList = []
    tList = []
    for H in Hs:
        rtMat = np.dot(np.linalg.inv(K), H)
        scale = 1/(np.linalg.norm(rtMat[:,0]))
        rtMat = scale*rtMat
        r1 = rtMat[:,0]
        r2 = rtMat[:,1]
        r3 = np.cross(r1,r2)
        Q = np.vstack((r1,r2, r3)).T
        u,_,v = np.linalg.svd(Q)
        R = np.matmul(u,v)
        RList.append(R)
        tList.append(rtMat[:,2])
    return RList, tList

def getParams(K, RList, tList):
    # K params
    alpha_x = K[0,0]
    s = K[0,1]
    x_0 = K[0,2]
    alpha_y = K[1,1]
    y_0 = K[1,2]
    K_param = np.asarray([alpha_x, s, x_0, alpha_y, y_0])

    # Convert R into 3DOF
    RtList_i = []
    for R,t in zip(RList, tList):
        val = (np.trace(R)-1)/2
        phi = np.arccos(val)
        W = phi / (2*np.sin(phi)) * np.asarray([R[2,1]-R[1,2],
                                                R[0,2]-R[2,0],
                                                R[1,0]-R[0,1]])
        Rt_i = np.append(W,t)
        RtList_i.append(Rt_i)
    p = np.append(K_param, np.concatenate(RtList_i))
    return p

def getKRt_mat(params):
    numRt_mats = int((params.shape[0]-5) / 6)
    Ks = params[:5]
    K = np.array([[Ks[0], Ks[1], Ks[2]], [0, Ks[3], Ks[4]], [0, 0, 1]])

    RList = []
    tList = []
    Rts = params[5:]
    for i in range(numRt_mats):
        W = Rts[i*6:i*6+3]
        t = Rts[i*6+3:i*6+6]
        phi = np.linalg.norm(W)
        Wx = np.array([[0, -W[2], W[1]], [W[2], 0, -W[0]], [-W[1], W[0], 0
                                                            ]])
```

```
        R = np.eye(3) + (np.sin(phi)/phi)*Wx + ((1-np.cos(phi))/(phi**2))*
                                                np.matmul(Wx,Wx)
        RList.append(R)
        tList.append(t)
    return K, RList, tList

def projPoints(H, coords):
    coords_hc = np.concatenate((coords, np.ones((coords.shape[0],1))),
                                            axis=1)
    projectedCoords_hc = np.dot(H, coords_hc.T).T
    projectedPts = projectedCoords_hc[:,:2]/projectedCoords_hc[:,2].
                                            reshape((coords.shape[0],1))
    return projectedPts

def lossFunc(params, corners, fixedPoints):
    ### d_geom = ||Xij - X^ij||
    K, RList, tList = getKRt_mat(params)
    projected_corners = []
    for R, t in zip(RList,tList):
        rt = np.vstack((R[:,0], R[:,1], t)).T
        H = np.dot(K, rt)
        projectedPts = projPoints(H, fixedPoints)
        projected_corners.append(projectedPts)

    projected_corners = np.concatenate(projected_corners, axis=0)
    actual_corners = np.concatenate(corners, axis=0)
    diff = projected_corners - actual_corners
    d_geom = diff.flatten()
    return d_geom

def reprojectGT(params, corners, fixedPoints, imgIdx, imgList, dataset, lm
                                        =False):
    K, RList, tList = getKRt_mat(params)
    Hs = []
    for R,t in zip(RList, tList):
        rt = np.vstack((R[:,0], R[:,1], t)).T
        H = np.dot(K, rt)
        Hs.append(H)

    diff_corners = []
    for i, H in enumerate(Hs):
        imageName = imgList[imgIdx[i]]
        img = cv.imread(imageName)
        projectedPts = projPoints(H, fixedPoints)
        imgPoints = drawPoints(img, projectedPts, 2, (255,90,10), 1, text
                                            = True)
        if lm == False:
            imgPoints = drawPoints(imgPoints, projectedPts, 2, (10,255,10)
                                                , 1, text = False)
        else:
            imgPoints = drawPoints(imgPoints, projectedPts, 2, (10,10,255)
                                                , 1, text = False)
        if dataset==1:
            if lm==False:
```

```
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/beforelm/
                                                    Pics_'+str(imgIdx[i]+
                                                    1)+'_blm_d1.jpg',
                                                    imgPoints)
            else:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/afterlm/
                                                    Pics_'+str(imgIdx[i]+
                                                    1)+'_alm_d1.jpg',
                                                    imgPoints)
        elif dataset==2:
            if lm==False:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/beforelm/
                                                    Pics_'+str(imgIdx[i]+
                                                    1)+'_blm_d2.jpg',
                                                    imgPoints)
            else:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/afterlm/
                                                    Pics_'+str(imgIdx[i]+
                                                    1)+'_alm_d2.jpg',
                                                    imgPoints)
        diff = projectedPts - corners[i]
        diff_corners.append(diff)

    diff_corners = np.array(diff_corners)
    diff_corners = diff_corners.reshape((-1,2))
    diff_indNorm = np.linalg.norm(diff_corners, axis=1)
    # Get individual mean, var and max
    avg = []
    var = []
    maxd = []
    for i in range(int(diff_indNorm.shape[0]/80)):
        cset = diff_indNorm[i*80 : i*80+80]
        avg.append(np.average(cset))
        var.append(np.var(cset))
        maxd.append(np.max(cset))

    # Get overall accuracy
    avg_all = np.average(diff_indNorm)
    var_all = np.var(diff_indNorm)
    max_d_all = np.max(diff_indNorm)

    return np.array([avg, var, maxd]), np.array([avg_all, var_all,
                                                 max_d_all])

def reprojectCorners(params, corners, imgIdx, imgList, fixedImgIdx,
                                            dataset, lm = False):
    K, RList, tList = getKRt_mat(params)
    Hs = []
    for R,t in zip(RList, tList):
        rt = np.vstack((R[:,0], R[:,1], t)).T
        H = np.dot(K, rt)
        Hs.append(H)

    # Index for the fixed image changes based on the parameters used
```

```
fixedIdx = np.where(imgIdx==fixedImgIdx)
fixedIdx = fixedIdx[0][0]-1
fixedImg_H = Hs[fixedIdx] # img 28
fixedCorners = corners[fixedIdx]
fixedImg = cv.imread(imgList[fixedImgIdx-1])
fixedImg = drawPoints(fixedImg, fixedCorners, 2, (255,90,10),1, text =
                                        True)

diff_corners = []
for i, H in enumerate(Hs):
    validIdx = imgIdx[i]
    if validIdx == fixedIdx:
        continue
    H_bet = np.dot(fixedImg_H, np.linalg.inv(H))
    projectedPts = projPoints(H_bet, corners[i])
    if lm == False:
        imgPoints = drawPoints(fixedImg, projectedPts, 2, (10,255,10),
                                                1, text = False)
    else:
        imgPoints = drawPoints(fixedImg, projectedPts, 2, (10,10,255),
                                                1, text = False)
    if validIdx in (1,5,9,10,12,29):
        if dataset==1:
            if lm == False:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                reprojection/
                                                Pics_'+str(
                                                validIdx+1)+'_d1.
                                                jpg', imgPoints)
            else:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                reprojection/
                                                Pics_'+str(
                                                validIdx+1)+'
                                                _lm_d1.jpg',
                                                imgPoints)
        elif dataset==2:
            if lm == False:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                reprojection/
                                                Pics_'+str(
                                                validIdx+1)+'_d2.
                                                jpg', imgPoints)
            else:
                cv.imwrite('HW8/outputs/dataset'+str(dataset)+'/
                                                reprojection/
                                                Pics_'+str(
                                                validIdx+1)+'
                                                _lm_d2.jpg',
                                                imgPoints)
    diff = projectedPts - fixedCorners
    diff_corners.append(diff)
diff_corners = np.array(diff_corners)
diff_corners = diff_corners.reshape((-1,2))
```

```
    diff_indNorm = np.linalg.norm(diff_corners, axis=1)
    avg = []
    var = []
    maxd = []
    for i in range(int(diff_indNorm.shape[0]/80)):
        cset = diff_indNorm[i*80 : i*80+80]
        avg.append(np.average(cset))
        var.append(np.var(cset))
        maxd.append(np.max(cset))
    return np.array([avg, var, maxd])
#=========================Main Function==========================#
dataset = 2
## Get images
if dataset == 1:
    imageList = sorted(glob.glob('HW8/Dataset1/Pic_??.jpg'))
elif dataset == 2:
    imageList = sorted(glob.glob('HW8/Dataset2/Pic_??.jpg'))

fixedPts = getFixedPts(gridSize = 10, hlines = 10, vlines = 8)

## Parameters for datasets
# Dataset 1: r = 0.5, thres = 60, nms = 0.24
# Dataset 2: r = 0.5, thres = 50, nms = 0.24

### Step 1: Corner Detection and homography calculation ###
Hs = []
cornersList = []
imgIdx = []
for num, imgName in enumerate(imageList):
    # Get Lines using Canny and HoughLines
    lines = getLines(imgName, num, 0.5, 50, dataset)
    corners = getCorners(imgName, num, lines, 0.24, dataset)
    # Get Homography from the corners
    if corners.shape[0]==80:
        H = getHomography(fixedPts, corners)
        Hs.append(H)
        cornersList.append(corners)
        imgIdx.append(num)
print("Images Used:", len(cornersList))
imgIdx = np.array(imgIdx)
Hs = np.array(Hs)
cornersList = np.array(cornersList)

### Step 2: Calculate Intrinsic and Extrinsic Parameters ###
# Calculate Omega
w = getOmega(Hs)
# Get K, [R|t] matrix for each image
K = getK(w)
RList, tList = getRt(Hs, K)
print("K before lm: \n",K)
if dataset==1:
    print("R before lm: \n",RList[3])
    print("t before lm: \n",tList[3])
    print("R before lm: \n",RList[4])
```

```
    print("t before lm: \n",tList[4])
elif dataset==2:
    print("R before lm: \n",RList[1])
    print("t before lm: \n",tList[1])
    print("R before lm: \n",RList[9])
    print("t before lm: \n",tList[9])
# Get initial values of the parameters p = [K, R_i, t_i | i=1,2,3...]^T
p = getParams(K, RList, tList)
# Reproject using parameters found before LM
acc, acc_all = reprojectGT(p, cornersList, fixedPts, imgIdx, imageList,
                                        dataset)

### Step 3: Refining the Calibration Parameters with LM ###
# Calculate loss
d_geom = lossFunc(p, cornersList, fixedPts)

# Apply LM
res = scipy.optimize.least_squares(fun=lossFunc, x0=p, method='lm', args =
                                         [cornersList, fixedPts])
K_refined, RList_refined, tList_refined = getKRt_mat(res.x)
print("K after lm: \n",K_refined)
if dataset==1:
    print("R after lm: \n",RList_refined[3])
    print("t after lm: \n",tList_refined[3])
    print("R after lm: \n",RList_refined[4])
    print("t after lm: \n",tList_refined[4])
elif dataset==2:
    print("R after lm: \n",RList_refined[1])
    print("t after lm: \n",tList_refined[1])
    print("R after lm: \n",RList_refined[9])
    print("t after lm: \n",tList_refined[9])
acc_lm, acc_lm_all = reprojectGT(res.x, cornersList, fixedPts, imgIdx,
                                        imageList, dataset, lm = True)

if dataset==1:
    # Overall Accuracy
    print("Overall Accuracy:")
    print(f'Before LM: Mean {acc_all[0]}, Var {acc_all[1]}, Max {acc_all[2
                                        ]}')
    print(f'After LM: Mean {acc_lm_all[0]}, Var {acc_lm_all[1]}, Max {
                                        acc_lm_all[2]}\n')

    # Individual Accuracy
    print("Individual Accuracy")
    print("Before LM:")
    print(f'For Image 4: Mean {acc[0][3]}, Var {acc[1][3]}, Max {acc[2][3]
                                        }')
    print(f'For Image 5: Mean {acc[0][4]}, Var {acc[1][4]}, Max {acc[2][4]
                                        }')
    print("After LM:")
    print(f'For Image 4: Mean {acc_lm[0][3]}, Var {acc_lm[1][3]}, Max {
                                        acc_lm[2][3]}')
    print(f'For Image 5: Mean {acc_lm[0][4]}, Var {acc_lm[1][4]}, Max {
                                        acc_lm[2][4]}')
```

```
if dataset==2:
    # Overall Accuracy
    print("Overall Accuracy:")
    print(f'Before LM: Mean {acc_all[0]}, Var {acc_all[1]}, Max {acc_all[2
                                        ]}')
    print(f'After LM: Mean {acc_lm_all[0]}, Var {acc_lm_all[1]}, Max {
                                        acc_lm_all[2]}\n')
    # Individual Accuracy
    print("Individual Accuracy")
    print("Before LM:")
    print(f'For Image 2: Mean {acc[0][1]}, Var {acc[1][1]}, Max {acc[2][1]
                                        }')
    print(f'For Image 10: Mean {acc[0][9]}, Var {acc[1][9]}, Max {acc[2][9
                                        ]}')
    print("After LM:")
    print(f'For Image 2: Mean {acc_lm[0][1]}, Var {acc_lm[1][1]}, Max {
                                        acc_lm[2][1]}')
    print(f'For Image 10: Mean {acc_lm[0][9]}, Var {acc_lm[1][9]}, Max {
                                        acc_lm[2][9]}')

### Step 4: Reproject corners onto fixed image using the refined
                                        parameters
# Image 28 works well as a "fixed" image for dataset 1
# Image 10 works well as "fixed" image for dataset 2
if dataset==1:
    fixedImgIdx = 28
if dataset==2:
    fixedImgIdx = 10

acc_c = reprojectCorners(p, cornersList, imgIdx, imageList, fixedImgIdx,
                                        dataset)
acc_lm_c = reprojectCorners(res.x, cornersList, imgIdx, imageList,
                                        fixedImgIdx, dataset, lm = True)

# Print acc for three specific images (2,6)
print("Reprojection Accuracy")
print("Before LM:")
print(f'For Image 2: Mean {acc_c[0][1]}, Var {acc_c[1][1]}, Max {acc_c[2][
                                        1]}')
print(f'For Image 6: Mean {acc_c[0][5]}, Var {acc_c[1][5]}, Max {acc_c[2][
                                        5]}')
print("After LM:")
print(f'For Image 2: Mean {acc_lm_c[0][1]}, Var {acc_lm_c[1][1]}, Max {
                                        acc_lm_c[2][1]}')
print(f'For Image 6: Mean {acc_lm_c[0][5]}, Var {acc_lm_c[1][5]}, Max {
                                        acc_lm_c[2][5]}')
```