ECE661: Homework 8

Submission By: Joseph Wang
wang3450@purdue.edu

Prepared for Dr. Avinash Kak
GTA: Fangda Li
Purdue University
November 14, 2022

Contents

List of Figures

1 Theory Question

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 ωin 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. Although w is defined as the image of the Absolute Conic, we cannot visually see it. The pixels that form ω are imaginary since $K^{-T}K^{-1}$ is positive definite. This result however is leveraged by Zhang's Algorithm for camera calibration. The image of the Absolute Conic is always invariant to rotation and translation distortions. Its relative position to a moving camera is only a function depending on the camera's intrisict parameter matrix K. This property of the Absolute Conic and its image allows us to compute K in Zhang's Algorithm.

2 Corner Detection

The first task in developing the camera calibration pipeline was identifying salient points on the calibration pattern for multiple views. Salient points were identified as corners of every black square on the pattern. In total, we identified 80 points for each view. Described below are the steps we took to identify these points.

1. Convert each image from the BGR space to grey-scale and apply a Canny edge detector to produce a binary mask of the edges in the input image. We set the upper and lower thresholds for the Canny edge detector to be 300.

2. Apply a Hough Transform to each of the binary masks to get a set of lines in polar form that constituted the edges seen in the binary mask.

3. Since the binary mask produced by the Canny edge detector was quite noisy, the Hough Transform yielded multiple lines in the same location. Thus we needed to cluster the lines before proceeding. To do this, we first clustered the lines into horizontal and vertical lines by thresholding θ. Any line where $\theta < \frac{\pi}{4}$ was classified as horizontal; the rest as vertical. After, we used Kmeans to form 8 clusters of vertical lines and 10 clusters of horizontal lines, before applying a heuristic to each cluster to yield a total of 18 unique lines.

4. From the 18 unique horizontal and vertical lines, we computed the 80 corners from all possible intersections. To keep track of corners, we sorted the horizontal and vertical lines based on their x- and y-intercepts respectively. This task is important later, when we estimate the homographies and need to make sure we have aligned point correspondences.

3 Zhang's Algorithm

The purpose of Zhang's Algorithm for camera calibration is to estimate a given camera's intrinsic and extrinsic parameters. This section looks to detail the steps we took to implement this method in our pipeline.

3.1 Estimating the Intrinsic Parameter k

We are interested in first estimating the image of the Absolute Conic because doing so allows us to compute the camera's intrinsic parameter matrix k. The steps to do so is as follows:

1. For at least three unique views of the calibration pattern, we build the following scaled system:

$$\vec{V}\vec{b} = \vec{0} \quad ; \quad \vec{V} = \begin{bmatrix} \vec{V_{12}} \\ (\vec{V_{11}} - \vec{V_{22}})^T \end{bmatrix} \quad ; \quad \vec{V_{ij}} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \quad ; \quad \vec{b} = \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}$$

   $\vec{V} \in \mathbb{R}^{2n\times 6}$, $\vec{b} \in \mathbb{R}^6$ where n is the number of unique views.
   Note: h_{ij} denotes the element from h at the i-th column and j-the row.

2. The solution to our system is solved using the technique of linear least squares. That is, $\vec{b}$, the image of the Absolute Conic, is the null space of $\vec{V}$.

3. Having quantified the Absolute Conic, we can then construct our intrinsic camera parameter k as:

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

   where:

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}} \quad \alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \quad s = -\frac{\omega_{11}\alpha_x^2\alpha_y}{\lambda} \quad x_0 = \frac{sy_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

$$y_0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \quad \lambda = \omega_{33} - \frac{\omega_{13} + y_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}}$$

3.2 Estimating the Extrinsic Parameters

Having estimated the intrinsic parameters k, we now estimate the rotations and translation, which constitute the camera's extrinsic parameters denoted as R and t. The implementation is as follows:

Since we assume our scene to lie completely in the $z = 0$ plane, the following holds true:

$$\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = K \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}$$

Applying some avious linear algebra properties, we can arrive at a closed from solution for both r and t as:

$$\begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix} = k^{-1} \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$$

However, since we are converting between a homogeneous representation and a non-homogeneous representation, we must apply a scale factor ξ to each operation.

Hence, the closed from solution for the rotational and translational matrices are:

$$R = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix} \in \mathbb{R}^{3\times3} \quad t = \begin{bmatrix} t_1 & t_2 & t_3 \end{bmatrix}^T \in \mathbb{R}^{3\times1}$$

where

$$\xi = \frac{1}{||k^{-1}h_1||} \quad r_1 = \xi k^{-1}h_1 \quad r_2 = \xi k^{-1}h_2 \quad r_3 = \xi r_1 \times r_2 \quad t = \xi k^{-1}h_3$$

At this point, the translation matrix is our final estimate of the translational distance for one specific view. However, we still need to perform singular value decomposition on R to ensure it meets the condition of being orthonormal as shown below:

$$UDV^T = svd(R)$$
$$R_{conditioned} = UV^T$$

This task of estimating the extrinsic parameters must be performed on all views of the calibration pattern.

3.3 Performing Estimation Refinement

3.3.1 Rodriguez Representation

Before actually performing non-linear least squares refinement on the camera parameters, we first need to introduce the Rodriguez representation of rotation matrices. In any optimization algorithm, the number of variables used to represent an entity must strictly equal the DoF of the entity. Since the rotation matrices we calculated in section 3.2 have 9 elements but only 3 DoF, we need to utilize the Rodriguez form to represent the 9 elements in 3.

This representation converts the R matrix into a 3-vector $\vec{w} = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}$ To allow conversion between R and $\vec{w}$, we represent $\vec{w}$ by the 3×3 matrix below:

$$\begin{bmatrix} w_X \end{bmatrix} = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$$

The Rodriguez Rotation formula is thus given by:

$$R = e^{\begin{bmatrix} w_X \end{bmatrix}} = I + \frac{\sin\phi}{\phi}\begin{bmatrix} w_X \end{bmatrix} + \frac{1-\cos\phi}{\phi^2}\begin{bmatrix} w_X \end{bmatrix}^2 \quad \phi = ||\vec{w_X}||$$

and the back transformation as:

$$\vec{w} = \frac{\phi}{2\sin\phi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \quad \phi = \arccos\frac{tr(R) - 1}{2}$$

3.3.2 Cost Function

Having converted each R matrix for the n number of views of the scene to its respective Rodriguez representation, we are now ready to establish the cost function we will minimize to refine our camera parameter estimates. The cost function we wish to optimize is defined as the sum of square of the geometric distance between the "ground truth" corner and the re-projected corner. Slight differences in implementation arise depending if radial distortion of the camera is considered during refinement. The two cost functions considered are detailed below:

1. Cost Function without Radial Distortions

   $$d_{geom}^2 = \sum_i \sum_j ||x_{ij} - \hat{x}_{ij}||^2 = \sum_i \sum_j ||x_{ij} - K[R_i t_i] X_{mj}||^2 = \sum_i \sum_j ||x_{ij} - K[r_{i,1}, r_{i,2}, t_i]||^2$$

   Note: The parameter vector will be of size $5 + 6n$ since there are 5 intrinsic parameters and 6 extrinsic for every view.

2. Cost Function with Radial Distortion
   We can consider the radial distortion with two parameters, k_1 and k_2, as follows:

   $$\begin{aligned} r^2 &= (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2 \\ \hat{x}_{rad} &= \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4] \\ \hat{y}_{rad} &= \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4] \end{aligned}$$

   Then, we use $(\hat{x}_{rad}, \hat{y}_{rad})$ as the new estimate. Note: The parameter vector will be of size $7 + 6n$ to account for the additional two radial distortion parameters.

4 Results

4.1 Tabulated Results For Given Image Set

$$k = \begin{bmatrix} 7.27611186e+02 & 2.60919073e-02 & 3.20441973e+02 \\ 0.00000000e+00 & 7.26434435e+02 & 2.42237902e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

$$R_1 = \begin{bmatrix} 0.78755375 & -0.18504673 & 0.58780677 \\ 0.19763533 & 0.97930997 & 0.04350012 \\ -0.58369458 & 0.0819127 & 0.80783101 \end{bmatrix}$$

$$t_1 = \begin{bmatrix} -37.29988982 & -103.08370954 & 441.81089493 \end{bmatrix}$$

$$R_{10} = \begin{bmatrix} 0.74717874 & 0.1886999 & -0.63727253 \\ 0.13193634 & 0.89765281 & 0.42049046 \\ 0.65139599 & -0.39826094 & 0.64581073 \end{bmatrix}$$

$$t_{10} = \begin{bmatrix} -58.55024996 & -95.95843495 & 426.46924173 \end{bmatrix}$$

Table 1: Reprojection Error Before LM		
Image	**Mean Error**	**Error Variance**
Pic 1	1.2124966087180533	0.4419708546416626
Pic 10	0.8883294030357984	0.2100304351027858

Table 2: Reprojection Error After LM no Rad Distortion		
Image	**Mean Error**	**Error Variance**
Pic 1	0.9566193624472742	0.277279372102838
Pic 10	0.7865193333734151	0.1926577380734602

Table 3: Reprojection Error After LM yes Rad Distortion		
Image	**Mean Error**	**Error Variance**
Pic 1	0.8937247765240638	0.193048584731136
Pic 10	0.7584877316371811	0.13360787393238654

Table 4: Radial Distortion Parameters	
Parameter	**Value**
k_1	-1.7930708530386655e-07
k_2	7.915913690154872e-13

4.2 Tabulated Results For Custom Image Set

$$k = \begin{bmatrix} 731.2361273 & -8.14104127 & 249.65586556 \\ 0. & 734.2382484 & 419.03144129 \\ 0. & 0. & 1. \end{bmatrix}$$

$$R_2 = \begin{bmatrix} 0.93632629 & 0.03682588 & -0.34919469 \\ 0.01124338 & 0.99083075 & 0.13464032 \\ 0.35095108 & -0.1299934 & 0.92732683 \end{bmatrix}$$

$$t_2 = \begin{bmatrix} -18.98573263 & -97.4432662 & 295.54423094 \end{bmatrix}$$

$$R_3 = \begin{bmatrix} 0.95250464 & -0.06049478 & -0.29845484 \\ 0.02189907 & 0.99113949 & -0.13100742 \\ 0.30373564 & 0.11824929 & 0.94538974 \end{bmatrix}$$

$$t_3 = \begin{bmatrix} -27.33030725 & -74.6409931 & 248.43488808 \end{bmatrix}$$

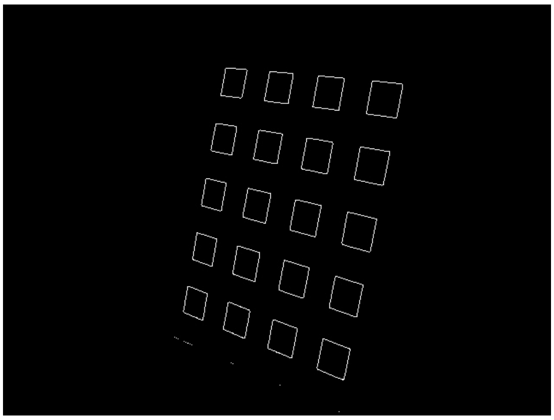
Table 5: Reprojection Error Before LM		
Image	**Mean Error**	**Error Variance**
Custom 2	5.673692471060289	7.681626701637438
Custom 3	2.144602223590031	0.8177726936095497

Table 6: Reprojection Error After LM no Rad Distortion		
Image	**Mean Error**	**Error Variance**
Custom 2	2.0237291672051625	1.2993388207586043
Custom 3	1.9446312218711668	1.437811608601527

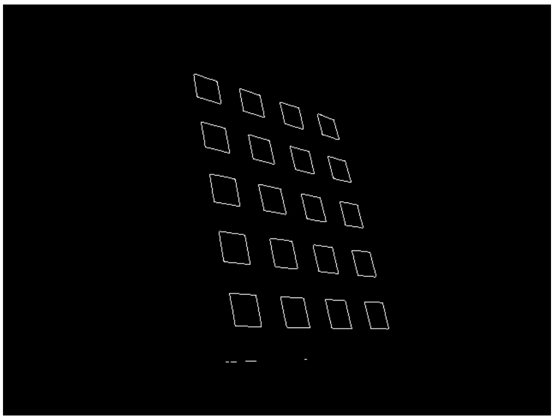
Table 7: Reprojection Error After LM yes Rad Distortion		
Image	**Mean Error**	**Error Variance**
Custom 2	1.1115186663304994	0.2962818868566509
Custom 3	1.0638838434119937	0.31979058085783557

Table 8: Radial Distortion Parameters	
Parameter	**Value**
k_1	2.5037809515903676e-07
k_2	-1.3757778821024775e-12

4.3 Graphical Results on Given Images

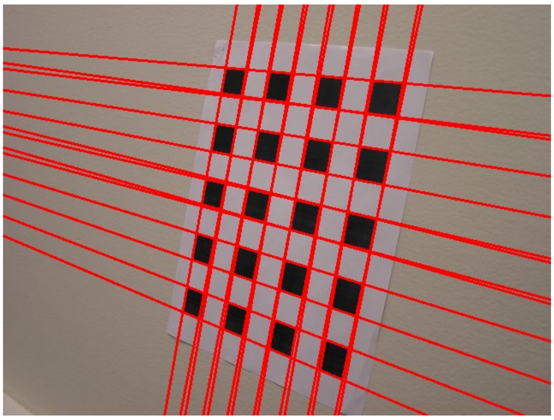


(a) Edge Bit Mask Pic 1

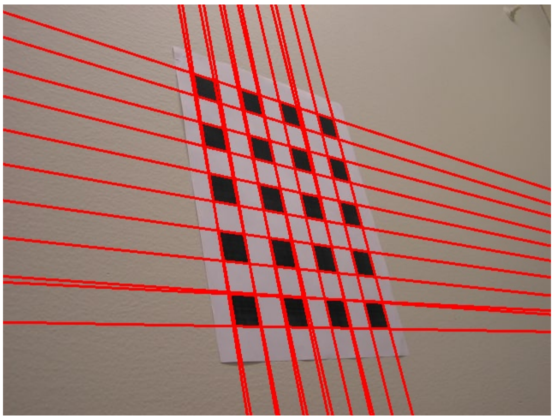


(b) Edge Bit Mask Pic 10

Figure 1: Canny Edge Detector on Given Image Set

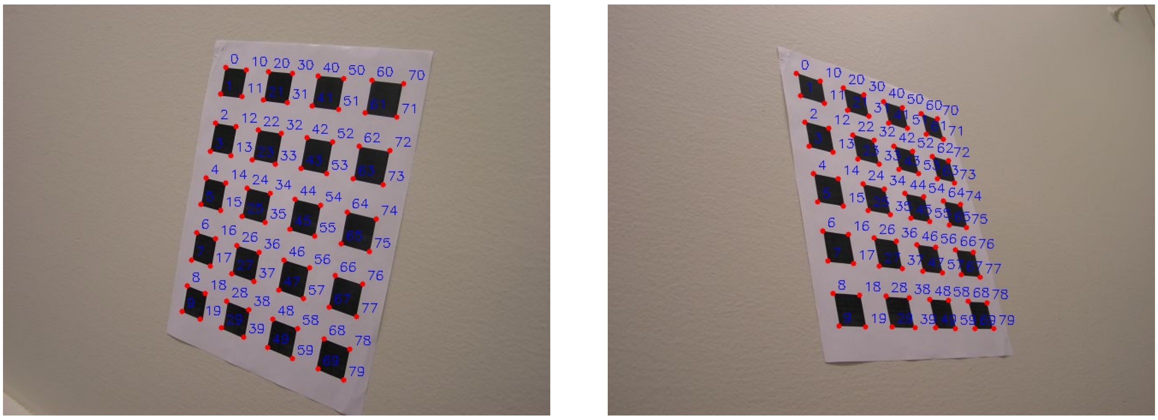


(a) Hough Lines Pic 1



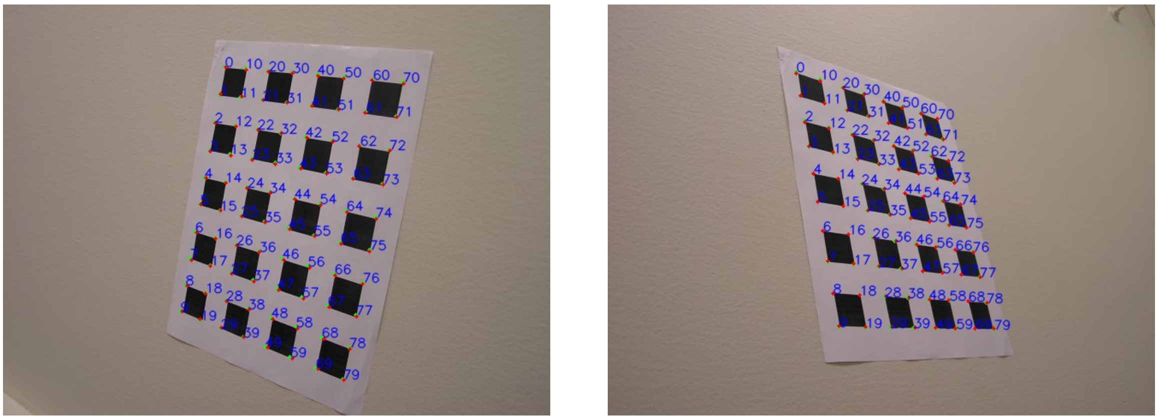
(b) Hough Lines Pic 10

Figure 2: Hough Lines on Given Image Set



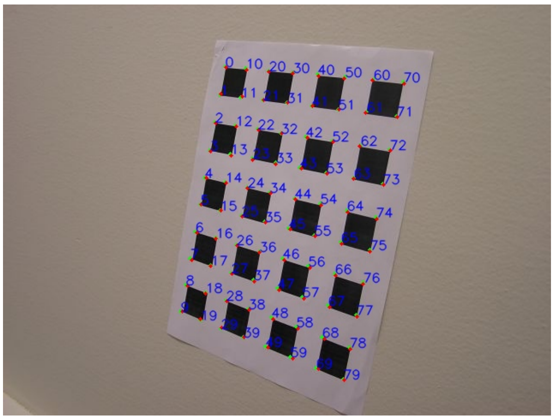
(a) Corners Pic 1 (b) Corners Pic 10

Figure 3: Identified "Ground Truth" Corners Given Set

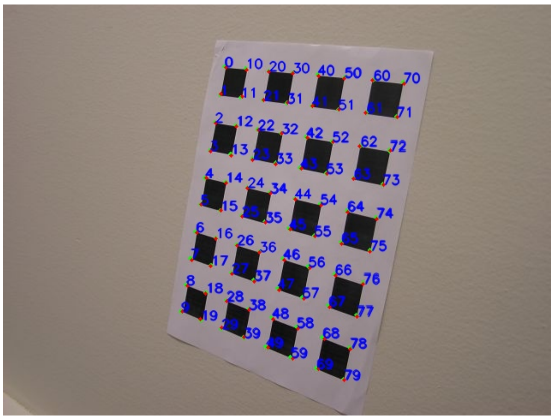


(a) Reprojected Corners Pic 1 (b) Reprojected Corners Pic 10

Figure 4: Reprojected Corners Given Set (red: original "ground truth", green: reprojected)

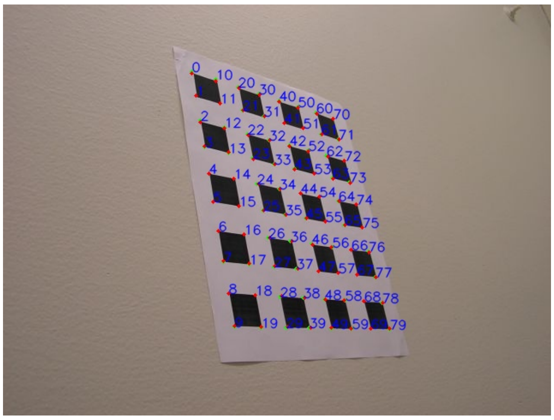


(a) Pic 1 Before Refinement

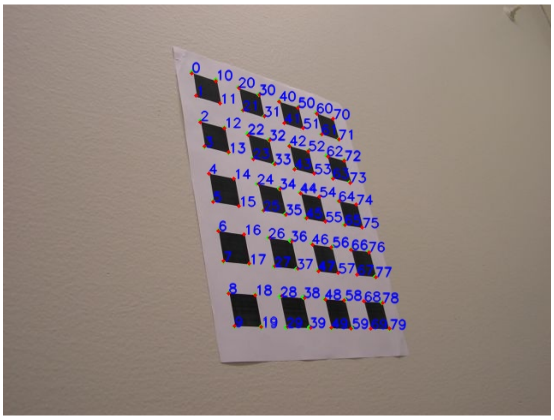


(b) Pic 1 Refined with LM no Radial Distortion

Figure 5: Before and After Reprojection with LM no Radial Distortion Pic 1 (red: original "ground truth", green: reprojected)

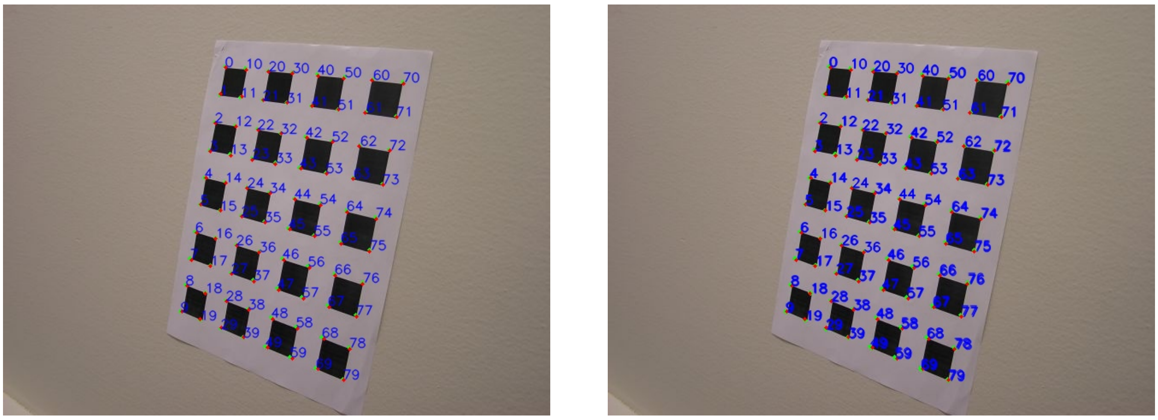


(a) Pic 10 Before Refinement



(b) Pic 10 Refined with LM no Radial Distortion

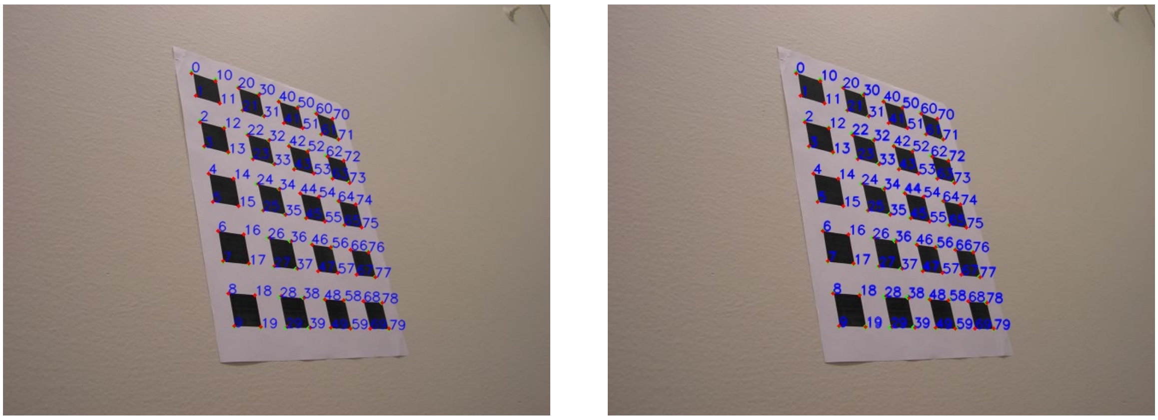
Figure 6: Before and After Reprojection with LM no Radial Distortion Pic 10 (red: original "ground truth", green: reprojected)



(a) Pic 1 Before Refinement

(b) Pic 1 Refined with LM Radial Distortion

Figure 7: Before and After Reprojection with LM and Radial Distortion Pic 1 (red: original "ground truth", green: reprojected)

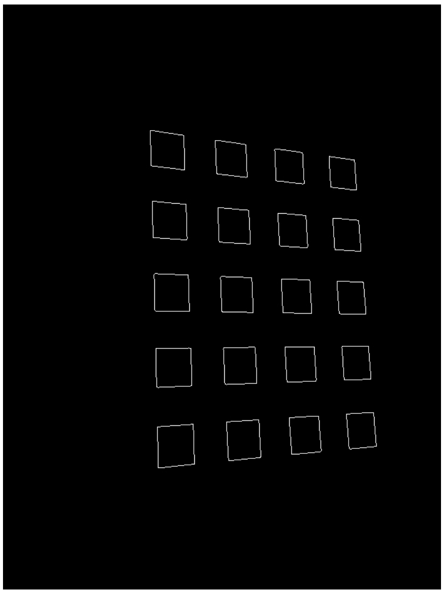


(a) Pic 10 Before Refinement

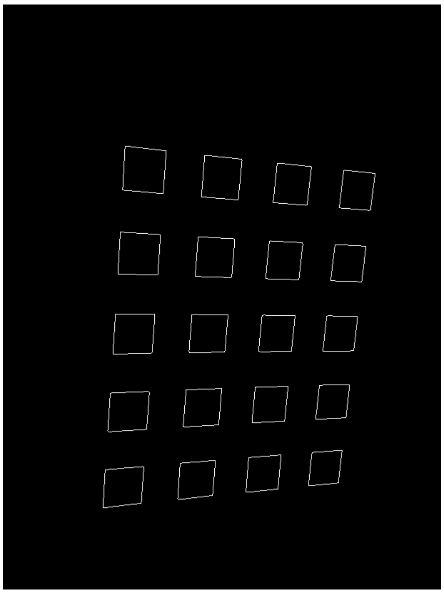
(b) Pic 10 Refined with LM Radial Distortion

Figure 8: Before and After Reprojection with LM Radial Distortion Pic 10 (red: original "ground truth", green: reprojected)

4.4 Graphical Results on Custom Images

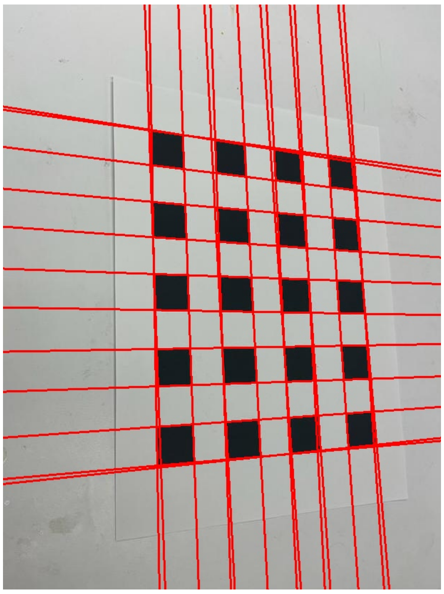


(a) Edge Bit Mask Custom 2

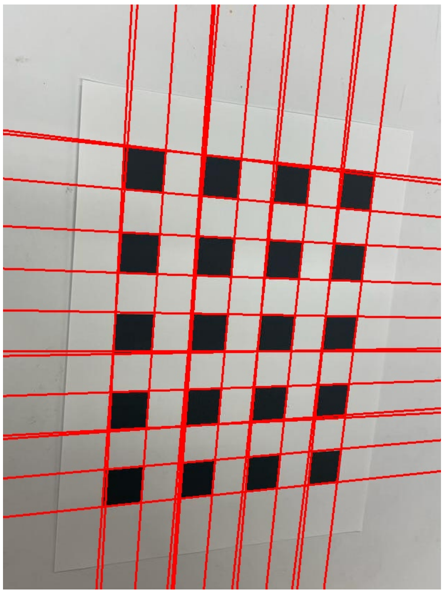


(b) Edge Bit Mask Custom 3

Figure 9: Canny Edge Detector on Custom Image Set

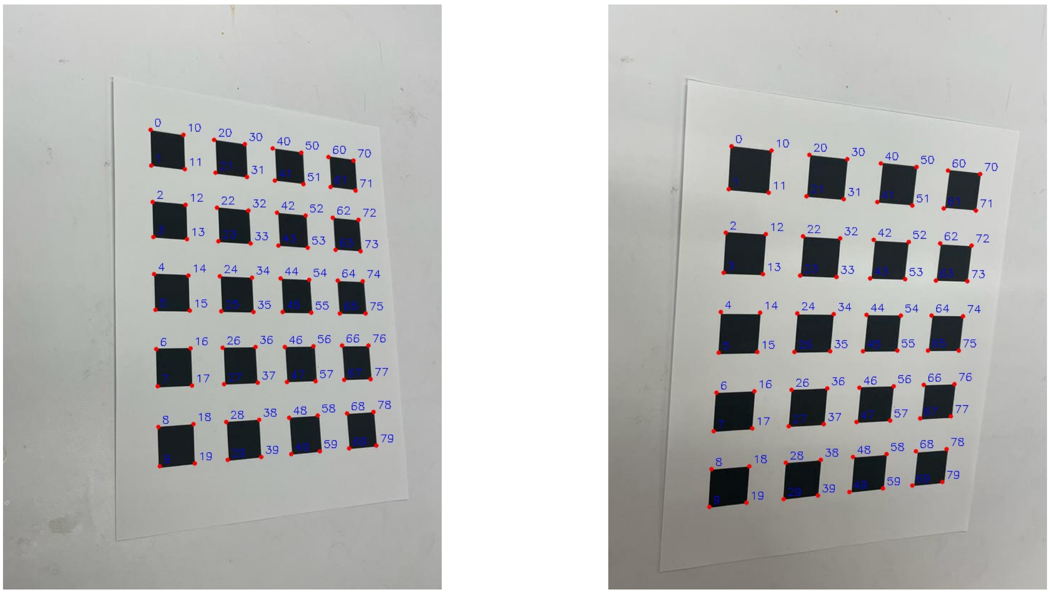


(a) Hough Lines Custom 2



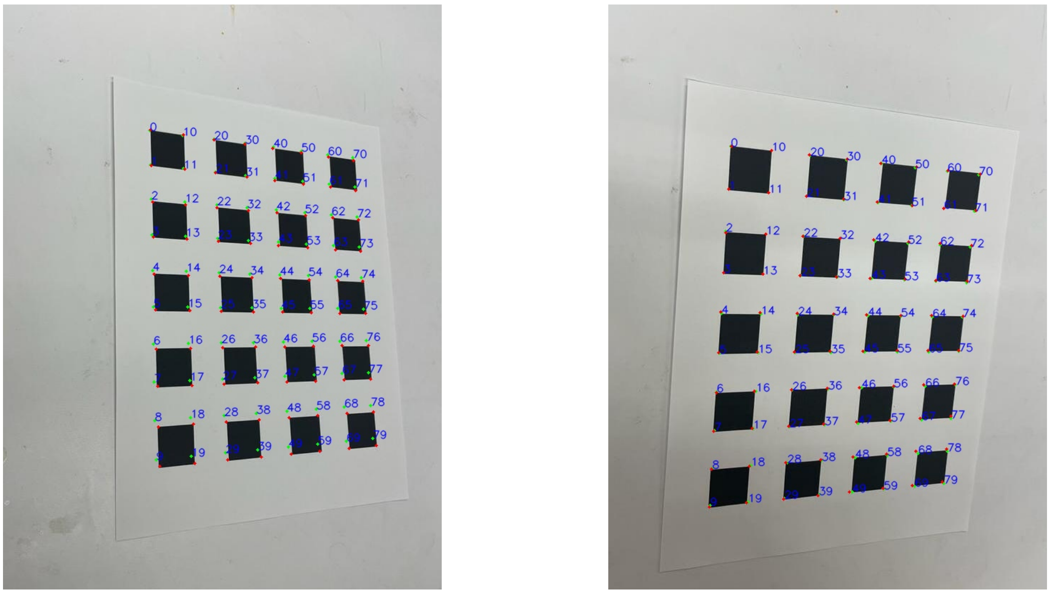
(b) Hough Lines Custom 3

Figure 10: Hough Lines on Custom Image Set



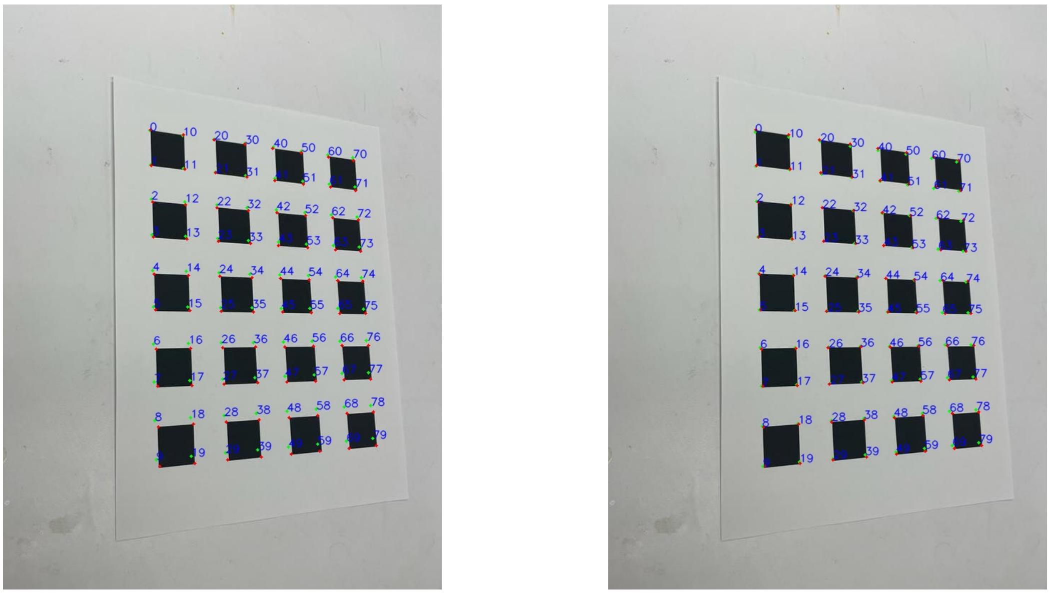
(a) Corners Custom 2 (b) Corners Custom 3

Figure 11: Identified "Ground Truth" Corners Custom Set



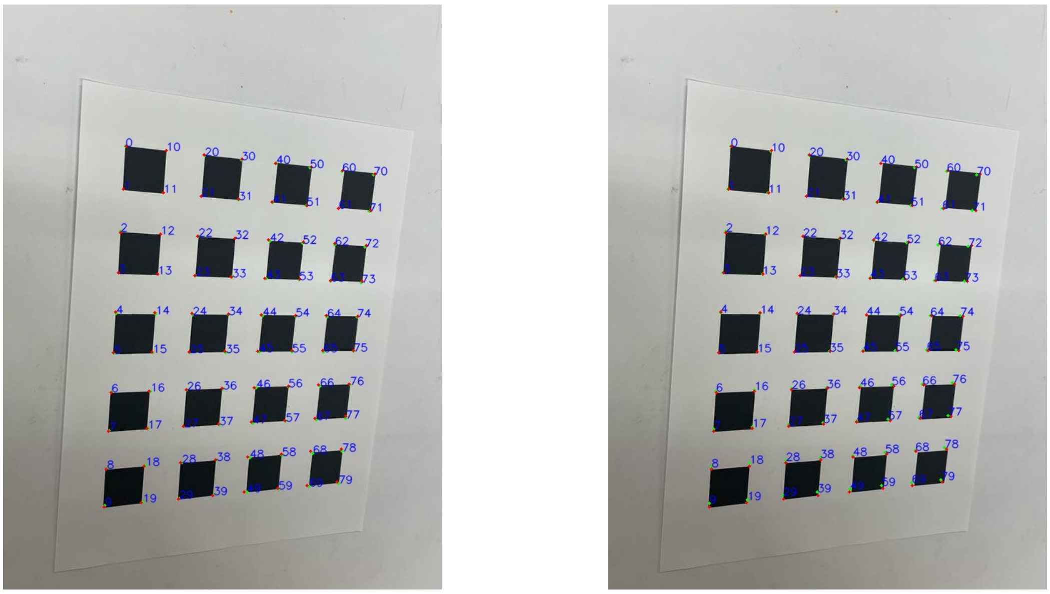
(a) Reprojected Corners Custom 2 (b) Reprojected Corners Custom 3

Figure 12: Reprojected Corners Custom Set (red: original "ground truth", green: reprojected)



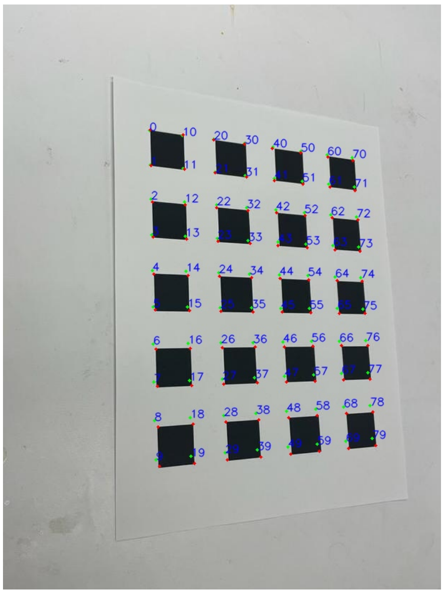
(a) Custom 2 Before Refinement (b) Custom 2 Refined with LM no Radial Distortion

Figure 13: Before and After Reprojection with LM no Radial Distortion Custom 2 (red: original ”ground truth”, green: reprojected)

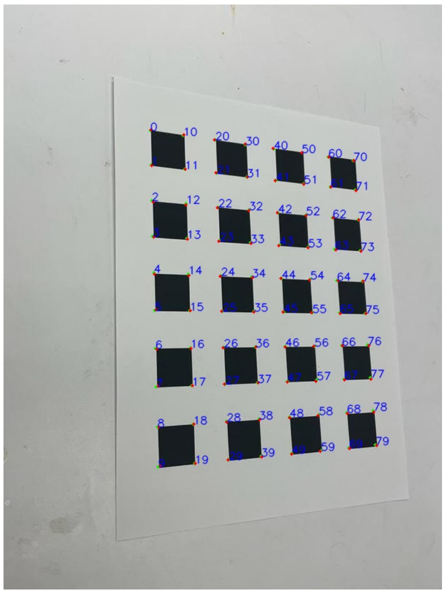


(a) Custom 3 Before Refinement (b) Custom 3 Refined with LM no Radial Distortion

Figure 14: Before and After Reprojection with LM no Radial Distortion Custom 3 (red: original ”ground truth”, green: reprojected)

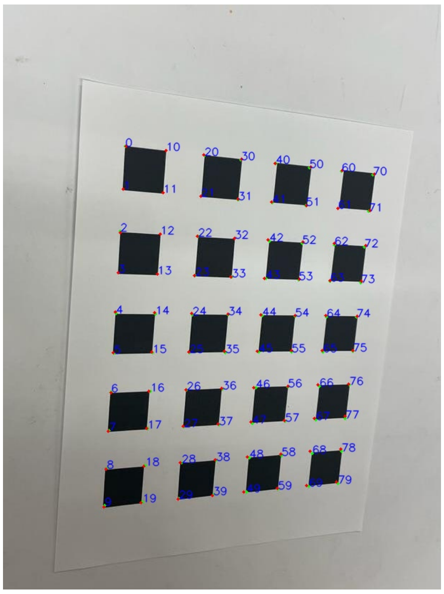


(a) Custom 2 Before Refinement

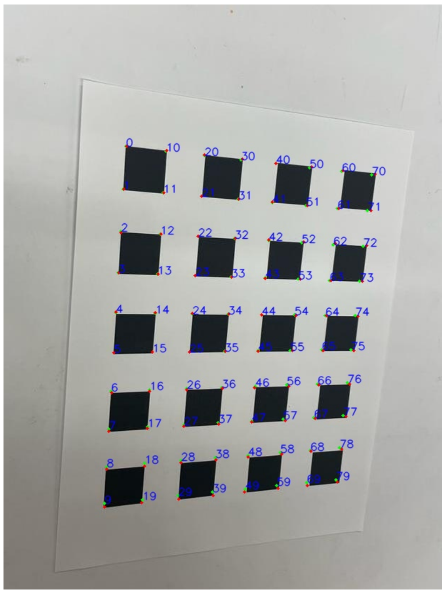


(b) Custom 2 Refined with LM Radial Distortion

Figure 15: Before and After Reprojection with LM and Radial Distortion Custom 2 (red: original ”ground truth”, green: reprojected)



(a) Custom 3 Before Refinement



(b) Custom 3 Refined with LM Radial Distortion

Figure 16: Before and After Reprojection with LM Radial Distortion Custom 3 (red: original ”ground truth”, green: reprojected)

5 Code Listings

5.1 Helper Functions

```
import os
import cv2
import numpy as np
from sklearn.cluster import KMeans
import sys

"""loadImages(dir_path)
Input: directory path
Output: list of grey scale images and labels
Purpose: given directory path, load images and labels"""
def loadImages(dir_path):
    raw_img_list = list()
    grey_img_list = list()
    img_labels = list()
    for filename in sorted(os.listdir(dir_path)):
        filepath = os.path.join(dir_path, filename)
        if os.path.isfile(filepath) and ('.jpg' in filepath or '.jpeg' in filepath):
            raw_img = cv2.imread(filepath)
            grey_img = cv2.cvtColor(raw_img, cv2.COLOR_BGR2GRAY)
            raw_img_list.append(raw_img)
            grey_img_list.append(grey_img)
            img_labels.append(filename)
    return raw_img_list, grey_img_list, img_labels


"""performCanny(grey_img_list)
Input: list of grey-scale images
Output: list of canny edge maps
Purpose: Given a list of grey scale images, apply canny on them"""
def performCanny(grey_img_list):
    edge_img_list = list()
    for img in grey_img_list:
        edge = cv2.Canny(img, 450, 450)
        edge_img_list.append(edge)
    return edge_img_list


"""performHoughTransform(edge_img_list)
Input: list of edge maps
Output: list of hough lines
Purpose: Given a list of edge maps, return a list of hough lines"""
def performHoughTransform(edge_img_list):
    hough_lines_list = list()
    for img in edge_img_list:
        line = cv2.HoughLines(img, 1, np.pi / 180, 60)
        hough_lines_list.append(line)
    return hough_lines_list


'''draw_hough_lines(line, img)
Input: line (list), img (np.ndarray)
Output: img (np.ndarray)
Purpose: Given a list of hough lines, draw them'''
def draw_hough_lines(line, img):
    for l in line:
        for rho, theta in l:
```

```python
            L = 1000
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 + L * (-b))
            y1 = int(y0 + L * (a))
            x2 = int(x0 - L * (-b))
            y2 = int(y0 - L * (a))
            cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 2)
    return img


"""get_Horizontal_Vert_Lines(lines)
Input: Hough lines for a single image as a list
Output: list of hor and vert lines
Purpose: separate hor and vert hough lines"""
def get_Horizontal_Vert_Lines(lines):
    h_lines = list()
    v_lines = list()
    for l in lines:
        for rho, theta in l:
            theta += -np.pi / 2
            if np.abs(theta) < np.pi / 4:
                h_lines.append(l)
            else:
                v_lines.append(l)
    return h_lines, v_lines


"""getCorners(v_lines, h_lines)
Input: horizontal and vertical lines as ndarrays
Output: list of 80 corners
Purpose: Given horizontal and vertical hough lines, find the corners"""
def getCorners(v_lines, h_lines):
    """y-intercept = horizontal line cross y-axis"""
    x_intercept = list()
    for i in range(v_lines.shape[0]):
        rho, theta = v_lines[i]
        x_intercept.append(np.divide(rho, np.cos(theta)))

    """y-intercept = horizontal line cross y-axis"""
    y_intercept = list()
    for i in range(h_lines.shape[0]):
        rho, theta = h_lines[i]
        y_intercept.append(np.divide(rho, np.sin(theta)))
    assert (len(x_intercept) == len(v_lines))
    assert (len(y_intercept) == len(h_lines))

    kmeans_v_lines = KMeans(n_clusters=8, random_state=0).fit(np.array(x_intercept).reshape(-1, 1))
    kmeans_h_lines = KMeans(n_clusters=10, random_state=0).fit(np.array(y_intercept).reshape(-1, 1))

    v_clustered_lines = list()
    h_clustered_lines = list()

    for i in range(8):
        v_clustered_lines.append(list(np.mean(v_lines[kmeans_v_lines.labels_ == i], axis=0)))
```

```python

    for i in range(10):
        h_clustered_lines.append(list(np.mean(h_lines[kmeans_h_lines.labels_ == i],
    axis=0)))

    v_lines_sorted = sorted(v_clustered_lines, key=lambda x: np.abs(x[0] / np.cos(x
    [1])))
    h_lines_sorted = sorted(h_clustered_lines, key=lambda x: np.abs(x[0] / np.sin(x
    [1])))

    corner_points = list()
    for v_line in v_lines_sorted:
        v_rho, v_theta = v_line
        v_HC = np.array([np.cos(v_theta), np.sin(v_theta), -v_rho])
        v_HC = v_HC / v_HC[-1]
        for h_line in h_lines_sorted:
            h_rho, h_theta = h_line
            h_HC = np.array([np.cos(h_theta), np.sin(h_theta), -h_rho])
            h_HC = h_HC / h_HC[-1]
            point = np.cross(h_HC, v_HC)
            # print(f'v_HC: {v_HC}')
            # print(f'h_HC: {h_HC}')
            # print(f'point: {point}')
            print('\n')
            if point[-1] == 0:
                continue
            point = point / point[-1]
            corner_points.append(tuple(point[:2].astype('int')))
    return corner_points


'''get_Ab(r2_points, projected_points)
Input: world points, corners as lists
Output: A, b matrices
Purpose: Given x and x' determne a and b'''
def get_Ab(r2_points, projected_points):
    A = list()
    for i, j in zip(r2_points, projected_points):
        r1 = i + [1] + [0, 0, 0] + [-i[0] * j[0], -i[1] * j[0]]
        r2 = [0, 0, 0] + i + [1] + [-i[0] * j[1], -i[1] * j[1]]
        A.append([r1, r2])
    b = np.array(projected_points).reshape(-1, 1)
    return np.array(A).reshape(-1, 8), b


'''get_H(world_points, corners)
Input: x and x'
Output: h
Purpose: Given x and x', find h'''
def get_H(world_points, corners):
    A, b = get_Ab(world_points, corners)
    H = list(np.linalg.solve(A.T @ A, A.T @ b).reshape(-1))
    H.append(1)
    return np.array(H).reshape(3, 3)


'''get_V(i,j,h)
Input: index i, j and homography h
Output: 6x1 matrix
Purpose: Given i, j, h, compute Vij'''
```

```
def get_V(i, j, h):
    v = np.zeros((6, 1))
    i -= 1
    j -= 1

    v[0][0] = h[0][i] * h[0][j]
    v[1][0] = (h[0][i] * h[1][j]) + (h[1][i] * h[0][j])
    v[2][0] = h[1][i] * h[1][j]
    v[3][0] = (h[2][i] * h[0][j]) + (h[0][i] * h[2][j])
    v[4][0] = (h[2][i] * h[1][j]) + (h[1][i] * h[2][j])
    v[5][0] = h[2][i] * h[2][j]

    return v


'''ReprojectPoints(img, world_coord, corner, k, r, t)
Input: img: raw colored image
       world_cord: list of world coords
       corners: list of identified corners
       k: intrinsic parameters
       r: rotation matrix
       t: translation vector
Output: img with points, mean error, var error
Purpose: Reproject world coords onto img'''
def ReprojectPoints(img, world_coord, Corners, K, R, t):
    X_hc = np.ones((len(world_coord), 3))
    X_hc[:, :-1] = np.array(world_coord)
    X_hc = X_hc.T
    P = np.concatenate((R[:, :2], t), axis=1)
    P = K @ P
    rep_pt_hc = P @ X_hc

    rep_pt_hc = rep_pt_hc / rep_pt_hc[-1]
    rep_pt = rep_pt_hc[0:2]
    e = np.array(Corners).T - rep_pt
    e = np.linalg.norm(e, axis=0)
    mean_e = np.mean(e)
    var_e = np.var(e)

    rep_img = np.copy(img)
    font = cv2.FONT_HERSHEY_SIMPLEX
    for i in range(len(world_coord)):
        rep_img = cv2.circle(img, (int(rep_pt[0, i]), int(rep_pt[1, i])), 2, (0, 255, 0), -1)
        rep_img = cv2.circle(img, (int(Corners[i][0]), int(Corners[i][1])), 2, (0, 0, 255), -1)
        rep_img = cv2.putText(img, str(i), (int(rep_pt[0, i]), int(rep_pt[1, i])), font, 0.5, (255, 0, 0), 1,
                              cv2.LINE_AA)
    return rep_img, mean_e, var_e


'''get_extrinsic(k,h)
Input: k: 3x3, h: 3x3
Output: R: 3x3, t: 3x1
Purpose: Given h and k (intrinsic/homo) compute extrinsic'''
def get_extrinsic(k, h):
    zeta = 1 / np.linalg.norm(np.linalg.inv(k) @ h[:, 0])

    r1 = zeta * np.linalg.inv(k) @ h[:, 0]
```

```
    r2 = zeta * np.linalg.inv(k) @ h[:, 1]
    r3 = zeta * np.cross(r1, r2)
    t = zeta * np.linalg.inv(k) @ h[:, 2]

    r1 = np.reshape(r1, (3, 1))
    r2 = np.reshape(r2, (3, 1))
    r3 = np.reshape(r3, (3, 1))
    t = np.reshape(t, (3, 1))

    R = np.hstack((r1, r2))
    R = np.hstack((R, r3))
    R = np.reshape(R, (3, 3))

    u, _, vh = np.linalg.svd(R)

    R = u @ vh

    return R, t


'''rotation2rod(R)
Input: 3x3 R rotation
Output: 3 vector rodriguez matrix
Purpose: Convert 9 dof to 3 dof rep of rotation matrix'''
def rotation2rod(R):
    phi = np.arccos((np.trace(R) - 1) / 2)
    w = (phi / (2 * np.sin(phi))) * np.array([(R[2, 1] - R[1, 2]),
                                              (R[0, 2] - R[2, 0]),
                                              (R[1, 0] - R[0, 1])])
    return (-w)


'''rod2rotation(w)
Input: 3 vector rodriguez matrix
Output: 3x3 R rotation
Purpose: Convert from 3 dof rep to 9 dof Rep'''
def rod2rotation(w):
    # make Wx from w
    Wx = np.array([[0, -1 * w[2], w[1]],
                   [w[2], 0, -1 * w[0]],
                   [-1 * w[1], w[0], 0]])
    phi = np.linalg.norm(w)
    R = np.eye(3) + (np.sin(phi) / phi) * (Wx) + ((1 - np.cos(phi)) / phi ** 2) * (Wx
     @ Wx)
    return (R)


'''cost_function_no_rad(p,x,x_m)
Input: p = [K,w1,t1,w2,t2,...wn,tn]
       x: list of list of corners for all images
       x_m: list of real world coordinates
Output: sum of square errors (scalar)
Purpose: cost function with no radial distortion'''
def cost_function_no_rad(p, x, x_m):
    # make K: intrinsic matrix
    a_x = p[0];
    a_y = p[1];
    s = p[2]
    x0 = p[3];
    y0 = p[4]
```

```
    K = np.array([[a_x, s, x0],
                  [0, a_y, y0],
                  [0, 0, 1]])

    num_img = int((len(p) - 5) / 6)
    N = len(x_m)
    cost = np.zeros(2 * num_img * N)
    for i in range(num_img):
        iw = p[6 * i + 5:6 * i + 8]
        it = p[6 * i + 8:6 * i + 11]
        iR = rod2rotation(iw)
        est_map = np.array([iR[:, 0].T, iR[:, 1].T, it.T])
        est_map = K @ (est_map.T)
        xij = np.array(x[i]);
        xij = xij.T
        x_m_hc = np.ones((len(x_m), 3));
        x_m_hc[:, :-1] = np.array(x_m)
        x_m_hc = x_m_hc.T
        x_hat_hc = est_map @ x_m_hc
        x_hat = np.linalg.inv(np.diag(x_hat_hc[-1, :])) @ x_hat_hc.T
        x_hat = x_hat.T
        x_hat = x_hat[:-1, :]
        temp = xij - x_hat
        cost[i * 2 * N:(i + 1) * 2 * N] = np.hstack((temp[0, :], temp[1, :]))
    return cost


'''cost_function_yes_rad
Input: p = [K,w1,t1,w2,t2,...wn,tn, k1,k2]
       x: list of list of corners for all images
       x_m: list of real world coordinates
Output: sum of square errors (scalar)
Purpose: cost function with radial distortion'''
def cost_function_yes_rad(p, x, x_m):
    a_x = p[0];
    a_y = p[1];
    s = p[2]
    x0 = p[3];
    y0 = p[4];
    k1 = p[-2];
    k2 = p[-1]
    K = np.array([[a_x, s, x0],
                  [0, a_y, y0],
                  [0, 0, 1]])
    num_img = int((len(p) - 7) / 6)
    N = len(x_m)
    cost = np.zeros(2 * num_img * N)
    for i in range(num_img):
        iw = p[6 * i + 5:6 * i + 8]
        it = p[6 * i + 8:6 * i + 11]
        iR = rod2rotation(iw)
        est_map = np.array([iR[:, 0].T, iR[:, 1].T, it.T])
        est_map = K @ est_map.T
        xij = np.array(x[i])
        xij = xij.T
        x_m_hc = np.ones((len(x_m), 3));
        x_m_hc[:, :-1] = np.array(x_m)
        x_m_hc = x_m_hc.T
        x_hat_hc = est_map @ x_m_hc
        x_hat = np.linalg.inv(np.diag(x_hat_hc[-1, :])) @ x_hat_hc.T
```

```
            x_hat = x_hat.T
            x_hat = x_hat[:-1, :]
            diff = x_hat - (np.kron(np.array([x0, y0]), np.ones((N, 1)))).T
            r_2 = np.sum(np.square(diff), axis=0)
            m = k1 * r_2 + k2 * np.square(r_2)
            m = np.vstack((m, m))
            x_hat_rad = x_hat + np.multiply(m, diff)
            temp = xij - x_hat_rad
            cost[i * 2 * N:(i + 1) * 2 * N] = np.hstack((temp[0, :], temp[1, :]))
        return cost
```

camera_callibration_helper.py

5.2 Driver

```
#!/usr/bin/env python
# coding: utf-8

# # Zhang's Algorithm For Camera Calibration

# ### Import Statements

# In[16]:


from camera_callibration_helper import *
import cv2
import numpy as np
from copy import deepcopy
from scipy.optimize import least_squares
import warnings
warnings.filterwarnings('ignore')


# ### Load the Images
# * raw_img_list (list): list of 40 BGR input images
# * grey_img_list (list): list of 40 grey scale input images
# * img_labels (list): list of 40 image filenames (mainly for debugging)

# In[17]:


# given_data_path = 'C:\\Users\jo_wang\Desktop\ECE661\HW08\Dataset1'
#given_data_path = "/Users/wang3450/Desktop/ECE661/HW08/Dataset1"

# given_data_path = "/home/jo_wang/Desktop/ECE661/HW08/Dataset1"
given_data_path = "/home/jo_wang/Desktop/ECE661/HW08/Dataset2"
raw_img_list, grey_img_list, img_labels = loadImages(given_data_path)
assert(len(grey_img_list) == 4)
assert(len(raw_img_list) == 4)
assert(len(img_labels) == 4)

# x = img_labels.index('Pic_1.jpg')
# y = img_labels.index('Pic_5.jpg')
# z = img_labels.index('Pic_10.jpg')
# w = img_labels.index('Pic_34.jpg')
#
# print(x,y,z,w)
```

```python


# ### Apply Canny Edge Detector On Grey Scale Images
# * edge_img_list (list): list of edge maps from Canny

# In[18]:


edge_img_list = performCanny(grey_img_list)
assert(len(edge_img_list) == 4)
cv2.imwrite('canny_custom1.jpg', edge_img_list[0])
cv2.imwrite('canny_custom2.jpg', edge_img_list[1])
cv2.imwrite('canny_custom3.jpg', edge_img_list[2])
cv2.imwrite('canny_custom4.jpg', edge_img_list[3])


# ### Apply Hough Transform To all the Images
# * hough_lines_list (list): list of 40 images after applying hough transform

# In[19]:


hough_lines_list = performHoughTransform(edge_img_list)
assert(len(hough_lines_list) == len(edge_img_list))

cv2.imwrite('hough_lines_custom1.jpg', draw_hough_lines(hough_lines_list[0], deepcopy
    (raw_img_list[0])))
cv2.imwrite('hough_lines_custom2.jpg', draw_hough_lines(hough_lines_list[1], deepcopy
    (raw_img_list[1])))
cv2.imwrite('hough_lines_custom3.jpg', draw_hough_lines(hough_lines_list[2], deepcopy
    (raw_img_list[2])))
cv2.imwrite('hough_lines_custom4.jpg', draw_hough_lines(hough_lines_list[3], deepcopy
    (raw_img_list[3])))


# ### Get the corner points from selected images
# * all_corners (list): at each index, list of 80 corner points
# * the_chosen_one (list): index of images to use

# In[20]:


# the_chosen_one = [0, 35, 1, 27]
the_chosen_one = [0, 1, 2, 3]


all_corners = list()
for i in the_chosen_one:
    h_lines, v_lines = get_Horizontal_Vert_Lines(hough_lines_list[i])

    v_lines = np.array(v_lines).reshape(-1,2)
    h_lines = np.array(h_lines).reshape(-1,2)

    img = deepcopy(raw_img_list[i])
    corner_points = getCorners(v_lines, h_lines)
    if len(corner_points) == 80:
        all_corners.append(corner_points)

    for j, point in enumerate(corner_points):
        try:
```

```python
            img = cv2.circle(img, point, 3, (0, 0, 255), -1)
            cv2.putText(img, str(j), (point[0]+5, point[1]-5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0), 1)
        except OverflowError:
            pass

    cv2.imwrite(f'points_{i+1}.jpg', img)


# ### Get world point coordinates
# * world_points (list): list of 80 world point coordinates in sorted order
# * Assumption made: squares are 20 pixels apart

# In[21]:


world_points = list()
for i in range(0, 160, 20):
    for j in range(0, 200, 20):
        world_points.append([i,j])


# ### Estimate Homographies between world points and all corners
# * all_homographies (list): list of 3x3 homographies relating world points to each image
# * DON'T DELETE THIS ONE CUZ IT WORKS FOR NOW!!!!!!

# In[22]:


all_homographies = list()
for corners in all_corners:
    h = get_H(world_points, corners)
    all_homographies.append(h)


# ### Compute W
# * W is a 3x3 matrix
# * Derived from the solution of Vb = 0
# * Use svd to solve Vb=0

# In[23]:


Big_V = np.zeros((1,6))
for h in all_homographies:
    r1 = get_V(i=1, j=2, h=h).T
    r2 = get_V(i=1,j=1,h=h).T - get_V(i=2,j=2,h=h).T
    Big_V = np.vstack((Big_V, r1))
    Big_V = np.vstack((Big_V, r2))

Big_V = Big_V[1:, :]

u, s, vh = np.linalg.svd(Big_V)
b = vh[-1]

w = np.zeros((3,3))
w[0][0] = b[0]
w[0][1] = b[1]
w[0][2] = b[3]
```

```
w[1][0] = b[1]
w[1][1] = b[2]
w[1][2] = b[4]
w[2][0] = b[3]
w[2][1] = b[4]
w[2][2] = b[5]


#

# ### Compute Intrinsic Camera Parameters Matrix k
# * k is 3x3 matrix
# * k is based on y0, a_x, a_y, skew, x0, lambda
#

# In[24]:


y0 = ((w[0][1] * w[0][2]) - (w[0][0] * w[1][2])) / (w[0][0] * w[1][1] - w[0][1] ** 2)
scale_lambda = w[2][2] - (w[0][2] ** 2 + y0 * (w[0][1] * w[0][2] - w[0][0] * w[1][2])
    ) / w[0][0]
a_x = np.sqrt(np.abs((scale_lambda / w[0][0])))
a_y = np.sqrt(np.abs((scale_lambda * w[0][0]) / (w[0][0] * w[1][1] - w[0][1] **2)))
skew = (-1 * w[0][1] * (a_x ** 2) * a_y) / scale_lambda
x0 = (skew * y0) / a_y - (w[0][2] * (a_x ** 2)) / scale_lambda

k = np.zeros((3,3))
k[0][0] = a_x
k[0][1] = skew
k[0][2] = x0
k[1][1] = a_y
k[1][2] = y0
k[2][2] = 1

print(k)


# ### Compute Extrinsic Parameters

# In[25]:


all_rotations = list()
all_translations = list()

for homographies in all_homographies:
    R, t = get_extrinsic(k, homographies)
    all_rotations.append(R)
    all_translations.append(t)

print(len(all_rotations))
print(len(all_translations))
assert(len(all_rotations) == len(all_translations))
assert(len(all_rotations) == len(the_chosen_one))

print("Pic 1")
print(f'Rotation Matrix: \n{all_rotations[0]}')
print(f'Translation Matrix: \n {all_translations[0]}')
print("\n")
print("Pic 5")
```

```
print(f'Rotation Matrix: \n{all_rotations[1]}')
print(f'Translation Matrix: \n {all_translations[1]}')
print("\n")
print("Pic 10")
print(f'Rotation Matrix: \n{all_rotations[2]}')
print(f'Translation Matrix: \n {all_translations[2]}')
print("\n")
print("Pic 34")
print(f'Rotation Matrix: \n{all_rotations[3]}')
print(f'Translation Matrix: \n {all_translations[3]}')
print("\n")


# ### Reproject the World Coordinates

# In[26]:


#the_chosen_one = [0, 35, 1, 27]
corner0 = [list(i) for i in all_corners[0]]
corner1 = [list(i) for i in all_corners[1]]
corner2 = [list(i) for i in all_corners[2]]
corner3 = [list(i) for i in all_corners[3]]

all_corners_list = [corner0, corner1, corner2, corner3]

rep_img0, rep_img0_mean_e, rep_img0_var_e = ReprojectPoints(deepcopy(raw_img_list[0])
    ,world_points,corner0,k,all_rotations[0],all_translations[0])

rep_img1, rep_img1_mean_e, rep_img1_var_e = ReprojectPoints(deepcopy(raw_img_list[1])
    ,world_points,corner1,k,all_rotations[1],all_translations[1])

rep_img2, rep_img2_mean_e, rep_img2_var_e = ReprojectPoints(deepcopy(raw_img_list[2])
    ,world_points,corner2,k,all_rotations[2],all_translations[2])

rep_img3, rep_img3_mean_e, rep_img3_var_e = ReprojectPoints(deepcopy(raw_img_list[3])
    ,world_points,corner3,k,all_rotations[3],all_translations[3])

cv2.imwrite('rep_custom1.jpg', rep_img0)
cv2.imwrite('rep_custom2.jpg', rep_img1)
cv2.imwrite('rep_custom3.jpg', rep_img2)
cv2.imwrite('rep_custom4.jpg', rep_img3)

print('Pic #     Mean Error              Error Variance')
print(f'Pic_1     {rep_img0_mean_e}       {rep_img0_var_e}')
print(f'Pic_5     {rep_img1_mean_e}       {rep_img1_var_e}')
print(f'Pic_10    {rep_img2_mean_e}       {rep_img2_var_e}')
print(f'Pic_34    {rep_img3_mean_e}       {rep_img3_var_e}')


# ### Refinement of Calibration Parameters

# 1). Prepare p0 depending on whether we want to consider radial distortion
# 2). Express R as rodriguez form
# 3). Resize translations (3,1) -> (3,)
#
# p0 is constituted by the intrinsic and extrinsic parameters
# * pack k = [a_x, a_y, s, x0, y0] into first 5 index of p
# * pack the linear least squares estimated rotational and translational matrices for
    each view thereafter
```

```
# In[27]:


rodrigues_rotation = list()
for R in all_rotations:
    rodrigues_rotation.append(rotation2rod(R))

translations_for_refine = [np.resize(translation, (3,)) for translation in
    all_translations]

'''Create p0 to be optimized (no radial distortion)'''
rad_dist = False
if rad_dist:
    k1,k2 = np.zeros(2)
    p0=np.zeros(7+6*len(the_chosen_one))
    p0[:5]=np.array([a_x,a_y,skew,x0,y0])
    for i in range(len(the_chosen_one)):
        p0[6*i+5:6*i+8]=rodrigues_rotation[i]
        p0[6*i+8:6*i+11]=translations_for_refine[i]
    p0[-2]=k1;  p0[-1]=k2
else:
    p0=np.zeros(5+6*len(the_chosen_one))
    p0[:5]=np.array([a_x,a_y,skew,x0,y0])
    for i in range(len(the_chosen_one)):
        p0[6*i+5:6*i+8]=rodrigues_rotation[i]
        p0[6*i+8:6*i+11]=translations_for_refine[i]


# Call the optimizer with:
#     * cost_function
#     * parameter to be optimized (p0)
#     * method = "lm"
#     * args = (all_corners_list, world_point)
#
# Note: all_corners_list = [corners0, corners1, corners,2]
# where [cornersX] = [[x1,y1], [x2,y2], ..., [xn,yn]]
#
# Optimum p_star = optim['x']
# p_star is same shape as p0

# In[28]:


if rad_dist:
    optim=least_squares(cost_function_yes_rad,p0,method='lm',args=(all_corners_list,
    world_points))
else:
    optim=least_squares(cost_function_no_rad,p0, method='lm',args=(all_corners_list,
    world_points))

p_star=optim['x']


# Unpack the intrinsic and extrinsic parameters from p_star
# * k = [a_x, a_y, s, x0, y0] located in first 5 indexes of p_star
# * unpack the refined rotational and translational matrices for each view.

# In[29]:

```

```

a_x=p_star[0]
a_y=p_star[1]
skew=p_star[2]
x0=p_star[3]
y0=p_star[4]

K_ref = np.zeros((3,3))
K_ref[0][0] = a_x
K_ref[0][1] = skew
K_ref[0][2] = x0
K_ref[1][1] = a_y
K_ref[1][2] = y0
K_ref[2][2] = 1

if rad_dist:
    k1=p_star[-2]; k2=p_star[-1]
    print('Radial Distortion parameters: k1='+str(k1)+' k2='+str(k2))

R_ref=[]
t_ref=[]
for i in range(len(the_chosen_one)):
    iw=p_star[6*i+5:6*i+8]
    it=p_star[6*i+8:6*i+11]
    iR=rod2rotation(iw)
    R_ref.append(iR)
    t_ref.append(it)


# In[30]:


t_ref[0] = np.reshape(t_ref[0], (3,1))
t_ref[1] = np.reshape(t_ref[1], (3,1))
t_ref[2] = np.reshape(t_ref[2], (3,1))
t_ref[3] = np.reshape(t_ref[3], (3,1))
refine_img0, refine_img0_mean_e, refine_img0_var_e = ReprojectPoints(raw_img_list[0],
    world_points,corner0,K_ref,R_ref[0],t_ref[0])
refine_img1, refine_img1_mean_e, refine_img1_var_e = ReprojectPoints(raw_img_list[1],
    world_points,corner1,K_ref,R_ref[1],t_ref[1])
refine_img2, refine_img2_mean_e, refine_img2_var_e = ReprojectPoints(raw_img_list[2],
    world_points,corner2,K_ref,R_ref[2],t_ref[2])
refine_img3, refine_img3_mean_e, refine_img3_var_e = ReprojectPoints(raw_img_list[3],
    world_points,corner3,K_ref,R_ref[3],t_ref[3])

cv2.imwrite('refine_no_rad_pic1.jpg', refine_img0)
cv2.imwrite('refine_no_rad_pic5.jpg', refine_img1)
cv2.imwrite('refine_no_rad_pic10.jpg', refine_img2)
cv2.imwrite('refine_no_rad_pic34.jpg', refine_img3)

print('Pic #     Mean Error              Error Variance')
print(f'Pic_1     {refine_img0_mean_e}      {refine_img0_var_e}')
print(f'Pic_5     {refine_img1_mean_e}      {refine_img1_var_e}')
print(f'Pic_10    {refine_img2_mean_e}      {refine_img2_var_e}')
print(f'Pic_34    {refine_img3_mean_e}      {refine_img3_var_e}')
```

camera_callibration.py