Purdue University

ECE 661 Computer Vision

Homework 9

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Theory Question

From the lecture we got to know that the image of the absolute conic is given by:

$$\omega = K^{-T}K^{-1} \tag{1}$$

The question posed to us is: Can we actually see this on the image? The answer is both yes and no.
So why is it a no?
The only reason we are able to justify the above equation is because we are acknowledging the fact that the absolute conic lies on the plan at infinity π_∞. Thus it is evident that this is just an imaginary concept which cannot be represented with real points on an image. The absolute conic depends only on the intrinsic camera parameters and not anyhow on any of the camera orientations or camera positions. It is a concept which is useful for the computation of the various camera parameters and has no real representation.
But how is it also a yes?
Take an arbitrary plane π which intersects the plane at infinity π_∞. Now, we know that this intersection will be along a line. From the lecture we know that this line will also intersect the absolute conic. It intersects the absolute conic at two points which are the circular points of the initial plane under consideration. Now, the image of these two circular points (which form a line) are the intersection points of the vanishing lines of the plane π and ω

Programming Tasks

In this homework we implement the popular Zhang's algorithm for camera calibration. Hence, by assuming that our camera is a pin hole camera, we will be calibrating it by extracting the **5 intrinsic and 6 extrinsic** parameters. In order to achieve this, we will be implementing the following sub tasks in the code:

- Corner detection
- Camera calibration using Zhang's algorithm
- Calibration refinement using LM algorithm for non-linear optimization.
- Conditioning the rotation matrix
- Re-projecting onto the initial fixed image.
- Estimation and incorporation of radial distortion

Let us briefly discuss these topics in the same order.

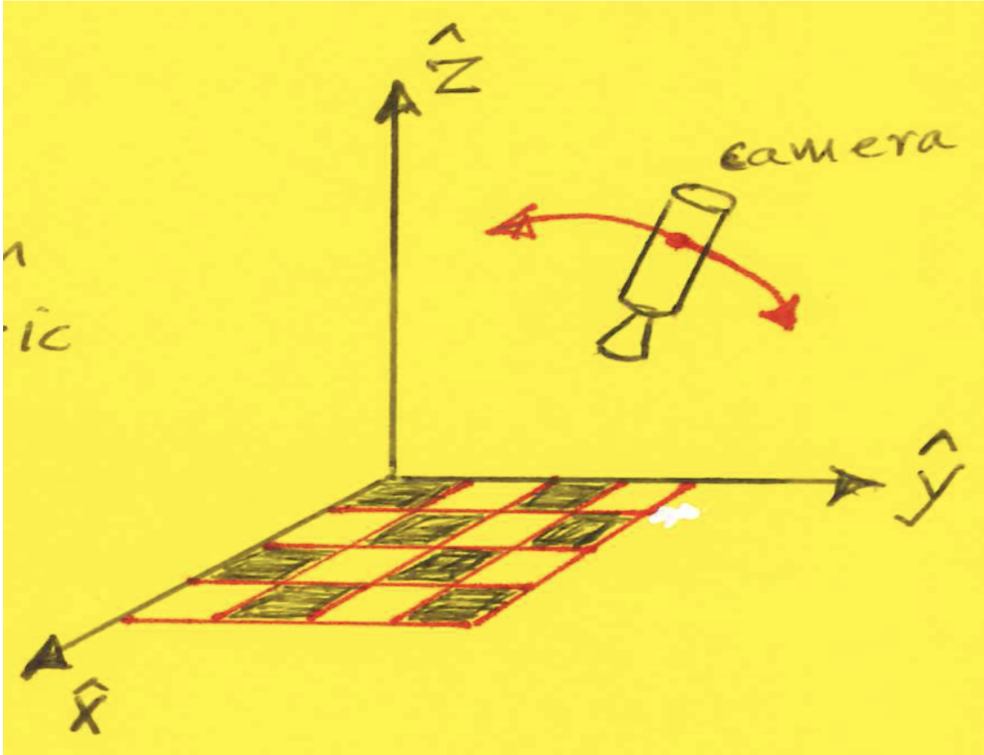


Figure 1: **Picture source: Dr. Avi Kak's lecture scroll - Lec 19**

Before we dive into the detailed explanation, take a look at the above picture. This is the set up we use for the calibration. We begin by taking a checkerboard pattern (usually physically printed). We **take for granted** that this pattern exactly lies on the Z=0 plane as you can see in the same figure. We take pictures of the pattern from different angles. The idea is to estimate the right correspondence between the camera coordinates and the world coordinates. If that is achieved properly then we are bound to have accurate intrinsic and extrinsic camera parameters.

1. Corner detection

This is arguably one of the most critical tasks of this whole exercise. Because this is the task which will be dictating how the correspondence estimation will come about in the next step. Estimating accurate correspondence is central for good camera calibration. Basically, we need a way to establish or estimate the relationship (or correspondence) between the projected camera point and the world coordinate. To this end, we will be extracting the corners from the given image and then establishing the correspondence which map the corners to the world coordinates.
The basic steps involved in this stage are:

1. Extract corners using the Canny edge detector from existing libraries: We do this by first converting the image into a grey scale image. On top of this we run a Gaussian blurring operator. Once that is done, we run the in built canny edge operator to extract the edges from the image.

2. Fit straight lines to the edges using the Hugh transform: From the edges we detected in step one, we try to fit perfectly straight lines on the edges that we detected.

3. Suppress false lines: In step two, there is almost no chance that a single line is detected multiple times. We obviously do not want this scenario. Therefore, we employ a compact checking algorithm to weed out the false line detection. It turns out that the method employed is relatively very simple. We sort the detected lines in a specific order. Subsequently, for every successive line we check the distance between the lines. If this distance is less than a certain pre-defined threshold value, the line on the left is selected and the other line is discarded.

4. Corner localization: Once we have the final straight lines drawn out, it is evident that the intersection of any two lines will give us one of the corners on the pattern. Due to radial distortion, the positions of these corners tend to be distorted as well. To overcome this, we apply a sub pixel refinement algorithm to get accurate localised corner locations. To make it watertight, we will label these corner points with indices formed by a specific naming order.

2. Camera calibration

Since calibration is essentially about estimating the correspondences between the camera world and the real world, we logically begin by estimating the homographies between the extracted corners and edges with the real world coordinates.
By world coordinates we mean the actual physical dimensions of the respective pattern. So, using a ruler, we can measure the distance between each of the squares in the checkerboard pattern.
Once we have the representations of both the worlds, we can begin estimating the homographies between them. Basically, our initial goal is to have a relationship which defines the mapping in very simple terms. Therefore, our job is to find a value for the homography which fits in the equation:

$$Point_{image} = H * Point_{world} \tag{2}$$

We then make use of the amazing property of the absolute conic Ω_∞ being truly independent of the extrinsic properties **R** and **vector t**. The absolute conic is given by:

$$\omega = K^{-T}K^{-1} \tag{3}$$

We know from the lecture that any plane **P** samples the absolute conic at exactly two points. These points being the two circular points of the plane **P**. So each of the two points have to satisfy the equation:

$$\vec{x}^T\omega\vec{x} = 0 \tag{4}$$

Therefore, we can write the two subsequent equation as:

$$\vec{h}_1^T\omega\vec{h}_1 - \vec{h}_2^T\omega\vec{h}_2 = 0 \tag{5}$$

and

$$\vec{h}_1^T\omega\vec{h}_2 = 0 \tag{6}$$

Now consider a vector which is a vector of unknowns represented as:

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

and

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

Now, we see that equations 4 and 5 can be written as:

$$(\vec{V_{11}} - \vec{V_{22}})^T \vec{b} = 0 \tag{7}$$

and

$$\vec{V_{12}}^T \vec{b} = 0 \tag{8}$$

If we view equation 6 in the form:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \end{bmatrix} \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{21} & \omega_{22} & \omega_{23} \\ \omega_{31} & \omega_{32} & \omega_{33} \end{bmatrix} \begin{bmatrix} h_{21} \\ h_{22} \\ h_{23} \end{bmatrix} = 0 \tag{9}$$

We begin to see that the best way to solve this is by solving a system of linear equations. We do this by stacking equation 8 in the 2x6 matrix form to finally get:

$$\begin{bmatrix} \vec{V_{12}}^T \\ (\vec{V_{11}} - \vec{V_{22}})^T \end{bmatrix} \vec{b} = \vec{0} \tag{10}$$

We solve equation 9 to get the which is the matrix of unknowns. Later, we use this to find the ω which is the image conic.

Intrinsic parameters

Now the question is how do we use the ω value to estimate the intrinsic and extrinsic parameters of the camera. We know that the intrinsic property of the camera which is given by **K** is represented as:

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

Good for us, the intrinsic parameter is related to the conic in the way as shown by equation 2. Using this equation, we use the omega value to determine each of the values in the camera intrinsic parameter matrix. The individual equations are:

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

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

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}} \tag{13}$$

$$\alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \tag{14}$$

$$s = -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \tag{15}$$

$$y_0 = \frac{sx_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda} \tag{16}$$

Extrinsic parameters
The next task would be to estimate the extrinsic parameters of the camera. The extrinsic parameters being the parameter **R** and **T**. Parameter R stands for the rotational matrix which maps the rotational adjustment of the camera center with respect to the world origin. Parameter T stands for the translational matrix which maps the translational adjustment of the camera center towards the world origin. Remember that we still have the pattern on the plane Z = 0. We find paramter R by using the following relation:

$$K^{-1}\vec{H} = \vec{R} \tag{17}$$

That is, we do the following:

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

We then obtain the final R value by doing a single value decomposition of the R matrix. We then find the second parameter which is the parameter T by using the equation:

$$K^{-1}\vec{h_3} = \vec{T} \tag{18}$$

Therefore, we now have estimated both the intrinsic and the extrinsic parameters of the camera. Now, we need to refine these parameters to increase the reliability and the accuracy of the camera projections.

3. Rotation matrix conditioning

In order to compute the extrinsic parameters, we need to ensure that the R matrix which is the rotation matrix is orthonormal. The procedure for this is:

- Perform a Frobenius norm minimization using the following relation:

$$||R - P||_F^2$$

- Wherein we get the value of P using a single value decomposition exercise:

$$P = UDV^T$$

- We set the value of R using the relation:

$$R = UV^T$$

4. Parameter Refinement

Since we estimated the intrinsic and extrinsic parameters through simplified linear equations, it is evident that the final estimations will be affected by the various non linear noises. Therefore, we need a final non-linear optimization to refine the estimated parameters. This is needed to get acceptable results as we will be seeing later on in the report.
To refine the estimations, we will be using the Levenberg-Marquardt algorithm for the non-linear optimization. The cost function at the center of this optimization exercise is:

$$(cost)^2_{geometric} = \sum_i \sum_j ||\vec{x}_{ij} - K[R_i|t_i]\vec{x}_{nj}||^2 \tag{19}$$

The basic idea is that we calculate the eucledian error or the distance between the actual camera pixel point and the projected pixel point. We use this as the cost function which is used to refine the estimations. In equation 12, $\vec{x}_{ij}$ is the true pixel point on the camera. $\vec{x}_{nj}$ is the jth salient point on the pattern which sits on Z=0 plane. i stands for the ith position of the camera. j stands for the jth point on the pattern. Additionally, we need to represent the rotation matrix from its nine parameter form to the three parameter i,e a matrix with three degrees of freedom. We do that by transforming the R matrix into its Rodrigues representation. The procedure for this is:

$$\vec{\omega} = \frac{\phi}{2\sin\phi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{20}$$

where:

$$\phi = \cos^{-1}\frac{trace(R) - 1}{2} \tag{21}$$

In order to maintain orthonormality of R we will need to convert these representations back from the Rodrigues representation. We do this by:

$$R = I + \frac{\sin\phi}{\phi}[\vec{\omega}]_x + \frac{1 - \cos\phi}{\phi^2}[\vec{\omega}]_x^2 \tag{22}$$

5. Incorporating radial distortion

We still have one final refinement required to ultimately calibrate the camera. Since, we are assuming that our camera model is a pin hole type camera, we will have to acknowledge that this kind of a camera performs very poorly for short focal-lengths. There is a lot of radial distortion that gets introduced into the image. We need to incorporate this into our calibration parameters to ensure higher accuracy. To do this, we use the following three relations:

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

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

where,

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

The individual variables are:

- x_{rad} and y_{rad} are the predicted pixel points with the radial distortion incorporated.
- x and y are the predicted pixel points without incorporating the radial distortion.
- k_1 and k_2 are the tuning values which will be tuned using an LM refinement algorithm.

6. Reprojecting onto the fixed image

This step if more for the visualization of our estimations until now. This step serves as a visual validation step to verify the effectiveness of our camera calibration.
The fundamental idea here is to retrace our steps and project an image onto the fixed reference image. We do this by using the camera parameters to predict the points.

For wholesomeness, we re-project the images using the homographies before and after the LM refinement. The homography is given by:

$$\vec{x}_{image} = H_{camera}\vec{x}_{world}$$

Where the homography is given by:

$$H_{camera} = K[r_1 r_2 t]$$

So, using this homography relation, we re-project the image onto the reference fixed image. The reprojection is done using the following relation:

$$P_{rep} = H_{ref} * H_{img}^{-1} * P_{img} \tag{26}$$

Where:

- P_{rep} is predicted point which will be re-projected onto the reference image.
- H_{ref} and H_{img} are the homographies of the reference images and the given testing image respectively.
- P_{img} is the point under consideration on the testing image.

7. Result & Analysis

Original Data set

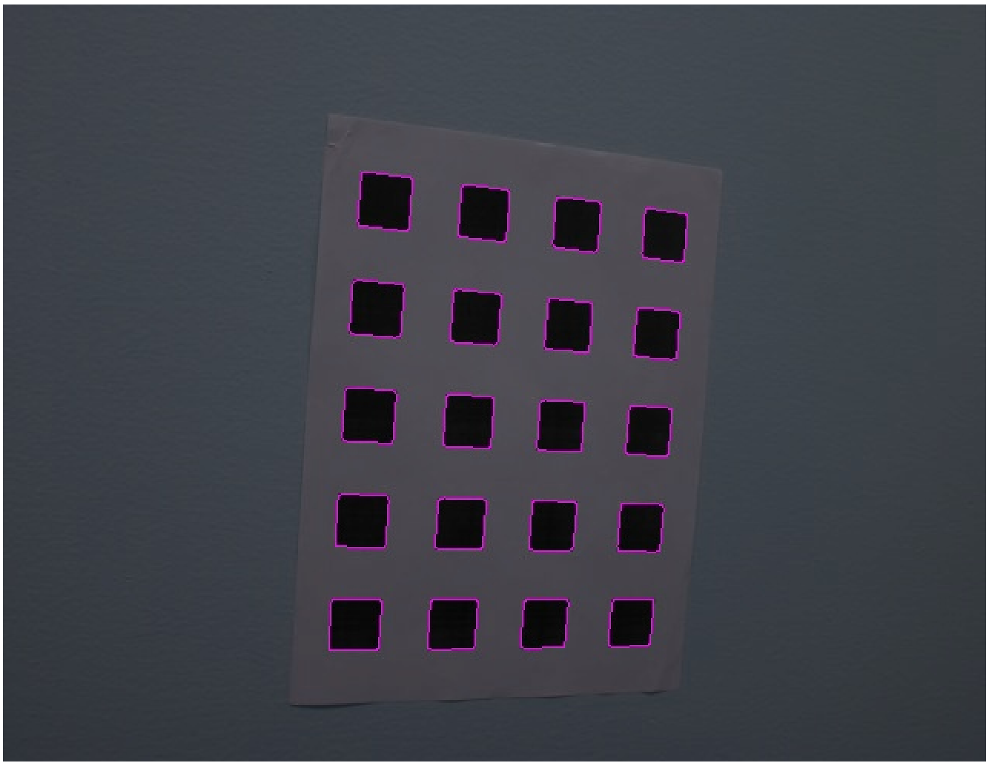


Figure 2: **Canny edges for image 34**

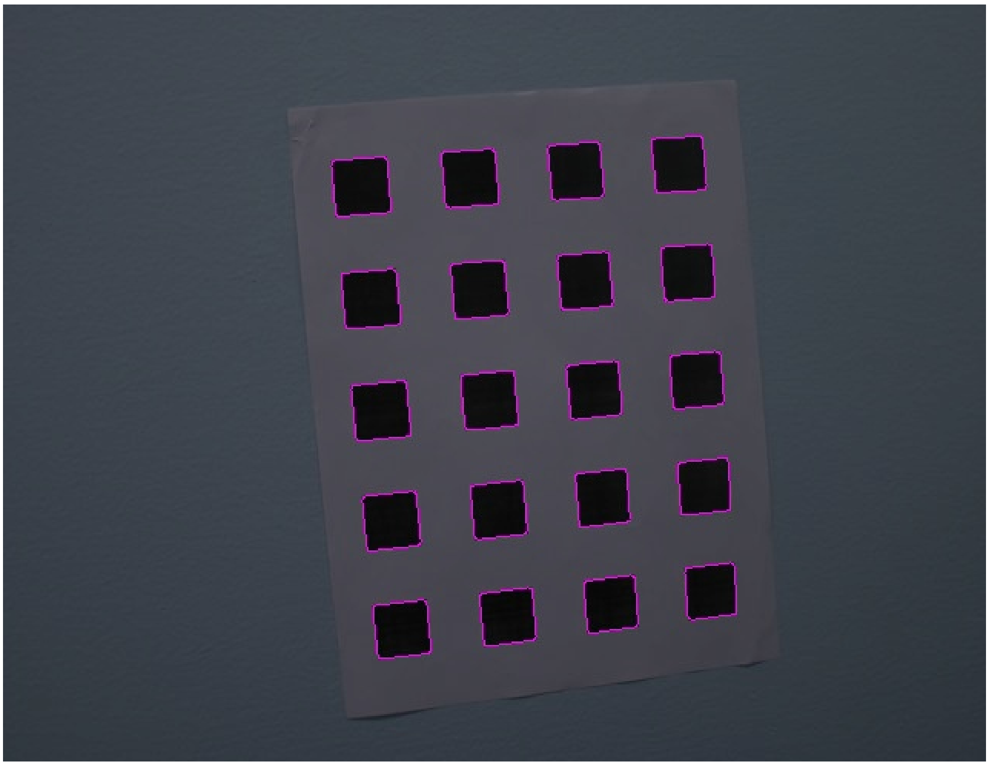


Figure 3: **Canny edges for image 35**

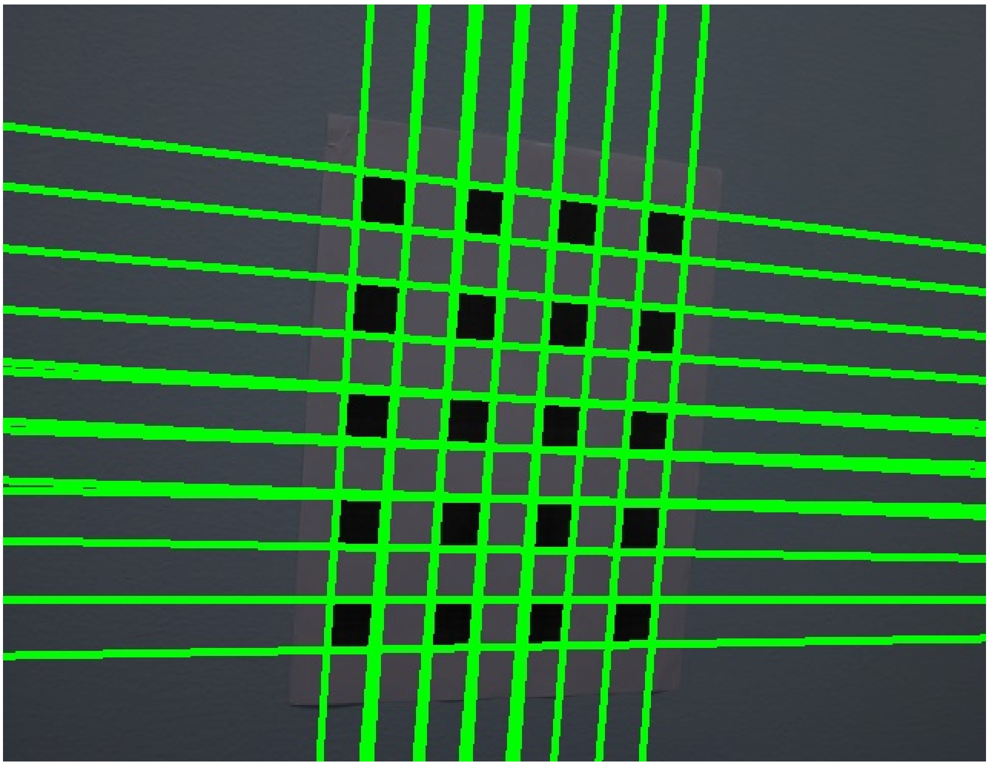


Figure 4: **All the detected Hough lines for image 34**

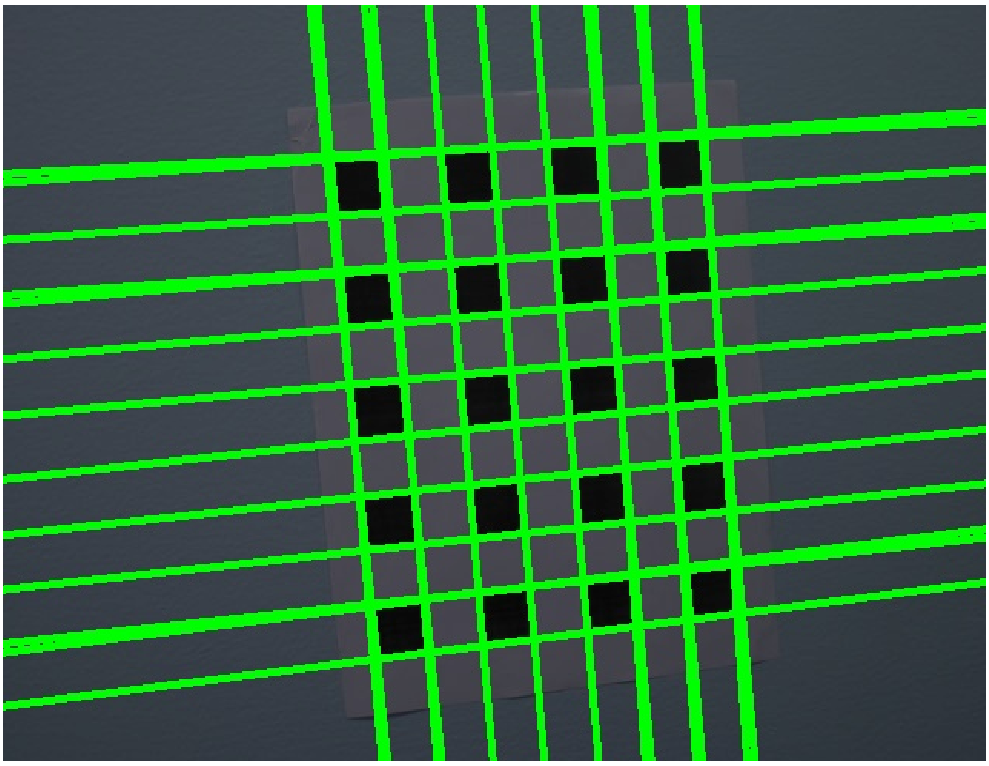


Figure 5: **All the detected Hough lines for image 35**

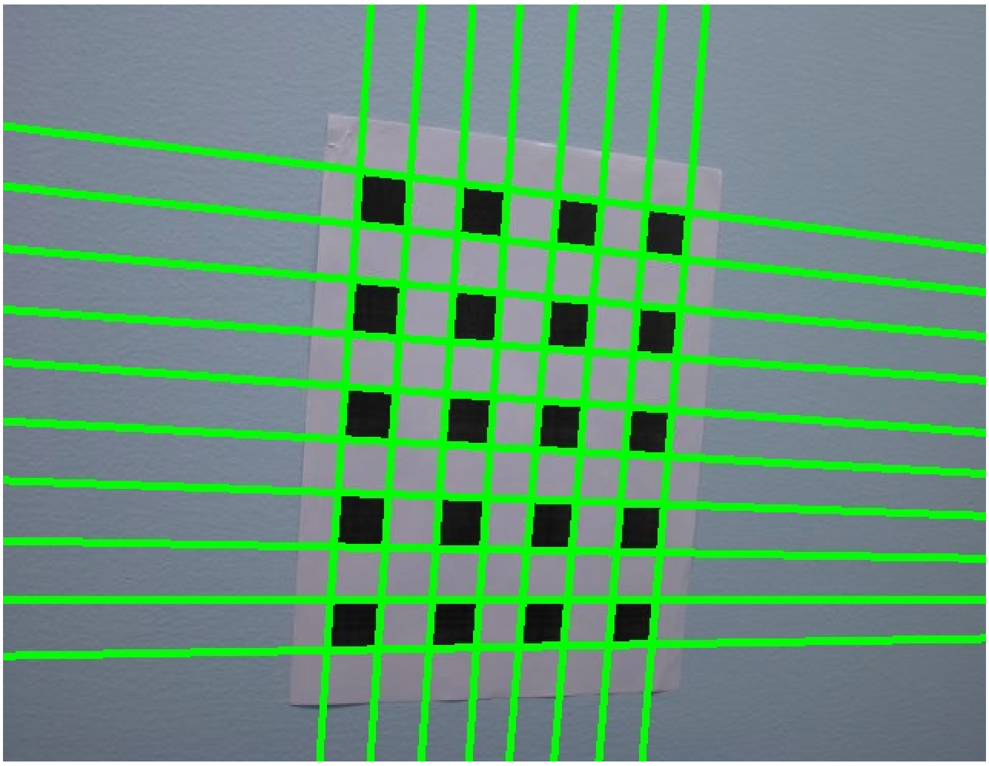


Figure 6: **Final Hough lines for image 34**

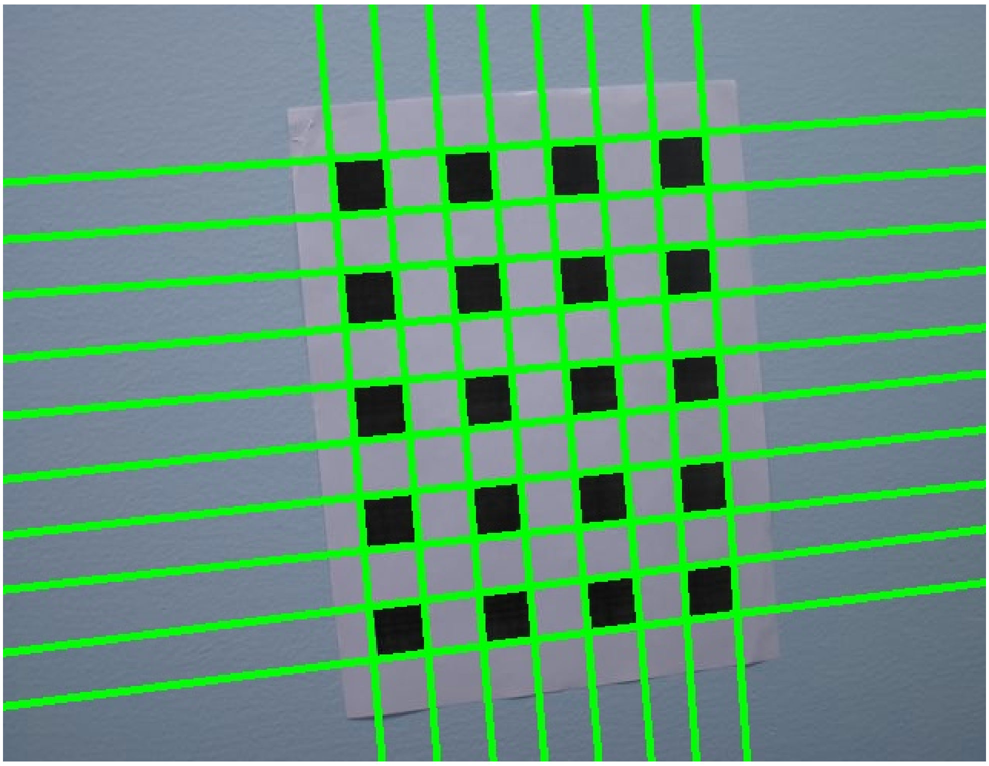


Figure 7: **Final Hough lines for image 35**

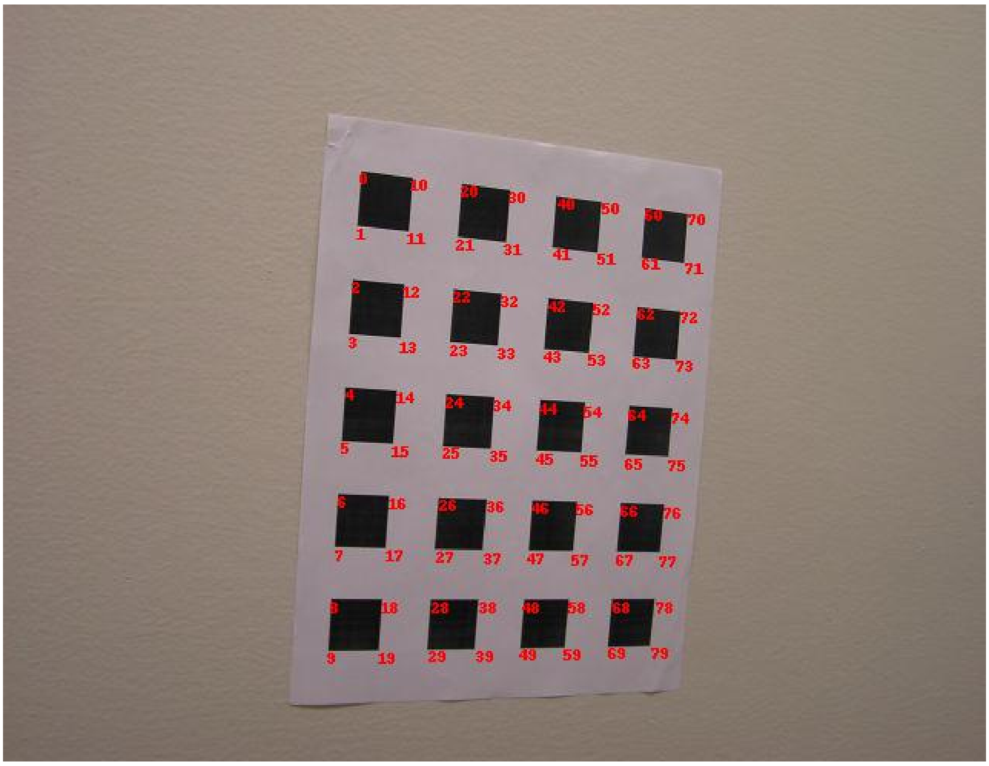


Figure 8: **Enumerated corners for image 34**

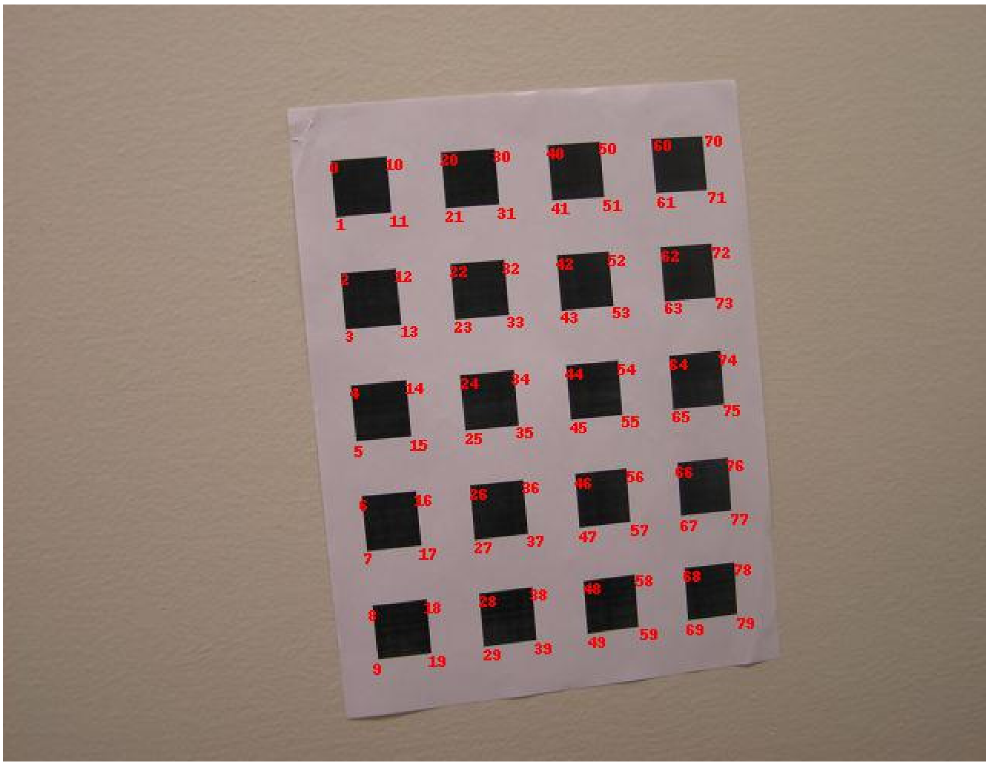


Figure 9: **Enumerated for image 35**

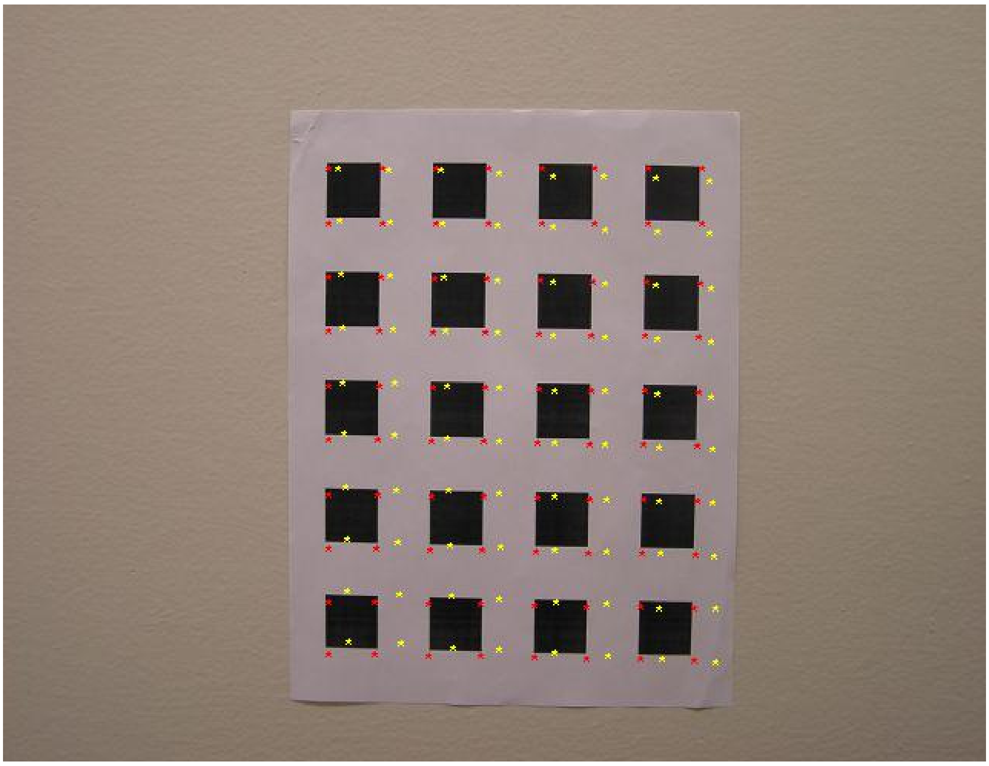


Figure 10: **Reprojection Before LM refine - Image 1 onto Reference image 11**

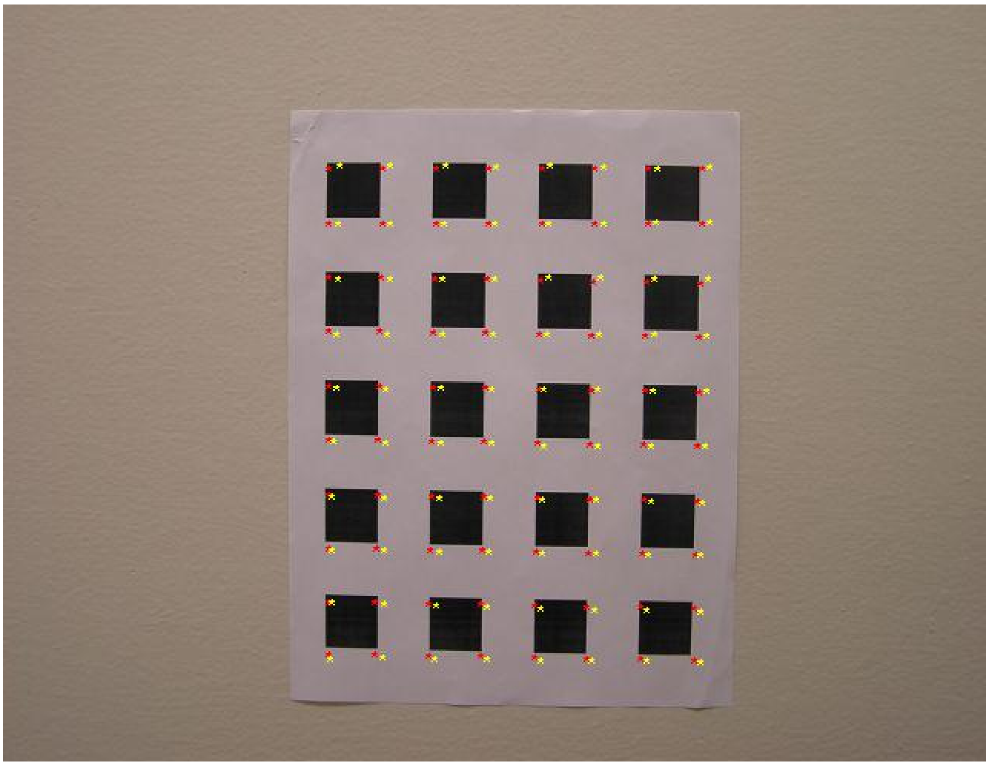


Figure 11: **Reprojection Before LM refine - Image 6 onto Reference image 11**

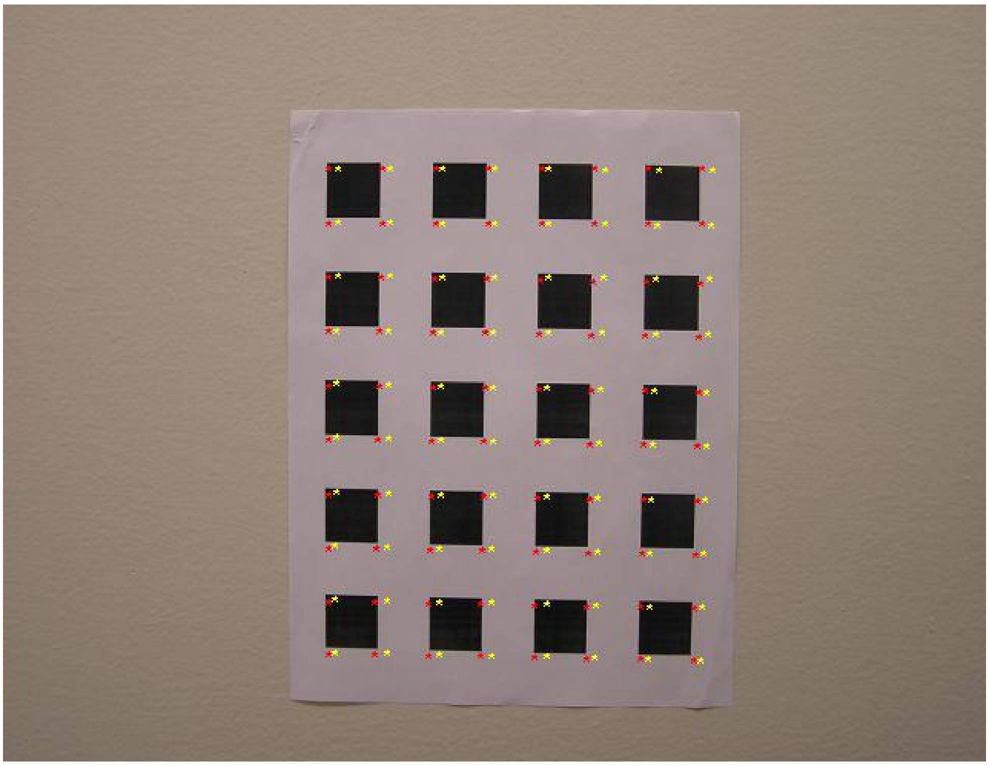


Figure 12: **Reprojection Before LM refine - Image 13 onto Reference image 11**

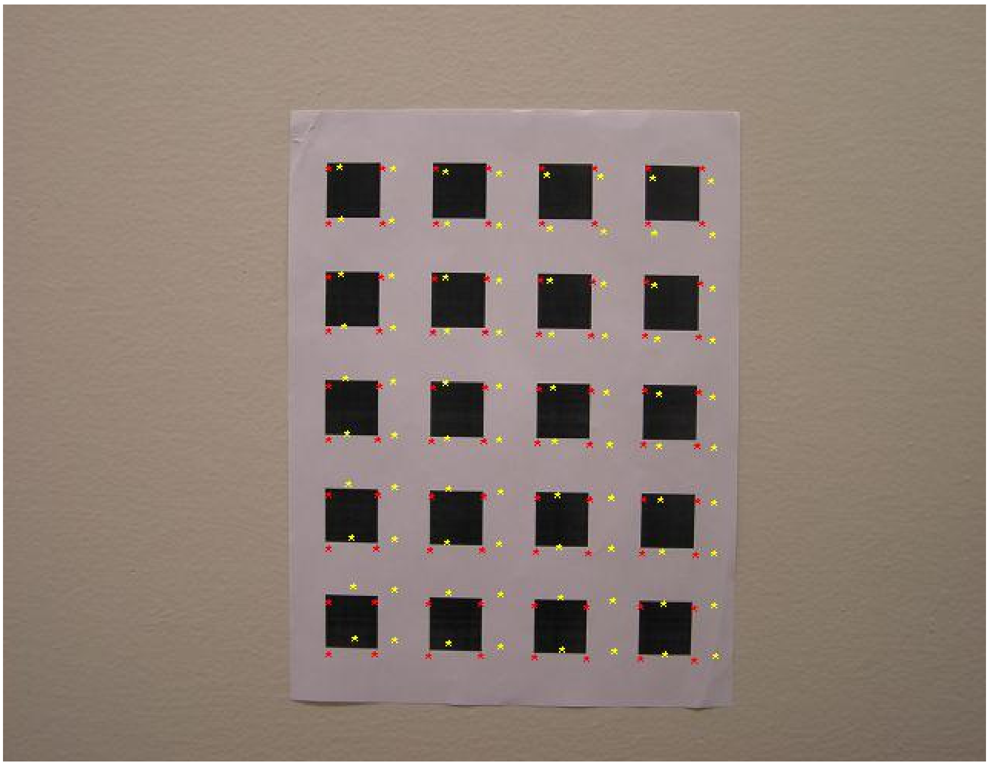


Figure 13: **Reprojection Before LM refine - Image 21 onto Reference image 11**

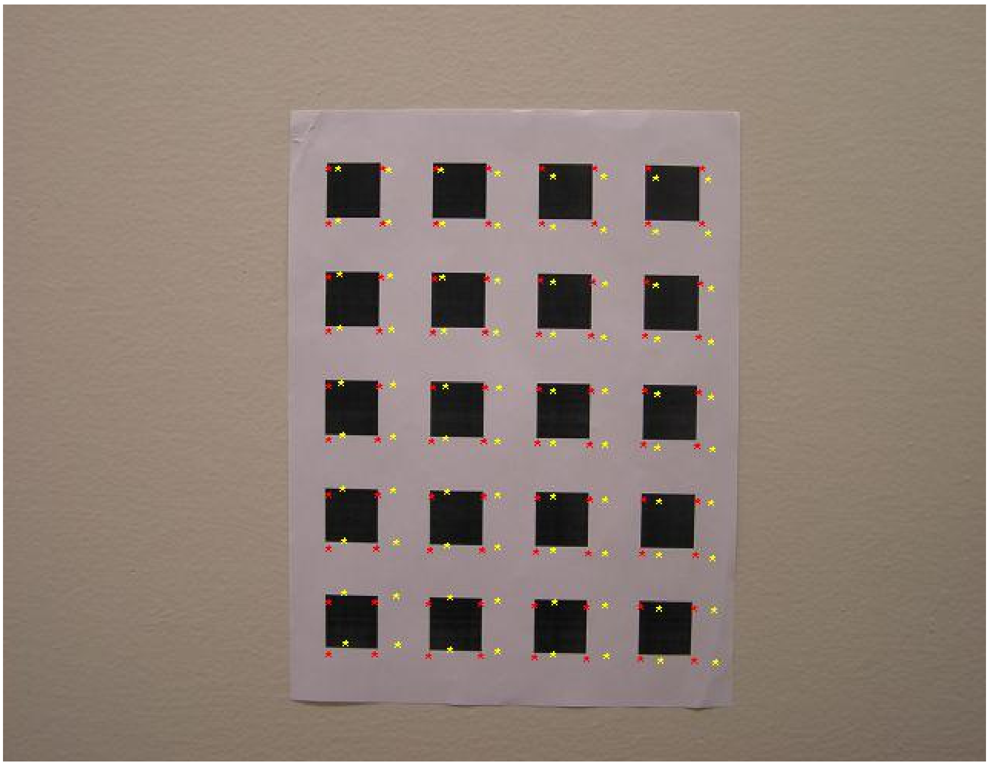


Figure 14: **Reprojection After LM refine - Image 1 onto Reference image 11**

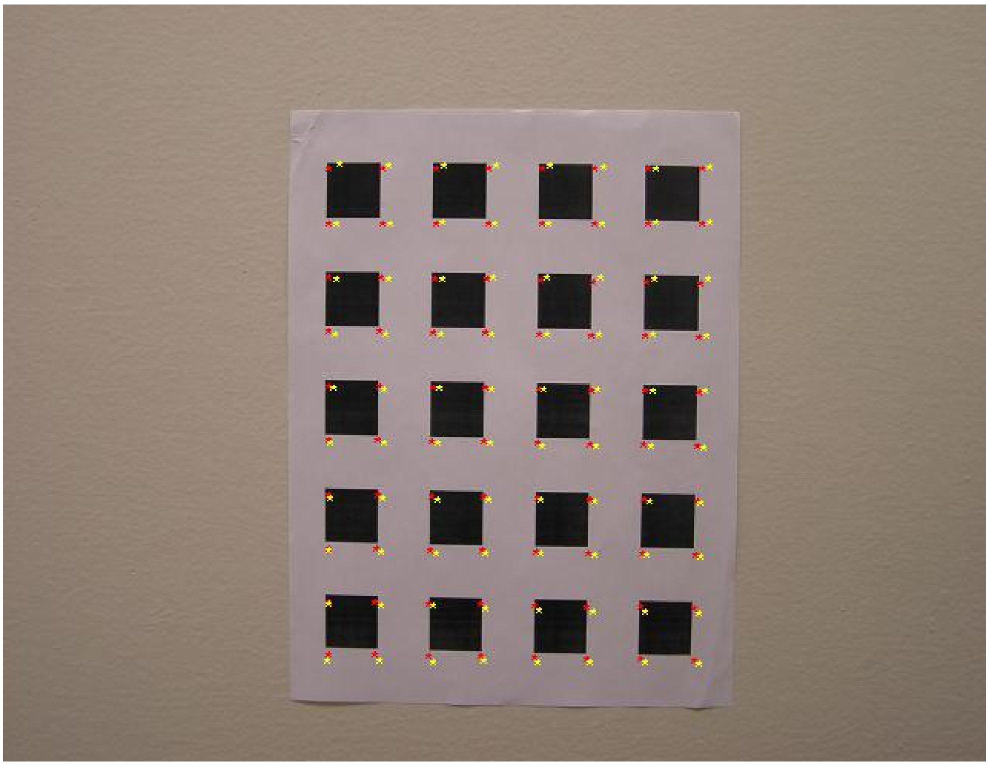


Figure 15: **Reprojection After LM refine - Image 6 onto Reference image 11**

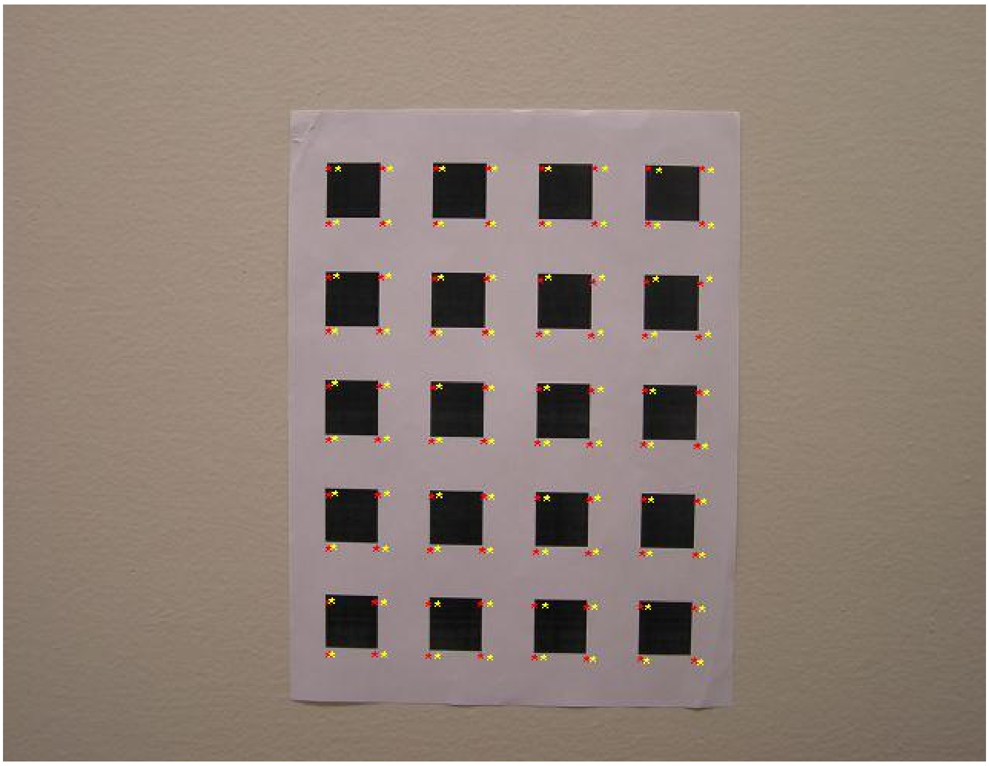


Figure 16: **Reprojection After LM refine - Image 13 onto Reference image 11**

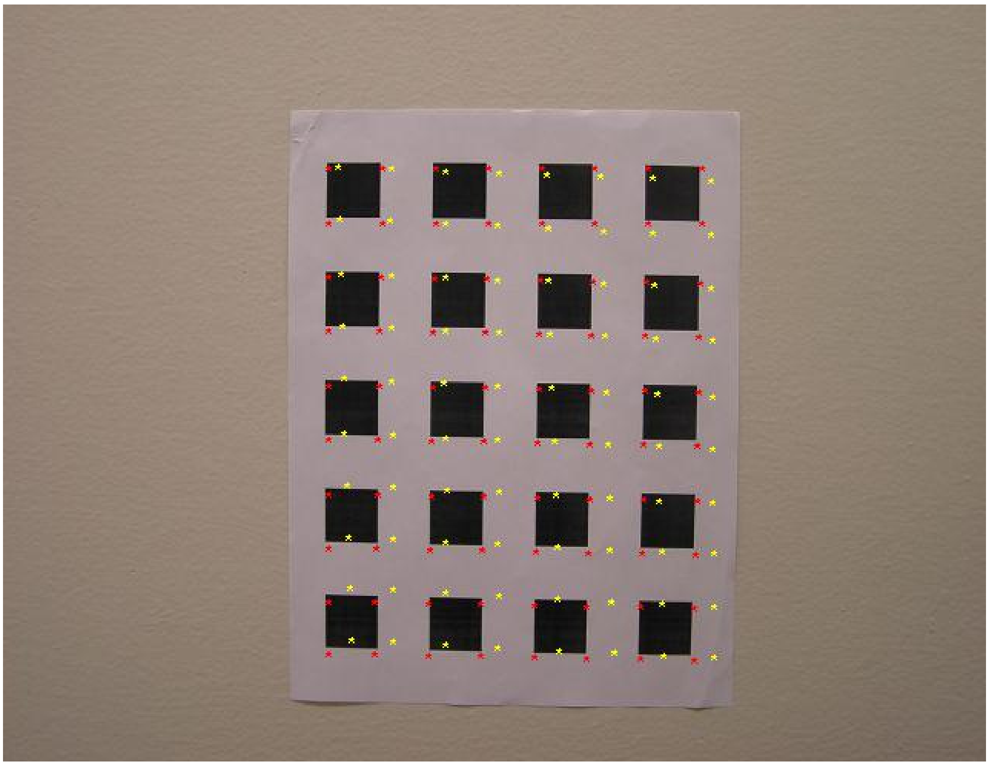
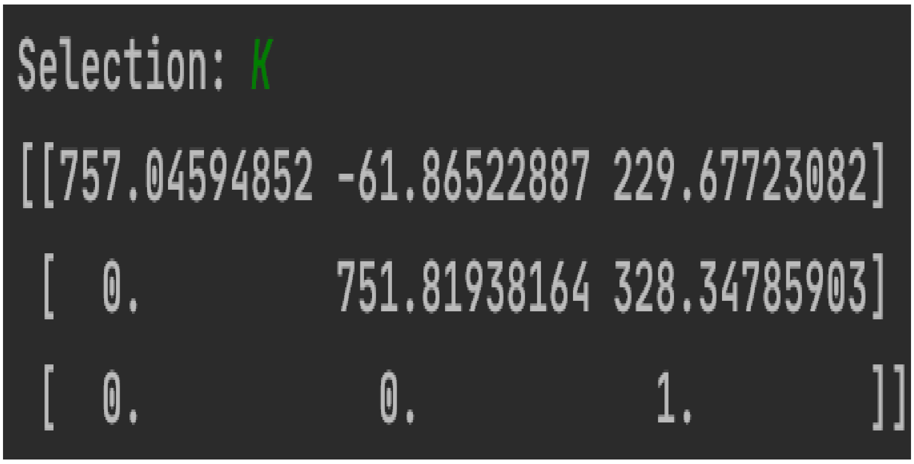


Figure 17: **Reprojection After LM refine - Image 21 onto Reference image 11**

K before LM Refine



```
Selection: K
[[757.04594852 -61.86522887 229.67723082]
 [  0.         751.81938164 328.34785903]
 [  0.           0.           1.        ]]
```

Figure 18: **K before LM**

K after LM Refine

```
Selection: refinedK
[[746.44152204 -72.75482957 339.1665864 ]
 [  0.         740.27786322 232.76358873]
 [  0.           0.           1.        ]]
```

Figure 19: **K after LM**

Extrinsic Camera Parameters Before LM

```
Selection: R
Need index:
3
[[ 8.18538734e-01  4.41275008e-01 -3.67791664e-01 -2.33684622e+00]
 [-4.16655866e-01  8.96818285e-01  1.48710634e-01 -1.18010442e+01]
 [ 3.95464575e-01  3.15171400e-02  9.17940325e-01  5.51569374e+01]]
```

Figure 20: **Extrinsic parameters for Image 4**

```
Selection: R
Need index:
6
[[ 9.97742662e-01 -3.97237075e-02  5.41443121e-02 -2.98330539e+00]
 [ 6.29546604e-03  8.58057103e-01  5.13515700e-01 -1.92435572e+01]
 [-6.68576591e-02 -5.12015658e-01  8.56370258e-01  6.03344557e+01]]
```

Figure 21: **Extrinsic parameters for Image 7**

```
Selection: R
Need index:
15
[[ 8.30616431e-01 -1.90653916e-02  5.56518513e-01 -5.25286655e+00]
 [ 7.72586667e-02  9.93693416e-01 -8.12680311e-02 -1.78520084e+01]
 [-5.51459376e-01  1.10498440e-01  8.26851046e-01  5.03537482e+01]]
```

Figure 22: **Extrinsic parameters for Image 16**

```
Selection: R
Need index:
29
[[  0.99163833   0.10419401   0.07613829  -5.33510962]
 [ -0.09710766   0.99105888  -0.09150083 -15.79206603]
 [ -0.08499136   0.08334212   0.99289     47.10876487]]
```

Figure 23: **Extrinsic parameters for Image 30**

Extrinsic Camera Parameters After LM

```
Selection: refinedRT
Need index:
3
[[ 0.75191661  0.42409525 -9.79422208]
 [-0.38795356  0.90366331 -4.78871794]
 [ 0.53302293  0.059463   52.35355412]]
```

Figure 24: **Extrinsic parameters for Image 4**

```
Selection: refinedRT
Need index:
6
[[ 9.99342330e-01  1.20632257e-02 -1.11648875e+01]
 [-2.79584311e-02  8.56868366e-01 -1.14576658e+01]
 [-2.30918496e-02 -5.15394103e-01  5.84760276e+01]]
```

Figure 25: **Extrinsic parameters for Image 7**

```
Selection: refinedRT
Need index:
15
[[ 8.70235651e-01  2.20749510e-03 -1.17699136e+01]
 [ 2.80365677e-02  9.98147285e-01 -1.11480156e+01]
 [-4.91837231e-01  6.08039874e-02  4.80133538e+01]]
```

Figure 26: **Extrinsic parameters for Image 16**

```
Selection: refinedRT
Need index:
29
[[  0.98792245   0.10876309 -11.64752064]
 [ -0.1006978    0.99199394  -9.50803167]
 [ -0.11776754   0.06417637  45.75933895]]
```

Figure 27: **Extrinsic parameters for Image 30**

Measured ground truth with respect to the reference image

```
Selection: R
Need index:
10
[[ 9.98566502e-01 -2.13310179e-02  4.90910342e-02 -2.75995388e+00]
 [ 2.38151329e-02  9.98435699e-01 -5.05865114e-02 -1.67370943e+01]
 [-4.79351793e-02  5.16831052e-02  9.97512444e-01  5.50487711e+01]]
```

Figure 28: **Ground truth**

Regarding the custom data set

I created the data set as per the instructions given in the handout. But for some reason, the edge detection and the line detections are not coming properly. So, I could not proceed to the next steps. I am attaching the results of the edge and line steps. Unfortunately that was as far as I could go. I am sure if I had more time, I could have created the data set properly and used my same code to run the calibration.

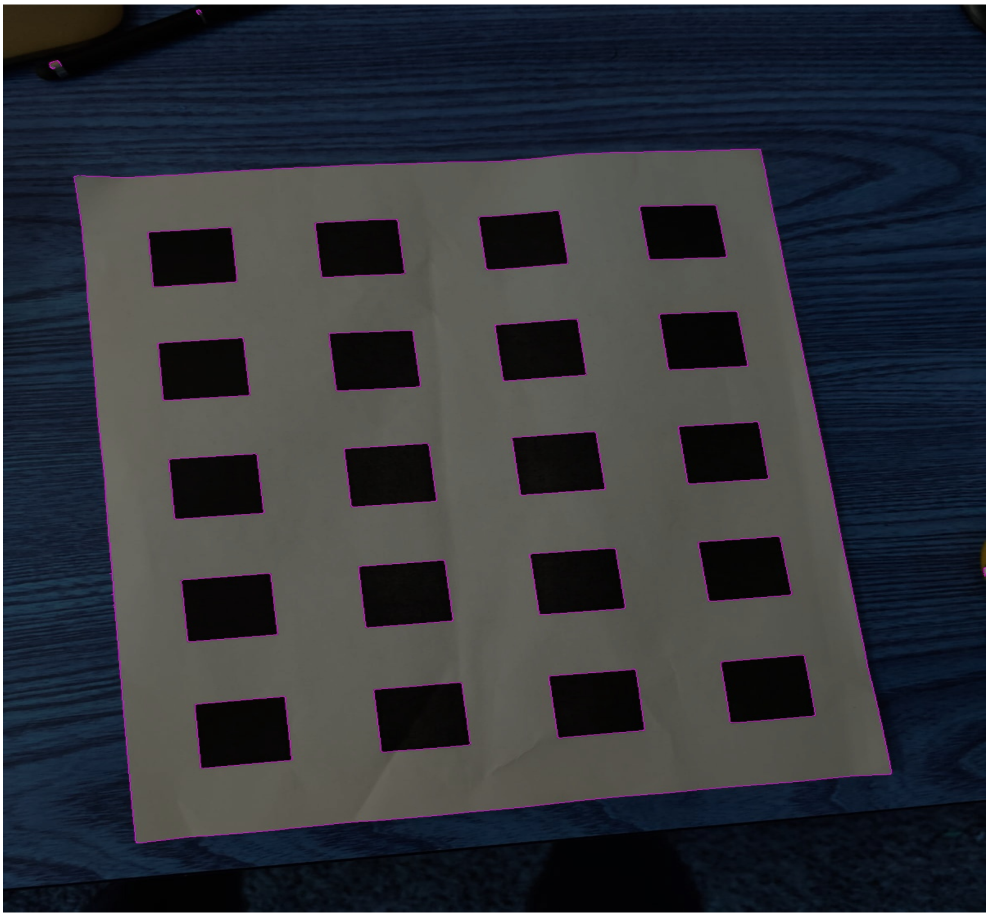


Figure 29: **Edge detection for custom data set**

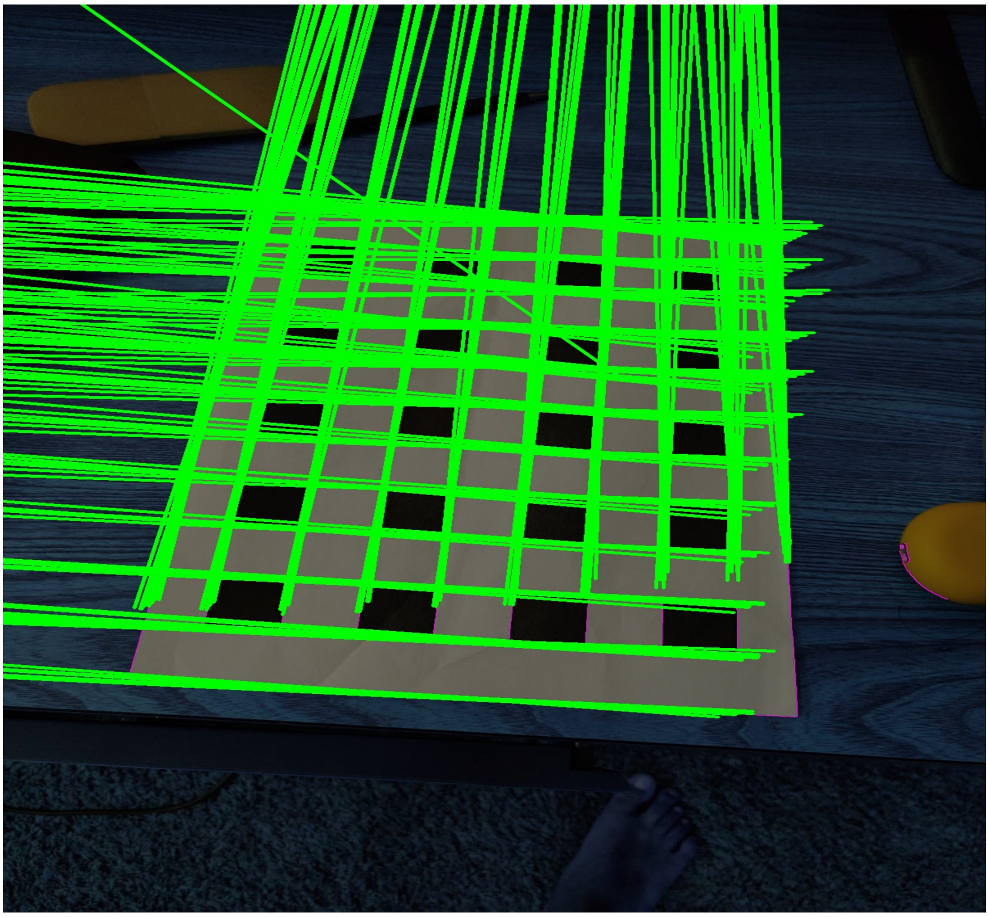


Figure 30: **Line detection for custom data set**

8. Source code

The code was written using the previous year solution as the reference. Reference: Link

```python
"""
Computer Vision - Purdue University - Homework 9

Author : Arjun Kramadhati Gopi, MS-Computer & Information
    Technology, Purdue University.
Date: Nov 2, 2020
-----------------------------------

Reference : (Ref: https://engineering.purdue.edu/RVL/ECE661_2018/
    Homeworks/HW8/2BestSolutions/1.pdf)

-----------------------------------
[TO RUN CODE]: python3 camera_calibration.py
"""
import re
import glob
import os
import pickle
from tqdm import tqdm
import cv2 as cv
```

```python
from PIL import Image,ImageFont,ImageDraw
from scipy.optimize import least_squares
import numpy as np
from pylab import *
import copy


class Calibrate:
    def __init__(self, image_path):
        """
        Initialization code
        :param image_path: Path to the images
        """
        print("Initializing Calibration process...")
        self.image_path = glob.glob(image_path)
        print("Loading image from path " + image_path)
        self.color_images_dict = dict()
        self.gray_images_dict = dict()
        self.lines_dict = dict()
        self.corner_size = (8,10)
        self.corner_list = []
        self.corner_list_filtered = []
        self.homographies = []
        self.cost_variable = []
        self.calibration_performance_raw = dict()
        self.calibration_performance_refined = dict()
        self.parameter_dict = dict()
        self.reference_image = Image.open('Files/Dataset1/Pic_11.
           jpg')
        self.draw = ImageDraw.Draw(self.reference_image)
        self.image_list_g = []
        self.image_list_c = []
        for image_index in range(len(os.listdir('Files/Dataset1
           /'))):
            imagepath = 'Files/Dataset1/Pic_'+str(image_index+1)
               +'.jpg'
            image = np.asarray(Image.open(imagepath))
            self.image_list_c.append(image)
            self.image_list_g.append(cv.cvtColor(image,cv.
               COLOR_BGR2GRAY))
        print(len(self.image_list_g))
        for index, element in enumerate(tqdm(self.image_path,
           ascii=True, desc='Image loading')):
            image = cv.imread(element)
            self.color_images_dict[index] = image
            self.gray_images_dict[index] = cv.cvtColor(image, cv.
               COLOR_BGR2GRAY)
        print("Initialization complete")
        print("-------------------------------------")
        print("-------------------------------------")

    def calibrate_camera(self, run_config = 'Run'):
        """
```

```
        This function sequences all the required functions needed
            to calibrate the camera
        :return:
        """
        if run_config == 'Run':
            #Extract lines
            self.extract_lines()
            #Extract corners from lines
            self.extract_corners()
            #
            self.estimate_corner_homography()
            self.compute_parameter_w()
            self.estimate_extrinsic()
            self.estimate_raw_H()
            self.reproject_and_save()
            self.refine_calibration()
            self.reproject_and_save(Htype='Refined')
            self.save_results()
        elif run_config == 'Analyse':
            self.analyse_results()

    def save_results(self):
        print("Saving Results...")
        pickle.dump(self.parameter_dict,open("
           calibration_parameters.p","wb"))
        pickle.dump(self.calibration_performance_raw, open("
           performance_raw.p","wb"))
        pickle.dump(self.calibration_performance_refined,open("
           performance_refined.p","wb"))
        print("Results saved")

    def analyse_results(self):
        """
        Analyse the results of the calibration
        :return:
        """
        parameters = pickle.load(open("calibration_parameters.p
           ","rb"))
        raw_performance = pickle.load(open("performance_raw.p","
           rb"))
        refined_performance = pickle.load(open("
           performance_refined.p","rb"))
        while True:
            print(parameters.keys())
            print("Enter Selection")
            selection = input('Selection: ')
            if selection == '0':
                break
            elif selection == '9':
                pass
            else:
                if selection =='R' or 'refinedRT':
                    print('Need index: ')
```

```
                    index = input()
                    print(parameters[selection][int(index)])
                else:
                    print(parameters[selection])
            print('Continue?')
            exit = input('0 for quit, 1 for continue')
            if exit == 0:
                break
            elif exit == 1:
                continue

        while True:
            print('Enter comparison image number')
            i = input()
            mean_raw = raw_performance[int(i)][0]
            var_raw =raw_performance[int(i)][1]
            mean_refined = refined_performance[int(i)][0]
            var_refined = refined_performance[int(i)][1]
            print("Image " + str(int(i)+1))
            print("Mean before LM : "+ str(mean_raw) + "||||| Mean after LM : " + str(mean_refined))
            print("Var before LM : " + str(var_raw) + "||||| Variance after LM : " + str(var_refined))
            print("Continue?")
            exit = input('0 for quit, 1 for continue')
            if exit == 0:
                quit()
            elif exit == 1:
                continue

    def get_line(self, rho, theta):
        """
        Get the coordinates of the end points of the lines
        :param rho: rho value
        :param theta: Thetha value
        :return: coordinates
        """
        proportion_one = np.cos(theta)
        proportion_two = np.sin(theta)
        centerX = rho*proportion_one
        centerY = rho*proportion_two
        p1 = int(centerX + 1000*(-proportion_two))
        p2 = int(centerY + 1000*(proportion_one))
        p3 = int(centerX - 1000*(-proportion_two))
        p4 = int(centerY - 1000*(proportion_one))
        return (p1,p2),(p3,p4)

    def extract_lines(self, cutoff = 50, output_path='Files/calibration_output/edges_lines/'):
        """
        Extract the Hough lines
        :param cutoff: Threshold
        :param output_path: Saving the output
```

```
        :return:
        """
        for key in tqdm(range(len(self.image_list_g)), ascii=True
            , desc='Edge & Line extraction'):
            color = (self.image_list_c[key].copy())/2
            edges = cv.Canny(cv.GaussianBlur(self.image_list_g[
                key],(5,5),0),2500, 4000, apertureSize=5)
            color[edges!=0] = (255,0,255)
            cv.imwrite('Files/calibration_output/edges/'+str(key)
                +'.jpg', color)
            hline = cv.HoughLines(edges,1, np.pi/180, cutoff)
            for line in hline:
                for rho, theta in line:
                    point_one, point_two = self.get_line(rho,
                        theta)
                    cv.line(color, point_one, point_two, (0, 255,
                        0), 3)
            self.lines_dict[key] = hline
            cv.imwrite(output_path+str(key)+'.jpg', color)
        print("Line extraction complete...")
        print("-------------------------------------")

    def filter_lines(self, houghlines):
        final_hough_list = np.asarray(houghlines).copy()
        final_hlist_hesse = []
        final_vlist_hesse = []
        for index in range(len(final_hough_list)):
            individual_final_list = np.array(final_hough_list[
                index]).tolist()
            individual_final_list_h = individual_final_list[0:10]
            individual_final_list_h.sort(key=lambda item:item
                [0][0])
            final_hlist_hesse.append(individual_final_list_h)
            individual_final_list_v = individual_final_list[10:]
            individual_final_list_v.sort(key=lambda item:abs(item
                [0][0]))
            final_vlist_hesse.append(individual_final_list_v)
        return final_hlist_hesse, final_vlist_hesse

    def filter_list(self, hlist, vlist, distance_cutoffH = 100,
        distance_cutoffV = 100):
        """
        Filter the list of the hough line selections.
        :param hlist: Horizontal hough lines
        :param vlist: Vertical hough lines
        :param distance_cutoffH: Cutoff criteria
        :param distance_cutoffV: Cutoff criteria
        :return: Filtered list of horizontal and vertical lines
        """
        filtered_hlist = []
        filtered_vlist = []
        while(len(filtered_hlist)<10):
            distance_cutoffH -=0.05
```

```
                filtered_hlist = []
                for index in range(len(hlist)):
                    selectedline = hlist[index][0][0]
                    reject = 0
                    for line in filtered_hlist:
                        if abs(abs(line[0][0])- abs(selectedline))<
                           distance_cutoffH:
                            reject = 1
                    if reject ==0:
                        filtered_hlist.append(hlist[index])

            while(len(filtered_vlist)<8):
                distance_cutoffV -=0.05
                filtered_vlist=[]
                for index in range(len(vlist)):
                    selectedline = vlist[index][0][0]
                    reject = 0
                    for line in filtered_vlist:
                        if abs(abs(line[0][0]) -abs(selectedline)) <
                           distance_cutoffV:
                            reject = 1
                    if reject == 0:
                        filtered_vlist.append(vlist[index])

            return filtered_hlist, filtered_vlist

        def draw_filtered_lines(self, linelist, path='Files/
           calibration_output/final_lines/'):
            """
            Draw the final selected Hough Lines.
            :param linelist: List of all detected hough lines
            :param path: Path to save
            :return:
            """
            for key in tqdm(range(len(self.image_list_g)), ascii=True
               , desc='Drawing filtered lines and saving'):
                lines=linelist[key]
                image = copy.deepcopy(self.image_list_c[key])
                for line in lines:
                    rho = line[0][0]
                    theta = line[0][1]
                    point_one, point_two = self.get_line(rho, theta)
                    cv.line(image, point_one, point_two, (0, 255, 0),
                        3)
                cv.imwrite(path+str(key)+'.jpg', image)

        def extract_corners(self):
            """
            Corner extraction algorithm. I referred the
               implementation from the link provided below.
            https://stackoverflow.com/a/383527/5087436
            :return:
            """
```

```
        linelist = []
        for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Line filtering'):
            horizontal_line_list = []
            vertical_line_list = []
            color = self.image_list_c[key].copy()
            lines = self.lines_dict[key]
            for line in lines:
                theta = line[0][1]
                if np.pi/4<theta<(np.pi*3)/4:
                    horizontal_line_list.append(line)
                else:
                    vertical_line_list.append(line)
            assert(len(horizontal_line_list)+len(vertical_line_list) == len(lines))
            horizontal_line_list, vertical_line_list = self.filter_list(horizontal_line_list, vertical_line_list)
            assert(len(horizontal_line_list) == 10)
            assert(len(vertical_line_list) == 8)
            linelist.append(horizontal_line_list+vertical_line_list)
        self.draw_filtered_lines(linelist)
        final_horizontal_lines, final_vertical_lines = self.filter_lines(linelist)
        corners = []
        for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Corner extraction'):
            individual_corners = []
            for index_vertical in range(len(final_vertical_lines[key])):
                for index_horizontal in range(len(final_horizontal_lines[key])):
                    rho_horizontal, theta_horizontal = final_horizontal_lines[key][index_horizontal][0]
                    rho_vertical, theta_vertical = final_vertical_lines[key][index_vertical][0]
                    A = np.array([
                        [np.cos(theta_vertical), np.sin(theta_vertical)],
                        [np.cos(theta_horizontal), np.sin(theta_horizontal)]
                    ])
                    B = np.array([[rho_vertical],[rho_horizontal]])
                    cornerX, cornerY = np.linalg.solve(A,B)
                    cornerX, cornerY = int(np.round(cornerX)), int(np.round(cornerY))
                    individual_corners.append([[cornerX,cornerY]])
            corners.append(individual_corners)
```

```
        corners_filtered = np.array(np.asarray(corners).copy()).
            tolist()
        self.enumerate_draw_corners(corners_filtered)
        self.corner_list = corners
        self.corner_list_filtered = corners_filtered

    def enumerate_draw_corners(self, corners, path ='Files/
        calibration_output/enumerated_corners/'):
        """
        Draw the numbers for each corner using the same numbering
            pattern.
        :param corners: List of corners
        :param path: Path to save the images.
        :return:
        """
        corners =np.array(np.asarray(corners).copy()).tolist()
        for key in tqdm(range(len(self.image_list_g)), ascii=True
            , desc='Enumerate coners'):
            image_path='Files/Dataset1/Pic_'+str(key+1)+'.jpg'
            image =Image.open(image_path)
            recreate_img = ImageDraw.Draw(image)
            for corner_index in range(len(corners[key])):
                recreate_img.text((corners[key][corner_index
                    ][0][0],corners[key][corner_index][0][1]),str(
                    corner_index),(255,0,0))
            image.save(path+str(key)+'.jpg')

    def estimate_extrinsic(self):
        """
        Estimates the extrinsic parameters
        :return: Stores in the dictionary
        """
        omega = self.parameter_dict['omega']
        centerX = ((omega[0][1]*omega[0][2])-(omega[0][0]*omega
            [1][2]))/((omega[0][0]*omega[1][1])-(omega[0][1]*omega
            [0][1]))
        lambdavalue = omega[2][2] - (((omega[0][2]*omega[0][2])+
            centerX*((omega[0][1]*omega[0][2])-(omega[0][0]*omega
            [1][2])))/omega[0][0])
        a_x,a_y = abs(np.sqrt(lambdavalue/omega[0][0])),abs(np.
            sqrt((lambdavalue*omega[0][0])/abs((omega[0][0]*omega
            [1][1])-(omega[0][1]*omega[0][1]))))
        svalue = -1*((omega[0][1]*a_x*a_x*a_y)/(lambdavalue))
        centerY = ((svalue*centerX)/a_y)-((omega[0][2]*a_x*a_x)/
            lambdavalue)
        K = np.zeros((3,3))
        K[0][0] = a_x
        K[0][1] = svalue
        K[0][2] = centerX
        K[1][0] = 0.0
        K[1][1] = a_y
        K[1][2] = centerY
        K[2][0] = 0.0
```

```
        K[2][1] = 0.0
        K[2][2] = 1.0
        self.parameter_dict['K'] = K
        self.parameter_dict['a_x'] = a_x
        self.parameter_dict['a_y'] = a_y
        self.parameter_dict['svalue'] = svalue
        self.parameter_dict['centerX'] = centerX
        self.parameter_dict['centerY'] = centerY

        matrixR =[]
        for key in tqdm(range(len(self.homographies)), ascii=True, desc='Extrinsic estimation'):
            evalue = 1/np.linalg.norm(np.matmul(np.linalg.pinv(K), self.homographies[key][: , 0]))
            firstR = evalue*np.matmul(np.linalg.pinv(K), self.homographies[key][: , 0])
            secondR = evalue*np.matmul(np.linalg.pinv(K) ,self.homographies[key][: ,1])
            thirdR = np.cross(firstR, secondR)
            matrixZ = self.condition_rotation_matrix([firstR, secondR,thirdR])
            firstR, secondR, thirdR = matrixZ[:,0],matrixZ[:,1], matrixZ[:,2]
            tvalue = evalue*np.matmul(np.linalg.pinv(K) ,self.homographies[key][:,2])
            rotationmatrix = np.zeros((3,4))
            rotationmatrix[:,0] = firstR
            rotationmatrix[:,1] = secondR
            rotationmatrix[:,2] = thirdR
            rotationmatrix[:,3] = tvalue
            matrixR.append(rotationmatrix)
        self.parameter_dict['R'] = matrixR

    def condition_rotation_matrix(self, rvalues):
        """
        Condition the rotation matrix
        :param rvalues: R matrix required to condition
        :return: Conditioned matrix
        """
        matrixQ = np.zeros((3,3))
        matrixQ[:,0] = rvalues[0]
        matrixQ[:,1] = rvalues[1]
        matrixQ[:,2] = rvalues[2]
        uvalue, dvalue, vvalueT = np.linalg.svd(matrixQ)
        matrixZ = np.matmul(uvalue,vvalueT)
        return matrixZ

    def get_omega_matrix(self, matrixb):
        matrix_omega = np.zeros((3, 3))
        matrix_omega[0][0] = matrixb[0]
        matrix_omega[0][1] = matrixb[1]
        matrix_omega[0][2] = matrixb[3]
        matrix_omega[1][0] = matrixb[1]
```

```
        matrix_omega[1][1] = matrixb[2]
        matrix_omega[1][2] = matrixb[4]
        matrix_omega[2][0] = matrixb[3]
        matrix_omega[2][1] = matrixb[4]
        matrix_omega[2][2] = matrixb[5]
        return matrix_omega

    def compute_parameter_w(self):
        """
        Computes the omega parameter of the camera.
        :return: Stores the omega value in the dictionary
        """
        matrixV = np.zeros((2*len(self.homographies),6))
        for key in tqdm(range(len(self.homographies)), ascii=True, desc='Omega estimation'):
            homography = np.transpose(self.homographies[key])
            templist = []
            for item in [(0,1),(0,0),(1,1)]:
                vmatrix = np.zeros((1, 6))
                vmatrix[0][0] = homography[item[0]][0] * homography[item[1]][0]
                vmatrix[0][1] = (homography[item[0]][0] * homography[item[1]][1])+(homography[item[0]][1] * homography[item[1]][0])
                vmatrix[0][2] = homography[item[0]][1] * homography[item[1]][1]
                vmatrix[0][3] = (homography[item[0]][2] * homography[item[1]][0])+(homography[item[0]][0] * homography[item[1]][2])
                vmatrix[0][4] = (homography[item[0]][2] * homography[item[1]][1]) + (
                            homography[item[0]][1] * homography[item[1]][2])
                vmatrix[0][5] = homography[item[0]][2] * homography[item[1]][2]
                templist.append(vmatrix)
            first_vmatrix = templist[0][0]
            second_vmatrix = (templist[1] - templist[2])[0]
            matrixV[2*key] = first_vmatrix
            matrixV[2*key+1] = second_vmatrix
        umatrix, dmatrix, vmatrixT = np.linalg.svd(matrixV)
        matrixB = np.transpose(vmatrixT)[:,-1]
        omega = self.get_omega_matrix(matrixB)
        self.parameter_dict['omega'] = omega
        self.parameter_dict['matrixV'] = matrixV

    def get_refined_omega(self,x,y,z):
        omegamatrix_x = np.zeros((3, 3))
        omegamatrix_x[0][0] = 0.0
        omegamatrix_x[1][1] = 0.0
        omegamatrix_x[2][2] = 0.0
        omegamatrix_x[0][1] = -1*z
        omegamatrix_x[0][2] = y
```

```
        omegamatrix_x[1][0] = z
        omegamatrix_x[1][2] = -1*x
        omegamatrix_x[2][0] = -1*y
        omegamatrix_x[2][1] = x
        return omegamatrix_x

    def set_temp_matrix(self, parameter, point):
        temp_estimate = np.matmul(parameter, np.asarray(point))
        return temp_estimate/temp_estimate[2]

    def set_gamma(self, temp_estimate, center):
        return np.sqrt(np.square(temp_estimate[0]-center[0])+np.square(temp_estimate[1]-center[1]))

    def calibration_cost(self, point):
        """
        Cost function for LM refinement
        :return: Cost vector
        """
        resid = []
        for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='LM Refine'):
            omega_x = point[6*key+5]
            omega_y = point[6*key+1+5]
            omega_z = point[6*key+2+5]
            first_t = point[6*key+3+5]
            second_t = point[6*key+4+5]
            third_t = point[6*key+5+5]
            a_x = point[0]
            svalue = point[1]
            centerX = point[2]
            a_y = point[3]
            centerY = point[4]
            omegamatrix = np.zeros((3,1))
            omegamatrix[0] = omega_x
            omegamatrix[1] = omega_y
            omegamatrix[2] = omega_z
            phivalue = np.linalg.norm(omegamatrix)
            omegamatrix_x = self.get_refined_omega(omega_x, omega_y,omega_z)
            matrixT = np.zeros((3))
            matrixT[0]=first_t
            matrixT[1]=second_t
            matrixT[2]=third_t
            matrix_first_R = np.zeros((3,3))
            second_R = (np.sin(phivalue) / phivalue) * omegamatrix_x
            third_R = ((1-np.cos(phivalue))/(phivalue*phivalue))*np.matmul(omegamatrix_x,omegamatrix_x)
            matrix_first_R[0][0] = 1.0
            matrix_first_R[1][1] = 1.0
            matrix_first_R[2][2] = 1.0
            final_R = matrix_first_R+second_R+third_R
```

```
            matrixK = np.zeros((3,3))
            matrixK[0][0] = a_x
            matrixK[2][2] = 1.0
            matrixK[0][1] = svalue
            matrixK[0][2] = centerX
            matrixK[1][1] = a_y
            matrixK[1][2] = centerY
            rotation_matrix = np.zeros((3,3))
            rotation_matrix[:,0]=final_R[:,0]
            rotation_matrix[:,1]=final_R[:,1]
            rotation_matrix[:,2]=matrixT
            for imagecorner_index in range(len(self.corner_list_filtered[key])):
                point_estimate = []
                x_est = (imagecorner_index/10)*2.5
                y_est = (imagecorner_index%10)*2.5
                point_coordinate = np.array(np.asarray(self.corner_list_filtered[key][imagecorner_index][0]).copy()).tolist()
                point_coordinate.append(1.0)
                point_estimate.append(x_est)
                point_estimate.append(y_est)
                point_estimate.append(1.0)
                camera_parameter = np.matmul(matrixK, rotation_matrix)
                temp_estimate = self.set_temp_matrix(camera_parameter, point_estimate)
                gammavalue = self.set_gamma(temp_estimate, (centerX,centerY))
                final_estimate = (np.asarray(point_coordinate)-temp_estimate)
                resid.append(final_estimate[0])
                resid.append(final_estimate[1])
                resid.append(final_estimate[2])
        return resid

    def refine_calibration(self):
        """
        Refine the calibration parameters of the camera.
        :return:
        """
        matrixR = list(np.asarray(self.parameter_dict['R']))
        self.cost_variable.append(self.parameter_dict['a_x'])
        self.cost_variable.append(self.parameter_dict['svalue'])
        self.cost_variable.append(self.parameter_dict['centerX'])
        self.cost_variable.append(self.parameter_dict['a_y'])
        self.cost_variable.append(self.parameter_dict['centerY'])
        for homography_index in range(len(self.homographies)):
            trace_value =(np.trace(matrixR[homography_index][:,0:3])-1)/2
            if trace_value>1.0:
                trace_value=1.0
            phivalue = np.arccos(trace_value)
```

```
            if phivalue==0:
                phivalue=1
            self.cost_variable.append((matrixR[homography_index][2][1]-matrixR[homography_index][1][2])*(phivalue/(2*np.sin(phivalue))))
            self.cost_variable.append((matrixR[homography_index][0][2] - matrixR[homography_index][2][0]) * (
                        phivalue / (2 * np.sin(phivalue))))
            self.cost_variable.append((matrixR[homography_index][1][0] - matrixR[homography_index][0][1]) * (
                        phivalue / (2 * np.sin(phivalue))))
            self.cost_variable.append(matrixR[homography_index][0][3])
            self.cost_variable.append(matrixR[homography_index][1][3])
            self.cost_variable.append(matrixR[homography_index][2][3])
        optimised_R = least_squares(self.calibration_cost, self.cost_variable, method='lm',max_nfev=800)
        self.estimate_refined_H(optimised_R)

    def reproject_and_save(self, Htype = 'Raw'):
        """
        Reproject and save the images.
        :param Htype:Type of homography used. Raw is for the normal homography and refined is for the
        optimised homography value.
        :return:
        """
        if Htype == 'Raw':
            for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Reprojection Raw'):
                if key == 10:
                    pass
                else:
                    homography = np.matmul(self.parameter_dict['rawH'][10], np.linalg.pinv(self.parameter_dict['rawH'][key-1]))
                    projection = []
                    self.reference_image = Image.open('Files/Dataset1/Pic_11.jpg')
                    self.draw = ImageDraw.Draw(self.reference_image)
                    for index in range(len(self.corner_list[10])):
                        coordinates = list(np.asarray(self.corner_list[key-1][index][0]).copy())
                        coordinates.append(1.0)
                        projectedpoint = np.matmul(homography, np.asarray(coordinates))
                        projectedpoint = projectedpoint/projectedpoint[2]
                        projection.append(projectedpoint[:-1])
```

```
                    distance = []
                    for corner_index in range(len(self.corner_list[10])):
                        self.draw.text(( self.corner_list[10][corner_index][0][0] , self.corner_list[10][corner_index][0][1]) ,"*",(255,0,0))
                        self.draw.text(( list(projection[corner_index]) [ 0 ] , list(projection[corner_index])[ 1 ] ) , "*" , ( 255 , 255 , 0 ))
                        distance.append(np.linalg.norm(np.asarray(self.corner_list[10][corner_index][0])-projection[corner_index]))
                    self.reference_image.save('Files/calibration_output/reprojection_raw/'+str(key)+'.jpg')
                    self.calibration_performance_raw[key] = (np.mean(distance),np.var(distance))
        elif Htype == 'Refined':
            for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Reprojection Refined'):
                if key == 10:
                    pass
                else:
                    homography = np.matmul(self.parameter_dict['refined_homography'][10], np.linalg.pinv(self.parameter_dict['rawH'][key-1]))
                    projection = []
                    self.reference_image = Image.open('Files/Dataset1/Pic_11.jpg')
                    self.draw = ImageDraw.Draw(self.reference_image)
                    for index in range(len(self.corner_list[10])):
                        coordinates = list(np.asarray(self.corner_list[key-1][index][0]).copy())
                        coordinates.append(1.0)
                        projectedpoint = np.matmul(homography, np.asarray(coordinates))
                        projectedpoint = projectedpoint/projectedpoint[2]
                        projection.append(projectedpoint[:-1])
                    distance = []
                    for corner_index in range(len(self.corner_list[10])):
                        self.draw.text(( self.corner_list[10][corner_index][0][0] , self.corner_list[10][corner_index][0][1]) ,"*",(255,0,0))
                        self.draw.text(( list(projection[corner_index])[0] , list( projection[corner_index])[ 1 ] ) , " *" , ( 255 ,
```

```
                            255 , 0 ))
                        distance.append(np.linalg.norm(np.asarray(self.corner_list[10][corner_index][0])-projection[corner_index]))
                    self.reference_image.save('Files/calibration_output/reprojection_refined/'+str(key)+'.jpg')
                    self.calibration_performance_refined[key] = (np.mean(distance),np.var(distance))

    def estimate_refined_H(self, refined_R):
        """
        Estimate the refined homography needed to reproject the points
        :param refined_R: Refined R matrix we got by running the LM least squares algorithm
        :return:
        """
        homography = []
        templist = []
        for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Refined Homography'):
            matrixK = np.zeros((3,3))
            matrixK[0][0] = refined_R.x[0]
            matrixK[0][1] = refined_R.x[1]
            matrixK[0][2] = refined_R.x[2]
            matrixK[1][1] = refined_R.x[3]
            matrixK[1][2] = refined_R.x[4]
            matrixK[2][2] = 1.0
            matrixW = np.zeros((3,1))
            matrixW_x = np.zeros((3,3))
            matrixR = np.zeros((3,3))
            matrixT = np.zeros((3))
            firstr = np.zeros((3,3))
            rotationmatrix = np.zeros((3,3))
            omega_x,omega_y,omega_z =refined_R.x[5+6*key], refined_R.x[5+1+6*key], refined_R.x[5+2+6*key]
            matrixW[0] = omega_x
            matrixW[1] = omega_y
            matrixW[2] = omega_z
            phivalue = np.linalg.norm(matrixW)
            matrixW_x[0][1] = -1*omega_z
            matrixW_x[0][2] = omega_y
            matrixW_x[1][0] = omega_z
            matrixW_x[1][2] = -1*omega_x
            matrixW_x[2][0] = -1*omega_y
            matrixW_x[2][1] = omega_x
            firstr[0][0] = 1.0
            firstr[1][1] =1.0
            firstr[2][2] = 1.0
            secondr = (np.sin(phivalue)/phivalue)*matrixW_x
            thirdr = ((1-np.cos(phivalue))/(phivalue*phivalue))*np.matmul(matrixW_x,matrixW_x)
```

```
            matrixR = firstr+secondr+thirdr
            matrixT[0] = refined_R.x[5+3+6*key]
            matrixT[1] = refined_R.x[5+4+6*key]
            matrixT[2] = refined_R.x[5+5+6*key]
            rotationmatrix[:,0]=matrixR[:,0]
            rotationmatrix[:,1] = matrixR[:,1]
            rotationmatrix[:,2] = matrixT
            homography.append(np.matmul(matrixK,rotationmatrix))
            templist.append(rotationmatrix)
            self.parameter_dict['refinedK'] = matrixK
        self.parameter_dict['refinedRT'] = templist
        self.parameter_dict['refined_homography'] = homography

    def estimate_raw_H(self):
        """
        Estimate the homography needed for the reprojection of the points.
        :return:
        """
        matrixR = np.asarray(self.parameter_dict['R'])
        raw_homographies = []
        K = self.parameter_dict['K']
        for key in tqdm(range(len(self.image_list_g)), ascii=True, desc='Raw matrix estimation'):
            raw_homographies.append(np.matmul(K,matrixR[key][:,[0,1,3]]))
        self.parameter_dict['rawH'] = raw_homographies

    def estimate_corner_homography(self):
        """
        Estimate the homography which maps the corners and their world coordinates
        :return:
        """
        H = []
        for key in tqdm(range(len(self.corner_list)), ascii=True, desc='Homography estimation'):
            matrixA = np.zeros((2*len(self.corner_list[key]), 9))
            for corner_index in range(len(self.corner_list[key])):
                matrixA[2 * corner_index + 0][0] = (corner_index/10)*2.5
                matrixA[2 * corner_index + 0][1] = (corner_index%10)*2.5
                matrixA[2 * corner_index + 0][2] = 1.0
                matrixA[2 * corner_index + 0][3] = 0.0
                matrixA[2 * corner_index + 0][4] = 0.0
                matrixA[2 * corner_index + 0][5] = 0.0
                matrixA[2 * corner_index + 0][6] = -1*((corner_index/10)*2.5)*self.corner_list[key][corner_index][0][0]
                matrixA[2 * corner_index + 0][7] = -1*((corner_index%10)*2.5)*self.corner_list[key][
```

```
corner_index][0][0]
                matrixA[2 * corner_index + 0][8] = -1*self.corner_list[key][corner_index][0][0]
                matrixA[2 * corner_index + 1][0] = 0.0
                matrixA[2 * corner_index + 1][1] = 0.0
                matrixA[2 * corner_index + 1][2] = 0.0
                matrixA[2 * corner_index + 1][3] = (corner_index/10)*2.5
                matrixA[2 * corner_index + 1][4] = (corner_index%10)*2.5
                matrixA[2 * corner_index + 1][5] = 1.0
                matrixA[2 * corner_index + 1][6] = -1*((corner_index/10)*2.5)*self.corner_list[key][corner_index][0][1]
                matrixA[2 * corner_index + 1][7] = -1 * ((corner_index % 10) * 2.5) * self.corner_list[key][corner_index][0][1]
                matrixA[2 * corner_index + 1][8] = -1*self.corner_list[key][corner_index][0][1]
            homography = np.zeros((3,3))
            umatrix, dmatrix, vmatrixT = np.linalg.svd(matrixA)
            H_matrix = np.transpose(vmatrixT)[:,-1]
            if H_matrix[8] == 0 or H_matrix[8] ==NaN:
                print("True divide conflict. Ignoring value...")
            else:
                H_matrix = H_matrix/H_matrix[8]
            homography[0][0] = H_matrix[0]
            homography[0][1] = H_matrix[1]
            homography[0][2] = H_matrix[2]
            homography[1][0] = H_matrix[3]
            homography[1][1] = H_matrix[4]
            homography[1][2] = H_matrix[5]
            homography[2][0] = H_matrix[6]
            homography[2][1] = H_matrix[7]
            homography[2][2] = 1.0
            H.append(homography)
        self.homographies = H


if __name__ == "__main__":
    """
    Program starts here.
    """
    tester = Calibrate('./Files/Dataset1/*')
    tester.calibrate_camera()
```