ECE661 Fall 2024: Homework 8

By – Aayush Bhadani

abhadani@purdue.edu

The goal of this assignment is to implement Zhang's algorithm for camera calibration.

1. Theory Question

1. The property that the plane containing calibration pattern samples the absolute conic at exactly two points which are circular is fundamental to Zhang's algorithm for camera calibration as it leads to estimating the intrinsic parameters (3x3 K matrix) of the camera. Estimating this intrinsic parameters matrix K is an important part of the overall camera calibration task.

Using the homography from the plane of calibration pattern to camera image plane on these two circular points we get two points on the image conic ω which helps us in setting two equations. With at least 3 such pairs obtained from different poses we can find unknowns of ω and thus estimate the intrinsic parameters K (as $\omega = K^{-T}K^{-1}$).

2. Absolute conic Ω_∞ lies on π_∞ and is defined as: $(x_1 x_2 x_3)\, \mathrm{I}_{3\mathrm{x}3} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 0$, where $\Omega_\infty \equiv I_{3x3}$

Camera image of a point $\vec{X} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix}$ on π_∞ is given by:

$$\vec{x} = P\vec{X} = KR\left[I_{3x3}\middle|-\tilde{\vec{C}}\right]\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix} = KR\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = H\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Using homography $H = KR$ we can obtain image of the absolute conic as:

$$\begin{aligned} \omega = H^{-T}I_{3x3}H^{-1} = (KR)^{-T}I_{3x3}(KR)^{-1} &= (KR)^{-T}(KR)^{-1} \\ &= ((KR)^T)^{-1}(KR)^{-1} \\ &= (R^TK^T)^{-1}(KR)^{-1} \\ &= K^{-T}R^{-T}R^{-1}K^{-1} \\ &= K^{-T}(RR^T)^{-1}K^{-1} \quad ; \ (R^T = R^{-1}) \\ &= K^{-T}K^{-1} \end{aligned}$$

Actual pixels on the image conic ω would be $\vec{x}^T\omega\vec{x} = 0$. However, all these pixels are imaginary as $K^{-T}K^{-1}$ is positive definite meaning $\vec{x}^T\omega\vec{x} > 0$.

3. Task – Camera Calibration using Zhang's algorithm

3.1 Corner Detection:

3.3.1 Edge detection for each image: using Canny() from OpenCV. Threshold parameters thres_l and thres_h have to selected accordingly in order to obtain edges corresponding to the grid pattern only. These values were set to 250 & 500.

3.3.2 Extract Hough Lines: using HoughLines() from OpenCV performed HoughLine Transform to detect straight lines. The threshold for this which was set to 50 in sample dataset and 100 in other dataset controls the number of lines we obtain through this. It will still give us more lines then needed. We have to process them to obtain the desired number of lines (8 vertical and 10 horizontal required) for our task.

3.3.3 Divide lines into Horizontal & Vertical sets: Using θ values from Hough Lines we can segregate the lines into two groups.

3.3.4 Within each (horizontal/vertical) set, group the lines: Divide the lines of a set into desired number of groups using K_means clustering based on coordinate of intersection of the lines with a test line. So for the vertical set of lines we find their intersection with the test line ($y = k$) and then use x-coordinate of intersection to clusters these lines into 8 groups. Similarly for horizontal set of lines, we use a test line ($x = k$) and cluster into 10 groups based on y-coordinate of intersection. For these computations HC representation is useful. Using a k value such that test line falls in a less crowded region can be helpful.

3.3.5 For each cluster estimate a single line: For vertical lines, find intersection of the lines in the cluster with $y = k_1 \; and \; y = k_2$ lines and then average x-coordinates. This way we get two points for creating an average line in the cluster. Similarly for horizontal lines in the cluster use $x = k_1 \; and \; x = k_2$ lines and average y-coordinates.

3.3.6 Sort the lines: Following the above steps we end up with the desired number of lines (8 vertical and 10 horizontal required) for our task. We sort line sets based on their intercepts in increasing order to obtain the filtered lines. This would give us order- top to bottom and left to right

3.3.7 Find intersection of sorted filtered lines: Finally, using the sorted filtered sets of horizontal and vertical lines we find their intersection points which corresponds to the corners. Thus we obtain 80 corner points starting from 1st at top left corner and 80th at bottom right corner. Note: Make sure the consistent labelling of corners across all images, as it is required by Zhang's algorithm.

3.2 Camera Calibration:

Here we estimate the camera parameters: intrinsic and extrinsic.

We start by finding the homographies between the image poses and the world coordinate. For establishing correspondences, we use extracted corner points from image poses and generated world coordinates (origin at top left corner and step as per block_step size).

Using these homographies we construct the V matrix $\begin{bmatrix} V_1 \\ V_2 \\ . \\ . \\ . \\ V_N \end{bmatrix}$, where $V_{i\,(2X6)} = \begin{bmatrix} \vec{V}_{12}^{T} \\ (\vec{V}_{11} - \vec{V}_{22})^{T} \end{bmatrix}$

and elements are $\vec{V}_{ij} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} for\ i, j \in \{1,2\}.\ h_i\ rep.\ i^{th} column.$

Unknown vector $b = \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}$ and we have the relation $\begin{bmatrix} V_1 \\ V_2 \\ . \\ . \\ . \\ V_N \end{bmatrix} b = 0.$

Using linear least squares, we can solve for vector b and thus obtain ω matrix which is:

$$\omega = \begin{bmatrix} \omega_{11} & \omega_{12} & \omega_{13} \\ \omega_{12} & \omega_{22} & \omega_{23} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{bmatrix}$$

We know that $\omega = K^{-T}K^{-1}$, where $K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$,

Based on Zhang's approach we estimate **intrinsic parameters**, K matrix with:

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

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

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

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

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

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

Next, we get the **extrinsic parameters**: $R = [\vec{r}_1\vec{r}_2\vec{r}_3]\ and\ \vec{t}$

based on relationship: $K^{-1}[\vec{h}_1\vec{h}_2\vec{h}_3] = [\vec{r}_1\vec{r}_2\vec{t}]$

Scale factor $\xi = \frac{1}{||K^{-1}\vec{h}_1||}$

$$\vec{r}_1 = \xi K^{-1}\vec{h}_1$$

$$\vec{r}_2 = \xi K^{-1}\vec{h}_2$$

$$\vec{r}_3 = \vec{r}_1 \times \vec{r}_2$$

$$\vec{t} = \xi K^{-1}\vec{h}_3$$

To **condition a rotation matrix**, we find its SVD and set all singular values to 1 i.e set $R = UV^T$

We also need a 3-parameter representation of rotation matrix for optimization as it has 3DoF.

We use **Rodrigues representation** for this where vector $\vec{w} = \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix}$

and its 3x3 representation $[\vec{w}]_X = \begin{bmatrix} 0 & -w_z & w_x \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$

To construct vector $\vec{w}$ for a given R, we get:

$$\varphi = \cos^{-1}\frac{tr(R)-1}{2} \text{ and subsequently } \vec{w} = \frac{\varphi}{2sin\varphi}\begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix};\ r_{ij}\ element\ of\ R$$

And to construct R matrix for a given vector $\vec{w}$, we use:

$$R = e^{[\vec{w}]_X} = I_{3X3} + \frac{sin\varphi}{\varphi}[\vec{w}]_X + \frac{1-cos\varphi}{\varphi^2}[\vec{w}]_X^2 \qquad ; \qquad where\ \varphi = ||\vec{w}||$$

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

3.3 Refining Parameters

The estimated camera calibration parameters are refined using non-linear least squares. We project points from calibration pattern plane into image plane and then use Euclidean distance between projected pixels and where they actually occur in the image as a metric for assessing the quality of calculated calibration parameters.

We have the squared geometric error as: $d_{geom}^2 = ||\vec{X} - \vec{f}(\vec{p})||^2$, $\vec{p}$ is parameter vector.

Levenberg-Marquadt(LM) method is used for performing this non-linear least squares.

3.4 Radial distortion (Extra)

We do the adjustment on the projected pixel position $(\hat{x}, \hat{y})$ based on:

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

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

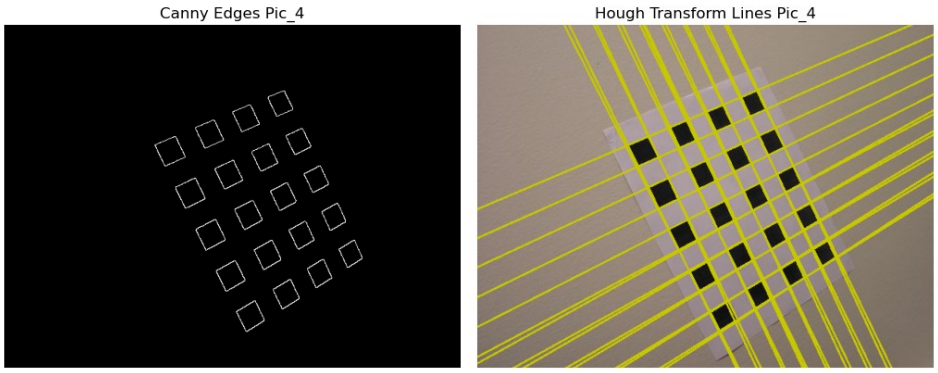
where $r = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$ and (x_0, y_0) is principal point. k_1, k_2 are the parameters which we tune using non-linear least squares along with other camera parameters as part of parameter vector $\vec{p}$ described above. k_1, k_2 are initialized with 0.

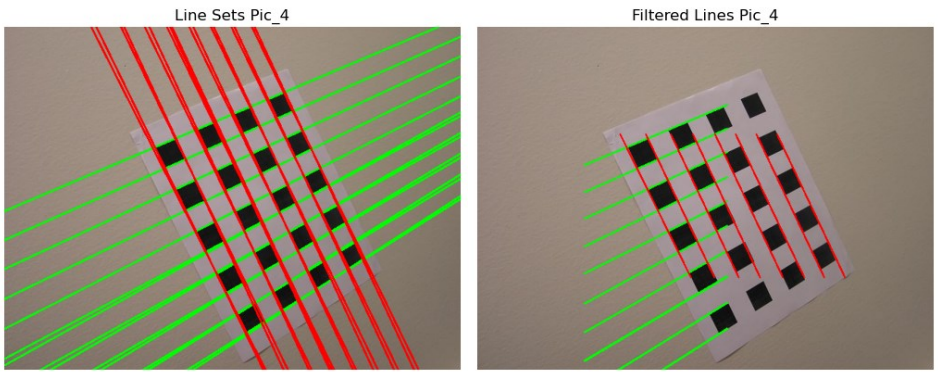
RESULTS:

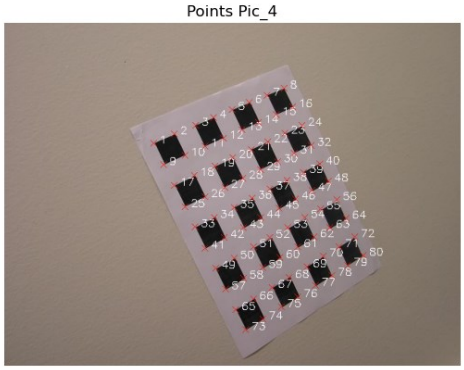
- **Given Dataset**

Corner Detection:

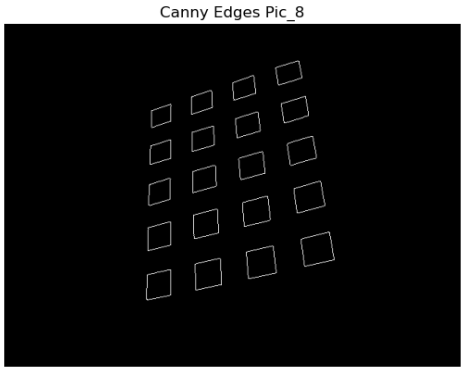
For Pic_4:

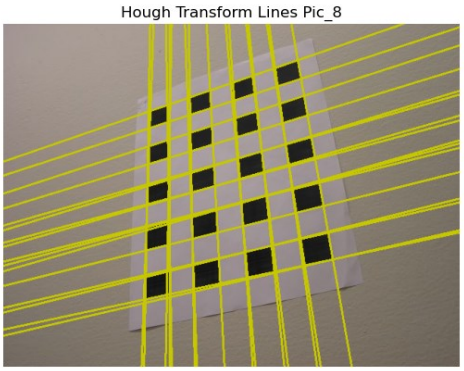


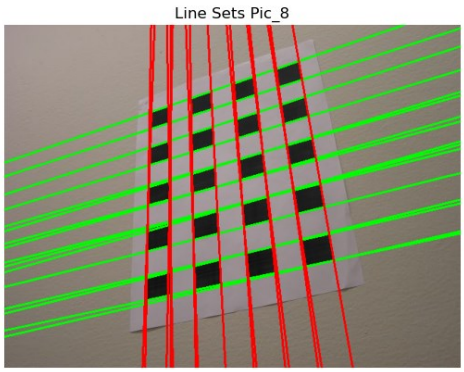


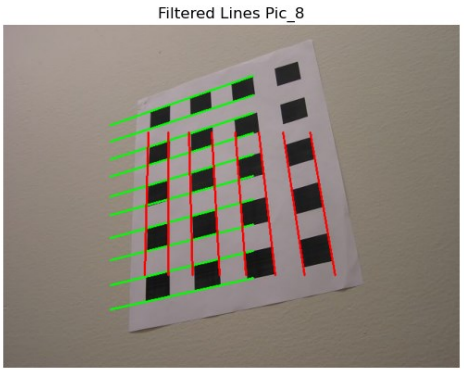


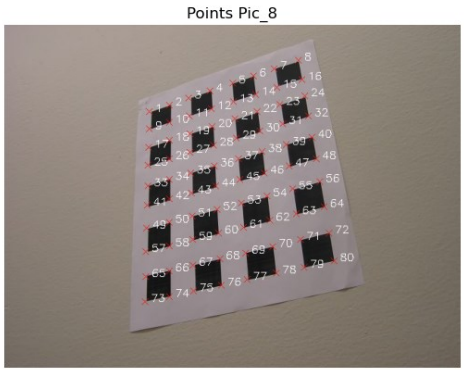
For Pic_8:











Camera Calibration:

Before LM:

Intrinsic Parameters:

K:

```
[[713.4223361    1.41855717 312.47157145]
 [  0.         709.82462158 232.20472434]
 [  0.           0.           1.        ]]
```

For Pic_4:

Extrinsic Parameters:

R :

```
 [[ 0.8265169   0.42879595 -0.36469693]
 [-0.41108208  0.90237646  0.12933775]
 [ 0.38455343  0.04302053  0.92209972]]
```

t :

```
 [-27.48156668 -17.31595275 191.2225407 ]
```

Error mean and variance:

mean: 22.79453445167551 , var: 86.5575439808355

For Pic_8:

Extrinsic Parameters:

R :

```
 [[ 0.92314196  0.06233997  0.37937139]
 [-0.25059488  0.84591273  0.47078005]
 [-0.29156667 -0.52966535  0.79651961]]
```

t :

```
 [-37.83417817 -39.13203493 249.08986909]
```

Error mean and variance:

mean: 14.20762350122664 , var: 50.431164366210986

Average Error mean and variance across the dataset:

mean: 22.06, var: 134.07

After LM:

Intrinsic Parameters:

K :

```
 [[718.03863435   2.60578339 316.06516286]
 [  0.         715.32344504 233.55997517]
 [  0.           0.           1.        ]]
```

For Pic_4:

Extrinsic Parameters:

R :

[[0.8265169 0.42879595 -0.36469693]

[-0.41108208 0.90237646 0.12933775]

[0.38455343 0.04302053 0.92209972]]

t :

[-27.48156668 -17.31595275 191.2225407]

Error mean and variance:

mean: 0.868258103476592 , var: 0.16117441596349963

For Pic_8:

Extrinsic Parameters:

R :

[[0.92314196 0.06233997 0.37937139]

[-0.25059488 0.84591273 0.47078005]

[-0.29156667 -0.52966535 0.79651961]]

t :

[-37.83417817 -39.13203493 249.08986909]

Error mean and variance:

mean: 0.9444420551441738 , var: 0.24323594365807022

Average Error mean and variance across the dataset:

mean: 0.894, var: 0.214

Considering radial distortion:

Radial distortion parameters:

K1,k2:[-2.71659995e-07 1.73250725e-12]

Error mean and variance – Pic_4:

Before LM: mean: 22.79453445167551 , var : 86.5575439808355

After LM: mean: 0.8740573292049344, var : 0.19137595832152643

Error mean and variance – Pic_8:

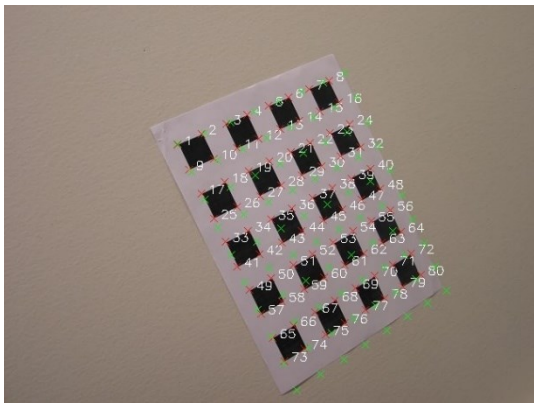
Before LM: mean: 14.20762350122664 , var : 50.431164366210986

After LM: mean: 0.9418112319408447, var : 0.2361416296689678

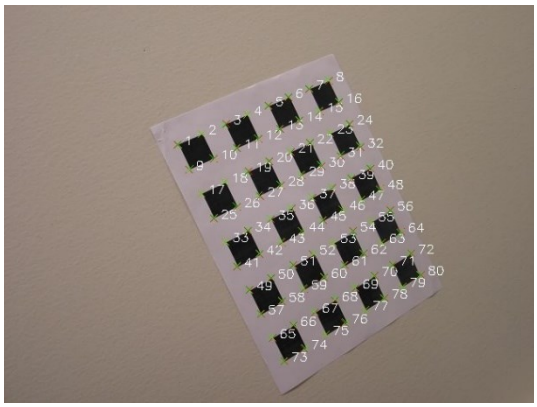
Output Images (green for reprojected, red for original)

Pic_4:

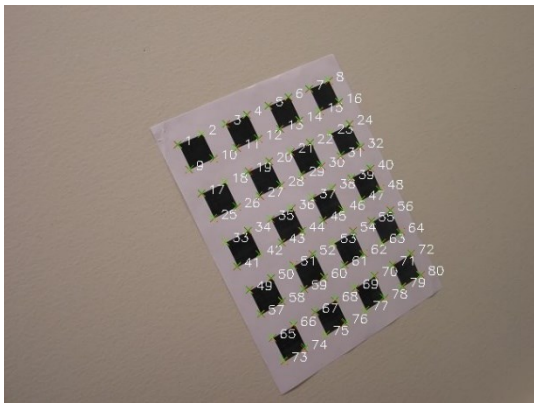
Before LM



After LM

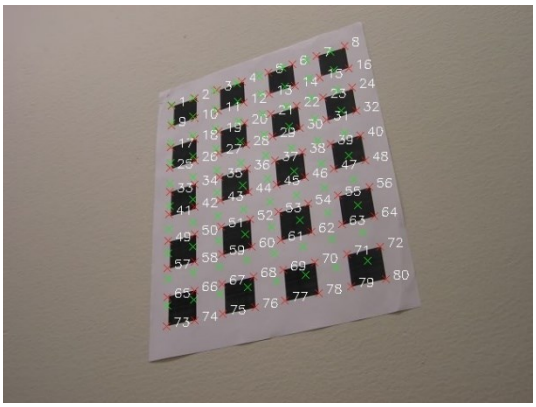


Considering radial distortion

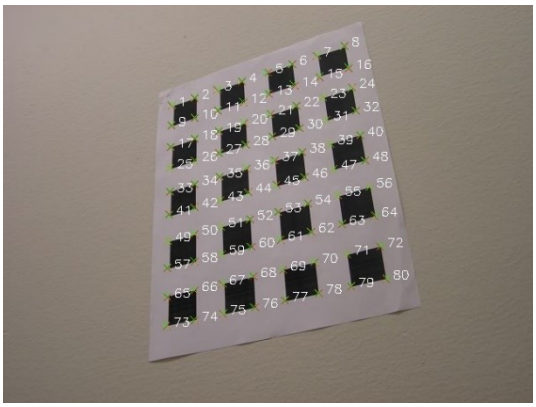


Pic 8:

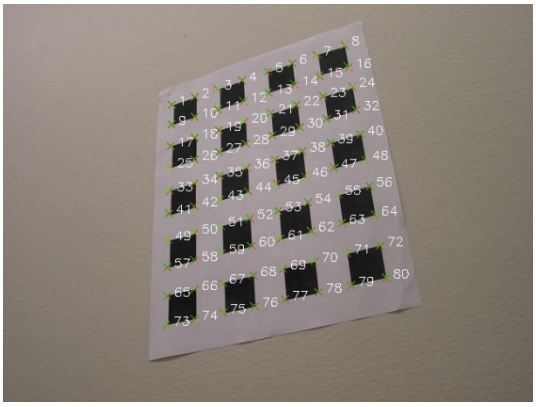
Before LM



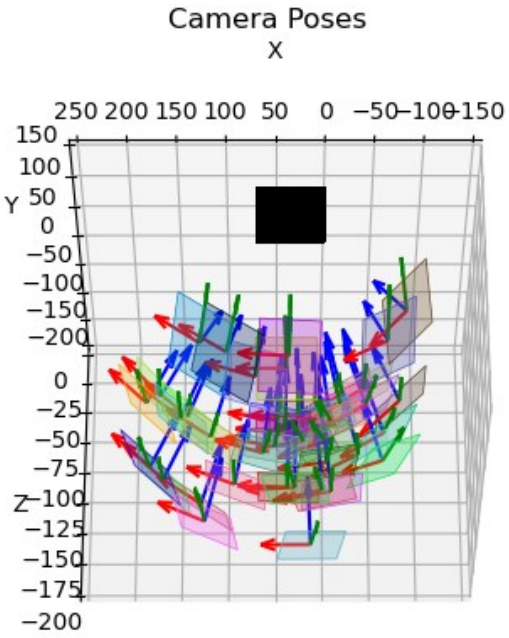
After LM



Considering radial distortion



Camera Poses:

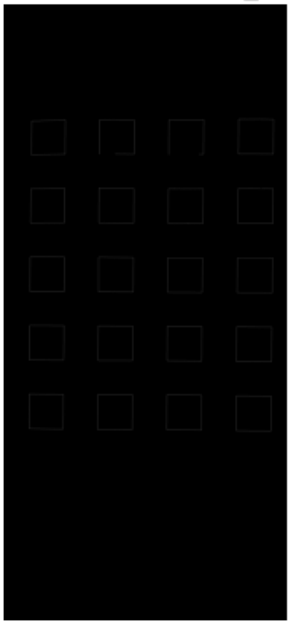


- **Other Dataset** (physical length of block size is 2.4cm)

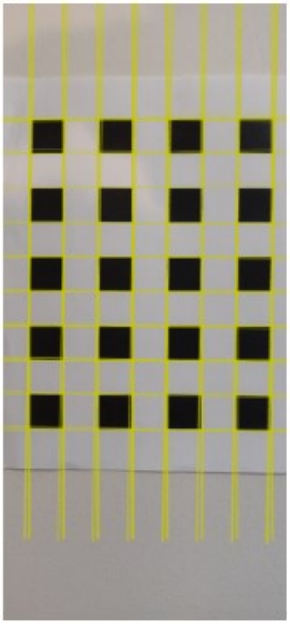
Corner Detection:

For Pic_1:

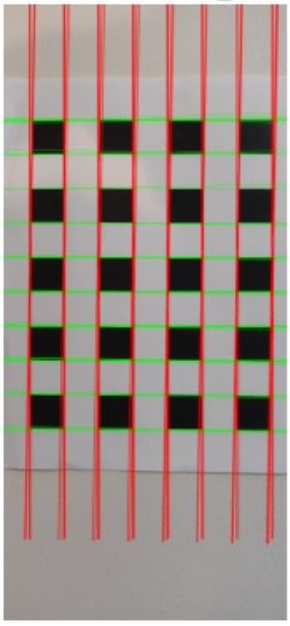
Canny Edges Pic_1



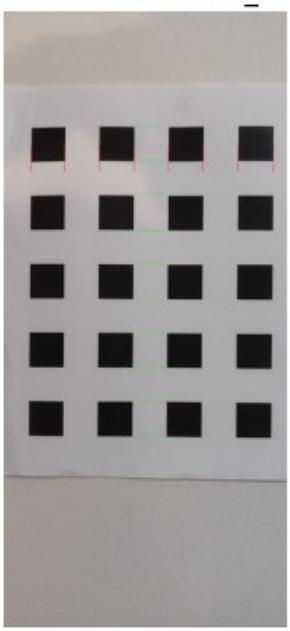
Hough Transform Lines Pic_1



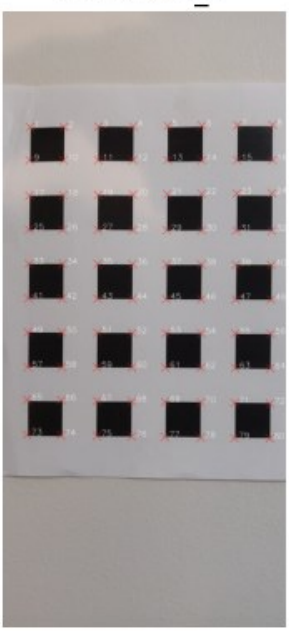
Line Sets Pic_1



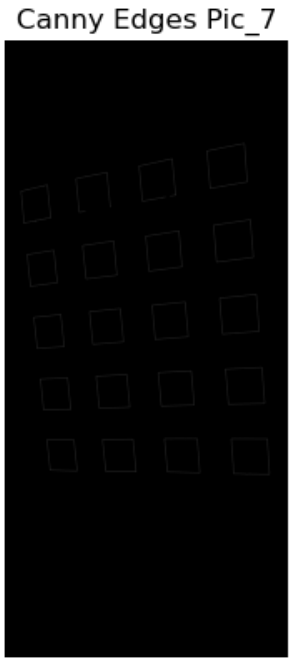
Filtered Lines Pic_1

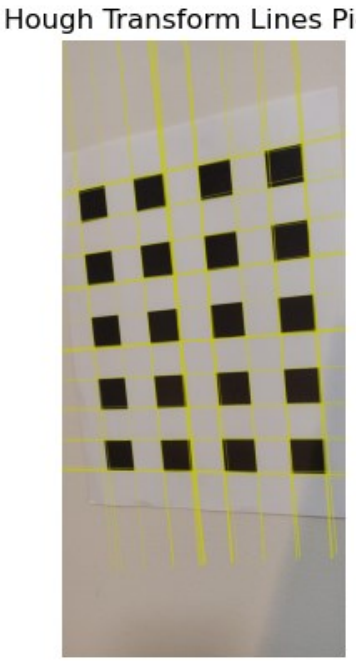


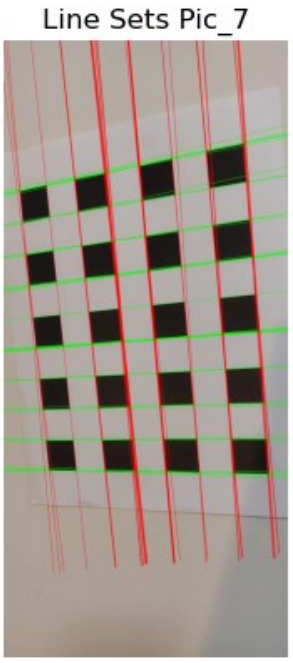
Points Pic_1

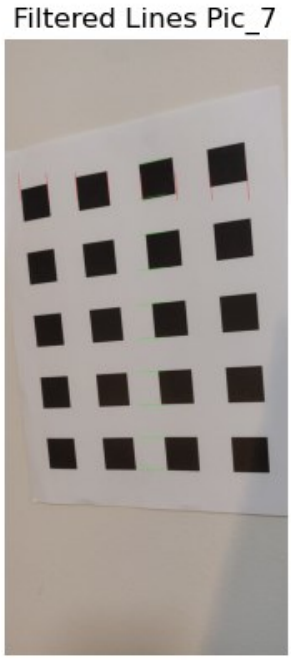


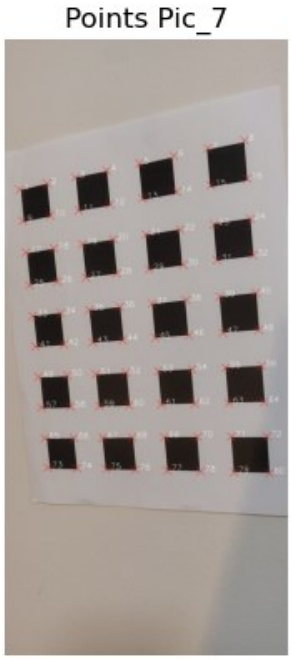
For Pic_7:











Camera Calibration:

Before LM:

Intrinsic Parameters:

K :

[[3.55028827e+03 2.27926773e+00 1.01239625e+03]
[0.00000000e+00 3.55259476e+03 2.31072212e+03]
[0.00000000e+00 0.00000000e+00 1.00000000e+00]]

For Pic_1:

Extrinsic Parameters:

R :

[[0.99956052 -0.00730489 0.02873002]

[0.00739443 0.99996813 -0.00301134]

[-0.0287071   0.00322246 0.99958267]]

t :

[-7.63453484 -13.64957406 33.46406335]

Error mean and variance:

mean: 11.742773414656185 , var : 15.958733389308534

For Pic_7:

Extrinsic Parameters:

R :

[[0.94058961 0.09334541 0.32646258]

[-0.06497568 0.99318308 -0.09677566]

[-0.33327067 0.06981406 0.94024287]]

t :

[-9.3051502 -12.2602291   36.49251055]

Error mean and variance:

mean: 28.182583046852027 , var : 140.07673682041533

Average Error mean and variance across the dataset:

mean: 67.33686866193874, var : 1584.3293417591024

After LM:

Intrinsic Parameters:

K :

[[3.67080898e+03 -5.60797835e+00 1.00700260e+03]

[0.00000000e+00 3.66676384e+03 2.34321779e+03]

[0.00000000e+00 0.00000000e+00 1.00000000e+00]]

For Pic_1:

Extrinsic Parameters:

R :

[[0.99956052 -0.00730489 0.02873002]

[0.00739443 0.99996813 -0.00301134]

[-0.0287071   0.00322246 0.99958267]]

t :

[-7.63453484 -13.64957406 33.46406335]

Error mean and variance:

mean: 4.2538865032707776 , var : 5.075647451126882

For Pic_7:

Extrinsic Parameters:

R :

[[0.94058961 0.09334541 0.32646258]

[-0.06497568 0.99318308 -0.09677566]

[-0.33327067 0.06981406 0.94024287]]

t :

[-9.3051502 -12.2602291   36.49251055]

Error mean and variance:

mean: 5.467656793032804 , var : 10.423543003831247

Average Error mean and variance across the dataset:

mean: 4.842638598750181, var : 8.117549438361635

Considering radial distortion:

Radial distortion parameters:

K1,k2:[1.96405607e-09 -6.11683010e-16

Error mean and variance – Pic_1:

Before LM: mean: 11.742773414656185 , var : 15.958733389308534

After LM: mean: 4.212952630686241 , var : 4.873034638599627

Error mean and variance – Pic_7:

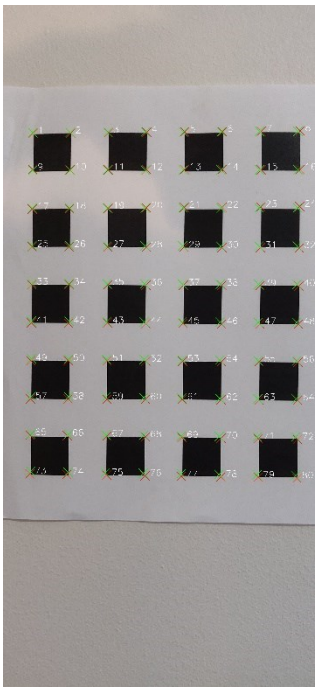
Before LM: mean: 28.182583046852027 , var : 140.07673682041533

After LM: mean: 5.470690247473348 , var : 10.644100621408754

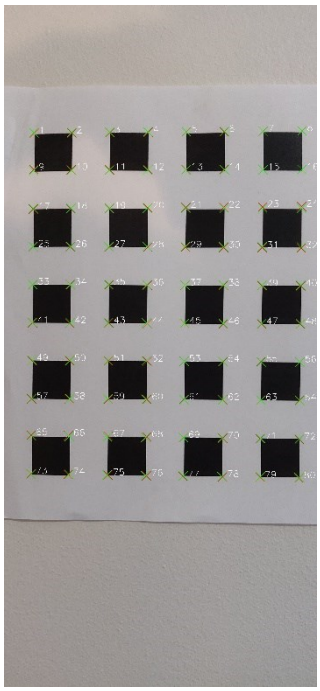
Output Images (green for reprojected, red for original)

Pic_1:

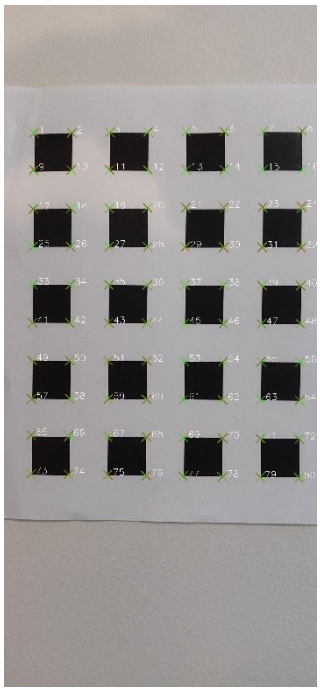
Before LM



After LM

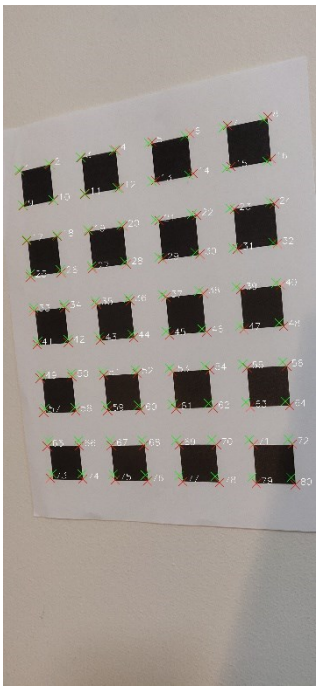


Considering radial dist.

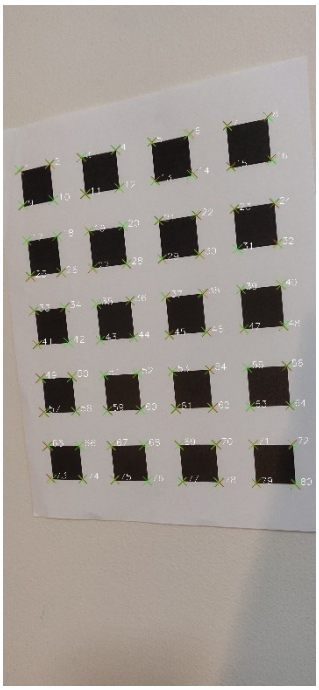


Pic 7:

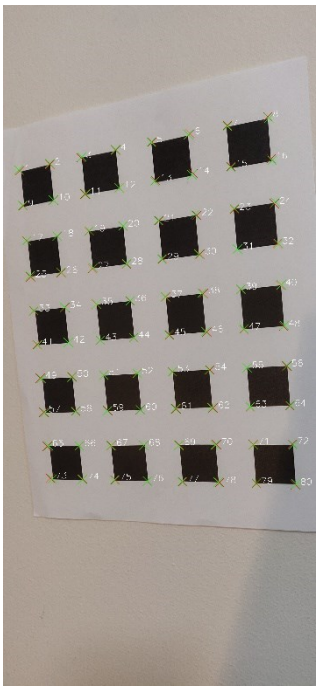
Before LM



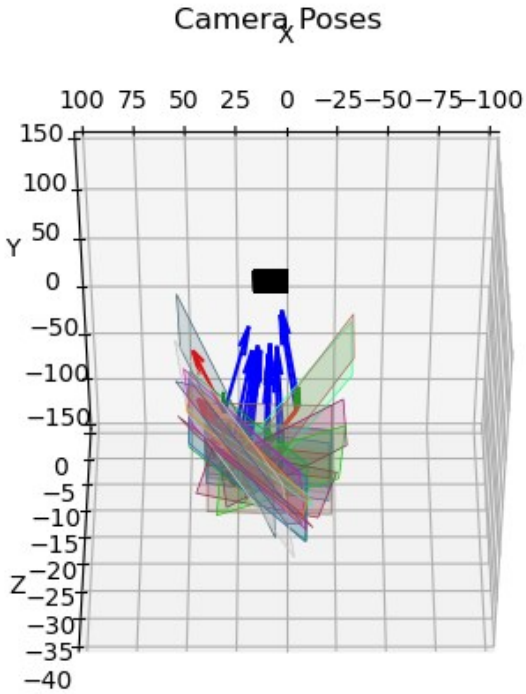
After LM



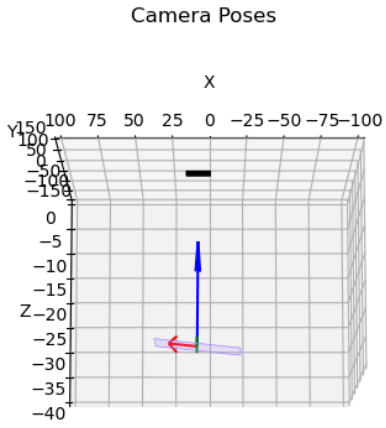
Considering radial dist.



Camera Poses:



Estimated Camera pose for ‘Fixed Image’ (Pic_1):



This is consistent with the measured ground truth distance which was 30cm.

CODE:

```
# %%
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import random
import math
import os
from skimage import io
from sklearn.cluster import KMeans,DBSCAN
import scipy.optimize as optim
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import Rectangle
from mpl_toolkits.mplot3d import art3d

# %%
db = 1 #1 for given DB 2 for self DB
data_path = 'HW8-Files\HW8-Files\Dataset1' if db==1 else 'self_db'
img_list = sorted(os.listdir(data_path),key=lambda x: int((x.split('_')[-1]).split('.')[0]))

# %%
# image_path = os.path.join(data_path,img_list[1])

# %%
def intersect_point(l1_HC,l2_HC):
    eps = 1e-8
    int_pt_HC = np.cross(l1_HC,l2_HC)
    int_pt = int_pt_HC/(int_pt_HC[2] + eps)
    return (int(int_pt[0]),int(int_pt[1]))

def filter_lines(lines_set,setType):
    # get intercepts
    k=50
    testLine = np.array([0,-1,k]) if setType == 'vertical' else np.array([-1,0,k])
    int_pt_list = []
    for l in lines_set:
        l_HC = l[6:]
        int_pt = intersect_point(l_HC,testLine)
        int_pt_list.append(int_pt[0] if setType == 'vertical' else int_pt[1])
    int_pt_list = (np.array(int_pt_list)).reshape(-1,1)
    # intercepts = int_pt_list[:,0] if setType == 'vertical' else int_pt_list[:,1]

    # Clustering
    n_clusters = 8 if setType == 'vertical' else 10 # given pattern-8 vertical & 10 horizontal lines
    kmeans = KMeans(n_clusters, random_state=42)
    kmeans.fit(int_pt_list)
    line_clusters = {i:[] for i in range(n_clusters)}
    for i,label in enumerate(kmeans.labels_):
        line_clusters[label].append(lines_set[i])

    # Est. average line in each cluster grp
    averaged_lines =[]
    k1,k2 = 150,350
    testLine1 = np.array([0,-1,k1]) if setType == 'vertical' else np.array([-1,0,k1])
    testLine2 = np.array([0,-1,k2]) if setType == 'vertical' else np.array([-1,0,k2])
    for cluster_num, lineArr_list in line_clusters.items():
        int_pt1_list, int_pt2_list = [],[]
        for l in lineArr_list:
            l_HC = l[6:]
            int_pt1 = intersect_point(l_HC,testLine1)
            int_pt2 = intersect_point(l_HC,testLine2)
            int_pt1_list.append(int_pt1[0] if setType == 'vertical' else int_pt1[1])
            int_pt2_list.append(int_pt2[0] if setType == 'vertical' else int_pt2[1])
        avg_int_pt1 = int(np.mean(np.array(int_pt1_list)))
        avg_int_pt2 = int(np.mean(np.array(int_pt2_list)))
        new_pt1 = np.array([avg_int_pt1,k1,1]) if setType == 'vertical' else np.array([k1,avg_int_pt1,1])
        new_pt2 = np.array([avg_int_pt2,k2,1]) if setType == 'vertical' else np.array([k2,avg_int_pt2,1])
        avg_line_HC = np.cross(new_pt1,new_pt2)
        avg_line_HC = avg_line_HC/(avg_line_HC[2] + 1e-8)
        avg_line = np.array([new_pt1[0],new_pt1[1],new_pt2[0],new_pt2[1]]+list(avg_line_HC))
        averaged_lines.append(avg_line)

    # sort filtered lines
    filtered_lines = sorted(averaged_lines, key=lambda l: (intersect_point(l[4:],testLine))[0] if setType == 'vertical' else (intersect_point(l[4:],testLine))[

    return filtered_lines



# %%

def get_coners(image_path,plot_edges=False,plot_allLines=False,plot_LineSets=False,plot_filteredLines=False,plot_pts=True,db=1):
    img_name = image_path.split("\\")[-1].split('.')[0]
    # print(img_name)
    thres_l = 250
    thres_h = 500
    img = cv.imread(image_path)
    if plot_allLines:
        img_copy = np.copy(img)
    if plot_LineSets:
        img_copy0 = np.copy(img)
    if plot_filteredLines:
        img_copy1 = np.copy(img)
    if plot_pts:
        img_copy2 = np.copy(img)
    img = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    edges = cv.Canny(img,thres_l,thres_h)

    rho=1; theta=np.pi/180; thres_HL = 50;
    lines = cv.HoughLines(edges,rho,theta,thres_HL)

    lines_set1 = []
    lines_set2 = []

    if lines is not None:
        for l in lines:
```

```
            p, o = l[0]    #dist. and angle
            a, b = np.cos(o), np.sin(o)
            x0, y0 = p*a, p*b
            x1, y1 = int(x0+1000*(-b)), int(y0+1000*(a))
            x2, y2 = int(x0-1000*(-b)), int(y0-1000*(a))
            l_HC = np.cross(np.array([x1,y1,1]),np.array([x2,y2,1]))
            l_HC = l_HC/(l_HC[2] + 1e-8)

            # Check if the line is horizontal (near pi/2)
            if np.abs(o-np.pi/2) < np.pi/4:
                lines_set1.append(np.array([p,o,x1,y1,x2,y2]+list(l_HC)))
                if plot_allLines:
                    cv.line(img_copy, (x1,y1), (x2,y2), (0,200,200), 2)  # Green for horizontal lines
            # Check if the line is vertical (near 0 or pi)
            elif np.abs(o)<np.pi/4 or np.abs(o-np.pi)<np.pi/4:
                lines_set2.append(np.array([p,o,x1,y1,x2,y2]+list(l_HC)))
                if plot_allLines:
                    cv.line(img_copy, (x1,y1), (x2,y2), (0,200,200), 2)  # Red for vertical lines


    filtered_lines_set1 = filter_lines(lines_set1,'horizontal')
    filtered_lines_set2 = filter_lines(lines_set2,'vertical')


    if plot_edges:
        plt.imshow(edges,cmap='gray')
        plt.axis('off')
        plt.title(f'Canny Edges {img_name}')
        plt.show()

    if plot_allLines:
        plt.imshow(img_copy[:,:,::-1])
        plt.axis('off')
        plt.title(f'Hough Transform Lines {img_name}')
        plt.show()

    if plot_LineSets:
        for l in lines_set1:
            cv.line(img_copy0, (int(l[2]),int(l[3])), (int(l[4]),int(l[5])), (0,255,0), 2)
        for l in lines_set2:
            cv.line(img_copy0, (int(l[2]),int(l[3])), (int(l[4]),int(l[5])), (0,0,255), 2)
        plt.imshow(img_copy0[:,:,::-1])
        plt.axis('off')
        plt.title(f'Line Sets {img_name}')
        plt.show()

    if plot_filteredLines:
        for l in filtered_lines_set1:
            cv.line(img_copy1, (int(l[0]),int(l[1])), (int(l[2]),int(l[3])), (0,255,0), 2)
        for l in filtered_lines_set2:
            cv.line(img_copy1, (int(l[0]),int(l[1])), (int(l[2]),int(l[3])), (0,0,255), 2)
        plt.imshow(img_copy1[:,:,::-1])
        plt.axis('off')
        plt.title(f'Filtered Lines {img_name}')
        plt.show()

    point_list = []
    for l1 in filtered_lines_set1:
        for l2 in filtered_lines_set2:
            x,y = intersect_point(l1[4:],l2[4:])
            point_list.append((int(x),int(y)))

    if plot_pts:
        for i,(x,y) in enumerate(point_list):
            pt_size = 4
            # cv.circle(img_copy2, (x,y), 5, (100, 200, 200), -1)
            cv.line(img_copy2, (x - pt_size, y - pt_size), (x + pt_size, y + pt_size), (10, 20, 240), 1)
            cv.line(img_copy2, (x + pt_size, y - pt_size), (x - pt_size, y + pt_size), (10, 20, 240), 1)
            cv.putText(img_copy2, f" {i+1}", (x, y), cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)
        # plt.imshow(img_copy2[:,:,::-1])
        # plt.axis('off')
        # plt.title(f'Points {img_name}')
        # plt.show()
        op_imgs_path = 'op_imgs1/db1' if db==1 else 'op_imgs1/db2'
        cv.imwrite(f'{op_imgs_path}/{img_name}.jpg' ,img_copy2)

    return np.array(point_list)

# for i in range(len(img_list)):
#     image_path = os.path.join(data_path,img_list[i])
#     print(i)
# image_path = os.path.join(data_path,img_list[7])
# point_list = get_coners(image_path,plot_edges=True,plot_allLines=True,plot_LineSets=True,plot_filteredLines=True,plot_pts=True)


# %%
box_size=10#2.4
rows=10
cols=8
world_coord = np.array([(i*box_size, j*box_size) for j in range(rows) for i in range(cols)])

# %%
def get_H(dom,rng):
    A = np.zeros((2*len(dom),9))
    for i,pt in enumerate(dom):
        A[2*i,:] =  np.array([pt[0],pt[1],1,0,0,0,-pt[0]*rng[i][0],-pt[1]*rng[i][0],-rng[i][0]])
        A[2*i+1,:] =  np.array([0,0,0,pt[0],pt[1],1,-pt[0]*rng[i][1],-pt[1]*rng[i][1],-rng[i][1]])
    U_mat,D_mat,V_mat = np.linalg.svd((A.T)@A)
    return V_mat[-1].reshape((3,3))


# %%
def Zhangs_algo(db=1):
    point_list_collection = []
    H_collection = []
    V =[]
```

```
    for i in range(len(img_list)):
        # print(i)
        image_path = os.path.join(data_path,img_list[i])
        point_list = get_coners(image_path,db=db)
        point_list_collection.append(point_list)
        H = get_H(world_coord,point_list)
        H_collection.append(H)
        V.append([H[0,0]*H[0,1],H[0,0]*H[1,1]+H[1,0]*H[0,1],H[1,0]*H[1,1],H[2,0]*H[0,1]+H[0,0]*H[2,1],H[2,0]*H[1,1]+H[1,0]*H[2,1],H[2,0]*H[2,1]])
        V.append([H[0,0]*H[0,0]-H[0,1]*H[0,1], H[0,0]*H[1,0]+H[1,0]*H[0,0]-H[0,1]*H[1,1]-H[1,1]*H[0,1], H[1,0]*H[1,0]-H[1,1]*H[1,1], H[2,0]*H[0,0]+H[0,0]*H[2,0
    V = np.array(V)
    U_mat,D_mat,V_mat = np.linalg.svd((V.T)@V)
    b = V_mat[-1]
    #omega
    w = np.array([[b[0],b[1],b[3]], [b[1],b[2],b[4]], [b[3],b[4],b[5]]])
    print('omega : \n',w)
    print('All eigenvalues of omega positive : ',np.all(np.linalg.eigvals(w)>0))
    #intrinsic parameters
    y0 = (w[0,1]*w[0,2]-w[0,0]*w[1,2])/(w[0,0]*w[1,1]-w[0,1]**2)
    lamd = w[2,2] - ((w[0,2]**2)+y0*(w[0,1]*w[0,2]-w[0,0]*w[1,2]))/w[0,0]
    alpha_x = np.sqrt(lamd/w[0,0])
    alpha_y = np.sqrt((lamd*w[0,0])/(w[0,0]*w[1,1]-(w[0,1]**2)))
    s = -(w[0,1]*(alpha_x**2)*alpha_y)/lamd
    x0 = (s*y0/alpha_y) - ((w[0,2]*alpha_x**2)/lamd)
    K = np.array([[alpha_x,s,x0],[0,alpha_y,y0],[0,0,1]])
    print('K : \n',K)
    K_inv = np.linalg.pinv(K)

    #extrinsic parameters
    Rt_list = []
    for H in H_collection:
        scale_e = 1/np.linalg.norm(K_inv*H[:,0])
        r1 = scale_e*(K_inv@H[:,0])
        r2 = scale_e*(K_inv@H[:,1])
        r3 = np.cross(r1,r2)
        t  = scale_e*(K_inv@H[:,2])
        R = np.vstack((r1,r2,r3)).T
        #conditioning
        U_mat,D_mat,V_mat = np.linalg.svd(R)
        R_c = U_mat@V_mat
        Rt_list.append([R_c,t])

    return point_list_collection,K,Rt_list



# %%
point_list_collection,K,Rt_list=Zhangs_algo(db=db)

# %%
def refineAndPack_camera_param(K,Rt_list,rot=False):
    param_list = []
    param_list += [K[0,0],K[0,1],K[0,2],K[1,1],K[1,2]]
    # refining param for R
    for R,t in Rt_list:
        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]])
        param_list+=list(w)
        param_list+=list(t)
    if rot:
        param_list+=[0.,0.] #inital k1, k2 for radial dist.

    return np.array(param_list)

# param_list = refineAndPack_camera_param(K,Rt_list)
param_list = refineAndPack_camera_param(K,Rt_list,rot=True)

def apply_H(points,H):
    one_vec = np.ones((len(points),1))
    points_HC = np.hstack((points,one_vec))
    proj_pts = (H@(points_HC.T)).T
    proj_pts = proj_pts[:,:2]/proj_pts[:,2].reshape(-1,1)
    return proj_pts


# %%
def proj_func(param_list,world_coord,point_list_collection,img_num=-1,rot=False,print_paraIdx=-1,db=1,meth='basic'):
    K_param = param_list[:5]
    K_mat = np.array([[K_param[0],K_param[1],K_param[2]],[0,K_param[3],K_param[4]],[0,0,1]])
    if rot:
        k1,k2 = param_list[-2:]
        x0,y0 = param_list[2],param_list[4]
    proj_pts_list = []
    for i in range(5,6*(int(len(param_list[5:])/6)),6):
        w = param_list[i:i+3]
        w_mat = np.array([[0,-w[2],w[1]],[w[2],0,-w[0]],[-w[1],w[0],0]])
        phi = np.linalg.norm(w)
        R_mat = np.eye(3) + (np.sin(phi)/phi)*w_mat + (((1-np.cos(phi))/phi**2)*w_mat@w_mat)
        t = param_list[i+3:i+6]
        H = K_mat@np.column_stack((R_mat[:,0],R_mat[:,1],t))
        proj_pts = apply_H(world_coord,H)
        if rot:
            x_proj_pts,y_proj_pts = proj_pts[:,0],proj_pts[:,1]
            r_sq = (x_proj_pts-x0)**2 + (y_proj_pts-y0)**2
            x_r = x_proj_pts + (x_proj_pts-x0)*(k1*r_sq + k2*r_sq**2)
            y_r = y_proj_pts + (y_proj_pts-y0)*(k1*r_sq + k2*r_sq**2)
            proj_pts = np.column_stack((x_r,y_r))
        proj_pts_list.append(proj_pts)

        if -1<print_paraIdx<len(img_list):
            print('K : \n',K_mat)
            print('R : \n',R_mat)
            print('t : \n',t)

    if img_num > -1:
        curr_diff = point_list_collection[img_num] - proj_pts_list[img_num]
        curr_mean = np.mean(np.linalg.norm(curr_diff,axis=1))
```

```
        curr_var = np.var(np.linalg.norm(curr_diff,axis=1))
        print(f"mean: {curr_mean} , var : {curr_var}")

        ip_img_db_path = 'op_imgs1/db1' if db==1 else 'op_imgs1/db2'
        img_num_path = f'{ip_img_db_path}/Pic_{img_num+1}.jpg'
        img_num_ip = cv.imread(img_num_path)
        for pt in proj_pts_list[img_num]:
            pt_size = 4
            # cv.circle(img_num_ip, (int(pt[0]),int(pt[1])), 3, (200, 200, 100), -1)
            cv.line(img_num_ip, (int(pt[0]) - pt_size, int(pt[1]) - pt_size), (int(pt[0]) + pt_size, int(pt[1]) + pt_size), (20, 240, 10), 1)
            cv.line(img_num_ip, (int(pt[0]) + pt_size, int(pt[1]) - pt_size), (int(pt[0]) - pt_size, int(pt[1]) + pt_size), (20, 240, 10), 1)
        # plt.imshow(img_num_ip[:,:,::-1])
        # plt.axis('off')
        # plt.title(f'Pic {img_num}')
        # plt.show()
        op_img_db_path = 'op_imgs2/db1/' if db==1 else 'op_imgs2/db2/'
        op_img_db_path += 'rot_' if rot==True else ''
        op_img_db_path += meth    #baisc/lm
        cv.imwrite(f'{op_img_db_path}/Pic_{img_num+1}.jpg' ,img_num_ip)

    proj_pts_list = np.concatenate(proj_pts_list)
    point_list_collection = np.concatenate(point_list_collection)
    diff = (point_list_collection - proj_pts_list)

    return diff.flatten()


# %%
for i in range(len(img_list)):
    image_path = os.path.join(data_path,img_list[i])
    print(i)
    # proj_func(param_list,world_coord,point_list_collection,img_num=i,db=db,meth='basic',rot=False)
    proj_func(param_list,world_coord,point_list_collection,img_num=i,rot=True,db=db,meth='basic')

# %%
# res_lsq = optim.least_squares(proj_func,param_list,method='lm',args=(world_coord,point_list_collection,-1))
res_lsq = optim.least_squares(proj_func,param_list,method='lm',args=(world_coord,point_list_collection,-1,True))

# %%
res_lsq

# %%
for i in range(len(img_list)):
    image_path = os.path.join(data_path,img_list[i])
    print(i)
    # proj_func(res_lsq.x,world_coord,point_list_collection,img_num=i,db=db,meth='lm',rot=False)
    proj_func(res_lsq.x,world_coord,point_list_collection,img_num=i,rot=True,db=db,meth='lm')

# %%
def print_param_fr_list(params,idx,rot=False):
    if -1<idx<len(img_list):
        # print(f'Pic_{idx+1} : \n')
        K = np.array([[params[0],params[1],params[2]],[0,params[3],params[4]],[0,0,1]])
        i = 5+6*idx
        w = param_list[i:i+3]
        w_mat = np.array([[0,-w[2],w[1]],[w[2],0,-w[0]],[-w[1],w[0],0]])
        phi = np.linalg.norm(w)
        R = np.eye(3) + (np.sin(phi)/phi)*w_mat + (((1-np.cos(phi))/phi**2)*w_mat@w_mat)
        t = param_list[i+3:i+6]
        if rot:
            k1_k2 = params[-2:]
            print(f'rad_dist_params (k1,k2) : {k1_k2}')
        return K,R,t
    else: print('invalid idx')

img_idx = 7
K_idx,R_idx,t_idx = print_param_fr_list(res_lsq.x,img_idx,rot=True)
print('K : \n',K_idx)
print('R : \n',R_idx)
print('t : \n',t_idx)
# print(print_param_fr_list(res_lsq.x,10))

# %%
def plot_all_camera_pose(calib_coord,params):
    def plot_camera_pose(calib_coord,params,img_idx,color='b',ax=None):
        K_idx,R_idx,t_idx = print_param_fr_list(params,img_idx)
        # camera center
        C = -(R_idx.T)@t_idx
        # camera axes
        X_cam = np.array([1,0,0])
        Y_cam = np.array([0,1,0])
        Z_cam = np.array([0,0,1])
        # world coord of camera axes
        X_w = (R_idx.T)@X_cam + C
        Y_w = (R_idx.T)@Y_cam + C
        Z_w = (R_idx.T)@Z_cam + C
        # plot the camera axes (red for X, green for Y, blue for Z)
        ax.quiver(C[0],C[1],C[2], X_w[0]-C[0],X_w[1]-C[1], X_w[2]-C[2], color='r', length=50)
        ax.quiver(C[0],C[1],C[2], Y_w[0]-C[0],Y_w[1]-C[1], Y_w[2]-C[2], color='g', length=50)
        ax.quiver(C[0],C[1],C[2], Z_w[0]-C[0],Z_w[1]-C[1], Z_w[2]-C[2], color='b', length=50)
        # ax.quiver(C[0],C[1],C[2], (X_w[0]-C[0])/np.linalg.norm((X_w[0]-C[0])),(X_w[1]-C[1])/np.linalg.norm((X_w[1]-C[1])), (X_w[2]-C[2])/np.linalg.norm((X_w[
        # ax.quiver(C[0],C[1],C[2], (Y_w[0]-C[0])/np.linalg.norm((Y_w[0]-C[0])),(Y_w[1]-C[1])/np.linalg.norm((Y_w[1]-C[1])), (Y_w[2]-C[2])/np.linalg.norm((Y_w[
        # ax.quiver(C[0],C[1],C[2], (Z_w[0]-C[0])/np.linalg.norm((Z_w[0]-C[0])),(Z_w[1]-C[1])/np.linalg.norm((Z_w[1]-C[1])), (Z_w[2]-C[2])/np.linalg.norm((Z_w[
        # plot principal plane
        pl_size = 30
        rect_pp_pts_cam = pl_size*np.array([[-1,1,0],[-1,-1,0],[1,-1,0],[1,1,0]])
        rect_pp_pts = ((R_idx.T)@rect_pp_pts_cam.T + C.reshape(-1,1)).T
        rect_pp = [[rect_pp_pts[0],rect_pp_pts[1],rect_pp_pts[2],rect_pp_pts[3]]]
        ax.add_collection3d(art3d.Poly3DCollection(rect_pp, color=color, alpha=0.3, lw=0.5))
        # calibration patch
        x_min,y_min = calib_coord[0]
        x_max,y_max = calib_coord[len(calib_coord)-1]
        rect_c_pts = np.array([[x_min,y_min,0],[x_max,y_min,0],[x_max,y_max,0],[x_min,y_max,0]])
        rect_c = [[rect_c_pts[0],rect_c_pts[1],rect_c_pts[2],rect_c_pts[3]]]
        ax.add_collection3d(art3d.Poly3DCollection(rect_c, color='k'))
```

```
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    for i in range(len(img_list)):  #40
        plot_camera_pose(calib_coord,params,img_idx=i,color=np.random.rand(3,),ax=ax)

    ax.view_init(elev=-30, azim=90)
    ax.set_xlim(-150, 250)
    ax.set_ylim(-200, 150)
    ax.set_zlim(-200, 0)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('Camera Poses',pad=20)
    plt.show()

plot_all_camera_pose(world_coord,param_list)
plot_all_camera_pose(world_coord,res_lsq.x)


# %%


# %%
```