ECE 661: Homework 8
Yechen Liu
Fall 2022

1 Theory Question

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

Answer: No, we cannot see the image of Absolute Conic ω. Because the pixels on the conic ω satisfy $\boldsymbol{x}^T\omega\boldsymbol{x} = 0$, and $\omega = \boldsymbol{K}^{-T}\boldsymbol{K}^{-1}$ is positive definite ($\boldsymbol{x}^T\omega\boldsymbol{x} > 0$), which means these pixels are imaginary. In camera calibration, the pattern plane samples the Absolute Conic at two circular points. The image of these two circular points must be on the conic ω. By obtaining at least three pairs of circular points by posing the pattern at different positions, the intrinsic parameters $\boldsymbol{K}$ can be estimated.

2 Camera Calibration - Zhang's Algorithm

2.1 Corner Detection

2.1.1 Canny Edge Detector

For this homework, I use the off-the-shelf Canny edge detector function `cv2.Canny` from OpenCV. The two threshold parameters are set to 400 and 300 respectively.

2.1.2 Hough Transform

For this homework, I use the open-source implementation of the Hough line transform `cv2.HoughLines` function from OpenCV. The threshold parameter is set to 50.

However, multiple lines will be generated for one true line. To improve the accuracy of the corner detection, the redundant lines must be eliminated. First, we need to separate vertical lines and horizontal lines. For the pattern we used in this homework, there are 8 true vertical lines and 10 true horizontal lines. Then we need to group vertical lines and horizontal lines separately. The critical step is to find a metric of lines by which they are grouped together. The metric I used is r (the distance from the line to origin) for horizontal lines and $|r|$ for vertical lines. Once the lines are gouped up, the final step is to generate an "average" line for each group of lines (Note: vertical lines and horizontal lines use different average function).

2.1.3 Establish Corner Correspondences

The previous step will generate exact 8 vertical lines and 10 horizontal lines. The corners are the intersection of these lines and thus 80 corners will be found. The world coordinates corresponding to these corners are generated by assuming 1) the calibration pattern is in the $Z = 0$ plane of the

world frame 2) the world coordinate of the first corner is (0,0) and 3) the distance between two contiguous corners is 10.

2.2 Camera Calibration

The pipeline of Zhang's Camera Calibration Algorithm involves three stages to extract camera parameters step by step: the intrinsic parameter, the extrinsic parameter, and the radial distortion parameter. We will use Levenberg-Marquadt(LM) non-linear optimization to refine the camera parameters.

2.2.1 Estimate the Intrinsic Parameters

By assuming the calibration pattern is in the $Z = 0$ plane of the world frame, the relationship between the homogeneous representation of a pixel coordinates $\boldsymbol{x} = (x, y, w)^T$ and the homogeneous representation of the corresponding world coordinates $\mathbf{x} = (\mathrm{x}, \mathrm{y}, \mathrm{z}, \mathrm{w})$ is

$$\boldsymbol{x} = K[R|\boldsymbol{t}] \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ 0 \\ \mathrm{w} \end{bmatrix} = H\boldsymbol{x}_M$$

where K is the camera intrinsic parameter, R is the world-to-camera rotation matrix, $\boldsymbol{t}$ is the world-to-camera translation vector, H is the homography estimated by corner correspondences from previous step, and $\boldsymbol{x}_M = [\mathrm{x}, \mathrm{y}, \mathrm{w}]^T$.

From the theory question, we know that the image of the Absolute Conic Ω_∞ is given by $\omega = \boldsymbol{K}^{-T}\boldsymbol{K}^{-1}$. And the two circular points on the image conic ω gives us two equations

$$\begin{cases} \boldsymbol{h}_1^T \omega \boldsymbol{h}_1 = \boldsymbol{h}_2^T \omega \boldsymbol{h}_2 \\ \boldsymbol{h}_1^T \omega \boldsymbol{h}_2 = 0 \end{cases}$$

Since ω is always a 3×3 symmetric matrix, there are only 6 unknowns in ω. Therefore the above two equation can be expressed as

$$\begin{cases} \boldsymbol{v}_{12}^T \boldsymbol{b} = 0 \\ (\boldsymbol{v}_{11} - \boldsymbol{v}_{22})^T \boldsymbol{b} = 0 \end{cases}$$

where

$$\boldsymbol{v}_{ij} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} \omega_{11} \\ \omega_{12} \\ \omega_{22} \\ \omega_{13} \\ \omega_{23} \\ \omega_{33} \end{bmatrix}$$

Each position of the camera gives us 2 equations represented by $\boldsymbol{V}\boldsymbol{b} = \boldsymbol{0}$. Using $\boldsymbol{V}_i$ to represent the 2×6 matrix $\boldsymbol{V}$ for the ith camera position, we can stack up all the resulting equations as $\begin{bmatrix} \boldsymbol{V}_1 \\ \vdots \\ \boldsymbol{V}_n \end{bmatrix} \boldsymbol{b} = \boldsymbol{0}$. These can be solved by using linear least-squares method.

Now we have the image ω of the Absolute Conic, next we want to find the camera intrinsic matrix $K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$ and we know $\omega = K^{-T}K^{-1}$. The problem to solve this equation is that we cannot set element-by-element equalities between the matrices on the two sides because the entity on the left is homogeneous whereas the entity on the right is inhomogeneous.

Zhang's Algorithm addresses this issue by implicitly introducing a new variable λ as shown by the following solution for the intrinsic parameters

$$
\begin{aligned}
x_0 &= \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}^2} \\
\lambda &= \omega_{33} - \frac{\omega_{13}^2 + x_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}} \\
\alpha_x &= \sqrt{\frac{\lambda}{\omega_{11}}} \\
\alpha_y &= \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}^2}} \\
s &= -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda} \\
y_0 &= \frac{sx_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}
\end{aligned}
$$

2.2.2 Calculate the Extrinsic Parameters

As shown previously, the relationship between the homograhpy H and the extrinsic parameters is $K^{-1}\begin{bmatrix}\boldsymbol{h}_1 & \boldsymbol{h}_2 & \boldsymbol{h}_3\end{bmatrix} = \begin{bmatrix}\boldsymbol{r}_1 & \boldsymbol{r}_2 & \boldsymbol{t}\end{bmatrix}$. Again, the left is homogeneous and the right must consider to be homogeneous. Therefore, the scale factor $\xi = \frac{1}{||K^{-1}\boldsymbol{h}_1||}$ should be added. Then the actual formulas we use for $\boldsymbol{R}$ and $\boldsymbol{t}$ become:

$$
\begin{cases}
\boldsymbol{r}_1 = \xi K^{-1}\boldsymbol{h}_1 \\
\boldsymbol{r}_2 = \xi K^{-1}\boldsymbol{h}_2 \\
\boldsymbol{r}_3 = \boldsymbol{r}_1 \times \boldsymbol{r}_2 \\
\boldsymbol{t} \ = \xi K^{-1}\boldsymbol{h}_3
\end{cases}
$$

Condition the Rotation Matrix

To make the rotation matrix orthonormal, we need to solve the following optimization problem

$$\min_R ||\boldsymbol{R} - \boldsymbol{Q}||_F^2 \quad \text{subject to } \boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}$$

where $\boldsymbol{Q}$ represents the computed rotation matrix and $\boldsymbol{R}$ represents the best orthonormal approximation. The solution to this optimization problem is $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{V}^T$ where $\boldsymbol{U}$ and $\boldsymbol{V}^T$ are the SVD components.

The rotation matrix R has 9 elements but only 3 DoF. In any optimization algorithm, the number of variables used to represent any entity must strictly equal the DoF of the entity. Therefore, we need a 3-parameter representation of a rotation matrix $\boldsymbol{R}$, which is Rodrigues Representation - a

three element vector $\boldsymbol{w} = \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix}$, the direction of the axis of the rotation is encoded in the unit vector $\frac{\boldsymbol{w}}{||\boldsymbol{w}||}$ and the angle φ of clockwise rotation around this axis by the magnitude $\varphi = ||\boldsymbol{w}||$.

To construct the vector $\boldsymbol{w}$ for a given $\boldsymbol{R}$, we use

$$\varphi = \cos^{-1}\frac{\text{trace}(\boldsymbol{R}) - 1}{2}$$
$$\boldsymbol{w} = \frac{\varphi}{2\sin\varphi}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}$$

To construct the $\boldsymbol{R}$ matrix for a given vector $\boldsymbol{w}$, we use

$$\boldsymbol{R} = \boldsymbol{I}_{3\times 3} + \frac{\sin\varphi}{\varphi}[\boldsymbol{w}]_X + \frac{1-\cos\varphi}{\varphi^2}[\boldsymbol{w}]_X^2$$

where

$$\varphi = ||\boldsymbol{w}|| \qquad [\boldsymbol{w}]_X = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_z \\ -w_y & w_x & 0 \end{bmatrix}$$

2.2.3* Incorporate Radial Distortion (Extra Credits)

To calibrate a camera with radial distortion, we first project a world coordinate into the image plane using the pinhole model and the tentative set of calibration parameters and denote the projected position of a pixel as $(\hat{x}, \hat{y})$. Let $(\hat{x}_{\text{rat}}, \hat{y}_{\text{rat}})$ be the pixel coordinates that would be predicted if the radial distortion is included

$$\hat{x}_{\text{rat}} = \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4]$$
$$\hat{y}_{\text{rat}} = \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4]$$

where $r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2$ and (x_0, y_0) are the principal point on the image. We will concatenate two new parameters k_1 and k_2 to the minimization variable $\boldsymbol{p}$ (discussed in the next section).

2.2.4 Refine the Calibration Parameters

The camera calibration parameters (intrinsic matrix K, extrinsic matrix $[\boldsymbol{R}|\boldsymbol{t}]$, and radial distortion parameters k_1 and k_2) up to this point is still not optimal. To get a sense of how accurate the calculated calibration parameters are, we reprojet a set of salient points from the calibration pattern into the image planes for each of the camera positions. The Euclidean distance between the projected pixels and where they actually occur in the image gives us a meaningful metric. The aggregated distances for all salient points and for all the camera positions could be used as cost function for nonlinear least-squares minimization, which is expressed as

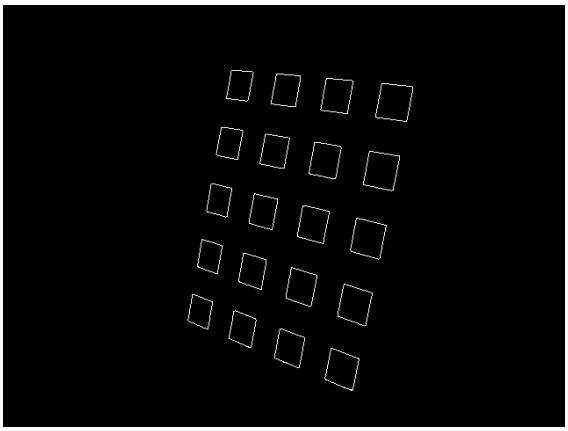
$$d_{\text{geom}}^2 = ||\boldsymbol{X} - \boldsymbol{f}(\boldsymbol{p})||^2 = \sum_i \sum_j ||\boldsymbol{x}_{ij} - \hat{\boldsymbol{x}}_{ij}||^2 = \sum_i \sum_j ||\boldsymbol{x}_{ij} - K\begin{bmatrix} \boldsymbol{r}_{i,1} & \boldsymbol{r}_{i,2} & \boldsymbol{t}_i \end{bmatrix} \boldsymbol{x}_{M,j}||^2$$

where $\boldsymbol{p} = (K, \boldsymbol{w}_i, \boldsymbol{t}_i|_{i=1,2,\ldots})$ without considering radial distortion or $\boldsymbol{p} = (K, \boldsymbol{w}_i, \boldsymbol{t}_i|_{i=1,2,\ldots}, k_1, k_2)$ with radial distortion.

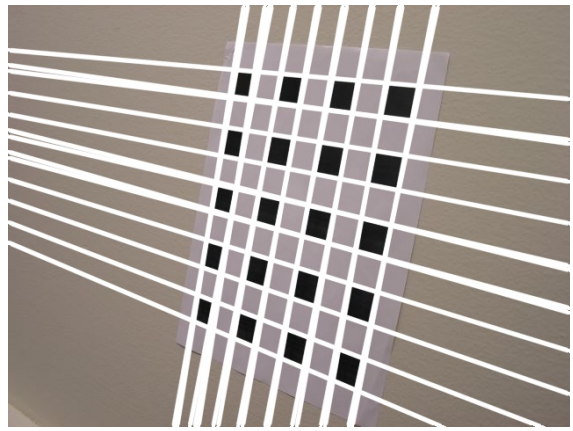
3 Results

3.1 Given Dataset

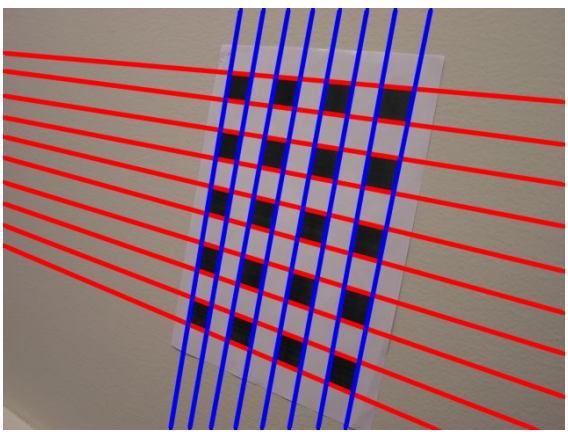
Corner Detection



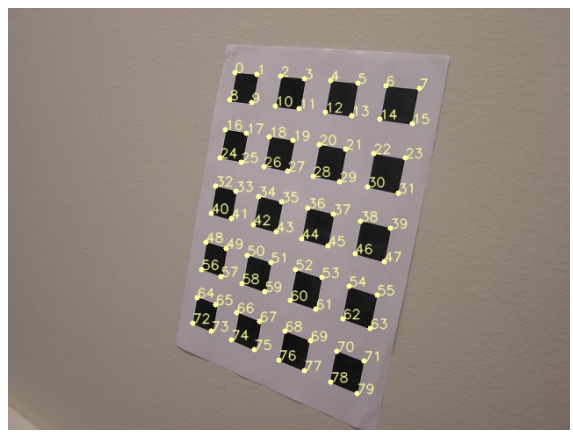
(a) Canny edge detection



(b) Hough transform

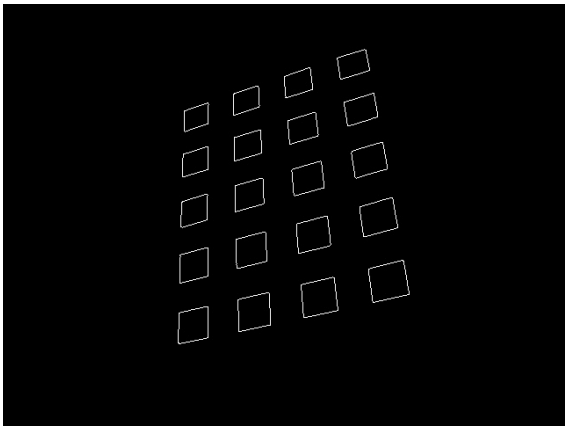


(c) Filtered lines

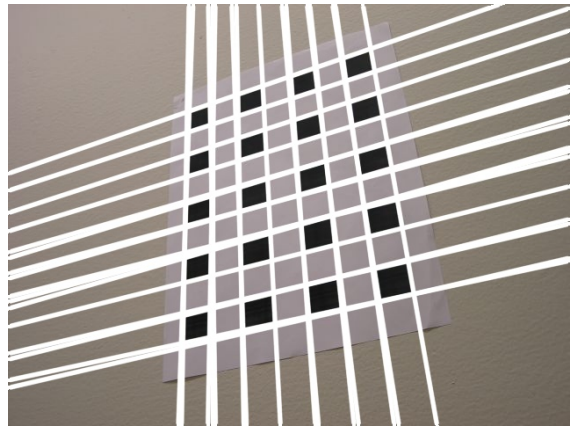


(d) Corners and labels

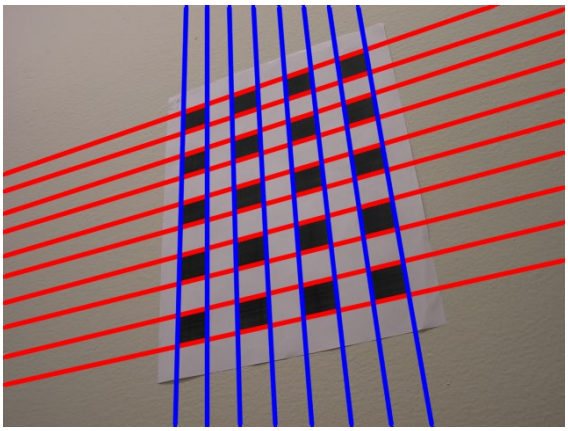
Figure 1: Corner detection steps for picture 1.



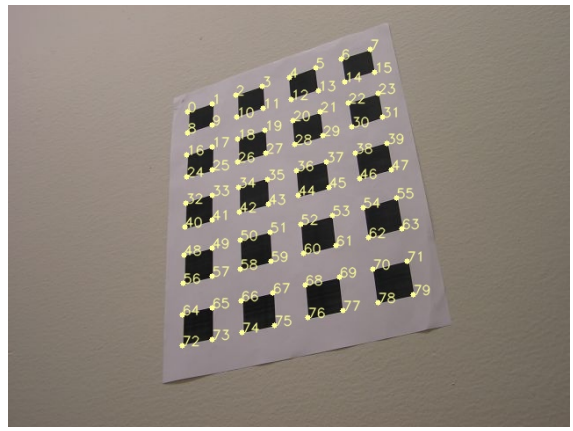
(a) Canny edge detection



(b) Hough transform



(c) Filtered lines



(d) Corners and labels

Figure 2: Corner detection steps for picture 8.

Camera Calibration

Before LM, The intrinsic matrix K is

$$K = \begin{bmatrix} 717.5 & 0.5909 & 237.4 \\ 0 & 714.2 & 317.8 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of the view 1 is

$$[\boldsymbol{R}_1|\boldsymbol{t}_1] = \begin{bmatrix} 0.762 & -0.206 & 0.614 & 6.71 \\ 0.230 & 0.972 & 0.040 & -78.0 \\ -0.605 & 0.111 & 0.788 & 227.0 \end{bmatrix}$$

The rotation and translation parameters of the view 8 is

$$[\boldsymbol{R}_8|\boldsymbol{t}_8] = \begin{bmatrix} 0.929 & 0.026 & 0.369 & -11.0 \\ -0.208 & 0.862 & 0.462 & -66.6 \\ -0.306 & -0.506 & 0.806 & 241.4 \end{bmatrix}$$

After LM, The intrinsic matrix K is

$$K = \begin{bmatrix} 722.4 & 1.74 & 321.5 \\ 0 & 719.7 & 238.3 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of the view 1 is

$$[\boldsymbol{R}_1|\boldsymbol{t}_1] = \begin{bmatrix} 0.787 & -0.181 & 0.590 & -19.0 \\ 0.199 & 0.980 & 0.035 & -50.3 \\ -0.584 & 0.090 & 0.806 & 218.7 \end{bmatrix}$$

The rotation and translation parameters of the view 8 is

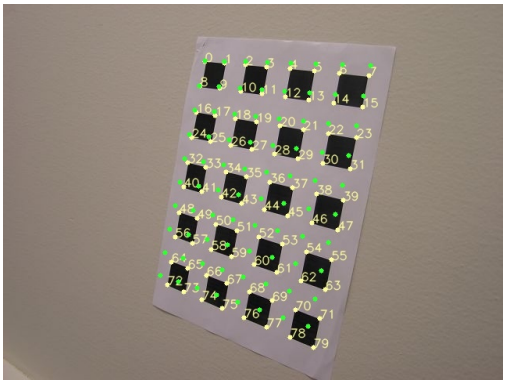
$$[\boldsymbol{R}_8|\boldsymbol{t}_8] = \begin{bmatrix} 0.920 & 0.061 & 0.387 & -37.6 \\ -0.252 & 0.849 & 0.465 & -38.0 \\ -0.300 & -0.525 & 0.796 & 233.0 \end{bmatrix}$$

Considering radial distortion, the radial distortion parameters

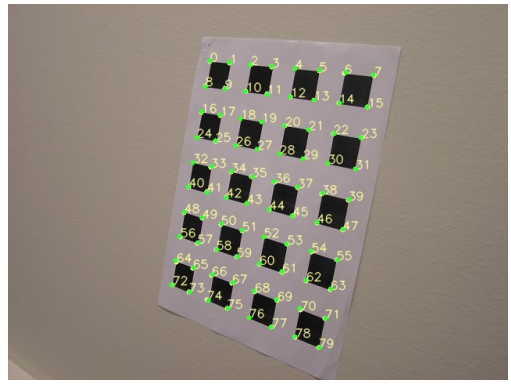
$$\boldsymbol{k} = \begin{bmatrix} -2.966 \times 10^{-7} & 1.948 \times 10^{-12} \end{bmatrix}$$

	Before LM	After LM	Consider Radial Distortion*
Error Mean (view 1)	12.767	0.869	0.821
Error Variance (view1)	39.887	0.173	0.161
Error Mean (view 8)	12.302	0.764	0.769
Error Variance (view8)	84.553	0.134	0.126

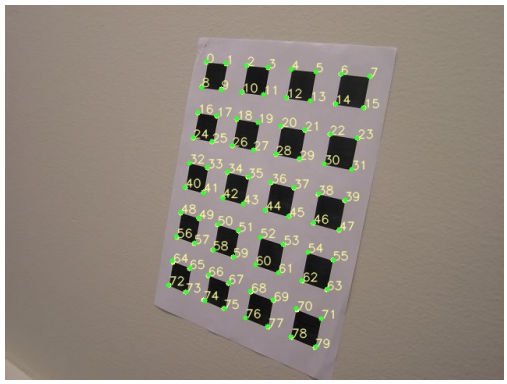
Table 1: Reductions in reprojection errors



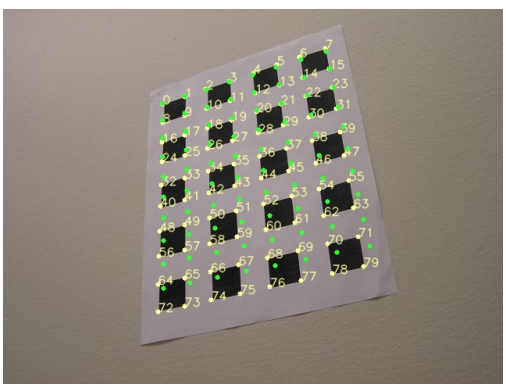
(a) Before LM (view 1)



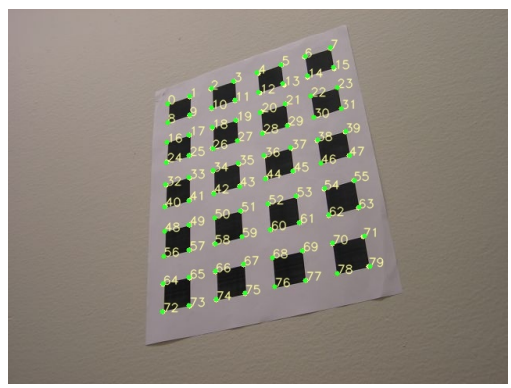
(b) After LM (view 1)



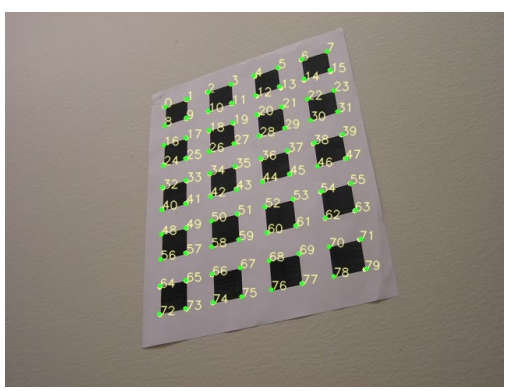
(c) Considering radial distortion (view 1)



(d) Before LM (view 8)



(e) After LM (view 8)



(f) Considering radial distortion (view 8)

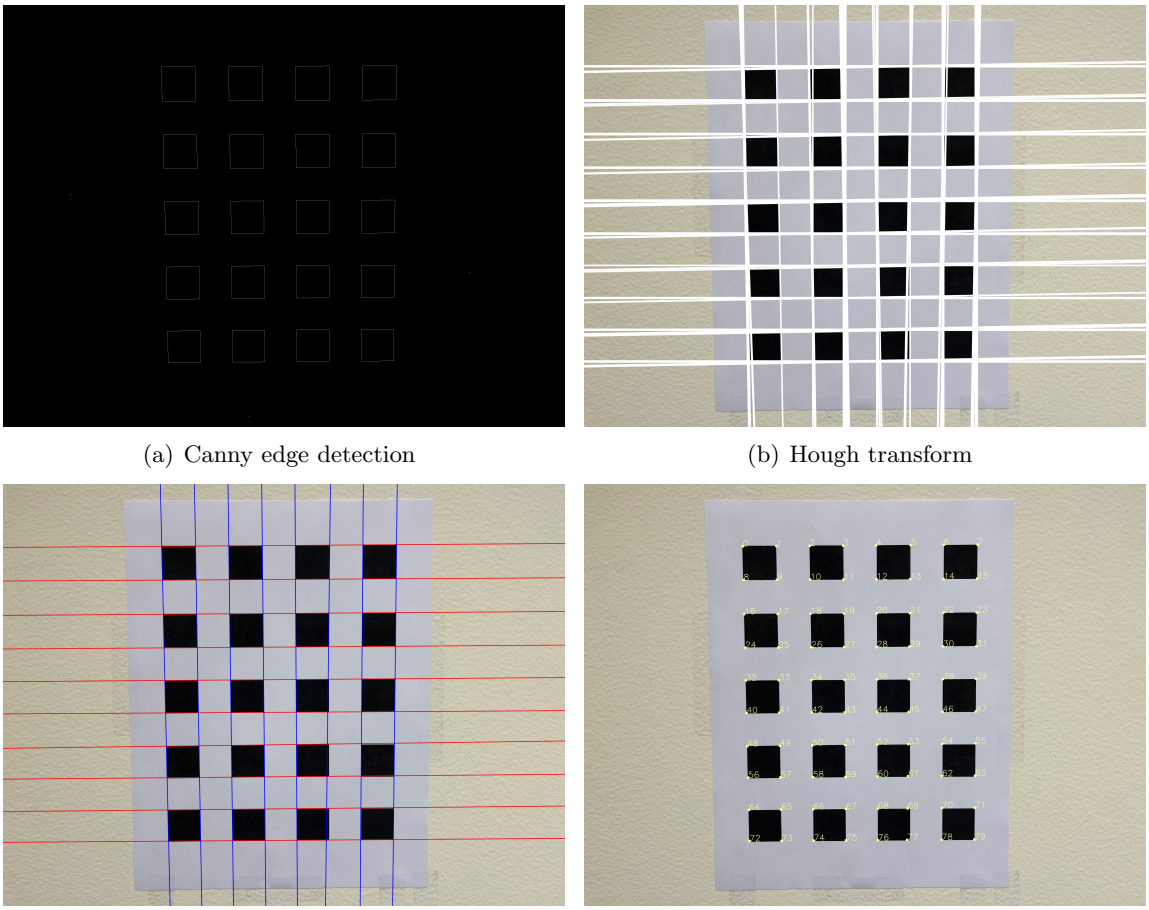
Figure 3: Reprojected green coners and original yellow corners.

3.2 My Dataset

Ground-Truth Parameters

Camera focal length (Distance from COP to image plane): 5.1 mm.
Distance from image plane to 'Fixed Image' in view 1: 30.5 cm.
Physical/Digital length of the pattern: 2.4 cm / 10
Digital distance from COP to 'Fixed Image' in view 1 should be: $(30.5 + 0.51)/(2.4/10) = 129.2$

Corner Detection



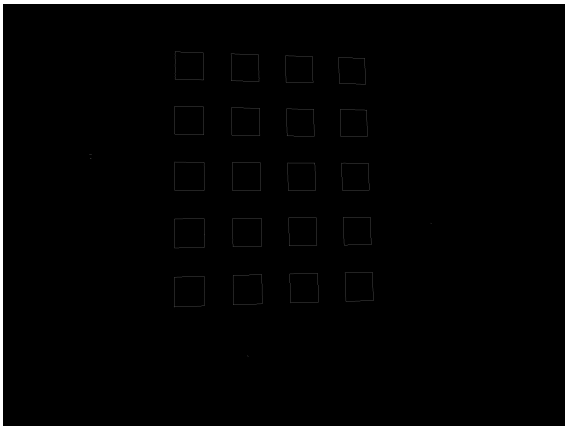
(a) Canny edge detection

(b) Hough transform

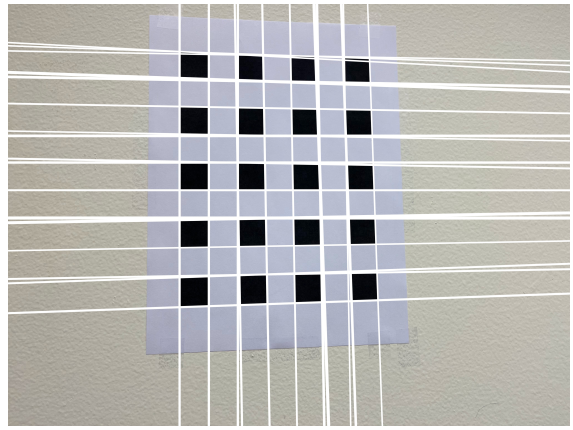
(c) Filtered lines

(d) Corners and labels

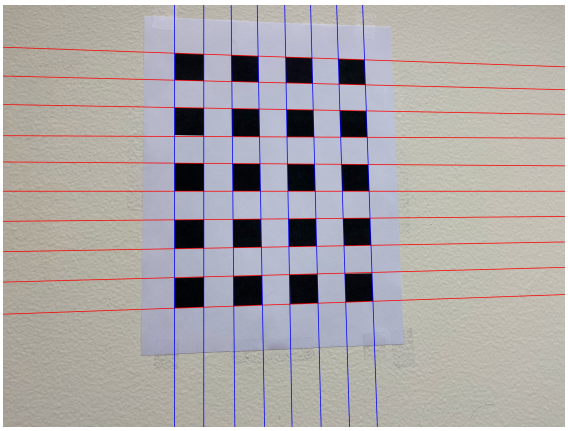
Figure 4: Corner detection steps for picture 1.



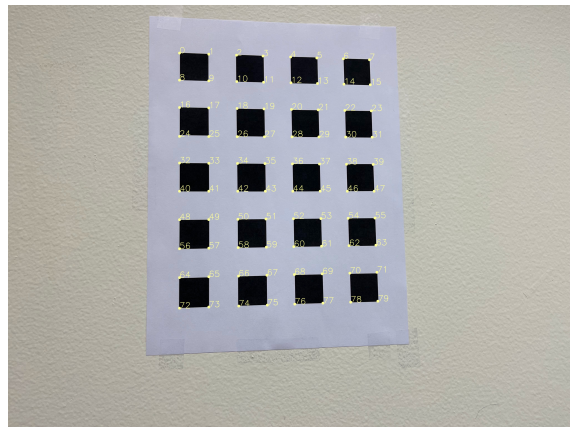
(a) Canny edge detection



(b) Hough transform



(c) Filtered lines



(d) Corners and labels

Figure 5: Corner detection steps for picture 8.

Camera Calibration

Before LM, The intrinsic matrix K is

$$K = \begin{bmatrix} 3125.6 & -7.021 & 1501.5 \\ 0 & 3129.6 & 2057.0 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of the view 1 is

$$[\boldsymbol{R}_1|\boldsymbol{t}_1] = \begin{bmatrix} 0.9999 & -0.0126 & 0.0012 & -15.2 \\ -0.0124 & 0.9977 & -0.0671 & -66.7 \\ -0.0021 & 0.0671 & 0.9977 & 130.2 \end{bmatrix}$$

The rotation matrix is very close to Identity matrix, which means nearly no rotation. And the Z-coordinate is 130.2, which is very close to 129.2.

The rotation and translation parameters of the view 8 is

$$[\boldsymbol{R}_8|\boldsymbol{t}_8] = \begin{bmatrix} 0.994 & 0.023 & -0.110 & -13.6 \\ -0.014 & 0.997 & 0.082 & -84.1 \\ 0.112 & -0.080 & 0.991 & 153.8 \end{bmatrix}$$

After LM, The intrinsic matrix K is

$$K = \begin{bmatrix} 3133.2 & -7.394 & 2053.4 \\ 0 & 3135.1 & 1511.1 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of the view 1 is

$$[\boldsymbol{R}_1|\boldsymbol{t}_1] = \begin{bmatrix} 0.99998 & -0.00659 & 0.00015 & -38.2 \\ -0.00659 & 0.99776 & -0.06656 & -44.3 \\ -0.00028 & 0.06656 & 0.99778 & 130.4 \end{bmatrix}$$

The rotation matrix is more close to Identity matrix. And the Z-coordinate is 130.4, which is a still close to 129.2.

The rotation and translation parameters of the view 8 is

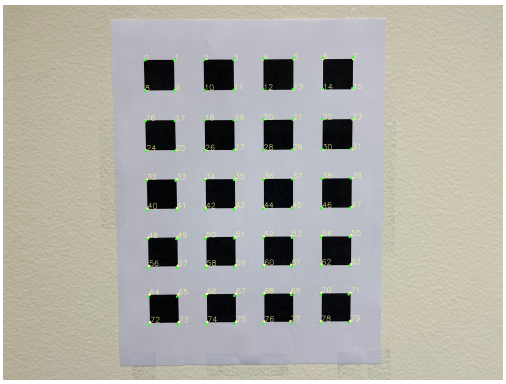
$$[\boldsymbol{R}_8|\boldsymbol{t}_8] = \begin{bmatrix} 0.993 & 0.021 & -0.119 & -41.2 \\ -0.011 & 0.996 & 0.084 & -58.3 \\ -0.120 & -0.082 & 0.989 & 156.9 \end{bmatrix}$$

Considering radial distortion, the radial distortion parameters

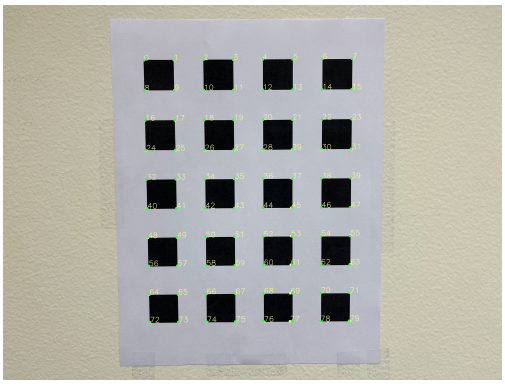
$$\boldsymbol{k} = \begin{bmatrix} 1.112 \times 10^{-8} & -4.204 \times 10^{-15} \end{bmatrix}$$

	Before LM	After LM	Consider Radial Distortion*
Error Mean (view 1)	11.535	2.991	2.799
Error Variance (view1)	49.212	9.073	8.439
Error Mean (view 8)	22.613	3.071	2.829
Error Variance (view8)	93.981	1.751	1.851

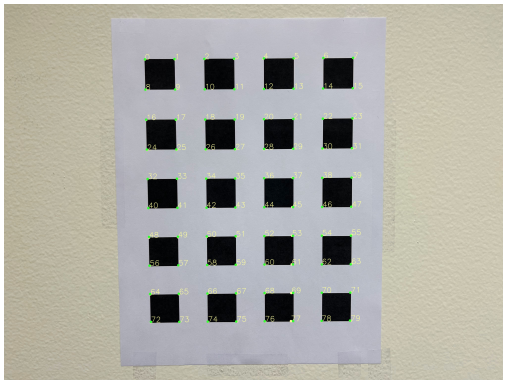
Table 2: Reductions in reprojection errors



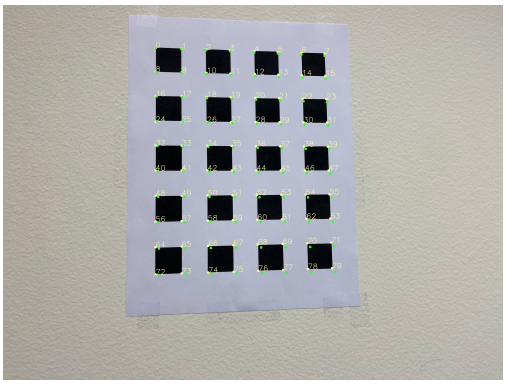
(a) Before LM (view 1)



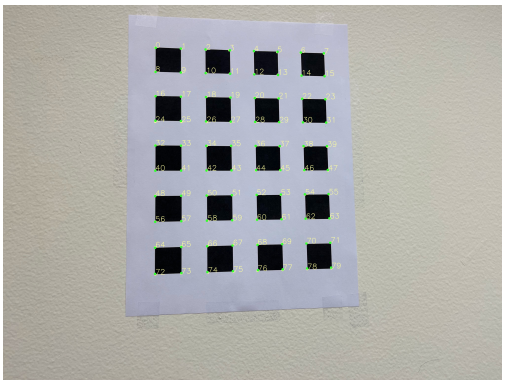
(b) After LM (view 1)



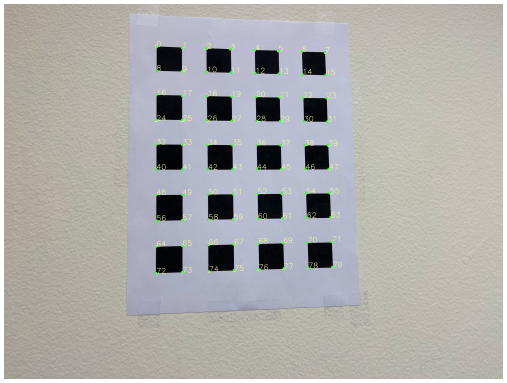
(c) Considering radial distortion (view 1)



(d) Before LM (view 8)



(e) After LM (view 8)



(f) Considering radial distortion (view 8)

Figure 6: Reprojected green coners and original yellow corners.

Source Code

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

# In[202]:


import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import math
import cv2
from scipy.optimize import least_squares
import os
from scipy.stats import gmean


# In[251]:


def find_intersection(hline, vline):
    hpt1 = [hline[0][0], hline[0][1], 1]
    hpt2 = [hline[1][0], hline[1][1], 1]
    vpt1 = [vline[0][0], vline[0][1], 1]
    vpt2 = [vline[1][0], vline[1][1], 1]

    homo_hline = np.cross(hpt1, hpt2)
    homo_vline = np.cross(vpt1, vpt2)
    inter_point = np.cross(homo_hline, homo_vline)
    return (int(inter_point[0]/inter_point[2]), int(inter_point[1]/inter_point[2]))

def preprocess_to_get_points(img_path, save_img=False, output_name=None, canny_t1=400, canny_t2=300, hough=50):
    img = cv2.imread(img_path)
    if save_img:
        cdst = np.copy(img)
        cdst2 = np.copy(img)
        cdst3 = np.copy(img)

    img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    edges = cv2.Canny(img, canny_t1, canny_t2)
    if save_img:
        cv2.imwrite('HW8-Files/Result2/' + output_name + '_edges.jpg', edges)

    lines = cv2.HoughLines(edges, 1, np.pi / 180, hough, None, 0, 0)
    vertical_lines = []
    horizontal_lines = []

    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            if (rho > 0 and theta < np.pi/4) or (rho < 0 and theta > 3*np.pi/4):
                vertical_lines.append(tuple((np.abs(lines[i][0][0]), lines[i][0][1], np.sign(rho))))
                #vertical_lines.append((lines[i][0][0], lines[i][0][1]))
            else:
                #horizontal_lines.append(tuple((np.abs(lines[i][0][0]), lines[i
```

```
][0][1], np.sign(rho))))
                horizontal_lines.append((lines[i][0][0], lines[i][0][1]))
            a = math.cos(theta)
            b = math.sin(theta)
            x0 = a * rho
            y0 = b * rho
            pt1 = (int(x0 + 5000*(-b)), int(y0 + 5000*(a)))
            pt2 = (int(x0 - 5000*(-b)), int(y0 - 5000*(a)))
            if save_img:
                cv2.line(cdst, pt1, pt2, (255,255,255), 10, cv2.LINE_AA)

        vdtype = [('rho', np.float32), ('theta', np.float32), ('sign', np.int)]
        hdtype = [('rho', np.float32), ('theta', np.float32)]
        horizontal_lines = np.array(horizontal_lines, dtype=hdtype)
        horizontal_lines = np.sort(horizontal_lines, order='rho')
        vertical_lines = np.array(vertical_lines, dtype=vdtype)
        vertical_lines = np.sort(vertical_lines, order='rho')

        # Determine the threshold of two lines belong to the same group
        distance = [horizontal_lines[i+1][0]-horizontal_lines[i][0] for i in range(
    len(horizontal_lines)-1)]
        horizontal_threshold = np.sort(distance)[-9]
        distance = [vertical_lines[i+1][0]-vertical_lines[i][0] for i in range(len(
    vertical_lines)-1)]
        vertical_threshold = np.sort(distance)[-7]

        # Aggregate the lines to groups
        previous_rho = horizontal_lines[0][0]
        real_hlines = []
        aggregated_lines = []
        for line in horizontal_lines:
            rho = line[0]
            theta = line[1]
            #sign = line[2]
            if rho - previous_rho >= horizontal_threshold:
                assert(len(aggregated_lines) > 0)
                real_hlines.append(aggregated_lines)
                aggregated_lines = []
            #aggregated_lines.append((rho, theta, sign))
            aggregated_lines.append((rho, theta))
            previous_rho = rho
        real_hlines.append(aggregated_lines)

        previous_rho = vertical_lines[0][0]
        real_vlines = []
        aggregated_lines = []
        for line in vertical_lines:
            rho = line[0]
            theta = line[1]
            theta = line[1]
            sign = line[2]
            if rho - previous_rho >= vertical_threshold:
                assert(len(aggregated_lines) > 0)
                real_vlines.append(aggregated_lines)
                aggregated_lines = []
            aggregated_lines.append((rho, theta, sign))
            #aggregated_lines.append((rho, theta))
            previous_rho = rho
        real_vlines.append(aggregated_lines)
```

```python

        vlines = []
        hlines = []
        for i, lines_type in enumerate([real_hlines, real_vlines]):
            if i == 0:
                assert(len(lines_type) == 10)
            else:
                assert(len(lines_type) == 8)
            for lines in lines_type:
                if i == 0:
                    lines = [[line[0], line[1]] if line[0]>0 else
                             [-line[0], line[1] - np.pi] for line in lines]
                if i == 1:
                    lines = [[line[0], line[1]] if line[2]==1 else
                             [line[0], line[1] - np.pi] for line in lines]
                #avg_line = np.median(lines, axis=0)
                avg_rho = gmean(np.array(lines)[:,0])
                avg_theta = np.mean(np.array(lines)[:,1])
                if avg_theta < 0:
                    avg_rho = - avg_rho
                    avg_theta = np.pi + avg_theta
                a = math.cos(avg_theta)
                b = math.sin(avg_theta)
                x0 = a * avg_rho
                y0 = b * avg_rho
                pt1 = (int(x0 + 5000*(-b)), int(y0 + 5000*(a)))
                pt2 = (int(x0 - 5000*(-b)), int(y0 - 5000*(a)))
                if i == 0:
                    hlines.append([pt1, pt2])
                else:
                    vlines.append([pt1, pt2])
                if save_img:
                    cv2.line(cdst2, pt1, pt2, (255*i,0,255*(1-i)), 3, cv2.LINE_AA)

    intersec_points = []
    i = 0
    for hline in hlines:
        for vline in vlines:
            point = find_intersection(hline, vline)
            intersec_points.append(point)
            if save_img:
                cv2.circle(cdst3, point, radius=10, color=(153, 255, 255), thickness=-1)
                out = cv2.putText(cdst3, str(i), point, cv2.FONT_HERSHEY_SIMPLEX, 2, (153, 255, 255), 2, cv2.LINE_AA)
                i += 1
    if save_img:
        cv2.imwrite('HW8-Files/Result2/' + output_name + '_lines.jpg', cdst)
        cv2.imwrite('HW8-Files/Result2/' + output_name + '_refined_lines.jpg', cdst2)
        cv2.imwrite('HW8-Files/Result2/' + output_name + '_points.jpg', cdst3)
    return intersec_points


# In[250]:


view_1 = preprocess_to_get_points('HW8-Files/Dataset1/Pic_1.jpg', save_img=True, output_name='pic_1')
```

```python


# In[3]:


def get_world_coord(grid_size, h, w):
    x = np.linspace(0, grid_size*(w-1), w)
    y = np.linspace(0, grid_size*(h-1), h)
    xv, yv = np.meshgrid(x, y)
    return np.concatenate([xv.reshape((-1, 1)), yv.reshape((-1,1))], axis=1)

world_coord = get_world_coord(10, 10, 8)


# In[212]:


def linear_least_squares(A):
    u, s ,vh = np.linalg.svd(np.matmul(np.transpose(A), A))
    return vh[-1]

def find_homography(domain_points, range_points):
    A = []

    for i in range(len(range_points)):
        x = domain_points[i][0]
        y = domain_points[i][1]
        x_prime = range_points[i][0]
        y_prime = range_points[i][1]
        A.append([0, 0, 0, -x, -y, -1, y_prime*x, y_prime*y, y_prime])
        A.append([x, y, 1, 0, 0, 0, -x_prime*x, -x_prime*y, -x_prime])

    H = linear_least_squares(np.array(A))
    return np.reshape(H, (3,3))

def preprocess_to_get_Hs(dataset_path, prefix='Pic_', save_img=False, canny_t1=400, canny_t2=300, hough=50):
    Hs = []
    all_intersec_points = []
    print(len(os.listdir(dataset_path)))
    for i in range(len(os.listdir(dataset_path))):
        img_path = os.path.join(dataset_path, prefix+str(i+1)+'.jpg')
        print(img_path)
        if img_path == 'HW8-Files/Dataset2/IMG_20.jpg':
            canny_t1 = 300
            canny_t2 = 500
        intersec_points = preprocess_to_get_points(img_path, save_img=save_img, output_name=prefix+str(i+1),
                                                   canny_t1=canny_t1, canny_t2=canny_t2, hough=hough)
        all_intersec_points.append(intersec_points)
        H = find_homography(world_coord, intersec_points)
        Hs.append(H)
    return Hs, all_intersec_points


# In[214]:


```

```python
Hs, all_intersec_points = preprocess_to_get_Hs('HW8-Files/Dataset1/')


# In[215]:


def find_omega(Hs):
    V = []
    for H in Hs:
        h11, h12, h13 = (H[0,0], H[1,0], H[2,0])
        h21, h22, h23 = (H[0,1], H[1,1], H[2,1])
        V.append([h11*h21, h11*h22 + h12*h21, h12*h22,                    h13*h21 + h11*h23, h13*h22 + h12*h23, h13*h23])
        V.append([h11**2 - h21**2, 2*h11*h12 - 2*h21*h22, h12**2 - h22**2, 2*h11*h13 - 2*h21*h23, 2*h12*h13 - 2*h22*h23, h13**2 - h23**2])

    b = linear_least_squares(np.array(V))
    omega = [[b[0], b[1], b[3]],
             [b[1], b[2], b[4]],
             [b[3], b[4], b[5]]]
    return np.array(omega)


# In[216]:


omega = find_omega(Hs)


# In[217]:


def find_K(omega):
    x0 = (omega[0,1] * omega[0,2] - omega[0,0] * omega[1,2]) / (omega[0,0] * omega[1,1] - omega[0,1]**2)
    lamda = omega[2,2] - (omega[0,2]**2 + x0 * (omega[0,1] * omega[0,2] - omega[0,0] * omega[1,2])) / omega[0,0]
    alpha_x = np.sqrt(lamda / omega[0,0])
    alpha_y = np.sqrt(lamda * omega[0,0] / (omega[0,0] * omega[1,1] - omega[0,1]**2))
    s = -omega[0,1] * alpha_x**2 * alpha_y / lamda
    y0 = s * x0 / alpha_y - omega[0,2] * alpha_x**2 / lamda
    K = [[alpha_x, s, x0],
         [0, alpha_y, y0],
         [0, 0, 1]]
    return np.array(K)


# In[218]:


K = find_K(omega)
print(K)


# In[219]:


def find_extrinsic_parameter(Hs, K):
```

```
    Rs = []
    ts = []
    for H in Hs:
        r1 = np.dot(np.linalg.inv(K), H[:,0])
        scaling = 1 / np.linalg.norm(r1)
        r1 = scaling * r1
        r2 = scaling * np.dot(np.linalg.inv(K), H[:,1])
        t = scaling * np.dot(np.linalg.inv(K), H[:,2])
        r3 = np.cross(r1, r2)
        R = np.column_stack((r1, r2, r3))
        u, s ,vh = np.linalg.svd(R)
        conditioned_R = np.matmul(u, vh)
        Rs.append(conditioned_R)
        ts.append(t)
    return Rs, ts


# In[220]:


Rs, ts = find_extrinsic_parameter(Hs, K)
print(Rs[7], ts[7])


# In[221]:


Rs, ts = find_extrinsic_parameter(Hs, K)
print(Rs[0], ts[0])


# In[222]:


def apply_homography(H, domain_points):
    homo_domain_points = np.concatenate((domain_points, np.ones((len(domain_points),
     1))), axis=1)
    range_points = np.matmul(H, homo_domain_points.T).T
    return range_points[:, :2] / np.expand_dims(range_points[:, 2], axis=1)

def camera_parameters(K, Rs, ts, radio_distortion=False):
    params = [K[0,0], K[0,1], K[0,2], K[1,1], K[1,2]]
    for R, t in zip(Rs, ts):
        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]])
        params = np.append(params, np.append(w, t))
    if radio_distortion:
        params = np.append(params, [0, 0])
    return params

def reconstruct_R(params):
    Rs = []
    ts = []
    K = np.array([[params[0], params[1], params[2]],
                  [0, params[3], params[4]],
                  [0, 0, 1]])
    for i in range(5, 6*((len(params)-5)//6), 6):
        w = params[i:i+3]
```

```
        t = params[i+3:i+6]
        phi = np.linalg.norm(w)
        w_matrix = np.array([[0, -w[2], w[1]],
                             [w[2], 0, -w[0]],
                             [-w[1], w[0], 0]])
        R = np.eye(3) + np.sin(phi)/phi * w_matrix + (1-np.cos(phi))/(phi**2) * np.matmul(w_matrix, w_matrix)
        Rs.append(R)
        ts.append(t)
    return K, Rs, ts

def remove_radio_distortion(projected_points, k1, k2, x0, y0):
    x = projected_points[:,0]
    y = projected_points[:,1]
    r2 = (x-x0)**2 + (y-y0)**2
    x_rad = x + (x-x0)*(k1*r2 + k2*r2**2)
    y_rad = y + (y-y0)*(k1*r2 + k2*r2**2)
    return np.column_stack((x_rad, y_rad))


# In[223]:


params = camera_parameters(K, Rs, ts)


# In[224]:


def cost_function(params, all_intersec_points, world_coord, radio_distortion=False):
    if radio_distortion:
        K, Rs, ts = reconstruct_R(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
        K, Rs, ts = reconstruct_R(params)

    all_projected_points = []
    for R, t in zip(Rs, ts):
        H = np.matmul(K, np.column_stack((R[:,0], R[:,1], t)))
        projected_points = apply_homography(H, world_coord)
        if radio_distortion:
            projected_points = remove_radio_distortion(projected_points, k1, k2, x0, y0)
        all_projected_points.append(projected_points)

    all_projected_points = np.concatenate(all_projected_points, axis=0)
    all_intersec_points = np.concatenate(all_intersec_points, axis=0)
    diff = all_intersec_points - all_projected_points
    return diff.flatten()


# In[225]:


res_lsq = least_squares(cost_function, params, method='lm', args=[all_intersec_points, world_coord])
```

```python


# In[226]:


K_refined, Rs_refined, ts_refined = reconstruct_R(res_lsq.x)


# In[229]:


print(K_refined)
print(Rs_refined[0], ts_refined[0])


# In[227]:


print(Rs_refined[7], ts_refined[7])


# In[17]:


def error_per_image(diff):
    dx = diff[:,0]
    dy = diff[:,1]
    distance = np.sqrt(dx**2 + dy**2)
    mean = np.mean(distance)
    var = np.var(distance)
    return mean, var


# In[268]:


def reprojection(world_coord, params, all_intersec_points, radio_distortion=False,
                 img_idx=-1, save_img=False, output_name=None):
    if radio_distortion:
        K, Rs, ts = reconstruct_R(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
        K, Rs, ts = reconstruct_R(params)

    all_projected_points = []
    K, Rs, ts = reconstruct_R(params)
    for R, t in zip(Rs, ts):
        H = np.matmul(K, np.column_stack((R[:,0], R[:,1], t)))
        projected_points = apply_homography(H, world_coord)
        if radio_distortion:
            projected_points = remove_radio_distortion(projected_points, k1, k2, x0, y0)
        all_projected_points.append(projected_points)

    if img_idx >= 0:
        diff = all_intersec_points[img_idx] - all_projected_points[img_idx]
```

```
        mean, var = error_per_image(diff)
        if save_img:
            img_path = 'HW8-Files/Result2/IMG_'+str(img_idx+1)+'_points.jpg'
            img = cv2.imread(img_path)

            for point in all_projected_points[img_idx]:
                cv2.circle(img, (int(point[0]), int(point[1])), radius=10, color=(51, 255, 51), thickness=-1)
            cv2.imwrite('HW8-Files/Result2/IMG_' + str(img_idx+1) +'_'+output_name+'_reproject.jpg', img)
    return mean, var


# In[269]:


mean, var = reprojection(world_coord, params, all_intersec_points, img_idx=0, save_img=True, output_name='init')
print(mean, var)


# In[270]:


mean, var = reprojection(world_coord, res_lsq.x, all_intersec_points, img_idx=0, save_img=True, output_name='refined')
print(mean, var)


# In[271]:


mean, var = reprojection(world_coord, params, all_intersec_points, img_idx=7, save_img=True, output_name='init')
print(mean, var)


# In[272]:


mean, var = reprojection(world_coord, res_lsq.x, all_intersec_points, img_idx=7, save_img=True, output_name='refined')
print(mean, var)


# In[273]:


params_rd = camera_parameters(K, Rs, ts, radio_distortion=True)
res_lsq_rd = least_squares(cost_function, params_rd, method='lm', args=[all_intersec_points, world_coord, True])


# In[274]:


print(res_lsq_rd.x[-2:])


```

```
# In[275]:


mean, var = reprojection(world_coord, res_lsq_rd.x, all_intersec_points,
    radio_distortion=True, img_idx=0, save_img=True, output_name='radio_dis')
print(mean, var)


# In[276]:


mean, var = reprojection(world_coord, res_lsq_rd.x, all_intersec_points,
    radio_distortion=True, img_idx=7, save_img=True, output_name='radio_dis')
print(mean, var)


# In[252]:


Hs, all_intersec_points = preprocess_to_get_Hs('HW8-Files/Dataset2/', prefix='IMG_',
     save_img=True,
                                                canny_t1=370, canny_t2=690, hough
    =100)


# In[253]:


omega = find_omega(Hs)
K = find_K(omega)
print(K)
Rs, ts = find_extrinsic_parameter(Hs, K)
print(Rs[0], ts[0])
print(Rs[7], ts[7])


# In[255]:


params = camera_parameters(K, Rs, ts)
res_lsq = least_squares(cost_function, params, method='lm', args=[
    all_intersec_points, world_coord])
K_refined, Rs_refined, ts_refined = reconstruct_R(res_lsq.x)


# In[256]:


print(K_refined)
print(Rs_refined[0], ts_refined[0])


# In[257]:


print(Rs_refined[7], ts_refined[7])


# In[260]:
```

```python


mean, var = reprojection(world_coord, params, all_intersec_points, img_idx=0, save_img=True, output_name='init')
print(mean, var)


# In[261]:


mean, var = reprojection(world_coord, res_lsq.x, all_intersec_points, img_idx=0, save_img=True, output_name='refined')
print(mean, var)


# In[262]:


mean, var = reprojection(world_coord, params, all_intersec_points, img_idx=7, save_img=True, output_name='init')
print(mean, var)


# In[263]:


mean, var = reprojection(world_coord, res_lsq.x, all_intersec_points, img_idx=7, save_img=True, output_name='refined')
print(mean, var)


# In[264]:


params_rd = camera_parameters(K, Rs, ts, radio_distortion=True)
res_lsq_rd = least_squares(cost_function, params_rd, method='lm', args=[all_intersec_points, world_coord, True])


# In[265]:


print(res_lsq_rd.x[-2:])


# In[266]:


mean, var = reprojection(world_coord, res_lsq_rd.x, all_intersec_points, radio_distortion=True, img_idx=0, save_img=True, output_name='radio_dis')
print(mean, var)


# In[267]:


mean, var = reprojection(world_coord, res_lsq_rd.x, all_intersec_points, radio_distortion=True, img_idx=7, save_img=True, output_name='radio_dis')
print(mean, var)
```