Homework 8 solution

Theory question

1. Zhang's algorithm for camera calibration involves identifying the intrinsic camera parameters from images of a known planar pattern (usually a checkerboard). The absolute conic Ω_∞ represents sets of points at infinity in the domain space, that it is invariant to Euclidian transformation and useful for obtaining intrinsic camera properties. When the plane π that contains the calibration pattern) intersects Ω_∞, it does so at these two circular points. These circular points provide constraints that help estimate the camera's intrinsic parameters without requiring knowledge of the camera's position or orientation.

2. In homogenous coordinate, the absolute conic is represented as

$$\Omega_\infty = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

When a camera observes Ω_∞, it is projected as the image of the absolute conic ω. The transformation from the domain coordinate to the image coordinate is governed by the intrinsic camera matric K:

$$K = \begin{matrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{matrix}$$

Where f_x and f_y are the focal lengths along the x and y axes, s is the skew coefficient and (c_x, c_y) is the principal point of the camera.

Projecting Ω_∞ through K gives the image of the conic ω on the image plane, resulting in,

$$\omega = (KK^T)^{-1}$$

(ii) ω is an imaginary conic in the image plane, meaning it does not correspond to any real points or pixel locations in the image. The Absolute Conic Ω_∞ in 3D space lies entirely in the "plane at infinity." When this conic is projected onto the image plane, it remains an imaginary entity because it represents directions rather than actual locations. Mathematically, the ω has complex conjugate points as its "points" of intersection in the projective plane, meaning it has no real solutions in the Euclidean sense.

Specifically, the matrix ω defines a quadratic form:

$$\boldsymbol{x}^T \omega \boldsymbol{x} = 0$$

Where x is a point in homogenous coordinates $(x, y, 1)^T$. For real values of x and y, this equation does not yield any real solutions because ω is an imaginary conic. Therefore, no real points (x, y) on the image plane satisfy this equation.

Canny edge detection, Hough line and intersection labelling

Canny edge detection was used to detect the edges in the checkerboard patterns before Hough Line Transformation, was used to identify and averages vertical and horizontal lines. The averaged lines were refined and used to determine intersections for corner extraction. The resulting visualization from this process can be seen in Figures 1, 3, 8 and 10. The homographies for all image views which map an image's points to the world coordinate system was also calculated.

Intrinsic and extrinsic camera parameters

Using the homography matrices derived from the previous section, the intrinsic parameters of a camera was obtained. First, a set of linear constraints from each homography was used to solve the image of the absolute conic. This matrix was used to extract intrinsic parameters such as focal lengths, skew, and principal point. The extrinsic camera parameters (rotation matrices and translation vectors), were derived using homographies and the intrinsic parameters. The rotation matrix was conditioned to make it orthonomal. These parameters are presented in later sections of this report. Using the intrinsic properties, the images were reprojected. The results of initial reprojection can be seen in Figures 2, 4, 9 and 11.

To get intrinsic parameters usisng Zhang's, method, the relationship between a 3D point (X,Y,Z)(*X*,*Y*,*Z*) in the world coordinates and its projection onto the image plane $x = (x, y, w)^T$ can be represented using a homography when $Z = 0$

$$\begin{matrix} x \\ \lambda\, y \\ 1 \end{matrix} \begin{matrix} \\ = H \\ \\ \end{matrix} \begin{matrix} X \\ Y \\ 1 \end{matrix}$$

H is the homography that maps points in image to the domain space. λ is a scaling factor

$$H = K[r_1, r_2, t]$$

K is the camera matric, r_1 and r_2 $are\ the\ rototational\ matrix\ and\ t$ is the translation vector

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

From multiple views of the calibration pattern, we obtain several homography matrices H_i. Using these homographies, we can estimate the intrinsic parameters by setting up the following constraints such that we have symmetric matrix B:

$$B = K^{-T}K^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$$

For each homography,

$$B = {h_{i1}}^T B h_{i2} = 0$$

$${h_{i1}}^T B h_{i1} = {h_{i2}}^T B h_{i2}$$

These constraints can be arranged as a system of linear equations in terms of the elements of $\boldsymbol{B}$, which are related to the intrinsic parameters.

After solving for B, we can now get the elements of the K matrix using the relationships below

$$c_x = \frac{B_{13}}{B_{11}}$$

$$c_y = \frac{B_{23}}{B_{22}}$$

$$f_x = \sqrt{\frac{B_{11}}{B_{33} - c_x^2}}$$

$$s = -\frac{B_{12}}{B_{11}}$$

Given a homography matrix $H = [h1\ h2\ h3]$ and knowing the intrinsic matrix K, we aim to compute the rotation matrix R and the translation vector t

First, a scale factor because the left side is homogeneous, and we must account for a scale factor ξ to make both sides equivalent

$$\xi = \frac{1}{||K^{-1}h_1||}$$

We compute the rotation vectors $r_1, r_2\ and\ r_3$

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

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

$$r_1 = r_1 \times r_1$$

$$t = \xi K^{-1} h_3$$

Levenberg-Marquardt optimization

It can be seen that the initial reprojection has a lot of errors visually. To refine the calibration parameters, Levenberg-Marquardt non-linear optimization was used better alignment between model and observed points. Figures 5, 6, 12 and 13 shows the visual results from this optimization. The new sets of extrinsic properties and camera matrix is also presented in the report.

Radial distortion correction

This radial distortion in projected image points was corrected by adjusting them based on their radial distance from the distortion center. This correction results in undistorted points, aligning the image projection with real-world geometry more accurately. Figures 5, 6, 12 and 13 shows the visual results after radial distortion removal and the radial distortion coefficients were presented in the report.

Reprojection error mean and variance

Reprojection analysis was performed to evaluate the accuracy of a camera model by projecting 3D world coordinates onto 2D image coordinates and measuring the error. The mean and variance of reprojection errors by comparing actual image points with the projected points are presented in Tables 1 and 2

Camera pose

The 3D representation of all views from the cameras are presented in Figures 7 and 14.

Custom dataset output

Image 4 output

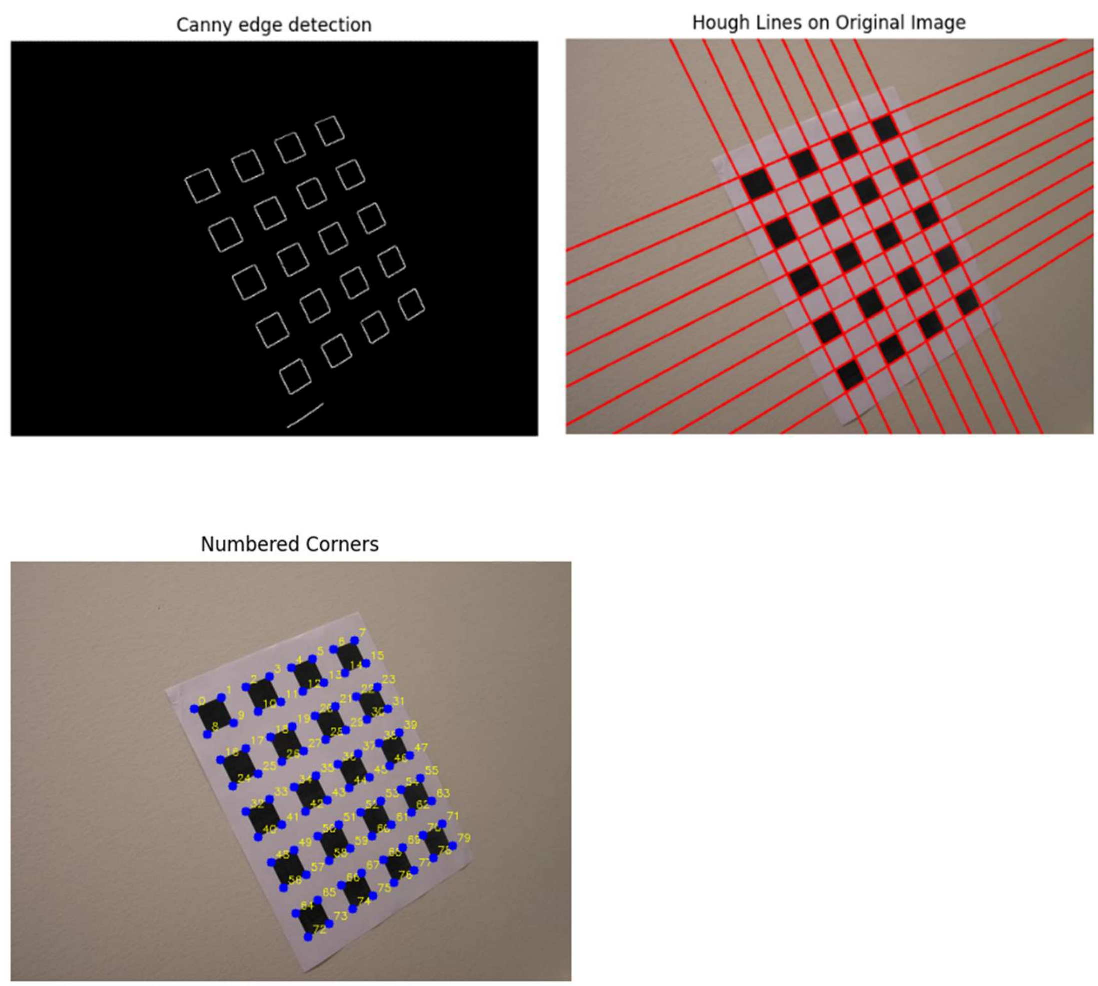


Figure 1: Result from canny edge detection, Hough lines and labeled corners for image 4

The intrinsic matrix K before LM is

$$K = \begin{bmatrix} 0.0202 & -0.0105 & 0.0368 \\ 0 & 0.0415 & 0.0427 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of image 4 before LM is

$$[R_4|t_4] = \left[\begin{array}{ccc|c} 0.863 & 0.568 & -0.00001 & 21.91 \\ -0.568 & 0.827 & 0.000015 & 6.609 \\ 0.000016 & -0.000006 & 1 & 0.0016 \end{array}\right]$$

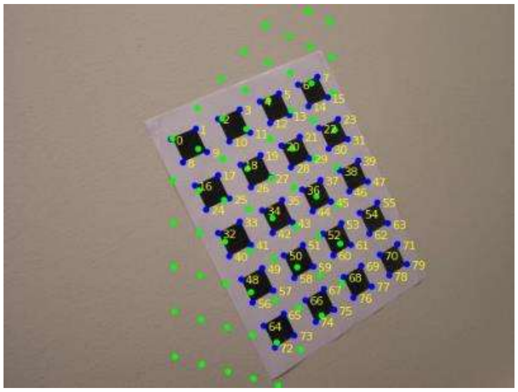
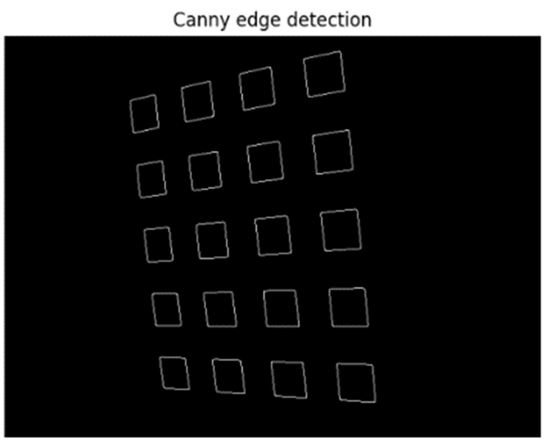
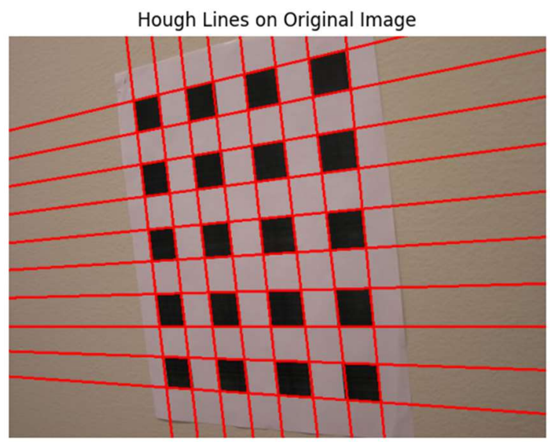


Figure 2: Initial reprojection for image 4

Image 19 Output





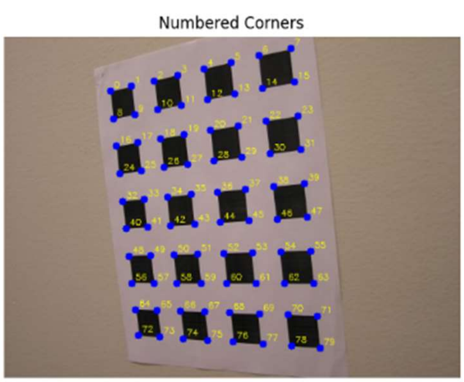


Figure 3: Result from canny edge detection, Hough lines and labeled corners for image 19

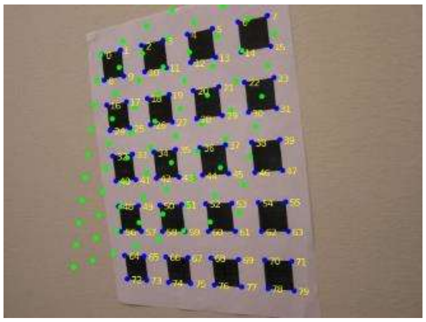


Figure 4: Initial reprojection for image 19

The rotation and translation parameters of image 19 before LM is

$$[R_{19}|t_{19}] = \left[\begin{array}{ccc|c} -0.895 & -0.446 & -0.000031 & -27.01 \\ 0.446 & -0.8949 & -0.000025 & -4.53 \\ 0.00004 & -0.000009 & 1 & -0.0028 \end{array}\right]$$

Refinement and radial distortion removal

$$K = \begin{bmatrix} 724.221 & 2.438 & 326.543 \\ 0 & 721.957 & 235.034 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of image 4 after LM is

$$[R_4|t_4] = \left[\begin{array}{ccc|c} 0.826 & 0.430 & -0.364 & -8.44 \\ -0.416 & 0.901 & 0.123 & -4.89 \\ 0.381 & 0.05 & 0.923 & 52.89 \end{array}\right]$$

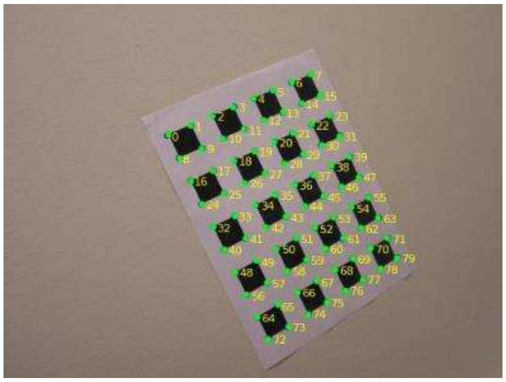
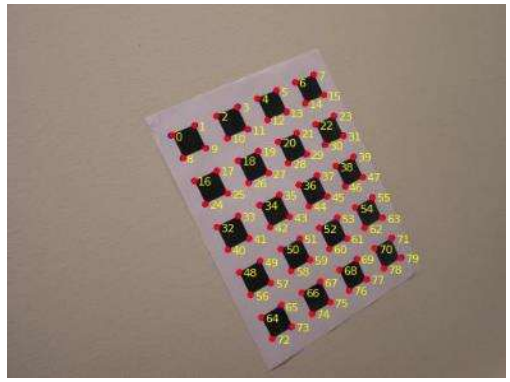
 

Figure 5: Reprojection after LM and Reprojection after radial distortion removal for image 4

The rotation and translation parameters of image 19 after LM is

$$[R_{19}|t_{19}] = \left[\begin{array}{ccc|c} -0.876 & -0.104 & 0.471 & 10.94 \\ 0.0724 & -0.994 & -0.085 & 9.75 \\ 0.477 & -0.04 & 0.878 & -44.38 \end{array}\right]$$

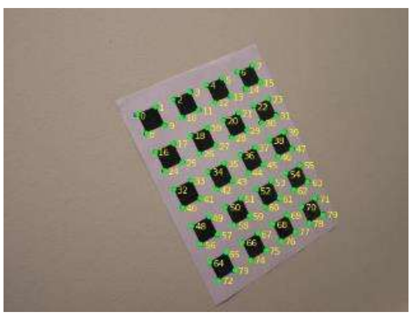
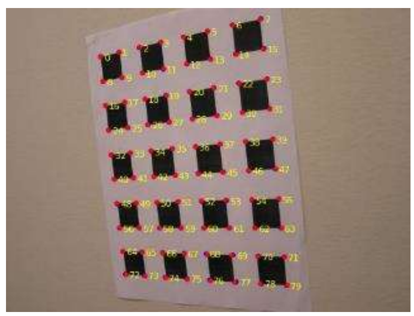
 

Figure 6: Reprojection after LM and Reprojection after radial distortion removal for image 19

Radial distortion parameters

$$k = [-1.06 \times 10^{-7}, \qquad -2.16 \times 10^{-12}]$$

Table 1. Comparison of mean and variance before LM, after LM and with radial distortion correction (custom dataset)

	Before LM	After LM	Radial distortion
Error mean (Image 4)	94.816	0.739	0.7301
Error variance (Image 4)	3076.916	0.138	0.16
Error mean (Image 19)	76.326	0.981	0.861
Error variance (Image 19)	1117.62	0.275	0.271

Camera pose

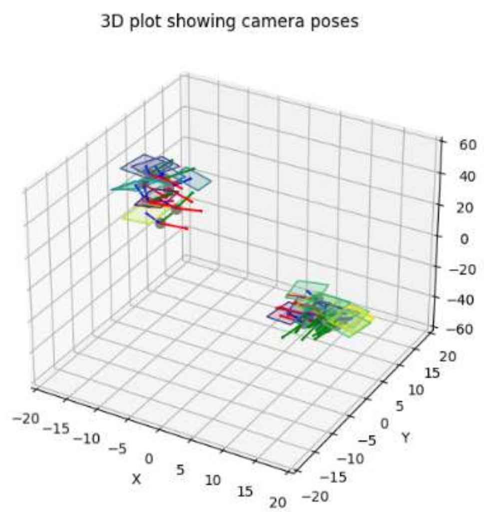
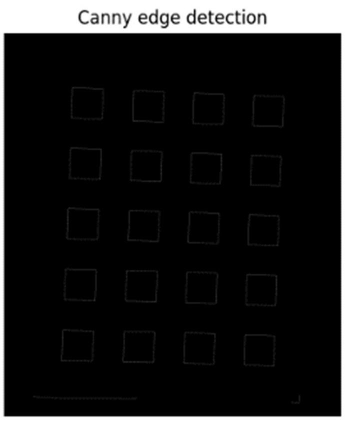
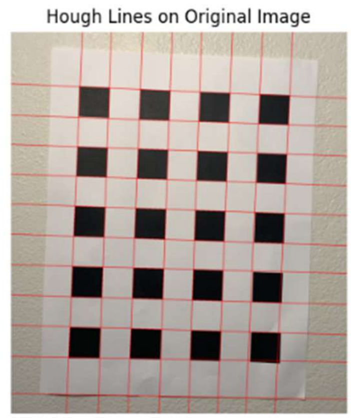


Figure 7: 3D view of all camera pose

My dataset





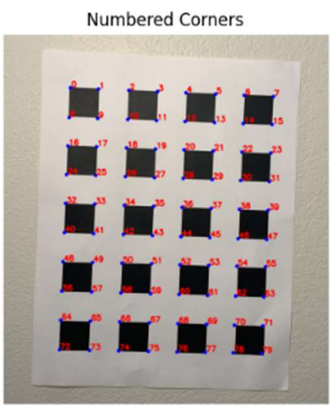


Figure 8: Result from canny edge detection, Hough lines and labeled corners for image 4

The intrinsic matrix K before LM is

$$K = \begin{bmatrix} 0.065 & -0.015 & 0.033 \\ 0 & 0.068 & 0.038 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of image 4 before LM is

$$[R_4|t_4] = \left[\begin{array}{ccc|c} -0.997 & -0.071 & -0.000001 & -6.34 \\ 0.0707 & -0.997 & 0.0000003 & -4.12 \\ -0.000001 & -0.000002 & 1 & -0.001 \end{array}\right]$$

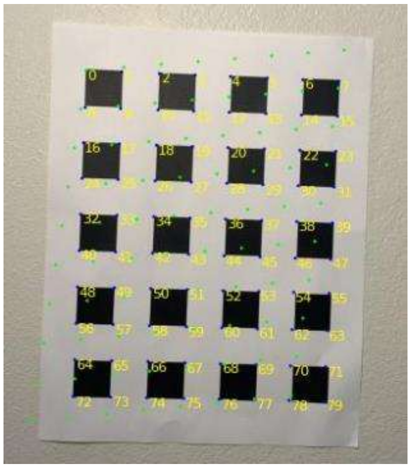


Figure 9: Initial reprojection for image 4

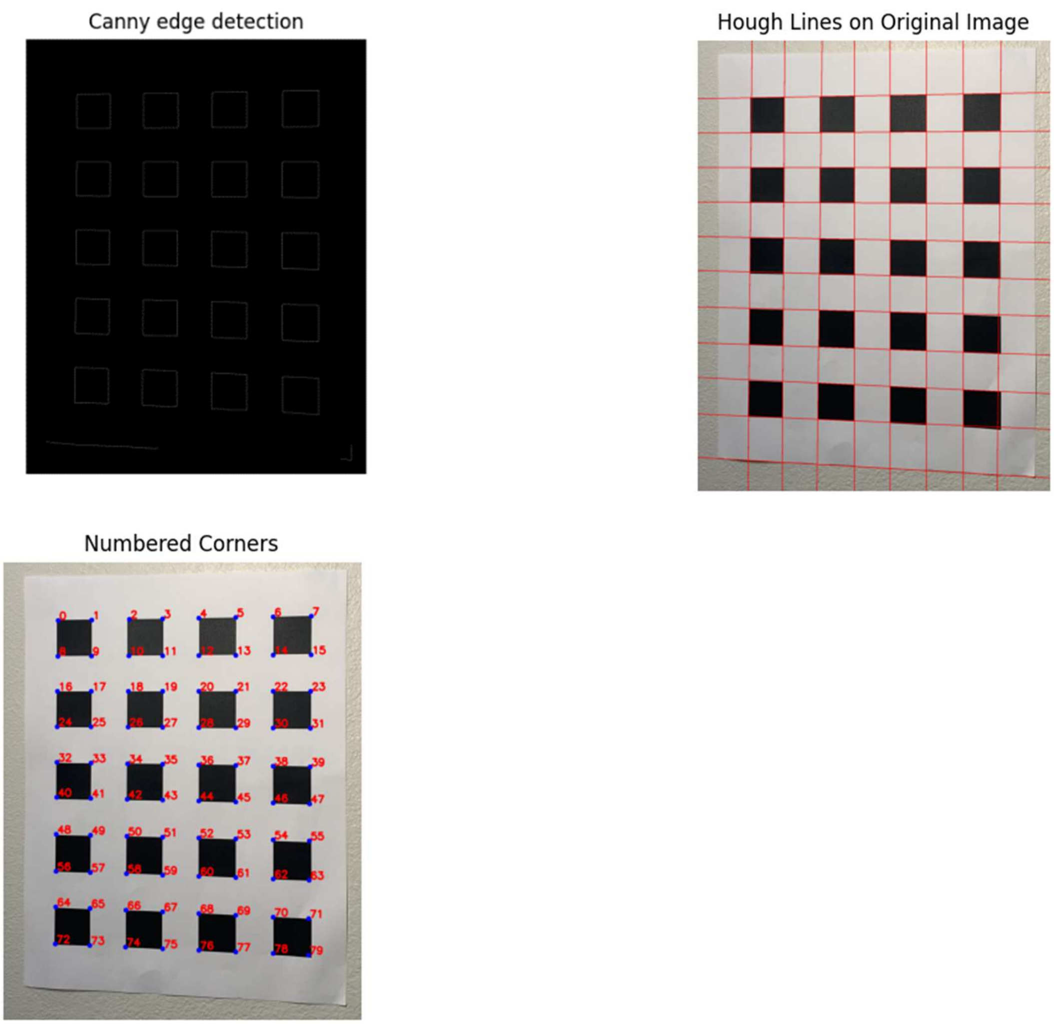


Figure 10: Result from canny edge detection, Hough lines and labeled corners for image 19

The rotation and translation parameters of image 19 before LM is

$$[R_{19}|t_{19}] = \left[\begin{array}{ccc|c} -0.992 & 1.223 & 0.000003 & 4.83 \\ -1.223 & 0.992 & 0.0000007 & 3.97 \\ -0.000003 & -0.0000004 & 1 & 0.001 \end{array}\right]$$

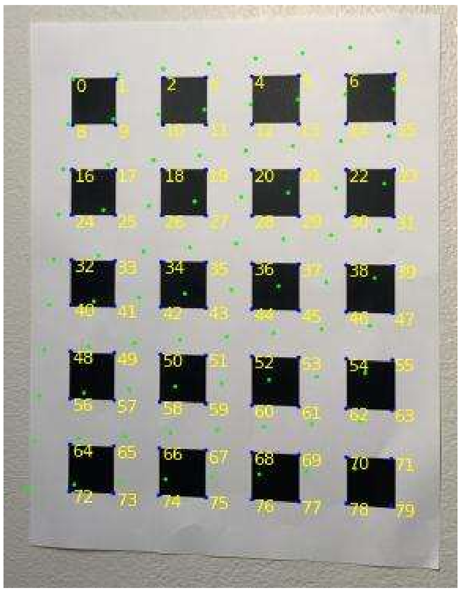


Figure 11: Initial reprojection for image 4

Refinement and radial distortion removal

$$K = \begin{bmatrix} 341.08 & -2.978 & 993.28 \\ 0 & 338.74 & 569.31 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation and translation parameters of image 4 after LM is

$$[R_4|t_4] = \left[\begin{array}{ccc|c} -0.998 & 0.038 & -0.531 & 9.25 \\ -0.039 & -1. & 0.011 & 4.022 \\ -0.052 & 0.013 & 0.999 & -50.57 \end{array}\right]$$

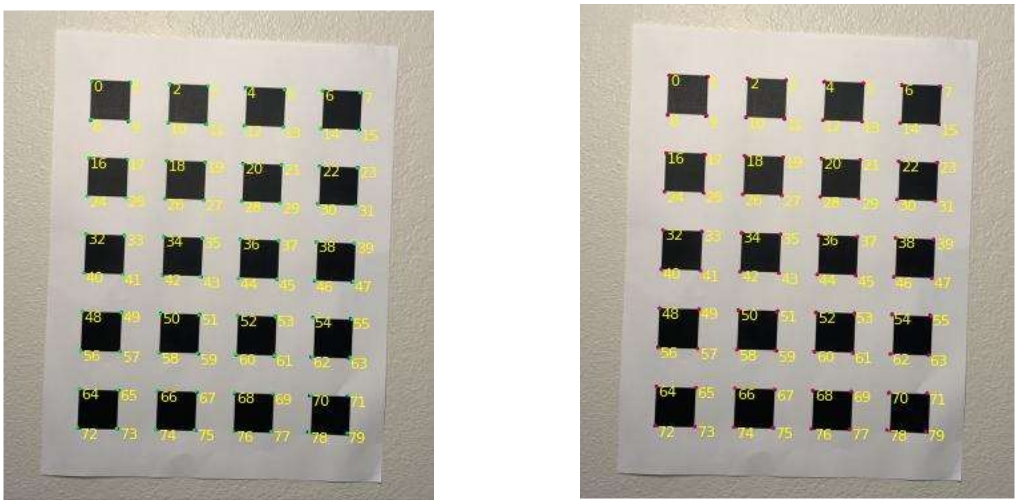


Figure 12: Reprojection after LM and Reprojection after radial distortion removal for image 4

The rotation and translation parameters of image 19 after LM is

$$[R_{19}|t_{19}] = \left[\begin{array}{ccc|c} 0.99 & -0.0035 & 0.141 & -12.44 \\ 0.00094 & 0.998 & -0.018 & -5.36 \\ -0.14 & -0.018 & 1 & 55.18 \end{array}\right]$$

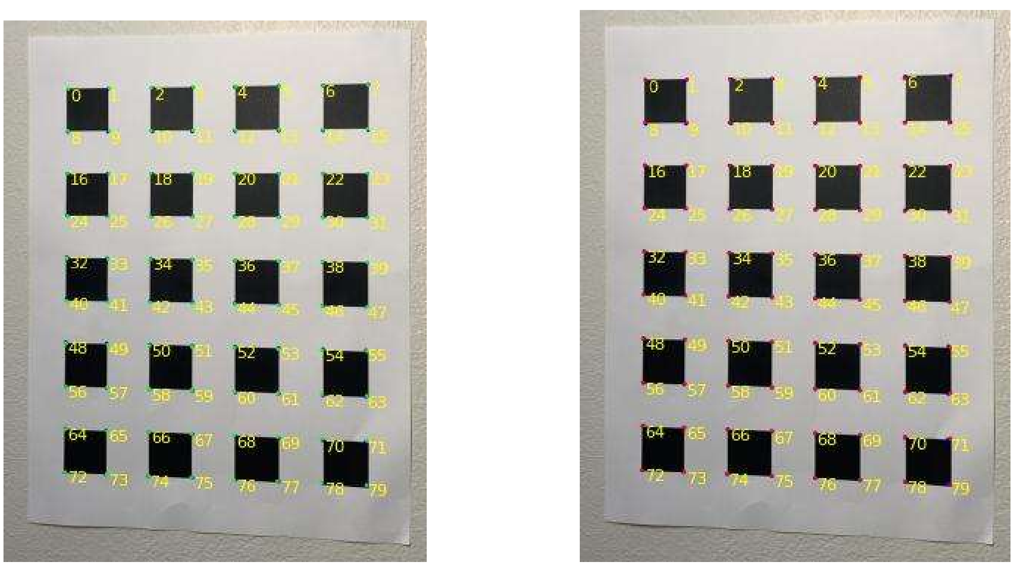


Figure 13: Reprojection after LM and Reprojection after radial distortion removal for image 19

Radial distortion parameters

$$k = [4.34 \times 10^{-9}, \qquad -6.58 \times 10^{-16}]$$

Table 2. Comparison of mean and variance before LM, after LM and with radial distortion correction (my dataset)

	Before LM	After LM	Radial distortion
Error mean (Image 4)	105.14	2.93	2.97
Error variance (Image 4)	2236.8	3.38	2.17
Error mean (Image 19)	92.19	3.45	3.35
Error variance (Image 19)	1531.57	3.099	2.62

Fixed image vs ground truth

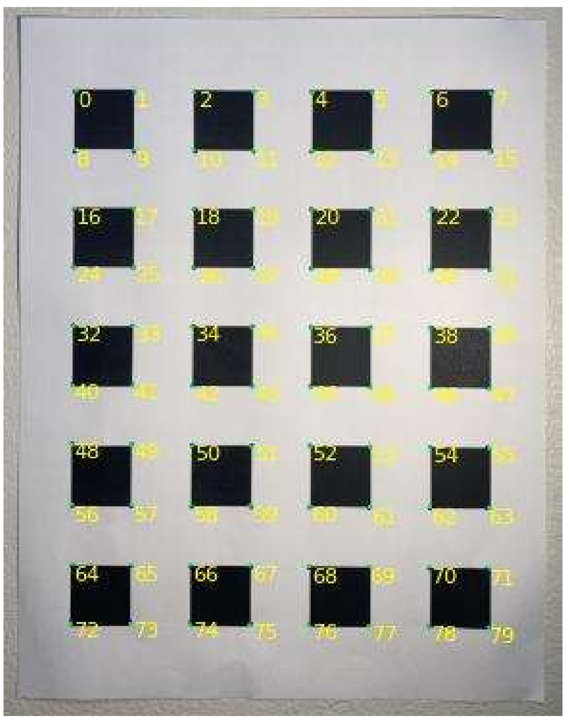


Figure 14: Fixed image vs ground truth

Camera pose

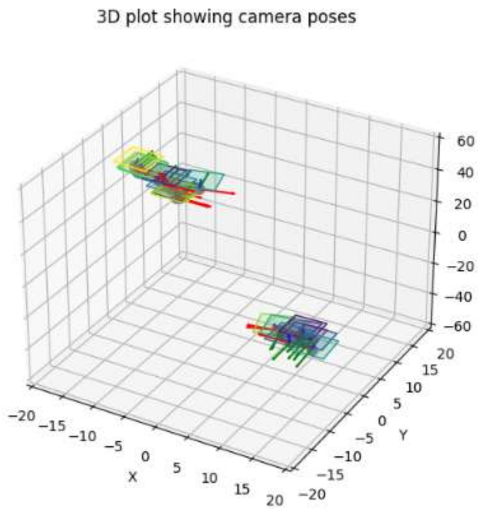


Figure 15: 3D view of all camera pose

Code

```
Run Cell | Run Below | Debug Cell
# %%
import cv2
import os
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import least_squares
import matplotlib.patches as patches
import re
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from matplotlib import cm

Run Cell | Run Above | Debug Cell
# %%
def group_lines_by_rho(lines, threshold=10):
    grouped_lines = []
    lines = sorted(lines, key=lambda x: abs(x[0]))  # Sort by |rho|

    current_group = [lines[0]]
    for line in lines[1:]:
        if abs(line[0] - current_group[-1][0]) < threshold:
            current_group.append(line)
        else:
            grouped_lines.append(current_group)
            current_group = [line]

    grouped_lines.append(current_group)
    return grouped_lines

def average_line(group):
    avg_rho = np.mean([line[0] for line in group])
    avg_theta = np.mean([line[1] for line in group])
    return avg_rho, avg_theta

def find_intersection(line1, line2):
    rho1, theta1 = line1
    rho2, theta2 = line2
    A = np.array([
        [np.cos(theta1), np.sin(theta1)],
        [np.cos(theta2), np.sin(theta2)]
    ])
    b = np.array([rho1, rho2])  # Make this a 1D array instead of a 2D array
    solution = np.linalg.solve(A, b)  # This will be a 1D array, not a 2D array
    x, y = solution[0], solution[1]
    return int(round(x)), int(round(y))

def order_coordinates(corners):
    # corners = sorted(corners, key=lambda point: point[0])
```

```python
    corners=sorted(corners, key=lambda point: (point[0], point[1]))

    # Group coordinates by their x-value into columns
    tolerance = 10  # Set a tolerance for grouping points within the same "column"
    columns = []
    current_column = [corners[0]]

    for i in range(1, len(corners)):
        if abs(corners[i][0] - current_column[-1][0]) < tolerance:
            # If x-value is close enough, it belongs to the same column
            current_column.append(corners[i])
        else:
            # New column
            columns.append(current_column)
            current_column = [corners[i]]

    # Add the last column
    if current_column:
        columns.append(current_column)

    # Sort each column by y-value to get top-to-bottom order within each column
    for col in columns:
        col.sort(key=lambda coord: coord[1])

    # Flatten the columns list to get the final left-to-right, top-to-bottom order
    corners = [coord for col in columns for coord in col]
    corners=[[x, y] for x, y in corners]
    return corners

def remove_close_points(points, threshold):

    new_points = []
    prev_point = None

    for point in points:
        if prev_point is None or np.linalg.norm(np.array(point) - np.array(prev_point)) > threshold:
            new_points.append(point)
            prev_point = point

    return new_points


def generate_world_coord(box_size=2.4, rows=5, cols=4):

    # To store the coordinates of all black box corners
```

```python
    box_corners = []

    # Generate coordinates with upper-left origin
    for row in range(rows):
        for col in range(cols):
            # Calculate the starting position of each black box (upper-left corner)
            x_start = col * 2 * box_size  # Every other column
            y_start = row * 2 * box_size  # Positive for each row downwards from the upper-left origin

            # Each black box has four corners, rounded to 1 decimal place
            corners = [
                (round(x_start, 1), round(y_start, 1)),  # Upper-left
                (round(x_start + box_size, 1), round(y_start, 1)),  # Upper-right
                (round(x_start + box_size, 1), round(y_start + box_size, 1)),  # Bottom-right
                (round(x_start, 1), round(y_start + box_size, 1)),  # Bottom-left
            ]

            box_corners.extend(corners)
    box_corners=[[x, y] for x, y in box_corners]
    return order_coordinates(box_corners)
world_coord=generate_world_coord()


# Function to solve for the last row of V in SVD
def linear_least_squares(A):
    _, _, vh = np.linalg.svd(A, full_matrices=False)
    return vh[-1]

# Function to compute homography matrix from domain and range points
def calculate_homography(domain_points, range_points):
    A = []

    # Build matrix A with corresponding points from domain and range
    for (x, y), (x_prime, y_prime) in zip(domain_points, range_points):
        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])

    # Perform linear least squares to solve for homography matrix
    H = linear_least_squares(np.array(A))

    # Reshape the result to a 3x3 matrix
    return H.reshape((3, 3))

def point_pre_process(points, threshold=10, max_points=80):
    new_points = []
    prev_point = None
```

```python

    for point in points:
        if prev_point is None or np.linalg.norm(np.array(point) - np.array(prev_point)) > threshold:
            new_points.append(point)
            prev_point = point

    if len(new_points) > max_points:
        return new_points[:max_points]

    return new_points


def group_lines_by_rho_and_theta(lines, rho_threshold=10, theta_threshold=np.pi / 36):
    """
    Groups lines based on proximity in both rho and theta.
    `rho_threshold` defines the maximum allowed difference in rho.
    `theta_threshold` defines the maximum allowed difference in theta.
    """
    grouped_lines = []
    lines = sorted(lines, key=lambda x: (x[0], x[1]))  # Sort by rho and theta

    current_group = [lines[0]]
    for line in lines[1:]:
        if (abs(line[0] - current_group[-1][0]) < rho_threshold and
            abs(line[1] - current_group[-1][1]) < theta_threshold):
            current_group.append(line)
        else:
            grouped_lines.append(current_group)
            current_group = [line]

    grouped_lines.append(current_group)
    return grouped_lines

def average_line(group):
    """
    Averages rho and theta for a group of lines.
    """
    avg_rho = np.mean([line[0] for line in group])
    avg_theta = np.mean([line[1] for line in group])
    return (avg_rho, avg_theta)




def get_homograpy_intersection(dir,threshold1=50, threshold2=150, hough_threshold=50):
    all_homographies=[]
    all_intersections=[]
```

```python
    all_edges=[]
    all_corner_images=[]
    all_hough_images=[]
    images=[dir+filename for filename in os.listdir(dir) if filename.endswith(".jpg")]
    images=sorted(images, key=lambda x: int(re.search(r'Pic_(\d+)', x).group(1)))



    for img in images:
        image = cv2.imread(img)  # Replace 'checkerboard.png' with your image path
        hough_image=image.copy()
        corner_image=image.copy()

        # Convert to grayscale
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # adjusted_image = cv2.convertScaleAbs(gray, alpha=1.5, beta=50)
        # Apply Gaussian blur
        # blurred = cv2.equalizeHist(gray)
        # blurred = cv2.GaussianBlur(blurred, (5, 5), 0)

        blurred = cv2.GaussianBlur(gray, (5, 5), 0)



        # Apply Canny edge detection
        edges = cv2.Canny(blurred, threshold1=threshold1, threshold2=threshold2)  # Adjust thresholds as needed


        lines = cv2.HoughLines(edges, rho=1, theta=np.pi/180, threshold=hough_threshold)

        vertical_lines = []
        horizontal_lines = []

        for line in lines:
            rho, theta = line[0]
            if abs(theta) < np.pi / 4 or abs(theta - np.pi) < np.pi / 4:
                vertical_lines.append((rho, theta))
            elif abs(theta - np.pi / 2) < np.pi / 4:
                horizontal_lines.append((rho, theta))

        grouped_vertical_lines = group_lines_by_rho_and_theta(vertical_lines, rho_threshold=70, theta_threshold=np.pi/36)
        grouped_horizontal_lines = group_lines_by_rho_and_theta(horizontal_lines, rho_threshold=70, theta_threshold=np.pi/36)

        average_vertical_lines = [average_line(group) for group in grouped_vertical_lines]
        average_horizontal_lines = [average_line(group) for group in grouped_horizontal_lines]

```

```python
        for rho, theta in average_vertical_lines + average_horizontal_lines:
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 + 3000 * (-b))
            y1 = int(y0 + 3000 * (a))
            x2 = int(x0 - 3000 * (-b))
            y2 = int(y0 - 3000 * (a))
            cv2.line(hough_image, (x1, y1), (x2, y2), (0, 0, 255), 2)

        corners = []
        for h_line in average_horizontal_lines:
            for v_line in average_vertical_lines:
                corner = find_intersection(h_line, v_line)
                corners.append(corner)


        corners= point_pre_process(corners)


        homography=calculate_homography(world_coord, corners)

        all_intersections.append(corners)
        all_homographies.append(homography)
        all_hough_images.append(hough_image)
        all_corner_images.append(corner_image)
        all_edges.append(edges)

    return all_homographies, all_intersections, all_edges, all_corner_images, all_hough_images




def plot_edges_hough_corner(corners, edges, corner_image, hough_image, img_no):
    # corners=intersection,

    for idx, point in enumerate(corners):
        # print(point)
        cv2.circle(corner_image, point, 10, (255, 0, 0), -1)  # Draw the point
        cv2.putText(corner_image, str(idx), (point[0] + 5, point[1] - 5),
                    cv2.FONT_HERSHEY_SIMPLEX, 1.4, (0, 0, 255), 5)  # Label with numbers starting from 0

    # # Display the results
    plt.imshow(edges, cmap='gray'), plt.title('Edges')
    plt.title('Canny edge detection')
    plt.axis('off')
    plt.savefig(f'{img_no} Canny edge detection')
    # plt.show()
    # plt.subplot(122), plt.imshow(cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB)), plt.title('Hough Line Segments')

    plt.imshow(cv2.cvtColor(hough_image, cv2.COLOR_BGR2RGB))
    plt.title('Hough Lines on Original Image')
    plt.axis('off')
    plt.savefig(f'{img_no} Hough Lines on Original Image')
    # plt.show()

    plt.imshow(cv2.cvtColor(corner_image, cv2.COLOR_BGR2RGB))
    plt.title('Numbered Corners')
    plt.axis('off')
    plt.savefig(f'{img_no} Numbered Corners')
    # plt.show()
all_homographies, all_intersections, edge_img, corners_img, hough_img  = get_homograpy_intersection('HW8-Files/Pattern/', threshold1=200, threshold2=500, hough_threshold=100)
plot_edges_hough_corner(all_intersections[3], edge_img[3], corners_img[3], hough_img[3], 4)
plot_edges_hough_corner(all_intersections[18], edge_img[18], corners_img[18], hough_img[18], 19)

Run Cell | Run Above | Debug Cell
# %%
# Image of the absolute conic
def find_omega(Hs):
    V = []
    for H in Hs:
        h11, h12, h13 = H[0, 0], H[0, 1], H[0, 2]
        h21, h22, h23 = H[1, 0], H[1, 1], H[1, 2]

        # Append two constraints derived from each homography
        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 - h21 * h22), h12 ** 2 - h22 ** 2, 2 * (h13 * h11 - h23 * h21), 2 * (h13 * h12 - h23 * h22), h13 ** 2 - h23 ** 2])

    # Solve for omega components
    b = linear_least_squares(np.array(V))

    # Create the symmetric matrix omega from b vector
    omega = np.array([
        [b[0], b[1], b[3]],
        [b[1], b[2], b[4]],
        [b[3], b[4], b[5]]
    ])

    return omega

absolute_conic=find_omega(all_homographies)
```

Run Cell | Run Above | Debug Cell

```python
# %%
def find_intrinsic(omega):
    # Extract terms from omega matrix
    x0 = (omega[0, 1] * omega[0, 2] - omega[0, 0] * omega[1, 2]) / (omega[0, 0] * omega[1, 1] - omega[0, 1] ** 2)
    # Calculate lambda as a scaling factor
    lambda_ = omega[2, 2] - ((omega[0, 2]**2 + x0 * (omega[0, 1] * omega[0, 2] - omega[0, 0] * omega[1, 2])) / omega[0, 0])
    # Calculate alpha values as focal lengths
    alpha_x = np.sqrt(lambda_ / omega[0, 0])
    alpha_y = np.sqrt(lambda_ * omega[0, 0] / (omega[0, 0] * omega[1, 1] - omega[0, 1] ** 2))

    # Compute skew and y0
    s = -omega[0, 1] * alpha_x**2 * alpha_y / lambda_
    y0 = s * x0 / alpha_y - omega[0, 2] * alpha_x**2 / lambda_

    # Construct intrinsic matrix K
    K = np.array([
        [alpha_x, s, x0],
        [0, alpha_y, y0],
        [0, 0, 1]
    ])

    return K
intrinsic_params=find_intrinsic(absolute_conic)

print('Intrisic param before LM',intrinsic_params)
```

Run Cell | Run Above | Debug Cell

```python
# %%
def find_extrinsic(Hs, K):

    Rs = []
    ts = []

    K_inv = np.linalg.inv(K)

    for H in Hs:
        # Extract the first two columns of the homography matrix and compute r1 and r2
        r1 = np.dot(K_inv, H[:, 0])
        r2 = np.dot(K_inv, H[:, 1])

        # Scaling factor to normalize r1
        scaling = 1.0 / np.linalg.norm(r1)
        r1 *= scaling
        r2 *= scaling
```

```python
        # Calculate the translation vector
        t = scaling * np.dot(K_inv, H[:, 2])

        # Compute r3 as the cross product of r1 and r2 to ensure orthogonality
        r3 = np.cross(r1, r2)

        # Stack r1, r2, and r3 to form the initial rotation matrix
        R = np.column_stack((r1, r2, r3))

        # Orthogonalize R using SVD to enforce a valid rotation matrix
        u, _, vh = np.linalg.svd(R)
        conditioned_R = np.dot(u, vh)

        # Append the conditioned rotation matrix and translation vector to the lists
        Rs.append(conditioned_R)
        ts.append(t)

    return Rs, ts

rotation_params, translation_params = find_extrinsic(all_homographies, intrinsic_params)
print('-----------------------------------------------------------------------')
print('Rotation 4 before LM',rotation_params[3], 'translation', translation_params[3])

print('Rotation 19 before LM',rotation_params[18], 'translation', translation_params[18])
# print('-----------------------------------------------------------------------')

Run Cell | Run Above | Debug Cell
# %%
# Extract camera parameters including rotation and translation
def camera_parameters(K, Rs, ts, radio_distortion=False):
    # Initialize parameter array with intrinsic parameters from K
    params = [K[0, 0], K[0, 1], K[0, 2], K[1, 1], K[1, 2]]

    for R, t in zip(Rs, ts):
        # Calculate rotation angle
        phi = np.arccos((np.trace(R) - 1) / 2)

        # Calculate Rodrigues vector w (rotation vector)
        if np.sin(phi) != 0:
            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]])
        else:
            w = np.array([0, 0, 0])  # Edge case when phi = 0 (no rotation)

        # Append rotation and translation to parameters
        params.extend(w)
        params.extend(t)
```

```python
    # Add radial distortion parameters if required
    if radio_distortion:
        params.extend([0, 0])

    return np.array(params)

params=camera_parameters(intrinsic_params, rotation_params, translation_params)

Run Cell | Run Above | Debug Cell
# %%

def apply_homography(H, domain_points):
    # Add homogeneous coordinate (1) to each point
    domain_points=np.array(domain_points)
    num_points = domain_points.shape[0]
    homo_domain_points = np.hstack((domain_points, np.ones((num_points, 1))))

    # Apply homography transformation
    range_points = (H @ homo_domain_points.T).T

    # Normalize by the third (homogeneous) coordinate
    range_points /= range_points[:, 2, np.newaxis]
    return range_points[:, :2]  # Return only x and y coordinates

def reconstruct_R(params):
    # Intrinsic matrix K
    K = np.array([[params[0], params[1], params[2]],
                  [0, params[3], params[4]],
                  [0, 0, 1]])

    Rs = []
    ts = []

    # Iterating through the parameters to calculate rotation and translation
    for i in range(5, 6 * ((len(params) - 5) // 6), 6):
        w = np.array(params[i:i+3])
        t = np.array(params[i+3:i+6])

        # Calculating rotation matrix using Rodrigues' formula
        phi = np.linalg.norm(w)
        if phi == 0:
            R = np.eye(3)  # No rotation if phi is zero
        else:
            w_matrix = np.array([[0, -w[2], w[1]],
                                 [w[2], 0, -w[0]],
                                 [-w[1], w[0], 0]])
            R = (np.eye(3) +
```

```python
                    (np.sin(phi) / phi) * w_matrix +
                    ((1 - np.cos(phi)) / (phi ** 2)) * np.dot(w_matrix, w_matrix))

        # Append rotation and translation matrices to the lists
        Rs.append(R)
        ts.append(t)

    return K, Rs, ts



# Remove radial distortion from projected points
def remove_radio_distortion(projected_points, k1, k2, x0, y0):
    x, y = projected_points[:, 0], projected_points[:, 1]
    # Calculate radial distance squared
    r2 = (x - x0) ** 2 + (y - y0) ** 2
    # Apply distortion correction
    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))

def cost_function(params, all_intersec_points, world_coord, radio_distortion=False):
    if radio_distortion:
        # print(params[:-10].shape)
        K, Rs, ts = reconstruct_R(params[:-2])
        k1, k2 = params[-2], params[-1]
        x0, y0 = params[2], params[4]
    else:
        K, Rs, ts = reconstruct_R(params)

    all_projected_points = []

    for R, t in zip(Rs, ts):
        H = np.dot(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)

    # print(all_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
```

```python
            projected_points = remove_radio_distortion(projected_points, k1, k2, x0, y0)


        all_projected_points.append(projected_points)

    # print(all_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()


# Run least squares optimization
res_lsq = least_squares(
    cost_function,
    params,
    method='lm',  # Levenberg-Marquardt method
    args=[all_intersections, world_coord]
)

Run Cell | Run Above | Debug Cell
# %%
refined_K, refined_R, refined_t=reconstruct_R(res_lsq.x)

print('After refinement camera matrix',refined_K,'Rotation 4 before LM',refined_R[3], 'translation', refined_t[3])
print('After refinement Rotation 19 before LM',refined_R[18], 'translation', refined_t[18])
# print('----------------------------------------------------------------------')

Run Cell | Run Above | Debug Cell
# %%
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

def reprojection(world_coord, params, all_intersec_points, rad_distortion=False, img_idx=0, save_img=False, output_name=None):
    if rad_distortion:
        K, Rs, ts = reconstruct_R(params[:-2])
        k1 = params[-2]
        k2 = params[-1]
        x0 = params[2]
        y0 = params[4]
    else:
```

```python
    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 rad_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/Pattern/Pic_' + str(img_idx+1) + '.jpg'
            img = plt.imread(img_path)

            # Set up the plot with the image
            fig, ax = plt.subplots()
            ax.imshow(img)

            # Draw circles at each projected point
            color='lime'
            if output_name =='radio_distortion':
                color='red'

            # Plot original target view points in blue
            for idx, point in enumerate(all_intersec_points[img_idx]):
                circle = patches.Circle((point[0], point[1]), radius=3, color='blue', fill=True)

                ax.add_patch(circle)
                ax.text(point[0] + 5, point[1] - 5, str(idx), fontsize=8, color='yellow',
                    ha='left', va='top')

            for point in all_projected_points[img_idx]:
                circle = patches.Circle((point[0], point[1]), radius=3, color=color, fill=True)
                ax.add_patch(circle)

            # Remove axis ticks for a cleaner image
            ax.axis('off')

            # Save the image
            output_path = 'Pic_' + str(img_idx+1) + '_' + output_name + '_reproject.jpg'
```

```python
            output_path = 'Pic_' + str(img_idx+1) + '_' + output_name + '_reproject.jpg'
            plt.savefig(output_path, bbox_inches='tight', pad_inches=0)
            plt.close(fig)

    return mean, var

Run Cell | Run Above | Debug Cell
# %%
mean, var=reprojection(world_coord, params, all_intersections, img_idx=3, save_img=True, output_name='init')
print('Img 4, init mean and variance',mean, var)

mean, var=reprojection(world_coord, res_lsq.x, all_intersections, img_idx=3, save_img=True, output_name='refined')
print('Img 4, refined mean and variance',mean, var)

mean, var=reprojection(world_coord, params, all_intersections, img_idx=18, save_img=True, output_name='init')
print('Img 19, init mean and variance',mean, var)

mean, var=reprojection(world_coord, res_lsq.x, all_intersections, img_idx=18, save_img=True, output_name='refined')
print('Img 19, refined mean and variance',mean, var)
print('-----------------------------------------------------------------------')


Run Cell | Run Above | Debug Cell
# %%
params_radio_distortion=camera_parameters(intrinsic_params, rotation_params, translation_params, radio_distortion=True)


# Run least squares optimization
res_lsq_radio_distortion = least_squares(
    cost_function,
    params_radio_distortion,
    method='lm',  # Levenberg-Marquardt method
    args=[all_intersections, world_coord, True]
)

print('Radial distortion is',res_lsq_radio_distortion.x[-2:])

Run Cell | Run Above | Debug Cell
# %%
mean, var=reprojection(world_coord, res_lsq_radio_distortion.x, all_intersections, rad_distortion=True, img_idx=3, save_img=True, output_name='radio_distortion')
print('Img 4, radio distortion mean and variance',mean, var)

mean, var=reprojection(world_coord, res_lsq_radio_distortion.x, all_intersections, rad_distortion=True, img_idx=18, save_img=True, output_name='radio_distortion')
print('Img 19, radio distortion mean and variance',mean, var)
print('-----------------------------------------------------------------------')
```

```python
# Function to plot camera pose
def plot_camera(ax, R, t, scale=5, color='gray',plane_color='cyan'):
    # Define camera frame axes in the camera's local coordinates
    X_cam = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]).T * scale

    # Rotate and translate camera frame axes to world coordinates
    X_world = R @ X_cam + t.reshape(-1, 1)

    # Plot the camera center
    ax.scatter(t[0], t[1], t[2], color=color, s=50)

    # Plot the axes
    ax.quiver(t[0], t[1], t[2], X_world[0, 0]-t[0], X_world[1, 0]-t[1], X_world[2, 0]-t[2], color='r', arrow_length_ratio=0.1)
    ax.quiver(t[0], t[1], t[2], X_world[0, 1]-t[0], X_world[1, 1]-t[1], X_world[2, 1]-t[2], color='g', arrow_length_ratio=0.1)
    ax.quiver(t[0], t[1], t[2], X_world[0, 2]-t[0], X_world[1, 2]-t[1], X_world[2, 2]-t[2], color='b', arrow_length_ratio=0.1)

    # Draw the camera plane
    plane_points = np.array([[0.5, 0.5, 1], [0.5, -0.5, 1], [-0.5, -0.5, 1], [-0.5, 0.5, 1]]).T * scale
    plane_world = R @ plane_points + t.reshape(-1, 1)

    verts = [list(zip(plane_world[0, :], plane_world[1, :], plane_world[2, :]))]
    ax.add_collection3d(Poly3DCollection(verts, color=plane_color, alpha=0.2))

# Set up the figure and 3D axis
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

K, Rs, ts = reconstruct_R(res_lsq.x)
# print(Rs[0])
# print('ts',ts[0])

# print('Rs',Rs[3])
# print('ts',ts[3])

camera_poses=[(Rs[0], ts[0]), (Rs[3], ts[3])]
# Plot each camera
cam=[]
for r,t in zip(Rs, ts):
    new=(r,t)
    cam.append(new)

colormap = cm.get_cmap('viridis', len(cam))


for i, (R, t) in enumerate(cam):
    color = colormap(i)
    plot_camera(ax, R, t, plane_color=color)
```

```
# Set plot limits and labels
ax.set_xlim(-20, 20)
ax.set_ylim(-20, 20)
ax.set_zlim(-60, 60)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.set_title("3D plot showing camera poses")

plt.show()
```