ECE 661 Homework 7

Michael Goldberg

October 30 2024

1 Theoretical Question

1.1 Question 1

The reason this is a fundamental observation is because if we can sample the absolute conic Ω_∞, we can construct ω, which is the image of the absolute conic. Because $\omega = K^{-T}K^{-1}$, where K is the intrinsic calibration matrix of the camera. Therefore, if ω can be found through sufficient number of sampling, then by proxy K can be determined and our camera's intrinsic parameters can be estimated.

1.2 Question 2

The algebraic relationship between Ω and ω can be realized through understanding the relationship between world3D points and their corresponding image within the camera. The projection of a 3D point (X, Y, Z) onto the image plane is:

$$x = K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{1}$$

This can be compared with the general transformation of a conic C through a homography H with:

$$C' = H^{-T}CH^{-1} \tag{2}$$

Here, K is our homography, as it projects a world point to a camera point. Therefore,

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

2 Programming Tasks

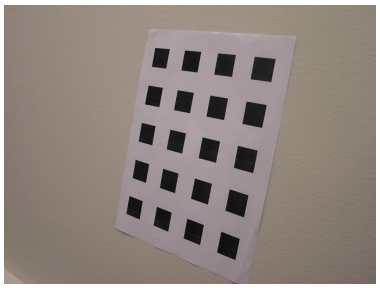
The following task was to use two datasets of a calibration board and calibrate the intrinsic and extrinsic parameters of the cameras that took the images. This was split into a series of steps:

1. Corner Detection

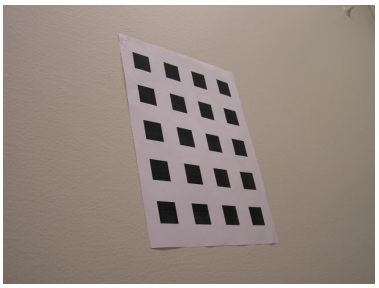
2. Zhang's Algorithm

3. Levenberg-Marquardt Nonlinear Optimization

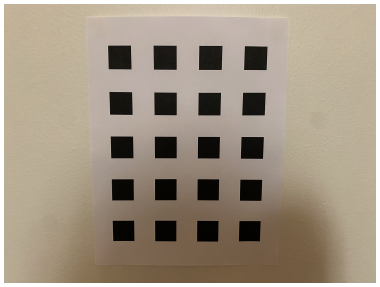
The following shows some example images of the calibration board from each dataset:



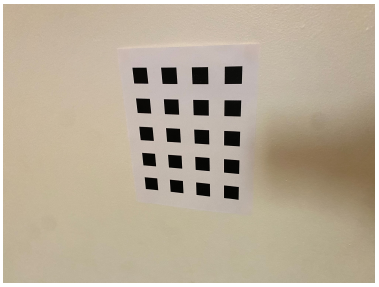
(a) Dataset 1 Image 1



(b) Dataset 1 Image 2



(c) Dataset 2 Image 1



(d) Dataset 2 Image 2

I will now go over how I addressed each task.

2.1 Corner Detection

This process was further broken down into a series of steps. The image was first converted to grayscale and Canny edge detection was performed. Then, Hough transform was performed on the Canny results to find lines that created the corners of the black squares. This process resulted in a lot of false positives, so a filtering process needed to be employed to ensure no duplicate lines or false positives were produced. Then, the intersections of each of these lines could be determined to find the corners. The lines also were sorted to ensure that corners were identified in the same order for any given image.

The most in-depth process was the line filtering and sorting. This was achieved in the following manner: First, the lines

The following shows the results of the Canny edge detection followed by the Hough Transform.

My best parameters for Canny edge detection were:

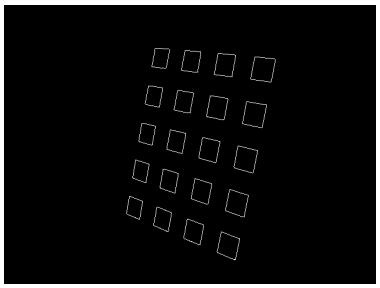
- Threshold1 = 200

- Threshold2 = 400

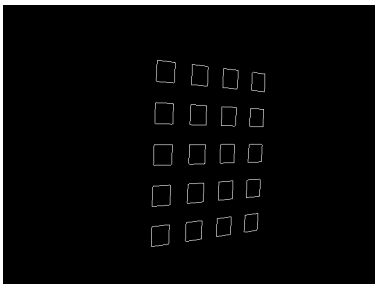
My best parameters for Hough transform were:

- Rho Threshold = 1
- Theta Threshold = $\pi/180$
- Hough Threshold = 50

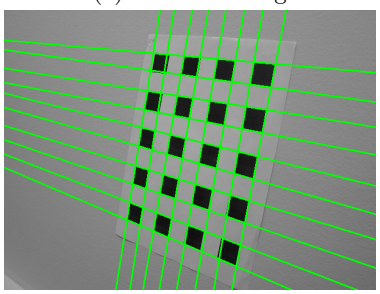
Dataset 1:



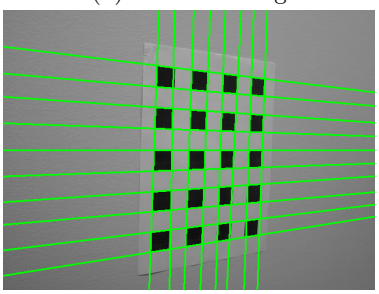
(a) Dataset 1 Edges 1



(b) Dataset 1 Edges 2

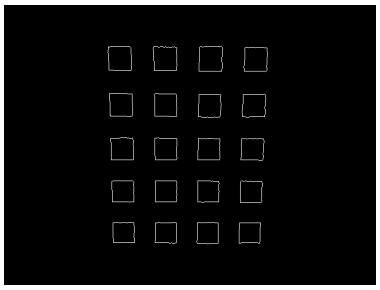


(c) Dataset 1 Hough Filtered 1

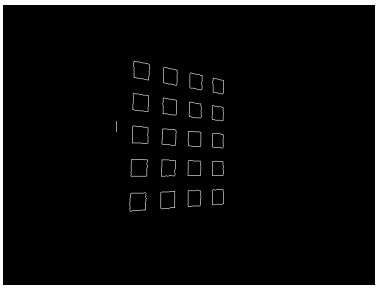


(d) Dataset 1 Hough Filtered 2

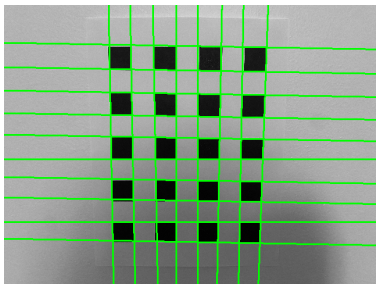
Dataset 2:



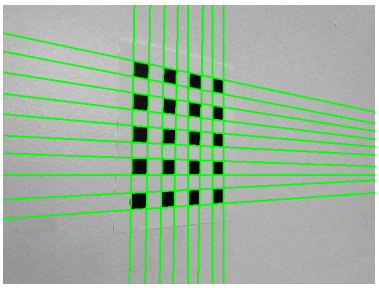
(a) Dataset 2 Edges 1



(b) Dataset 2 Edges 2



(c) Dataset 2 Hough Filtered 1



(d) Dataset 2 Hough Filtered 2

The Hough transform returns a list of lines given in the form (ρ, θ), where ρ is the length of the perpendicular distance between the origin and the line that passes through a point of interest, and θ is the angle in radians of this perpendicular segment from the horizontal. The origin that this is measured from was observed to be from the image's origin, which is the top-left corner of the image. Therefore, to filter the lines, the lines were split into two groups - those that fell within a certain θ buffer $= 50°$ around pure vertical (e.g. $abs(\theta) < deg2rad(50) or abs(\theta - \pi) < deg2rad(50)$ and those that fell within the same buffer around pure horizontal (e.g. $abs(\theta - \pi/2) < deg2rad(50) or abs(\theta - 3\pi/2) < deg2rad(50)$. Across these two groups, I could then check if any other given line within the group fell within a small buffer around the same value of ρ and θ. If these two values are too similar, then that means a slightly different line describing the same edge was picked up in the Hough transform and needs to be discarded. The buffer I used for this checking was 15 for ρ and $5°$ for θ. Now that we had discarded the duplicates, we were left with a single unique line for each edge.

For any pair of lines l_1 and l_2 in homogeneous coordinates, their intersection is given by

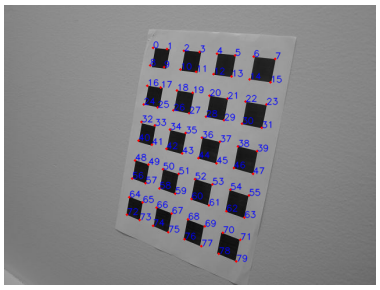
$$x_{intersection} = l_1 \times l_2 \tag{4}$$

This was done for all filtered lines, and any ideal points or points outside of the image bounds were discarded. This allowed me to find the corner points of the black squares on the calibration board. Now, all that was left was labelling the lines in a consistent manner, which could be achieved by sorting the lines in each group by increasing ρ. This resulted in line 0 of the horizontal group

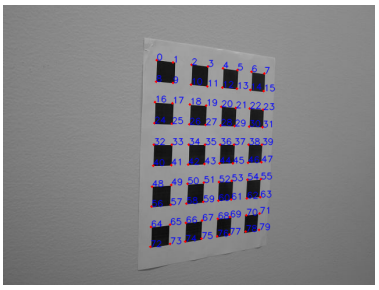
being the topmost line, and line 0 of the vertical group being the leftmost line. With a consistent sorting, the corners could be trivially labeled by which lines made up the intersection.

Below is the results of the corner detection for both datasets:

Dataset 1:

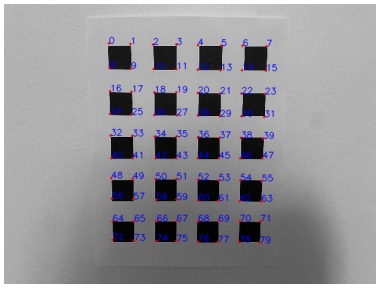


(a) Dataset 1 Detected Corners 1

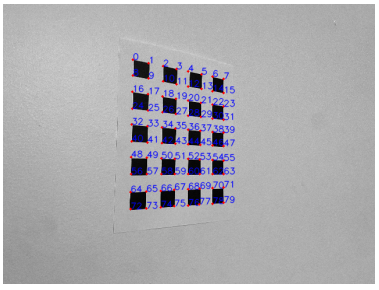


(b) Dataset 1 Detected Corners 2

Dataset 2:



(a) Dataset 2 Detected Corners 1



(b) Dataset 2 Detected Corners 2

Now, Zhang's calibration algorithm can be implemented.

2.2 Zhang's Algorithm Implementation

There are two main steps to Zhang's Camera Calibration Algorithm:

1. Intrinsic Calibration
2. Extrinsic Calibration

We begin with intrinsic calibration. We assume that the calibration pattern is on the world XY plane, which allows us to construct the following homography relationship:

$$x_{cam} = K[R|t] \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

where we can express $K[R|t]$ as H.

The homography can be estimated by understanding where the detected corners lie in the world coordinate frame and using SVD with point-to-point correspondencies. I estimated the size of the black squares to be 25mm along each axis, as well as being separated by the same distance. I also assumed the top left corner (corner 0 labeled above) to be at world coordinates X = 0 and Y = 0. With these assumptions, I had the coordinates of all of the corners both in the world coordinate frame as well as the camera image. The following relationship could then be established:

$$Ah = x \tag{6}$$

where

$$A = \begin{bmatrix} -X & -Y & -1 & 0 & 0 & 0 & xX & xY & x \\ 0 & 0 & 0 & -X & -Y & -1 & yX & yY & y \end{bmatrix} \tag{7}$$

, where X and Y are the world coordinates of a point and x and y are the image points. This matrix A can be populated with all 80 corner correspondencies, leading to 160 rows and 9 columns. This can be solved using SVD to find H.

Once H is established, we want to estimate the intrinsic parameters using the relationship that $\omega = K^{-T}K^{-1}$. This can be done using the homography we found for the image as well as the sampled circular points where we know the following to be true:

$$V_{01}^T b = 0 \tag{8}$$

and

$$(V_{00} - V11)^T b = 0 \tag{9}$$

, where

$$V_{i,j} = \begin{bmatrix} h_{i,0}h_{j,0} \\ h_{i,0}h_{j,1} + h_{i,1}h_{j,0} \\ h_{i,1}h_{j,1} \\ h_{i,2}h_{j,0} + h_{i,0}h_{j,2} \\ h_{i,2}h_{j,1} + h_{i,1}h_{j,2} \\ h_{i,2}h_{j,2} \end{bmatrix} \tag{10}$$

and

$$b = \begin{bmatrix} \omega_{00} \\ \omega_{01} \\ \omega_{11} \\ \omega_{02} \\ \omega_{12} \\ \omega_{22} \end{bmatrix} \tag{11}$$

By calculating $V_{01}^T b$ and $(V_{00} - V_{11})^T b$ for each calibration image, we can

construct the overall relationship:

$$\begin{bmatrix} V_1 \\ V_2 \\ . \\ . \\ . \\ V_{n-1} \\ V_n \end{bmatrix} b = 0 \tag{12}$$

We can then use SVD to solve for b, which gives us ω. We now need to solve for K. This can be done through the following process:

$$x_0 = \frac{\omega_{01}\omega_{02} - \omega_{00}\omega_{12}}{\omega_{00}\omega_{11} - \omega_{01}^2} \tag{13}$$

$$\lambda = \omega_{22} - \frac{\omega_{02}^2 + x_0(\omega_{01}\omega_{02} - \omega_{00}\omega_{12})}{\omega_{00}} \tag{14}$$

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{00}}} \tag{15}$$

$$\alpha_x = \sqrt{\frac{\lambda\omega_{00}}{\omega_{00}\omega_{11} - \omega_{01}^2}} \tag{16}$$

$$s = -\frac{\omega_{01}\alpha_x^2\alpha_y}{\lambda} \tag{17}$$

$$y_0 = \frac{sx_0}{\alpha_y} - \frac{\omega_{02}\alpha_x^2}{\lambda} \tag{18}$$

Through these calculations, we can construct the intrinsic matrix, as its form is:

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{19}$$

Through this process, I achieved the following intrinsic matrices for the two cameras:

Dataset 1:

$$K_1 = \begin{bmatrix} 725.968 & 5.492 & 240.793 \\ 0 & 721.778 & 319.591 \\ 0 & 0 & 1 \end{bmatrix} \tag{20}$$

Dataset 2:

$$K_2 = \begin{bmatrix} 484.402 & 3.856 & 229.551 \\ 0 & 487.079 & 309.5772 \\ 0 & 0 & 1 \end{bmatrix} \tag{21}$$

We can now calculate the extrinsic parameters. This is done through the following process: It can be seen from Equation 5 that

$$[r1, r2, t] = K^{-1}H \tag{22}$$

However, we need to implement a normalization term in order to preserve homogeneity. Therefore, solving for $r1$, $r2$ and t can be done through:

$$r1 = \zeta K^{-1}h1 \tag{23}$$

$$r2 = \zeta K^{-1}h2 \tag{24}$$

$$t = \zeta K^{-1}h3 \tag{25}$$

, where $\zeta = \frac{1}{||K^{-1}h1||}$

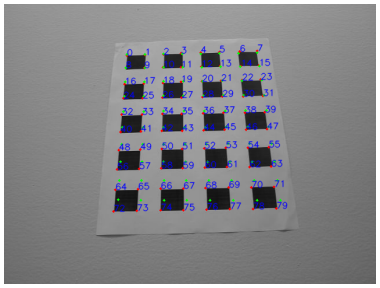
The final parameter $r3$ can be found through the cross product of r1 and r2, yielding $r3 = r1 \times r2$.

This then provides us with our three rotation vectors and our single translation vector that estimates the extrinsic parameters of the image. The final extrinsic matrix can be constructed through:

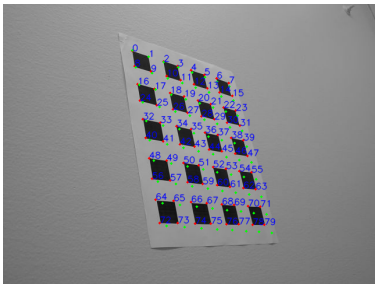
$$[R|t] = \begin{bmatrix} r1 & r2 & r3 & t \end{bmatrix} \tag{26}$$

The initial results of this can be observed through reprojection of the world coordinate points back onto the image and comparing the locations of the originally found corners vs. the reprojected corners. The results of this can be seen across both datasets below:

Dataset 1:

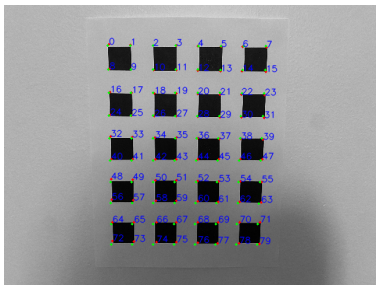


(a) Dataset 1 Reprojected Corners 1

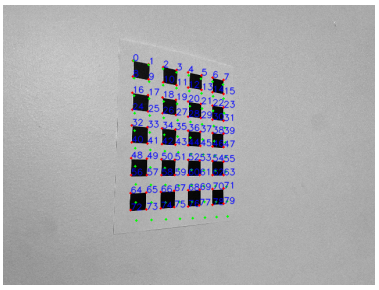


(b) Dataset 1 Reprojected Corners 2

Dataset 2:



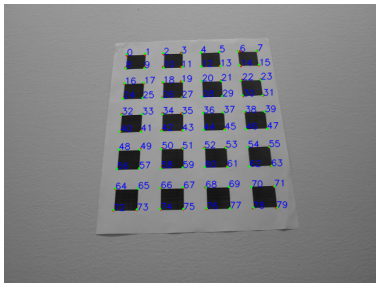
(a) Dataset 2 Reprojected Corners 1



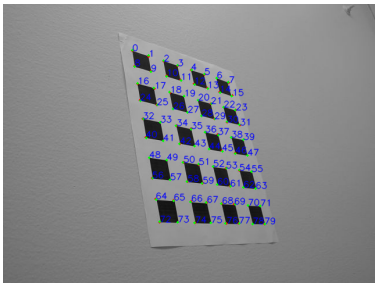
(b) Dataset 2 Reprojected Corners 2

It can be seen at large angles, the extrinsic parameters break down a bit and are not that accurate. I therefore applied Levenberg-Marquadt (LM) nonlinear least squares to minimize the reprojection error and therefore optimize the extrinsic calibration. The results of this can be seen below:

Dataset 1:



(a) Dataset 1 Reprojected Corners after LM 1



(b) Dataset 1 Reprojected Corners after LM 2

The final estimated extrinsic matrices for these two images from Dataset 1 were:

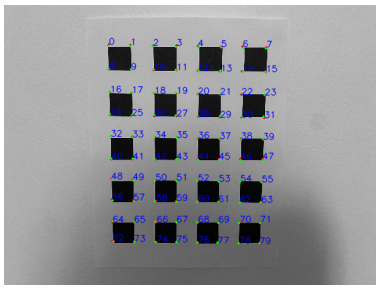
$$[R|t]_{dataset1,image1} = \begin{bmatrix} 1.061 & -0.247 & 0.608 & 20.018 \\ 0.389 & 1.429 & 0.039 & -275.63 \\ -0.829 & 0.119 & 0.793 & 803.36 \end{bmatrix} \tag{27}$$

and

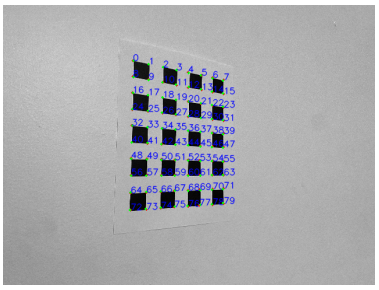
$$[R|t]_{dataset1,image2} = \begin{bmatrix} 807.17 & -27.38 & -0.526 & 1.482e4 \\ -56.97 & 894.133 & -0.079 & -1.39e5 \\ 499.617 & 69.128 & 0.847 & 4.486e5 \end{bmatrix} \tag{28}$$

For image 1 the mean reprojection error (Euclidean distance) was 0.91 and the variance was 0.2. For image 2 the mean reprojection error was 0.89 and the variance was 0.45.

Dataset 2:



(a) Dataset 2 Reprojected Corners after LM 1



(b) Dataset 2 Reprojected Corners after LM 2

The final extrinsic matrices for these two images from Dataset 2 were:

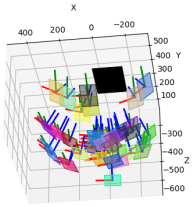
$$[R|t]_{dataset2,image1} = \begin{bmatrix} 1.04e2 & 1.18 & -5.82e-3 & -3.29e3 \\ 1.36 & 1.02e2 & -1.2e-1 & -1.6e4 \\ 8.97e-1 & 1.27e1 & 9.93e-1 & 3.25e4 \end{bmatrix} \tag{29}$$

and

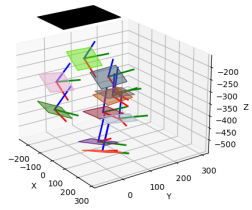
$$[R|t]_{dataset2,image2} = \begin{bmatrix} 2.9 & -9.23e-2 & -4.8e-1 & -8.18 \\ -7.1e-2 & 3.1 & 5.18e-2 & -5.78e2 \\ 1.58 & -2.26e-1 & 8.77e-1 & 1.32e3 \end{bmatrix} \tag{30}$$

For image 1 the mean reprojection error (Euclidean distance) was 1.17 and the variance was 0.37. For image 2 the mean reprojection error was 0.78 and the variance was 0.14.

The camera poses could then be plotted in 3D to see all of the different positions and orientations of the camera in relation to the calibration board - this serves as yet another visual check that the extrinsic calibration is correct. This can be seen below:



(a) Dataset 1 Camera Poses



(b) Dataset 2 Camera Poses

3 Code

```python
import cv2
import numpy as np
import time
import random
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from scipy.optimize import least_squares


# Canny edge detection parameters
canny_threshold1 = 200
canny_threshold2 = 400

# Hough transform parameters
rho_threshold = 1
theta_threshold = np.pi / 180
hough_threshold = 50

# Calibration pattern parameters
square_distance = 25
pattern = [5, 4]

visualize_reprojection = True
visualize_poses = True

perform_LM = True


def HC(point):
    # Helper function to convert a point to homogeneous coordinates
    return np.array([point[0], point[1], 1])

def polar_to_cartesian(rho, theta, length=1500):
    # Converting from polar coordinates to cartesian coordinates to get arbitrary endpoints of the line
    x0 = rho * np.cos(theta)
    y0 = rho * np.sin(theta)
    x1 = int(x0 + length * (-np.sin(theta)))
    y1 = int(y0 + length * (np.cos(theta)))
    x2 = int(x0 - length * (-np.sin(theta)))
    y2 = int(y0 - length * (np.cos(theta)))

    return (x1, y1, x2, y2)

def detect_corners(image, visualize_corner_detection=False):

    # Performing canny edge detection
    edges = cv2.Canny(image, canny_threshold1, canny_threshold2)

    detected_edge_image = cv2.cvtColor(image.copy(), cv2.COLOR_GRAY2BGR)
    detected_corners_image = detected_edge_image.copy()

    image_height, image_width = image.shape

    # Performing hough transform to extract lines
```

```
    lines = cv2.HoughLines(edges, rho_threshold, theta_threshold,
    hough_threshold)

    # Filtering out duplicate lines for the same edge that deviate
    by small angles
    rho_filter = 15
    theta_filter = np.deg2rad(5)

    filtered_lines = []

    # Sorting lines by rho
    if lines is not None:
        lines = sorted(lines, key=lambda x: x[0][0])

    # Checking if both rho and theta differences are large to
    qualify as a unique line
    for line in lines:
        # collect individual line's rho and theta values
        rho, theta = line[0]
        is_unique = True

        for unique_rho, unique_theta, _ in filtered_lines:
            # Checking if rho falls within a threshold of any
    previous rho as well as theta falling within a theta threshold
    of any previous theta to check if lines are duplicates. Check
    is complicated due to bounds of rho and theta for Hough
            # First check is for positive rho, and therefor angle
    differences close to zero. If rho and unique_rho are opposite
    in sign, angle will be close to pi if lines are duplicates
            if (abs(rho - unique_rho) < rho_filter and abs(theta -
    unique_theta) < theta_filter) or (abs(abs(rho) - abs(unique_rho
    )) < rho_filter and abs(theta - unique_theta) > (np.pi -
    theta_filter)):
                is_unique = False
                break

        # If unique flag still true, line passed checking against
    all other lines and is in fact not a duplicate
        if is_unique:
            x1, y1, x2, y2 = polar_to_cartesian(rho, theta)
            line_cartesian = (x1, y1, x2, y2)

            filtered_lines.append((rho, theta, line_cartesian))



    # Sorting lines by rho and vertical/horizontal to provide
    consistent labelling for each eventually found corner
    vertical_lines = []
    horizontal_lines = []

    for line in filtered_lines:
        rho, theta, line_cartesian = line
        if(abs(theta) < np.deg2rad(50) or abs(theta - np.pi) < np.
    deg2rad(50)):
            vertical_lines.append(line)
```

```
        elif(abs(theta - np.pi/2) < np.deg2rad(50) or abs(theta - 3*np.pi/2) < np.deg2rad(50)):
            horizontal_lines.append(line)

    # Sorting by abs for vertical lines only as they were more prone to angle switching
    vertical_lines.sort(key=lambda x: abs(x[0]))
    horizontal_lines.sort(key=lambda x: x[0])

    # Rearranging filtered_lines such that horizontal are always considered first - helps with consistent labelling
    filtered_lines = horizontal_lines + vertical_lines


    # Finding corners using cross products in HC
    corners = []

    for i in range(len(filtered_lines)):
        line_cartesian_test = filtered_lines[i][2]

        endpoint1_hc_test = HC((line_cartesian_test[0], line_cartesian_test[1]))
        endpoint2_hc_test = HC((line_cartesian_test[2], line_cartesian_test[3]))

        line_hc_test = np.cross(endpoint1_hc_test, endpoint2_hc_test)

        for j in range(len(filtered_lines)):
            if(i != j):
                # If not the same line, extract both line's homogeneous coordinates and calculate intersection point
                line_cartesian = filtered_lines[j][2]

                endpoint1_hc = HC((line_cartesian[0], line_cartesian[1]))
                endpoint2_hc = HC((line_cartesian[2], line_cartesian[3]))

                line_hc = np.cross(endpoint1_hc, endpoint2_hc)

                intersection = np.cross(line_hc_test, line_hc)

                is_valid_corner = True
                is_unique_corner = True

                # Checking if ideal point, if it is, flas as invalid as these do not reside in the image
                if(intersection[2] != 0):
                    intersection = intersection / intersection[2]
                    if(intersection[0] < 0 or intersection[1] < 0 or intersection[1] > image_height or intersection[0] > image_width):
                        is_valid_corner = False # Intersection out of the bounds of the image
                else:
```

```
                    is_valid_corner = False # Intersection is ideal point

                # Check if corner has already been collected as a unique corner
                for corner in corners:
                    if(intersection[0] == corner[0] and intersection[1] == corner[1]):
                        is_unique_corner = False
                        break

                # Only add to corners if the corner is in the image and unique
                if is_unique_corner and is_valid_corner:
                    corners.append((intersection[0], intersection[1]))


    # Visualizing detected lines
    if(visualize_corner_detection):
        for i, filtered_line in enumerate(filtered_lines):
            x1, y1, x2, y2 = filtered_line[2]

            cv2.line(detected_edge_image, (x1, y1), (x2, y2), (0, 255, 0), 2)

        cv2.imshow("original", image)
        cv2.imshow("edges", edges)
        cv2.imshow("hough after filtering", detected_edge_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    return corners

def generate_world_corners():
    world_corners = []

    # Multiplying parameters by 2 because there are two corners along each direction of the square
    for row in range(2 * pattern[0]):
        for col in range(2 * pattern[1]):
            world_corners.append((square_distance * col, square_distance * row, 0))

    return world_corners

def find_homography(world_corners, image_corners):
    A = []

    # Creating point correspondencies and solving using SVD
    for idx in range(len(image_corners)):
        x_w, y_w = world_corners[idx][0], world_corners[idx][1]
        x_i, y_i = image_corners[idx][0], image_corners[idx][1]
        A.append([-x_w, -y_w, -1, 0, 0, 0, x_i * x_w, x_i * y_w, x_i])
        A.append([0, 0, 0, -x_w, -y_w, -1, y_i * x_w, y_i * y_w, y_i])
```

```
188
189     A = np.array(A)
190
191     _, _, Vt = np.linalg.svd(A) # Using SVD to solve for homography
         terms
192     H = Vt[-1].reshape((3, 3))
193
194     return H / H[2, 2]
195
196 def find_V(H, i, j):
197     # Creating indexed v-vector for intrinsic calibration in
        accordance with Zhang paper
198     return np.array([
199         H[0][i] * H[0][j],
200         H[0][i] * H[1][j] + H[1][i] * H[0][j],
201         H[1][i] * H[1][j],
202         H[2][i] * H[0][j] + H[0][i] * H[2][j],
203         H[2][i] * H[1][j] + H[1][i] * H[2][j],
204         H[2][i] * H[2][j]
205     ])
206
207 def intrinsic_calibration(homographies):
208     # Creating Vb = 0 relationship and solving for b using SVD
209     V = []
210     for H in homographies:
211         V.append(find_V(H, 0, 1))
212         V.append(find_V(H, 0, 0) - find_V(H, 1, 1))
213
214     V = np.array(V)
215     _, _, Vt = np.linalg.svd(V)
216     b = Vt[-1]
217
218     # Extracting omega from b
219     w11, w12, w22, w13, w23, w33 = b
220
221     # Calculating intrinsic parameters from image of absolute conic
222     x0 = (w12 * w13 - w11 * w23) / (w11 * w22 - w12**2)
223     lambda_ = w33 - (w13**2 + x0*(w12 * w13 - w11 * w23)) / w11
224     alpha_x = np.sqrt(lambda_ / w11)
225     alpha_y = np.sqrt((lambda_ * w11) / (w11 * w22 - w12**2))
226     s = -(w12 * alpha_x**2 * alpha_y) / lambda_
227     y0 = s*x0 / alpha_y - (w13 * alpha_x**2) / lambda_
228
229     # Creation of intrinsic calibration matrix from intrinsic
        parameters
230     K = np.array([
231         [alpha_x, s, x0],
232         [0, alpha_y, y0],
233         [0, 0, 1]
234     ])
235
236     return K
237
238 def condition_R(R):
239     U, _, Vt = np.linalg.svd(R)
240     R_conditioned = np.dot(U, Vt)
241
```

```
    # If determinant is negative, negate last column to create a proper determinant of 1
    if np.linalg.det(R_conditioned) < 0:
        U[:, -1] *= -1
        R_conditioned = U @ Vt

    # Normalization of the columns of the orthogonal matrix to produce an orthonormal matrix
    R_conditioned[:, 0] /= np.linalg.norm(R_conditioned[:, 0])
    R_conditioned[:, 1] /= np.linalg.norm(R_conditioned[:, 1])
    R_conditioned[:, 2] /= np.linalg.norm(R_conditioned[:, 2])

    return R_conditioned

def reproject_points(world_corners, K, Rt):
    # Applying extrinsic calibration matrix to world points and extracting image location
    projected_points = []
    for x in world_corners:
        x_hc = np.array([x[0], x[1], x[2], 1])
        projected_point = K @ Rt @ x_hc
        projected_point /= projected_point[2]
        projected_points.append((projected_point[0], projected_point[1]))

    return np.array(projected_points)

def reprojection_error(params, K, world_corners, image_corners):
    # Helper function that defines reprojection error function used for LM minimization
    R = params[:9].reshape(3, 3)
    t = params[9:].reshape(3, 1)

    projected_points = reproject_points(world_corners, K, np.hstack((R, t)))

    error = (projected_points - image_corners).ravel()

    return error

def reprojection_visualization(homographies, corners_list, extrinsics, optimized_extrinsics, images):
    # Labels corners, before_lm reprojected corners, and after LM reprojected corners for visualization on how well the extrinsic parameters are both before and after LM
    for j in range(len(homographies)):
        corners = corners_list[j]
        extrinsic = extrinsics[j]
        homography = homographies[j]
        optimized_extrinsic = optimized_extrinsics[j]
        detected_corners_image = cv2.cvtColor(images[j], cv2.COLOR_GRAY2BGR)
        optimized_corners_image = detected_corners_image.copy()

        print("Extrinsic matrix: ", optimized_extrinsic)

        # Using Euclidean distance to calculate reprojection error
```

```
        projected_points = reproject_points(world_corners, K, optimized_extrinsic)

        reprojection_error = []

        for c, corner in enumerate(corners):
            distance = np.sqrt((corner[0] - projected_points[c][0])**2 + (corner[1] - projected_points[c][1])**2)
            reprojection_error.append(distance)

        mean_rproj_error = np.mean(reprojection_error)
        var_rproj_error = np.var(reprojection_error)

        print("Mean of reprojection error: ", mean_rproj_error)
        print("Variance of reprojection error: ", var_rproj_error)

        # Reprojecting corners with both non-LM and LM
        reprojected_corners_initial = reproject_points(world_corners, K, extrinsic)
        reprojected_corners_optimized = reproject_points(world_corners, K, optimized_extrinsic)

        for i, corner in enumerate(corners):
            corner_location = (int(corner[0]), int(corner[1]))
            reprojected_corner_initial = (int(reprojected_corners_initial[i][0]), int(reprojected_corners_initial[i][1]))
            reprojected_corner_optimized = (int(reprojected_corners_optimized[i][0]), int(reprojected_corners_optimized[i][1]))

            cv2.circle(detected_corners_image, corner_location, 2, (0, 0, 255), -1)
            cv2.circle(detected_corners_image, reprojected_corner_initial, 2, (0, 255, 0), -1)

            cv2.circle(optimized_corners_image, corner_location, 2, (0, 0, 255), -1)
            cv2.circle(optimized_corners_image, reprojected_corner_optimized, 2, (0, 255, 0), -1)

            cv2.putText(detected_corners_image, "{}".format(i), corner_location, cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1, cv2.LINE_AA)
            cv2.putText(optimized_corners_image, "{}".format(i), corner_location, cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1, cv2.LINE_AA)

        cv2.imshow("corners", detected_corners_image)
        cv2.imshow("corners optimized", optimized_corners_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()


world_corners = generate_world_corners()
homographies = []
corners_list = []
```

```
extrinsics = []
optimized_extrinsics = []
images = []

# Load in images and estimate homography
for i in range(1, 41):
    image = cv2.imread('HW8-Files/Dataset1/Pic_' + str(i) + '.jpg',
     cv2.IMREAD_GRAYSCALE)
    # image = cv2.resize(cv2.imread('HW8-Files/Dataset2/' + str(i)
    + '.jpg', cv2.IMREAD_GRAYSCALE), (640, 480))

    corners = detect_corners(image)

    # Ensuring to discard images that don't have all 80 corners
    detected
    if(len(corners) != len(world_corners)):
        continue

    H = find_homography(world_corners, corners)
    homographies.append(H)
    corners_list.append(corners)
    images.append(image)

# Calculate intrinsics
K = intrinsic_calibration(homographies)

print("Intrinsic Matrix K: ", K)

# Calculating extrinsic parameters and thereby extrinsic matrices
    for each image
for i in range(len(homographies)):
    H = homographies[i]
    h1, h2, h3 = H[:, 0], H[:, 1], H[:, 2]
    zeta = 1 / np.linalg.norm(np.dot(np.linalg.inv(K), h1))
    r1 = zeta * np.dot(np.linalg.inv(K), h1)
    r2 = zeta * np.dot(np.linalg.inv(K), h2)
    r3 = np.cross(r1, r2)
    t = zeta * np.dot(np.linalg.inv(K), h3)

    R = np.column_stack((r1, r2, r3))

    R = condition_R(R)

    Rt = np.column_stack((R, t))
    extrinsics.append(Rt)

    # print("Original extrinsic calibration matrix is: ", Rt)

    initial_guess = np.hstack((R.flatten(), t))

    corners_array = np.array(corners_list[i])
    world_corners_array = np.array(world_corners)

    # Taking the reprojection error function as input to LM and
    trying to minimize it using initial extrinsic calibration as
    initial guess
    if(perform_LM):
```

```
        lm_result = least_squares(reprojection_error, initial_guess, args=(K, world_corners, corners_array), method="lm")

        optimized_params = lm_result.x
        optimized_R = optimized_params[:9].reshape(3, 3)
        optimized_t = optimized_params[9:].reshape(3, 1)

        optimized_Rt = np.hstack((optimized_R, optimized_t))
    else:
        optimized_Rt = Rt

    optimized_extrinsics.append(optimized_Rt)

# Visualization of the reprojection error
if visualize_reprojection:
    reprojection_visualization(homographies, corners_list, extrinsics, optimized_extrinsics, images)
if visualize_poses:
    # Parameters for camera pose visualization
    camera_axes = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
    scaling_factor = 100
    plane_size = scaling_factor
    plane_points = np.array([[-plane_size/2, -plane_size/2, 0], [plane_size/2, -plane_size/2, 0], [plane_size/2, plane_size/2, 0], [-plane_size/2, plane_size/2, 0]])
    calibration_board_points = np.array([[-scaling_factor, -scaling_factor, 0], [scaling_factor, -scaling_factor, 0], [scaling_factor, scaling_factor, 0], [-scaling_factor, scaling_factor, 0]])

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

    for i, Rt in enumerate(extrinsics):
        R = Rt[:, :3]
        t = Rt[:, 3:]

        # Extracting the camera axes
        X_cam_x = camera_axes[0].T.reshape((3, 1))
        X_cam_y = camera_axes[1].T.reshape((3, 1))
        X_cam_z = camera_axes[2].T.reshape((3, 1))

        # Calculation of camera center
        camera_center = -R.T @ t

        # Applying extrinsics to find world coordinates of camera axes
        X_world_x = R.T @ X_cam_x + camera_center
        X_world_y = R.T @ X_cam_y + camera_center
        X_world_z = R.T @ X_cam_z + camera_center

        # Applying extrinsics to plane vertices to find where camera principal plane lies in world3D
        world_plane_points = (R.T @ plane_points.T + camera_center).T

        # Plotting camera center
```

```
        ax.scatter(camera_center[0], camera_center[1],
    camera_center[2], color='r', s=1, label="Camera {}".format(i))

        # Normalization and scaling the camera axes
        direction_x = X_world_x - camera_center
        direction_y = X_world_y - camera_center
        direction_z = X_world_z - camera_center

        direction_x_norm = (direction_x / np.linalg.norm(
    direction_x)) * scaling_factor
        direction_y_norm = (direction_y / np.linalg.norm(
    direction_y)) * scaling_factor
        direction_z_norm = (direction_z / np.linalg.norm(
    direction_z)) * scaling_factor

        # Plotting camera axes
        ax.plot3D(
            [camera_center[0], camera_center[0] + direction_x_norm
    [0]],
            [camera_center[1], camera_center[1] + direction_x_norm
    [1]],
            [camera_center[2], camera_center[2] + direction_x_norm
    [2]], color='r', linewidth=2
        )

        ax.plot3D(
            [camera_center[0], camera_center[0] + direction_y_norm
    [0]],
            [camera_center[1], camera_center[1] + direction_y_norm
    [1]],
            [camera_center[2], camera_center[2] + direction_y_norm
    [2]], color='g', linewidth=2
        )

        ax.plot3D(
            [camera_center[0], camera_center[0] + direction_z_norm
    [0]],
            [camera_center[1], camera_center[1] + direction_z_norm
    [1]],
            [camera_center[2], camera_center[2] + direction_z_norm
    [2]], color='b', linewidth=2
        )

        # Plotting camera principal plane as rectangle
        plane_color = (random.randint(0, 255) / 255.0, random.
    randint(0, 255)/ 255.0, random.randint(0, 255) / 255.0)
        vertices = [world_plane_points]

        plane = Poly3DCollection(vertices, facecolors=plane_color,
    linewidths=1, edgecolors=plane_color, alpha=0.5)
        ax.add_collection3d(plane)

    # Plotting calibration board
    calibration_board_color = (0, 0, 0)
    calibration_board_vertices = [calibration_board_points]
    calibration_board = Poly3DCollection(calibration_board_vertices
    , facecolors=calibration_board_color, linewidths=1, edgecolors=
```

```
    'black', alpha=1)
    ax.add_collection3d(calibration_board)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.show()
```

Listing 1: Python Code