ECE 66100 Homework #8

by

Adrien Dubois (dubois6@purdue.edu)

November, 13, 2024

Contents

1 Theory Questions

1.1 Question 1:

Why is the following theoretical observation fundamental to Zhang's algorithm for camera calibration? If π denotes the plane that contains the calibration pattern, we can show that π samples the Absolute Conic $\Omega\infty$ at exactly two points that are the Circular Points for π. This is critical for Zhang's algorithm since we need these two circular points to sample the image of the absolute conic. Since the absolute conic is not actually present in the image, we need some other way to estimate its parameters so that it can be used in the following formula: $\omega = K^{-T}K^{-1}$. Through the method described above, we can move a camera around a calibration pattern to form different intersections between the plane at infinity

(represented by the plane superimposed on the calibration pattern) and any circle. These intersect in two points which are called circular points. By sampling enough of these points, we can form an accurate estimate of the image of the absolute conic and therefore calculate the intrinsic parameters of the camera.

1.2 Question 2:

As you might suspect, the image of the Absolute Conic $\Omega\infty$ on the camera sensor plane is also a conic that is typically denoted ω. How would you derive the algebraic form of ω from $\Omega\infty$? Can you prove that ω does not contain any real pixel locations? It is known that the absolute conic is defined by the intersection betwen the plane and infinity and any sphere. It is also known that any points on the plane at infinity have the form: $\begin{bmatrix} x & y & z & 0 \end{bmatrix}^T$ If we analyze the pixel coordinates for that point we can see that:

$$x_{pixel} = PX = KR[I|-C]\begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = KR\begin{bmatrix} x \\ y \\ z \end{bmatrix} = KRx_{direction} = Hx_d$$

It is also known that a conic transforms in the following way under a homography H:

$$C\prime = H^{-T}CH^{-1}$$

Also, since have derived that $H = KR$, we can derive the equation for the aboslute conic:

$$\omega = H^{-T}I_{3\times3}H^{-1} = (KR)^{-T}(KR)^{-1} = ((KR)^T)^{-1}(KR)^{-1}$$

$$= (R^TK^T)^{-1}(KR)^{-1} = K^{-T}R^{-T}R^{-1}K^{-1} = K^{-T}K^{-1} = K^{-T}K^{-1}$$

Next, it is also known that any pixel on the absolute conic must satisfy:

$$x^T\omega x = 0$$

Therefore, since we know that $\omega = K^{-T}K^{-1}$ and that $K^{-T}K^{-1}$ is positive definite we can derive that in this case:

$$x^T\omega x(\text{must be}) > 0$$

. So, these points are imaginary and ω cannot contain any real pixel locations.

2 Corner Detection:

I implemented the corner detection logic by first converting the image to pure black and white pixels through a threshold at a gray level of 50. I found this to produce much more consistent results compared to the baseline graylevel conversion recommended in the instructions. Afterwards, I use a canny operator with parameters: **threshold1 = 40, threshold2 = 500**. I then ran scipy's HoughLinesP functionality as it performed better than the basic HoughLines function. My parameters for the HouglinesP function where:

$$\text{params} = \begin{cases} \text{rho} & 1 \\ \text{thera} & \frac{\pi}{180} \\ \text{threshold} & 40 \\ \text{minLineLength} & 50 \\ \text{maxLineGap} & 100 \end{cases}$$

It is important to note that these parameters worked for all 62 images except 2; however, the output of HoughLinesP were still very sensitive to small changes in the parameters above. Running this function produces a set of lines along the boundary pixels of the calibration squares. Therefore, we need to clean up the extra lines in the image. I do this through two passes of K-means:

- I first split the lines into two sets of vertical and horizontal lines by the angles with respect to the y-axis. If the absolute value of the angle is less than 45deg or greater than 135deg, I consider it to be vertical and the rest are horizontal.
- For horizontal lines, I form 10 clusters by the y-intercept parameter, and return the cluster centers as my line equations with both the slope and y-intercepts.

- On the other hand, for vertical lines, I form 8 clusters by the x-intercept, and also return the average slope and y-intercept in each cluster.

This method was very fast, consistent, and successful for cleaning up the hough lines into a set of 18 lines directly along the borders of the calibration pattern.

Lastly, to create my corner points, I sort the horizontal lines by y-intercept, and vertical lines by x-intercepts. This allows me to calculate the intersection points between two homogenous coordinate equations of line pairs through their cross product. Additionally, the order of the intersection points will be consistent along all images and orientations which is key for the performance of Zhang's algorithm.

3 Zhang's Algorithm:

3.1 V matrix calculation:

From a list of homographies, intersection points and real world coordinates, we can now all the information required for Zhang's algorithm. The first step of Zhang's algorithm involves generate the V matrix which is used to ω, the image of the absoulte conic. To do so, we need the following values for each image:

$$\begin{bmatrix} V_{1,2}^T \\ (V_{1,1} - V_{2,2})^T \end{bmatrix}$$

Therefore, the resulting V matrix will have a shape of $(2 * \text{number of images}, 6)$. Additionally, we define each $V_{i,j}$ using the following equations:

$$\begin{bmatrix} h_{0,i} * h_{0,j} \\ h_{0,i} * h_{1,j} + h_{1,i} * h_{0,j} \\ h_{1,i} * h_{1,j} \\ h_{2,i} * h_{0,j} + h_{0,i} * h_{2,j} \\ h_{2,i} * h_{1,j} + h_{1,i} * h_{2,j} \\ h_{2,i} * h_{2,j} \end{bmatrix}$$

Using the representation above, we can calculate the paramters of V through the following equation:

$$\begin{bmatrix} V_{1,2}^T \\ (V_{1,1} - V_{2,2})^T \end{bmatrix} \begin{bmatrix} w_{11} \\ w_{12} \\ w_{22} \\ w_{13} \\ w_{23} \\ w_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \end{bmatrix}$$

Therefore, we can calculate the paramters of omega through the Null space of the complete V matrix. This can be easily calculated by performed a singular value decomposition (S.V.D.) of the V matrix into U, V, V^T, and the null space will be the first row of V^T.

3.2 K matrix calculation:

From the paramaters for ω, we can calculate the intrinsic parameters of the camera. We know that:

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

Therefore, solving for K requires the following equations:

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

Where:

$$\begin{cases} y_0 = & (\omega_1 2 * \omega_1 3 - \omega_1 1 * \omega_2 3)/(\omega_1 1 * \omega_2 2 - \omega_1 2 * *2) \\ lambda_p aram = & \omega_3 3 - \frac{\omega_1 3 * *2 + y_0 * (\omega_1 2 * \omega_1 3 - \omega_1 1 * \omega_2 3)}{\omega_1 1} \\ alpha_x = & \sqrt{\frac{\lambda}{\omega_1 1}} \\ alpha_y = & \sqrt{\frac{(\lambda * \omega_1 1)}{(\omega_1 1 * \omega_2 2 - \omega_1 2 * *2)}} \\ s = & -\frac{\omega_1 2 * \alpha_x * *2 * \alpha_y}{\lambda} \\ x_0 = & ((s \times y_0)/\alpha_y) - ((\omega_1 3 * \alpha_x * *2)/\lambda) \end{cases}$$

3.3 Linear Estimation of R and T:

Afterwards, we can use the intrinsic parameters for the camera, and each image homography from the real world coordinates of the corners to the estimated corners points using the Hough Transform to calculate an initial solution for the rotation and translation matrices for each camera.

Given a homography

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{h_1} & \boldsymbol{h_2} & \boldsymbol{h_2} \end{bmatrix}$$

, we can estimate the R and t matrices as follows:

$$\begin{cases} \xi & = \frac{1}{\|K^{-1} h_1\|} \\ r_1 & = \xi K^{-1} h_1 \\ r_2 & = \xi K^{-1} h_2 \\ r_3 & = r_1 \times r_2 \\ t & = \xi K^{-1} h_3 \end{cases}$$

It is important to note that we apply the scaling factor ξ to normalize the matrix. Additionally, since a rotation matrix must be orthonormal we can condition $\boldsymbol{R}$ as follows:

$$\boldsymbol{R} = UDV^T$$

$$R = UV^T$$

3.4 Levenberg Marquadt Non-Linear Least Squares Estimation:

To compute the L.M. refined estimate for the intrinsic and extrinsic camera paramters, we need a list of all the learnable parameters, and a relevant cost function. It is important to note that while $\boldsymbol{R}$ and $\boldsymbol{K}$ are 3×3 matrices, they actually only have 3 and 5 degrees of freedom respectively. While the five degrees of freedom for the K matrix are clear from its definition, we must convert the 3×3 R matrix into a 3×1 Rodriguez representation $\boldsymbol{w}$. The forward and backward passes for this conversion is included below:

Forward Pass:

Included below is the computation of $\boldsymbol{w}$ from $\boldsymbol{R}$:

$$\begin{cases} \varphi & = \arccos \frac{(Tr(R) - 1)}{2} \\ w & = \frac{\varphi}{2 \sin(\varphi)} \begin{bmatrix} R_{3,2} - R2,3 \\ R_{1,3} - R3,1 \\ R_{2,1} - R1,2 \end{bmatrix} \end{cases}$$

Backward Pass:

Included below is the computation of $\boldsymbol{R}$ from $\boldsymbol{w}$:

$$\mathbf{R} = \begin{cases} \mathbf{I}_{3\times3} + \dfrac{\sin\varphi}{\varphi}[\mathbf{w}]_X + \dfrac{1-\cos\varphi}{\varphi^2}[\mathbf{w}]_X^2 \\ [\mathbf{w}]_X = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \\ \varphi = \|\mathbf{w}\| \end{cases}$$

Cost Function:

We define the geometric error for L.M. as:

$$\begin{cases} x_{ij,} & : \text{The detected corner points.} \\ x_{M,j} & : \text{The world coordinates of the corners on the calibration pattern.} \\ \epsilon_{geo} & = \sum_i \sum_j \left\| x_{i,j} - K \begin{bmatrix} r_{i,1} & r_{i,2} & t_i \end{bmatrix} \right\|^2 \end{cases}$$

Moreover, it is important to note that the geometric mean must be computed with respect to the actual coordinates $[x, y]$ not homogenous coordinates since the error could be minimized by multiplying the result by a very small scalar k without actually optimizing the parameters.

4 Plotting the Camera Poses:

Lastly, for the results section of this report I generated a 3D plot with the reconstructed camera poses around the calibration pattern. The centers $\boldsymbol{C}$ and axes $\boldsymbol{X}$ of the camera poses are calculated as follows:

$$C = -R^T t$$

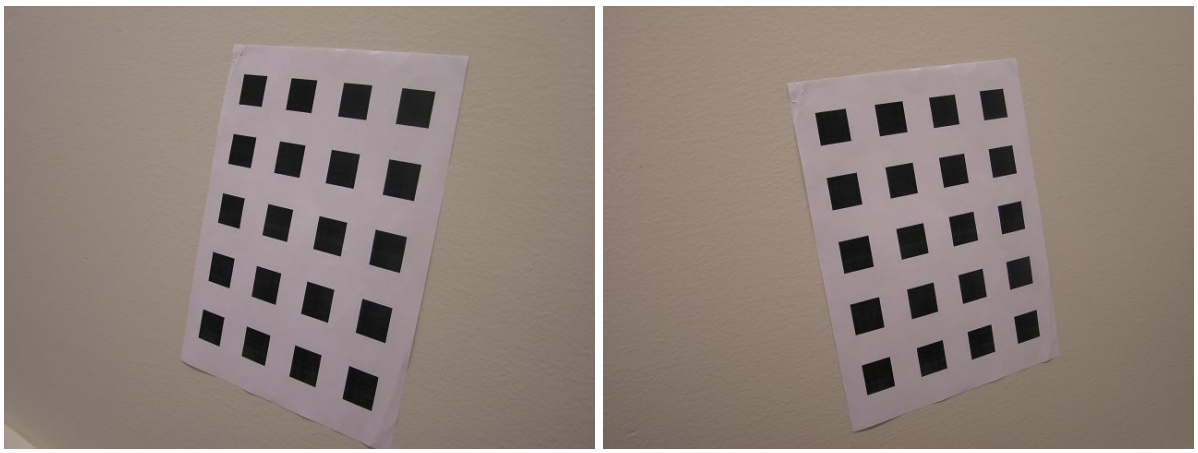
$$X_{camera} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$X = R^T X_{camera} + C$$

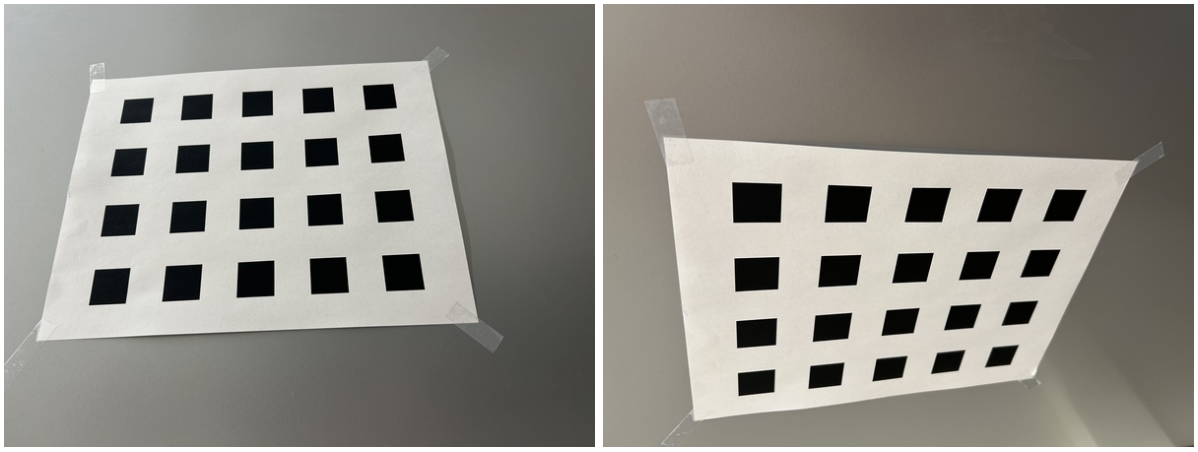
5 Results:

5.1 Original Image Samples:

Given Dataset:

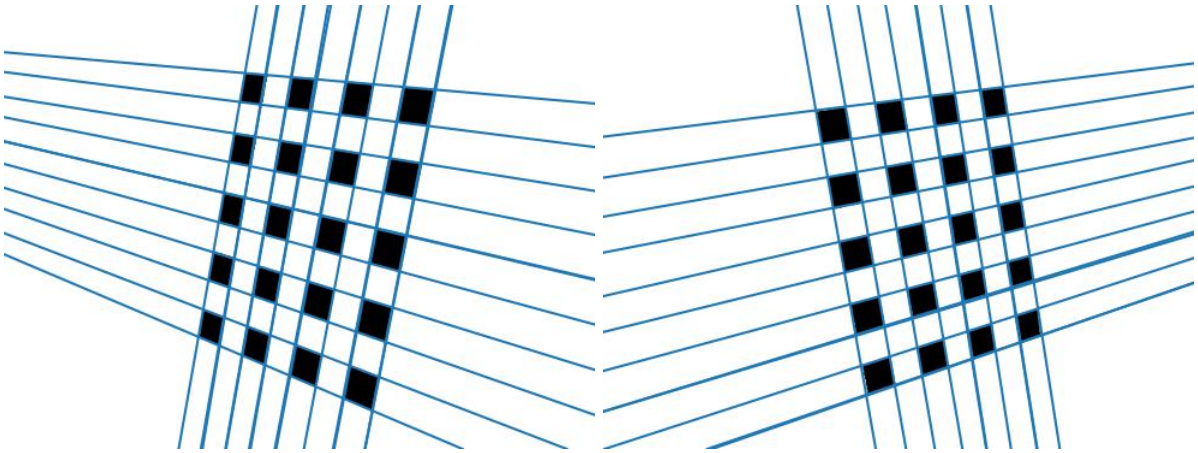


My Dataset:

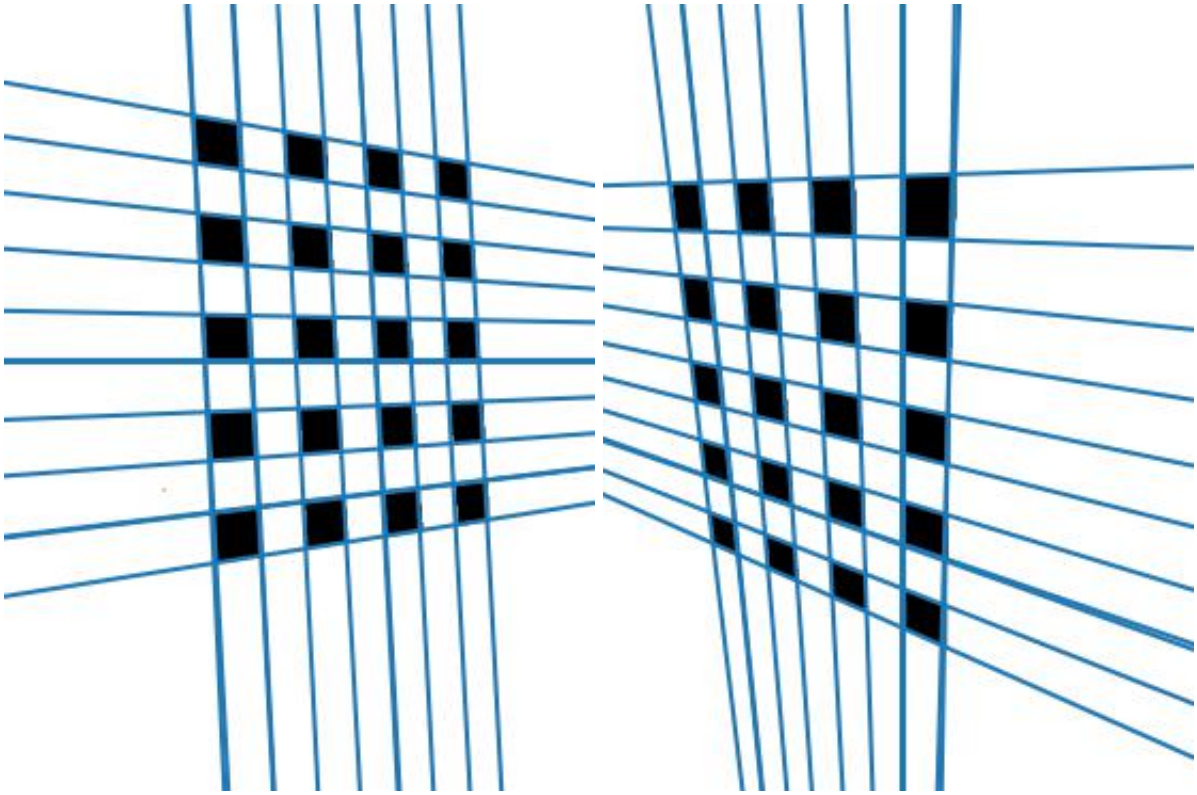


5.2 Lines created through HoughLinesP

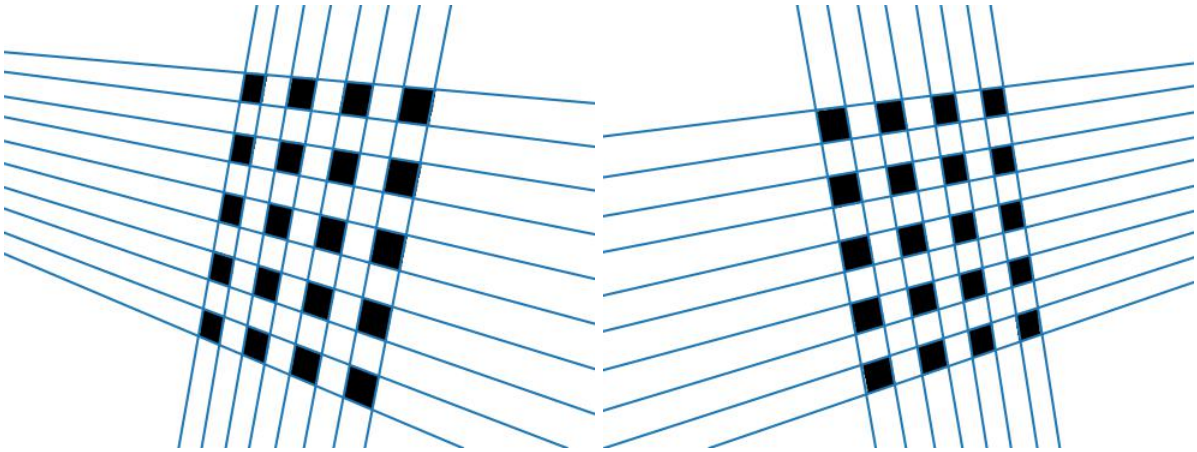
Given Dataset:



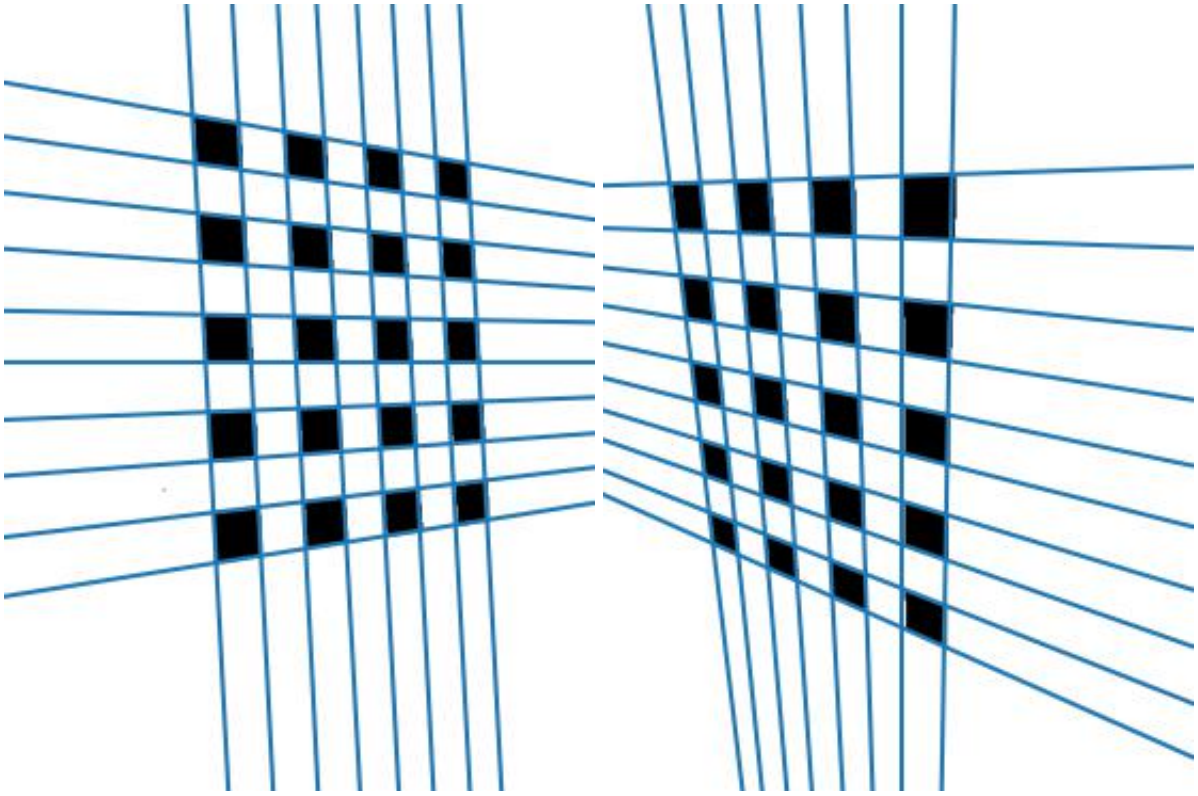
My Dataset:



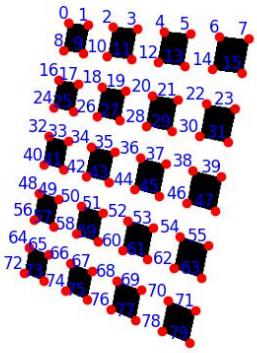
5.3 Cleaned up lines:

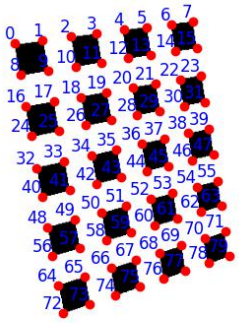


My Dataset:

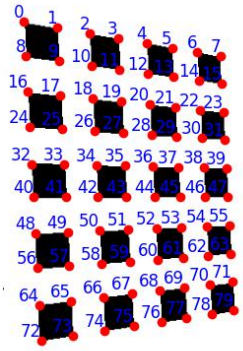


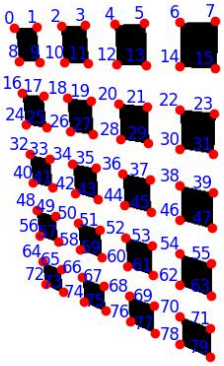
5.4 Extracted Points:



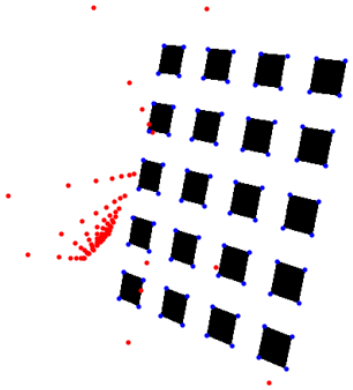


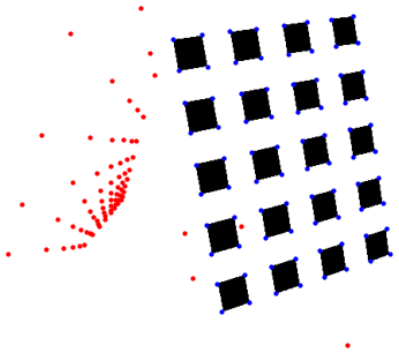
My Dataset:



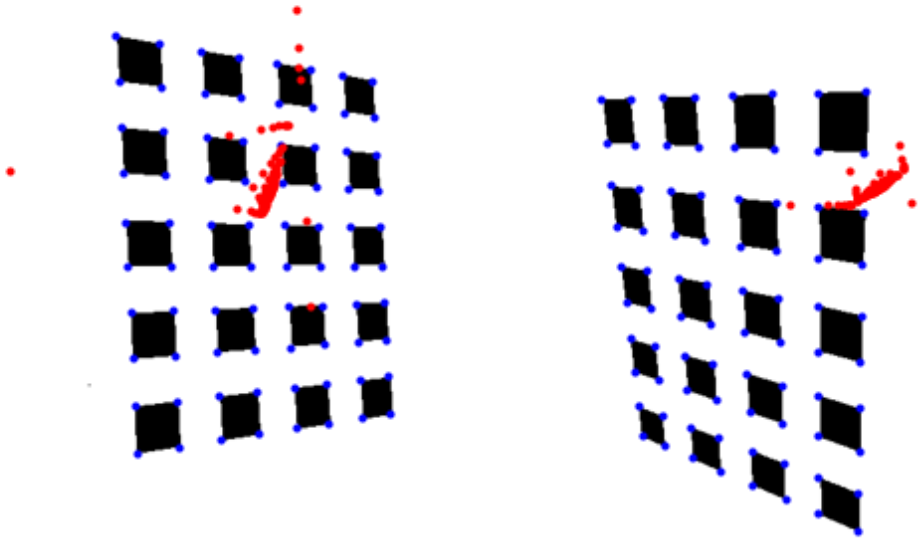


5.5 Reprojected world points through K, R and t:





My Dataset:



5.6 Reprojection results after L.M.:

5.6.1 Error Costs for LM:

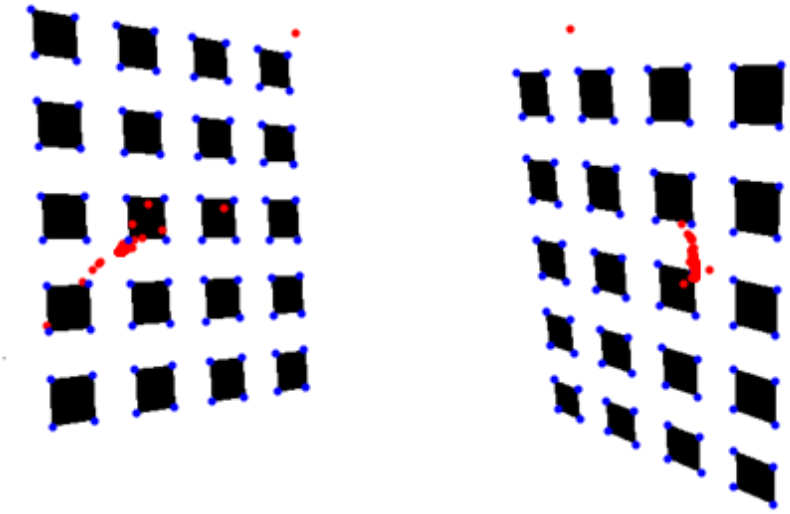
For the given dataset, LM reduced the cost function as follows:

Function evaluations 590670, initial cost 3.4033e+09, final cost 9.0467e+06, first-order optimality 2.92e+06.

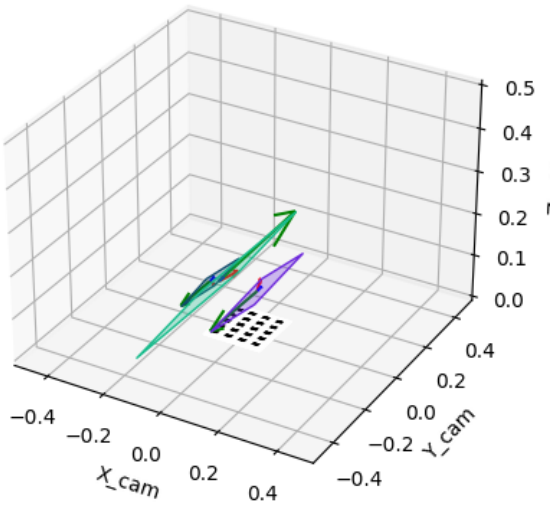
On the other hand, for my dataset the LM reduced the error as follows:

Function evaluations 600659, initial cost 2.8931e+07, final cost 8.8417e+06, first-order optimality 2.93e+02

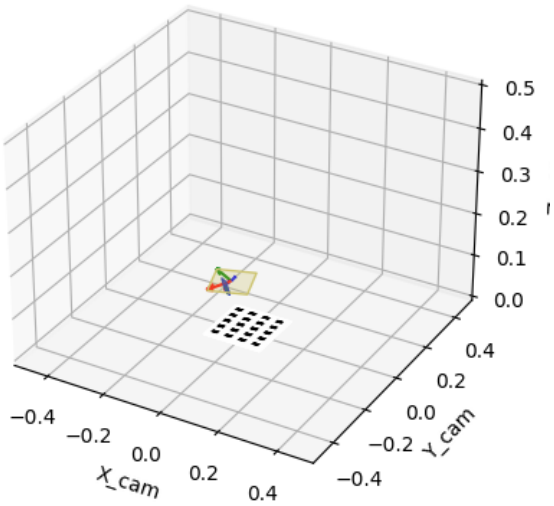
My Dataset:



5.7 Plotted Camera Poses:



My Dataset:



6 Full Code Printout:

6.1 Corner Detection

```
def open_image_in_black_and_white(img_path):
    # Open the image:
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Convert the image to pure black and white:
    _, img = cv2.threshold(img, 50, 255, cv2.THRESH_BINARY)

    return img

def get_sorted_lines(hlines, vlines):
    # Sort the horizontal lines in ascending order by y_intercept:
    hlines = hlines[np.argsort(hlines[:, 1])]

    # Sort the vertical lines in ascending order by x_intercept:
    x_intercepts = []
    for (slope, y_intercept) in vlines:
        x_intercept = -y_intercept / slope if slope != 0 else np.inf
        x_intercepts.append(x_intercept)
    x_intercepts = np.array(x_intercepts)
    vlines = vlines[np.argsort(x_intercepts)]

    return hlines, vlines

def get_real_intersection_coordinates(real_world_scaling):
    real_world_points = []
    for row in range(10):
        for col in range(8):
            real_world_points.append(Point(real_world_scaling*col, real_world_scaling*
    row))
    return real_world_points

def get_image_homography(img_list, real_world_scaling):
    homography_list = []
    all_intersection_points = []
    all_real_world_points = []
    for img_full_name in img_list:
        img_path = imgs_dir_path + img_full_name

        # Open the image thresholded to be black and white
        img = open_image_in_black_and_white(img_path)

```

```
        # Run canny operator for corner detection on the image
        edges = cv2.Canny(image=img, threshold1=40, threshold2=500)

        # Calculate the hough points
        hp = cv2.HoughLinesP(edges, rho=1, theta=np.pi/180, threshold=40, minLineLength
    =50, maxLineGap=100).squeeze()

        # Get the slope,y_intercept for the best 18 lines (removing noise)
        hlines, vlines = get_most_dissimilar_lines(hp)

        # Get the sorted lines:
        hlines, vlines = get_sorted_lines(hlines, vlines)

        # Convert the lines to homogenous coordinates:
        intersection_points = []
        for hline in hlines:
            for vline in vlines:
                hhline = Line.from_slope_and_intercept(hline[0], hline[1])
                hvline = Line.from_slope_and_intercept(vline[0], vline[1])

                intersection_points.append(hhline.get_intersection(hvline))
        all_intersection_points.append([point.hc for point in intersection_points])

        # Create world_coordinates
        real_world_points = get_real_intersection_coordinates(real_world_scaling=
    real_world_scaling)
        all_real_world_points.append([point.hc for point in real_world_points])

        # detected = H * real_world_points
        homography_list.append(Homography().estimate_projective_homography(x_points=
    real_world_points, x_prime_points=intersection_points))

    return homography_list, np.array(all_intersection_points), np.array(
    all_real_world_points)
```

6.1.1 Line Cleanup

```
def get_most_dissimilar_lines(hp):
"""Finds the 10 most dissimilar horizontal lines and 8 most dissimilar vertical
lines to remove noise in the output of the hough transform.

Args:
    hp (np.array): (N, 4) array of lines in the form (x1,y1,x2,y2)
"""
# Calculate angle of the lines with the y axis:
angles = np.degrees(np.arctan2((hp[:, 2] - hp[:, 0]), (hp[:, 3] - hp[:, 1])))

horizontal_mask = np.logical_and(np.abs(angles) > 45, np.abs(angles) < 135)
vertical_mask = np.logical_not(horizontal_mask)

# Remove noise from the horizontal lines, we do this through the y-intercept
if np.any(horizontal_mask):
    horizontal_lines = hp[horizontal_mask]

    # I first find the slope and intersept of each line and join them into one array
    # Since we want the clustering to mainly occur due to differences in the y-int,
it is fine to not normalize these value
    # Slopes ~= 0, y-ints = [100, 600] so this scaling issue would be important for
normal clustering
    slopes = (horizontal_lines[:, 3] - horizontal_lines[:, 1]) / (horizontal_lines
[:, 2] - horizontal_lines[:, 0] + 1e-10)  # avoid division by zero
    y_intercepts = horizontal_lines[:, 1] - slopes * horizontal_lines[:, 0]
    hline_params = np.vstack((slopes, y_intercepts)).transpose(1, 0)

    # Apply kmeans on these lines and return the average slope/y-intercept for each
cluster. This joins noisy line pairs together
    hkm = KMeans(n_clusters=10, random_state=0, n_init='auto')
    hkm.fit(hline_params)
    hlines_avg = hkm.cluster_centers_

# Remove noise from the vertical lines, we do this through the x-intercept:
if np.any(vertical_mask):
```

```
        vertical_lines = hp[vertical_mask]

        slopes = (vertical_lines[:, 3] - vertical_lines[:, 1]) / (vertical_lines[:, 2] -
     vertical_lines[:, 0] + 1e-10)  # avoid division by zero
        y_intercepts = vertical_lines[:, 1] - slopes * vertical_lines[:, 0]
        x_intercepts = -y_intercepts/slopes

        # Perform KMeans clustering on the x-intercepts:
        x_intercepts_2d = x_intercepts.reshape(-1, 1)
        vkm = KMeans(n_clusters=10, random_state=0, n_init='auto')
        vkm.fit(x_intercepts_2d)
        labels = vkm.labels_

        # Calculate average slope and y_intercept for each cluster:
        avg_slopes = []
        avg_y_intercepts = []

        # Loop over each cluster index (0 to 7 for 8 clusters)
        for cluster_id in range(8):
            # Get the indices of the lines in the cluster
            indices = np.where(labels == cluster_id)[0]

            # Compute average slope and y-intercept
            avg_slope = slopes[indices].mean()
            avg_y_intercept = y_intercepts[indices].mean()

            # Append the averages to the lists
            avg_slopes.append(avg_slope)
            avg_y_intercepts.append(avg_y_intercept)
        vlines_avg = np.stack((avg_slopes, avg_y_intercepts)).transpose(1, 0)

    return hlines_avg, vlines_avg
```

6.2 Zhang's Algorithm:

```
# Code to estimate omega
def get_V_ij_matrix(h, i, j):
    """ Computes V_ij from the 3by3 homography.

    Args:
        H (np.array): (3, 3) homography from intersection coordinates to real world
    points there
        i (int): row index
        j (int): col index
    """
    # Calculate the V_ij matrix from Avi's lecture 21 as described in Zhang's method
    V_ij = np.array([h[0][i] * h[0][j],
                         h[0][i] * h[1][j] + h[1][i] * h[0][j],
                         h[1][i] * h[1][j],
                         h[2][i] * h[0][j] + h[0][i] * h[2][j],
                         h[2][i] * h[1][j] + h[1][i] * h[2][j],
                         h[2][i] * h[2][j]])
    return V_ij

def get_complete_V_matrix(homography_list):
    V_matrix = []
    for homography in homography_list:
        v_11 = get_V_ij_matrix(homography, 0, 0)
        v_12 = get_V_ij_matrix(homography, 0, 1)
        v_22 = get_V_ij_matrix(homography, 1, 1)

        V_matrix.append(v_12.T)
        V_matrix.append((v_11 - v_22).T)
    v_mat = np.array(V_matrix)
    return v_mat

def get_omega_parameters(v_mat):
    # Perform Singular Value Decomposition
    _, _, Vt = np.linalg.svd(v_mat)

    # The null space is the last row of Vt
    omega = Vt[-1]
```

```
    return omega

# Code for camera specific parameters:
def get_K_matrix(omega):
    # Omega is: [w11, w12, w22, w13, w23, w33]
    w11, w12, w22, w13, w23, w33 = omega[0], omega[1], omega[2], omega[3], omega[4],
    omega[5]

    # Calculate the parameters of K using omega following Avi's lecture 21 page 3
    y_0                 = (w12*w13 - w11*w23) / (w11*w22 - w12**2)
    lambda_param        = w33 - (( w13**2 + y_0*(w12*w13 - w11*w23) ) / w11)
    alpha_x             = np.sqrt(lambda_param / w11)
    alpha_y             = np.sqrt((lambda_param*w11) / (w11*w22 - w12**2))
    s                   = -((w12 * alpha_x**2 * alpha_y)/lambda_param)
    x_0                 = ((s*y_0)/alpha_y) - ((w13 * alpha_x**2)/lambda_param)

    # Reconstruct K:
    K_mat = np.array([[alpha_x, s, x_0],
                      [0, alpha_y, y_0],
                      [0, 0, 1]])
    return K_mat

def get_r_and_t_mats_from_K_and_H(K_mat, homography_list):
    K_inv = np.linalg.inv(K_mat)

    R_mats = []
    t_mats = []
    for homography in homography_list:
        # Get the columns of the homography:
        h1, h2, h3 = homography[:, 0], homography[:, 1], homography[:, 2]

        # We need to rescale the rotation matrix to make it othernormal:
        scaling = 1 / np.linalg.norm(K_inv @ h1)
        r1 = scaling * K_inv @ h1
        r2 = scaling * K_inv @ h2
        r3 = np.cross(r1, r2)
        t = scaling * K_inv @ h3

        # Next, I have to orthonomalize R by getting R = UDVt and setting R to UVt
        # R is made from its three column vector components
        R = np.vstack([r1, r2, r3]).T
        U, _, Vt = np.linalg.svd(R)
        R_mats.append(U @ Vt)

        t_mats.append(t)

    return np.array(R_mats), np.array(t_mats)
```

6.3 Levenberg Marquadt Estimation:

```
# LM relevant code:
def get_lm_learnable_params(R_mats, t_mats, K_mat):
    # Create the w_matrix from R, where w only has 3 DoF
    w_mats = []
    for R in R_mats:
        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]]])
        w_mats.append(w)

    k_params = np.array([K_mat[0, 0], K_mat[0, 1], K_mat[0, 2], K_mat[1,1], K_mat[1,
    2]])

    return np.array(np.concatenate((np.array(w_mats).flatten(), t_mats.flatten(),
    k_params.flatten())))


def compute_R_mats_from_w(w_mats):
    R_mats_list = []
    for w_mat in w_mats:
        phi = np.linalg.norm(w_mat)
```

```
        sin_phi_over_phi = np.sin(phi) / phi
        one_minus_cos_phi_over_phi2 = (1 - np.cos(phi)) / (phi ** 2)

        wx, wy, wz = w_mat[0], w_mat[1], w_mat[2]

        # Calculate [w]_x matrices as per the skew-symmetric definition
        w_cross = np.zeros((3, 3))
        w_cross[0, 1] = -wz
        w_cross[0, 2] = wy
        w_cross[1, 0] = wz
        w_cross[1, 2] = -wx
        w_cross[2, 0] = -wy
        w_cross[2, 1] = wx

        # Identity matrix repeated for each w
        I = np.eye(3)  # Shape: (3, 3)

        # Rodrigues formula for each w in W_mats
        R_mats = I + sin_phi_over_phi * w_cross + one_minus_cos_phi_over_phi2 * np.
    matmul(w_cross, w_cross)

        R_mats_list.append(R_mats)

    return np.array(R_mats_list)


def unpack_parameters_for_lm(l_params, num_views):
    # l_params is:
    # - W_mats: 117 params, representing a list of 39 rotation matrices (3-element
    vectors each)
    # - t_mats: 117 params, representing a list of 39 translation vectors (3-element
    vectors each)
    # - K: 9 params, representing the 3x3 intrinsic matrix

    # First we extract a list of 3 element vectors for the rotation matrix from w
    w_mats = l_params[:num_views*3].reshape((-1, 3))

    # Next we need to regenerate the R matrices used for LM
    R_mats = compute_R_mats_from_w(w_mats)

    # T_mats is a list of 3 element vectors for the translation of the camera
    t_mats = l_params[num_views*3:2*num_views*3].reshape((-1, 3))

    # We need to make K a 3by3 array again
    K_params = l_params[-5:]
    K = np.array([[K_params[0], K_params[1], K_params[2]],
                  [0, K_params[3], K_params[4]],
                  [0, 0, 1]])

    return R_mats, t_mats, K

def cost_func(l_params, real_world_points, intersection_points, num_views):
    # First we extract the camera parameters from the learnable parameter array
    R_mats, t_mats, K = unpack_parameters_for_lm(l_params, num_views)
    projection_mat = K @ np.array([R_mats[..., 0], R_mats[..., 1], t_mats]).transpose(1,
     0, 2)

    # Expand the arrays for broadcasting
    proj_mat_expanded = projection_mat[:, np.newaxis, ...] # (39, 3, 3) to (39, 1, 3, 3)
    real_points_expanded = real_world_points[..., np.newaxis] # (39, 80, 3) to (39, 80,
    3, 1)

    # Calculate the projected points:
    projected_points = (proj_mat_expanded @ real_points_expanded).squeeze(-1)

    # Normalize by third_coord
    projected_points = projected_points / projected_points[..., 2][..., np.newaxis]

    # Calculate the residuals:
    residuals = intersection_points[..., 0:2] - projected_points[..., 0:2]

    return residuals.ravel()
```

6.4 Helper Functions:

Calling function:

```
imgs_dir_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW8/HW8-Files/Dataset1/"
# imgs_dir_path = "/mnt/cloudNAS3/Adubois/Classes/ECE661/HW8/Images/"
img_list = os.listdir(imgs_dir_path)
img_list = img_list[0:8]
num_views = len(img_list)
homography_list, intersection_points, real_world_points = get_image_homography(img_list,
     real_world_scaling=2.5)

# Calculate the V matrix:
v_mat = get_complete_V_matrix(homography_list)

# Solve for the omega paramters
omega = get_omega_parameters(v_mat)

# Find K matrix
K_mat = get_K_matrix(omega)

# Find R and T initial solution
R_mats, t_mats = get_r_and_t_mats_from_K_and_H(K_mat, homography_list)

# Get Levangerg Marquadt improved matrices
learnable_params = get_lm_learnable_params(R_mats, t_mats, K_mat)
optimal_params = least_squares(cost_func, learnable_params, args=(real_world_points,
    intersection_points, num_views), method='lm', verbose=2)
optim_R, optim_t, optim_K = unpack_parameters_for_lm(optimal_params.x, num_views)
```

Reprojecting Points:

```
projection_mat = optim_K @ np.array([optim_R[..., 0], optim_R[..., 1], optim_t]).
    transpose(1, 0, 2)

# Expand the arrays for broadcasting
proj_mat_expanded = projection_mat[:, np.newaxis, ...] # (39, 3, 3) to (39, 1, 3, 3)
real_points_expanded = real_world_points[..., np.newaxis] # (39, 80, 3) to (39, 80, 3,
    1)

# Calculate the projected points:
projected_points = (proj_mat_expanded @ real_points_expanded).squeeze(-1)

for img_name, image_points, img_proj_points in zip(img_list, intersection_points,
    projected_points):
    # Regenerate the image background:
    img_path = imgs_dir_path + img_name
    # Open the image thresholded to be black and white
    img = open_image_in_black_and_white(img_path)

    for point, proj_point in zip(image_points, img_proj_points):
        x, y, _ = point
        proj_point = proj_point / proj_point[2]
        x2, y2, _ = proj_point

        plt.plot(x, y, "bo", markersize=2)
        plt.plot(x2, y2, "ro", markersize=2)

    plt.imshow(img, 'gray')
    plt.axis("off")
    plt.show()
    plt.close()
```

Center and Axes Calculation:

```
C_mats = []
X_mats = []
for R_img, t_img in zip(optim_R, optim_t):
    C = - R_img.T @ t_img
    C_mats.append(C)
```

```
    X_cam = np.eye(3)

    X = R_img.T @ X_cam + C
    X_mats.append(X)
```