Computer Vision Homework 1 – Camera Calibration

Group 15 - 313553014 廖怡誠、313553013 李品妤、312553007 林芷萱

1. Introduction

Camera calibration is the process of determining a camera's intrinsic and extrinsic parameters, which reveals precise internal settings like focal length and lens distortion while also pinpointing its exact position and orientation in relation to a scene. This process transforms blurry, skewed images into crisp, accurate visuals, allowing us to convert 2D photos into reliable 3D data. Correcting distortions ensures every shot aligns perfectly with the real world, making it essential for applications ranging from virtual reality to robotic vision.

Camera calibration is crucial in computer vision tasks such as object tracking, image stitching, and 3D reconstruction. In robotics, it enables precise navigation and object manipulation by providing accurate depth information. Additionally, camera calibration is essential in augmented reality to align virtual objects with the real world, and it is also used in photogrammetry to create accurate 3D models from photographs.

In this assignment, we dive into camera calibration by implementing our own method and comparing it to the calibration technique from OpenCV. Using photos of a 2D chessboard taken from various angles, we derive both the extrinsic and intrinsic matrices. We then explore how the accuracy of our results changes as we adjust the number of photos used in the calibration process.

2. Implementation Procedure

In this section, we will briefly derive the mathematical formulas for camera calibration and explain what our target object is. At the same time, we will describe how our code is implemented.

2.1. Notation

A 2D point is denoted by m = $[u,\ v]^T$ and a 3D point is denoted by M = $[X,\ Y,\ Z]^T$. The relationship between a 3D point M and its image projection m is given by

$$s\mathbf{m} = A[R \quad t]\mathrm{M}$$

where s is an arbitrary scale factor, (R, t), called the extrinsic parameters, is the rotation and translation which related the world coordinate system to the camera coordinate system, and A, called the camera intrinsic matrix, is given by

$$\mathrm{A} = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

with (u_0, v_0) the coordinates of the principal point, α and β the scale factor in image u and v axes, and γ the parameter describing the skewness of the two image axes.

2.2. Solving for the Homography

Since all of the points in chessboards lie in the same plane, we assume the model plane is on $Z = 0$ of the world coordinate system. Let's denote the i^{th} column of the rotation matrix R by r_i. From (1), we have

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A[r_1 \quad r_2 \quad r_3 \quad t]\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = A[r_1 \quad r_2 \quad t]\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Where $A[r_1 \quad r_2 \quad t]$ is homography **H**, which is the 3×3 matrix and defined up to a scale factor. Therefore, a model point M and its image m is related by H:

$$s\mathbf{m} = \mathbf{H}M \text{ with } \mathbf{H} = A[r_1 \quad r_2 \quad t] \tag{2}$$

Through image analysis, we can obtain the 2D coordinates (**m**) of the grid points in each checkerboard picture. At the same time, we know that the 3D coordinates (M) of the checkerboard are (0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 2, 0), (2, 1, 0) and so on. Thus, we can solve for H and then we can infer $A[r_1 \quad r_2 \quad t]$.

Let $x = [\bar{h}_1^T \quad \bar{h}_2^T \quad \bar{h}_3^T]^T$, where $\bar{h}_i$ is the i^{th} row of H. Then equation (2) can be rewritten as

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} \frac{\bar{h}_1 \mathrm{M_i}}{\bar{h}_3 \mathrm{M_i}} \\ \frac{\bar{h}_2 \mathrm{M_i}}{\bar{h}_3 \mathrm{M_i}} \end{bmatrix} \tag{3}$$

Where $\mathrm{M_i}$ known from calibration rig. By expanding equation (3), we can obtain the following expression:

$$u_i(\bar{h}_3 \mathrm{M_i}) - \bar{h}_1 \mathrm{M_i} = 0$$

$$v_i(\bar{h}_3 \mathrm{M_i}) - \bar{h}_2 \mathrm{M_i} = 0$$

Then we also can be rewritten as:

$$\begin{bmatrix} -\mathrm{M}_i^T & 0 & u_i \mathrm{M}_i^T \\ 0 & -\mathrm{M}_i^T & v_i \mathrm{M}_i^T \end{bmatrix} \begin{bmatrix} \bar{h}_1^T \\ \bar{h}_2^T \\ \bar{h}_3^T \end{bmatrix} = \begin{bmatrix} -\mathrm{M}_i^T & 0 & u_i \mathrm{M}_i^T \\ 0 & -\mathrm{M}_i^T & v_i \mathrm{M}_i^T \end{bmatrix} x = Lx = 0$$

By expanding matrix **L**, each M_i^T can be $[\mathrm{X_i} \quad \mathrm{Y_i} \quad 1]$, resulting in the following expression:

$$\begin{bmatrix} -\mathrm{X_i} & -\mathrm{Y_i} & -1 & 0 & 0 & 0 & u_i\mathrm{X_i} & v_i\mathrm{Y_i} & u_i \\ 0 & 0 & 0 & -\mathrm{X_i} & -\mathrm{Y_i} & -1 & u_i\mathrm{X_i} & v_i\mathrm{Y_i} & v_i \end{bmatrix} x = Lx = 0 \qquad (4)$$

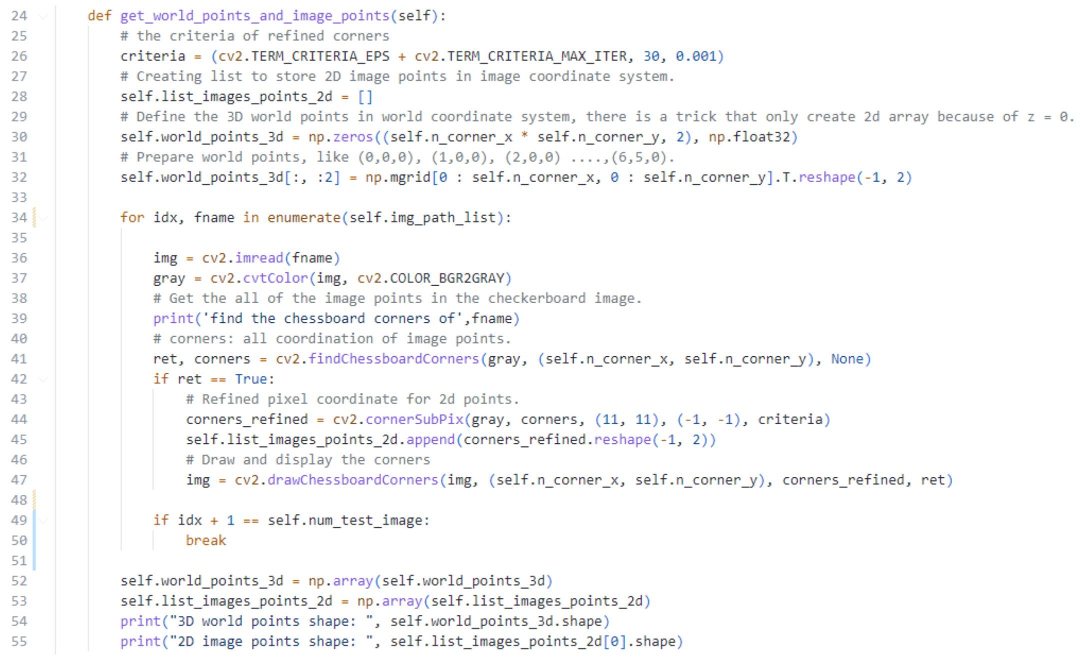
When we are given n points, we have n above equations, which can be written in matrix equation as $Lx = 0$, where L is as $2n \times 9$ matrix. Since x is defined up to a scale factor, the solution is well known to be the right singular vector of L associated with the smallest singular value (or equivalently, the eigenvector of $L^T L$ associated with the smallest eigenvalue). Simply put, we need to solve for the value of x. Based on certain properties, we can use Singular Value Decomposition (SVD) to decompose L into the following formula:

$$L = UDV^T$$

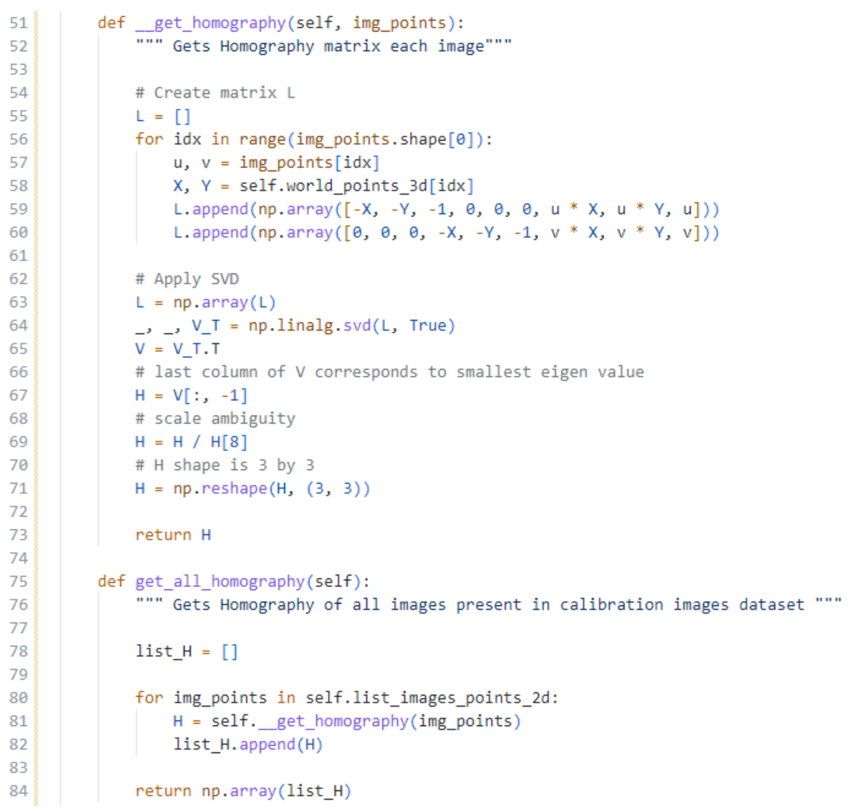
where the last column of V is x.

2.2.1. Our implemental code

First, we need to obtain the coordinates of the image points and world points. Since the Z-axis of the world points is set to 0, we only use a 2D array. After obtaining the coordinates, we can calculate the homography H. We need to construct matrix L and use SVD to derive H.



```python
def get_world_points_and_image_points(self):
    # the criteria of refined corners
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
    # Creating list to store 2D image points in image coordinate system.
    self.list_images_points_2d = []
    # Define the 3D world points in world coordinate system, there is a trick that only create 2d array because of z = 0.
    self.world_points_3d = np.zeros((self.n_corner_x * self.n_corner_y, 2), np.float32)
    # Prepare world points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0).
    self.world_points_3d[:, :2] = np.mgrid[0 : self.n_corner_x, 0 : self.n_corner_y].T.reshape(-1, 2)

    for idx, fname in enumerate(self.img_path_list):

        img = cv2.imread(fname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Get the all of the image points in the checkerboard image.
        print('find the chessboard corners of',fname)
        # corners: all coordination of image points.
        ret, corners = cv2.findChessboardCorners(gray, (self.n_corner_x, self.n_corner_y), None)
        if ret == True:
            # Refined pixel coordinate for 2d points.
            corners_refined = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
            self.list_images_points_2d.append(corners_refined.reshape(-1, 2))
            # Draw and display the corners
            img = cv2.drawChessboardCorners(img, (self.n_corner_x, self.n_corner_y), corners_refined, ret)

        if idx + 1 == self.num_test_image:
            break

    self.world_points_3d = np.array(self.world_points_3d)
    self.list_images_points_2d = np.array(self.list_images_points_2d)
    print("3D world points shape: ", self.world_points_3d.shape)
    print("2D image points shape: ", self.list_images_points_2d[0].shape)
```



```python
    def __get_homography(self, img_points):
        """ Gets Homography matrix each image"""

        # Create matrix L
        L = []
        for idx in range(img_points.shape[0]):
            u, v = img_points[idx]
            X, Y = self.world_points_3d[idx]
            L.append(np.array([-X, -Y, -1, 0, 0, 0, u * X, u * Y, u]))
            L.append(np.array([0, 0, 0, -X, -Y, -1, v * X, v * Y, v]))

        # Apply SVD
        L = np.array(L)
        _, _, V_T = np.linalg.svd(L, True)
        V = V_T.T
        # last column of V corresponds to smallest eigen value
        H = V[:, -1]
        # scale ambiguity
        H = H / H[8]
        # H shape is 3 by 3
        H = np.reshape(H, (3, 3))

        return H

    def get_all_homography(self):
        """ Gets Homography of all images present in calibration images dataset """

        list_H = []

        for img_points in self.list_images_points_2d:
            H = self.__get_homography(img_points)
            list_H.append(H)

        return np.array(list_H)
```

2.3. Solving for the Intrinsic Parameters

In the section 2.2, we estimated an homography H. Let's denote it by $H = [h_1 \quad h_2 \quad h_3]$ with h_i, the i^{th}column of H. Form (2), we have $[h_1 \quad h_2 \quad h_3] = \lambda A[r_1 \quad r_2 \quad t]$, where λ is an arbitrary scalar. Then we get

$$r_1 = K^{-1}h_1 \text{ and } r_2 = K^{-1}h_2 \ .$$

Using the knowledge that r_1 and r_2 are orthonormal, we have following two formulas by properties of orthonormality:

$$h_1^T A^{-T} A^{-1} h_2 = 0 \qquad (5)$$

$$h_1^T A^{-T} A^{-1} h_1 = h_2^T A^{-T} A^{-1} h_2 \qquad (6)$$

Let

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

Then we have

$$h_1^T B h_2 = 0$$

$$h_1^T B h_1 = h_2^T B h_2$$

Note that B is symmetric, defined by a 6D vector

$$b = [B_{11} \quad B_{12} \quad B_{22} \quad B_{13} \quad B_{23} \quad B_{33}]^T \cdot \tag{7}$$

Let the i^{th} column vector of **H** be $h_i = [h_{i1} \quad h_{i2} \quad h_{i3}]^T$. Then, we have

$$h_i^T A^{-T} A^{-1} h_j = v_{ij}^T b$$

with $v_{ij} = \left[h_{i1}h_{j1}, h_{i1}h_{j2} + h_{i2}h_{j1}, h_{i2}h_{j2}, h_{i3}h_{j1} + h_{i1}h_{j3}, h_{i3}h_{j2}, h_{i3}h_{j3}\right]^T \cdot$

Therefore, the two fundamental constraints (5) and (6) can be rewritten as two homogeneous equations in $\boldsymbol{b}$:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} b = 0\,. \tag{8}$$

If n images of the model plane are observed, by stacking n such equations as (8) we have

$$\mathbf{V}b = 0\ , \tag{9}$$

where $\mathbf{V}$ is a $2n \times 6$ matrix. Each plane (picture) gives 2 equations, B has 6 DoF, we need at least 3 planes from different views. Therefore, if $n \geq 3$, we will have in general a unique solution b defined up to a scale factor. Since solution to (9) is the eigenvector of $V^T V$ associated with the smallest eigenvalue, to solve for b, we apply SVD to decompose V:

$$V = UDC^T$$

where the last column of C is b.

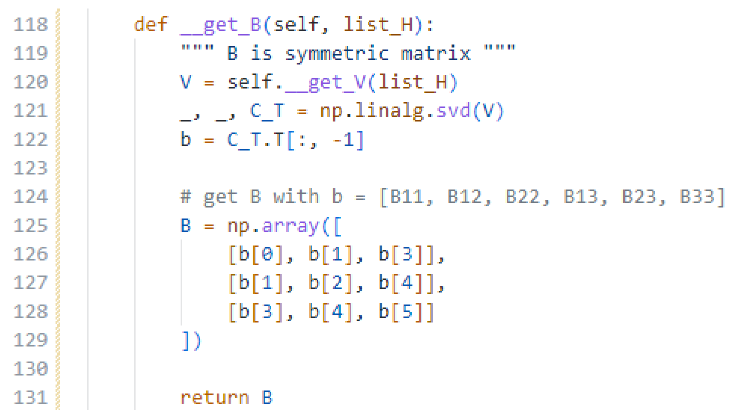
Once **b** is estimated, based on equation (7), we can obtain all the values of matrix $\boldsymbol{B}$. Then, we can compute all parameters of camera intrinsic matrix $\boldsymbol{A}$ from matrix $\boldsymbol{B}$.

$$\begin{aligned}
v_0 &= (B_{12}B_{13} - B_{11}B_{23})/(B_{11}B_{22} - B_{12}^2) \\
\lambda &= B_{33} - [B_{13}^2 + v_0(B_{12}B_{13} - B_{11}B_{23})]/B_{11} \\
\alpha &= \sqrt{\lambda / B_{11}} \\
\beta &= \sqrt{\lambda B_{11}/(B_{11}B_{22} - B_{12}^2)} \\
\gamma &= -B_{12}\alpha^2\beta/\lambda \\
u_0 &= \gamma v_0/\beta - B_{13}\alpha^2/\lambda
\end{aligned}$$

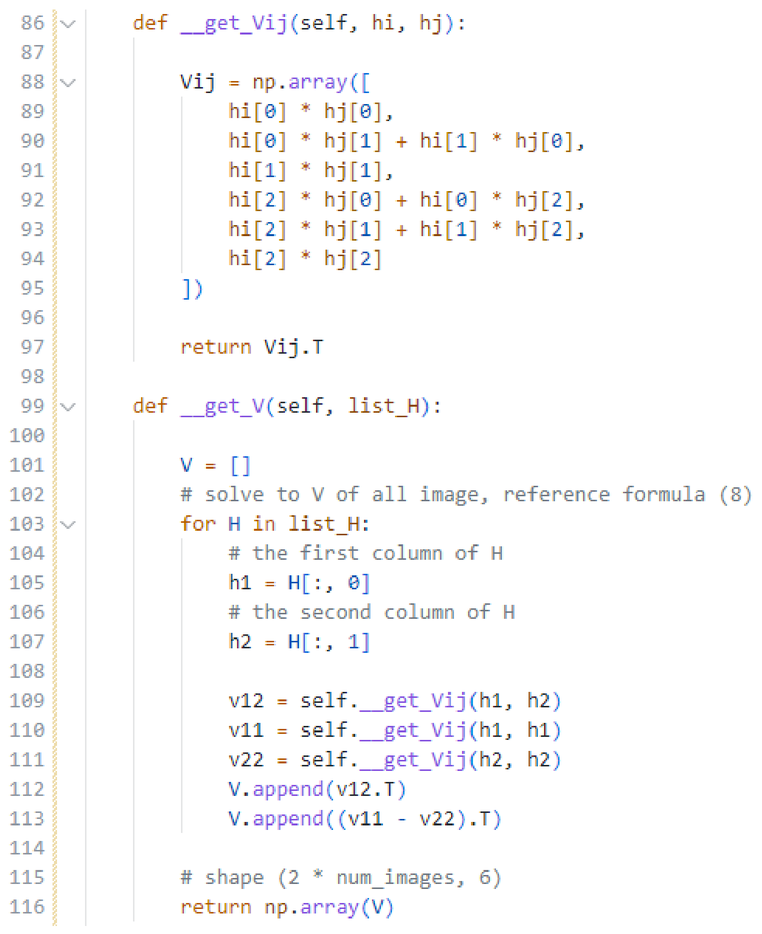
By substituting it back according to the definition (1), we can obtain intrinsic matrix $\boldsymbol{A}$.

2.3.1. Our implemental code

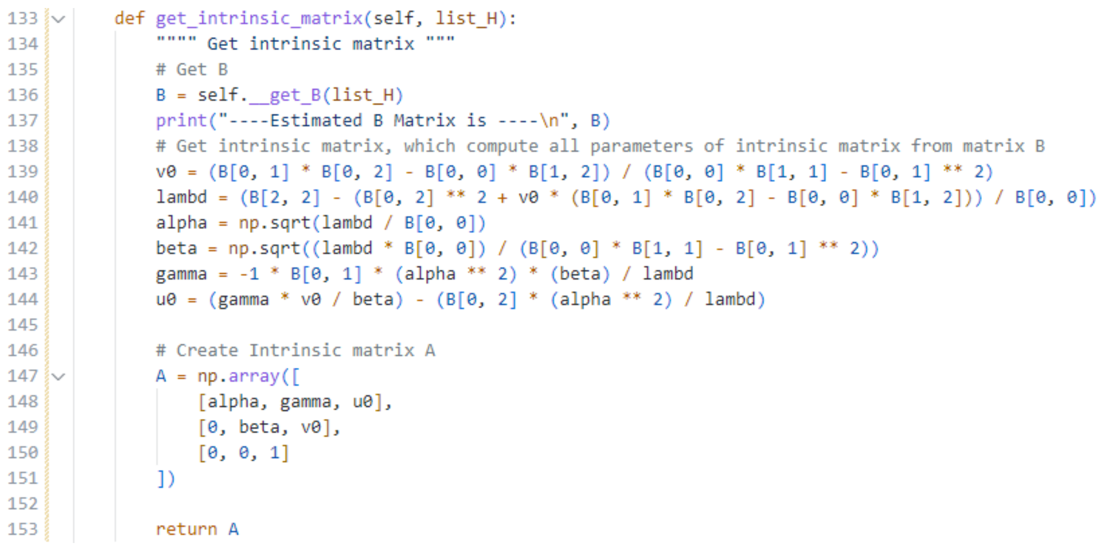
Before obtaining the intrinsic matrix, we need to sequentially compute matrices v_{ij}, V, and $\boldsymbol{B}$. The calculation method follows the derivation in section 2.3. Afterward, the values of matrix $\boldsymbol{B}$ are substituted into matrix $\boldsymbol{A}$ to solve for the intrinsic matrix.



```python
    def __get_B(self, list_H):
        """ B is symmetric matrix """
        V = self.__get_V(list_H)
        _, _, C_T = np.linalg.svd(V)
        b = C_T.T[:, -1]

        # get B with b = [B11, B12, B22, B13, B23, B33]
        B = np.array([
            [b[0], b[1], b[3]],
            [b[1], b[2], b[4]],
            [b[3], b[4], b[5]]
        ])

        return B
```



```python
    def __get_Vij(self, hi, hj):

        Vij = np.array([
            hi[0] * hj[0],
            hi[0] * hj[1] + hi[1] * hj[0],
            hi[1] * hj[1],
            hi[2] * hj[0] + hi[0] * hj[2],
            hi[2] * hj[1] + hi[1] * hj[2],
            hi[2] * hj[2]
        ])

        return Vij.T

    def __get_V(self, list_H):

        V = []
        # solve to V of all image, reference formula (8)
        for H in list_H:
            # the first column of H
            h1 = H[:, 0]
            # the second column of H
            h2 = H[:, 1]

            v12 = self.__get_Vij(h1, h2)
            v11 = self.__get_Vij(h1, h1)
            v22 = self.__get_Vij(h2, h2)
            V.append(v12.T)
            V.append((v11 - v22).T)

        # shape (2 * num_images, 6)
        return np.array(V)
```



```python
    def get_intrinsic_matrix(self, list_H):
        """" Get intrinsic matrix """
        # Get B
        B = self.__get_B(list_H)
        print("----Estimated B Matrix is ----\n", B)
        # Get intrinsic matrix, which compute all parameters of intrinsic matrix from matrix B
        v0 = (B[0, 1] * B[0, 2] - B[0, 0] * B[1, 2]) / (B[0, 0] * B[1, 1] - B[0, 1] ** 2)
        lambd = (B[2, 2] - (B[0, 2] ** 2 + v0 * (B[0, 1] * B[0, 2] - B[0, 0] * B[1, 2])) / B[0, 0])
        alpha = np.sqrt(lambd / B[0, 0])
        beta = np.sqrt((lambd * B[0, 0]) / (B[0, 0] * B[1, 1] - B[0, 1] ** 2))
        gamma = -1 * B[0, 1] * (alpha ** 2) * (beta) / lambd
        u0 = (gamma * v0 / beta) - (B[0, 2] * (alpha ** 2) / lambd)

        # Create Intrinsic matrix A
        A = np.array([
            [alpha, gamma, u0],
            [0, beta, v0],
            [0, 0, 1]
        ])

        return A
```

2.4. Solving for the Extrinsic Parameters

Because the intrinsic matrix $\boldsymbol{A}$ is fixed for pictures taken with the same camera, and each picture can compute a unique homography $\mathbf{H}$, we can use following equation to solve for the all parameters of extrinsic matrix.

$$\mathbf{H} = [h_1 \quad h_2 \quad h_3] = A[r_1 \quad r_2 \quad r_3 \quad t],$$

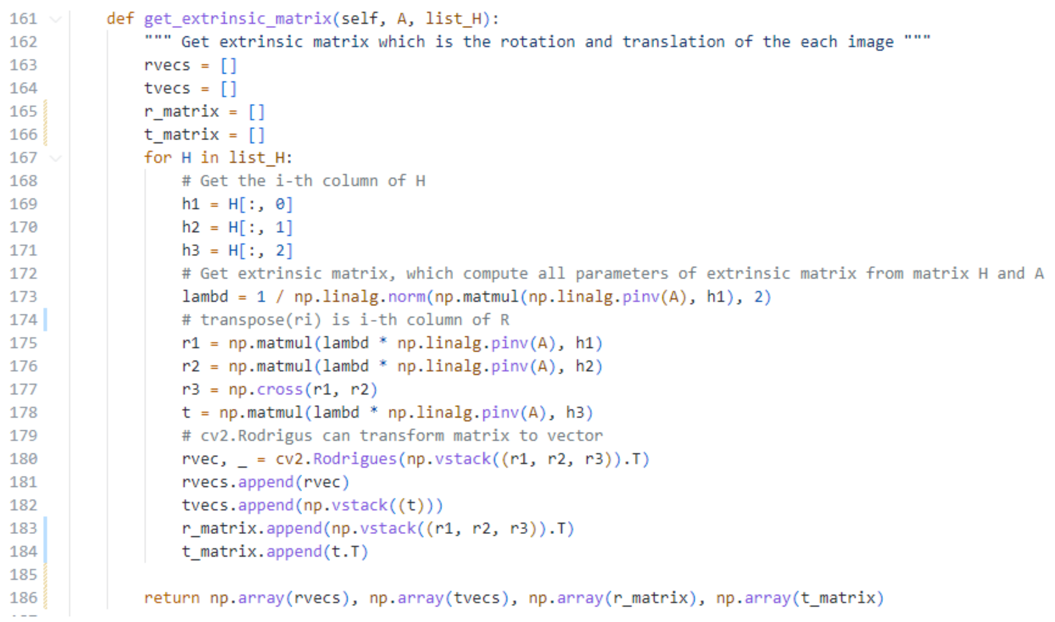
where $[r_1 \quad r_2 \quad r_3 \quad t]$ is extrinsic matrix. Finally, by using $\mathbf{H}$ and A to solve for the parameters below, we can obtain the extrinsic matrix.

$$\begin{aligned} r_1 &= \lambda A^{-1} h_1 \\ r_2 &= \lambda A^{-1} h_2 \\ r_3 &= r_1 \times r_2 \\ t &= \lambda A^{-1} h_3 \end{aligned}$$

with $\lambda = 1/\|A^{-1}h_1\| = 1/\|A^{-1}h_2\|$

2.4.1. Our implemental code

Since we already have matrix A and the homography $\mathbf{H}$, we can directly substitute them into the equations from section 2.4 to solve for the rotation matrix and translation matrix of the extrinsic matrix. It is important to note that, to align with the upcoming visualization function, the rotation matrix is specifically converted into a rotation vector.



```python
def get_extrinsic_matrix(self, A, list_H):
    """ Get extrinsic matrix which is the rotation and translation of the each image """
    rvecs = []
    tvecs = []
    r_matrix = []
    t_matrix = []
    for H in list_H:
        # Get the i-th column of H
        h1 = H[:, 0]
        h2 = H[:, 1]
        h3 = H[:, 2]
        # Get extrinsic matrix, which compute all parameters of extrinsic matrix from matrix H and A
        lambd = 1 / np.linalg.norm(np.matmul(np.linalg.pinv(A), h1), 2)
        # transpose(ri) is i-th column of R
        r1 = np.matmul(lambd * np.linalg.pinv(A), h1)
        r2 = np.matmul(lambd * np.linalg.pinv(A), h2)
        r3 = np.cross(r1, r2)
        t = np.matmul(lambd * np.linalg.pinv(A), h3)
        # cv2.Rodrigus can transform matrix to vector
        rvec, _ = cv2.Rodrigues(np.vstack((r1, r2, r3)).T)
        rvecs.append(rvec)
        tvecs.append(np.vstack((t)))
        r_matrix.append(np.vstack((r1, r2, r3)).T)
        t_matrix.append(t.T)

    return np.array(rvecs), np.array(tvecs), np.array(r_matrix), np.array(t_matrix)
```

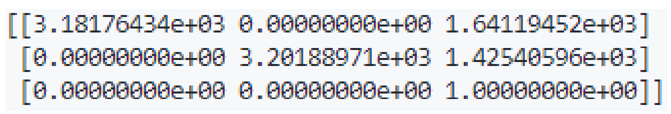
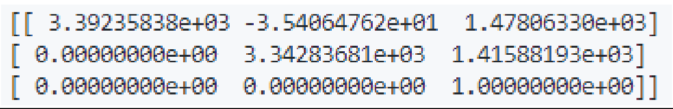
3. Experimental Result

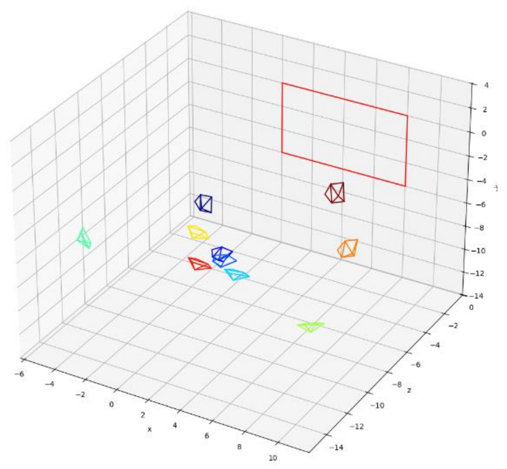
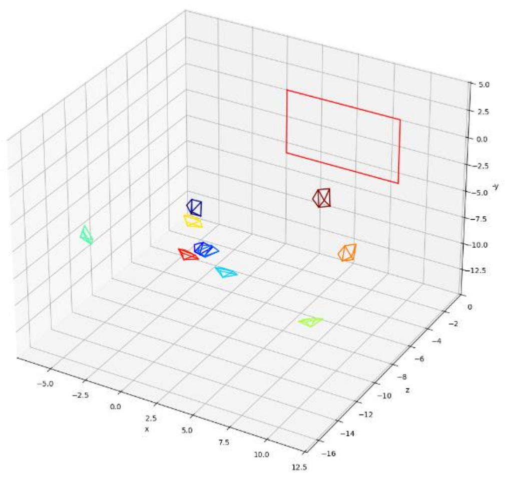
In this section, we explore the comparison between our experimental results and the OpenCV method. We'll also take a closer look at the re-projection errors and highlight the differences between our approach and OpenCV's performance.

3.1. Data provided by TA

The reason the reprojection error between our method and OpenCV's shows as below table. OpenCV is about 0.9323 and out method is about 3.2440. This is a slightly significant difference (a total difference of **3.4795%**).

Re-projection Error	
cv2.calibrateCamera()	0.9323
Our Implementation	3.2440

Intrinsic Matrix
cv2.calibrateCamera()
 `[[3.18176434e+03 0.00000000e+00 1.64119452e+03]` `[0.00000000e+00 3.20188971e+03 1.42540596e+03]` `[0.00000000e+00 0.00000000e+00 1.00000000e+00]]`
Our Implementation
 `[[ 3.39235838e+03 -3.54064762e+01  1.47806330e+03]` `[ 0.00000000e+00  3.34283681e+03  1.41588193e+03]` `[ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]`

Extrinsic Visualization	
cv2.calibrateCamera()	Our Implementation
	

Extrinsic Matrix

cv2.calibrateCamera()

```
[[-0.0976766   0.1271425   1.55337424  2.31260517 -0.44409973 10.74385315]
 [-0.26351752  0.09267554  1.06974423  0.39178744 -2.21818952 13.8985134 ]
 [-0.26717462  0.08798489  0.76061733 -1.10348038 -2.33952103 14.31627843]
 [-0.53352584  0.05243785  0.14660211 -1.11626471 -3.76228638 12.48128035]
 [-0.92747762 -0.59414438 -1.08732274 -4.85242763  4.72103921 12.98374561]
 [-0.83570666  0.26355852  0.44001177 -1.37171309 -1.3279609  16.47810794]
 [-0.23960177 -0.14565337  0.13122707 -2.39643634 -2.50619544 11.17466321]
 [-0.61574756  0.66235337  1.36546864  0.79769951  0.66209442 16.77020824]
 [-0.46417178 -0.04226902  0.02586661 -1.41987282 -2.74242464 12.54281531]
 [-0.30936529  0.71748394  1.61350838  2.98108313 -1.85178669 12.6550632 ]]
```

Our Implementation

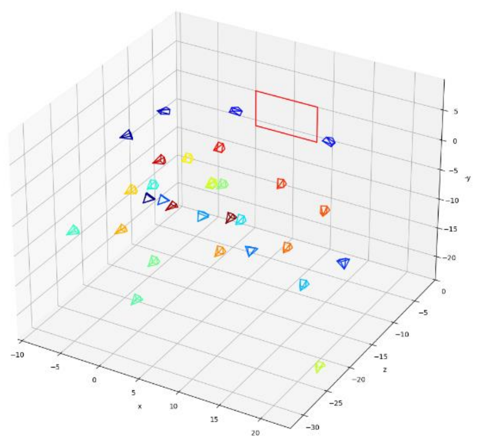
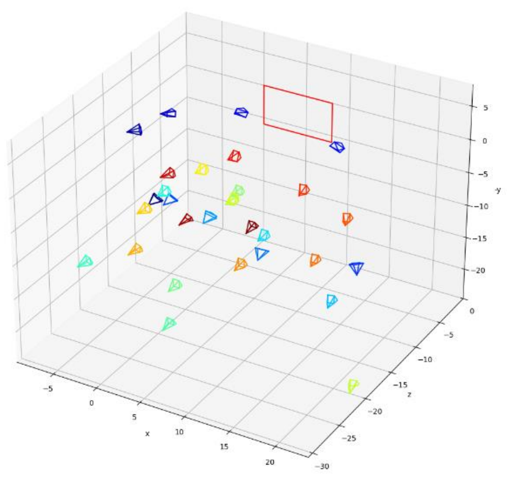
```
[[-2.06843401e-02  6.63416945e-02  1.55364238e+00  2.85907178e+00
  -4.19514236e-01  1.14258482e+01]
 [-1.69726635e-01  5.66057325e-02  1.07477843e+00  1.07725611e+00
  -2.20839408e+00  1.45754691e+01]
 [-1.67950468e-01  8.04129650e-02  7.64547222e-01 -4.01383936e-01
  -2.32603121e+00  1.50202118e+01]
 [-5.46535853e-01  1.23635371e-02  1.50136931e-01 -4.74494035e-01
  -3.85496552e+00  1.32902187e+01]
 [-9.67146934e-01 -5.57719988e-01 -1.06497070e+00 -4.08212251e+00
   4.77905309e+00  1.44443531e+01]
 [-7.96129426e-01  3.22722305e-01  4.55819419e-01 -5.72495091e-01
  -1.29165240e+00  1.73712764e+01]
 [-1.48933975e-01 -1.44020224e-01  1.28908170e-01 -1.86090967e+00
  -2.51531088e+00  1.16623252e+01]
 [-5.71952872e-01  6.97036868e-01  1.38349048e+00  1.67397847e+00
   7.36583903e-01  1.80225969e+01]
 [-4.57308854e-01 -9.82722284e-02  2.95837551e-02 -7.63906851e-01
  -2.72720313e+00  1.30612348e+01]
 [-2.11606789e-01  7.15076416e-01  1.63050853e+00  3.70387904e+00
  -1.89935845e+00  1.35579796e+01]]
```

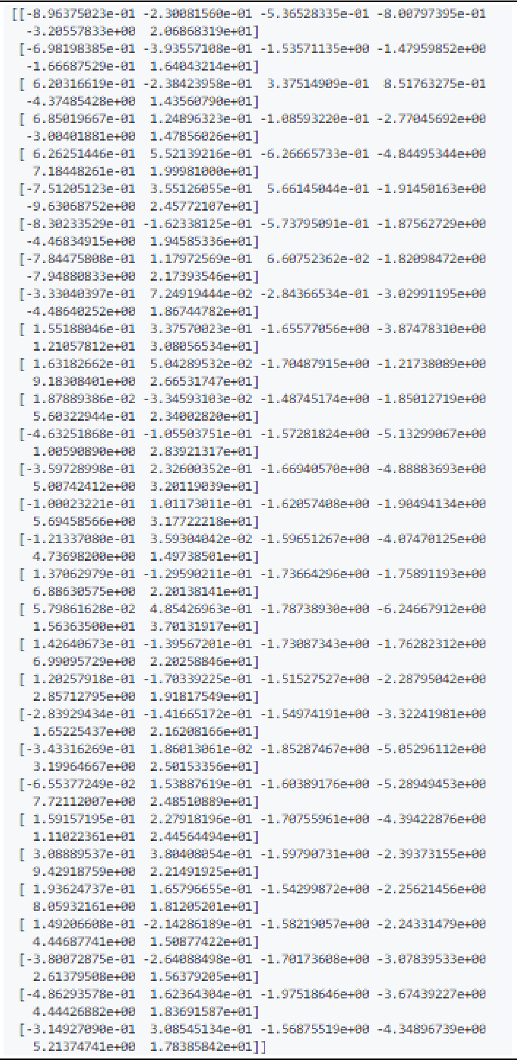
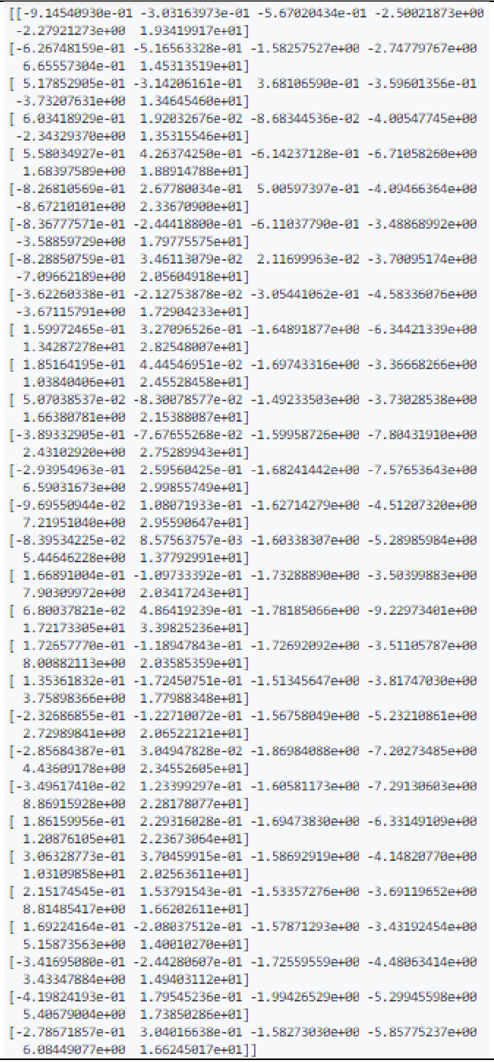
3.2. Our Data

The reason the reprojection error between our method and OpenCV’s shows the same proportional difference (a total difference of 3.5163%) compared to using TA’s data is that. From these two experimental results, we can observe a clear difference between our method and the OpenCV method, leading to a disparity of more than threefold in the outcomes. We speculate that OpenCV's method likely employs various optimization techniques to enhance performance. Otherwise, simply comparing the values of the intrinsic and extrinsic matrices shows little difference between our results and those of OpenCV.

Re-projection Error	
cv2.calibrateCamera()	0.2479
Our Implementation	0.8717

Intrinsic Matrix
cv2.calibrateCamera()
`[[1.27777129e+03 0.00000000e+00 5.92884123e+02]`
`[0.00000000e+00 1.27670970e+03 6.85338345e+02]`
`[0.00000000e+00 0.00000000e+00 1.00000000e+00]]`
Our Implementation
`[[ 1.18696603e+03 -7.07636090e+00  6.97880390e+02]`
`[ 0.00000000e+00  1.18015828e+03  6.25062482e+02]`
`[ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]`

Extrinsic Visualization	
cv2.calibrateCamera()	Our Implementation
	

Extrinsic Matrix	
cv2.calibrateCamera()	Our Implementation
	

```
[[-8.96375023e-01 -2.30081560e-01 -5.36528335e-01 -8.00797395e-01
  -3.20557833e+00  2.06868319e+01]
 [-6.98198385e-01 -3.93557108e-01 -1.53571135e+00 -1.47959852e+00
  -1.66687529e-01  1.64043214e+01]
 [ 6.20316619e-01 -2.38423958e-01  3.37514909e-01  8.51763275e-01
  -4.37485428e+00  1.43560790e+01]
 [ 6.85019667e-01  1.24896323e-01 -1.08593220e-01 -2.77045692e+00
  -3.00401881e+00  1.47856026e+01]
 [ 6.26251446e-01  5.52139216e-01 -6.26665733e-01 -4.84495344e+00
   7.18448261e-01  1.99981000e+01]
 [-7.51205123e-01  3.55126055e-01  5.66145044e-01 -1.91450163e+00
  -9.63068752e+00  2.45772107e+01]
 [-8.30233529e-01 -1.62338125e-01 -5.73795091e-01 -1.87562729e+00
  -4.46834915e+00  1.94585336e+01]
 [-7.84475808e-01  1.17972569e-01  6.60752362e-02 -1.82098472e+00
  -7.94880833e+00  2.17393546e+01]
 [-3.33040397e-01  7.24919444e-02 -2.84366534e-01 -3.02991195e+00
  -4.48640252e+00  1.86744782e+01]
 [ 1.55188046e-01  3.37570023e-01 -1.65577056e+00 -3.87478310e+00
   1.21057812e+01  3.08056534e+01]
 [ 1.63182662e-01  5.04289532e-02 -1.70487915e+00 -1.21738089e+00
   9.18308401e+00  2.66531747e+01]
 [ 1.87889386e-02 -3.34593103e-02 -1.48745174e+00 -1.85012719e+00
   5.60322944e-01  2.34002820e+01]
 [-4.63251868e-01 -1.05503751e-01 -1.57281824e+00 -5.13299067e+00
   1.00590890e+00  2.83921317e+01]
 [-3.59728998e-01  2.32600352e-01 -1.66940570e+00 -4.88883693e+00
   5.00742412e+00  3.20119039e+01]
 [-1.00023221e-01  1.01173011e-01 -1.62057408e+00 -1.90494134e+00
   5.69458566e+00  3.17722218e+01]
 [-1.21337080e-01  3.59304042e-02 -1.59651267e+00 -4.07470125e+00
   4.73698200e+00  1.49738501e+01]
 [ 1.37062979e-01 -1.29590211e-01 -1.73664296e+00 -1.75891193e+00
   6.88630575e+00  2.20138141e+01]
 [ 5.79861628e-02  4.85426963e-01 -1.78738930e+00 -6.24667912e+00
   1.56363500e+01  3.70131917e+01]
 [ 1.42640673e-01 -1.39567201e-01 -1.73087343e+00 -1.76282312e+00
   6.99095729e+00  2.20258846e+01]
 [ 1.20257918e-01 -1.70339225e-01 -1.51527527e+00 -2.28795042e+00
   2.85712795e+00  1.91817549e+01]
 [-2.83929434e-01 -1.41665172e-01 -1.54974191e+00 -3.32241981e+00
   1.65225437e+00  2.16208166e+01]
 [-3.43316269e-01  1.86013061e-02 -1.85287467e+00 -5.05296112e+00
   3.19964667e+00  2.50153356e+01]
 [-6.55377249e-02  1.53887619e-01 -1.60389176e+00 -5.28949453e+00
   7.72112007e+00  2.48510889e+01]
 [ 1.59157195e-01  2.27918196e-01 -1.70755961e+00 -4.39422876e+00
   1.11022361e+01  2.44564494e+01]
 [ 3.08889537e-01  3.80408054e-01 -1.59790731e+00 -2.39373155e+00
   9.42918759e+00  2.21491925e+01]
 [ 1.93624737e-01  1.65796655e-01 -1.54299872e+00 -2.25621456e+00
   8.05932161e+00  1.81205201e+01]
 [ 1.49206608e-01 -2.14286189e-01 -1.58219057e+00 -2.24331479e+00
   4.44687741e+00  1.50877422e+01]
 [-3.80072875e-01 -2.64088498e-01 -1.70173608e+00 -3.07839533e+00
   2.61379508e+00  1.56379205e+01]
 [-4.86293578e-01  1.62364304e-01 -1.97518646e+00 -3.67439227e+00
   4.44426882e+00  1.83691587e+01]
 [-3.14927090e-01  3.08545134e-01 -1.56875519e+00 -4.34896739e+00
   5.21374741e+00  1.78385842e+01]]
```

```
[[-9.14540930e-01 -3.03163973e-01 -5.67020434e-01 -2.50021873e+00
  -2.27921273e+00  1.93419917e+01]
 [-6.26748159e-01 -5.16563328e-01 -1.58257527e+00 -2.74779767e+00
   6.65557304e-01  1.45313519e+01]
 [ 5.17852905e-01 -3.14206161e-01  3.68106590e-01 -3.59601356e-01
  -3.73207631e+00  1.34645460e+01]
 [ 6.03418929e-01  1.92032676e-02 -8.68344536e-02 -4.00547745e+00
  -2.34329370e+00  1.35315546e+01]
 [ 5.58034927e-01  4.26374250e-01 -6.14237128e-01 -6.71058260e+00
   1.68397589e+00  1.88914788e+01]
 [-8.26810569e-01  2.67780034e-01  5.00597397e-01 -4.09466364e+00
  -8.67210101e+00  2.33670900e+01]
 [-8.36777571e-01 -2.44418800e-01 -6.11037790e-01 -3.48868992e+00
  -3.58859729e+00  1.79775575e+01]
 [-8.28850759e-01  3.46113079e-02  2.11699963e-02 -3.70095174e+00
  -7.09662189e+00  2.05604918e+01]
 [-3.62260338e-01 -2.12753878e-02 -3.05441062e-01 -4.58336076e+00
  -3.67115791e+00  1.72904233e+01]
 [ 1.59972465e-01  3.27096526e-01 -1.64891877e+00 -6.34421339e+00
   1.34287278e+01  2.82548007e+01]
 [ 1.85164195e-01  4.44546951e-02 -1.69743316e+00 -3.36668266e+00
   1.03840406e+01  2.45528458e+01]
 [ 5.07038537e-02 -8.30078577e-02 -1.49233503e+00 -3.73028538e+00
   1.66380781e+00  2.15388087e+01]
 [-3.89332905e-01 -7.67655268e-02 -1.59958726e+00 -7.80431910e+00
   2.43102920e+00  2.75289943e+01]
 [-2.93954963e-01  2.59560425e-01 -1.68241442e+00 -7.57653643e+00
   6.59031673e+00  2.99855749e+01]
 [-9.69550944e-02  1.00071933e-01 -1.62714279e+00 -4.51207320e+00
   7.21951040e+00  2.95590647e+01]
 [-8.39534225e-02  8.57563757e-03 -1.60338307e+00 -5.28985984e+00
   5.44646228e+00  1.37792991e+01]
 [ 1.66891004e-01 -1.09733392e-01 -1.73288890e+00 -3.50399883e+00
   7.90309972e+00  2.03417243e+01]
 [ 6.80037821e-02  4.86419239e-01 -1.78185066e+00 -9.22973401e+00
   1.72173305e+01  3.39825236e+01]
 [ 1.72657770e-01 -1.18947843e-01 -1.72692092e+00 -3.51105787e+00
   8.00882113e+00  2.03585359e+01]
 [ 1.35361832e-01 -1.72450751e-01 -1.51345647e+00 -3.81747030e+00
   3.75898366e+00  1.77988348e+01]
 [-2.32686855e-01 -1.22710072e-01 -1.56758049e+00 -5.23210861e+00
   2.72989841e+00  2.06522121e+01]
 [-2.85684387e-01  3.04947828e-02 -1.86984088e+00 -7.20273485e+00
   4.43609178e+00  2.34552605e+01]
 [-3.49617410e-02  1.23399297e-01 -1.60581173e+00 -7.29130603e+00
   8.86915928e+00  2.28178077e+01]
 [ 1.86159956e-01  2.29316028e-01 -1.69473830e+00 -6.33149109e+00
   1.20876105e+01  2.23673064e+01]
 [ 3.06328773e-01  3.70459915e-01 -1.58692919e+00 -4.14820770e+00
   1.03109858e+01  2.02563611e+01]
 [ 2.15174545e-01  1.53791543e-01 -1.53357276e+00 -3.69119652e+00
   8.81485417e+00  1.66202611e+01]
 [ 1.69224164e-01 -2.08037512e-01 -1.57871293e+00 -3.43192454e+00
   5.15873563e+00  1.40010270e+01]
 [-3.41695080e-01 -2.44280607e-01 -1.72559559e+00 -4.48063414e+00
   3.43347884e+00  1.49403112e+01]
 [-4.19824193e-01  1.79545236e-01 -1.99426529e+00 -5.29945598e+00
   5.40679004e+00  1.73850286e+01]
 [-2.78671857e-01  3.04016638e-01 -1.58273030e+00 -5.85775237e+00
   6.08449077e+00  1.66245017e+01]]
```

4. Discussion

4.1. Comparison of Dataset on Camera Calibration

We want to create a more complex dataset from the TA's data to make a comparison. As shown in Figure 1, Compared to the TA's dataset, our dataset includes a broader range of angles in the images and has more variations in scene complexity. We will discuss how these dataset differences impact the result of camera calibration.

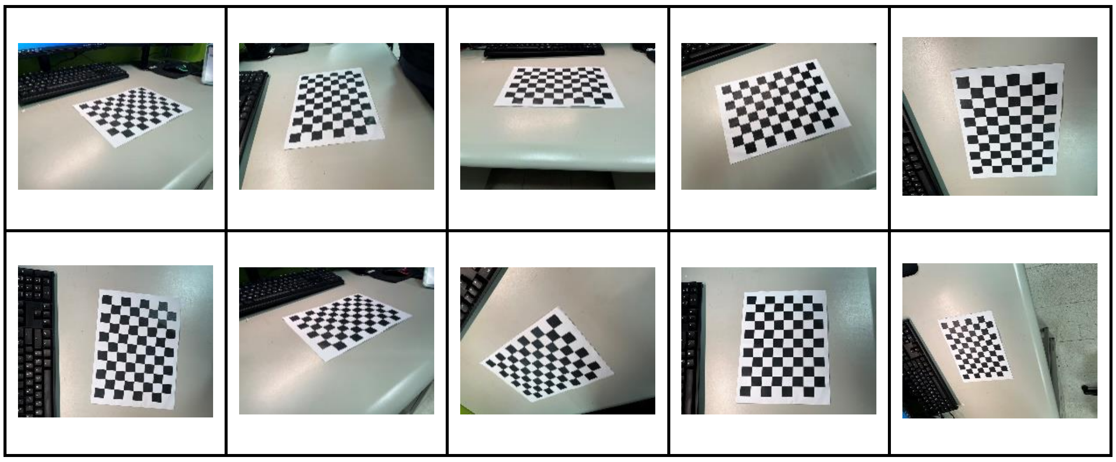


Figure 1: Our data

4.2. Skew Parameter in Intrinsic Matrix

We observe a difference between our result and cv2.calibrateCamera() in the Extrinsic Matrix. In cv2.calibrateCamera(), the skew parameter is set to zero, while our implementation calculates a non-zero skew parameter (γ). Ideally, the skew parameter should be zero or very close to it, as in a true CCD camera [1], where pixel elements are generally perpendicular. A rare non-zero skew might only occur in scenarios such as re-photographing images or enlarging negatives. This suggests that our calculated skew parameter should have been closer to zero to match real-world conditions.

4.3. Difference Re-projection Error with Considering Distortion

TA's data shows a slight difference in the re-projection error between cv2.calibrateCamera() and our implementation. Our team tried three different codes but obtained the same results. We think this is related to the Levenberg-Marquardt (LM) optimization algorithm that OpenCV uses, which minimizes the re-projection error by simultaneously optimizing all parameters across all images. Additionally, cv2.calibrateCamera() considers distortion. This may explain the observed differences.

4.4. Comparison of Results with and without Undistorting

To solve the issue mentioned in section 4.3, as shown in Figure 2, where the re-projection error increases with the number of photos, we use the LM optimization in our implementation, referencing cv2.calibrateCamera(). However, the re-projection error did not improve.

We then use the distortion parameters calculated by cv2.calibrateCamera() to cv2.undistort(). As seen in Figure 2, the results show that the undistorted re-projection error is lower than the original data, suggesting that the main issue is our omission of the distortion component.

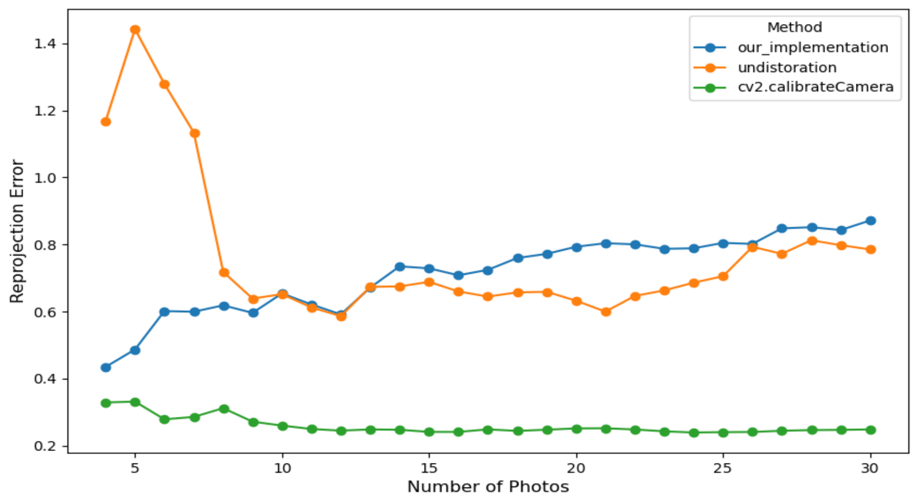


Figure 2: Reprojection error on our dataset

5. Conclusion

This assignment has some differences between our experimental results and OpenCV's. The possible reason, we believe, is that OpenCV's method is more refined, allowing it to maintain a stable reprojection error even when using varying amounts of images. On the other hand, our method neglects the distortion component, so as we gradually increase the number of images and, consequently, the amount of information, our reprojection error decreases accordingly.

This report provides a concise introduction to the camera calibration technique, followed by a detailed explanation of our implementation process. Implementing the 2D camera calibration ourselves has been an engaging challenge and a valuable learning experience, as it allowed us to see firsthand how the 3D world coordinates are transformed into 2D image coordinates.

6. Work Assignment Plan

廖怡誠	1. implement code 2. write report - the **Implement Procedure** part
李品妤	1. implement code 2. write report - the **Discussion** part
林芷萱	1. implement code 2. write report – the **Introduction** part, **Experimental Result** part and **Conclusion** part.

7. References

[1] Richard Hartley, Andrew Zisserman, Multiple View Geometry in Computer Vision, 2004

[2] Z. Zhang, "A flexible new technique for camera calibration," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 11, pp. 1330-1334, Nov. 2000, doi: 10.1109/34.888718.

8. Command

data
python camera_calibration.py -d ./data/ -x 7 -y 7 -sn 10 -en 10
my_data
python camera_calibration.py -d ./my_data/ -x 10 -y 7 -sn 30 -en 30