CV2024 HW1 Camera Calibration 📷

Group10 - 313551073 顏琦恩, 313581027 葉彥谷, 313552001 黃梓濤

I. Introduction

The task of this homework is to implement the camera calibration by using the 2D calibration chessboard method. The 2D calibration chessboard is used widely in camera calibration. Unlike the 3D calibration rig, which is not only hard to make, but also causes a high complexity calculation, we can easily produce a 2D chessboard image. With multiple views of the chessboard, we can acquire sufficient data to reconstruct the camera's 3D geometry without the need for additional 3D equipment or more complex algorithms. For our camera calibration functions, we implemented them from scratch, without using any libraries from opencv.

II. Implementation Procedure

To calculate the intrinsic matrix and extrinsic matrix of a camera, we get object points and image points from the photo of the chessboard first. After finishing the data acquisition, we can start our reconstruction of the camera's 3D geometry. The implementation is separated into three steps, which are (1) calculating the homography, (2) calculating the intrinsic matrix, and (3) calculating the extrinsic matrix. We will explain the three steps in the following section.

Step 1. Calculating the Homography

When transforming objects in the world coordinate system onto the pixel coordinate system, we usually apply the following equation:

$$x = K[R\ t]X$$

by using this equation with given x and X, we can also obtain the homography matrix of the camera. By setting the Z axis to 0, the equation takes the following form:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

then we can obtain the conditions:

$$\begin{cases} h_{11}X + h_{12}Y + h_{13} - u(h_{31}X + h_{32}Y + h_{33}) = 0 \\ h_{21}X + h_{22}Y + h_{23} - v(h_{31}X + h_{32}Y + h_{33}) = 0 \end{cases}$$

after writing the conditions into matrix format ($AH = 0$), we can get:

$$\begin{bmatrix} X & Y & 1 & 0 & 0 & 0 & -uX & -uY & -u \\ 0 & 0 & 0 & X & Y & 1 & -vX & -vY & -v \end{bmatrix} H = 0$$

using the given value, the homography matrix can be calculated through applying SVD (Singular Value Decomposition) to the A matrix. The detailed code is shown in the following code block.

```
"""Calculate H"""
def get_Ai(obj_ps, img_ps):
    A_i = []
    for obj_p, img_p in zip(obj_ps, img_ps):
        X = obj_p[0]
        Y = obj_p[1]
        u = img_p[0][0]
        v = img_p[0][1]
        A_row1 = [X, Y, 1, 0, 0, 0, -X * u, -Y * u, -u]
        A_row2 = [0, 0, 0, X, Y, 1, -X * v, -Y * v, -v]
        A_i.append(A_row1)
        A_i.append(A_row2)
    return np.array(A_i)

# calculate H
H = []
for i in range(0, len(objpoints)):
    A_i = get_Ai(objpoints[i], imgpoints[i])
    u, s, vt = np.linalg.svd(A_i)
    h_i = vt[-1, :]
    h_i = h_i / h_i[-1] # consider scale coef
    H.append(h_i.reshape((3,3)))
H = np.array(H)
```

Step 2. Calculating the Intrinsic Matrix

After finish calculating the homography matrix, we can obtain the intrinsic matrix by this equation:

$$H = K \begin{bmatrix} R & t \end{bmatrix}$$

from the equation and the feature of R — (r1, r2, r3) form an orthonormal basis, we can know that:

$$\begin{cases} {h_1}^T K^{-T} K^{-1} h_2 = 0 \\ {h_1}^T K^{-T} K^{-1} h_1 = {h_2}^T K^{-T} K^{-1} h_2 \end{cases}$$

Let $B = K^{-T}K^{-1}$ then rewrite and expand the equation , we can get two condition equations. Since B is symmetric and positive definite, we can define a new matrix $b = (b_{11}, b_{12}, b_{13}, b_{22}, b_{23}, b_{33})$, which leads to the system of $Vb = 0$. Same as calculating the homography matrix, we utilize SVD to V to obtain b. After that, we can use the equations below to attain our goal, the intrinsic matrix of the camera. The detailed code is shown in the following code block.

$$K = \begin{bmatrix} f_x & s_k & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$
\begin{aligned}
where\ o_y &= (b_{12}b_{13} - b_{11}b_{23})/(b_{11}b_{22} - {b_{12}}^2) \\
\lambda &= b_{33} - \big((b_{13}^2 + o_y(b_{12}b_{13} - b_{11}b_{23}))/b_{11}\big) \\
f_x &= \sqrt{\lambda/b_{11}} \\
f_y &= \sqrt{\lambda b_{11}/(b_{11}b_{22} - b_{12}^2)} \\
s_k &= -b_{12}f_x^2 f_y/\lambda \\
o_x &= o_y/f_y - b_{13}f_x^2/\lambda
\end{aligned}
$$

```
"""Calculate Intrinsic"""
# calculate V
def v_pq(H_i, p, q):
    v = np.array([
        H_i[0, p] * H_i[0, q],
        H_i[0, p] * H_i[1, q] + H_i[1, p] * H_i[0, q],
        H_i[1, p] * H_i[1, q],
        H_i[2, p] * H_i[0, q] + H_i[0, p] * H_i[2, q],
        H_i[2, p] * H_i[1, q] + H_i[1, p] * H_i[2, q],
        H_i[2, p] * H_i[2, q]
    ])
    return v

def get_V(H):
    V = []
    for H_i in H:
        v12 = v_pq(H_i, 0, 1)
        v11 = v_pq(H_i, 0, 0)
        v22 = v_pq(H_i, 1, 1)

        V.append(v12)
        V.append((v11 - v22))
    return np.array(V)
V = get_V(H)

# calculate b (b11, b12, b13, b22, b23, b33)
u, s, vt = np.linalg.svd(V)
b = vt[-1, :]
b11, b12, b22, b13, b23, b33 = b[0], b[1], b[2], b[3], b[4], b[5]

# calculate intrinsics
o_y = (b12 * b13 - b11 * b23) / (b11 * b22 - b12**2)
lamb = b33 - (b13**2 + o_y * (b12 * b13 - b11 * b23)) / b11
alpha = np.sqrt(lamb / b11)
beta = np.sqrt(lamb * b11 / (b11 * b22 - b12**2))
gamma = -b12 * alpha**2 * beta / lamb
o_x = gamma * o_y / beta - b13 * alpha**2 / lamb

K = np.array([[alpha,   0,   o_x],
              [0,     beta,  o_y],
              [0,     0,     1]])
```

Step 3. Calculating the Extrinsic Matrix

Now, we've already got all the needed portion for calculating the camera extrinsic matrix, just follow the equations below to finish the procedure. The detailed code is shown in the following code block.

$$\lambda = \frac{1}{||K^{-1}h_1||}$$
$$r_1 = \lambda K^{-1}h_1$$
$$r_2 = \lambda K^{-1}h_2$$
$$r_3 = r_1 \times r_2$$
$$t = \lambda K^{-1}h_3$$

```
"""Calculate Extrinsics"""
# calculate Extrinsic (R t)
extrinsics = []
for H_i in H:
    h1 = H_i[:, 0]
    h2 = H_i[:, 1]
    h3 = H_i[:, 2]
    K_inv = np.linalg.inv(K)

    lambda_ = 1 / np.linalg.norm(np.dot(K_inv, h1))
    r1 = lambda_ * np.dot(K_inv, h1)
    r2 = lambda_ * np.dot(K_inv, h2)
    r3 = np.cross(r1, r2)
    R = np.column_stack((r1, r2, r3))
    t = lambda_ * np.dot(K_inv, h3)

    extrinsics.append(np.column_stack((R, t)))
extrinsics = np.array(extrinsics)
```

III. Experimental Results

We conducted the experiment with two different data setups: the first used chessboard images provided by the TA, and the second used pictures we captured ourselves in the real world. The former contained 10 images, while the latter contained 14 images.

First Setup

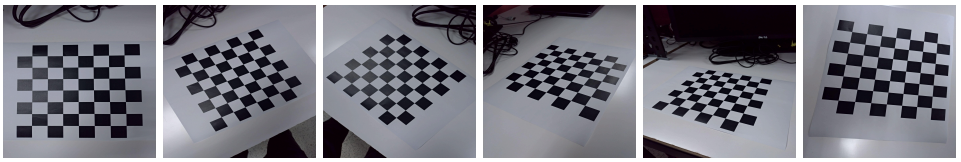

Fig.1 Sample data provided by the TA.

We compared our results with the results from the opencv function using Reprojection Error (Appendix I). The results can be seen in Table 1. We can observe that our result is not

as accurate as OpenCV's. The error may be caused by the optimization method used in the calibration process. The camera pose can be visualized in Fig.2.

Table 1. Comparison of results using data provided by the TA

	Reprojection Error
OpenCV	0.93
Ours	24.32

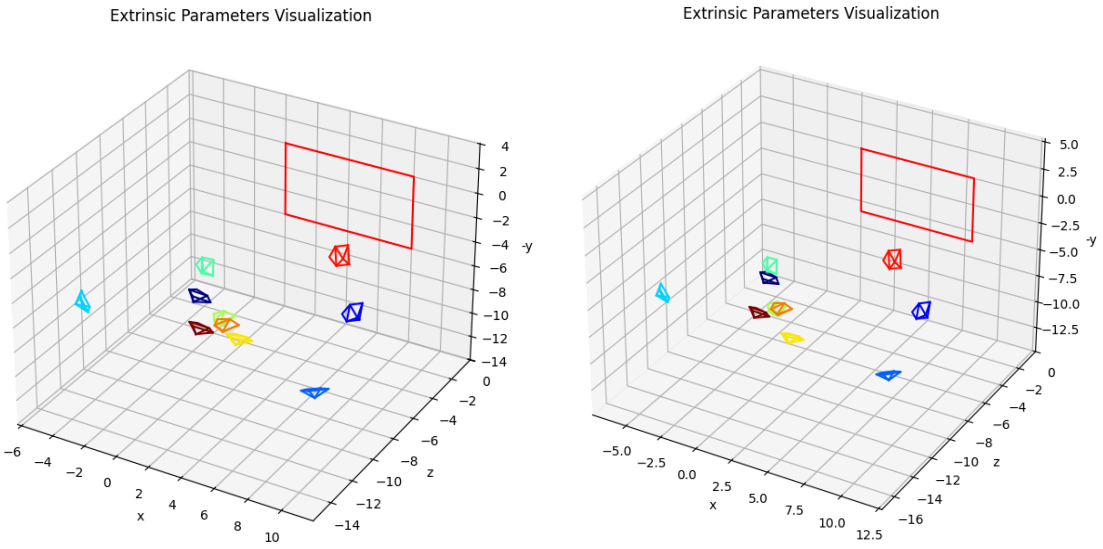

Fig.2 Visualization of the camera pose using different method: (left) OpenCV, (right) Ours

Second Setup : Real-World Setting

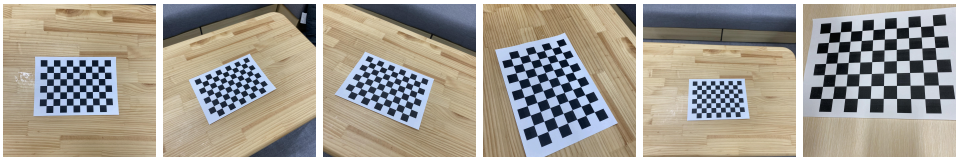

Fig.3 Sample data captured in the real-world.

We also compared the results estimated from our captured images. The experimental setup is the same as in the first case, and the results are shown in Table 2 and Fig.4.

Table 2. Comparison of results using our data

	Reprojection Error
OpenCV	0.10
Ours	4.32

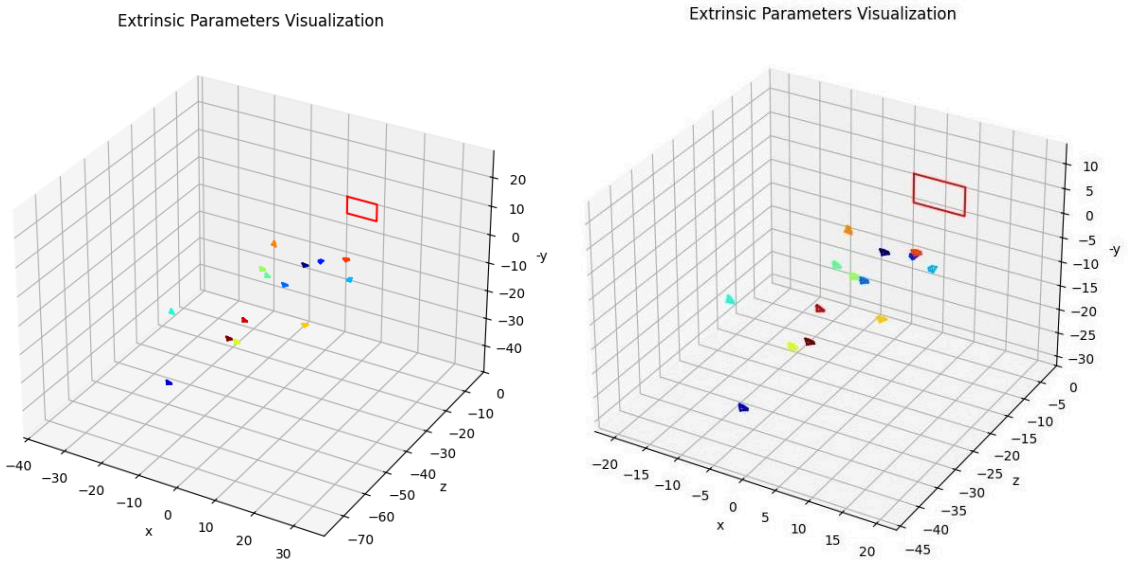

Fig.4 Visualization of the camera pose using different method: (left) OpenCV, (right) Ours

IV. Discussion

1. Impact of Photo Quantity

Comparing the error scores between our data and the data provided by the TA, we observe that as the number of photos increases, the reprojection error decreases. A larger sample improves camera parameter estimation, reduces noise, and provides more perspectives, enhancing the model's accuracy and robustness.

2. Impact of Shooting Angle

The shooting angle significantly affects the reprojection error, particularly the x and y axis rotations that cause the image plane to not be parallel with the chessboard. These angles reduce the accuracy of feature point detection and matching, making it harder to precisely analyze the camera angles and increasing the reprojection error.

Fig.5 illustrates that the left photo, taken from nearly overhead, has a lower reprojection error, while the right photo, captured from a side angle, shows a higher reprojection error. This highlights the importance of optimal shooting angles for accurate camera calibration.

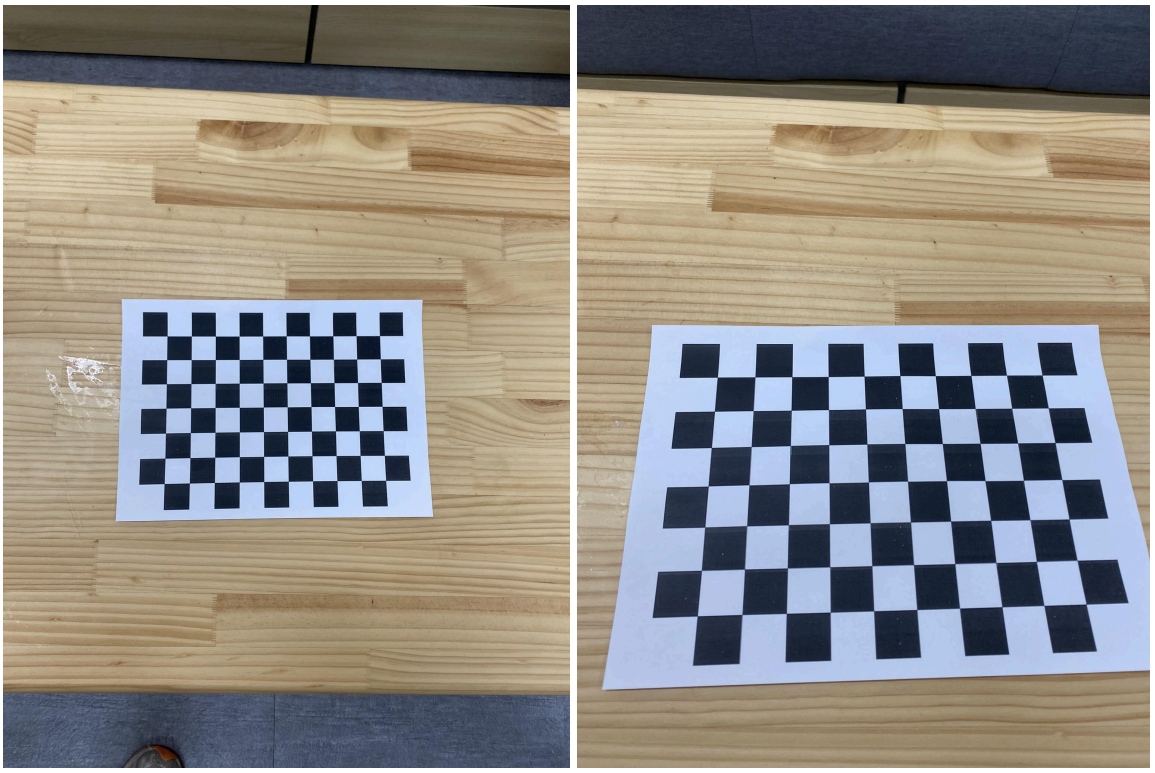

Fig.5 The lowest reprojection error (left, error=3.7) and the highest reprojection error (right, error=5.66)

V. Conclusion

In conclusion, we've accomplished three tasks in this homework, which is (1) implementing the camera calibration procedure, (2) experiments on real-world datasets, and (3) evaluating our method.

VI. Work Assignment Plan

- ❖ 葉彥谷 : Programming and report.
- ❖ 顏琦恩 : Debug and report.
- ❖ 黃梓濤 : Experiments, validation and report.

Appendix I

```python
def calculate_reprojection_error(objpoints, imgpoints, mtx, dist, extrinsics):
    print('Calculating reprojection error...')
    total_error = 0
    total_points = 0

    for i in range(len(objpoints)):
        # Extract rotation and translation vectors from extrinsics
        rvec, tvec = extrinsics[i][:, :3], extrinsics[i][:, 3]

        # Project points using the camera calibration parameters and extrinsics
        projected_image_points, _ = cv2.projectPoints(
            objpoints[i], rvec, tvec, mtx, dist)

        # Compute the error between the detected points and projected points
        error = cv2.norm(imgpoints[i], projected_image_points,
                         cv2.NORM_L2) / len(projected_image_points)
        print("Image: ", i, "Error: ", error)
        total_error += error
        total_points += len(objpoints[i])

    # Compute average reprojection error
    mean_error = total_error / len(objpoints)
    return mean_error
```

The function **calculate_reprojection_error** is designed to evaluate the reprojection error which quantifies the difference between the detected 2D image points and the projected points. Using the **projectPoints** function of OpenCV, the 3D object points are projected into the image plane based on the current extrinsics and the camera parameters (mtx and dist) and calculate the error between the detected 2D image points and the projected points.