ENPM673 Project 3

Brendan Neal

April 12, 2023

Contents

1 Introduction

This project has students calculating a camera's intrinsic matrix from a set of given world and image points using only the numpy library. Additionally, this project has students performing camera calibration and calculating re-projection error using any of Python3's in-built functions.

2 Problem 1

This problem had me calculating a camera's intrinsic matrix from a set of given world and image points using only the numpy library.

2.1 Minimum Matching Points

The minimum number of point correspondences to calculate the projection matrix P (and thus all other required camera calibration parameters) is 6.

2.2 Pipeline

2.2.1 General Camera Calibration Pipeline

Figure 1 shows the flowchart for calibrating a camera. In Question 1, steps one through five are already completed for us. See the next section for specifics on how steps six through nine are performed.

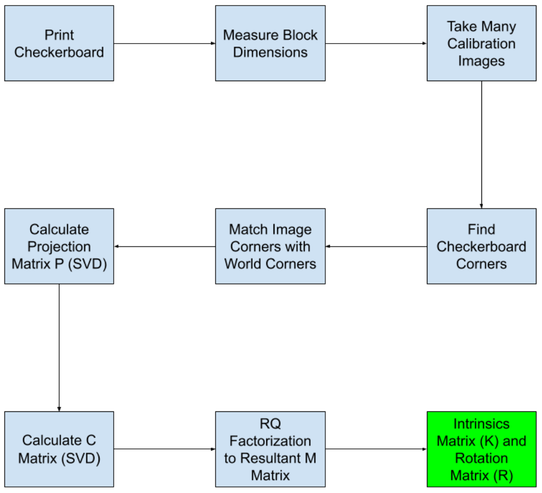
Print Checkerboard
Measure Block Dimensions
Take Many Calibration Images
Find Checkerboard Corners
Match Image Corners with World Corners
Calculate Projection Matrix P (SVD)
Calculate C Matrix (SVD)
RQ Factorization to Resultant M Matrix
Intrinsics Matrix (K) and Rotation Matrix (R)

Figure 1: Camera Calibration Flowchart

2.2.2 Mathematical Formulation and Operation Pipeline

The first step to solving for the camera intrinsic matrix is to solve for the projection matrix. First, I had to set up a system of equations to solve for my projection matrix. Since

$$x_i = PX_i$$

and

$$x_i \times PX_i = 0$$

I can set up an equation to solve for P. Using the relationship above, I form the following matrix (A) below, where u and v represent the image x and y coordinates, X Y Z represent the world coordinates and the matrix is stacked from n=1 to n=8 (forming a 16x12 matrix overall):

$$\begin{bmatrix} X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_n * X_n & -u_n * Y_n & -u_1 * Z_n & -u_n \\ 0 & 0 & 0 & 0 & X_1 & Y_n & Z_n & 1 & -v_n * X_n & -v_n * Y_n & -v_n * Z_n & -v_n \\ .. & .. & .. & .. & .. & .. & .. & .. & .. & .. & .. & .. \end{bmatrix}$$

Please note that this is simply the matrix from the class notes with mathematical operations performed to it in order to change its form. The matrix was also garnered from Arunava's office hours.

Solving the equation AP = 0 where P is 12x1 column matrix will yield the projection matrix for this particular image. To solve this equation, I used eigenvalue decomposition and partitioned the eigenvector of v with the smallest eigenvalue, which is the projection matrix P. I then reshaped this matrix to be 3x4 and normalized it based on the bottom-left element (divide every other element by the bottom left element). The numerical value for my projection matrix can be found in the Results section.

Now that I have my projection matrix, I can begin decomposing my projection matrix into my translation matrix C, my rotation matrix R, and finally my intrinsic matrix K. The translation matrix C is the null vector of P. To solve for C, I perform eigenvalue decomposition on P to get the eigenvector of v with the smallest eigenvalue, which is the translation matrix C. C is a 4x1 matrix and I then normalize it by dividing every element by the bottom-most element. The numerical value for my translation matrix can be found in the Results section.

Now that I have C, I can classify the left 3x3 matrix of P to be matrix M. To solve for R, I have to perform RQ factorization manually on the M matrix since numpy only has an in-build function for QR factorization. I perform RQ factorization via the Gram-Schmidt Process. A1, A2, and A3 are the rows of the M matrix. The steps below outline the process:

$$U_1 = A_1$$

$$E_1 = \frac{U_1}{norm(U_1)}$$
$$U_2 = A_2 - proj_{U_1} A_2$$
$$E_2 = \frac{U_2}{norm(U_2)}$$
$$U_3 = A_3 - proj_{U_1} A_3 - proj_{U_2} A_3$$
$$E_3 = \frac{U_3}{norm(U_3)}$$

The expanded formula for projecting one vector onto another is:

$$(\frac{U_1 dot A_2}{U_1 dot U_1}) * U_1$$

This yields R to be:

$$[E_1 E_2 E_3]$$

The numerical value of the R matrix can be found in the Results section. Finally, with M and R, I can solve for the intrinsic matrix K. Since

$$M = KR$$

we can solve for K via:

$$K = MR^{-1}$$

After performing that operation I noticed that my output was a lower triangular matrix rather than an upper triangular matrix. So to get the actual K I took the transpose of the result and normalized it to the bottom-right element to get the camera intrinsic matrix. The numerical result can be found in the Results section.

I can compute the reprojection error for each individual point by using the projection matrix. To do so, I take the dot product between the projection matrix and an individual homogeneous world point to project the world coordinate into the image frame. Next, I normalize the reprojected image coordinate and keep only the x and y components. Then, I calculate the distance between the reprojected point and the known image coordinate. That distance is equal to the reprojection error. I repeat this process for each point and append all the reprojection point errors into an array. Finally, I take an average of all the reprojected point errors to get an error for the whole image.

2.3 Results

2.3.1 Projection Matrix

The estimated projection matrix P is:

$$\begin{bmatrix} 28.73 & -1.757 & -70.07 & 756.89 \\ -20.14 & 65.89 & -22.21 & 213.26 \\ -0.028 & -0.0026 & -0.0314 & 1 \end{bmatrix}$$

2.3.2 Rotation Matrix

The estimated rotation matrix R is:

$$\begin{bmatrix} 0.379 & -0.023 & -0.925 \\ -0.342 & 0.925 & -0.163 \\ 0.860 & -0.378 & -0.343 \end{bmatrix}$$

2.3.3 Translation Matrix

The estimated translation matrix C is:

$$\begin{bmatrix} 15.96 \\ 7.43 \\ 17.16 \\ 1 \end{bmatrix}$$

2.3.4 Intrinsic Matrix

The estimated intrinsic matrix K is:

$$\begin{bmatrix} 2130 & 320 & 0.522 \\ 0 & 2010 & 0.343 \\ 0 & 0 & 1 \end{bmatrix}$$

2.3.5 Reprojection Error for Each Point

Table 1 shows the reprojection error for each world point as well as the average reprojection error for all eight points.

Reprojection Error Results	
World Point	Error (Pixels)
(0 0 0)	0.15431612130217331
(0 3 0)	1.327136555040056
(0 7 0)	1.3365047318540064
(0 11 0)	0.15027370990674171
(7 1 0)	0.18693373537240632
(0 11 7)	0.3229231073336791
(7 9 0)	0.19188728856454418
(0 1 7)	0.31208098430963105
8 Point Average	0.49775702921040477

2.4 Problems Encountered

Overall this problem was very challenging because the notes were hard to follow and a bit unclear. I had to seek a lot of help from the TA staff and they introduced techniques that were not discussed in class.

3 Problem 2

This problem had me performing camera calibration and calculating re projection error using any of Python3's in-built functions.

3.1 Pipeline

My camera calibration pipeline was adapted from OpenCV's official camera calibration example. I combined the pipeline we learned in class with the specifics for getting the functions to work on their official website and I feel like I should cite their documentation. Here is the link to the example on their website: `https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html`

The first step to solving this problem was to set all the necessary parameters for OpenCV's calibrateCamera() function. The first step was to set the termination criteria for the corner search. I just used the default settings found on OpenCV's website since our project has no special considerations. Next, I initialized my object points array as a numpy 54 by 3 float32 array. The reason for 54 is because our calibration target is of dimension 10 by 7. Because it is good practice to ignore the outermost corners, I dialed back the dimensions to

9 by 6, thus producing the 54 length array for object points. Next, I set this object points array to a mesh-grid and reshaped it. Afterward, I initialized 2 more non-numpy arrays to store the found object and image points.

Now my corner finding algorithm begins. The following steps are performed for each of the calibration images. I read in an image, then convert the image to gray. Next, I use the cv2.findChessboardCorners() function with the gray image and the 9 by 6 pattern size as inputs. Next, If the corners are successfully found, I append the found object points to the object points array. Next, I refine the found corners using the cv2.cornerSubPix() function and append the refined image corners to the image points array. For visualization, I then plot the refined corners over the original images.

Once I have a collection of corresponding object and image corner point arrays, I then use the cv2.calibrateCamera() function with the two arrays as inputs in order to output the intrinsic matrix (K), the distortion coefficients, the rotation vectors, and the translation vectors as outputs.

Finally, to calculate the reprojection error, I had to loop through each of the images once again and use the cv2.projectPoints() function to calculate the re projection points for each image. I then calculated the average reprojection error for each image using the cv2.norm() function. Finally, I displayed each image with each of the reprojected points overlaid.

3.2 Results

3.2.1 Corner Detection

Figure 2 is the first calibration image after undergoing the corner detection algorithm.

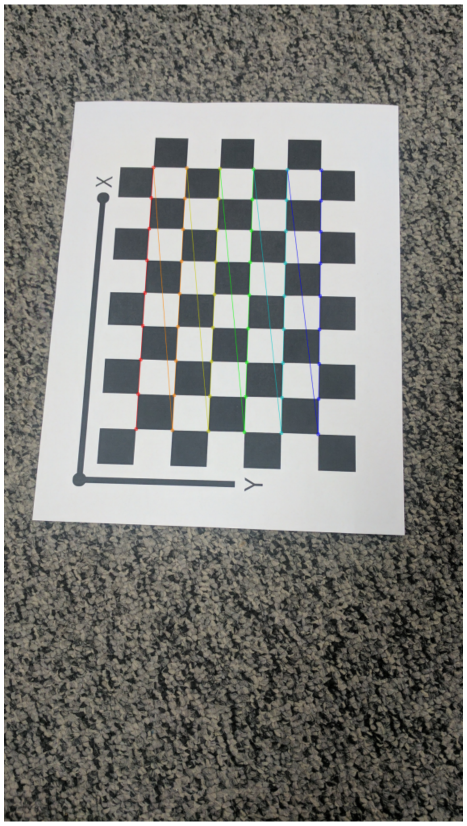

Figure 2: Found Corners

3.2.2 Reprojection Error

Figure 3 is the first calibration image after plotting the reprojected points on top of the image in red.

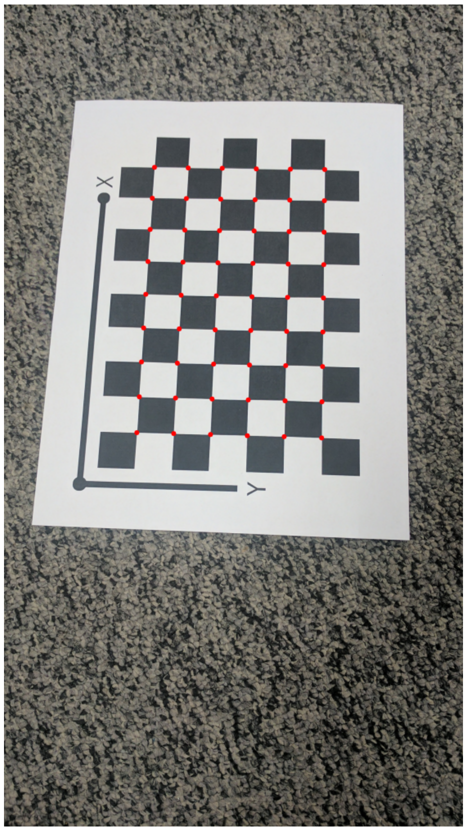

Figure 3: Reprojected Points

Table 2 shows the reprojection error for each calibration image as well as the average reprojection error for all thirteen images.

Reprojection Error Results	
Image	Error (Pixels)
1	0.07290379963579774
2	0.0891622825506153
3	0.1159750678564013
4	0.1381726699574623
5	0.0623548647524438
6	0.07547120291245601
7	0.03201701753001409
8	0.05727789035017854
9	0.0712065630717738
10	0.0748890154196616
11	0.10805795834398212
12	0.122906389094966
13	0.11822716757172298
13 Image Average	0.08758629915749812

3.2.3 Camera Intrinsic Matrix

The matrix below is the found camera intrinsic matrix after undergoing the camera calibration algorithm:

$$\begin{bmatrix} 2042.73 & 0 & 764.36 \\ 0 & 2035.02 & 1359.0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.3 Methods to Improve Accuracy

One method to improve the accuracy of the K matrix is to take more calibration images. In addition to taking more images, we could improve the accuracy by covering many more orientation and depth options. Furthermore, we have to make sure to cover the whole field of view in our region of interest. Finally, you could take into account radial distortion to further optimize your camera intrinsic matrix.

3.4 Problems Encountered

No major problems encountered in this section. The biggest challenges came from understanding the documentation for the required functions as well as developing the visualization for the re projected points.