MLMI12 Computer Vision Coursework 3

J902K

3 Chessboard Detection with Harris Corner Detection

We pivoted to using Harris corner detection instead, another intuitive choice for detecting corners in chessboards. With quite a few steps, this worked well enough. We will discuss each step of this algorithm on a high level in each of the following subsections. Interested readers can find commented and runnable code with visualizations at the GitHub repository `https://github.com/GavinPHR/calibration-pattern-detection`.

3.1 Pre-requisites

We will use a concrete calibration image that was supplied to us in coursework 1 as an example throughout the walk-through, as shown in Figure 8.

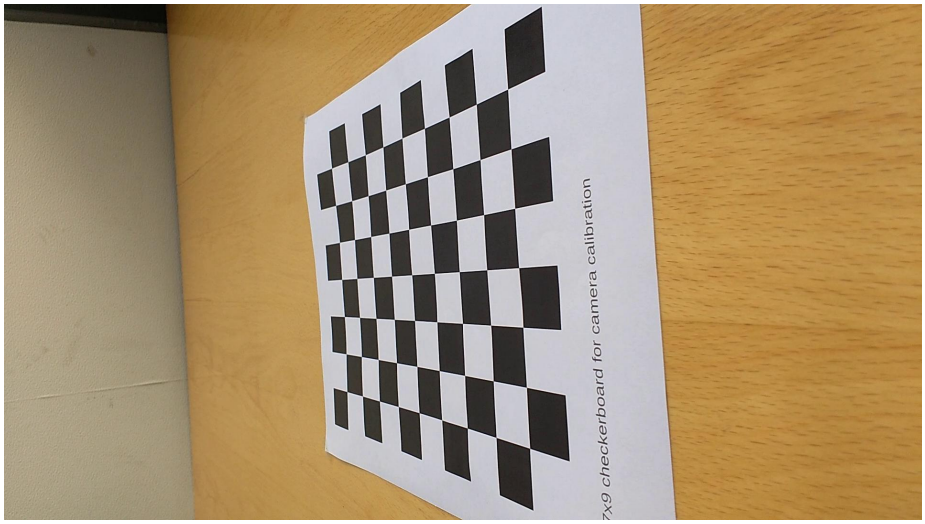


Figure 8: Example calibration image (`calibration-23.jpg`)

It is a 9-by-7 chessboard, with 63 internal corners and 34 external corners. The goal is to find and order the 63 internal corners. We will refer to the 63 internal corners as **target corners**, the 97 internal + external corners as **square corners**, and everything else as **irrelevant corners**.

3.2 Corner Detection

GitHub link to relevant function(s).

The first step is to detect the corners using Harris corner detector. In the presence of noise in an image, we can not expect the detector to return only the target corners exactly. The parameters are tuned so that the detector returns many more corners. It is always better to have more corners and try filtering them down than to have fewer corners and fail directly.

The detected corners are shown in Figure 9. The detector returned 2081 corners for the example image, which include all the target corners we are interested in. We will filter them down in subsequent subsections.

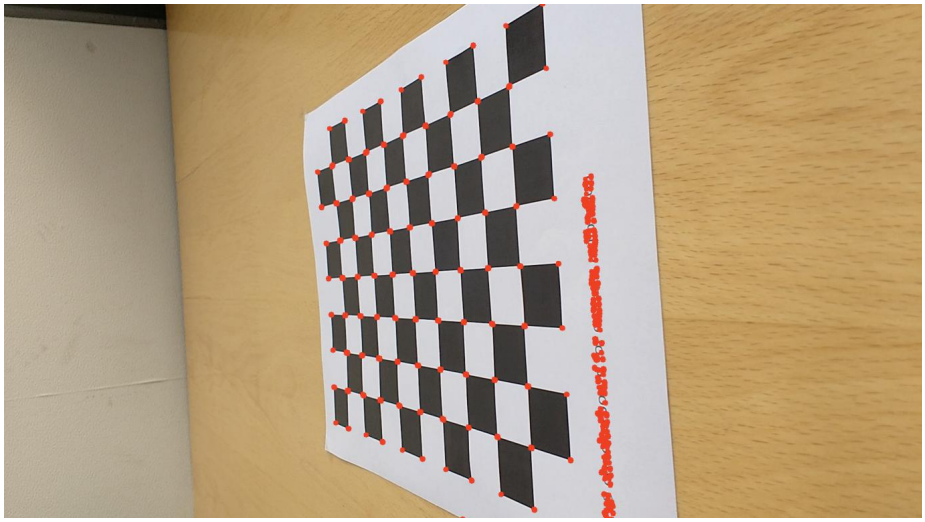


Figure 9: 2081 corners (represented by red dots) detected using Harris corner detector

3.3 Corner Filtering 1: Clustering

GitHub link to relevant function(s).

An observation we made was that many detected corners are very close (<10px in a 1080p image) to each other. This is why in Figure 9, it does not seem like there are 2081 corners detected.

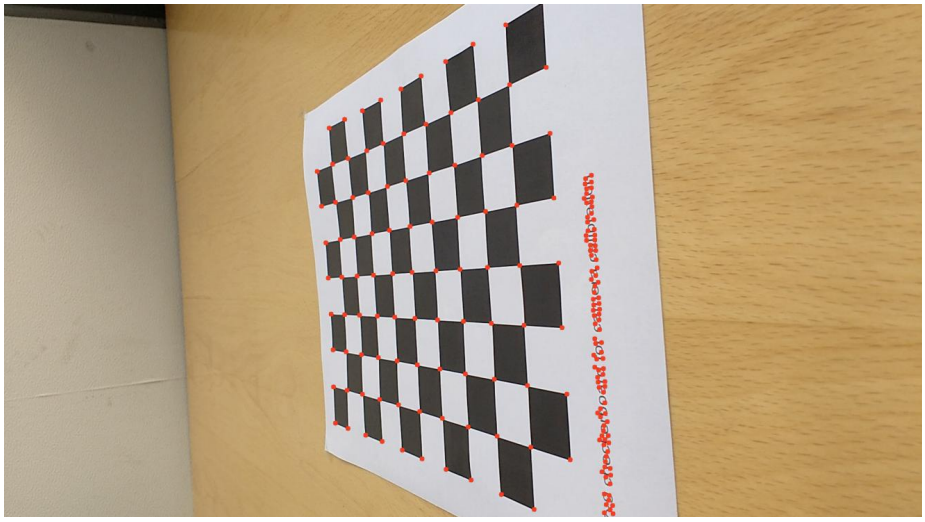


Figure 10: Corners after cluster-based filtering, an reduction from 2081 to 194 corners

We can cluster the corners and use the cluster locations as the "new" corners, this significantly reduces the number of corners. For the example image, we are left with 194/2081 corners as shown in Figure 10.

3.4 Corner Filtering 2: Connected Components

GitHub link to relevant function(s).

After cluster-based filtering, the square corners become distinct (i.e. there are no two points that are close to each other that represent the same corner). We can use this fact to filter further.

Let us build a graph G where two corners c_i, c_j are connected if the distance between them is less than a threshold ϵ and not connected otherwise. If we represent G using an adjacency matrix, then:

$$G[i,j] = \begin{cases} 1 & \text{if } |c_i - c_j| < \epsilon \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

If a corner is one of the square corners, we should expect it to be in a connected component of size 1 (i.e. the component contains only that corner). By filtering out all the corners in components of size >1, we are left with 99/194 corners, as shown in Figure 11.

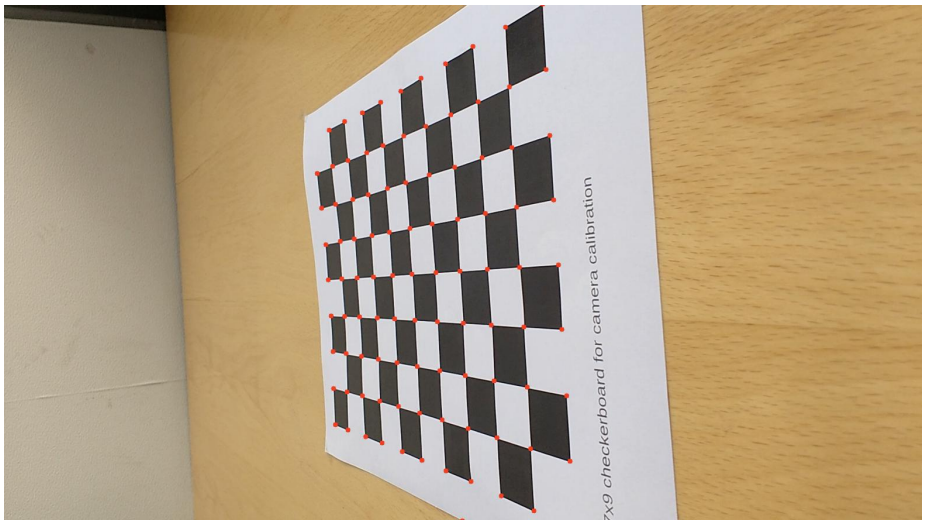
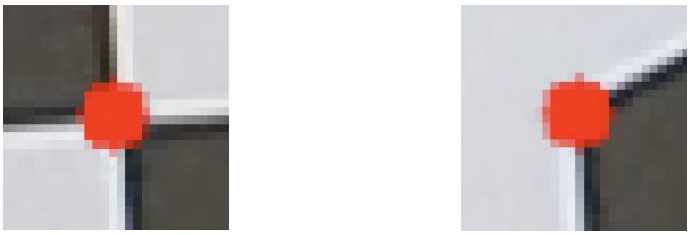


Figure 11: Corners after connected component filtering, an reduction from 194 to 99 corners

3.5 Corner Filtering 3: Square Response

GitHub link to relevant function(s).

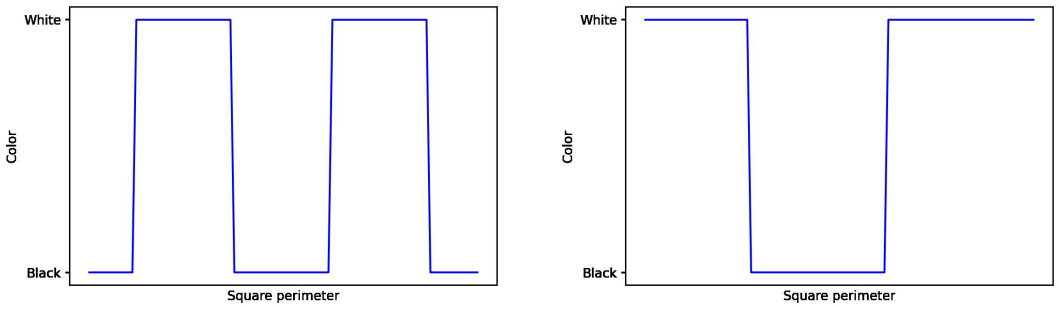
We are very close to our goal after the previous filtering step, being left with 97 square corners and 2 irrelevant corners. To find only the target corners, we can look at each corner's surrounding area.



(a) (b)

Figure 12: (a) Surrounding area of an internal corner (b) Surrounding area of an external corner

If we zoom in at each corner and centre a square on it, the surrounding area would look different for different corner types. For an internal corner (Figure 12(a)), its surrounding area consists of 4 patches: black → white → black → white. While for an external corner (Figure 12(b)), its surrounding area consists of 2 patches: black and white. If we trace out the perimeters of the squares in Figure 12(a) and (b), we would get very different response graphs.



(a) (b)

Figure 13: (a) Internal corner square response (b) External corner square response

If we count distinct continuous black/white segments, we should expect 4 to 5 segments in an internal corner square response as shown in Figure 13(a). If

there are more or fewer segments, the response most likely did not come from an internal corner and that corner is eliminated.

After this step, we are left with 63/99 corners for the example image as shown in Figure 14.

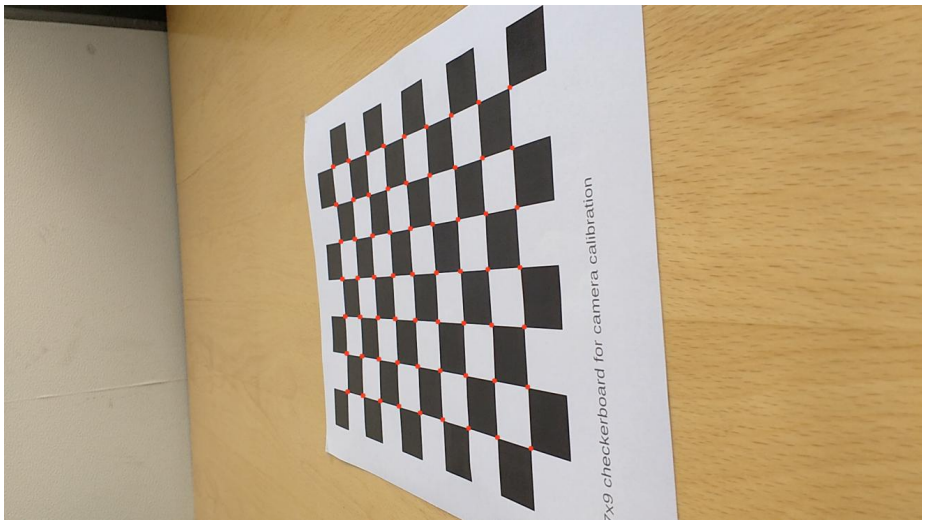


Figure 14: Corners after square response filtering, an reduction from 99 to 63 corners

3.6 Corner Filtering 4: Sum Distance

GitHub link to relevant function(s).

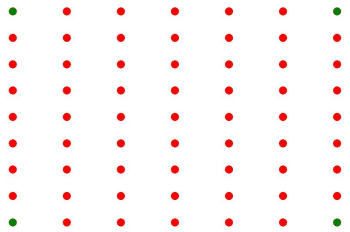
An additional optional filtering step is applied if, after the previous step, we still have more than the 63 target corners. We first calculate pairwise distance between all the corners, then filter out the corners with the largest sum of distances to other corners. This way, we can eliminate corners that are far away from the chessboard.

4 Corner Ordering

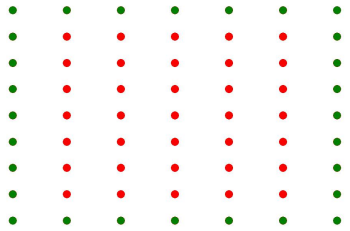
Using the corner detection algorithm described in Section 3, we should have obtained all the internal corners. But it remains an issue to establish their ordering.

Our approach is to:

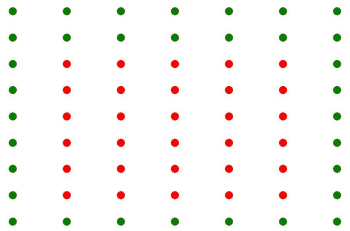
1. Find the 4 vertex corners.



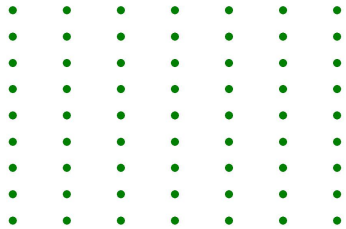
2. Find all the corners on the perimeter.



3. Iteratively find corners on the same horizontal lines.



4. Until all the corners are found.



We will expand each of them in subsequent subsections.

4.1 Finding Vertex Corners

First retrieve the convex hull on the corners, if the corners are aligned perfectly, the convex hull should have vertices that correspond to the 4 vertex corners. But due to distortion and noise, this is rarely the case. We can approximate the convex hull with a quadrilateral, and the vertices of the quadrilateral are the corners we are looking for. See details in code, the approximate quadrilateral for the example image is shown in Figure 15.

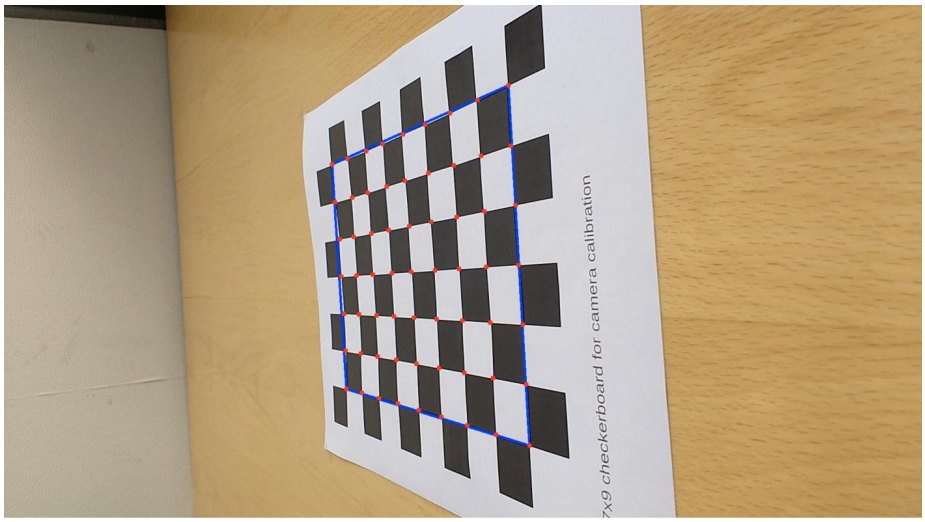


Figure 15: Approximate quadrilateral hull (in blue) on the corners

4.2 Finding Perimeter Corners

Once we have the 4 vertex corners, we can calculate the parameters of the 4 lines that form the perimeter. We define a line as follows:

$$ax + by = 1$$

where (x, y) are points that lie on that line.

We solve for the parameters for all 4 lines, each determined by 2 of the vertex corners. For each line, we find all the corners that lie on that line. A corner (x_k, y_k) lies on the line parameterised by a_i, b_i if:

$$|a_i x_k + b_i y_k - 1| < \epsilon$$

where ϵ is some small number determined empirically.

All the corners on the same line are sorted and recorded. We should then have all the perimeter corners.

4.3 Finding All Other Corners

We can find all the other corners using the same method as the one used to find perimeter corners. Because with the perimeter corners found and sorted, we can determine all the horizontal/vertical lines of the chessboard and find the corners that lie on these lines. The final result is shown in Figure 16.

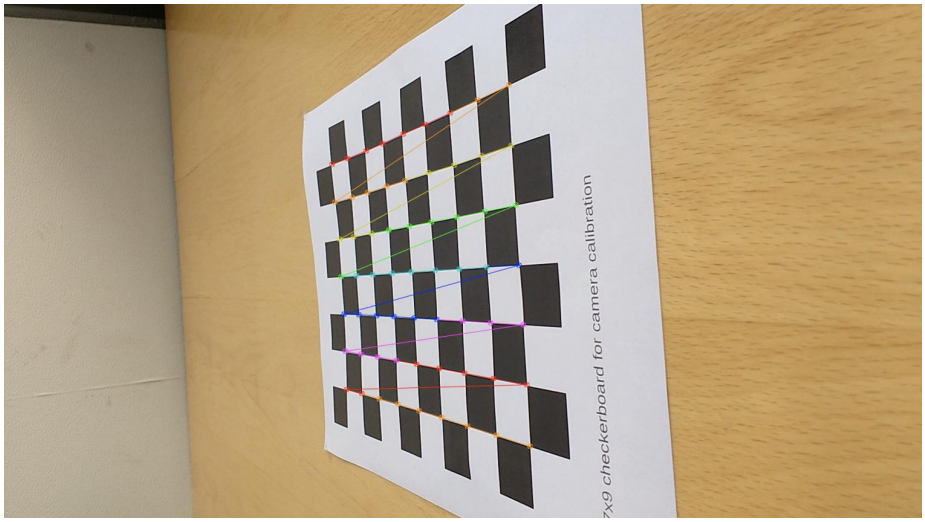


Figure 16: Ordered chessboard corners

5 Circles Grid Detection

Circles grid detection is not the focus of this report, we will only go over briefly. The new pattern is a 9x7 symmetric circles grid, as shown in Figure 17.

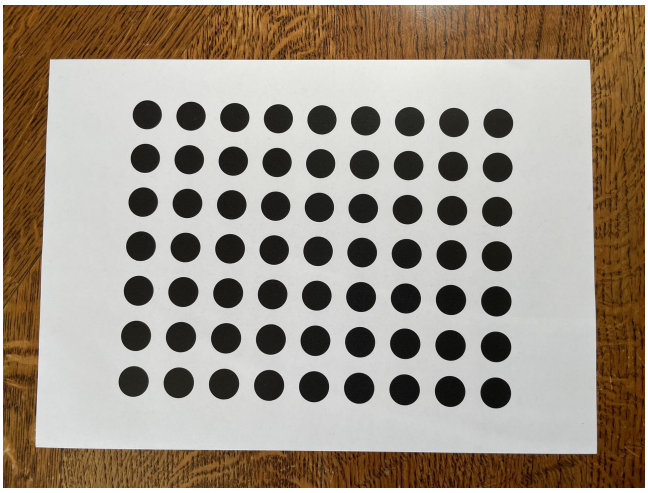


Figure 17: 9x7 symmetric circles grid

5.1 Change to the Provided code

We simply need to substitute the function `findChessBoardCorners()` with a new `findCirclesGrid()` function:

```
def findCirclesGrid(image, corner_size):
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    w, h = corner_size
    object_points = np.zeros((w * h,3), np.float32)
    for i in range(h):
        for j in range(w):
            object_points[i * w + j] = np.array([i, j, 0])

    blobParams = cv2.SimpleBlobDetector_Params()
    blobParams.filterByArea = True
    blobParams.minArea = 300
    blobParams.maxArea = 2000
    blobDetector = cv2.SimpleBlobDetector_create(blobParams)

    keypoints = blobDetector.detect(image)
    retval, corners = cv2.findCirclesGrid(image, corner_size, None,
                                          cv2.CALIB_CB_SYMMETRIC_GRID, blobDetector)
```

```
if retval == True:
    return True, object_points, corners
else:
    return False, None, None
```

The difference is that we are using the `cv2.findCirclesGrid()` function instead of the `cv2.findChessboardCorners()` function. We also supplied a manually tuned blob detector that detects, well, circular blobs on the pattern. One thing to note that the default settings for `cv2.findCirclesGrid()` do not seem to work with large images (1080p) very well and we had to supply 540p images. There are many parameters that users can tune to make things work but it is certainly not as convenient as detecting chessboards.

5.2 Change to Our Custom Algorithm

Similar `cv2.findCirclesGrid()`, we also needed to tune a blob detector. Once the blob detector is well-tuned, it can detect exactly the 63 blobs in all our calibration images. An example is shown in Figure 18

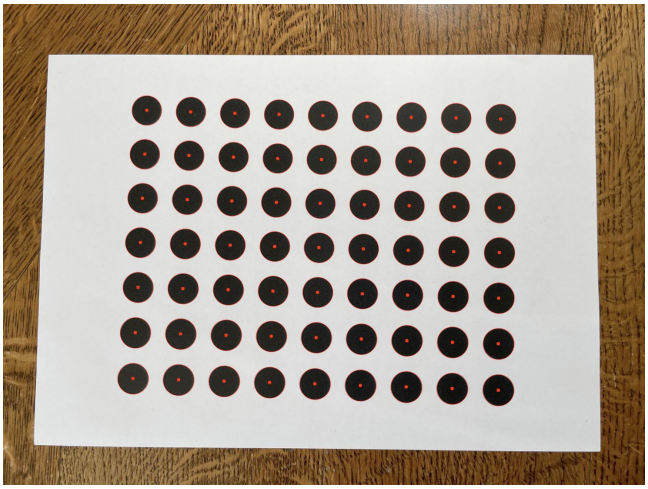


Figure 18: Circles grid detected using a blob detector

With all the corners detected, we can reuse the algorithm in Section 4 to order them.

6 Quantitative Results

We were provided 33 chessboard calibration images in coursework 1. Of the 33 images, 1 contains an incomplete chessboard. We collected 14 circles grid calibration images ourselves. Of the 14 images, 1 contains an incomplete circles grid. Thus the success rates are measured out of 32 and 13, respectively.

	Detection Rate	Reprojection Error
OpenCV Chessboard	32/32	0.1224
Custom Chessboard	31/32	0.1162
OpenCV Circles Grid	13/13	0.0195
Custom Circles Grid	13/13	0.0532

Table 1: Chessboard detection rate and reprojection error after calibration

In Table 1, we see that our custom chessboard detection algorithm could not detect one of the chessboards. We found that this was because the pattern in that image had a lot of shear, and the corner detector missed some corners. Though the reprojection errors between OpenCV's chessboard detection algorithm and ours are very similar.

Our custom circles grid detection algorithm could detect all the grids but the reprojection error is significantly worse than OpenCV's algorithm. If time allows, our algorithm could be improved by refining the blob center positions further.

We can also compare camera calibration results. Let the camera matrix and the distortion parameters found using OpenCV's algorithm (with chessboards) be C and D, respectively. And let the parameters found using our custom algorithm be C' and D'.

$$C = \begin{bmatrix} 1594.62 & 0.00 & 969.71 \\ 0.00 & 1486.25 & 522.79 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}$$

$$C' = \begin{bmatrix} 1587.50 & 0.00 & 968.72 \\ 0.00 & 1479.79 & 526.61 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}$$

$$D = \begin{bmatrix} 0.188 & -0.027 & -0.008 & 0.002 & -1.308 \end{bmatrix}$$

$$D' = \begin{bmatrix} 0.181 & 0.001 & -0.006 & 0.002 & -1.330 \end{bmatrix}$$

We can see that the calibration parameters found are not too dissimilar, and we will shown some virtual objects in the next section to support this.

To process all 33 chessboard calibration images, OpenCV's algorithm took 3 seconds and our custom algorithm took 10 seconds. This comparison is not very useful since our algorithm can be much faster if written in C++.