CAMERA CALIBRATION: PINHOLE MODEL(INTRINSIC PARAMETERS)+DISTORTION ELIMINATION(EXTRINSIC PARAMETERS)+CHESS BOARD PARAMETER REGRESS(THE SOLUTION)

Bo Wu[⋆†]

[⋆]data for the demo: OpenCV Official
[†] Toronto, ON, CA on July 06, 2020

ABSTRACT

TBD

* TBD
* TBD

***Index Terms*—** Camera Calibration

1. BACKGROUND

TBD

1. TBD;
2. TBD;
3. TBD.

TBD

2. GOAL

The ultimate goal of camera calibration is to undistort images based off the intrinsic and extrinsic properties of a camera.

3. CAMERA CALIBRATION: 针孔(-内参) + 消畸(-外参) + 标定板参数拟合(-SOLUTION)

3.1. CC在做什么？

1. 由像素坐标到世界坐标的相互切换。(基于针孔模型，4个内参)
2. 需要借助图像坐标、相机坐标作为中间辅助。
3. 除去针孔模型，还需要考虑相机的径向畸变(radial distortion)和切向畸变(tangential distortion)。(5个外参)
4. 用标定板进行参数拟合的时候还得多计算旋转矩阵R_{B2C}和平移矩阵T_{B2C}。(B2C: Board标定板 to Camera)

3.2. Camera Calibration–@BW notes

4. MATH EQUATIONS OF ABOVE SECTION

I've made some corrections of my previous notes here, refered from [1] and [2].

4.1. Correction I-Four Coordinates

★ **BW: There are 4 coordinates not 3.**

1. **Pixel Coordinates**: (u, v). As shown in Fig. 1, its origin (u=0,v=0) is at its upper-left corner.
2. **Image Coordinates**: (x, y). As shown in Fig. 1, its origin (x=0,y=0) is at the image center.
3. **Camera Coordinates**: (X_C, Y_C, Z_C). As shown in Fig. 1, its origin $(X_C = 0, Y_C = 0, Z_C = 0)$ is at its optical point F_C.
4. **World Coordinates**: (X, Y, Z). As shown in Fig. 1, often its origin (X=0,Y=0,Z=0) is at the same with Camera Coordinates.

Their relations are as follows.

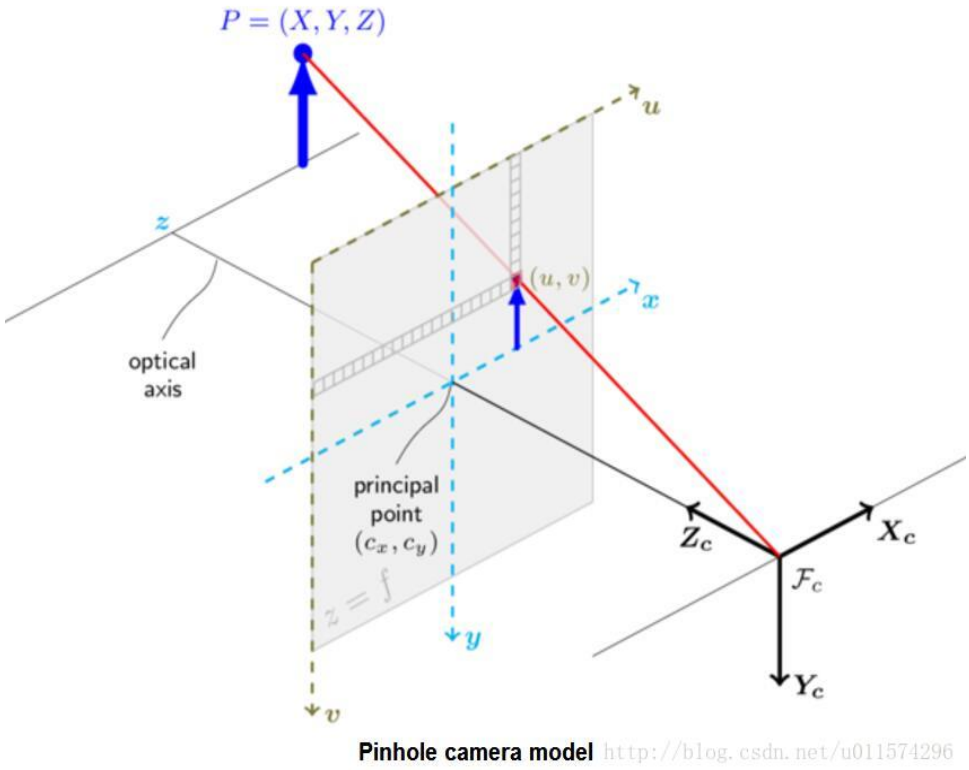


Fig. 1: Pinhole Camera Model.

★ Pixel to Image:

$$\begin{aligned} u &= \tfrac{x}{k} + c_x \\ v &= \tfrac{y}{l} + c_y \end{aligned} \tag{1}$$

where k, l represent the physical size of one pixel in image array, respectively (unit: mm/pixel); c_x, c_y denote the coordinates of [the Origin of Image Coordinates] at the Pixel Coordinates. It also can be written in vector multiplication form as below.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{k} & 0 & c_x \\ 0 & \frac{1}{l} & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2}$$

★ Image to Camera:

$$\begin{aligned} \tfrac{x}{f} &= \tfrac{X_C}{Z_C} \\ \tfrac{y}{f} &= \tfrac{Y_C}{Z_C} \end{aligned} \tag{3}$$

where f is the focal length, a.k.a. the distance between the image and the Origin of the Camera Coordinates. And, $f = f_x \cdot k = f_y \cdot l$. Since the calibration tools in OpenCV and Matlab don't provide the calculation of focal length f of the lens but they do provide the computation of pixel-focal-length f_x and f_y. We also can reform the above equation to vector format as follows.

$$Z_C \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = s \begin{bmatrix} f_x \cdot k & 0 & 0 \\ 0 & f_y \cdot l & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \tag{4}$$

▷ Combining (2) and (4), we can get Pixel to Camera transformation below.

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \tag{5}$$

It's also termed as the $camera\ matrix$:

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

f_x, f_y, c_x, c_y are the intrinsic parameters specific to a camera.

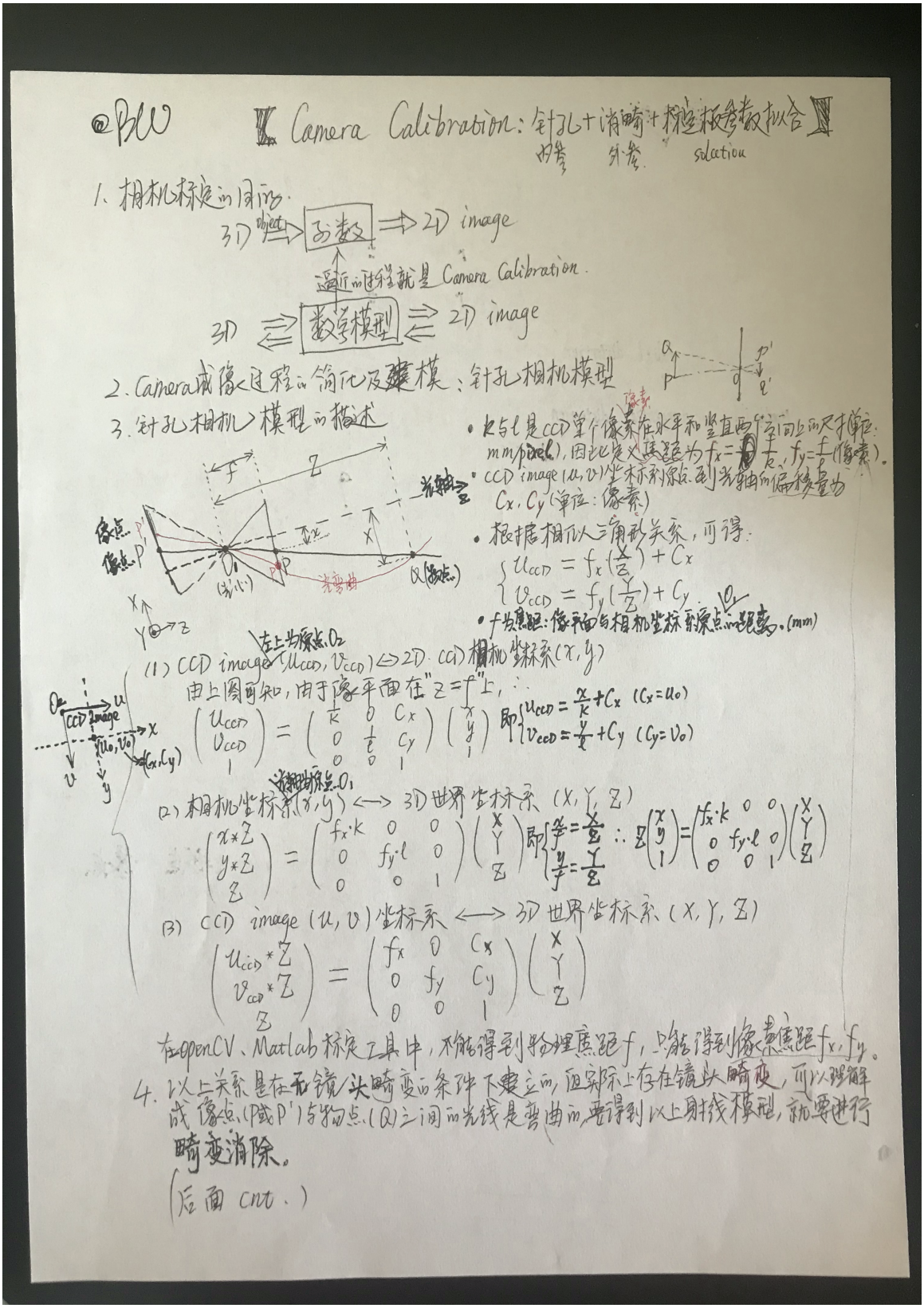


Fig. 2: 摄像头标定个人之前笔记之Page 1/2.

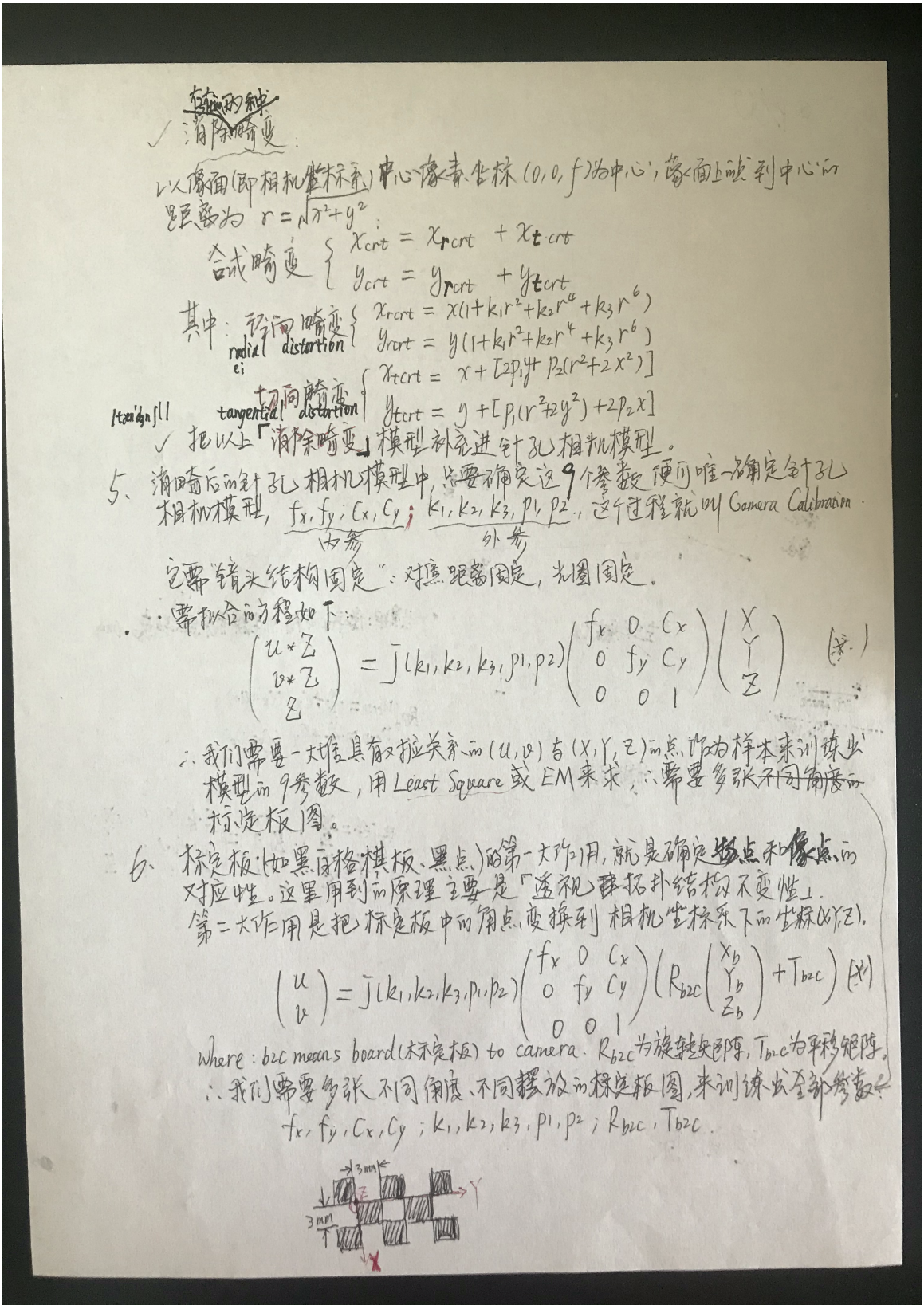


Fig. 3: 摄像头标定个人之前笔记之Page 2/2.

★ Camera to World:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{7}$$

where R denotes the orthogonal rotation matrix with the size of 3x3; r is the 3D translation vector with the size of 3x1.

★ Pixel to World: Combining (5) and (7), we can derive the transformation of Pixel-to-World as follows.

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{8}$$

4.2. Correction II-Extrinsic and Intrinsic Parameters

★ Extrinsic Parameters:
Extrinsic parameters corresponds to rotation R and translation vectors t which translates a coordinates of a 3D point to a coordinate system.

★ Intrinsic Parameters:
Intrinsic parameters include information like focal length (f_x,f_y) and optical centers (c_x,x_y), a.k.a. the $camera\,matrix$ (6). The focal length and optical centers can be used to create a camera matrix, which can be used to remove distortion due to the lenses of a specific camera. The camera matrix is unique to a specific camera, so once calculated, it can be reused on other images taken by the same camera.

In short, we need to find four parameters, also known as **camera matrix** given by:

$$Coefficients\,in\,camera\,matrix = (f_x, f_y, c_x, c_y) \tag{9}$$

4.3. Correction III-Distortion

Some pinhole cameras introduce significant distortion to images. Two major kinds of distortion are radial distortion and tangential distortion.

★ Radial Distortion:
Radial distortion causes straight lines to appear curved. Radial distortion becomes larger the farther points are from the center of the image. For example, one image is shown in Fig. 4 in which two edges of a chess board are marked with red lines. But, you can see that the border of the chess board is not a straight line and doesn't match with the red line. All the expected straight lines are bulged out. Radial distortion can be represented as follows:

$$\begin{aligned} x_{\text{rdistorted}} &= x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ y_{\text{rdistorted}} &= y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \end{aligned} \tag{10}$$

where $r = \sqrt{x^2 + y^2}$; (x, y) is the coordinates in Image Coordinates.

★ Tangential Distortion:
Similarly, tangential distortion occurs because the image-taking lense is not aligned perfectly parallel to the imaging plane. So, some areas in the image may look nearer than expected. The amount of tangential distortion can be represented as below:

$$\begin{aligned} x_{\text{tdistorted}} &= x + \left[2p_1 xy + p_2(r^2 + 2x^2)\right] \\ y_{\text{tdistorted}} &= y + \left[p_1(r^2 + 2y^2) + 2p_2 xy\right] \end{aligned} \tag{11}$$

★ Overall Distortion:

$$\begin{aligned} x_{\text{distorted}} &= x_{\text{rdistorted}} + x_{\text{tdistorted}} \\ y_{\text{distorted}} &= y_{\text{rdistorted}} + y_{\text{tdistorted}} \end{aligned} \tag{12}$$

In short, we need to find five parameters, known as **distortion coefficients** given by:

$$Distortion\,coefficients = (k_1, k_2, k_3, p_1, p_2) \tag{13}$$

4.4. Correction IV-Put All Together

$$Z_C \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = J(k_1, k_2, k_3, p_1, p_2) \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{14}$$

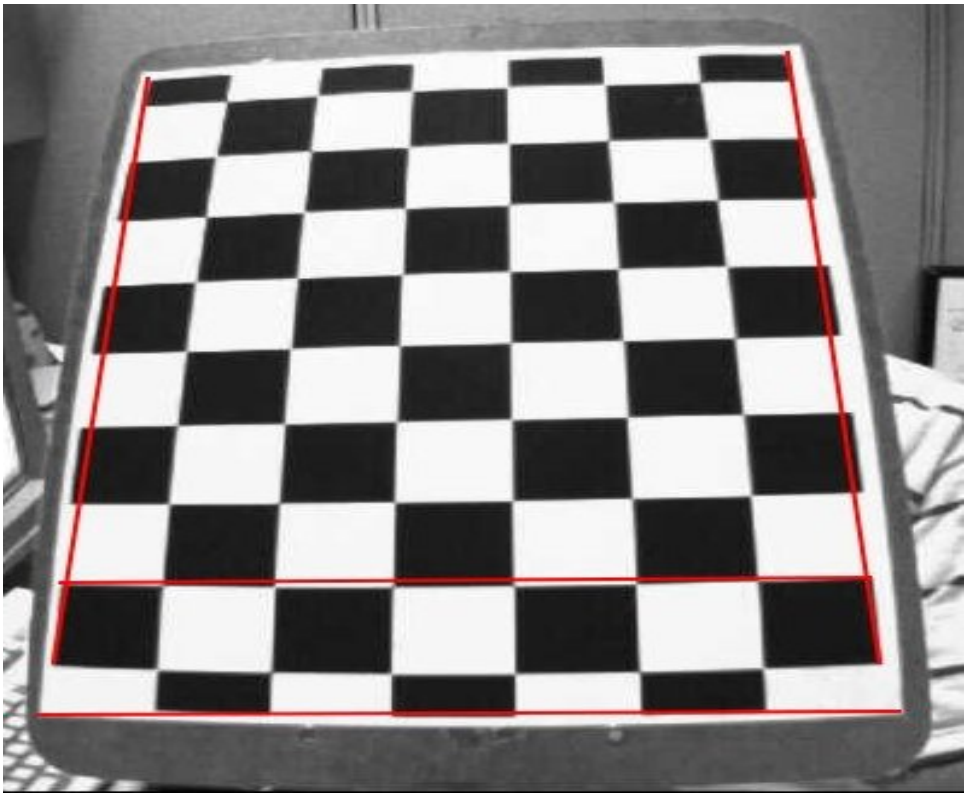


Fig. 4: Radial distortion example.

5. CAMERA CALIBRATION-PYTHON

5.1. Step 1: Object Points and Image Points Detection

As mentioned above, we need at least 10/(BW:11) test patterns for camera calibration [2]. OpenCV comes with some images of a chess board (see samples/data/left01.jpg – left14.jpg, BW:no left10.jpg?) [3], so we will utilize these. Consider an image of a chess board. The important input data needed for calibration of the camera is the set of 3D real world points and the corresponding 2D coordinates of these points in the image. 2D image points are OK which we can easily find from the image. (**These image points are locations where two black squares touch each other in chess boards**)

What about the 3D points from real world space? Those images are taken from a static camera and chess boards are placed at different locations and orientations. So we need to know (X,Y,Z) values. But for simplicity, we can say chess board was kept stationary at XY plane, (so Z=0 always) and camera was moved accordingly. This consideration helps us to find only X,Y values. Now for X,Y values, we can simply pass the points as (0,0), (1,0), (2,0), ... which denotes the location of points. In this case, the results we get will be in the scale of size of chess board square. But if we know the square size, (say 30 mm), we can pass the values as (0,0), (30,0), (60,0), ... Thus, we get the results in mm. (In this case, we don't know square size since we didn't take those images, so we pass in terms of square size).

3D points are called **object points** and 2D image points are called **image points**.

So to find pattern in chess board, we can use the function, cv.findChessboardCorners(). We also need to pass what kind of pattern we are looking for, like 8x8 grid, 5x5 grid etc. In this example, we use 7x6 grid (i.e. 7x6 corners). (Normally a chess board has 8x8 squares and 7x7 internal corners). It returns the corner points and **retval=ReturnValue** which will be True if pattern is obtained. These corners will be placed in an order (from red to blue as the resulted graph shows)

Listing 1: Chessboard Corner Detection

```
import numpy as np
import cv2 as cv
import glob
# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6 * 7, 3), np.float32)
objp[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)
# Arrays to store object points and image points from all the images.
objpoints = []  # 3d point in real world space
```

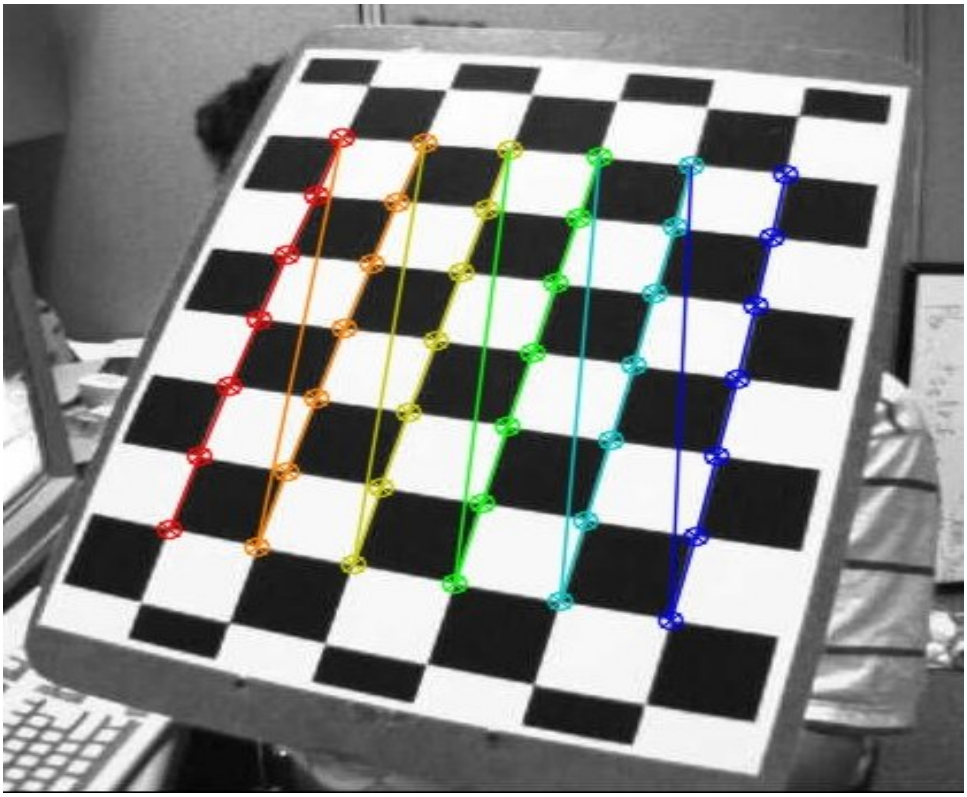


Fig. 5: Demo of detected chessboard corners.

```
imgpoints = []  # 2d points in image plane.

images = glob.glob('./sample_data_from_OpenCV/*.jpg')
for fname in images:
img = cv.imread(fname)
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
# Find the chess board corners
ret, corners = cv.findChessboardCorners(gray, (7, 6), None)
# If found, add object points, image points (after refining them)
if ret == True:
objpoints.append(objp)
corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
imgpoints.append(corners)
# Print current object points and image points
print(objp)
print(corners)
# Draw and display the corners
cv.drawChessboardCorners(img, (7, 6), corners2, ret)
cv.imshow('img', img)
cv.waitKey(500)
#cv.waitKey()
cv.destroyAllWindows()
```

5.2. Step 2: Calibration

Now that we have our **object points** and **image points**, we are ready to go for calibration. We can use the function, cv.calibrateCamera() which returns the camera matrix, distortion coefficients, rotation and translation vectors etc. Now you can store the camera matrix and distortion coefficients using write functions in NumPy (np.savez, np.savetxt etc) for future uses.

Listing 2: Camera Calibration

```
# Calibration
retval, cameraMatrix, distCoeffs, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, No
# Print Properties of Camera
```

```
print(f'retval:{retval}')
print(f'camera matrix: {cameraMatrix}')
print(f'distortion coefficients: {distCoeffs}')
print(f'rotation vectors: {rvecs}')
print(f'translation vectors: {tvecs}')
# Save Calibrated Parameters
np.savetxt('retval.txt', np.array([retval]), delimiter=',')
np.savetxt('cameraMatrix.txt', cameraMatrix, delimiter=',')
np.savetxt('distCoeffs.txt', distCoeffs, delimiter=',')
rvecs_squeezed = np.squeeze(np.array(rvecs), axis=2)
np.savetxt('rotationVectors.txt', rvecs_squeezed, delimiter=',', header=str(np.array(rvecs).shape))
tvecs_squeezed = np.squeeze(np.array(tvecs), axis=2)
np.savetxt('translationVectors.txt', tvecs_squeezed, delimiter=',', header=str(np.array(tvecs).shape))
```

5.3. Step 3: Undistortion

Now, we can take an image and undistort it. OpenCV comes with two methods for doing this. However first, we can refine the camera matrix based on a free scaling parameter using cv.getOptimalNewCameraMatrix(). If the scaling parameter alpha=0, it returns undistorted image with minimum unwanted pixels. So it may even remove some pixels at image corners. If alpha=1, all pixels are retained with some extra black images. This function also returns an image ROI which can be used to crop the result.

So, we take a new image (left12.jpg in this case. That is the first image in this chapter)

★ 1. Using cv.undistort()
This is the easiest way. Just call the function and use ROI obtained above to crop the result.

★ 2. Using remapping
This way is a little bit more difficult. First, find a mapping function from the distorted image to the undistorted image. Then use the remap function.

Listing 3: Undistortion

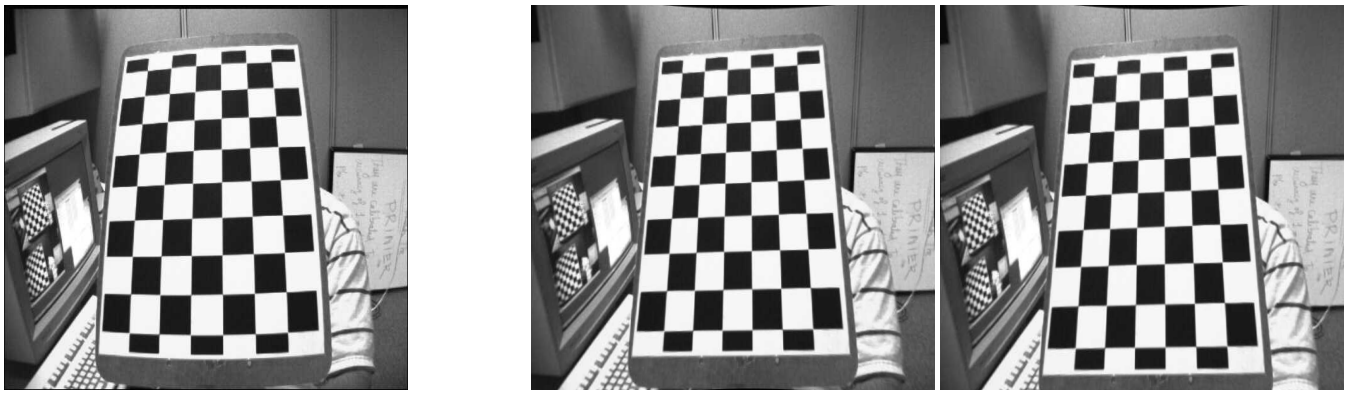
```
# Undistortion
img = cv.imread('./sample_data_from_OpenCV/left12.jpg')
h, w = img.shape[:2]
newcameraMatrix, roi = cv.getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, (w, h), 1, (w, h))

# 1. Using cv.undistort()
# This is the easiest way. Just call the function and use ROI obtained above to crop the result.
# undistort
dst = cv.undistort(img, cameraMatrix, distCoeffs, None, newcameraMatrix)
# crop the image
x, y, w, h = roi
dst = dst[y:y + h, x:x + w]
cv.imwrite('./undistortion_images/calibresult_of_left12.png', dst)

# 2. Using remapping
# This way is a little bit more difficult. First, find a mapping function from the distorted image to the undistorted
# undistort
mapx, mapy = cv.initUndistortRectifyMap(cameraMatrix, distCoeffs, None, newcameraMatrix, (w, h), 5)
dst = cv.remap(img, mapx, mapy, cv.INTER_LINEAR)
# crop the image
x, y, w, h = roi
dst = dst[y:y + h, x:x + w]
cv.imwrite('./undistortion_images/calibresult_of_left12_M2.png', dst)
```

5.4. Evaluation Calibration: Re-projection Error

Re-projection error gives a good estimation of just how exact the found parameters are. The closer the re-projection error is to zero (e.g. total error: 0.023686000375385673 in this demo), the more accurate the parameters we found are. Given the intrinsic, distortion, rotation and translation matrices, we must first transform the object point to image point using cv.projectPoints(). Then, we can calculate the absolute norm between what we got with our transformation and the corner finding algorithm. To find the average error, we calculate the arithmetical mean of the errors calculated for all the calibration images.

Listing 4: Evaluation of the Calibration

```
# Evaluation the calibration
mean_error = 0
for i in range(len(objpoints)):
```



(a) (b) (c)

Fig. 6: Calibration results demo. (a) The original image before calibration. (b) Calibrated RoI image of (a). (c) Calibrated remapping image of (a).

```
imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], cameraMatrix, distCoeffs)
error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2) / len(imgpoints2)
mean_error += error
print("total error: {}".format(mean_error / len(objpoints)))
```

6. STEREO VISION - TBD

For stereo applications, these distortions need to be corrected first [2]. To find these parameters, we must provide some sample images of a well defined pattern (e.g. a chess board). We find some specific points of which we already know the relative positions (e.g. square corners in the chess board). We know the coordinates of these points in real world space and we know the coordinates in the image, so we can solve for the distortion coefficients. For better results, we need at least **10** or BW: **11** test patterns since we have 4+5+2=11 unknowns.

7. REFERENCES

[1] ZealCV, "The relations of four coordinates in camera calibration model," [Online; posted 06-24-2017].

[2] OpenCV, "Camera calibration by opencv," [Online; posted 06-24-2017].

[3] OpenCV, "Chessboard sample snapshot by opencv," [Online; posted 06-24-2017].