MPhil in Advanced Computer Science

2018-2019
Coursework submission
Cover sheet

Student name……Zhuoni Jie……

College……Wolfson College……

CRSID……zj245……

Submit to Graduate Education
Date of submission

Module name Computer Vision

Module Code No. L248

Assignment No. 1

Assignment mark/grade

Assessor comments

GE Records……..

CRSID……zj245……

Module No……L248……

Assignment No……1……

GE stamp date and initials

Computer Vision: Exercise 1

Zhuoni Jie
zj245@cam.ac.uk

February 1, 2019

1 Camera calibration

Camera calibration is an important step in computer vision. In many cases, the overall performance of the machine vision system strongly depends on the accuracy of the camera calibration. Accurate calibration of cameras is necessary for applications that involve quantitative measurements such as dimensional measurements, depth from stereoscopy, or motion from images ([1]). The objective of camera calibration is to estimate the internal and external parameters of each camera, using which the mapping between 3-D reference coordinates and 2-D image coordinates can be described.

Two major distortions caused by cameras are radial distortion and tangential distortion. Due to radial distortion, straight lines will appear curved, and this effect increases when moving away from the center of image. The presence of the radial distortion manifests in form of the "barrel" or "fish-eye" effect. For the radial factor, one uses the following formula to represent the correction process:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1}$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2}$$

Similarly, another distortion is the tangential distortion which occurs because image taking lense is not aligned perfectly parallel to the imaging plane. So some areas in image may look nearer than expected. It is solved as below:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{3}$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{4}$$

These distortion coefficients are needed: (k_1 k_2 p_1 p_2 k_3).

The intrinsic and extrinsic parameters of a camera are also needed. The internal parameters describe camera geometric and optical characteristics, and external

parameters describe the 3-D position and orientation of the camera frame relative to a certain world coordinate system. Intrinsic parameters are specific to a camera, including includes information such as focal length (f_x, f_y) and optical centers (c_x, c_y). It is also called camera matrix. Once calculated, the camera-specific intrinsic parameters can be stored for future purposes. For the unit conversion we use the following formula:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{5}$$

Here the presence of w is explained by the use of homography coordinate system, and $w = Z$.

To calculate these parameters, the coordinates of 3D object points and their corresponding 2D projections in each view must be specified. Sample images of a well defined pattern such as a chessboard. We could detect some specific points (e.g., square corners in chessboard) in the pattern. Currently, initialization of intrinsic parameters is only implemented for planar calibration patterns (where Z-coordinates of the object points must be all zeros). 3D calibration rigs can also be used as long as initial camera matrix is provided. With the points' coordinates in real world space and coordinates in image known, the equation can be solved to get the distortion coefficients. To form a well-posed equation system and because of the amount of noise, for good results, we will probably need at least 10 good snapshots of the input pattern in different positions.

The chessboard images included in OpenCV are used here. One sample image of a chessboard is presented here for the sake of understanding. First, the function *cv2.findChessboarCorners()* is used to find pattern in chessboard. It returns the corner points and retval which will be True if pattern is obtained. These corners will be placed in an order of from left-to-right and top-to-bottom. Once we find the corners, we can increase their accuracy using *cv2.cornerSubPix()*. The pattern can be visualized as in Figure 1.

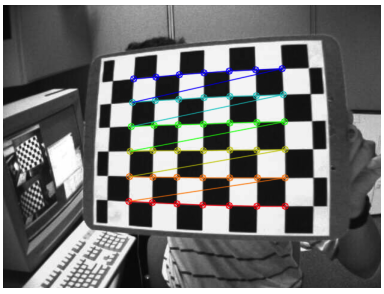

Figure 1: One image with pattern drawn on it

With these object points and image points, function *cv2.calibrateCamera()* can

Figure 2: The original image

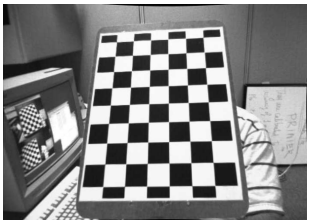

Figure 3: The calibrated image

be used for calibration. It returns the camera matrix, distortion coefficients, rotation and translation vectors. The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The algorithm is based on ([2]) and ([3]). The function returns the final re-projection error. The algorithm performs the following steps:

- Compute the initial intrinsic parameters (the option only available for planar calibration patterns) or read them from the input parameters. The distortion coefficients are all set to zeros initially unless some related parameters are specified.
- Estimate the initial camera pose as if the intrinsic parameters have been already known. This is done using *solvePnP()*.
- Run the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints.

Then the undistortion was done using the function *cv2.undistort*. A particular subset of the source image that will be visible in the corrected image can be regulated by *newCameraMatrix*, and is computed using *getOptimalNewCameraMatrix*, for a better control over scaling. In this function, if the scaling parameter alpha=0, it returns undistorted image with minimum unwanted pixels. If alpha=1, all pixels are retained with some extra black images. It also returns an image ROI which can be used to crop the result. First a mapping function from distorted image to undistorted image was found using *initUndistortRectifyMap*, and then the *remap* function was used. Figure 3 presents the result that all the edges are straight. The camera matrix and distortion coefficients can be stored for future uses.

Re-projection error describes the parameter estimation performance. Given the intrinsic, distortion, rotation and translation matrices, the object point was transformed to image point using *cv2.projectPoints()*. Then the absolute norm between transformation result and the corner finding algorithm was calculated. The arithmetical mean of the errors was calculated for all the calibration images,

and for OpenCV chessboard images I got an average error of 0.0237.

2 Image processing and perspective correction

A set of photos of the rubber stamp impressions in my ACS Research Skills logbook were taken for this perspective correction analysis, using a iPhone7 camera. Figure 4 shows some stamps presented with different rotations and illumination conditions.

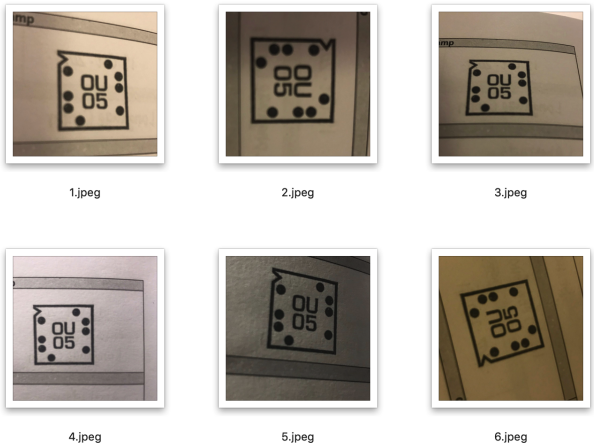

Figure 4: Stamps with different rotations and illumination conditions

The images were first loaded, resized, and converted to grayscale for processing. I then blur the image slightly by using the *cv2.bilateralFilter* function. Bilateral filtering has the nice property of removing noise in the image while still preserving the actual edges. The Canny edge detection was applied.

To find contours in an image, I implemented the *cv2.findContours* function. The method is destructive, which means it manipulates the image passed in, so I copied the data for manipulation. The first parameter passes the image to be processed. The second parameter *cv2.RETR_TREE* tells OpenCV to compute the hierarchy (relationship) between contours, and the *cv2.RETR_LIST* option can be used as well. Finally, the contours are compressed to save space using *cv2.CV_CHAIN_APPROX_SIMPLE*. This gave a list of contours in the image.

As major part of the images, the areas of stamps are quite large with respect to the rest of the regions in the images. The contours were therefore sorted from largest to smallest, by calculating the area of the contour using *cv2.contourArea*. The 10 largest contours were returned. Then the contour *stampCnt* corresponded to the stamp was initiated. Looping over the 10 largest contours in the query image, the contour was approximated using *cv2.arcLength*

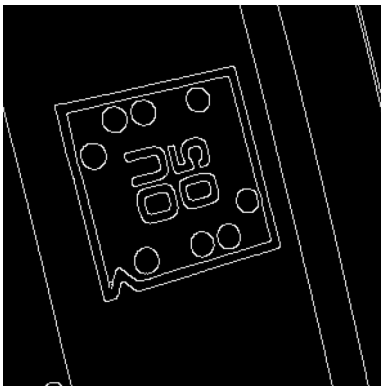

Figure 5: Stamp image after applying Canny edge detection

and *cv2.approxPolyDP*. I used 1.5% of the perimeter of the contour for the approximation level of precision.

Our stamp here is a hexagon, thus has six vertices. I checked the number of points each approximated contour had. If the contour had six points then it is likely the desired stamp area, and it was stored for check. Because only 10 largest contours were stored and considered, only a very small number of contours were investigated. The likelihood of another contour top 10 largest with a hexagon approximation is quite low. Drawing the contours detected, the stamp was successfully detected in this image as shown in Figure 6.

Figure 6: Stamp image with stamp contour correctly detected

A top-down view of stamp without major distortions is desired. In order to apply a perspective transformation, we need to know the top-left, top-right, bottom-right, and bottom-left corners of the contour. The top-left point will have the smallest sum of x and y coordinates, whereas the bottom-right point will have the largest sum of x and y coordinates. Taking the difference between

(x,y) coordinates, the top-right point will have the smallest difference, whereas the bottom-left point will have the largest difference.

First, in order to determine the width of the image, the distance between the bottom-right and bottom-left points was computed. Similarly, I computed the distance between the top-right and top-left points. Then I took the maximum of the width and height to determine the dimensions of the new transformed image. Third, a matrix dst was constructed to handle the mapping. The first entry in dst is the origin of the image — the top-left corner. I then specified the top-right, bottom-right, and bottom-left points based on calculated width and height. To compute the perspective transformation, the actual transformation matrix was calculated by making a call to *cv2.getPerspective* transformation and passing in the coordinates of the stamp area in the original image, followed by the four points specified for output image. Then we got the transformation matrix M. Finally, the transformation was applied by calling the *cv2.warpPerspective* function. The first parameter is the original image that we want to warp, the second is the transformation matrix M, and the final parameter is a tuple which is used to indicate the width and height of the output image. Now our top-down view of the stamp, as shown in Figure 7.

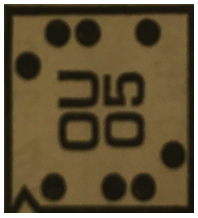

Figure 7: Stamp image after perspective correction and cropping

Figure 8: Stamp image after rotation

Then in order to get upright perspective corrected stamp images, the rotation degrees need to be determined. Notice the angle on one side of the stamp contour. If we view the envelop line of the stamp as a rectangle, then two points on one side are closer to two vertexes of the rectangle. When the stamp is in upright position, the angle and the two points are on the left side, which means the two points are closer to the top-left and bottom-left vertexes. Using this feature, we could decide the rotation degree of the stamp to make it upright, as shown in figure 8.

Appendix

The code written in python can be accessed here:
https://nbviewer.jupyter.org/github/karlinjzn/jiezn/blob/master/L248%20exercise%201.ipynb

References

[1] Weng, Juyang, Paul Cohen, and Marc Herniou. "Camera calibration with distortion models and accuracy evaluation." IEEE Transactions on Pattern Analysis & Machine Intelligence 10 (1992): 965-980.

[2] Zhang, Zhengyou. "A flexible new technique for camera calibration." IEEE Transactions on pattern analysis and machine intelligence 22 (2000).

[3] J.Y.Bouguet. MATLAB calibration tool.
http://www.vision.caltech.edu/bouguetj/calib_doc/