Camera Calibration

Project: Marker-based Localization

Niharika Jayanthi, Dheeraj Kamath
Mentor: Sanam Shakya

Goals

- To understand how a camera is calibrated
- Using cv2.calibrateCamera() function

Theory

Cheap pinhole cameras used for capturing images or videos introduce significant distortion in them. These distortions are constants and can be eliminated. Through calibration, one may also calculate the relationship between image units and real world units. On calibration, we obtain the camera matrix, which can be used later during pose estimation.

Here, to perform calibration, we have used a chessboard. The chessboard was shot from several different angles. A chessboard is used as it has well-defined points. Certain specific corners in the chessboard are detected. We obtain the image points from the picture and the object(real world) points are already known to us.

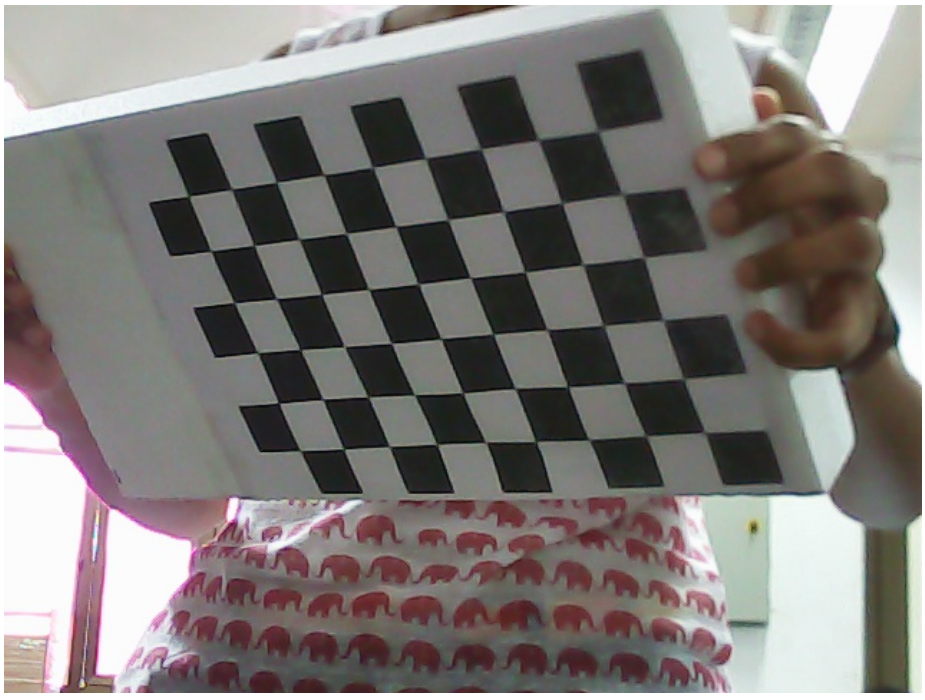

Figure 1: Chessboard held at an angle

Setup

You will need-

- A pinhole camera
- A chessboard

In the code, we will need to specify what kind of pattern we are looking for (9x6 grid, 5x5 grid etc). As specified above, we need the object points and their corresponding image points for calibration. It is easy to obtain the image points from the image itself. But for the object points, we can assume that the camera is in an XY plane, i.e. in Z=0 plane. The camera is assumed to be moved away. So the object points can be specified in terms of size of chessboard square. If the size of square is 1 unit, object points can be (0,0), (1,0), (2,0)...

Code

The main functions used here are-

cv2.calibrateCamera(objectPoints, imagePoints, imageSize[, cameraMatrix[, distCoeffs[, rvecs[, tvecs[, flags[, criteria]]]]]]) → retval, cameraMatrix, distCoeffs, rvecs, tvecs

where-

- **objectPoints** – In the new interface it is a vector of vectors of calibration pattern points in the calibration pattern coordinate space. The outer vector contains as many elements as the number of the pattern views. If the same calibration pattern is shown in each view and it is fully visible, all the vectors will be the same. Although, it is possible to use partially occluded patterns, or even different patterns in different views. Then, the vectors will be different. The points are 3D, but since they are in a pattern coordinate system, then, if the rig is planar, it may make sense to put the model to a XY coordinate plane so that Z-coordinate of each input object point is 0. In the old interface all the vectors of object points from different views are concatenated together.
- **imagePoints** – In the new interface it is a vector of vectors of the projections of calibration pattern points. imagePoints.size() and objectPoints.size() and imagePoints[i].size() must be equal to objectPoints[i].size() for each i. In the old interface all the vectors of object points from different views are concatenated together.
- **imageSize** – Size of the image used only to initialize the intrinsic camera matrix.

- **cameraMatrix** – Output 3x3 floating-point camera matrix. If CV_CALIB_USE_INTRINSIC_GUESS and/or CV_CALIB_FIX_ASPECT_RATIO are specified, some or all of fx, fy, cx, cy must be initialized before calling the function.
- **distCoeffs** – Output vector of distortion coefficients (k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]]) of 4, 5, or 8 elements.
- **rvecs** – Output vector of rotation vectors (see Rodrigues()) estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, that is, a real position of the calibration pattern in the k-th pattern view (k=0.. M -1).
- **tvecs** – Output vector of translation vectors estimated for each pattern view.
- **flags** – Different flags that may be zero or a combination of the following values: **CV_CALIB_USE_INTRINSIC_GUESS**- cameraMatrix contains valid initial values of fx, fy, cx, cy that are optimized further. Otherwise, (cx, cy) is initially set to the image center (imageSize is used), and focal distances are computed in a least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate extrinsic parameters. Use solvePnP() instead.

CV_CALIB_FIX_PRINCIPAL_POINT- The principal point is not changed during the global optimization. It stays at the center or at a different location specified when CV_CALIB_USE_INTRINSIC_GUESS is set too.

CV_CALIB_FIX_ASPECT_RATIO- The functions considers only fy as a free parameter. The ratio fx/fy stays the same as in the input cameraMatrix . When CV_CALIB_USE_INTRINSIC_GUESS is not set, the actual input values of fx and fy are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST- Tangential distortion coefficients (p_1, p_2) are set to zeros and stay zero. CV_CALIB_FIX_K1,. . . ,CV_CALIB_FIX_K6 The corresponding radial distortion coefficient is not changed during the optimization. If CV_CALIB_USE_INTRINSIC_GUESS is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.

CV_CALIB_RATIONAL_MODEL- Coefficients k4, k5, and k6 are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.

- **criteria** – Termination criteria for the iterative optimization algorithm.

---

cv2.findChessboardCorners(image, patternSize[, corners[, flags]]) → retval, corners

where-

image - Source chessboard view. It must be an 8-bit grayscale or color image.

patternSize - Number of inner corners per a chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows)).

corners - Output array of detected corners.

flags - Various operation flags that can be zero or a combination of the following values:

CV_CALIB_CB_ADAPTIVE_THRESH- Use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).

CV_CALIB_CB_NORMALIZE_IMAGE- Normalize the image gamma with equalizeHist() before applying fixed or adaptive thresholding.

CV_CALIB_CB_FILTER_QUADS- Use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads extracted at the contour retrieval stage.

CALIB_CB_FAST_CHECK- Run a fast check on the image that looks for chessboard corners, and shortcut the call if none is found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.

The complete code is -

```
#Imports
import cv2
import numpy as np

# Termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Define object points for a 9x6 grid
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
objp = objp * 26

# Arrays to store object points and image points
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.
```

```
vd = cv2.VideoCapture(0)

while(True):

    ret, img = vd.read()
    cv2.imshow("Video cap", img)

    inp = cv2.waitKey(1)

    if inp == 115: #If input is 's'
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, (9,6), None)

        # If found, add object points and image points
        if ret == True:
            objpoints.append(objp)

            #Refine image points
            cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
            imgpoints.append(corners)

            # Draw and display the corners
            cv2.drawChessboardCorners(img, (9,6), corners, ret)
            if ret == True:
                cv2.imshow('img',img)
                cv2.waitKey(500)

    elif inp == 27: break


ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
                                                   imgpoints, gray.shape[::-1],
                                                   None,None)



print "Camera calibration matrix\n\n", mtx

#Save camera matrix and distortion coefficients to be used later
np.save('cam_broke_mtx', mtx)
np.save('cam_broke_dist', dist)

cv2.destroyAllWindows()
```

```
mtx [[  9.62339259e+03   0.00000000e+00   3.17289374e+02]
 [  0.00000000e+00   7.56470056e+03   2.49651028e+02]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00]]
```

Figure 2: Rotation matrix

Resources

- http://docs.opencv.org/2.4.1/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
- https://www.ics.uci.edu/~majumder/vispercep/cameracalib.pdf
- http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/EPSRC_SSAZ/node5.html
- http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html
- http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_calib3d/py_calibration/py_ca

Exercises

- Try to calibrate different cameras.
- Try using different patterns to detect.