# Project 1

## Calibrating Camera and Superimposing an Object on the Calibration Pattern

Domagoj Korais and Marco Zullich

## Problem Statement

The problem may be split into two logical steps:

1. Camera calibration

   Based upon a set of pictures from the same camera depicting a common coplanar calibration pattern, compute the intrinsic and extrinsic parameters of the camera and its pose from the correspondences between 3D coordinates of the pattern's points of interest and the corresponding 2D coordinates of such points in the image reference frame.

   The task is ran twice, first without, then with compensation for radial distortion.

2. Superimposition of an object to the calibration plane

   Now that the parameters are known, they can be used to project points and solids of arbitrary 3D coordinate into any of the previous calibration images.

## Approach

Task 1 (camera calibration) is carried out using Zhang's method for homography estimation. The calibration object is a white/black 2D checkerboard with 35 intersections (7 columns x 5 rows) stuck on a paperback cardboard. All checkers within the object are square. Note that the calibration images provided during lectures were dropped since the rig does not comply with OpenCV's requirements of having a white area around the checkerboard pattern[1- findChessboardCorners].

As far as radial distortion is concerned, after the estimation of the homography and calculation of the intrinsic ($K$) and extrinsic ($R, t$) parameters, we estimate distortion coefficients using the radial model up to the 3rd order, we compensate for it refining the 3D-2D correspondences between checkerboard intersections in real world and within the images and re-running Zhang's procedure iteratively.

After obtaining $K, R, t$, we have a rule for transforming any point within the world's reference frame (whose origin is fixed at the upper-left intersection of the checkerboard) into the 2D coordinates of any one of our calibration pictures, and hence we can superimpose any (virtual) object of our choice to our images, thus carrying out task 2.

## Implementation

We opted in favor of implement our program in Python 3, using mainly two libraries:

- OpenCV (v. 3.4.3.18)

- NumPy

The former in particular uses very efficient methods to quickly carry out some common computer vision and image processing tasks.

The code is contained within the folder "Scripts", which contains two Python libraries:

- CameraCalibration.py is a library that encapsulates a number of methods constructed mainly over built-in OpenCV routines. It's designed to carry out task 1. Though specifically thought for this project, they can be easily reproduced on similar settings.
- drawingHelper.py is a library that contains only one method (draw_solid) and is designed for task 2.

The libraries are put in display within the IPython notebook `FinalCalibration.ipynb`, in which the actual tasks of the project are carried out.

## Code description

After importing all the images within the calibration folder, we first need to detect the checkerboard intersections within the picture and define the correspondences between points in world reference frame and image reference frame. This is done via the OpenCV method `findChessboardCorners`, which returns a flag indicating whether the specified patter was found, and a list containing the 2D coordinates of the intersections found within the image. 3D coordinates of such intersections are easily obtainable by setting the origin of the world reference frame one of the "extreme" corners of checkerboard (upper/lower-left/right), while taking the X and Y axis along the horizontal and vertical orientation of the checkerboard, such that the Z axis is perpendicular to it: so, all of the intersections have Z equal to 0. Then, the unit of measurement is conveniently fixed at the distance from one intersection and the following one along either the X or Y axis. Doing so, we have computed the correspondences between 2D and 3D checkerboard intersections. A refinement to such correspondences is obtained by improving the location of the 2D points by means of OpenCV's `cornerSubPix`, which is applicable to corner detectors such as the one used by `findChessboardCorners`. The method makes considerations on the gradient along the edges near the corners to rectify the location of the corners at subpixel precision. To visually check whether the algorithm correctly located the corners, we can show the images returned by our own method `detectAndSaveCheckerboardInList`.

After finding the correspondences, the next step is to calibrate the camera, thus getting the matrix $K$containing the intrinsic parameters for the camera and the matrix $[R|t]$representing the rigid transformation from the world to the camera reference frame. The first matrix is unique and is obtained from the collection of images previously imported, the transformation matrices are one for each image, since in every picture the checkerboard is positioned differently, hence we need each time a different transformation from world reference frame to image reference frame. To perform the calibration we use `calibrateCamera` from OpenCV. The method estimates such quantities plus it provides an estimate for distortion coefficients. Due to performance, we allowed a model with unknowns up to the 3rd order $(k_1, k_2, k_3)$. The model doesn't vary much from the one showed in lecture no. 2.

In addition to all of those quantities, the method returns also an estimate for reprojection error, which is actually the quantity the algorithm minimizes during its iterations. Eventually an estimate for total reprojection error for the whole set of calibration images is returned, though different from the one defined during lectures:, with $M_{(j)}^{rep}$the $[n \times 2]$matrix containing the $n$reprojected points for the image $j$, and with $M_{(j)}$the matrix containing the real coordinates of such points, OpenCV defines the total reprojection error for the set of images $(1, \dots, m)$ as:

$$\varepsilon_{TOT}(P_1, \dots, P_m) = \frac{\sum_{j=1}^{m} \left\| M_{(j)}^{rep} - M_{(j)} \right\|_2}{n}$$

This is returned as a performance metric for the camera calibration procedure.

We have defined another routine returning the calibration error as defined in lectures: by denoting with $( {}_j u'_{(i)},\ {}_j v'_{(i)})$ the reprojection of the $i$-th checkerboard intersection of the $j$-th image, the total reprojection error for image $j$is $\varepsilon(P_j) = \sum_{i=0}^{n} \left[ \left( {}_j u'_{(i)} - {}_j u_{(i)} \right)^2 + \left( {}_j v'_{(i)} - {}_j v_{(i)} \right)^2 \right]$. We don't define an aggregate for the whole image set, but just return a vector containing the metric for each calibration picture.

The calibration is repeated twice within the code: the first time we force the `calibrateCamera` not to estimate the distortion coefficents; the second time, instead, we let it estimate the distortion coefficients.

We then estimate the reprojection error for both results via OpenCV's `projectPoints` which, by inputting a list of 3D points, projects them onto an image of our choice. We choose to do it with the 3D coordinates of the checkerboard corners for the whole calibration image set, then we compute the reprojection error as aforementioned. To visualize the result of the calibration we plot the reprojected points corresponding to the checkerboard's corners within one of the calibration images; we then provide the percentage variation before and after considering radial distortion.

Eventually, we provide the code for plotting three solids (cube, pyramid, cylinder) over the calibration pattern in an image of user's choice using the coefficients obtained via camera calibration. Focusing on the cylinder, given the origin (center) of the base, the radius and the height of the cylinder, we calculate 256 points of the base circle (z-coordinate = 0) and compute the points for the ceiling circle by offsetting the z-coordinate of the base circle by the specified height. We then project all of the obtained points (base and ceiling) using OpenCV's `projectPoints` and draw the base and ceiling circle via `drawContours`.

A final function `calibrateLive` is provided that lets the user calibrate the camera while continuously taking pictures from one of the PC's webcams for a specified period of time.

## Results

We performed the analysis using two different subsets of images, a first set obtained using the camera embedded in our laptops, and the second using a reflex camera (Nikon D7000) and a fisheye lens (Samyang 8mm), in this way it was possible to investigate better the effects of radial distortion.

The complete results are showed in the IPython notebook `FinalCalibration.ipynb`; here we present a summary of the most important results achivied.

## Step 1 - Calibrate using Zhang procedure:

The results for the webcam (17 images) are the following:

The intrinsic computed parameters are the following:

$$K = \begin{bmatrix} 609.6151 & 0 & 324.9135 \\ 0 & 608.7214 & 244.5720 \\ 0 & 0 & 1 \end{bmatrix}$$

The extrinsic parameters (17 pairs of matrix/vector) aren't show here and are retrievable within the notebook.

## Step 2 - Computing total reprojection error:

The reprojection error was computed for each of the 17 calibration images, obtaining the following results:

| | |
|---|---|
| Mean reprojection error | 7 $px^2$ |
| Standard deviation | 6 $px^2$ |

We hereby present a reprojection of the intersections' points for the first calibration image.

*Image 1: example of reprojection of the checkerboard's corners for the first calibration picture*

## Step 3 - Add radial distortion compensation:

- The intrinsic computed parameters are the following:

$$K_{rad.dist.} = \begin{bmatrix} 609.7046 & 0 & 325.6246 \\ 0 & 608.8886 & 244.5244 \\ 0 & 0 & 1 \end{bmatrix}$$

For sake of comparison, we re-present the intrinsic coefficients from previous step:

$$K_{no\ rad.dist.} = \begin{bmatrix} 609.6151 & 0 & 324.9135 \\ 0 & 608.7214 & 244.5720 \\ 0 & 0 & 1 \end{bmatrix}$$

We see that the coefficients don't change much:

$$\Delta_{\%}(K_{rad.dist.}, K_{no\ rad.dist.}) = \begin{bmatrix} 0.0147\% & & 0.2189\% \\ & 0.0275\% & -0.0195\% \end{bmatrix}$$

The only quantity with a variation between 0.1% and 1.0% (still a very low variation) is the horizontal coordinate of the camera centre's projection.

- The distortion coefficients (up to third order):

$$k_1 = 0.0323; k_2 = -0.2811; k_3 = 0.8105$$

- As for step no. 2, the extrinsic parameters may be found within the notebook.

## Step 4 - Comparing total reprojection error:

The results are almost the same as step no.2, due to the fact that the radial distortion of the webcam is very low, in fact the leading distortion coefficient $k_1$ is close to 0.

This chart will help better understand the similarity:

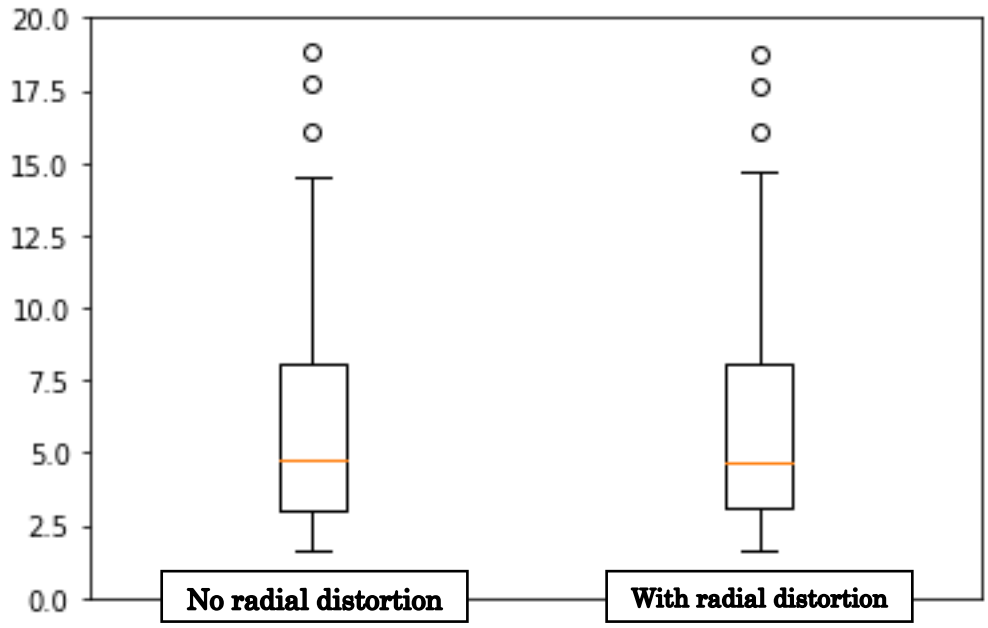


*Image 2: boxplot presenting the results for reprojection error for calibration without and with radial distortion*

## Step 5 - Cylinder superposition:

The superposition has been performed on all the images, here we will show only one example:

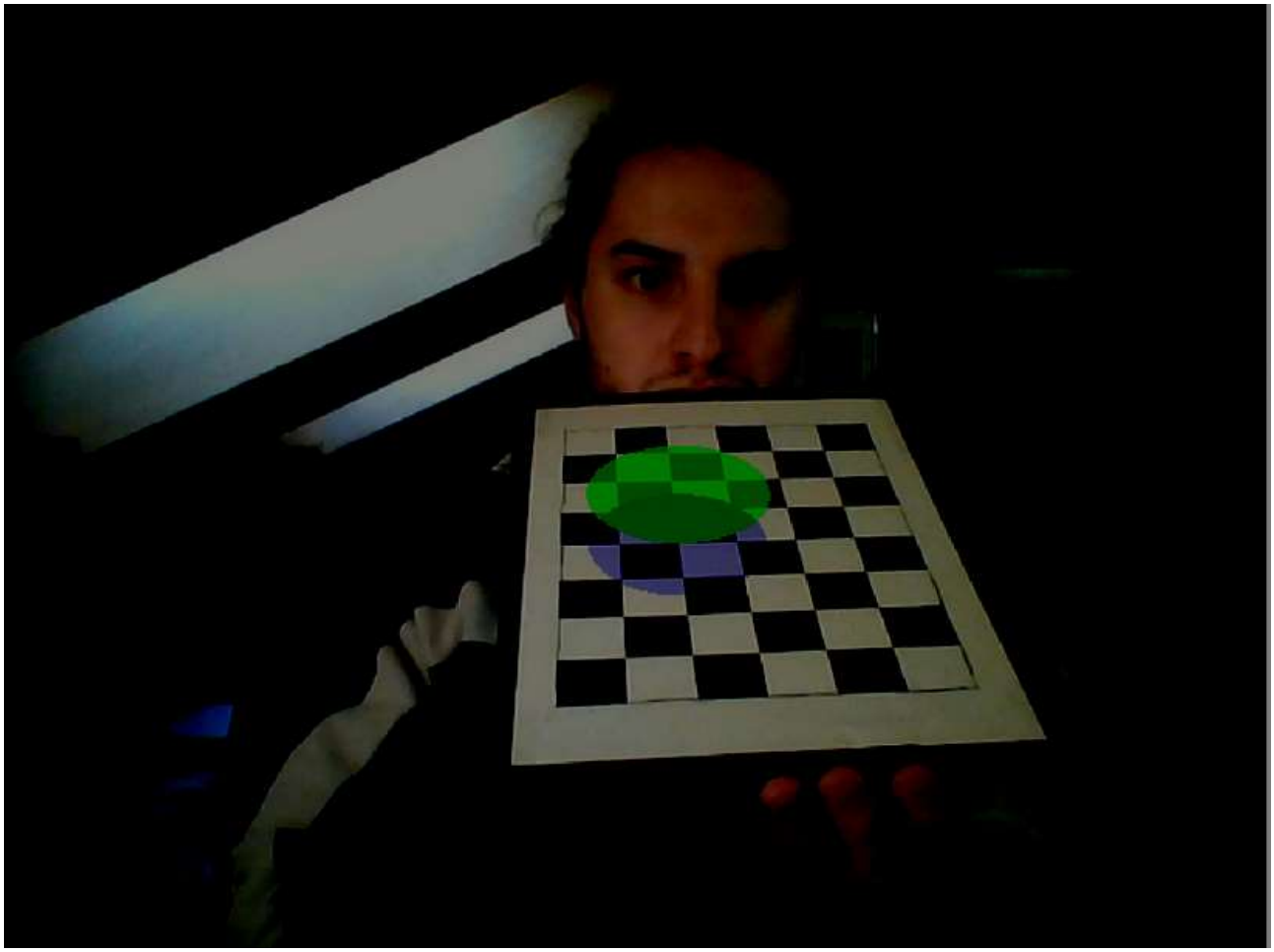

*Image 3: superimposing a cylinder on the calibration rig*

The cylinder has a radius 1.5 and height 1.0, while its centre is located 3 corners to the left and 3 down with respect to the upper-right corner of the checkerboard.

For other examples, check the notebook.

## Step 6 - Fisheye calibration:

To check better the distortion effects we decided to test the calibration procedure using a fisheye lens.

A total of 28 images were used to perform the calibration.

The results are presented in the notebook `FisheyeCalibration`, here we report only the final comparison results.

| Mean | | Standard Deviation | |
|---|---|---|---|
| No radial distortion | With radial distortion | No radial distortion | With radial distortion |
| 3568 px$^2$ | 17 px$^2$ | 5409 px$^2$ | 9 px$^2$ |

Correcting for distortion the total reprojection error passed from 3568 pixels$^2$, with a standard deviation of 5409 pixels$^2$, to a mean error of 17 pixels$^2$, with a standard deviation of 9 pixels$^2$. So we get a decrease in the error of 99%.

Next, a visual comparison of the reprojection error with radial distortion for webcam and reflex with fisheye:

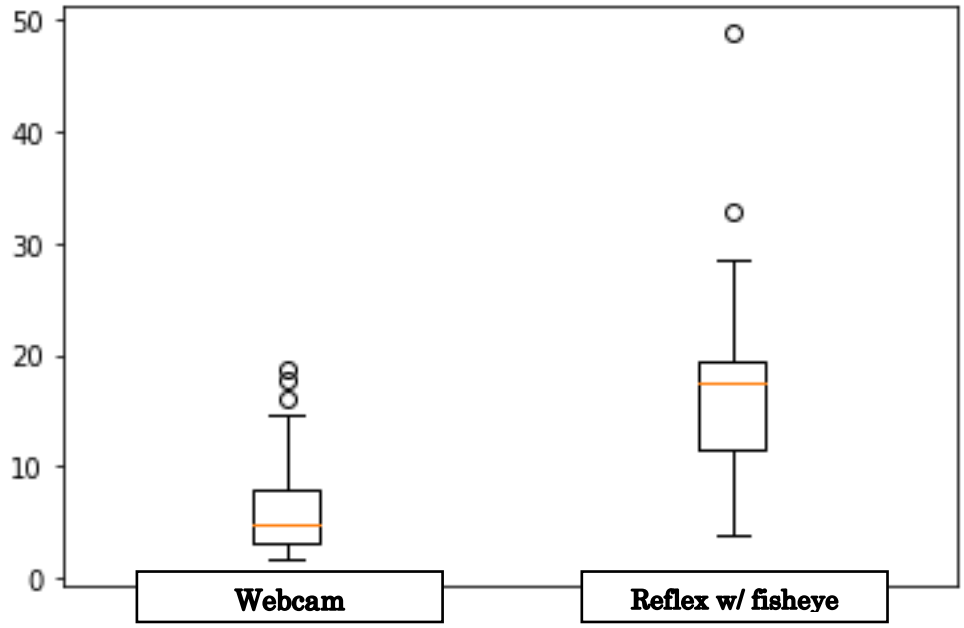


*Image 4: comparison of reprojection error: Webcam VS Fisheye*

## Conclusions

Within this project we compared the total reprojection errors obtained using the Zhang procedure on two different optical systems: a webcam and a reflex with a fisheye. The results shows that if there is not a strong distortion there is no need to compensate for radial distortion; in both cases the radial distortion compensation offer very solid results with a mean reprojection error lower than 20 px$^2$ (with some notable outliers in the case of the reflex). To improve such results, there're two possible roads to follow:

- Compensate for tangential distortion (not seen in lectures);
- Increase the number of iterations for the `calibrateCamera` method.

## References

1. OpenCV's 3.4.1 C++ and Python documentation for Camera Calibration and 3d Reconstruction: https://docs.opencv.org/3.4.1/d9/d0c/group__calib3d.html
2. OpenCV's 3.4.1 C++ and Python drawing functions: https://docs.opencv.org/3.4.1/d6/d6e/group__imgproc__draw.html
3. OpenCV's Python tutorial for pose estimation: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_pose/py_pose.html