Calibration of the Vision Setup in Robolab

Lecture notes by Thorbjørn Mosekjær Iversen

August 2019

1 Introduction

In this project, you will use the sensor input from a camera to locate an object in the real world. The setup is illustrated in Fig. 1. You will have to use the machine vision algorithms from the "Signalbehandling og machine vision" course to locate an object in the image and convert this position into a position in the physical work cell such that it can be picked up by the robot. To convert between image coordinates and real-world coordinates the vision system needs to be calibrated.

There are various methods for calibrating the setup. We utilize that the setup in Robolab is a special case where all points of interest in the image lie on a plane in the real world (the table surface). In this special case, we can compute a linear mapping from undistorted image coordinates to table coordinates without explicit knowledge of the camera's *pose* (position and rotation) in the work cell. The calibration has two steps:

1. Determine the *intrinsic* parameters of the camera (focal length, image center, skew) and lens distortion parameters. This information is used to undistort the recorded images thus undoing the distortion caused by the lens (see Fig. 2).
2. Compute a *projective transformation*, also called a *homography*, which maps points in the image plane to points in the table plane and vice versa.

The rest of this document describes how the calibration is performed as well as how the calibration results are applied.

2 Camera calibration

The standard method of camera calibration is based on [1] and relies on multiple recordings of a calibration object of known size (e.g. a checkerboard) to infer the intrinsic camera parameters as well as the lens

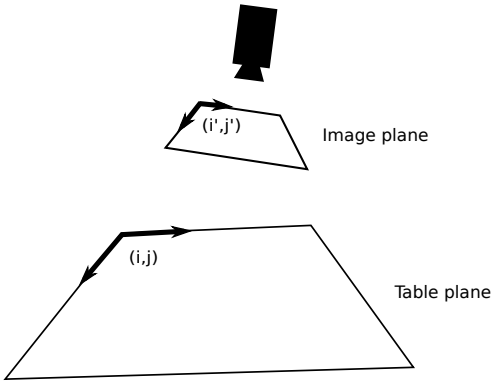

Figure 1: Setup in Robolab. It is possible to determine a linear transformation between points in the image plane (i', j') and points in the table plane (i, j)

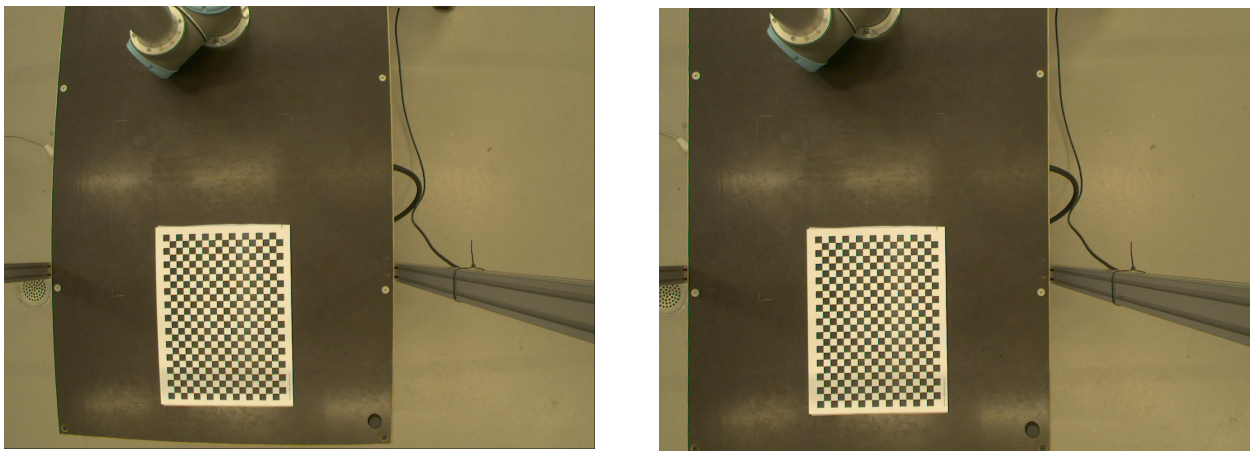

Figure 2: Rectification is the process of undistorting the recorded image. Notice that the table edge in the original image is curved due to barrel distortion and straight in the rectified image where the distortion has been rectified.

distortion.

In this project you will use the calibration to undistort the recorded images. The following note on the distortion coefficients is an extract from the OpenCV tutorial on camera calibration [2]:

> For the distortion OpenCV takes into account the radial and tangential factors. For the radial factor one uses the following formula:
>
> $$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
> $$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
>
> So for an old pixel point at (x, y) coordinates in the input image, its position on the corrected output image will be $(x_{corrected}, y_{corrected})$. The presence of the radial distortion manifests in the form of barrel or fish-eye effect.
>
> Tangential distortion occurs because the image taking lenses are not perfectly parallel to the imaging plane. It can be corrected via the formulas:
>
> $$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
> $$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$
>
> So we have five distortion parameters which in OpenCV are presented as one row matrix with 5 columns: $\text{Distortion}_{\text{coefficients}} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$

Note that x and y in the OpenCV tutorial fragment above are in normalized image coordinates. For a precise definition of how the undistortion is applied see the API doc for the OpenCV method `cv::initundistortrectifymap(...)` (link to API doc)

The calibration process is as follows:

1. Record `n` images of a calibration checkerboard at various poses relative to the camera. The checkerboard must be recorded at different angles relative to the camera in all areas of the camera's field of view. 20-30 images should be enough for your project.

2. Find the checkerboard corners in each image using steps below and collect them in a vector `std::vector<std::vector<cv::Point2f> > q`

   (a) Find the corners of the checker board markers using `cv::findChessboardCorners(...)`.

   (b) Improve the accuracy of the detected corners to sub-pixel accuracy using `cv::cornerSubPix(...)`.

   (c) Optional: Verify correct corner detection by drawing the detected corners using `cv::drawChessboardCorners(...)`.

3. Construct a `std::vector< std::vector< cv::Point3f > > Q` containing the 3D coordinates of the checker board corner points defined in the checker board's reference frame.

   (a) The OpenCV calibration method allows for the use of different calibration boards in different views. The size of `Q` is therefore equal to the number of images `n`. When the same checkerboard is used in all views, `Q` is thus a vector of n identical `std::vector<cv::Point3F> QView` containing the 3D coordinates of the checkerboard corner points in the checker board's frame. Since we can define this frame arbitrarily, we normally chose Z=0 and origo to be one of the outermost corner points. Note that the order of the points in `QView` has to match with the corresponding points in `cv::findChessboardCorners(...)`.

4. Calibrate the camera using `cv::calibrateCamera(...)`.

This procedure provides you with the (3×3) intrinsic camera matrix as well as the (1×5) distortion coefficients. To undistort an image you have to do the following:

1. Prepare a map for the image rectification using `cv::initUndistortRectifyMap(...)`. This method uses the camera matrix and distortion coefficients from the camera calibration. Note that this method has additional functionality which is not needed in our calibration, so set the parameters `R`, `newCameraMatrix`, and `size` to a 3×3 identity matrix, the original camera matrix, and the size of the original image.

2. Rectify the image using `cv::remap(...)`. This method uses the rectification map computed in the previous step.

Congratulations, you now have the tools to rectify your images. You will have to develop your own application for calibration and rectification using the OpenCV methods discussed above. On blackboard, you will find the c++ project `calibration_and_rectification` which you can use as a starting point.

3 Computing the Homography

As mentioned in the introduction, the special setup in Robolab makes it possible to compute a homography $\mathbf{H}$ which relates coordinates $\mathbf{x}'$ in the image plane to the coordinates $\mathbf{x}$ in the table plane. The problem is best formulated using homogeneous coordinates.

3.1 Homogeneous coordinates

First, the concept of homogeneous coordinates and projective space needs to be introduced. A proper introduction into these concepts is beyond the scope of this calibration guide, so the following is a very brief, intuitive introduction to homogeneous coordinates of 2D points. For a more in-depth introduction to projective geometry please refer to [3].

Imagine a light projector projecting an image unto a wall (Fig. 3). If we move the wall further away, the projected image remains unchanged except that the image has been enlarged. One way to define points in the image irrespective of the distance to the wall is through the use of homogenous coordinates: Instead of using 2D coordinates to describe an image point, we use 3D coordinates where the third coordinate is the

distance to the wall. Two image points at different projection distances are then said to be similar if their homogenous coordinates are equal down to a constant scale factor α:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \alpha \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Points in a digital image is often defined using the inhomogenous pixel coordinates (i, j). The normalized form of homogenous coordinates is when the third coordinate is 1. This makes for simple conversion between inhomogeous coordinates and homogenoues coordinates using the following equations:

Inhomogenoues coordinates to homogeneous coordinates: $f_{hom}(i, j) = \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$

Homogeneous coordinates to inhomogeneous coordinates: $f_{in}(x, y, z) = \begin{pmatrix} x/z \\ y/z \end{pmatrix}$

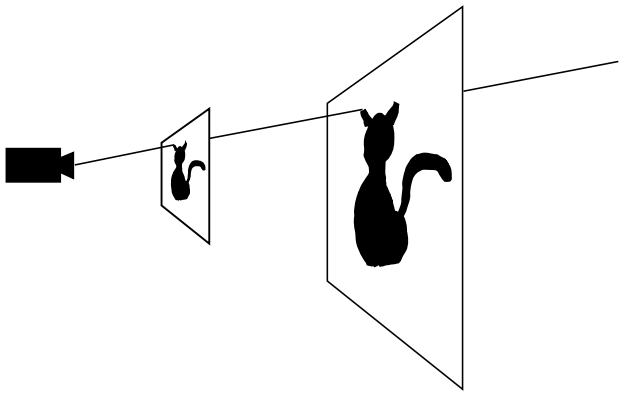

Figure 3: The image is the same regardless of the distance from the projector to the wall. The only difference is the scale of the image. Homogeneous coordinates can be used to describe image coordinates irrespective of the distance to the wall.

Homogenous coordinates makes it possible to write certain equations in a simpler more concise way. Take for instance the rigid transformation of a 2D vector $\mathbf{x} = (x, y)$ (rotation $\mathbf{R}$ followed by a translation $\mathbf{t}$). This can be written as:

$$\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$$

where $\mathbf{R}$ is a 2×2 rotation matrix.

This can also be expressed using homogenous coordinates:

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}$$
$$\rightarrow \quad \mathbf{x}'_{\text{hom}} = \mathbf{T}\mathbf{x}_{\text{hom}}$$

This is even more pronounced when several rigid transformations are applied in succession:

$$\text{Inhomogenous coordinates:} \quad \mathbf{x}' = \mathbf{R}_3(\mathbf{R}_2(\mathbf{R}_1\mathbf{x} + \mathbf{t}_1) + \mathbf{t}_2) + \mathbf{t}_3$$
$$\text{Homogenous coordinates:} \quad \mathbf{x}' = \mathbf{T}_3\mathbf{T}_2\mathbf{T}_1\mathbf{x}$$

It should now be clear that it is easy to convert back and forth between homogeneous and inhomogeneous coordinates, and that homogeneous coordinates have properties which makes it possible to simplify certain expressions.

3.2 Computing the homography from four points

The homography between a point $\mathbf{x}'$ in the image plane and a point $\mathbf{x}$ on the table plane can be expressed using homogenous coordinates as:

$$\mathbf{x} \sim \mathbf{H}\mathbf{x}' \tag{1}$$

where $\sim$ expresses that $\alpha\mathbf{x} = \mathbf{H}\mathbf{x}'$ for a non-zero scale factor α.

This relationship can be rewritten as a system of equations using the non-homogeneous image coordinates (i', j') and table coordinates (i, j):

$$\begin{pmatrix} i \cdot w \\ j \cdot w \\ w \end{pmatrix} = \begin{pmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & 1 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ 1 \end{pmatrix} \tag{2}$$

Note that since equation 1 is in homogeneous coordinates, we can freely choose a non-zero scaling factor on both the left-hand and right-hand side of the equation. It is, therefore, possible to scale the homography matrix $\mathbf{H}$ such that the entry $p_{22} = 1$. This is the starting point of section 5.3 in [4], which is similar to the one presented here.

Equation 2 expresses three equations of the eight unknown homography entries. However, the third equation ($w = p_{20}i' + p_{21}j' + 1$) can be substituded into the first two to produce:

$$\begin{aligned} i &= p_{00} \cdot i' + p_{01} \cdot j' + p_{02} - p_{20} \cdot i \cdot i' - p_{21} \cdot i \cdot j' \\ j &= p_{10} \cdot i' + p_{11} \cdot j' + p_{12} - p_{20} \cdot j \cdot i' - p_{21} \cdot j \cdot j' \end{aligned}$$

$$\iff \quad \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i' & j' & 1 & 0 & 0 & 0 & -ii' & -ij' \\ 0 & 0 & 0 & i' & j' & 1 & -ji' & -jj' \end{pmatrix} \begin{pmatrix} p_{00} \\ p_{01} \\ p_{02} \\ p_{10} \\ p_{11} \\ p_{12} \\ p_{20} \\ p_{21} \end{pmatrix}$$

A coresponding set of image↔table points, $(i', j') \leftrightarrow (i, j)$, thus leads to two equations of the eight unknown homography points. We can get eight equations by including four point pairs:

$$\begin{pmatrix} i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \\ i_4 \\ j_4 \end{pmatrix} = \begin{pmatrix} i'_1 & j'_1 & 1 & 0 & 0 & 0 & -i_1 i'_1 & -i_1 j'_1 \\ 0 & 0 & 0 & i'_1 & j'_1 & 1 & -j_1 i'_1 & -j_1 j'_1 \\ i'_2 & j'_2 & 1 & 0 & 0 & 0 & -i_2 i'_2 & -i_2 j'_2 \\ 0 & 0 & 0 & i'_2 & j'_2 & 1 & -j_2 i'_2 & -j_2 j'_2 \\ i'_3 & j'_3 & 1 & 0 & 0 & 0 & -i_3 i'_3 & -i_3 j'_3 \\ 0 & 0 & 0 & i'_3 & j'_3 & 1 & -j_3 i'_3 & -j_3 j'_3 \\ i'_4 & j'_4 & 1 & 0 & 0 & 0 & -i_4 i'_4 & -i_4 j'_4 \\ 0 & 0 & 0 & i'_4 & j'_4 & 1 & -j_4 i'_4 & -j_4 j'_4 \end{pmatrix} \begin{pmatrix} p_{00} \\ p_{01} \\ p_{02} \\ p_{10} \\ p_{11} \\ p_{12} \\ p_{20} \\ p_{21} \end{pmatrix} \quad \Leftrightarrow \quad \mathbf{b} = \mathbf{A} \cdot \mathbf{p}$$

The entries of the homography matrix is found by solving the matrix equation $\mathbf{A} \cdot \mathbf{p} = \mathbf{b}$ [1]. With the entries of the homography matrix known, Equation 2 can be used to convert between the image plane and the table plane.

[1] Note that it is possible to extend the method by including additional points which results in a matrix $\mathbf{A}$ of size $2n \times 8$. This is an overdetermined system of equations for which the least-squares solution is:

$$\mathbf{p} = \mathbf{A}^{+}\mathbf{b}$$

where $\mathbf{A}^{+} = (\mathbf{A}^{\top}\mathbf{A})^{-1}\mathbf{A}^{\top}$ is the pseudo inverse.

3.3 Using the OpenCV implementation

The OpenCV implementation of the the above method is called `cv::getPerspectiveTransform(...)`. The method takes four image points and corresponding table points and computes the homography **H** between them.

The homography matrix is used in the following way:

1. Find a point in the image expressed in pixel coordinates (i', j')
2. Convert the pixel coordinates to homogeneous coordinates $(\mathbf{x}')$
3. Use the homography to compute the corresponding homogeneous table coordinates $(\mathbf{x} = \mathbf{H}\mathbf{x}')$
4. Convert the homogeneous table coordinates to inhomogenous coordinates. These coordinates express the real world points on the table plane.

Note that even though OpenCV does most of the work, it is important that you understand the algorithm described in section 3.2.

References

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

[2] OpenCV camera calibration tutorial, Webpage: https://docs.opencv.org/3.4.6/d4/d94/tutorial_camera_calibration.html, *accessed 13.08.2019*

[3] Henrik Aanæs, *Lecture Notes on Computer Vision*, Webpage: http://www2.imm.dtu.dk/courses/02503/F11/LectureNotesOpt_II.pdf, *accessed 15.08.2019*

[4] Kenneth Dawson-Howe *A Practical Introduction to Computer Vision with OpenCV*. John Wiley & Sons, 2014