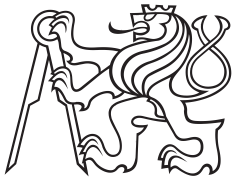


Bachelor's thesis



**Czech
Technical
University
in Prague**

F3

Faculty of Electrical Engineering

Department of Cybernetics

Automated camera calibration from laser scanning data in natural environments

Jan Brabec

Open Informatics

2014

Supervisor: Tomas Svoboda

BACHELOR PROJECT ASSIGNMENT

Student: Jan B r a b e c

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Automated Camera Calibration from Laser Scanning Data
in Natural Environments

Guidelines:

Propose an algorithm for matching entities in a cloud of 3D points acquired by laser scanning (Lidar) and corresponding image features. We assume a reasonable overlap between Lidar and image data. A non-laboratory environment is considered but the indoor/outdoor scene may be partly arranged in order to ease the matching and calibration. A human operator may stay in the computational loop but the process should guide/automate the process as much as possible. Process may run iteratively with human-check points between iterations.

The work has two main objectives. First, it should ease the process of integrating a new camera into the existing calibrated system. Second, it allows dynamic re-calibration of a camera positioned on the robotic arm and/or on a pan-tilt unit.

SW model shall be integrated into ROS middleware (ros.org) and tested with the already existing modules.

Bibliography/Sources:

- [1] Hartley, R. & Zisserman, A. (2003): Multiple view geometry in computer vision. Cambridge University, Cambridge.
- [2] Kubelka, V. & Svoboda, T. (2011): NIFTi Lidar-Camera Calibration(CTU--CMP--2011--15). Technical report, Center for Machine Perception, K13133 FEE Czech Technical University, Prague, Czech Republic.
- [3] Mirzaei, Faraz M and Kottas, D. G. & Roumeliotis, S. I. (2012): 3D LIDARcamera intrinsic and extrinsic calibration: Identifiability and analytical least-squares-based initialization, International Journal of Robotics Research 31(4), 452-467.
- [4] Nascimento, E. R.; Oliveira, G. L.; Campos, M. F. M.; Vieira, A. W. & Schwartz, W. R. (2012): BRAND: A robust appearance and depth descriptor for RGB-D images, in IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pp. 1720-1726.
- [5] Rusu, R. & Cousins, S. (2011): 3D is here: Point Cloud Library (PCL), in Robotics and Automation (ICRA), IEEE International Conference on.
- [6] Scaramuzza, D.; Harati, A. & Siegwart, R. (2007): Extrinsic Self Calibration of a Camera and a 3D Laser Range Finder from Natural Scenes, in IEEE International Conference on Intelligent Robots and Systems.

Bachelor Project Supervisor: doc. Ing. Tomáš Svoboda, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Jan B r a b e c

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Automatická kalibrace kamery z dat laserového dálkoměru v přirozeném prostředí

Pokyny pro vypracování:

Navrhnete vhodný algoritmus pro párování pozic v oblaku 3D bodů, získaných laserovým dálkoměrem a pozic korespondujících projekcí v obraze. Uvažujte přirozené, ale příhodné prostředí kolem robota. Tedy nikoliv laboratoř s kalibračními objekty, ale ani neprázdňé prostranství bez výrazných bodů. Součástí procesu může být zásah operátora, ale proces sám by měl úlohu operátora maximálním způsobem usnadnit, navádět ho, například postupným odhadem a zpřesňováním modelu. Algoritmus může využít dostupných orientačních kalibračních parametrů.

Práce sleduje dva hlavní cíle. Výsledný software by měl maximálně usnadnit integraci nové kamery do systému. Automatické párování dovolí dynamickou re-kalibraci kamery na pohyblivém rameni, a/nebo na natáčecí a naklápěcí jednotce (pan-tilt).

SW modul integrujte do prostředí ROS (ros.org) a napojte na existující moduly snímající data.

Seznam odborné literatury:

- [1] Hartley, R. & Zisserman, A. (2003): Multiple view geometry in computer vision. Cambridge University, Cambridge.
- [2] Kubelka, V. & Svoboda, T. (2011): NIFTi Lidar-Camera Calibration(CTU–CMP–2011--15). Technical report, Center for Machine Perception, K13133 FEE Czech Technical University, Prague, Czech Republic.
- [3] Mirzaei, Faraz M and Kottas, D. G. & Roumeliotis, S. I. (2012): 3D LIDARcamera intrinsic and extrinsic calibration: Identifiability and analytical least-squares-based initialization, International Journal of Robotics Research 31(4), 452-467.
- [4] Nascimento, E. R.; Oliveira, G. L.; Campos, M. F. M.; Vieira, A. W. & Schwartz, W. R. (2012): BRAND: A robust appearance and depth descriptor for RGB-D images, in IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, pp. 1720-1726.
- [5] Rusu, R. & Cousins, S. (2011): 3D is here: Point Cloud Library (PCL), in Robotics and Automation (ICRA), IEEE International Conference on.
- [6] Scaramuzza, D.; Harati, A. & Siegwart, R. (2007): Extrinsic Self Calibration of a Camera and a 3D Laser Range Finder from Natural Scenes, in IEEE International Conference on Intelligent Robots and Systems.

Vedoucí bakalářské práce: doc. Ing. Tomáš Svoboda, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Acknowledgement / Declaration

I would like to thank my advisor Tomas Svoboda for his invaluable guidance and assistance throughout this project and for the opportunity to work on other interesting projects with him. I would also like to thank Tomas Petricek for providing a valuable insight and other members of the NIFTi team for their assistance in operating the ground robot.

Finally, I would like to thank my family for their endless support. My father deserves a special mention because he introduced me to programming and was my first mentor.

The work was supported by EC project FP7-ICT-609763 TRADR and by the CTU project SGS13/142/OHK3/2T/13. Any opinions expressed in this paper do not necessarily reflect the views of the European Community. The Community is not liable for any use that may be made of the information contained herein.

I declare that I have developed the presented work independently and that I have listed all information sources used in accordance with the Methodical guidelines on maintaining ethical principles during the preparation of higher education theses.

.....

Prague, 23 May 2014

Abstrakt / Abstract

Vyvinuli jsme aplikaci pro vnější kalibraci kamer z dat získaných LIDAR scannerem. Protože prosté hloubkové obrazy z LIDAR scanneru nejsou dostatečně detailní, zpracovali jsme je, abychom zvýraznili hrany a rohy a umožnili tak operátorovi vytvářet korespondence mezi body v prostoru a body v obraze. Také jsme vyvinuli techniku pro lokální opravu korespondencí pro případ, že se operátor dopustí drobných chyb. Aplikace je implementována jako node v ROSu. Vykonali jsme experimenty na mobilním robotovi vyvíjeném pro vyhledávání a záchranné práce v městském prostředí. Experimentálně jsme ukázali, že aplikace může být použita i mimo laboratorní prostředí pro rychlou kalibraci nové kamery nebo rekalibraci již přítomné kamery. To je velká výhoda ve srovnání se současnými nástroji dostupnými v ROSu, které vyžadují použití speciálních kalibračních vzorů a jsou tak omezené pouze na laboratorní prostředí.

Klíčová slova: kamera, kalibrace, LIDAR, ROS

Překlad titulu: Automatická kalibrace kamery z dat laserového dálkoměru v přirozeném prostředí

We have developed an application for extrinsic camera calibration from the data acquired by the LIDAR scanner. Since the raw range images from the LIDAR scanner do not possess enough detail, we processed the range images to highlight edges and corners and allow the operator to create correspondences between the world points and the image points. We have also developed a technique for local correction of the correspondences in case the operator makes a slight mistake. The application is implemented as a node in Robot Operating System (ROS). We have performed experiments on a mobile robot intended for urban search and rescue. We experimentally show that the application can be used outside the laboratory to quickly calibrate a new camera in the system or recalibrate an already present camera. That is a big advantage compared to the present tools available in ROS that usually require the use of special calibration patterns and are restricted to the laboratory environment only.

Keywords: camera, calibration, LIDAR, ROS

Contents /

1 Introduction	1
2 Camera calibration	4
2.1 Camera geometry	4
2.1.1 Pinhole camera model	4
2.1.2 Non-linear distortion	6
2.2 Pose estimation and PnP	6
2.2.1 Direct Linear Transformation algorithm	6
2.2.2 Reprojection error minimization	7
2.2.3 EPnP	7
2.3 Point cloud visualizations	7
2.3.1 Directional images	8
2.4 Local correction	9
2.5 RANSAC	10
2.6 Point cloud coloring	10
3 Architecture	12
3.1 Package structure	12
3.2 Top-level architecture	13
3.3 Calibration launcher	14
3.4 Graphical user interface	15
3.4.1 Scene views and correspondences	15
3.4.2 Range image visualizations	16
3.4.3 Calibration	17
4 User manual	19
4.1 Applications requirements	19
4.2 Installation and build	19
4.2.1 ROS Fuerte Turtle	19
4.2.2 Newer versions of ROS ..	19
4.3 Required ROS components	20
4.3.1 Using Bag files	20
4.4 Launching the application	21
4.4.1 Launch parameters	21
4.4.2 Launching the GUI	22
4.5 Using the GUI to calibrate a camera	22
4.5.1 Using different point cloud visualizations	23
4.5.2 Creating correspondences	24
4.5.3 Running the calibration ..	25
4.5.4 Calibration results	26
4.6 Other useful tools	26
4.6.1 rviz	27
4.6.2 Cloud coloring	27
5 Experiments	29
5.1 Scenario 1 - Corridor	30
5.1.1 Comparison with the original calibration	32
5.1.2 Validation using the point cloud coloring	33
5.1.3 Calibration by inexperienced operator	34
5.2 Scenario 2 - Hall	35
5.2.1 Comparison with the original calibration	36
5.2.2 Validation using the point cloud coloring	37
5.2.3 Calibration by inexperienced operator	39
5.3 Scenario 3 - Courtyard 1	40
5.3.1 Comparison with the original calibration	42
5.3.2 Validation using the point cloud coloring	43
5.4 Scenario 4 - Courtyard 2	44
5.4.1 Comparison with the original calibration	45
5.4.2 Validation using the point cloud coloring	47
5.5 Scenario 5 - Calibration of an external camera	47
5.5.1 Validation using the point cloud coloring	49
5.6 Summary	49
6 Conclusion	51
References	53
A Enclosed CD	55

Chapter 1

Introduction

In this thesis, we describe an application for *extrinsic camera calibration* estimation, that we have developed for ROS[1]. The application has two main objectives. First, the aim of the application is to ease the process of integrating a new camera into the existing calibrated system. Second, we wanted to allow the dynamic re-calibration of a camera positioned on the robotic arm or on a pan-tilt unit. The application is made for a mobile robot intended for urban search and rescue, see Figure 1.2. There is no barrier, however, that would prevent the application to be used in other robotic systems using ROS.

The camera calibration is one of the most important tasks in the field of computer vision. The results of many other algorithms depend on the quality of the underlying camera's calibration. Many different approaches and tools for both *intrinsic* and *extrinsic* calibration exist[2]. However, most of those tools, such as the `camera_calibration` package[3] in ROS, require the use of special calibration patterns or other tools. This is cumbersome inside the lab and almost impossible in exteriors. In our case, the robot can be even exploring dangerous environments where the operator has no access to it.

Most of the current algorithms for camera calibration require a set of *correspondences* between the *world points* and the *image points* as an input. Since our robot is equipped with a LIDAR scanner, we decided to compute the mutual orientation and translation between the LIDAR scanner and the camera that needs to be calibrated. The most challenging part of our application is the matching between the depth data gained from the LIDAR scanner and the image data from the camera. That is because the nature of the data acquired from the LIDAR scanner is completely different compared to the camera's images.

Human is well trained in analysing standard color images but fairly unskilled when inspecting point clouds or depth data in general. Therefore, it is difficult to create a sufficient number of reasonable correspondences from raw *range images*, see Figure 1.3. Our work is mainly inspired by a manual approach [4] which suggested to replace the usual range images by several alternatives. We processed the range images to highlight edges and corners that would not be otherwise recognizable. An important part of our application is the graphical user interface that makes the manual creation of the correspondences as easy and comfortable for the operator as possible. The application also has an ability to perform a local correction of the correspondences, created manually by the operator, and can help with the identification of the correspondences that are not correct. We intended to go one step further and tried to create the correspondences automatically. We processed the range images and experimented with SIFT descriptors to identify the correspondences but we were unsuccessful with this approach.

The application performs only the extrinsic calibration for several reasons. The main reason is, that in the presence of lens distortion, it is difficult to correctly estimate all of the camera's intrinsic parameters from a limited number of correspondences the

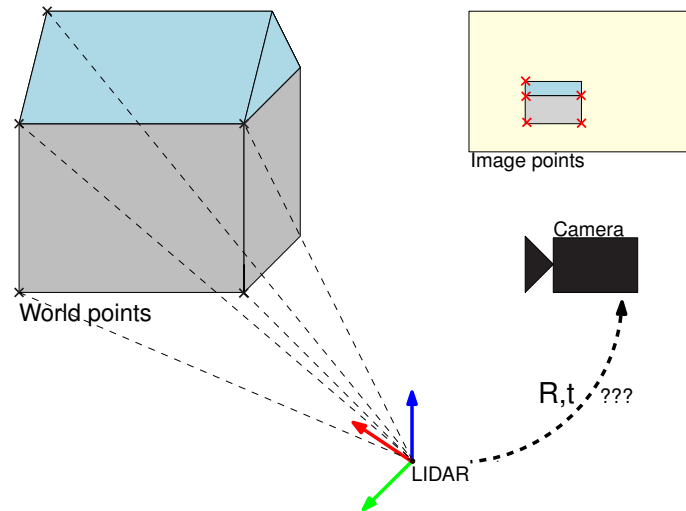


Figure 1.1. Our approach. The transformation between the LIDAR and the camera is estimated from corresponding points marked in the data from the LIDAR and the camera image.

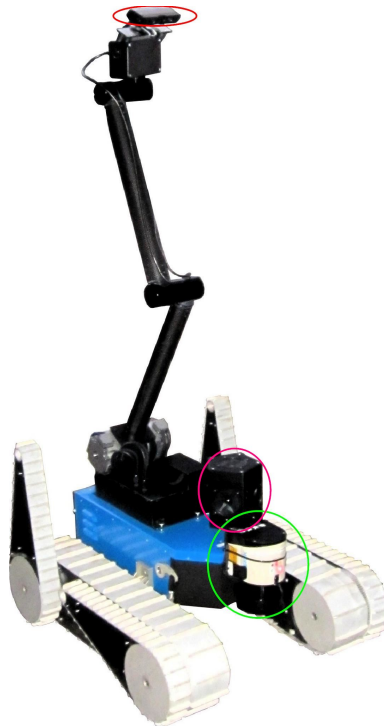


Figure 1.2. NIFTi robot with its robotic arm extended. The camera on the pan-tilt unit is highlighted in the red ellipse. The LIDAR scanner is in the green ellipse. The omnica camera used for most of the experiments is highlighted in the pink ellipse. The goal of our application is to compute position and orientation of a camera with respect to the LIDAR. Original image (without the highlighting) is from [5].

operator is able to create. Secondly, the intrinsic parameters of the camera are far less volatile and there is usually no problem to estimate them with higher accuracy and precision, using the usual tools mentioned above, in the laboratory.

This thesis is structured into the following chapters. The chapter *camera calibration* explains the mathematics behind the camera models and the algorithms we used for cal-

ibration. The *architecture* chapter explains the actual design of the application and the reasoning behind it. The *user manual* chapter describes the application from the user's point of view. We have performed various experiments to evaluate the application. Their summary can be found in the *experiments* chapter.

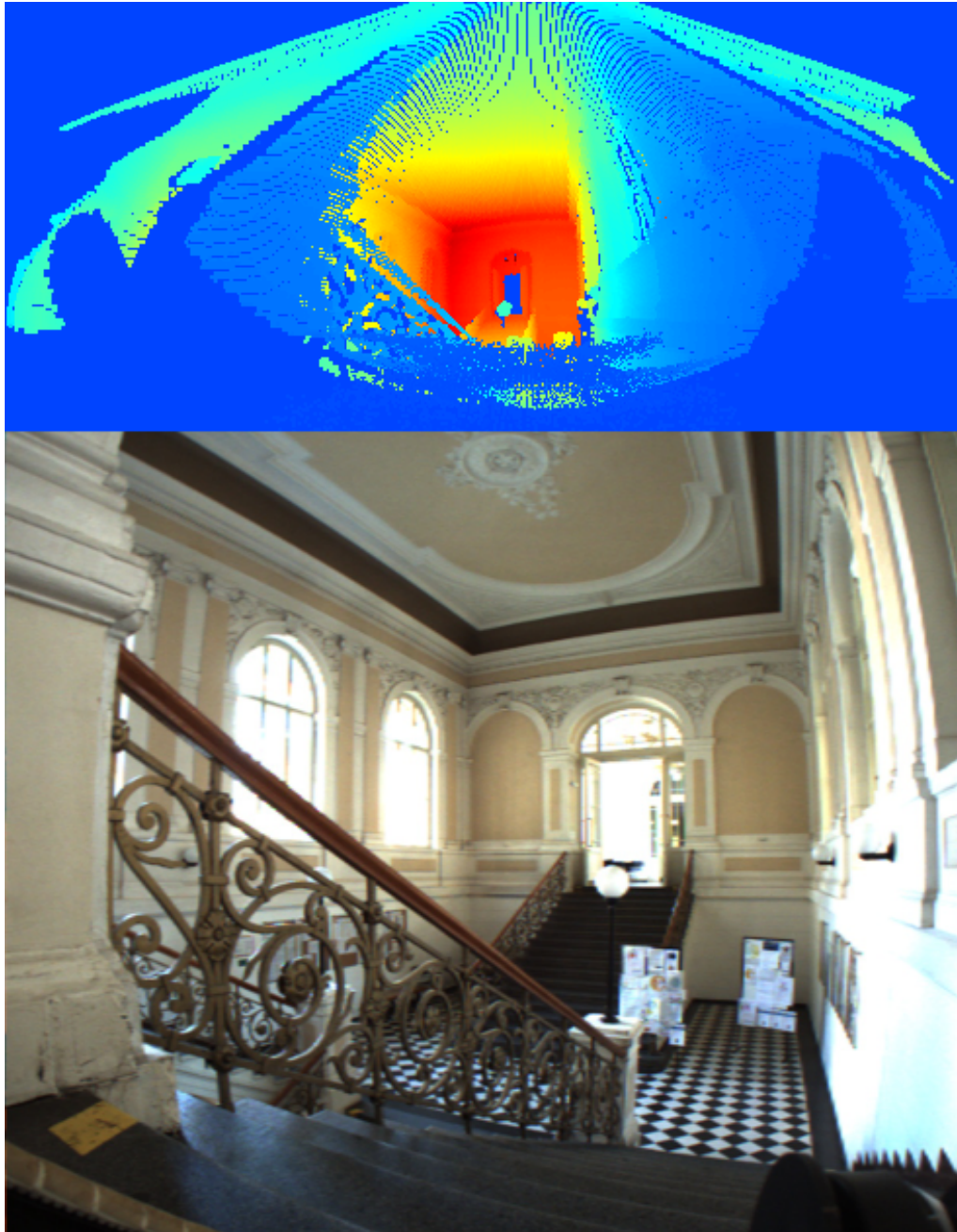


Figure 1.3. The top image is a raw range image and the bottom is RGB image from camera. It shows how difficult it is to find correspondences.

Chapter 2

Camera calibration

2.1 Camera geometry

A point in 3-dimensional Euclidean space is usually represented by a real vector (X, Y, Z) . In projective geometry, however, it is more convenient to use *homogeneous coordinates*. The same point (X, Y, Z) can be expressed in homogeneous coordinates as $(X, Y, Z, 1)$ or more generally (wX, wY, wZ, w) . Given a point in homogeneous coordinates, we can get the corresponding point in Cartesian coordinates by dividing it by w . There is an exception to this when $w = 0$ as in $(X, Y, Z, 0)$. These coordinates represent points at infinity that exist in projective space but not in Euclidean space.

2.1.1 Pinhole camera model

A camera is a mapping between the 3D world (object space) and a 2D image[2]. A number of different camera models exists but we are interested only in the *pinhole camera model*. The pinhole camera model describes a *central projection* where all of the rays meet in a single point C known as the *camera centre*.

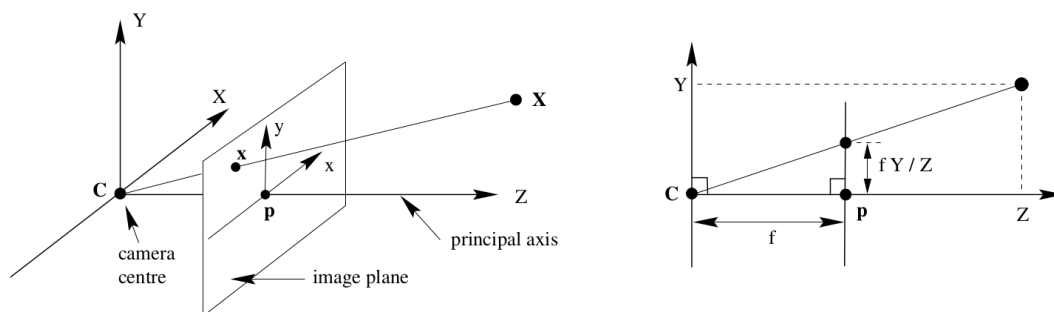


Figure 2.1. Pinhole camera geometry. C is the camera centre and p is the principal point. The camera centre is here placed at the coordinate origin. Note the image plane is placed in front of the camera centre. Illustration and caption taken from [2].

On the image above we can see the placement of a camera in its coordinate frame. The axis with the same orientation the camera is facing is called the *principal axis*. The point of intersection of the principal axis and the *image plane* is called the *principal point* and its 3D coordinates are $(0, 0, f)$ where f is the distance between the image plane and the camera centre. Often the principal point is not the origin of coordinates in the image plane. Instead the top-left corner of the image is the origin and the principal point is in the middle of the image at some coordinates (p_x, p_y) . Also because the y axis is usually pointed downwards in most images, the y and x axes in the object space are also inverted. By similar triangles we can see that the mapping between the object space and the image plane in Cartesian coordinates is:

$$(X, Y, Z)^T \mapsto \left(\frac{fX}{Z} + p_x, \frac{fY}{Z} + p_y \right)^T \quad (1)$$

This is not a linear mapping. However, the same mapping can be expressed as linear mapping in homogeneous coordinates using the matrix multiplication:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} f & p_x & 0 \\ & f & p_y \\ & & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2)$$

$$\mathbf{K} = \begin{pmatrix} f & p_x \\ & f & p_y \\ & & 1 \end{pmatrix} \quad (3)$$

The matrix \mathbf{K} is called the *camera calibration matrix*[2].

Usually, the 3D points are located in a different coordinate frame known as the *world coordinate frame*. In our case, world coordinate frame is the frame of the point cloud. By using homogeneous coordinates, affine transformation from the world coordinate frame to the camera coordinate frame can be expressed in terms of matrix multiplication:

$$\mathbf{X}_{\text{cam}} = \begin{pmatrix} \mathbf{R} & -\mathbf{R}\tilde{\mathbf{C}} \\ 0 & 1 \end{pmatrix} \mathbf{X}_{\text{world}} \quad (4)$$

$\mathbf{X}_{\text{world}}$ is a 3D point in the world coordinate frame and \mathbf{X}_{cam} is a 3D point in the camera coordinate frame. \mathbf{R} is a 3×3 rotation matrix and $\tilde{\mathbf{C}}$ is the position of the camera centre in the world coordinate frame. The camera calibration matrix can be multiplied with the frame transformation matrix to obtain a projection matrix from the world coordinate frame:

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \quad (5)$$

$$\mathbf{t} = -\mathbf{R}\tilde{\mathbf{C}}$$

The parameters contained in \mathbf{K} are called the *internal*¹⁾ camera parameters, or the *internal orientation* of the camera. The parameters of \mathbf{R} and $\tilde{\mathbf{C}}$ which relate the camera orientation and position to a world coordinate system are called the *external* parameters or the *exterior orientation*[2]. Our application estimates the external parameters as they relate the camera's position to the position of the LIDAR.

¹⁾ The terms “intrinsic” and “internal” are both used to denote the same parameters. They can typically be used interchangeably as there is no danger of confusion.

2.1.2 Non-linear distortion

Real lenses often do not behave exactly as the pinhole camera model. Due to imperfections in lens manufacturing some deviations to the imaging process are introduced. The most significant deviation is usually the *radial distortion*. This error tends to be more significant in lenses with wider field of view. Another usual form of distortion is *tangential distortion* which is caused by misalignment of the physical elements in the whole camera. Brown's distortion model also called *Plumb Bob* model is a 5-parameter model often used for the correction. It requires three parameters to model the radial distortion and two for the tangential distortion. More information about the Plumb Bob model can be found in the original article[6]. After the correction the camera again acts as a linear device.

Removing the distortion from the image is called *rectification*. The process can also be reversed and the point projected by the pinhole camera model can be unrectified by applying the distortion function on it. This is useful when we want to obtain the correct pixel coordinates in the original image.

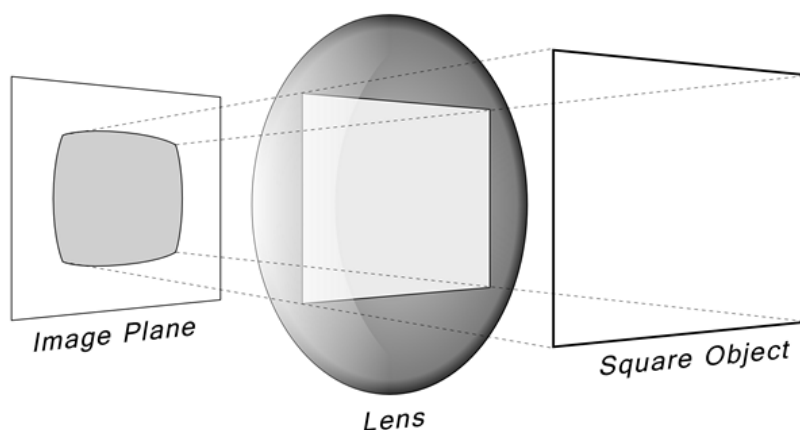


Figure 2.2. Radial distortion demonstrated[7].

2.2 Pose estimation and PnP

The procedure of extrinsic calibration of a camera is often called *pose estimation*. When the input is a set of correspondences $\mathbf{X}_i \leftrightarrow \mathbf{x}_i$ between the world points and the image points and the intrinsic camera calibration is known, the problem is known as *Perspective-n-Point* problem or simply *PnP*. Many different algorithms have been developed both iterative and non-iterative. They often offer different trade-offs between speed, precision, robustness and convergence. Different methods also behave differently when the correspondences are close to coplanar or collinear configuration.

2.2.1 Direct Linear Transformation algorithm

One method that can be used for pose estimation is called the *Direct Linear Transformation (DLT) algorithm*[2]. This algorithm finds the P matrix from the set of similarity relations $\mathbf{x}_k \propto P\mathbf{X}_k$. Camera translation can be found in the fourth column of P and the rotation can be obtained using the RQ decomposition of the left 3×3 sub-matrix.

This method minimizes the *algebraic error*. The meaning of the algebraic error and its relationship to the reprojection error is fully described in[2].

2.2.2 Reprojection error minimization

Reprojection error also sometimes called geometric error is defined in the following way:

$$\sum_i d(\mathbf{x}_i, P\mathbf{X}_i)^2 \quad (6)$$

It is the sum of squared Euclidean distances between the image points and the projected world points using the camera model. Because the correspondences are defined in terms of homogeneous coordinates, the squared Euclidean distance between two points \mathbf{x} and $\hat{\mathbf{x}}$ in 2D is computed using the formula:

$$d(\mathbf{x}, \hat{\mathbf{x}})^2 = \left\| \frac{1}{x_3} \cdot \mathbf{x} - \frac{1}{\hat{x}_3} \cdot \hat{\mathbf{x}} \right\|_2^2 \quad (7)$$

Levenberg-Marquardt iterative method is often employed for minimizing the reprojection error. Known parameters (in our case matrix K) can be enforced and the P matrix can be explicitly computed in terms of the remaining parameters. Since the iterative method requires an initial guess, one is either provided by the operator as a launch parameter to the application or computed using the DLT algorithm. We used the implementation provided by the OpenCV library in the function `cv::solvePnP`.

2.2.3 EPnP

We also used the EPnP method that is implemented in OpenCV. EPnP is a non-iterative method with $O(n)$ complexity. It is supposed to be a reliable method that is robust even when the correspondences are arranged in coplanar configuration. It can also be used as an initial guess for the iterative method. Full description of EPnP can be found in the paper[8].

2.3 Point cloud visualizations

Data measured with the laser scanner needed to be properly visualized in order to help the operator recognize correspondences in them. The input to the application is a *point cloud* of points measured by the laser. Point cloud is simply a list of 3D points measured with respect to some specified coordinate frame. The 3D points do not have any particular order or identification number. A range image is a typical visualization of a point cloud. This is done by placing a standard pinhole camera model into the origin of the point cloud and projecting all the points through it. As a result the visible points are ordered in a 2D image matrix. Instead of intensity or color, the point distance is assigned as the image value. We used the `pcl::RangeImage` class from the PCL library for this. In contrast to the usual camera images, in range images the principal axis is the x axis.

The range image can be visualized using the range information of the points. This is done by linearly mapping the ranges from the $\langle min_range; max_range \rangle$ interval into the $\langle 0; 255 \rangle$ interval. The color is then assigned according to some color map. In our case it is the *jet* colormap. This is necessary because the human eye is not good at noticing details in grey images.

It can be seen that there are not many details recognizable. Because of that, we also used the following visualization methods.

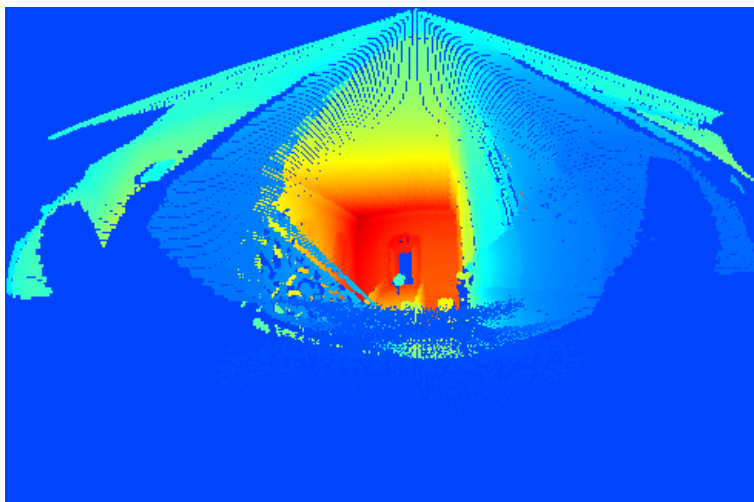


Figure 2.3. A range image. The warmer the color the farther the point is from the laser.

2.3.1 Directional images

Standard edge detectors are not of much use when working with range data. Lots of edges and corners show only a subtle difference in range with respect to their surroundings. Instead we tried to highlight the edges and corners by measuring the direction changes of the surface. This idea was presented in the paper[4] as *bearing angle* images. Our approach is based on the bearing angle images but we use a different formula to compute the angle. We do not measure the angle between the surface and the ray of the laser but measure the angle between the surface and the image plane. The reason is that the color does not change on large flat surfaces. For each point in the range image we measure the angle by using it's neighbour. We pick neighbours in horizontal, vertical, diagonal and the opposite diagonal directions and as a result create four different images. Each of these images is more sensitive to the edges oriented in different directions e.g. horizontal image is most sensitive to the vertical edges. The angle between two points A and B is measured using the following formula:

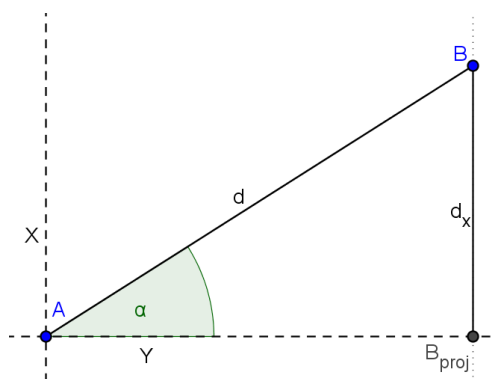


Figure 2.4. Geometry of the directional images. Points exist in three dimensions. In range images the x axis is facing forward.

$$\alpha = \arcsin \frac{B_x - A_x}{|B - A|} \quad (8)$$

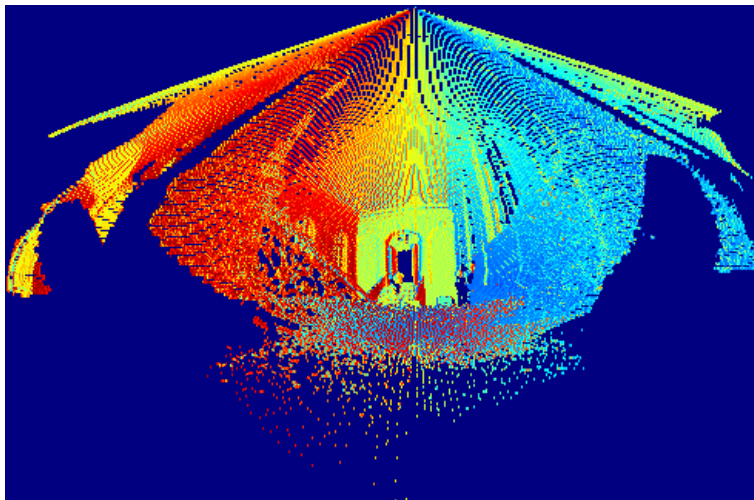


Figure 2.5. Visualization of a range image using the horizontal directional image. Notice the enhanced level of detail (door, windows) compared to the raw range image, see Figure 2.3.

2.4 Local correction

Because it is sometimes difficult for the operator to mark the exact pixel in the range image and small error can cause an entirely different world point to be used, we attempted to make the process more robust. We came up with the following algorithm for local correction of the correspondences:

- 1) Sorts the correspondences.
- 2) Corrects the correspondences one by one. Terminates if the maximal number of total calibrations performed is exceeded or there is no more work to do.
- 3) Returns the corrected correspondences.

At first the correspondences are sorted in descending order according to the range variance of the pixels in their neighbourhood. The neighbourhood N_i is a square of a reasonable size around the i -th correspondence in the range image. We chose a square with 9 pixel side length making it contain 81 pixels. The N_i symbol thus represents 81 values surrounding the clicked one. The range variance of i -th correspondence is determined using the classic formula:

$$\text{Var}(N_i) = E(N_i^2) - E(N_i)^2 \quad (9)$$

The correspondences are sorted because now they are going to be corrected in that order. The correspondences with larger range variance tend to be more dangerous for the calibration and therefore are corrected first.

The correction of each correspondence is achieved by trying the calibration for every point in it's neighbourhood instead of the clicked one. The point with the lowest reprojection error in the calibration is picked as the correct one. The previously corrected correspondences are used when correcting the current correspondence. This makes sense because if we assume that the corrected location is more accurate than the original one then there is no reason to use the original one.

Because the calibration itself is really fast there is no problem in running it hundreds of times.

2.5 RANSAC

It is possible that some of the created correspondences are just wrong. This might happen either because the operator did not really recognize the objects he was matching or he might have made some other type of error e.g. forgot to assign one of the created correspondences, unintentionally switched two correspondences between each other in one image. We used the RANSAC[9] scheme to help with identification of these *outliers*. RANSAC is used to find the subset of the correspondences (*inliers*) that fit the model. The model in our case is the camera's calibration and the correspondence fits the model if it's reprojection error is lesser than some predefined threshold. When the set of inliers is obtained the calibration is computed from it using the iterative method. In the result, outliers can be quickly identified by their large reprojection error and the operator might choose to fix them or delete them. The main difference from the local correction is that RANSAC does not adjust the correspondences in any way. It only finds the subset of correspondences that seem to be "right".

The first image in Figure 2.6 shows the correspondences created in the point cloud. In the second image the correspondences in the camera image are marked in accordance with the point cloud. The red circles show the projection of the correspondences into the image using the calibration (without RANSAC) obtained from these correspondences. In the third image the highlighted correspondence is wrong. The normal calibration without RANSAC is affected by it. In the fourth image RANSAC was used. The calibration ignores the wrong correspondence. It can be easily recognized by large reprojection error and fixed.

In more detail, our implementation of the RANSAC algorithm works in the following way:

- 1) Randomly selects n correspondences from the set of all correspondences.
- 2) Determines the extrinsic calibration from these n correspondences.
- 3) Every correspondence with smaller reprojection error than some predefined threshold t is put into the set of inliers S_i
- 4) Increases the iteration number and goes back to (1).
- 5) Terminates if the number of iterations is greater than some N . The calibration is estimated from the largest set S_i

In our case, $n = 4$ because it's the minimal number of correspondences needed for calibration and RANSAC uses as small initial set as feasible. Threshold t is equal to 1 percent of the camera image width.

2.6 Point cloud coloring

Point cloud coloring is a utility that can be used to validate the calibration. It assigns a color to each point in a point cloud that is visible by the camera. The algorithm is simple:

At first, the point is transformed into the camera's coordinate frame. After that it has to be checked that the point is located in front of the camera. This is done by simply checking that the z coordinate is greater than 0. If the point is in front of the camera it is projected onto the image plane using the pinhole camera model. Then it has to be

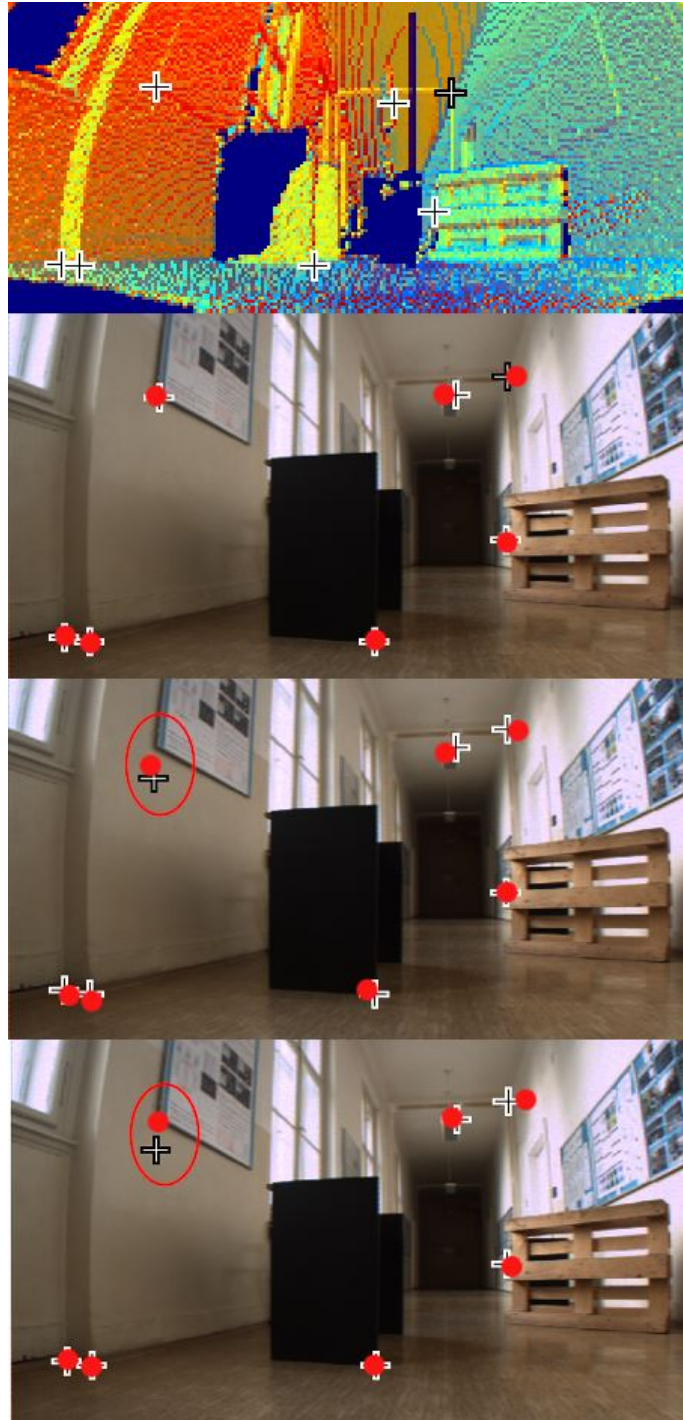


Figure 2.6. RANSAC example.

distorted according to the camera's distortion coefficients. If the distorted point lies in the image, the point in the point cloud is assigned the color of the corresponding pixel.

Analogically, every point in the range image can be also colored using the same algorithm.

Chapter 3

Architecture

In this chapter we describe the architecture of the application and explain the design decisions we made. We start with the high-level overview of the architecture and then describe the details when we feel it is necessary. Reading the user manual first may help in understanding. User manual is mainly meant for users while this chapter is mainly for programmers who might want to accommodate the software to their needs.

We implemented the application as a ROS[1] package. At first we used the version *Fuerte Turtle* but later we also added support for newer versions of ROS using the new build system: catkin. The whole codebase is written in C++03. We did not use a newer C++ standard, such as C++11, because it was not supported[10] by ROS at the time the application was written. Apart from ROS we used the following libraries: Boost[11], OpenCV[12], PCL[13] and Qt[14]. All of those libraries were distributed together with ROS. We also heavily used the ROS *tf* package[15]. Tf is a decentralized system that keeps track of all the coordinate frames in the robotic system.

The priority was for the application to be robust and easily extensible to some extent. On the other hand we strived to keep the design simple and avoided introducing unnecessary abstraction. There is no time-critical context in the application so we did not need to specifically optimize for performance. When we had to choose between speed and code clarity, we almost always opted for clarity.

3.1 Package structure

The package structure differs in different ROS versions. The following package structure is the one used in newer versions of ROS using the catkin build system. In the older versions of ROS the package structure is almost similar. The main difference is that the `package.xml` file is called `manifest.xml` and the structure of the `CMakeLists.txt` file is totally different.

- **[export]**
The application exports data into this directory.
- **[images]**
Images used by the application in the GUI are stored here.
- **[include]**
All of the header files are located here.
- **[launch]**
Different launch configurations are stored here.
- **[msg]**
Directory for the custom ROS messages. `CalibrateCamera` message is stored here.
- **[src]**
All of the source files are located here.

- **CMakeLists.txt**
This file controls the build process.
- **package.xml**
Every ROS package must contain this file. Information about the package such as its name, author and license under which it can be distributed has to be specified here. Also the dependencies on other ROS packages are listed here.
- **resources.qrc**
Resource-definition file describing resources used by the applications GUI. Paths to images are defined here.

The package consists of the following three ROS nodes:

- **cloud_camera_autocalibration**
This is the main node. When we mention “the application” in this text we are talking about this node.
- **keyboard_teleop**
Launched together with the application. This node allows the user to control the application from the terminal. It is done by publishing the `CalibrateCamera` messages onto the applications topic.
- **cloud_coloring**
Separate, simple to use, helper utility that colors the point clouds by projecting their points onto camera images. It can be used to validate the results of the calibration.

3.2 Top-level architecture

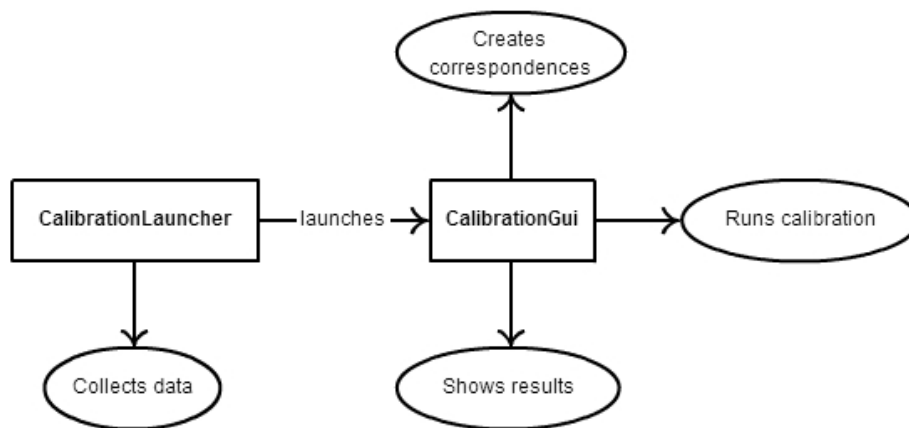


Figure 3.1. Top-level architecture overview.

At the top level the application consists of the `CalibrationLauncher` and the `CalibrationGui` classes. The sole purpose of the `CalibrationLauncher` is to collect all of the sensor data needed for the actual calibration and start the `CalibrationGui`.

The sensor data are not collected in the GUI mainly for historical reasons. At first we did not plan for the GUI to play the main part in the application. Our first design was inspired by the following pipeline:

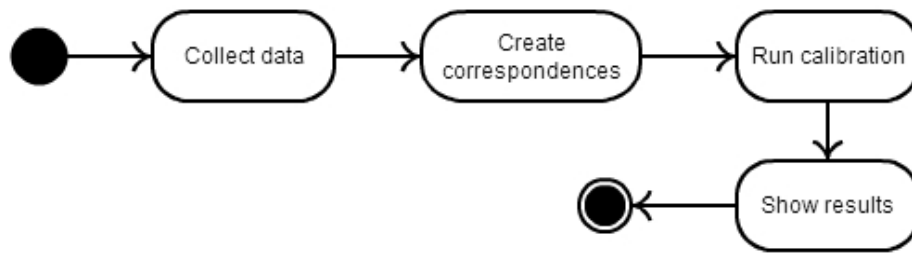


Figure 3.2. Old architecture design.

In this old design every activity was represented by its own top level class. The GUI was planned to be used only for the “create correspondences” step. At that stage we also thought that we would replace the GUI by some automatic correspondences creator entirely. As the design evolved we decided to move more responsibilities straight into the GUI. The data collecting class was however already written so we just renamed it to the `CalibrationLauncher` class.

Also it was not trivial to move the responsibilities of the `CalibrationLauncher` straight into the GUI because the application is single-threaded. The reason is that the ROS message loop has to keep spinning to collect data from the camera and the laser scanner. Unfortunately, the GUI contains its own message loop which blocks the ROS message loop.

Retrospectively, the application should have been made multi-threaded and GUI only. The cost of such change, however, compared to its value was not worth it for us.

3.3 Calibration launcher

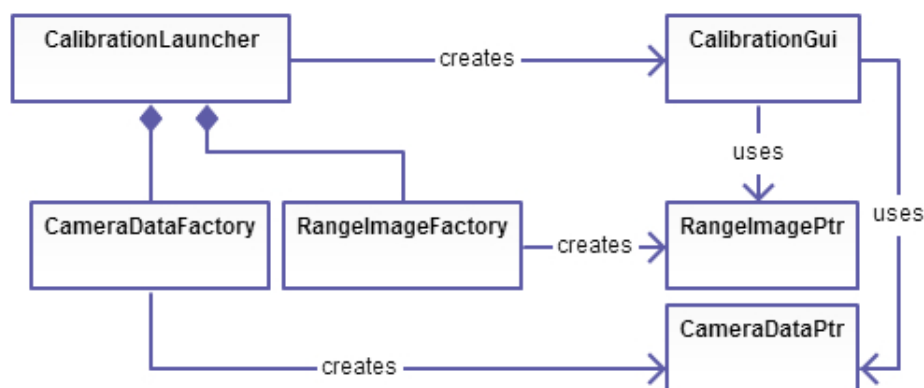


Figure 3.3. Calibration launcher architecture.

The calibration launcher acquires the range image, the camera data and then starts the GUI. `CameraData` is a container class that contains the camera image, camera info, tf transform from the cloud frame to the parent frame and the initial calibration guess if available. Range image is an instance of the `pcl::RangeImage` class. Each point of this range image contains the information about the 3D point it represents.

The `CameraDataFactory` class is used to create the camera data. This class subscribes the necessary ROS topics in the constructor. The actual `CameraData` object is created by its method `createCameraData()`. This method throws exceptions if the camera data are not available or can not be created for some other reason. Analogically, the `RangeImageFactory` class creates the range images from the subscribed point clouds.

3.4 Graphical user interface

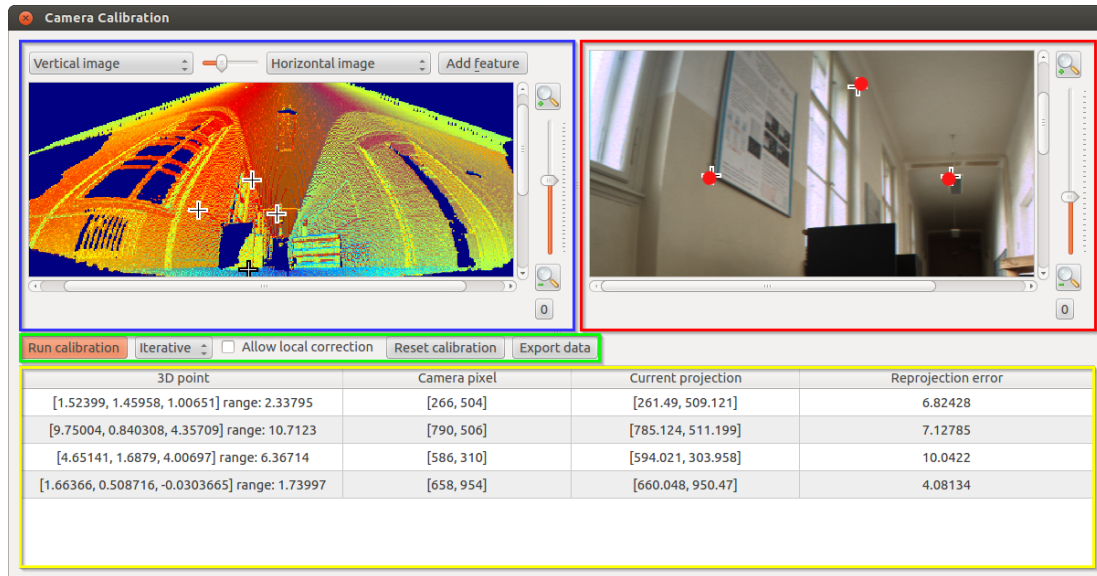


Figure 3.4. Screenshot of the GUI with major parts highlighted. **Blue:** Range image scene view. **Red:** Camera image scene view. **Green:** Calibration panel. **Yellow:** Correspondences table.

We built the GUI on the Qt framework. Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks[16].

Large part of the widget tree is defined in the `CalibrationGui.ui` file. To edit this file it is best to use Qt Creator[17] which contains a designer for these files. However, all of the widgets inside the scene views are defined directly in the source code of those views.

The class `CalibrationGui` itself does not contain much logic. It contains other parts of the GUI, holds them together and provides callbacks for the widgets in the green box.

3.4.1 Scene views and correspondences

Scene views are custom widgets used to display the range image and the camera image comfortably. There are two scene views present within the GUI, see Figure 3.4. The one with the red border displays the camera image and it is an object of class `SceneView`. The other one with the blue border displays the range image and it is an object of class `RangeView` which extends the `SceneView` with additional functionality regarding mostly the range image visualizations.

Apart from scrollbars and other controllers, scene views contain a `QGraphicsScene` object which acts as a surface for managing 2D `QGraphicsItems`. Each `QGraphicsScene` contains its own 2D coordinate system where the `QGraphicsItems` live. We have based this coordinate system on the underlying images in those scenes. That means that e.g. the pixel located at the coordinates `[50; 100]` in the image is also located at the same coordinates in the scene. We created two custom `QGraphicsItem` types:

- **FeatureMarker**

Feature markers can be directly manipulated by the operator. Their movement is restricted to the image on the background. They are used to assign the correspondences between the range image and the camera image. They can only exist in pairs. That means that each marker has its own sibling on the other image.

- **ProjectionMarker**

Projection markers can not be directly manipulated by the operator. They exist only in the camera scene view and are connected by the `CorrespondencesModel` to a parent feature marker in the range image scene view. Their position is determined by the projection of the corresponding 3D point to the camera image. To determine the position of the projection, these markers have knowledge of the `CameraData`. The class `image_geometry::pinhole_camera_model` is used inside the projection markers to compute the correct projection.

The class `CorrespondencesModel` keeps track of all the correspondences and acts as a central authority. It is responsible for connecting the markers with their relatives. It also acts as a table model[18] for the correspondence table (yellow border on the image).

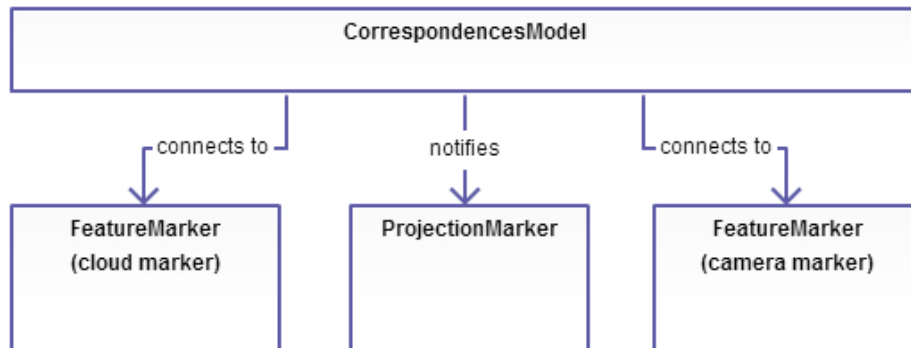


Figure 3.5. Relationships between the different marker types.

When the button `Add feature` in the `RangeView` is clicked, the `addCorrespondence` slot method of the correspondences model is called. The correspondences model itself then handles the creation of the markers and establishment of the necessary connections. Analogously, when the `delete` button is pressed, the corresponding slot method is called in the correspondences model and it handles the destruction of all the markers and other data relating to the specific correspondence.

■ 3.4.2 Range image visualizations

The code responsible for the creation of the range image visualizations is located in the `include/range_visualisations` folder. Only the classes `RangeVisualisationsFactory` and `CameraColoredCloudImageFactory` are used publicly.

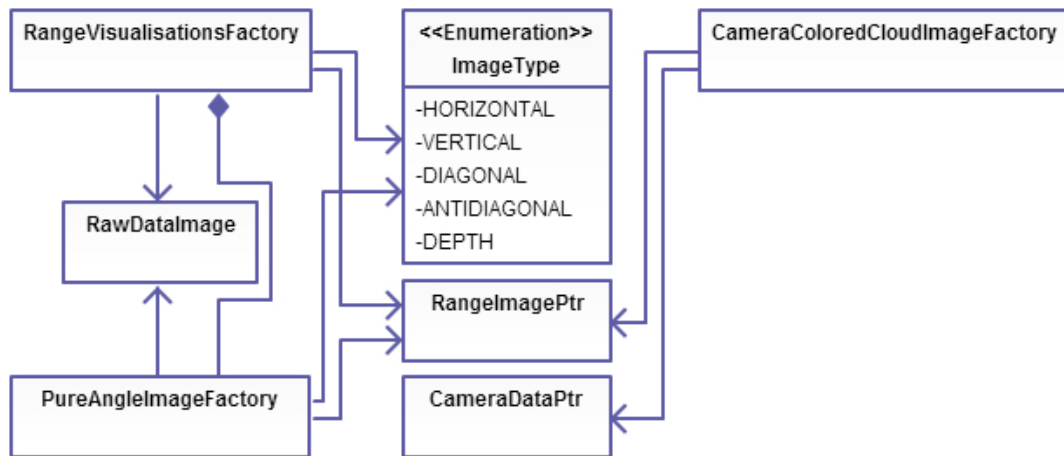


Figure 3.6. Visualization factories and their dependencies.

At first, we did not consider creating the camera colored images. When we got the idea of implementing them, the `RangeVisualisationsFactory` was already made and was using the `ImageType` enum to decide which image type it will create. Unfortunately, the creation of the camera colored images required different parameters than all the other images. For that reason we did not add the camera colored images into the enum. After that decision, it seemed cleaner to us to create a new factory just for them. This factory projects each point in the range image to the camera image using the pinhole camera model. If the projected point is in the image frame then the corresponding pixel color is used. If the point is outside the image frame then gray color is used. If there is no 3D point associated with the pixel in the range image then black color is used.

When creating the directional images 2.3.1, the `RangeVisualisationsFactory` delegates most of the work to the `PureAngleImageFactory`. This factory creates the `RawDataImages`. Those images do not have the color information in them but store the actual angle instead. `RangeVisualisationsFactory` then assigns the color by mapping the raw data into the range from 0 to 255 and applying a colormap.

■ 3.4.3 Calibration

We use the `solvePnP` function available in the OpenCV library to estimate the extrinsic camera parameters. If the operator selected RANSAC scheme in the dropdown list we use the `solvePnPRansac` method to discover the inliers and then use the `solvePnP` method with the inliers only. This logic is encapsulated inside the `ExtrinsicParametersEstimator` class. This class transforms the data into the format `solvePnP` and `solvePnPRansac` functions can use. After it receives the results from `solvePnP` it transforms them into the `tf` format. However, this class does not know anything about the actual `tf` transform tree. For that reason the result is relative to the *cloud frame* and not the *parent frame*.

Local correction is implemented inside the `IterativeParametersEstimator` class. The class is a wrapper for the `ExtrinsicParametersEstimator` that runs the calibration multiple times with slightly randomized correspondences. The number of iterations is an input parameter. If 0 is supplied as the number of iterations this class acts exactly

the same as the `ExtrinsicParametersEstimator`. For that reason the GUI callback running the calibration uses only the `IterativeParametersEstimator` class.

The results are shown using the `CalibrationExport` class. First, this class changes the parent of the result transform to the *parent frame*. After that it displays the dialog with the calibration output. If the `debugFrame` launch parameter is specified it also starts a *static transform publisher*[19] in a separate process that publishes a tf frame with the calibration result. This process shares the stdin with the application's process so it can be shut down using SIGINT together with the application.

Finally, `CorrespondencesModel` class contains a method `invalidateProjections()`. After the new result is stored, this method has to be called to update the positions of the `ProjectionMarkers` according to the new calibration.

Chapter 4

User manual

The purpose of this chapter is to be the complete reference of the applications abilities from the user's point of view. The sections are ordered according to their chronological position while using the application to create an extrinsic camera calibration. It is expected that the user has at least some basic knowledge of ROS[1].

4.1 Applications requirements

Before installing the application it is necessary to have the following system configuration:

- **Ubuntu 12.04 “Precise Pangolin” LTS**

The application was developed and tested on this system. It is possible that it will work without any problems on some of the older distributions (10.04, 11.10) and also newer distributions of the Ubuntu operating system (or any other operating system ROS supports) but it has not been tested.

- **ROS Fuerte Turtle or newer**

The application was developed on *ROS Fuerte Turtle*. It has also been verified that the application is compatible with newer distributions of ROS, namely *ROS Groovy Galapagos* and *ROS Hydro Medusa*. Installation of ROS is described in it's official documentation[20].

4.2 Installation and build

The installation is slightly different depending on the user's ROS distribution. This is because newer versions of ROS use different package structure and build system.

4.2.1 ROS Fuerte Turtle

The contents of the `/rosbuild` folder need to be extracted somewhere into the *ROS package path*. It can be determined by running the following command in a shell:

```
echo $ROS_PACKAGE_PATH
```

After that it is just needed to run `rosmake` in the applications package.

4.2.2 Newer versions of ROS

The contents of the `/catkin` folder need to be extracted somewhere into the *ROS package path*. If everything is right the last folder of this path is going to be called `src`. After that it is needed to run `catkin_make` in the catkin workspace root.

4.3 Required ROS components

Before launching the application it is needed to start some other ROS components. At first it is necessary that the ROS server is running. ROS server can be started by executing the command:

```
roscore
```

After the server is started it has to be ensured that the following topics are published. The list of published topics can be displayed by the command:

```
rostopic list
```

The mandatory topics are:

- **/tf**
On this topic all of the coordinate frames are published.
- **/point_cloud_topic**
The name of this topic is not important (It can be set as a launch parameter.), but the type of the message has to be `sensor_msgs/PointCloud2`.
- **/camera_topic/image**
Messages have to be of type `sensor_msgs/Image`.
- **/camera_topic/camera_info**
Messages have to be of type `sensor_msgs/CameraInfo`.

Also there are some important tf transformations that have to or can be published. Their exact names are not important as they can be set as launch parameters to the application:

- **/cloud_frame**
This transform is mandatory and it should represent the transformation origin of the points in the point cloud.
- **/camera_parent**
Some transform has to be selected as the parent frame of the camera frame that is going to be created.
- **/calibration_guess**
This transform is optional and it can be used as an initial guess for the calibration. More about this can be found in the section about launch parameters.

4.3.1 Using Bag files

For testing purposes all of the prerequisites can be easily satisfied by running one of the provided bag files in the `/bags` folder. It is important that the `/clock` topic is published when using bags and the `/use_sim_time` parameter is set to `True`. Convenience bash scripts located in the `/bags` folder can be used to properly play the bag files or it can be done manually with commands similar to:

```
roslaunch <bag-to-play>.bag --clock
roscpp
roscppparam set /use_sim_time "True"
```

4.4 Launching the application

After all the prerequisites from the previous section are satisfied the application can be launched via a launch file using a command:

```
roslaunch <package> cloud_camera_autocalibration.launch
```

The file is located at `<package>/launch/cloud_camera_autocalibration.launch` and it contains several parameters that should be adjusted. The following is the description of the launch parameters and how they should be set with respect to one another.

4.4.1 Launch parameters

All of the parameters in the following list are mandatory.

- **cloudTopic**
This should be set to the name of the topic the cloud is published on.
- **cloudFrame**
This should be set to the name of the `/cloud_frame` transform.
- **cameraTopic**
This should be set to the name of the topic under which `camera_info` and `image` are published.
- **parentFrame**
This should be set to the name of the frame `/camera_parent`. The result of the calibration will have this frame as a parent.

The application can be provided some initial guess for the calibration. This can be done by providing one of the following parameters. If both parameters are provided then the initial guess is taken from the `initialTransformToCamera` parameter. If no calibration guess is provided then the application computes it's own guess using *Direct Linear Transform* algorithm.

- **cameraFrame**
This parameter should be used if some calibration guess already exists in the tf system.
- **initialTransformToCamera**
This parameter should be used if it is preferred to provide the initial guess explicitly or the initial guess is not available in tf. The format of this parameter is `x y z yaw pitch roll`. The position should be measured in meters and the rotation in radians. The parent of this transform is the `/cloud_frame` and not the `/parent_frame`!

The following parameters are optional:

- **debugFrame**
If this parameter is provided, the application automatically starts a tf publisher with calibration after each calibration attempt. The names of the published frames are created by concatenating this parameter with a timestamp from the time the calibration was run.
- **initialTransformToParent**
This parameter is almost always not necessary, because the transform can

be obtained from the `tf` in most cases. It follows the same format as the `initialTransformToCamera` parameter and should represent the transform from the `/cloud_frame` to the `/camera_parent`. Specifying the transform explicitly can be useful in situations when it is difficult to start the calibration procedure with the same `tf` configuration as when the image from the camera and the cloud were created. The reason for this difficulty can be e.g. the camera is on an arm that is constantly rotating.

4.4.2 Launching the GUI

After the application started it needs some time to collect the data from all of the sources. Usually it is necessary to wait for the point cloud because point clouds are published with the lowest frequency. When the point cloud is available the application outputs:

```
New pointcloud arrived.
```

Now the GUI can be launched by either pressing the `space` button or by sending `CalibrateCamera` message to the `/cloud_camera_autocalibration` topic. The latter option can be used e.g. from some external node to request recalibration of a camera automatically.

It is important to make sure that the robot doesn't change its position in between receiving the point cloud, the image and starting the GUI. If it is somehow difficult to achieve, the launch parameter `initialTransformToParent` might be useful.

4.5 Using the GUI to calibrate a camera

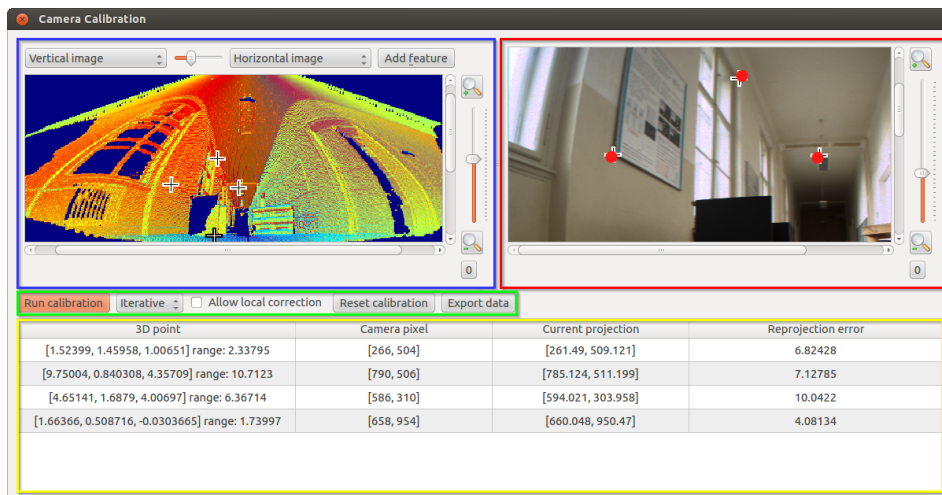


Figure 4.1. Screenshot of the whole GUI with highlighted blocks.

To calibrate a camera it is necessary to find *correspondences* between the point cloud and the camera. A correspondence is a pair of points (Further on those points are called *features*.) from the point cloud and the camera image that represent the same physical point. At least four correspondences are necessary to be able to run the calibration procedure but the more correspondences can be found the better.

The whole purpose of this GUI is to make finding the correspondences as easy and precise as possible. In the image above the GUI is logically divided into four blocks:

- **Point cloud block (blue)**
The point cloud is visualized in this block. Also the button to create new correspondences is present here.
- **Camera block (red)**
The image from the camera is shown here. The image features are modified here.
- **Calibration block (green)**
The calibration can configured and run from here.
- **Correspondences block (yellow)**
More detailed information about the current correspondences can be found here.

While the GUI is open the robot can move around freely. That is possible because all the transforms are requested at the time of the GUI start and the application doesn't perform any further requests to tf. The result is therefore only dependent on the transforms during the start of the GUI.

■ 4.5.1 Using different point cloud visualizations

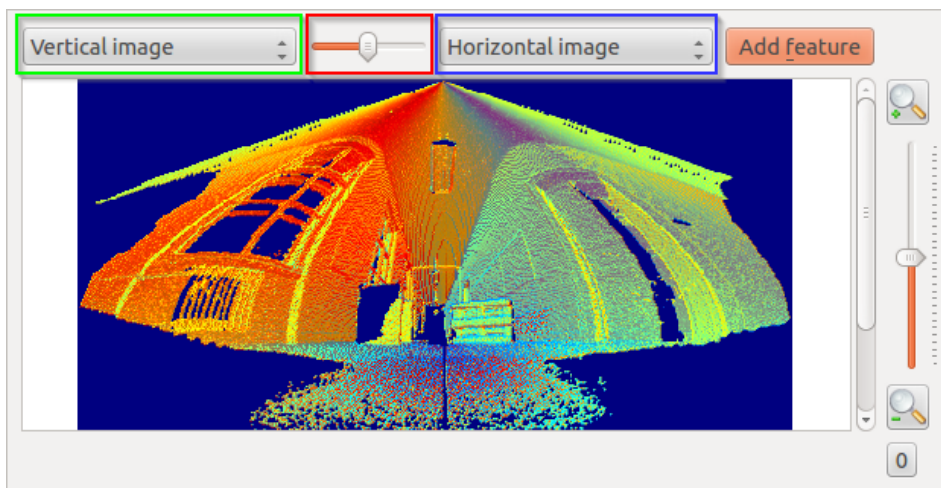


Figure 4.2. Screenshot of the point cloud block with highlighted visualization controls.

There are six visualization methods available. Their goal is to make physical objects such as edges or corners more recognizable. In different scenes different methods can be more useful.

The point cloud is always visualized as a *range image*. The position of the points in the range image stays the same irrespective of the visualization method. Different visualization methods, however, assign different colors to the points. The available visualization methods are:

- **Horizontal image**
Has similar effects as edge detection on visual images. Highlights edges perpendicular to the horizontal direction the most.
- **Vertical image**
Has similar effects as edge detection on visual images. Highlights edges perpendicular to the vertical direction the most.

- **Diagonal image**
Has similar effects as edge detection on visual images. Highlights diagonal edges the most.
- **Antidiagonal image**
Has similar effects as edge detection on visual images. Highlights the opposite diagonals than the *diagonal image* the most.
- **Range image**
Colors the points with respect to their range. This is the most natural visualization. It is, however, quite ineffective in highlighting the edges that lie on the same object.
- **Camera colored image**
This visualization serves different purpose than the previous ones. It can be used to validate the current camera calibration by coloring the points with the color of the pixels in the camera image they project to.

It is possible to choose a different visualization method for the foreground (drop-down list with the blue border) and background (drop-down list with the green border). The foreground and background can be then blended together with the slider (red border).

Blending is useful to highlight more edges at once by blending e.g. the horizontal and the vertical image together. It's main purpose, however, is to make the verification of the calibration easier. This can be done by setting the *camera colored image* as background and some other image as foreground. By blending them together one can spot the errors in the coloring and therefore the calibration.

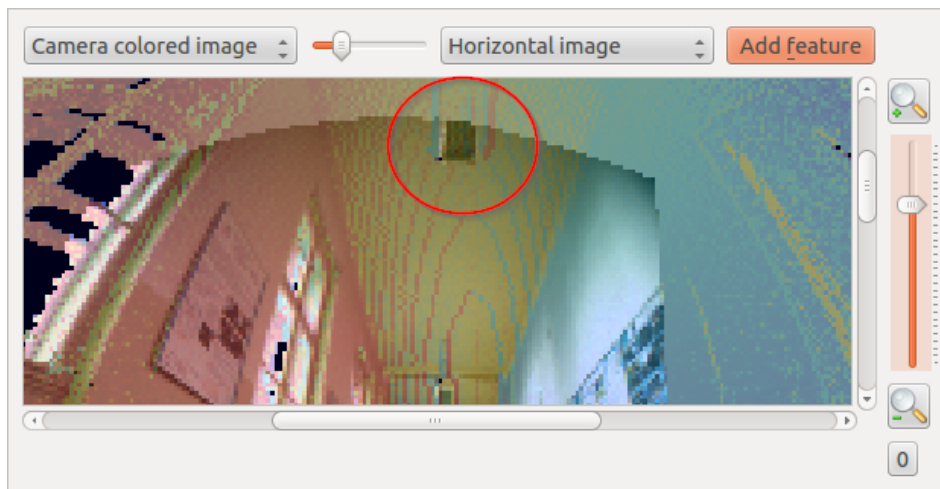



Figure 4.3. Camera colored point cloud blended with the horizontal image visualization.


On the image above a slightly trained eye can see that the coloring of the light at the ceiling is not correct (highlighted in the red circle).


■ 4.5.2 Creating correspondences

Correspondences are created with the button **Add feature**. Feature markers are then created in the center of the current viewport in both the range image and the camera image. They can be moved by dragging them with a mouse. The currently selected correspondence can be deleted with the `delete` key.

The following marker types are used:

- 

This marker is used for the currently selected correspondence.
- 

This marker is used for all of the other correspondences.
- 

This marker represents a projection of the feature marker in the range image to the camera image using the current calibration.

To achieve good calibration it is important to make the correspondences diverse. That means that the correspondences should not be collinear (aligned in a line) or coplanar (aligned in a plane). The more coplanar the correspondences are the more numerically unstable the calibration procedure is. Sometimes it is quite tricky to realize that the set of correspondences is coplanar:

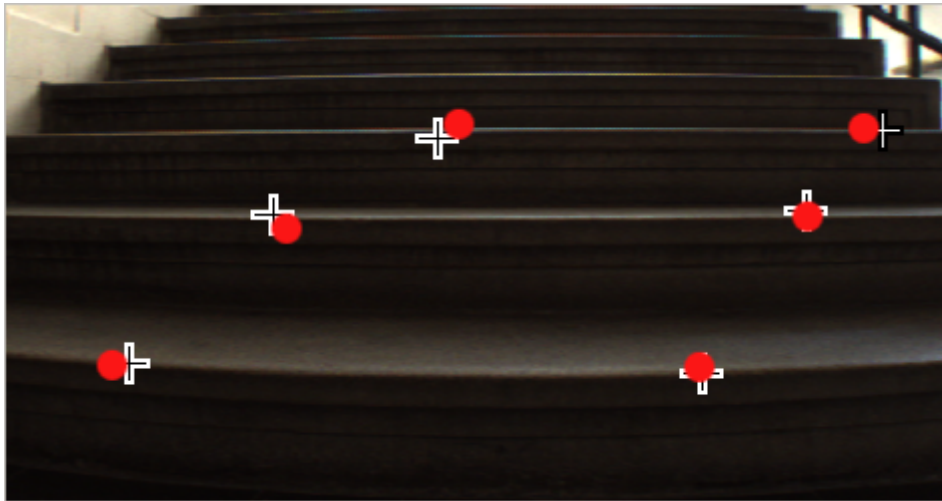


Figure 4.4. On this screenshot correspondences are found on a staircase. This is an example of coplanar correspondences that might not be easy to spot at first sight.

4.5.3 Running the calibration

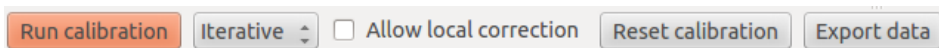


Figure 4.5. The calibration block.

After a sufficient number of correspondences is created the calibration procedure can be run with the button `Run calibration`. Next to this button there is a drop-down list where it is possible to select a calibration method:

- Iterative**

This is the recommended method. It needs an initial estimate and is reliable given the correspondences are diverse enough.
- EPnP**

Theoretically this method should be better but it didn't perform that well during experiments. It might have it's use in certain situations. Theoretically it should also

handle coplanar data better. It doesn't need initial estimate and it should perform global optimization.

▪ RANSAC

This method combines the classic iterative method with RANSAC scheme to recognize the correspondences that are clearly wrong. The result of this method is calibration from only the correspondences that fit the model. Unused correspondences can be recognized by large reprojection error and should be deleted or manually fixed.

Farther to the right there is a checkbox `Allow local correction`. If checked the calibration is run multiple times and each time the range features are slightly randomized. It might improve the calibration result when there is a lot of noise near the range features and it is hard to mark the correct spot exactly by hand.

Button `Reset calibration` resets the current calibration in the GUI to the initial guess provided on launch. This is useful when the calibration result is horribly wrong because succeeding calibration attempts always use the current calibration in the GUI as an initial guess.

Button `Export data` exports all the visualisations into the `<package>/export` folder. It also exports the data from the application as a MATLAB script. The files have the current timestamp in their name.

■ 4.5.4 Calibration results

When the calibration succeeds the results are shown in the following dialog window:

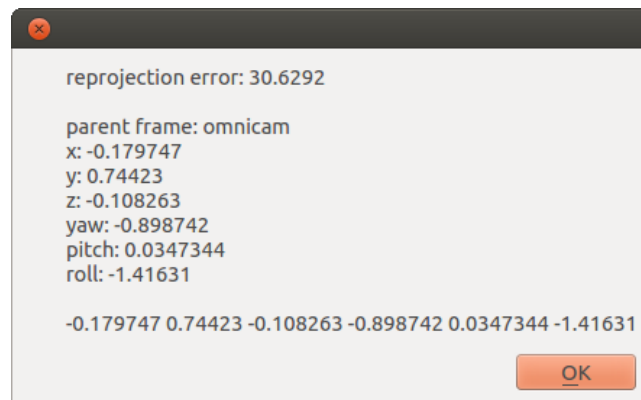


Figure 4.6. Calibration results.

At the top the average reprojection error for the correspondences is shown. In the most bottom line the same data as visible above are formatted so it can be copy-pasted as an argument to `static_transform_publisher` node[19]. Also a debug tf frame with the calibration is automatically published if it was set as a launch parameter. Meanwhile in the calibration GUI, projection markers adjust to the new calibration.

Now it is possible to keep refining the calibration by modifying the correspondences and running the calibration again. This procedure can be repeated as many times as it is necessary.

■ 4.6 Other useful tools

In this section some other useful tools are briefly described that will most likely be used together with the application.

■ 4.6.1 rviz

Rviz is a visualization tool for ROS. It is most likely to be used to directly visualize the resulting tf transforms. It is however much more powerful and can be used to visualize transformations, camera images, point clouds, and more. More information can be found in the official rviz user guide[21].

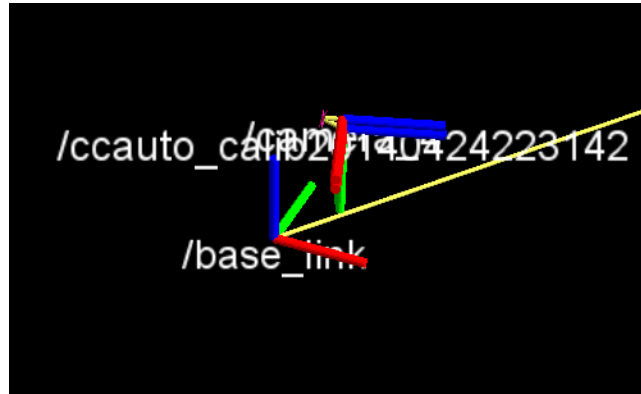


Figure 4.7. Screenshot from rviz visualizing the result of calibration.

The text on the image above is difficult to read because the current camera transform and the result of the calibration are so close together that they overlap.

■ 4.6.2 Cloud coloring

Cloud coloring is a node in the application package that can be used to color the points in the point cloud with the color of a pixel they project on. It's main purpose is to help with manually validating the calibration. The principle is similar to the *Camera colored image* range image visualization.

The advantage here is that the color is associated directly with the point cloud and therefore it can be used in other ROS nodes. Most importantly it can be used in rviz to visualize the colored point cloud in 3D.

Before launching the node it is necessary to set launch parameters in the launch file located at `<package>/launch/cloud_coloring.launch` to the correct values. The parameters are:

- **cloudTopic**
This should be set to the name of the topic the cloud is published on.
- **cloudFrame**
This should be set to the name of the base cloud transform.
- **cameraTopic**
This should be set to the name of the topic under which `camera_info` and `image` are published.
- **cameraFrame**
This should be set to the name of the camera transform.
- **outputTopic**
This specifies the name of the topic the node should publish the cloud on.

Now it is possible to start the node with the command:

```
roslaunch <package> cloud_coloring.launch
```

To visualize a colored point cloud in rviz it is necessary to add a display of type `PointCloud2` and set the `Topic` property to the `outputTopic` parameter. Then it is necessary to set the `Color Transformer` property to the `RGB8` value.

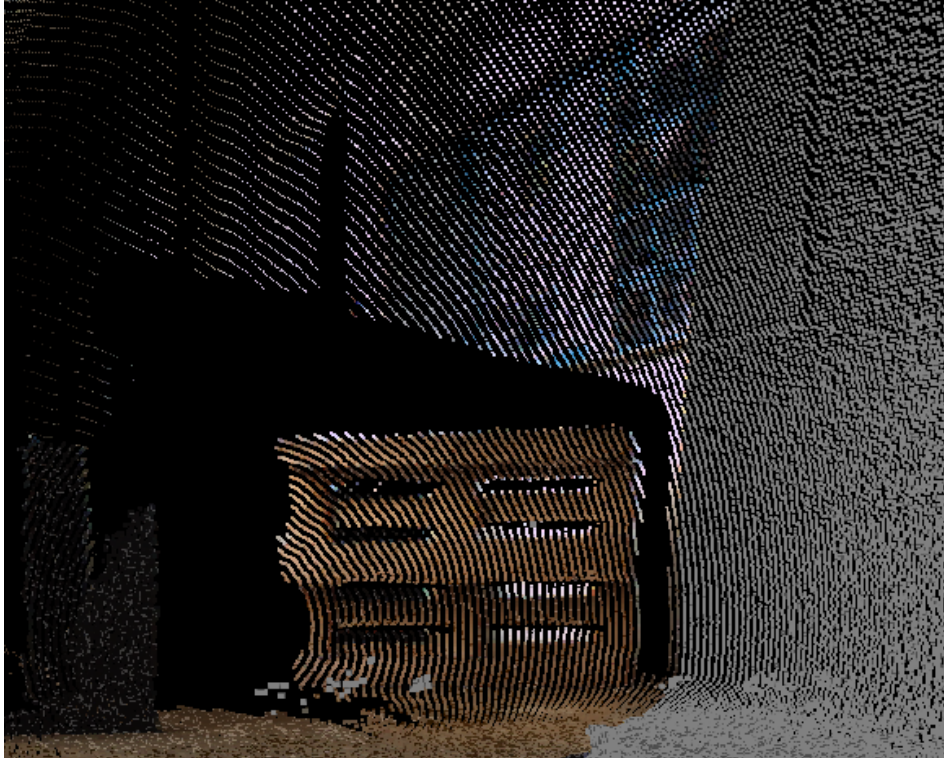


Figure 4.8. Result of the cloud coloring visualized in rviz.

Chapter 5

Experiments

This chapter contains the summary of the experiments. We test the application in five scenarios of various difficulty. In all but one of these experiments we used an already-calibrated camera fixed on the robot. The intrinsic calibration of this camera had a very high quality so it should not have affected the extrinsic calibration performed by the application in any negative way. On the other hand the extrinsic calibration was hand-crafted, essentially by measuring the physical location of the camera on the robot. This was not good enough to be used as a ground truth for the camera's position.

Since the operator performs the most crucial part of the calibration we had decided to test how much is the calibration's result dependent on the operator's experience with the application. We did test this by having a second operator also perform the calibration. This operator had no previous experience with the application and had only read the user manual prior to the experiments.

The reprojection error is the only purely objective measure of the calibration's quality. However, because the original calibration was not good enough, the reprojection error could not be accurately measured for every point available in the point cloud. Instead it was measured only for the correspondences provided by the operator. This did not make the reprojection error useless but some other subjective criteria were also used to evaluate the calibration.

Because we had a quite solid extrinsic calibration available for the camera, we could compare the application's result with it. Comparing the differences between individual components of the transforms gave us a valuable insight regarding the admissibility of the application's results.

Point cloud coloring was also used to subjectively evaluate the calibration results. By coloring the cloud one can see the projections of all the points in the cloud at once. By looking at miscolored points one can reveal the errors in the calibration.

The difficulty of the scenarios is influenced by the following issues:

- **Number of recognizable correspondences**

In each scenario only a relatively small number of correspondences is recognizable. In reasonable scenarios it should be possible to recognize at least five correspondences.

- **Variability of the correspondences**

It is not enough to recognize a large number of correspondences but they also have to be variable enough. They should not be coplanar (lie in the same plane) and should also lie in different segments of the image. This is often a problem for scenarios in exteriors.

- **Camera alignment**

The more the camera is misaligned with the laser the more difficult it is to find correspondences.

- **Intrinsic calibration quality**

If the intrinsic calibration of the camera is poor the whole calibration procedure is unreliable.

The scenarios differ mostly in the number of recognizable correspondences and their variability. Only in the last scenario the camera alignment is a bit different. Still the difference is not that severe to cause problems. We did not experiment with the effects of a bad intrinsic calibration on the application's result.

In each scenario we tried to create as many precise correspondences as possible. After that we ran the calibration with every calibration method available in the application. Then we picked the promising calibration results for further study. We compared the transforms against the original transform and also inspected the colored point clouds for miscolored points.

5.1 Scenario 1 - Corridor

In this indoor scenario objects are artificially arranged in a way that produces lots of recognizable correspondences. Camera resolution is 1616×1232 pixels.



Figure 5.1. Image used for the calibration.



Figure 5.2. 13 correspondences used for the calibration.

Calibration method	Avg. reprojection error (in pixels)	Max. reprojection error
Original calibration	14.3	27.5
Iterative	8.7	16.2
EPnP	13.1	33.0
Iterative with LC¹⁾	8.5	18.5
EPnP with LC	20.8	45.7

Table 5.1. Calibration results in scenario 1.

The reprojection error was always measured with respect to the user defined correspondences. This approach discriminates the methods using the local correction. The reprojection error would be lower if it was measured with respect to the corrected features when the local correction was used. *Iterative* method results and *iterative with*

¹⁾ LC stands for local correction

local correction method results were both promising and we decided to further inspect them.

5.1.1 Comparison with the original calibration

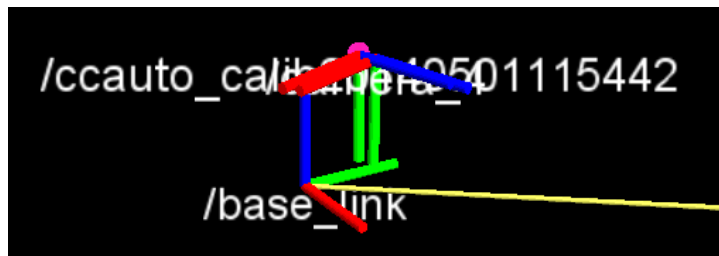


Figure 5.3. The original transform (`/camera_4`) compared with the result of the iterative method.

Value	Original	Result	Difference
X	0.015	0.023	0.008
Y	0.039	0.011	0.028
Z	-0.000	0.004	0.005
Yaw	-18.04°	-18.13°	0.27°
Pitch	0.06°	0.65°	0.59°
Roll	-90.62°	-90.06°	0.56°

Table 5.2. The original transform compared with the result of the iterative method. Values are in meters and degrees.

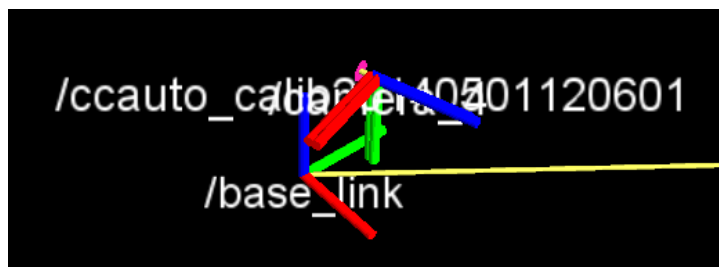


Figure 5.4. The original transform (`/camera_4`) compared with the result of the iterative method with local correction.

Value	Original	Result	Difference
X	0.015	0.007	0.008
Y	0.039	0.026	0.013
Z	-0.000	-0.001	0.001
Yaw	-18.04°	-18.55°	0.04°
Pitch	0.06°	0.79°	0.74°
Roll	-90.62°	-89.89°	0.73°

Table 5.3. The original transform compared with the result of the iterative method with local correction.

■ 5.1.2 Validation using the point cloud coloring



Figure 5.5. Visualization of the colored point cloud using the calibration result of the iterative method with local correction.

It can be seen that the point cloud is colored quite nicely. The only wrongly colored object is the light on the ceiling. It is difficult to create correspondences precisely on the light because the laser has trouble correctly measuring the points on the light due to its reflective surface. The original calibration has the same trouble with the color of the light.

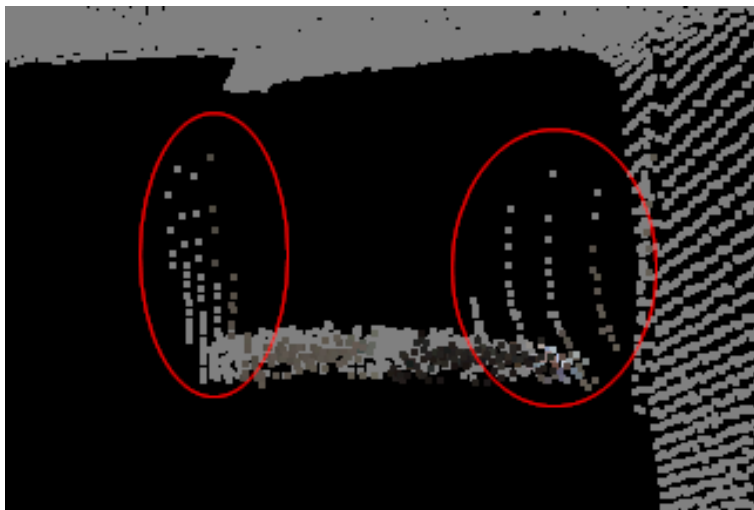


Figure 5.6. Detail of the light with highlighted artifacts.

5.1.3 Calibration by inexperienced operator

We asked our colleague who was not directly involved in the development of the application to try calibrating the camera. The following are the results of his calibration. It can be seen that they are a bit worse than the previous results.



Figure 5.7. Correspondences used for calibration and their projections. Average reprojection error: 11.6, Maximal reprojection error: 19.0

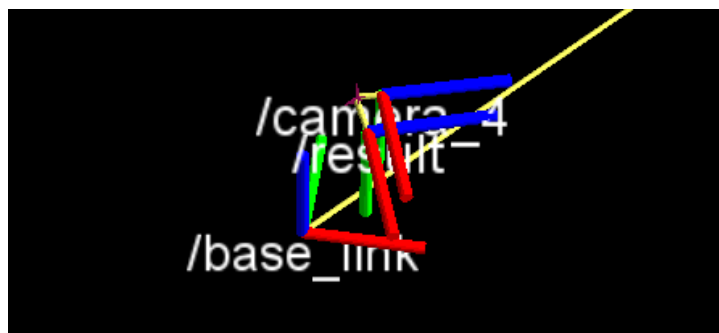


Figure 5.8. The original transform (`/camera_4`) compared with the result of the calibration by inexperienced operator.

Value	Original	Result	Difference
X	0.015	0.050	0.036
Y	0.039	-0.004	0.043
Z	-0.000	-0.027	0.027
Yaw	-18.04°	-17.53°	0.51°
Pitch	0.06°	-0.29°	0.35°
Roll	-90.62°	-89.67°	0.95°

Table 5.4. The original transform compared with the result of the calibration by inexperienced operator.

5.2 Scenario 2 - Hall

Camera resolution is 1616×1232 pixels.



Figure 5.9. 14 correspondences used for the calibration. The point cloud density was too low to create precise correspondences on the rail.

Calibration method	Avg. reprojection error (in pixels)	Max. reprojection error
Original calibration	6.1	12.5
Iterative	5.0	9.7
EPnP	6.6	16.5
Iterative with LC	6.2	10.9
EPnP with LC	8.3	20.0

Table 5.5. Calibration results in scenario 2.

5.2.1 Comparison with the original calibration



Figure 5.10. The original transform (`/camera_4`) compared with the result of the iterative method.

Value	Original	Result	Difference
X	0.015	-0.053	0.068
Y	0.039	0.010	0.029
Z	-0.000	0.018	0.018
Yaw	-18.04°	-18.20°	0.16°
Pitch	0.06°	-0.52°	0.58°
Roll	-90.62°	-90.70°	0.08°

Table 5.6. The original transform compared with the result of the iterative method. Values are in meters and degrees.

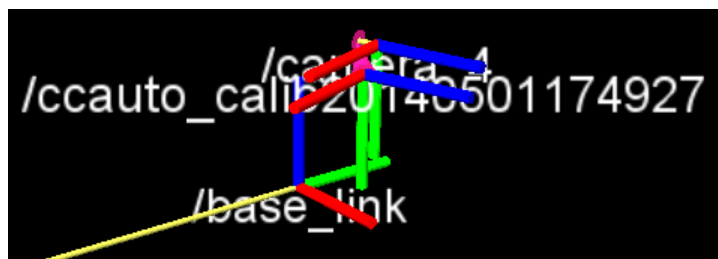


Figure 5.11. The original transform (`/camera_4`) compared with the result of the EPnP method.

Value	Original	Result	Difference
X	0.015	0.089	0.075
Y	0.039	0.036	0.003
Z	-0.000	-0.027	0.027
Yaw	-18.04°	-17.65°	0.39°
Pitch	0.06°	-0.09°	0.15°
Roll	-90.62°	-90.37°	0.25°

Table 5.7. The original transform compared with the result of the EPnP method. Values are in meters and degrees.

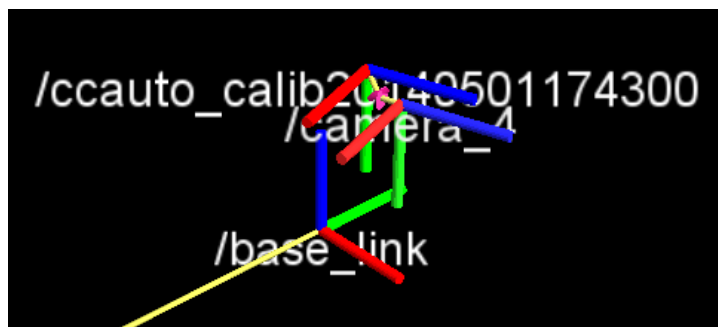


Figure 5.12. The original transform (/camera.4) compared with the result of the iterative method with local correction.

Value	Original	Result	Difference
X	0.015	-0.035	0.050
Y	0.039	-0.028	0.067
Z	-0.000	0.031	0.031
Yaw	-18.04°	-17.85°	0.19°
Pitch	0.06°	0.52°	0.47°
Roll	-90.62°	-90.49°	0.13°

Table 5.8. The original transform compared with the result of the iterative method with local correction. Values are in meters and degrees.

Even though the reprojection errors were lower than in the scenario 1, the calibration results are quite far away from the original calibration. It can be seen that the reprojection error of the original calibration is really high. To check if the correspondences were created correctly we did use the point cloud coloring.

■ 5.2.2 Validation using the point cloud coloring

Only the result of the iterative method was tested in the cloud coloring. Because the point cloud is too sparse on the opposite wall of the hall the 3D visualization was not much helpful. We used the range image coloring present in the application instead.

The quality of the coloring is good and we could not find any significant artifacts by comparing the colored range image with other visualization methods. Even the small parts of the rail are colored correctly. This is great result because there were no correspondences created on the rail or even the lower left quadrant of the image.

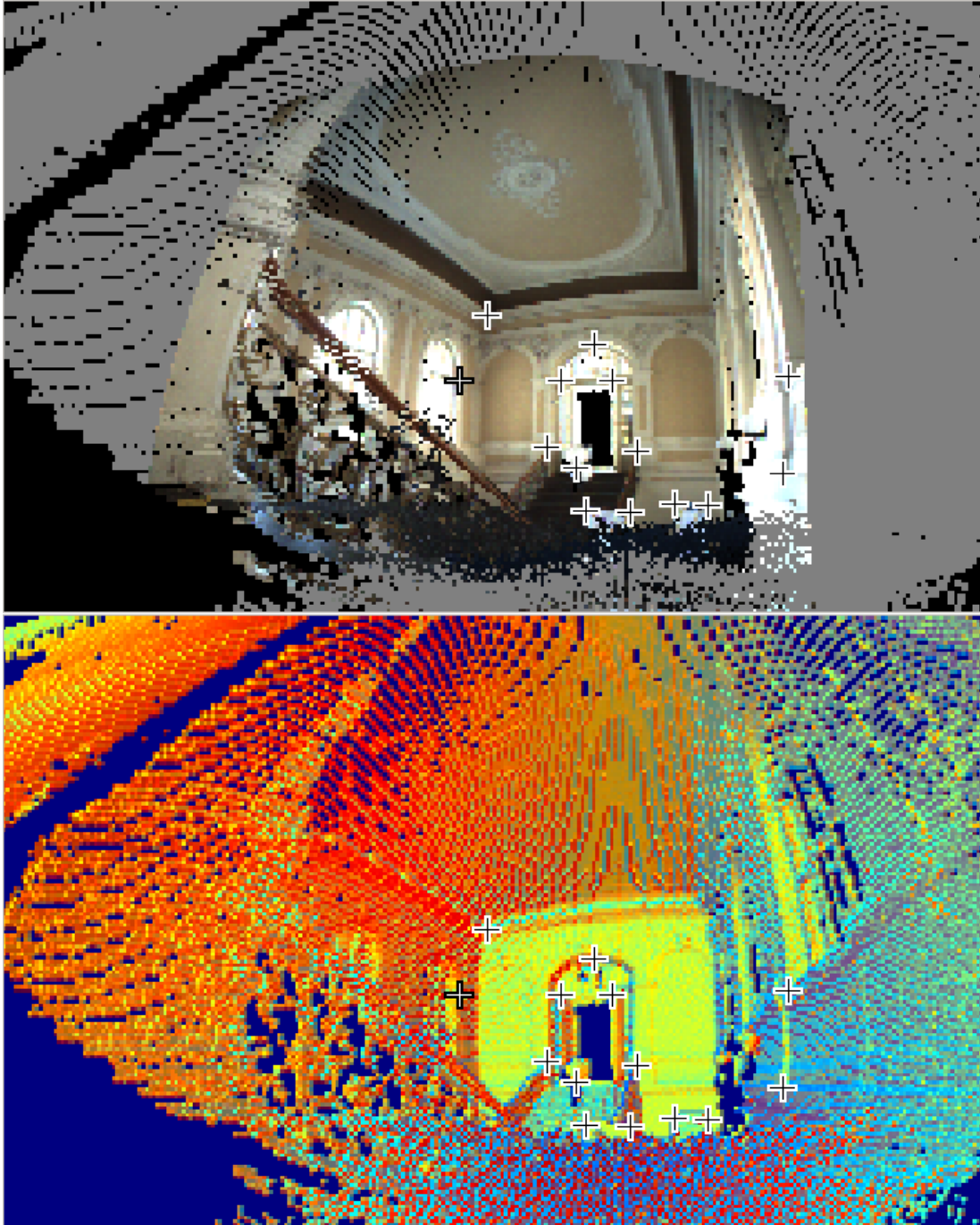


Figure 5.13. Colored range image is at the top. Other visualization of the same image is at the bottom.

5.2.3 Calibration by inexperienced operator

In this scenario the results are significantly worse than the previous results. However, they are still good enough to color the point cloud without many noticeable artifacts.



Figure 5.14. Correspondences used for calibration and their projections. Average reprojection error: 10.3, Maximal reprojection error: 19.0

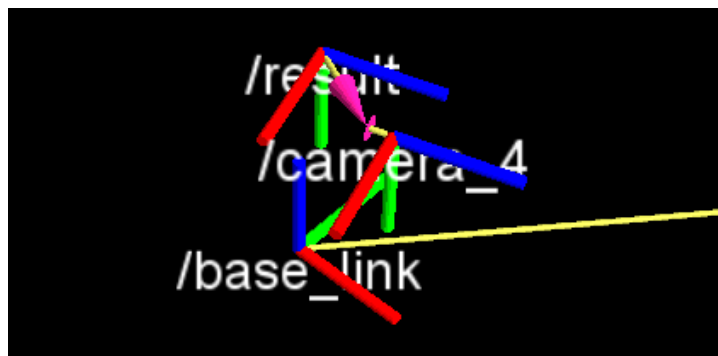


Figure 5.15. The original transform (`/camera_4`) compared with the result of the calibration by inexperienced operator.

Value	Original	Result	Difference
X	0.015	-0.160	0.174
Y	0.039	-0.083	0.122
Z	-0.000	-0.018	0.018
Yaw	-18.04°	-18.54°	0.51°
Pitch	0.06°	-0.18°	0.24°
Roll	-90.62°	-90.28°	0.34°

Table 5.9. The original transform compared with the result of the calibration by inexperienced operator.

5.3 Scenario 3 - Courtyard 1

This is an outdoor scenario. Camera resolution is 1616×1232 pixels. It is the most difficult scenario because the camera is facing the sun and the image is oversaturated.

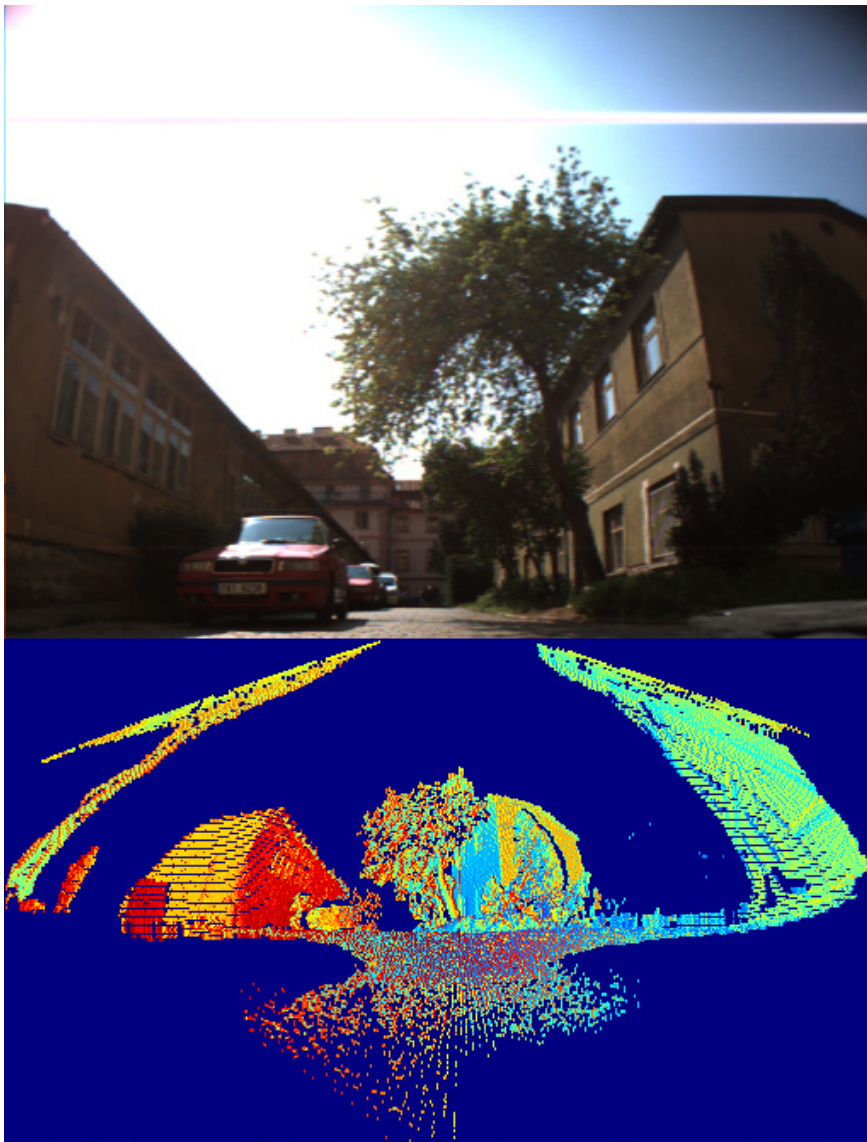


Figure 5.16. Image from the camera and the horizontal range image visualization.

Calibration method	Avg. reprojection error (in pixels)	Max. reprojection error
Original calibration	11.2	26.7
Iterative	5.2	11.0
EPnP	5.4	12.3
Iterative with LC	6.4	10.7
EPnP with LC	10.2	23.6

Table 5.10. Calibration results in scenario 3.

The reprojection errors are good. However, the maximal reprojection error in the original calibration indicates that the correspondences were not created correctly. We have picked all but the last calibration result for further inspection.



Figure 5.17. 9 correspondences used for the calibration.

5.3.1 Comparison with the original calibration

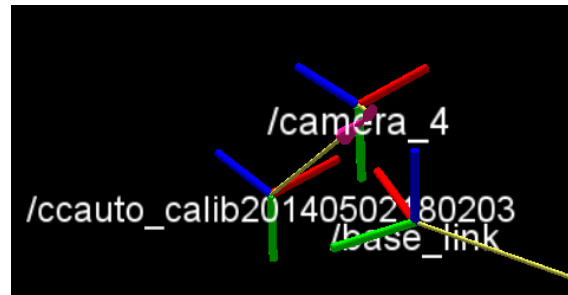


Figure 5.18. The original transform (`/camera_4`) compared with the result of the iterative method.

Value	Original	Result	Difference
X	0.015	-0.046	0.061
Y	0.039	0.242	0.203
Z	-0.000	-0.281	0.281
Yaw	-18.04°	-18.37°	0.34°
Pitch	0.06°	1.70°	1.64°
Roll	-90.62°	-88.22°	2.40°

Table 5.11. The original transform compared with the result of the iterative method. Values are in meters and degrees.

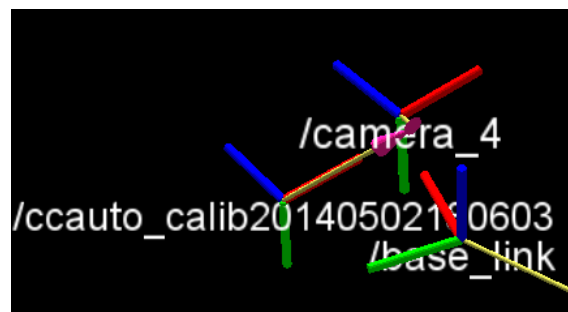


Figure 5.19. The original transform (`/camera_4`) compared with the result of the EPnP method.

Value	Original	Result	Difference
X	0.015	-0.076	0.091
Y	0.039	0.277	0.238
Z	-0.000	-0.276	0.276
Yaw	-18.04°	-18.44°	0.41°
Pitch	0.06°	1.76°	1.70°
Roll	-90.62°	-88.12°	2.50°

Table 5.12. The original transform compared with the result of the EPnP method. Values are in meters and degrees.

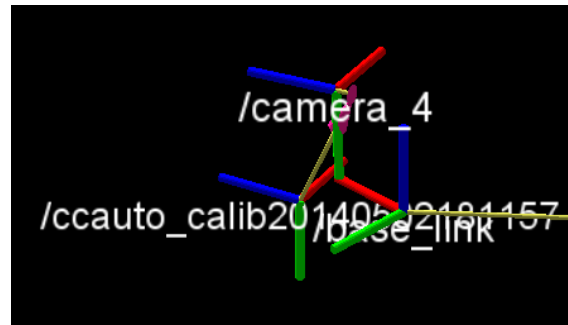


Figure 5.20. The original transform (`/camera_4`) compared with the result of the iterative method with local correction.

Value	Original	Result	Difference
X	0.015	0.088	0.074
Y	0.039	0.138	0.099
Z	-0.000	-0.308	0.308
Yaw	-18.04°	-17.68°	0.35°
Pitch	0.06°	1.49°	1.43°
Roll	-90.62°	-88.27°	2.35°

Table 5.13. The original transform compared with the result of the iterative method with local correction. Values are in meters and degrees.

It is clear that some of the correspondences were wrong. All of the calibration methods converged to the same place. The estimated camera position is significantly far from the hand measured one. It is interesting to note that the rotational components of the results are actually quite good.

■ 5.3.2 Validation using the point cloud coloring

Even though the results were bad we still inspected how the point cloud coloring was affected. We had done the point cloud coloring only for the result of the iterative method. The results were actually quite good. The reason is that the error was mostly in the translational component of the calibration. At long distances this error is not as critical as the error in the rotation.



Figure 5.21. Colored range image using the calibration from the iterative method.

5.4 Scenario 4 - Courtyard 2

This scenario is similar to the scenario 3. However, the robot is not facing the sun and the correspondences were easier to find. Camera resolution is 1616×1232 pixels.



Figure 5.22. 11 correspondences used for the calibration.

Calibration method	Avg. reprojection error (in pixels)	Max. reprojection error
Original calibration	7.1	15.4
Iterative	6.0	10.0
EPnP	6.0	11.1
Iterative with LC	6.2	10.1
EPnP with LC	6.6	14.5

Table 5.14. Calibration results in scenario 4.

5.4.1 Comparison with the original calibration

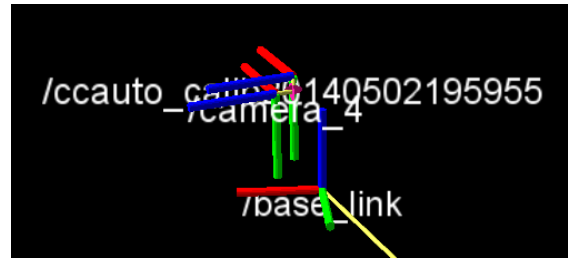


Figure 5.23. The original transform (`/camera_4`) compared with the result of the iterative method.

Value	Original	Result	Difference
X	0.015	0.024	0.009
Y	0.039	-0.018	0.057
Z	-0.000	0.025	0.025
Yaw	-18.04°	-17.63°	0.40°
Pitch	0.06°	-0.18°	0.24°
Roll	-90.62°	-90.82°	0.20°

Table 5.15. The original transform compared with the result of the iterative method. Values are in meters and degrees.



Figure 5.24. The original transform (`/camera_4`) compared with the result of the EPnP method.

Value	Original	Result	Difference
X	0.015	0.069	0.054
Y	0.039	-0.045	0.084
Z	-0.000	0.050	0.050
Yaw	-18.04°	-17.32°	0.72°
Pitch	0.06°	-0.01°	0.07°
Roll	-90.62°	-91.03°	0.41°

Table 5.16. The original transform compared with the result of the EPnP method. Values are in meters and degrees.

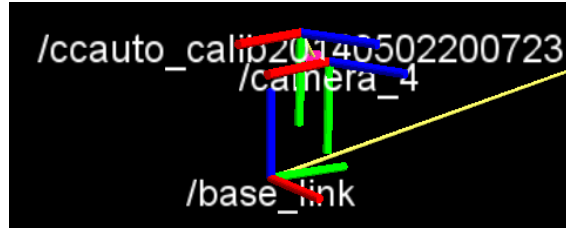


Figure 5.25. The original transform (`/camera_4`) compared with the result of the iterative method with local correction.

Value	Original	Result	Difference
X	0.015	0.047	0.033
Y	0.039	-0.013	0.052
Z	-0.000	0.063	0.063
Yaw	-18.04°	-17.40°	0.64°
Pitch	0.06°	-0.11°	0.16°
Roll	-90.62°	-90.95°	0.33°

Table 5.17. The original transform compared with the result of the iterative method with local correction. Values are in meters and degrees.

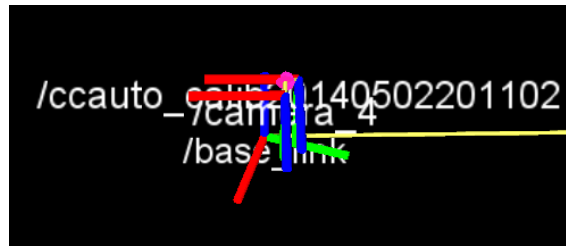


Figure 5.26. The original transform (`/camera_4`) compared with the result of the EPnP method with local correction.

Value	Original	Result	Difference
X	0.015	-0.036	0.050
Y	0.039	-0.016	0.055
Z	-0.000	-0.021	0.021
Yaw	-18.04°	-17.86°	0.17°
Pitch	0.06°	-0.65°	0.71°
Roll	-90.62°	-90.45°	0.17°

Table 5.18. The original transform compared with the result of the iterative method with local correction. Values are in meters and degrees.

All of the results are not bad. The rotational components of the calibrations are solid but the translations are not that good. It is hard to tell which method was the best in this case. The EPnP method with local correction had the greatest reprojection error but the result is the closest to the results of the previous experiments.

5.4.2 Validation using the point cloud coloring

As in the scenario 4, the results were good enough to successfully color the point cloud. At the distances most of the points are, the errors in the translations were too small to be noticeable.

5.5 Scenario 5 - Calibration of an external camera

In this scenario we reused the scene from scenario 1 and tried to calibrate an external camera that was not mounted on the robot. Instead, the camera was sitting on a table next to the robot. The goal was for the calibration to be good enough to successfully color the point cloud. The camera resolution was 320×240 pixels therefore the reprojection errors were smaller than in the previous scenarios. The intrinsic calibration of the camera was not as good. The distortion was unknown.



Figure 5.27. Overview of the scenario 5. Camera is sitting on the table next to the notebook. The bottom-right image is the actual image from the camera.

Calibration method	Avg. reprojection error (in pixels)	Max. reprojection error
Iterative	3.3	6.2
EPnP	3.4	6.6
Iterative with LC	3.4	11.0
EPnP with LC	3.7	8.2

Table 5.19. Calibration results in scenario 5.

For further evaluation only the results of the iterative method were used. However, all of the results are solid.

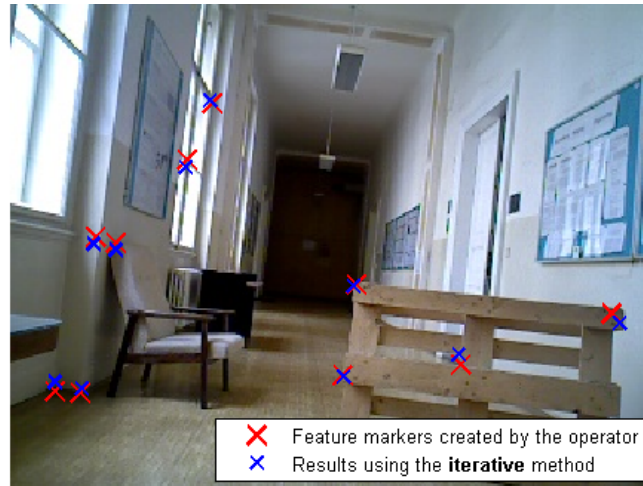


Figure 5.28. 10 correspondences used for the calibration.

In this scenario there is no original calibration available, so the results could not be compared to it. On the following image the result of the iterative method is visualized in rviz. By visual comparison with the photo of the actual camera placement the result seems to be quite good.

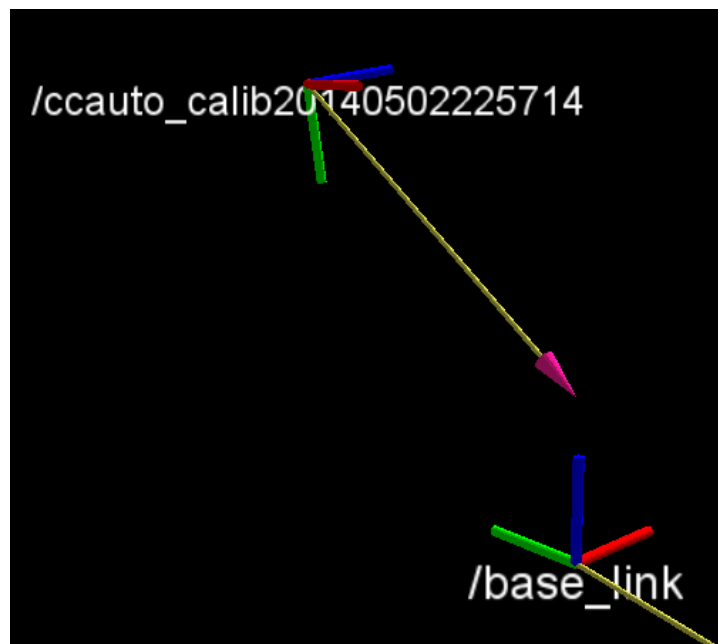


Figure 5.29. The result of the iterative method.

5.5.1 Validation using the point cloud coloring

The quality of the cloud coloring is really good. Rather interesting phenomenon is highlighted in the red area. The pallet texture is wrongly attached to the wall which is caused by the parallax between the camera and the laser scanner. The external camera is placed much further from the laser scanner than the Ladybug camera. Certain parts of the scene are visible only by the camera and other only by the laser scanner.

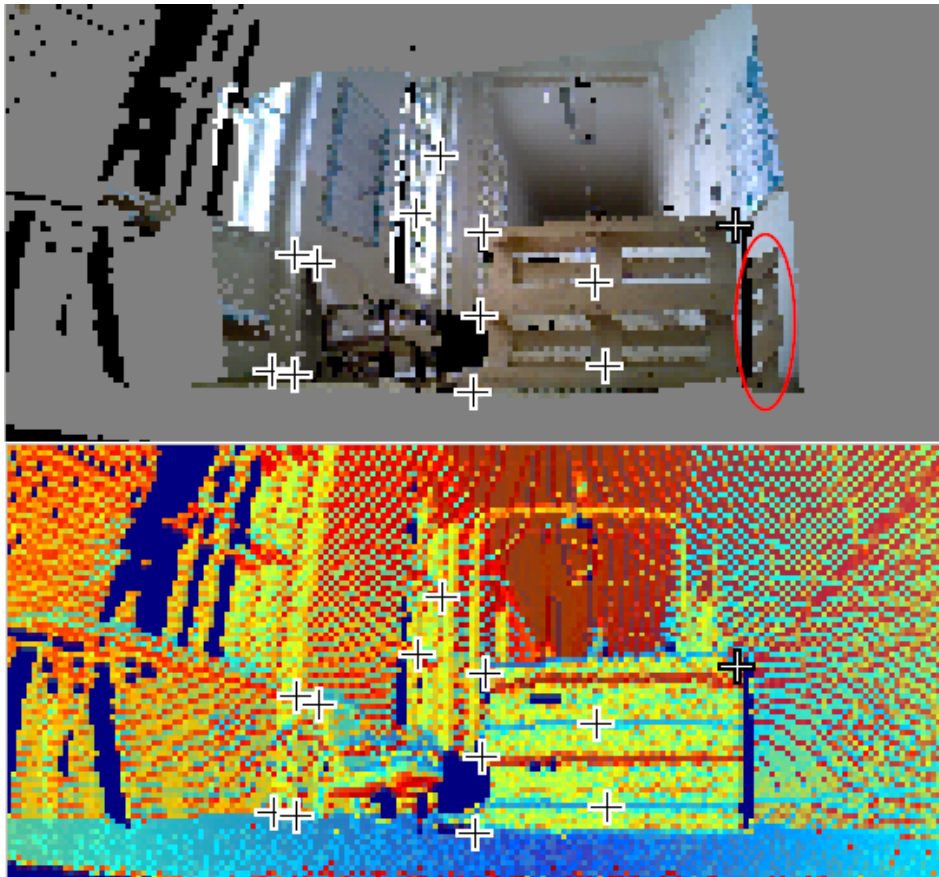


Figure 5.30. Colored range image is at the top. Mix of range image visualizations is at the bottom.

5.6 Summary

Throughout all of the scenarios, the iterative method has consistently performed better than the EPnP method. Both the average and maximal reprojection errors tend to be smaller when using the iterative method compared to the EPnP method. The results of the iterative method were also more consistent with the original calibration in all of the scenarios.

The results using the local correction tend to have greater reprojection errors. However, this does not necessarily imply that they are worse. As explained before this happens mainly because the reprojection error was always measured with respect to the correspondences created by the operator and not to the corrected correspondences. Additionally, we took great care when creating the correspondences in all of the scenarios and the local correction was created for situations when the operator is more

imprecise. Still, when compared to the original calibration, the local correction improved the results in scenario 1 with the iterative method and in scenario 4 with the EPnP method.

Since the positions of the camera and the laser scanner were fixed in the first four scenarios, we measured the consistence of the results. We ignored the results of scenario 3 because they were considerably worse than the rest. The following table shows the standard deviations for all of the extrinsic parameters. Only the best results from scenarios 1, 2 and 4 were taken in account. Because the dataset was really tiny this measurement is not a very reliable estimate of the underlying distribution. Still, it offers us some degree of insight.

Value	σ
X	0.040
Y	0.022
Z	0.013
Yaw	0.46°
Pitch	0.68°
Roll	0.50°

Table 5.20. Standard deviations of the results from scenarios 1, 2 and 4. The distance of the camera from the base frame of the point cloud is approximately 0.3 m.

It is evident that the estimation of the camera’s position was more problematic than the estimation of the camera’s orientation. This is better than if it was the other way round. The reason is that in applications, such as the point cloud coloring, the error in orientation has much greater impact. We have shown this by coloring the point cloud without noticeable artifacts in all scenarios. Even in the scenario 3, which was excluded from the previous statistics, we colored the point cloud without problems. It is also easier to fix the position of the camera manually than to fix it’s orientation.

In the first two scenarios we asked our colleague, inexperienced with the application, to try and calibrate the camera. The results show that he was able to calibrate a camera to a degree useful for point cloud coloring. However, the results are a bit worse compared to our calibration. We also noticed that our colleague had more trouble finding the correspondences and it took him more than twice as long than it took us.

During the development, we used only scenario 1 for testing and calibration. Therefore, we expected that our results will be better than his in scenario 1. However, we did not use the scenario 2 at all during development so we had the same experience with it as our colleague when we performed the calibration. Still, the results of our colleague were worse than ours in scenario 2. This implies that the application requires some training to get used to. However, it also implies that it is not necessary to have previous experience with a particular scene to be able to calibrate a camera successfully.

Chapter 6

Conclusion

We have implemented and tested an application for extrinsic camera calibration in ROS. The application provides an interface for manual location of keypoints and their association in point cloud and image data. It implements several calibration methods, visualizes and exports the results. The application provides several visualizations of the point cloud data in order to enhance details and allow the operator to create enough correspondences. Furthermore, we created an advanced graphical user interface that makes the calibration process fast and intuitive.

For the actual calibration the operator can choose between the iterative algorithm, using the Levenberg-Marquardt minimization, and the EPnP algorithm. The iterative algorithm can be used together with the RANSAC method to identify the correspondences that are clearly wrong. Additionally, we developed a local correction method for the correspondences. The local correction method aims at fixing small inaccuracies introduced by the operator.

To help with the validation of the results we implemented two different tools for point cloud coloring by the calibrated camera. The first tool is incorporated directly in the graphical user interface and it colors the range images. The second tool is a standalone ROS node that colors the point clouds directly. We also implemented export of the data from the application directly into MATLAB source file.

In the experiments we have shown that the application can be used for extrinsic calibration in real situations. Only one outdoor scene did not allow for an acceptable calibration. Even in this scenario, the calibration was good enough to allow reasonable coloring of the point cloud. The usability of the application was tested on a subject with no previous experience of the application. He managed to calibrate the camera to color the point cloud without any visible artifacts. Still, his results were slightly worse than the calibration that we performed ourselves.

The application focuses on speed, flexibility and simplicity of the calibration. The calibration can be often performed in less than five minutes to a precision that is sufficient for many tasks. However, if the time and conditions allow it, the usual techniques using the calibration patterns are probably going to provide more precise calibration.

One of our initial goals was to eliminate the operator from the calibration process completely. We have done some preliminary experiments with automatic matching of the features between the processed range images and the camera images. We experimented with the SIFT descriptors to identify the local features in the images. Sadly, the results were very poor. That is not surprising because, as far as we know, no one was able to create correspondences automatically between point clouds and RGB images yet. After our initial experiments we decided not to pursue this goal further and focus on making the application friendly to an unskilled user.

In conclusion, we have achieved all of our main goals and developed an application that is certainly going to be further used in our team. We have solved the problem and experimented with various different approaches to make the application intuitive and the calibration as good as possible.

References

- [1] *ROS*.
<http://ros.org/>, visited 2014-05-03.
- [2] Richard Hartley, and Andrew Zisserman. *Multiple view geometry in computer vision*. 2nd edition. Cambridge: Cambridge University, 2003. ISBN 0-521-54051-8.
- [3] *Camera calibration package in ROS*.
http://wiki.ros.org/camera_calibration, visited 2014-05-10.
- [4] D Scaramuzza, A Harati, and R Siegwart. *Extrinsic Self Calibration of a Camera and a 3D Laser Range Finder from Natural Scenes*. In: *Proc. of The IEEE International Conference on Intelligent Robots and Systems (IROS)*. 2007.
- [5] *NIFTi arm source codes*.
https://github.com/NIFTi-Fraunhofer/nifti_arm, visited 2014-05-15.
- [6] Duane C. Brown. Close-range camera calibration. *PHOTOGRAMMETRIC ENGINEERING*. 1971, 37 (8), 855–866.
- [7] *How to solve the Image Distortion Problem*.
<http://www.arlab.com/blog/tag/image-distortion/>, visited 2014-05-10.
- [8] V. Lepetit, F. Moreno-Noguer, and P. Fua. *EPnP: An Accurate $O(n)$ Solution to the PnP Problem*. 2008.
- [9] Martin A. Fischler, and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. 1981, 24 (6), 381–395. DOI 10.1145/358669.358692.
- [10] *ROS support of the C++11 standard*.
<http://www.ros.org/repos/rep-0003.html#c>, visited 2014-05-03.
- [11] *Boost C++ Libraries*.
<http://boost.org>, visited 2014-05-03.
- [12] *OpenCV*.
<http://opencv.org>, visited 2014-05-03.
- [13] *PCL - Point Cloud Library*.
<http://pointclouds.org>, visited 2014-05-03.
- [14] *Qt*.
<http://qt-project.org/>, visited 2014-05-03.
- [15] *Tf package in ROS*.
<http://wiki.ros.org/tf>, visited 2014-05-10.
- [16] *Qt signals and slots*.
<http://qt-project.org/doc/qt-4.8/signalsandslots.html>, visited 2014-05-03.
- [17] *QtCreator*.
<http://qt-project.org/wiki/category:tools::qtcreator>, visited 2014-05-03.
- [18] *Qt Model/View tutorial*.
<http://qt-project.org/doc/qt-4.8/modelview.html>, visited 2014-05-03.

- [19] *ROS - Static transform publisher.*
http://wiki.ros.org/tf#static_transform_publisher, visited 2014-05-03.
- [20] *ROS wiki.*
<http://wiki.ros.org/>, visited 2014-05-03.
- [21] *Rviz - user guide.*
<http://wiki.ros.org/rviz/UserGuide>, visited 2014-05-03.



Appendix A

Enclosed CD

- **[bags]**
Contains the bag files for scenario 1 and scenario 5. Also contains convenience scripts able to run them directly with necessary parameters. Please note that in scenario 5 the camera topics are published under different names.
- **[catkin]**
Contains the sources for ROS Hydro Medusa.
- **[rosbuild]**
Contains the sources for ROS Fuerte Turtle.
- **[thesis]**
Contains the sources for this thesis.
- **thesis.pdf**
PDF version of this thesis.