

Morten Astad

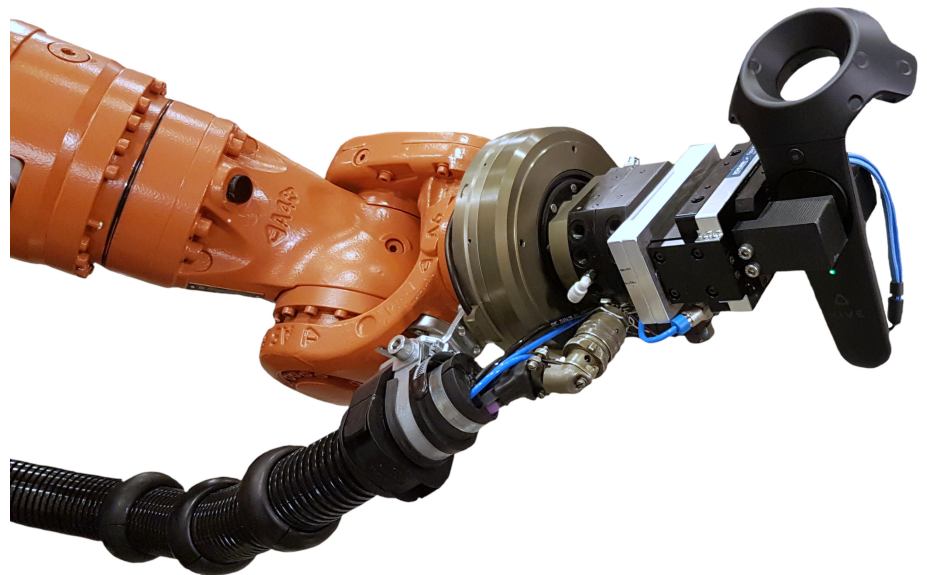
Vive for Robotics

Rapid Robot Cell Calibration

Master's thesis in Cybernetics and Robotics

Supervisor: Jan Tommy Gravdahl

June 2019



Morten Astad

Vive for Robotics

Rapid Robot Cell Calibration

Master's thesis in Cybernetics and Robotics
Supervisor: Jan Tommy Gravdahl
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

This thesis is dedicated to my fiancée, whose unending support has helped me through thick and thin during my 6 years of higher education. This also holds true for my family and close friends who have always been there for me.

— Morten Astad

This page is intentionally left blank.



MSc thesis assignment

Name of the candidate: Morten Astad
Subject: Engineering Cybernetics
Title:

Background:

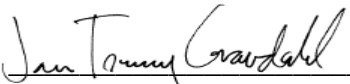
HTC Vive is quite famous for their new virtual reality gaming headset. It has quickly become one of the best in the market. One of the reasons for its success was the innovative technology for tracking users' hands and head. In many industrial automation cases, a robot work cell must be calibrated such that the location of tools, surfaces, and parts are known to the robot. This can be done by jogging the robot (moving it by the remote controller), or by guiding the robot by hand (when the robot is in compliant mode), to specific setpoints in the work cell and recording their location. Another way is to construct the work cell with a high-degree of positional accuracy. In general for robots performing many tasks, and visiting many locations, this is time consuming and requires the robot to be installed ahead of time, or expensive and laborious.

In this master thesis the student is to explore the use of an HTC Vive for rapid robot cell calibration. The work will be focused on interfacing the HTC Vive with ROS: robotic operating system, and investigating the usefulness of the HTC Vive for positioning objects and obstacles in a robot cell.

Tasks:

1. Create a calibration procedure for the HTC Vive controller for use in rapid robot cell calibration. The positioning error should be $<1\text{cm}$, ideally around $\sim 1\text{mm}$
 - a. Map and investigate biases
 - b. Investigate technological limitations of the HTC Vive for sub-centimetric positioning errors
2. Design a robot cell calibration procedure for some standard geometric primitives (e.g. plane/sphere/box). This includes:
 - a. Allow for placement of objects by using the HTC Vive controllers
 - b. Allow for sampling points on an object using the HTC Vive controllers
 - c. Use said sampled points to place 3D objects in a virtual robot cell scene
 - d. Save the resulting scene to an appropriate format (e.g. SDF/URDF)
3. Test the procedure on an assembly use-case

To be handed in by: 10/6-2019


Jan Tommy Gravdahl
Professor, supervisor

This page is intentionally left blank.

ABSTRACT

The use of an HTC Vive; a virtual reality (VR) system and its innovative tracking technology is explored in order to create an approximate one-to-one mapping to the virtual representation of a robot cell. This mapping is found by performing hand-eye calibration, establishing a spatial relationship between the inertial frames of the robot cell and tracking system.

Automated calibration procedures were realized as open-source robotic operating system (ROS) packages, together with a framework that was used to define geometric primitives such as boxes, spheres, cylinders and cones in the coordinates of the robot cell. This framework was tested on an assembly scenario, where the outline of a mock-up was defined by registering points with a Vive Tracker. The dimensions of the mock-up were defined with a centimetric accuracy and a millimetric deviation between similar parts.

The calibrated system has problems that are related to specific issues of the tracking technology. These issues, their cause, and potential fixes are outlined in a concise manner after thoroughly studying them. A considerable effort has been directed towards trying to reduce the influence of a spatially dependant bias on the tracking. This effort culminated in a minor millimetric accuracy improvement, resulting in a centimetric positioning error overall. The potential use cases of the calibrated system are limited by its accuracy, and depends on the required tolerances.

This page is intentionally left blank.

SAMMENDRAG

Bruken av en HTC Vive; et system for virtuell virkelighet og dets innovative sporingsteknologi er utforsket for å opprette en tilnærmet en-til-en avbildning til den virtuelle representasjonen av en robot celle. Denne avbildningen ble funnet ved å bruke hand-eye kalibrering.

Automatiserte kalibreringsprosedyrer ble realisert som Robot Operating System (ROS) pakker basert på åpen-kildekode, sammen med et rammeverk som ble brukt for å definere geometriske former slik som bokser, kuler, sylindre og kjegler i koordinatene til robot cellen. Dette rammeverket ble testet i et monteringsscenario, der omrisset av en mock-up ble definert ved å registrere punkter med en Vive Tracker. Dimensjonene av mock-upen ble definert med en nøyaktighet i centimeterstørrelse og et avvik i millimeterstørrelse mellom lignende deler.

Det kalibrerte systemet har flere utfordringer som er relatert til spesifikke problemer med sporingsteknologien. Disse problemene, hvordan de oppstår, og mulige løsninger er lagt frem på en kortfattet måte etter å ha studert dem grundig. En betraktelig innsats har blitt rettet mot å prøve å redusere virkningen av en bias som avhenger av lokasjonen til de sporede enhetene. Dette arbeidet oppnådde en mindre nøyaktighetsforbedring i millimeterstørrelse, som resulterte i et måleavvik i centimeterstørrelse. De mulige bruksområdene til det kalibrerte systemet er begrenset av nøyaktigheten, og avhenger av de nødvendige toleransene.

This page is intentionally left blank.

*We have to prove that digital manufacturing is inclusive.
Then, the true narrative will emerge: Welcome, robots.
You'll help us. But humans are still our future.*

— Joe Kaeser [1]

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisors: Mathias Hauan Arbo, Jan Tommy Gravdahl and Esten Ingar Grøtli, for invaluable input throughout the project period. The help that I received with practical aspects of the project, from the hard working people at the mechanical workshop in the department of engineering cybernetics, NTNU, was also very much appreciated. I would also like to thank the LibSurvive community for help with implementing their library, and providing documentation and interesting discussions on the VIVE and its tracking system.

The work presented in this thesis was partially funded by the Research Council of Norway through the projects SFI Manufacturing (contract number: 237900) and Dynamic Robot Interaction and Motion Compensation (contract number: 270941).

Morten Andre Astad
Trondheim
June 10, 2019

This page is intentionally left blank.

CONTENTS

1	INTRODUCTION	1
1.1	Previous work	2
1.2	Structure	3
1.3	Mathematical notation	3
2	HARDWARE AND SOFTWARE	5
2.1	Thrivaldi	5
2.2	Leica Absolute Tracker AT960	5
2.3	Robotic Operating System (ROS)	5
2.3.1	Robot geometry library	6
2.3.2	MoveIt	6
2.3.3	RViz	9
2.4	Vive Bridge	9
2.5	libsurvive	10
2.6	Other libraries	10
2.6.1	Eigen	10
2.6.2	Sophus	11
2.6.3	Ceres solver	11
3	THEORY	13
3.1	Hand-eye calibration	13
3.1.1	Problem formulation	13
3.1.2	Overview of solutions	15
3.1.3	Solving $AX = XB$ on the Euclidean Group	15
3.2	Quaternion averaging	21
3.3	Closed-form solution of absolute orientation	25
3.4	Perpendicular distance from a point to a line	27
4	HTC VIVE	29
4.1	Lighthouse tracking	29
4.1.1	Lighthouse tracking 2.0	31
4.2	Accuracy and precision	32
4.3	Tracking issues	32
4.4	Minor issues	33
4.4.1	Controller timeout	34
4.4.2	Tracker roles	34
5	VIVE-ROBOT CELL SETUP AND CALIBRATION	37
5.1	Internal Vive calibration	37
5.2	Existing calibration procedure	39
5.2.1	Generating sample poses for calibration	39
5.2.2	Computing the mapping	40
5.2.3	Performing the calibration for Thrivaldi	40

6	IMPROVING THE CALIBRATED SYSTEM	43	
6.1	Improving the existing calibration procedure	43	43
6.1.1	Sampling procedure	43	
6.1.2	Reducing the mapping error	44	
6.1.3	Nonlinear optimization step	45	
6.2	Mapping the error with a robot	46	
6.2.1	Generating a set of sampling poses	47	47
6.2.2	Running the sampling procedure	49	49
6.3	Mapping the error with a laser tracker	52	
6.3.1	Reasons for the large changes in offset	54	54
6.4	LibSurvive	56	
7	RAPID ROBOT CELL CALIBRATION	59	
7.1	Defining geometric primitives from points	59	59
7.1.1	Plane of finite size	59	
7.1.2	Box	61	
7.1.3	Sphere	62	
7.1.4	Cylinder and Cone	63	
7.2	Representing a virtual robot cell in ROS	65	65
7.2.1	Simulation Description Format (SDF)	65	65
7.3	Calibration Tool	66	
7.4	Assembly scenario	68	
8	DISCUSSION, FUTURE WORK AND CONCLUSION	71	
8.1	Discussion	71	
8.1.1	Summarizing the tracking issues	71	71
8.1.2	Improving the calibration procedure	73	73
8.2	Future work	74	
8.3	Conclusion	75	
Appendix			
A	VIVE BRIDGE ROS PACKAGE README	79	
B	SUBMITTED CONFERENCE PAPER TO ICCMA 2019	89	89
	BIBLIOGRAPHY	97	

INTRODUCTION

The use of traditional industrial robots in small and medium-sized enterprises (SMEs) is often too inflexible for the current market demands of the manufacturing industries. SMEs are companies whose staff headcount is less than 250 employees, and are often referred to as the backbone of Europe's economy, providing the majority of all new jobs. The European Commission considers SMEs and entrepreneurship as key to ensuring economic growth, innovation, job creation, and social integration in the European Union (EU) [2].

SMErobotics was an EU-funded research project that ran from 2012 to 2016, and aimed to create robots suitable for SMEs. A published article about the project suggested that one of the main challenges preventing adoption of industrial robots in SMEs, is the fact that current robot programming techniques are not suitable for frequent changes of often highly customized products manufactured in small batches [3]. The demanding situation of manufacturing within SMEs is apparent in this challenge, where flexibility and versatility are required traits in order to accommodate a wide range of products with small lot sizes.

SMErobotics demonstrated multiple solutions that focused on intuitive human-robot interaction (HRI) and robust automatic operation through embedded cognition, of which this thesis will focus primarily on the former. These solutions tie into the research area of robust and flexible automation, which aims to overcome such challenges with novel technologies and methodologies.

This thesis explores the use of an HTC Vive, a virtual reality (VR) system codeveloped by Valve and HTC, in order to create an approximate one-to-one mapping to the virtual representation of a robot cell. The innovative technology that allows for such a mapping in a room-scaled environment is called lighthouse tracking. This technology is able to track the user's hands, head or other objects in real-time through tracked devices. The devices have sub-millimeter precision within an area, whose diagonal is up to 5 meters in length. These specifications are remarkable for a consumer-grade product such as the Vive, and

its relatively low retail price makes it an affordable tracking solution.

A potential use case is the ability to program robot systems with the intuitive user interface that VR experiences such as the Vive provides. Assuming that mapping a robot cell to its virtual counterpart is possible with sufficient accuracy; the interface could enable the user to interact directly with the robot and its environment. The user could then define the location of parts, surfaces and tools for the robot without any expertise in robotics, by simply pointing a tracked device at their respective locations in the robot cell.

The work on creating such a user interface has been focused on interfacing with Robot Operating System (ROS); an open-source collection of frameworks for writing software for robots. Most of the methods presented in this thesis are implemented as ROS packages in some shape or form, and ROS has been an invaluable asset in the development flow of this work.

1.1 PREVIOUS WORK

This thesis is written as the continuation of a summer job at SINTEF Digital and a specialization project [4], and aims to build upon and complete unfinished tasks from the project. Software for interfacing the Vive with ROS was created during the summer job, and this software has been developed further throughout the last year (2018 - 2019). The specialization project was: primarily concerned with how a calibration procedure can be established for the Vive's tracking system.¹ This procedure will be introduced in chapter 5.

Since the work that is presented in this thesis is closely linked to the specialization project, specified parts of the text will be reproduced from the specialization project. This text has been typeset as shown above in a different font with a footnote to distinguish it from the new content. Marks for these footnotes in the text will refer to the same footnote on this page, as it is the only footnote that was used in this thesis.

Two problems related to specific issues of the Vive's tracking system was identified in the specialization project; the tracking dynamics of the Vive was very slow and the tracking was affected by a spatially dependant bias. This thesis will focus on

¹ This text has been reproduced from a specialization project that is closely linked to the work that is presented in this thesis [4].

trying to understand these issues and explore potential solutions to (hopefully) achieve sub-centimetric accuracy.

1.2 STRUCTURE

The thesis is split into 7 chapters:

- CHAPTER 2 Gives a brief overview of the hardware and software that was used in this work.
- CHAPTER 3 Presents the theory that is used throughout this thesis, with an emphasis on hand-eye calibration and quaternion averaging.
- CHAPTER 4 Introduces the Vive and its tracking system in a thorough manner.
- CHAPTER 5 Explains how the Vive was set up in a robot cell, and introduces the calibration procedure from the specialization project.
- CHAPTER 6 Tests several approaches to improve the accuracy and precision of the calibrated system, including mapping the spatially dependant bias.
- CHAPTER 7 Presents a framework that was used to define the virtual representation of an assembly scenario with geometric primitives.
- CHAPTER 8 Summarizes the findings of this work, and gives a brief discussion on each of them before concluding the thesis.

1.3 MATHEMATICAL NOTATION

The following notation will be used consistently:

- Vectors and matrices are shown with bold text in equations, and points will generally be represented as vectors. However, vectors and matrices in the main text will not be typeset as bold, in order to avoid distracting elements that sticks out in the text.
- $\mathbf{o}_{m \times n}$ and $\mathbf{I}_{m \times n}$ denotes a $m \times n$ matrix with zeros and ones along its diagonal respectively.

- A rigid transformation T_1^0 is defined as the transformation that maps a vector from frame $\{1\}$ to frame $\{0\}$:

$$\mathbf{p}^0 = \mathbf{T}_1^0 \mathbf{p}^1, \quad \mathbf{p}^0, \mathbf{p}^1 \in \mathbb{R}^n, \mathbf{T}_1^0 \in SE(3) \quad (1.1)$$

Where $SE(3)$ is the special Euclidean group of dimension 3, as defined by (3.3).

- Operators mapping a vector to a matrix have been surrounded by brackets in order to explicitly show that the resulting matrix is multidimensional. An example of such an operator is the vee-operator $[\hat{\cdot}]$, as defined by (3.25b).

HARDWARE AND SOFTWARE

This chapter gives a brief overview of the hardware and software that was used in this work. As the Vive is the main subject of this thesis, it will be explored in more detail in chapter 4.

2.1 THRIVALDI

Thrivaldi is a KUKA robotics laboratory at the Department of Engineering Cybernetics, NTNU. The laboratory consists of a robot cell containing two, 6 degrees of freedom (DOF), KUKA KR16-2 industrial robots. One of which is mounted on a 3-axis gantry system from Güdel, giving it 9 DOF in total. Thereby the anglicized name Thrivaldi (Privaldi), a 9-headed jötunn from Norse mythology.

The setup and interface of the laboratory was documented extensively by Eriksen in his Master thesis [5], and practical documentation is also available online [6].¹

2.2 LEICA ABSOLUTE TRACKER AT960

The Leica Absolute Tracker AT960 is a portable 6-DOF laser measurement system from Hexagon Manufacturing Intelligence. It is intended for large-scale metrology applications, and allows for a large 360° measurement volume of up to 160 m in diameter. This volume depends on the model and type of measurement (3D or 6-DOF).

The AT960-SR short range model that was used in this project has a maximum volume of 12 m in diameter for 3D measurements. This model was used to track the position of a 1.5 inch sphere-shaped red ring reflector to within an absolute accuracy of 15 microns.

2.3 ROBOTIC OPERATING SYSTEM (ROS)

ROS is an open-source collection of frameworks for writing software for robots, commonly referred to as a middleware. At its core it offers a communication system, which provides a message passing interface between distributed nodes. This infrastructure encourages the user to

implement a clear interface between the nodes in their system, making ROS a distributed and modular framework by design [7].

The main motivation behind ROS is code reusability in robotics research and development. And ROS has an active community with a large collection of tools and libraries, which simplifies the task of writing complex and robust software for robots.¹

2.3.1 *Robot geometry library*

There are many coordinate frames in a robot system. These frames and their relationships are maintained in a distributed tree structure that is buffered in time with the tf2 transform library for ROS [8]. This library allows the user to transform vectors, quaternions, poses and so forth between any two frames in the tree structure. It also acts as a buffer for the poses of the tracked devices, which are available in any frame of the transform tree, and to all nodes in the ROS environment.

2.3.2 *MoveIt*

MoveIt is a motion planning library for ROS, and the existing MoveIt setup for Thrivaldi was used in order to plan and execute trajectories for the robot. This hardware-agnostic framework enables the use of any ROS-Industrial supported robot with a MoveIt package.

The primary move group interface of MoveIt makes use of kinematic robot states to plan the trajectory. These states contains the desired joint positions, velocities, accelerations and efforts at the start and goal of a trajectory. Figure 2.1 shows an example of start and goal states with different joint positions. It is also possible to introduce path constraints along the trajectory.

2.3.2.1 *Safety*

The robot could cause harm to itself or its environment. This is especially true for the MoveIt interface that is used with Thrivaldi, which overrides some of the safety features of the KUKA robot controller (KRC) [5]. It is therefore important that the MoveIt interface is configured properly. Most notably, there were two issues that could cause harm:

- The gripper was not defined as a part of the robot, and planned trajectories could self-intersect with the gripper.

Center (XYZ)	-0.40	0.49	1.22
Size (XYZ)	3.25	3.25	1.63

Table 2.1: MoveIt workspace bounds in meters, as defined by a box relative to the root coordinate system that is shown in figure 2.2.

- The workspace bounds of MoveIt were not configured properly, and allows for trajectories that could slam the robot into the ground. Although the KRC would emergency stop the robot before it reaches the floor, it would still try to reach the floor. This behaviour is undesired and could be harmful for the robot.

Joint limits on the robot's wrist axes, 5 and 6, were changed to $\pm\pi/2$, such that the self-intersections are impossible. These joint limits are defined in the `config\joint_limits.yaml`-file of the MoveIt configuration package for Thrivaldi:

```

floor_joint_a5:
  has_velocity_limits: true
  max_velocity: 5.75958653158
  has_acceleration_limits: false
  max_acceleration: 0
  max_position: 1.570796
  min_position: -1.570796
floor_joint_a6:
  has_velocity_limits: true
  max_velocity: 10.7337748998
  has_acceleration_limits: false
  max_acceleration: 0
  max_position: 1.570796
  min_position: -1.570796

```

An identical change was also performed on the joint limits of the gantry robot. The workspace bounds of MoveIt was modified to better reflect the (KUKA KR16-2) robot's workspace, including the floor. These bounds were reconfigured as represented by the blue box in figure 2.2 to the values in table 2.1.

2.3.2.2 Pre-planning robot trajectories

The last safety measure was to ensure that the planned robot trajectories are correct in the first place. A MoveIt based interface was created in order to pre-plan all of the trajectories. These trajectories are pre-planned by setting the goal state of a trajectory as the start state of the next trajectory, as shown in listing 2.1.

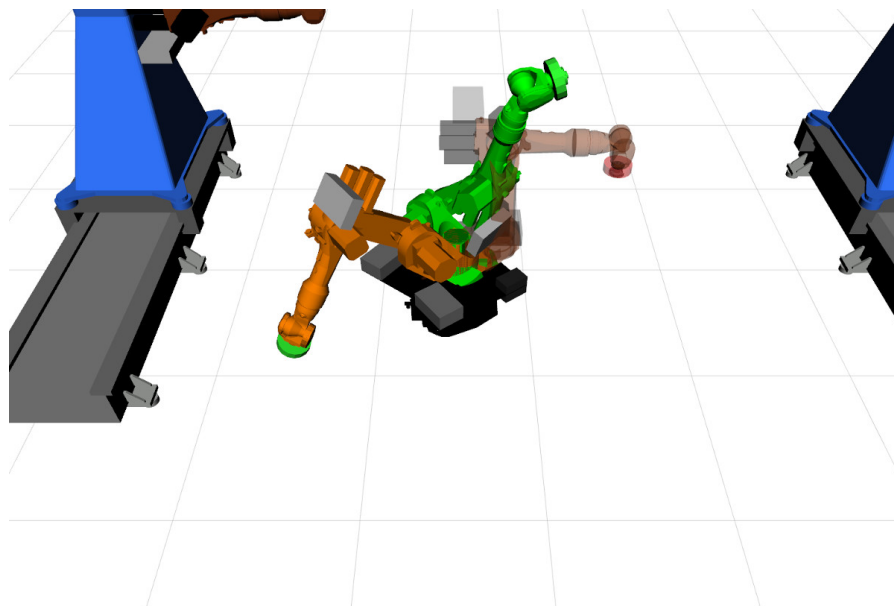


Figure 2.1: Start and goal states of the robot, as shown in green and orange respectively. The primary move group interface of MoveIt plans a trajectory that satisfies these start and goal constraints.

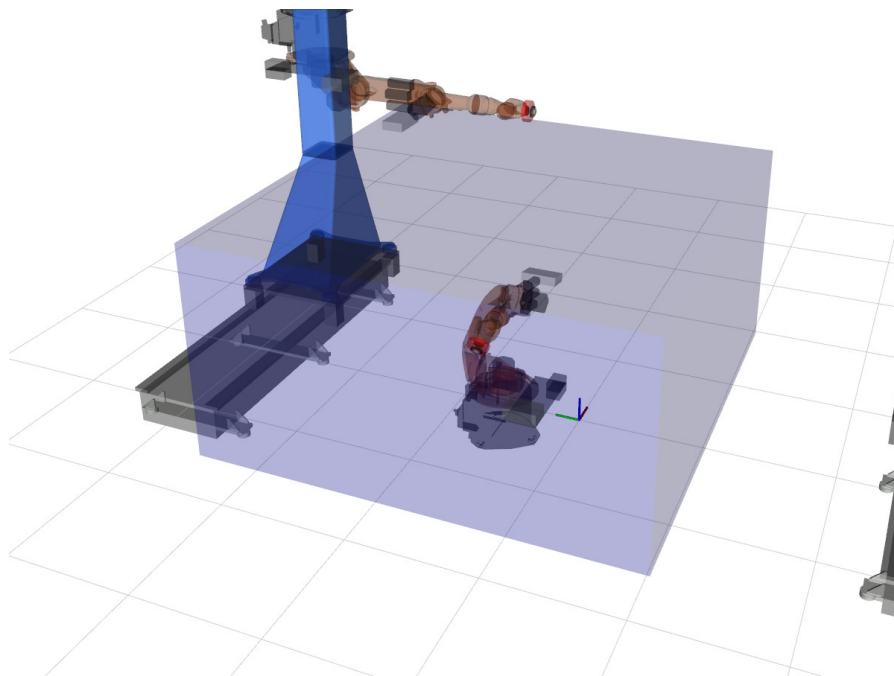


Figure 2.2: New MoveIt workspace bounds.

It is then possible to pre-plan all of the robot trajectories. And the user can verify that the trajectories are correct, before running a full experiment with the verified trajectories. This method also has the added benefit that the trajectories can be executed identically over multiple runs. The execution stops automatically if the robot's start or goal states deviates from the pre-planned states at any point during the execution.

2.3.3 RViz

RViz is a 3D visualizer for ROS that provides a graphical user interface (GUI). It is able to visualize data from the message definitions of ROS through so-called displays, which are standardized for displaying specific message types or composition of message types. RViz will be used to present a 3D visualization of the virtual robot cell, and every figure of it in this thesis was taken directly from RViz.

2.3.3.1 Rviz Visual Tools

Rviz Visual Tools is a library that wraps the message definitions of ROS to display geometric primitives and meshes in Rviz via helper functions in C++. This library will be used to visualize objects in the robot cell, as represented by geometric primitives.

2.4 VIVE BRIDGE

Vive Bridge is a ROS node that makes use of SteamVR through the OpenVR software development kit (SDK) by Valve, allowing access to VR hardware such as the Vive in a ROS environment. The package

Listing 2.1: Setting the goal state of a trajectory as the start state of the next trajectory.

```

if (move_group.plan(plan) ) { // If planning was successful
    // Get goal state of this plan as its joint positions
    std::vector<double> positions = plan.trajectory_.
        joint_trajectory.points.back().positions;
    // Set joint positions in a robot state object
    state->setVariablePositions(joint_names, positions);
    // Set goal state of this plan as start state of next plan
    move_group.setStartState(*state);
}

```

exposes the pose of each tracked device as a coordinate frame with respect to an inertial tracking frame. These frames are maintained in the transform tree.

The inertial tracking frame is defined relative to some arbitrary inertial frame that is chosen by the user, in order to make sense of the environment. These frames are related by a transformation that is exposed as x , y , z and roll, pitch, yaw (RPY) parameters. A standard interface is provided to change the parameters at any time. The robot cell or any other space is then calibrated by updating these parameters.¹

Other features of the package includes: controller inputs, haptic feedback (vibration), linear and angular velocities (twists) and 3D visualization in RViz of the tracked devices. A standard interface to interact with and calibrate the node at any time is also available through the dynamic reconfigure package for ROS.

The package is open-source under the MIT License and it is freely available from: https://github.com/mortaas/vive_rrcc. The documentation for this package has been included in appendix A, and it is recommended that the reader skims through this documentation for an overview of the complete package.

2.5 LIBSURVIVE

Libsurvive is a library that aims to reverse engineer and be an open-source alternative to the SteamVR software stack by Valve. In other words, it provides everything that is needed to perform lighthouse tracking, from low-level device drivers to a high-level application programming interface (API).

Contrary to OpenVR, this library allows for access to the low-level components of the lighthouse tracking. It supports the use of different tracking algorithms, or so-called posers, which have been implemented by its community. The default tracking algorithm was tested in this work, and according to the developers, this should yield optimal results.

2.6 OTHER LIBRARIES

2.6.1 *Eigen*

Eigen is a C++ template library for linear algebra. This library has been used for most of the computations in this work. ROS has a compatibility package named `eigen_conversions`, which is

able to convert back and forth between the message definitions of ROS and the matrix format of Eigen.

2.6.2 *Sophus*

The Sophus library implements Lie groups and their operations using the Eigen library, and also includes the required definitions for automatic differentiation (AD) with the Ceres solver.

2.6.3 *Ceres solver*

Ceres solver is an open source C++ library for modeling and solving non-linear least squares problems [9]. It has been used to solve several optimization problems throughout this work. A brief explanation on how to use this library will be given in section [6.1.3](#).

This page is intentionally left blank.

THEORY

This chapter presents the theory that is used throughout this thesis. An emphasis will be placed on hand-eye calibration and quaternion averaging. It is assumed that the reader is familiar with the use of homogeneous transformation matrices, quaternions and coordinate frames. The reader is referred to [10] for an introduction to these topics.

3.1 HAND-EYE CALIBRATION

The text in this section about hand-eye calibration is based on a specialization project that is closely linked to the work that is presented in this thesis [4].

Finding the pose of a sensor with respect to a robot's tool frame is known as the hand-eye calibration problem. This problem got its name from the robotics community, where a camera (eye) was attached to the robot's gripper (hand).

3.1.1 Problem formulation

The standard hand-eye calibration problem was formulated by Shiu and Ahmad [11]. They stated the problem as an equation of homogeneous transformations:

$$\mathbf{AX} = \mathbf{XB}, \quad \mathbf{A}, \mathbf{B}, \mathbf{X} \in \text{SE}(3) \quad (3.1)$$

Where A depicts a change in the robot's tool pose, B represents the resulting sensor displacement from changing the tool pose, and X is an unknown transformation relating the tool frame to the sensor frame. The unknown transformation X is constant under the assumption that the sensor is firmly attached to the robot's gripper. Figure 3.1 shows a geometric interpretation of this problem.

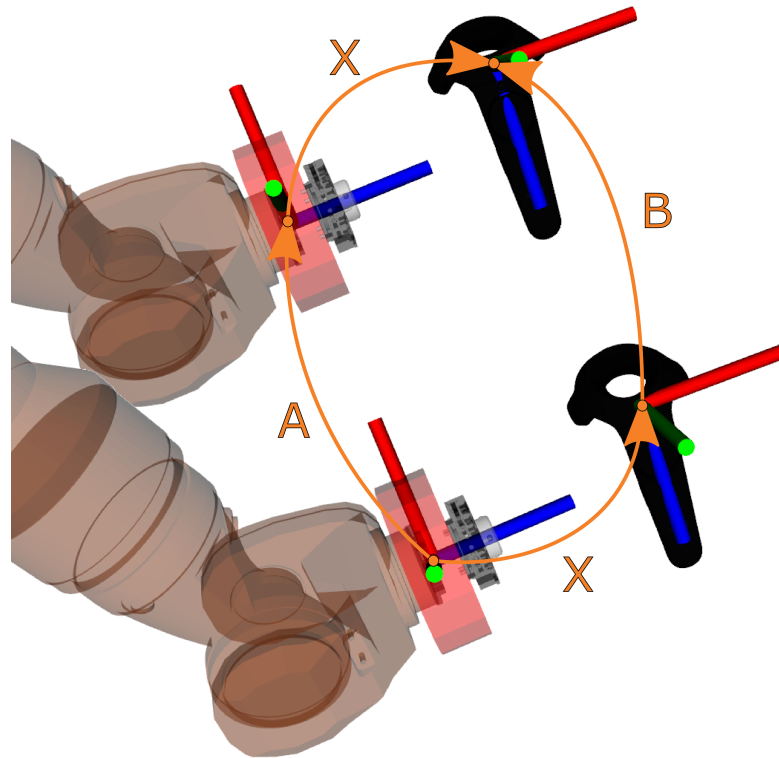


Figure 3.1: Geometric interpretation of the $AX = XB$ problem, showing two different robot states [4].

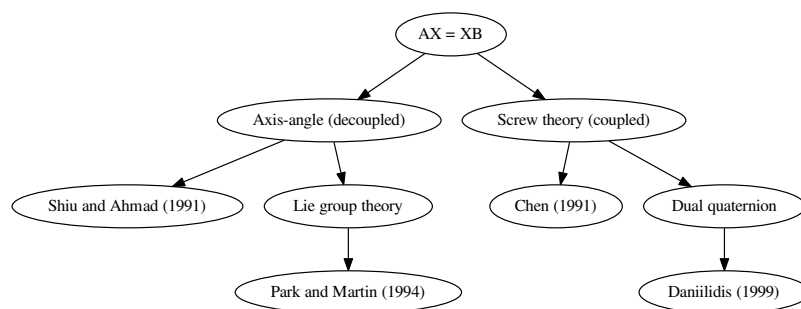


Figure 3.2: A couple of hand-eye problem representations, and associated solutions presented in the literature [4].

3.1.2 Overview of solutions

Hand-eye calibration is an established field with a large body of literature. Shiu and Ahmad [11] used an angle-axis representation of (3.1) and least-squares fitting to solve the rotational part of X , and then solved the translational part with the rotation. They also showed that at least three pairs of A and B are necessary to get a unique solution. Similarly, Park and Martin [12] also decoupled the rotation and translation, and presented a method using Lie group theory and least-squares fitting.

Decoupling the rotation and translation, despite its simplicity, has the problem that errors in the rotational part could propagate to the translational part. Chasles' theorem, as stated by Chen [13], says that a rigid body displacement can be composed of a translation along a unique screw axis, and a rotation about the same axis. This representation of a rigid body displacement is known as a screw. Chen was the first that solved both parts of X simultaneously by applying the theory of screws. Similarly, Daniilidis [14] applied unit dual quaternions, an algebraic representation of screws. Interestingly, the hand-eye problem (3.1) can be reduced to solving a second order equation, by using dual quaternions to represent the transformations A , B and X .

The solutions introduced here and their relations are summarized in figure 3.2. There are many other solutions that are not mentioned, and a more comprehensive resource is available in [15]. Of the mentioned solutions, Park and Martin [12] were chosen specifically for its mathematical elegance.¹

3.1.3 Solving $AX = XB$ on the Euclidean Group

Park and Martin [12] used Lie group theory to concisely state the conditions for existence and uniqueness of their solutions. The input to their method is measured pairs of homogeneous transformation matrices $(A_i, B_i) \in SE(3)$, as defined by the deviation between consecutive samples of tool $\{t\}$ and sensor $\{s\}$ poses:

$$\mathbf{A}_i = \mathbf{T}_{t_i}^{-1} \mathbf{T}_{t_{i+1}}, \quad \mathbf{T}_{t_i}, \mathbf{T}_{t_{i+1}} \in SE(3) \quad (3.2a)$$

$$\mathbf{B}_i = \mathbf{T}_{s_i}^{-1} \mathbf{T}_{s_{i+1}}, \quad \mathbf{T}_{s_i}, \mathbf{T}_{s_{i+1}} \in SE(3) \quad (3.2b)$$

Where $SE(3)$ is the special Euclidean group of dimension 3:

$$\text{SE}(3) := \left\{ \mathbf{T} \mid \mathbf{T} = \begin{pmatrix} \mathbf{R} & \mathbf{r} \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix}, \mathbf{R} \in \text{SO}(3), \mathbf{r} \in \mathbb{R}^3 \right\} \quad (3.3a)$$

$$\text{SO}(3) := \left\{ \mathbf{R} \mid \mathbf{R} \in \mathbb{R}^{3 \times 3}, \mathbf{R}^T \mathbf{R} = \mathbf{I}_{3 \times 3}, \det(\mathbf{R}) = 1 \right\} \quad (3.3b)$$

This Lie group is a C^∞ or smooth manifold with the group structure $\text{SO}(3) \times \mathbb{R}^3$, such that its group operations are smooth maps, that is, derivatives of all orders exist. The most important Lie group property for the purpose of this derivation, is the existence of a well-defined logarithmic mapping from the manifold to a local Euclidean structure on the manifold:

$$\log : \text{SO}(3) \mapsto \text{so}(3) \quad (3.4)$$

Where $\text{so}(3)$ is defined by:

$$\text{so}(3) := \left\{ [\hat{\boldsymbol{\omega}}] \mid [\hat{\boldsymbol{\omega}}] \in \mathbb{R}^{3 \times 3}, [\hat{\boldsymbol{\omega}}]^T = -[\hat{\boldsymbol{\omega}}] \right\} \quad (3.5)$$

And $[\hat{\mathbf{k}}]$ is the skew-symmetric matrix form of a vector \mathbf{k} :

$$[\hat{\mathbf{k}}] := \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix} \in \text{so}(3), \quad \mathbf{k} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \in \mathbb{R}^3 \quad (3.6)$$

This local structure is associated with the tangent space at the group's identity element, called the group's Lie algebra. The inverse exponential mapping is also well defined:

$$\exp : \text{so}(3) \mapsto \text{SO}(3) \quad (3.7)$$

With this knowledge in mind, let us start by expanding the standard hand-eye problem (3.1) as:

$$\begin{pmatrix} \mathbf{R}_A & \mathbf{r}_A \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_X & \mathbf{r}_X \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}_X & \mathbf{r}_X \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{R}_B & \mathbf{r}_B \\ \mathbf{0}_{1 \times 3} & 1 \end{pmatrix} \quad (3.8a)$$

$$\mathbf{R}_A \mathbf{R}_X = \mathbf{R}_X \mathbf{R}_B \quad (3.8b)$$

$$\mathbf{R}_A \mathbf{r}_X + \mathbf{r}_A = \mathbf{R}_X \mathbf{r}_B + \mathbf{r}_X \quad (3.8c)$$

$$(\mathbf{R}_A - \mathbf{I}_{3 \times 3}) \mathbf{r}_X = \mathbf{R}_X \mathbf{r}_B - \mathbf{r}_A \quad (3.8d)$$

The main result of Park and Martin [12] is that an exact solution to (3.1) exists if and only if:

$$\|\log(\mathbf{A})\| = \|\log(\mathbf{B})\|, \quad \mathbf{A}, \mathbf{B} \in \text{SE}(3) \quad (3.9)$$

Where the logarithm of a homogeneous transformation matrix $\mathbf{T} \in \text{SE}(3)$ is given by:

$$\log(\mathbf{T}) = \begin{bmatrix} \log(\mathbf{R}) & \mathbf{T}_{v \rightarrow r}^{-1} \mathbf{r} \\ \mathbf{0}_{1 \times 3} & 0 \end{bmatrix} \quad (3.10a)$$

$$= \begin{bmatrix} \theta[\hat{\mathbf{k}}] & \left[\int_0^1 \exp(\theta[\hat{\mathbf{k}}] s) ds \right]^{-1} \mathbf{r} \\ \mathbf{0}_{1 \times 3} & 0 \end{bmatrix} \in \text{se}(3) \quad (3.10b)$$

And the ordered pair (θ, \mathbf{k}) in (3.10) is the angle-axis parameters of a rotation matrix \mathbf{R} . These parameters are given by:

$$\theta = \arccos\left(\frac{\text{Tr}(\mathbf{R}) - 1}{2}\right), \quad [\hat{\mathbf{k}}] = \frac{1}{2 \sin(\theta)} (\mathbf{R} - \mathbf{R}^T) \quad (3.11)$$

Angle θ is not unique when the matrix trace $\text{Tr}(\mathbf{R})$ is equal to (-1) , as $\theta = \pm\pi$. If the angle θ is not unique, then the logarithm in (3.10) is also not unique.

The $\mathbf{T}_{v \rightarrow r}^{-1} \mathbf{r}$ element in logarithm (3.10) is not needed for this deduction, but it is included for completeness. It is possible to compute this element from $\mathbf{T}_{v \rightarrow r} \mathbf{T}_{v \rightarrow r}^{-1} = \mathbf{I}_{3 \times 3}$, as shown in Condurache [16]:

$$\mathbf{T}_{v \rightarrow r}^{-1} = \left[\int_0^1 \exp(\theta[\hat{\mathbf{k}}] s) ds \right]^{-1} \quad (3.12a)$$

$$= \mathbf{I}_{3 \times 3} - \frac{1}{2}\theta[\hat{\mathbf{k}}] + \left[1 - \frac{\theta}{2} \cot\left(\frac{\theta}{2}\right) \right] [\hat{\mathbf{k}}]^2 \quad (3.12b)$$

Where $\mathbf{T}_{v \rightarrow r}$ is an integral of the Rodrigues' rotation matrix:

$$\begin{aligned} \mathbf{T}_{v \rightarrow r} &= \int_0^1 \exp(\theta[\hat{\mathbf{k}}] s) ds \\ &= \mathbf{I}_{3 \times 3} + \frac{1 - \cos(\theta)}{\theta} [\hat{\mathbf{k}}] + \frac{\theta - \sin(\theta)}{\theta} [\hat{\mathbf{k}}]^2 \end{aligned} \quad (3.13a)$$

$$\exp(\theta[\hat{\mathbf{k}}] s) = \mathbf{I}_{3 \times 3} + \sin(\theta s) [\hat{\mathbf{k}}] + [1 - \cos(\theta s)] [\hat{\mathbf{k}}]^2 \quad (3.13b)$$

Rotation R_X is assumed to be decoupled from the translation r_X in this solution to the hand-eye problem, and it is only the rotational part of (3.10) that is considered. This assumption simplifies (3.9) to its rotational part:

$$\|\theta_A [\hat{\mathbf{k}}_A]\| = \|\theta_B [\hat{\mathbf{k}}_B]\| \quad (3.14)$$

Which can be further simplified as (3.15b) for axis vectors of unit length and positive angles:

$$|\theta_A| = |\theta_B|, \quad \|\hat{\mathbf{k}}_A\|, \|\hat{\mathbf{k}}_B\| = 1 \quad (3.15a)$$

$$\theta_A = \theta_B, \quad \theta_A, \theta_B \geq 0 \quad (3.15b)$$

It is now simple to show the main result of Shiu and Ahmad [11] from (3.8b):

$$\theta_A \mathbf{k}_A = \mathbf{R}_X \theta_B \mathbf{k}_B \quad (3.16a)$$

$$\mathbf{k}_A = \mathbf{R}_X \mathbf{k}_B, \quad \theta_A = \theta_B \geq 0 \quad (3.16b)$$

Where they have used the identities:

$$\mathbf{R} \exp([\hat{\mathbf{k}}]) \mathbf{R}^T = \exp(\mathbf{R} [\hat{\mathbf{k}}] \mathbf{R}^T) \quad (3.17a)$$

$$\mathbf{R} [\hat{\mathbf{k}}] \mathbf{R}^T = [\mathbf{R} \hat{\mathbf{k}}] \quad (3.17b)$$

For any skew-symmetric matrix $[\hat{\mathbf{k}}]$. And $\mathbf{R} = \exp(\theta[\hat{\mathbf{k}}])$ is the angle-axis parametrization of rotation matrix \mathbf{R} :

$$\mathbf{R}_A = \mathbf{R}_X \mathbf{R}_B \mathbf{R}_X^T \quad (3.18a)$$

$$\exp(\theta_A [\hat{\mathbf{k}}_A]) = \mathbf{R}_X \exp(\theta_B [\hat{\mathbf{k}}_B]) \mathbf{R}_X^T = \exp(\theta_B [\mathbf{R}_X \hat{\mathbf{k}}_B]) \quad (3.18b)$$

$$\theta_A [\hat{\mathbf{k}}_A] = \theta_B [\mathbf{R}_X \hat{\mathbf{k}}_B] \quad (3.18c)$$

The result in (3.16b) implies that any exact solution to (3.8b) is independent of the angles θ_A and θ_B ; except for the special case where both axis vectors \mathbf{k}_A and \mathbf{k}_B are collinear. However, this result is not true in general, as the measured pairs of A and B are affected by noise. Therefore, the more general result in (3.16a) is used to estimate a solution to (3.8b) instead.

A least squares minimization problem (3.19) can be formulated from (3.16a) to estimate a solution to the hand-eye problem

(3.1). The objective is minimized over all the N measured pairs of A and B:

$$\hat{\mathbf{R}}_X = \arg \min_{\mathbf{R}_X} \sum_{i=1}^N \|\mathbf{R}_X \theta_B \mathbf{k}_B - \theta_A \mathbf{k}_A\|^2 \quad (3.19)$$

Similarly, by assuming that the estimated rotation $\hat{\mathbf{R}}_X$ is decoupled and known, a least squares minimization problem (3.20) can be formulated from (3.8d) to estimate the translation \mathbf{r}_X :

$$\hat{\mathbf{r}}_X = \arg \min_{\mathbf{r}_X} \sum_{i=1}^N \|(\mathbf{R}_{A_i} - \mathbf{I}_{3 \times 3}) \mathbf{r}_X + \mathbf{r}_{A_i} - \hat{\mathbf{R}}_X \mathbf{r}_{B_i}\|^2 \quad (3.20)$$

Separately solving the minimization problems in (3.19) and (3.20) is simpler than solving the coupled problem, and their optimal solutions do in fact have a closed form. The optimal rotation $\hat{\mathbf{R}}_X$ can be expressed explicitly as (3.21):

$$\hat{\mathbf{R}}_X = (\mathbf{M}^T \mathbf{M})^{-\frac{1}{2}} \mathbf{M}^T, \quad \mathbf{M} = \sum_{i=1}^n (\theta_{B_i} \mathbf{k}_{B_i})^T \theta_{A_i} \mathbf{k}_{A_i} \quad (3.21)$$

This rotation is unique if $\mathbf{M}^T \mathbf{M}$ is non-singular and has no repeated eigenvalues. These requirements will be satisfied in general, making the rotation $\hat{\mathbf{R}}_X$ unique [12]. The optimal translation $\hat{\mathbf{r}}_X$ is then also unique, and it is simply given by the linear least squares solution:

$$\hat{\mathbf{r}}_X = \mathbf{C}^\dagger \mathbf{d} = (\mathbf{C}^T \mathbf{C})^{-1} \mathbf{C}^T \mathbf{d} \quad (3.22a)$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_1 \\ \mathbf{C}_2 \\ \vdots \\ \mathbf{C}_n \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} - \mathbf{R}_{A_1} \\ \mathbf{I}_{3 \times 3} - \mathbf{R}_{A_2} \\ \vdots \\ \mathbf{I}_{3 \times 3} - \mathbf{R}_{A_n} \end{bmatrix} \in \mathbb{R}^{3n \times 3} \quad (3.22b)$$

$$\mathbf{d} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \vdots \\ \mathbf{d}_n \end{bmatrix} = \begin{bmatrix} \mathbf{r}_{A_1} - \mathbf{R}_X \mathbf{r}_{B_1} \\ \mathbf{r}_{A_2} - \mathbf{R}_X \mathbf{r}_{B_2} \\ \vdots \\ \mathbf{r}_{A_n} - \mathbf{R}_X \mathbf{r}_{B_n} \end{bmatrix} \in \mathbb{R}^{3n} \quad (3.22c)$$

Where \mathbf{C}^\dagger is the left Moore-Penrose inverse. The closed-form least squares solution by Park and Martin [12] can now be summarized as algorithm 1.

Algorithm 1: An implementation of the closed-form least squares solution by Park and Martin [12].

Data: Pairs of tool and sensor displacements ($N \geq 3$):

$$\{(\mathbf{A}, \mathbf{B})_1, (\mathbf{A}, \mathbf{B})_2, \dots, (\mathbf{A}, \mathbf{B})_N\}, \quad \mathbf{A}, \mathbf{B} \in \text{SE}(3)$$

Initialize $\mathbf{M} = \mathbf{0}_{3 \times 3}$, $\mathbf{C} = \mathbf{0}_{3N \times 3}$, $\mathbf{d} = \mathbf{0}_{3n}$;

for $i = 1$ **to** n **do**

$$\quad | \quad \mathbf{M} = \mathbf{M} + (\theta_{\mathbf{B}_i} \mathbf{k}_{\mathbf{B}_i})^\top \theta_{\mathbf{A}_i} \mathbf{k}_{\mathbf{A}_i};$$

end

Compute SVD of $\mathbf{M}^\top \mathbf{M} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$;

Compute rotation $\hat{\mathbf{R}}_X = \mathbf{U} \mathbf{\Sigma}^{-0.5} \mathbf{V}^\top \mathbf{M}^\top$;

for $i = 1$ **to** n **do**

$$\quad | \quad \mathbf{C}_i = \mathbf{I}_{3 \times 3} - \mathbf{R}_{\mathbf{A}_i};$$

$$\quad | \quad \mathbf{d}_i = \mathbf{r}_{\mathbf{A}_i} - \hat{\mathbf{R}}_X \mathbf{r}_{\mathbf{B}_i};$$

end

Compute translation $\mathbf{r}_X = (\mathbf{C}^\top \mathbf{C})^{-1} \mathbf{C}^\top \mathbf{d}$;

Result: Transformation from tool to sensor: $\hat{\mathbf{X}} = \begin{bmatrix} \hat{\mathbf{R}}_X & \hat{\mathbf{r}}_X \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix}$

Singular-value decomposition (SVD) was used to compute the symmetric and positive definite square root $(\mathbf{M}^\top \mathbf{M})^{-1/2}$.

3.2 QUATERNION AVERAGING

Quaternion averaging can be used to combine multiple pose measurements in a simple filter to reduce the influence of noise. Consider a set of n unit quaternions $Q = \{q_0, q_1, \dots, q_n\} \in \mathbb{H}^*$, where \mathbb{H}^* is given by (3.23), and S^3 is called the unit 3-sphere (4D unit sphere).

$$\mathbb{H}^* := \left\{ \mathbf{q} \mid \mathbf{q} = \begin{bmatrix} \eta & \boldsymbol{\epsilon}^\top \end{bmatrix}^\top, \quad \eta \in \mathbb{R}, \quad \boldsymbol{\epsilon} \in \mathbb{R}^3, \quad \mathbf{q}^\top \mathbf{q} = 1 \right\} \in S^3 \quad (3.23)$$

The goal is to find the average of these unit quaternions in an optimal manner. Markley et al. [17] derived and presented an algorithm that determines the optimal average from a set of weighted quaternions. They noted that there are mainly two challenges when averaging quaternions:

- Ensure that the resulting average is a unit quaternion:

$$\mathbf{q}^\top \mathbf{q} = 1 \quad (3.24)$$

- Handle the 2 to 1 homomorphism from unit quaternions to the 3D rotation group $SO(3)$, that is, q and $-q$ maps to the same rotation matrix $R(q)$

These challenges are solved by mapping the unit quaternions to rotation matrices $R(q)$ with (3.25), and then finding the average of these rotation matrices.

$$R(\mathbf{q}) = \left(\eta^2 - \boldsymbol{\epsilon}^\top \boldsymbol{\epsilon} \right) \mathbf{I}_{3 \times 3} + 2 \left(\boldsymbol{\epsilon} \boldsymbol{\epsilon}^\top - \eta [\hat{\boldsymbol{\epsilon}}] \right) \in SO(3) \quad (3.25a)$$

$$[\hat{\boldsymbol{\epsilon}}] := \begin{bmatrix} 0 & -\epsilon_3 & \epsilon_2 \\ \epsilon_3 & 0 & -\epsilon_1 \\ -\epsilon_2 & \epsilon_1 & 0 \end{bmatrix} \in so(3) \quad (3.25b)$$

The average quaternion q can then be found as the minimizer of a weighted sum of squared Frobenius norms, which induces a squared distance between the average q and the unit quaternions Q . This minimization problem is given by:

$$\mathbf{q} = \arg \min_{\mathbf{q} \in S^3} \sum_{i=1}^n w_i \|R(\mathbf{q}) - R(\mathbf{q}_i)\|_F^2 \quad (3.26)$$

This expression can be simplified as shown in (3.27), by using the definition of the Frobenius norm, properties of orthogonality ($\mathbf{R}^\top \mathbf{R} = \mathbf{R} \mathbf{R}^\top = \mathbf{I}_{3 \times 3}$), and the matrix trace ($\text{Tr}\{\mathbf{R}\} = \text{Tr}\{\mathbf{R}^\top\}$):

$$\|\mathbf{R}(\mathbf{q}) - \mathbf{R}(\mathbf{q}_i)\|_F^2 := \text{Tr} \left\{ [\mathbf{R}(\mathbf{q}) - \mathbf{R}(\mathbf{q}_i)]^\top [\mathbf{R}(\mathbf{q}) - \mathbf{R}(\mathbf{q}_i)] \right\} \quad (3.27a)$$

$$= \text{Tr} \left\{ [\mathbf{R}(\mathbf{q})]^\top \mathbf{R}(\mathbf{q}) - 2 \mathbf{R}(\mathbf{q}) [\mathbf{R}(\mathbf{q}_i)]^\top + [\mathbf{R}(\mathbf{q}_i)]^\top \mathbf{R}(\mathbf{q}_i) \right\} \quad (3.27b)$$

$$= 2 \text{Tr} \left\{ \mathbf{I}_{3 \times 3} - \mathbf{R}(\mathbf{q}) [\mathbf{R}(\mathbf{q}_i)]^\top \right\} = 6 - 2 \text{Tr} \left\{ \mathbf{R}(\mathbf{q}) [\mathbf{R}(\mathbf{q}_i)]^\top \right\} \quad (3.27c)$$

The simplified expression reveals that the original minimization problem (3.26) can be restated as the simpler maximization problem:

$$\mathbf{q} = \arg \max_{\mathbf{q} \in \mathcal{S}^3} \text{Tr} \left\{ \mathbf{R}(\mathbf{q}) \mathbf{B}^\top \right\}, \quad \mathbf{B} := \sum_{i=1}^n w_i \mathbf{R}(\mathbf{q}_i) \in \mathbb{R}^{3 \times 3} \quad (3.28)$$

Where \mathbf{B} is known as the *attitude profile matrix*, because it includes all the information that is known about the orientation [17]. This simplified problem is in a form that is found when solving Wahba's problem, whose goal is to find the optimal rotation matrix between two sets of vectors [18]. It is possible to use any solution from the literature on this problem, notably Davenport's q-method [19] and the computationally efficient QUaternion ESTimator (QUEST) algorithm [20]. The algorithm that is presented here, by Markley et al., uses Davenport's q-method as a basis for its solution.

The trace in (3.28) can be split into three simpler parts by using the mapping (3.25) for $\mathbf{R}(\mathbf{q})$:

$$\text{Tr} \left\{ \mathbf{R}(\mathbf{q}) \mathbf{B}^\top \right\} = \text{Tr} \left\{ \left[\left(\eta^2 - \mathbf{e}^\top \mathbf{e} \right) \mathbf{I}_{3 \times 3} + 2 \left(\mathbf{e} \mathbf{e}^\top - \eta [\hat{\mathbf{e}}] \right) \right] \mathbf{B}^\top \right\} \quad (3.29a)$$

$$= \left(\eta^2 - \mathbf{e}^\top \mathbf{e} \right) \text{Tr} \{ \mathbf{B} \} + 2 \text{Tr} \left\{ \mathbf{e} \mathbf{e}^\top \mathbf{B}^\top \right\} - 2\eta \text{Tr} \left\{ [\hat{\mathbf{e}}] \mathbf{B}^\top \right\} \quad (3.29b)$$

The fact that the matrix trace is invariant under cyclic permutations can then be used to further simplify the second part:

$$2 \text{Tr} \left\{ \mathbf{B} \mathbf{e} \mathbf{e}^\top \right\} = 2 \text{Tr} \left\{ \mathbf{e}^\top \mathbf{B} \mathbf{e} \right\} = 2 \mathbf{e}^\top \mathbf{B} \mathbf{e} = \mathbf{e}^\top \left(\mathbf{B} + \mathbf{B}^\top \right) \mathbf{e} \quad (3.30)$$

Evaluating the third part shows that it can be expressed as a dot product:

$$\text{Tr} \left\{ [\hat{\mathbf{e}}] \mathbf{B}^T \right\} = \text{Tr} \left\{ \begin{bmatrix} 0 & -\epsilon_3 & \epsilon_2 \\ \epsilon_3 & 0 & -\epsilon_1 \\ -\epsilon_2 & \epsilon_1 & 0 \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} \right\} \quad (3.31a)$$

$$= -\epsilon_3 B_{2,1} + \epsilon_2 B_{3,1} + \epsilon_3 B_{1,2} - \epsilon_1 B_{3,2} - \epsilon_2 B_{1,3} + \epsilon_1 B_{2,3} \quad (3.31b)$$

$$= \begin{bmatrix} \epsilon_1 & \epsilon_2 & \epsilon_3 \end{bmatrix} \begin{bmatrix} B_{2,3} - B_{3,2} \\ B_{3,1} - B_{1,3} \\ B_{1,2} - B_{2,1} \end{bmatrix} = \mathbf{e}^T [\mathbf{B} - \mathbf{B}^T]^\vee \quad (3.31c)$$

Where $[\cdot]^\vee$ is the inverse (vee) operator of (3.25b), and maps a skew-symmetric matrix to its vector representation. It is now possible to write the trace in a quadratic form:

$$\begin{aligned} \text{Tr} \{ \mathbf{R}(\mathbf{q}) \mathbf{B}^T \} &= \eta^2 \text{Tr} \{ \mathbf{B} \} + \eta ([\mathbf{B}^T - \mathbf{B}]^\vee)^T \mathbf{e} \\ &+ \mathbf{e}^T [\mathbf{B} - \mathbf{B}^T]^\vee \eta + \mathbf{e}^T (\mathbf{B} + \mathbf{B}^T - \text{Tr} \{ \mathbf{B} \}) \mathbf{e} \end{aligned} \quad (3.32a)$$

$$= \mathbf{q}^T \mathbf{K} \mathbf{q}, \quad \mathbf{K} := \begin{bmatrix} \text{Tr} \{ \mathbf{B} \} & ([\mathbf{B}^T - \mathbf{B}]^\vee)^T \\ [\mathbf{B} - \mathbf{B}^T]^\vee & \mathbf{B} + \mathbf{B}^T - \text{Tr} \{ \mathbf{B} \} \mathbf{I}_{3 \times 3} \end{bmatrix} \quad (3.32b)$$

The matrix is known as Davenport's K matrix, and by using the definition of B in (3.28), it can be simplified to:

$$\begin{aligned} \mathbf{K} &= \sum_{i=0}^n w_i \begin{bmatrix} 4\eta_i^2 - \mathbf{q}_i^T \mathbf{q}_i & 4\eta_i \mathbf{e}_i^T \\ 4\eta_i \mathbf{e}_i & 4\mathbf{e}_i \mathbf{e}_i^T - \mathbf{q}_i^T \mathbf{q}_i \mathbf{I}_{3 \times 3} \end{bmatrix} \\ &= \sum_{i=0}^n w_i \left[4\mathbf{q}_i \mathbf{q}_i^T - \mathbf{q}_i^T \mathbf{q}_i \mathbf{I}_{4 \times 4} \right] \end{aligned} \quad (3.33)$$

The part that is subtracted from this matrix is simply a constant diagonal matrix, as \mathbf{q}_i is a unit quaternion, and it does not change the solution. This results in the final representation of the maximization problem:

$$\mathbf{q} = \arg \max_{\mathbf{q} \in \mathbb{S}^3} \frac{1}{2} \mathbf{q}^T \mathbf{M} \mathbf{q}, \quad \mathbf{M} := \sum_{i=1}^n w_i \mathbf{q}_i \mathbf{q}_i^T \quad (3.34)$$

The Lagrangian of this problem and its derivative, where \mathbf{q} is subject to the constraint of being a unit quaternion, is given by:

$$\mathcal{L}(\mathbf{q}, \lambda) = \frac{1}{2} \mathbf{q}^T \mathbf{M} \mathbf{q} - \lambda (\mathbf{q}^T \mathbf{q} - 1) \quad (3.35a)$$

$$\nabla_{\mathbf{q}} \mathcal{L}(\mathbf{q}, \lambda) = \mathbf{M} \mathbf{q} - \lambda \mathbf{q} = 0 \quad (3.35b)$$

It is now simple to see that the optimal average is the eigenvector of M that corresponds to the largest eigenvalue. The first challenge of averaging quaternions is then solved by simply normalizing the eigenvector. The sign in front of \mathbf{q} does not change the solution of (3.34), and the second challenge is also solved.

3.3 CLOSED-FORM SOLUTION OF ABSOLUTE ORIENTATION

Assume that a set of 3D-points are measured in two different coordinate frames. The problem of finding the transformation that relates these frames is known as absolute orientation. Horn presented a closed-form solution to the least-squares problem of absolute orientation [21], which was simplified by representing the rotation as a unit quaternion. The solution starts with two sets of correspondent 3D-points:

$$\{\mathbf{a}_i\}, \{\mathbf{b}_i\} \quad \mathbf{a}_i, \mathbf{b}_i \in \mathbb{R}^3, i \in \{1, \dots, N\} \subset \mathbb{N} \quad (3.36)$$

Which are related by a rigid transformation from \mathbf{a}_i to \mathbf{b}_i :

$$\mathbf{b}_i = s\mathbf{R}\mathbf{a}_i + \mathbf{r}, \quad \mathbf{R} \in \text{SO}(3), \mathbf{r} \in \mathbb{R}^3 \quad (3.37)$$

Where s is a scalar scaling factor. The problem of finding this transformation can be stated as a least-squares minimization problem:

$$\min_{\hat{\mathbf{R}} \in \text{SO}(3), \hat{\mathbf{r}} \in \mathbb{R}^3} \sum_{i=1}^N \|\mathbf{b}_i - s\hat{\mathbf{R}}\mathbf{a}_i - \hat{\mathbf{r}}\|^2 \quad (3.38)$$

It is possible to simplify this problem by referring the points to their centroids:

$$\begin{aligned} \tilde{\mathbf{a}}_i &= \mathbf{a}_i - \bar{\mathbf{a}} \\ \tilde{\mathbf{b}}_i &= \mathbf{b}_i - \bar{\mathbf{b}} \end{aligned} \quad \bar{\mathbf{a}} = \frac{1}{n} \sum_{i=1}^N \mathbf{a}_i, \quad \bar{\mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathbf{b}_i \quad (3.39)$$

By exploiting the fact that the optimal translation is equal to the difference between the centroids, as expressed with aligned coordinates:

$$\hat{\mathbf{r}} = \bar{\mathbf{b}} - \hat{\mathbf{R}}\bar{\mathbf{a}} \quad (3.40)$$

Which results in the simplified sum:

$$\sum_{i=1}^N \|\tilde{\mathbf{b}}_i - s\hat{\mathbf{R}}\tilde{\mathbf{a}}_i\|^2 = \sum_{i=1}^N \tilde{\mathbf{b}}_i^T \tilde{\mathbf{b}}_i - 2s\tilde{\mathbf{b}}_i^T \hat{\mathbf{R}}\tilde{\mathbf{a}}_i + s^2\tilde{\mathbf{a}}_i^T \tilde{\mathbf{a}}_i \quad (3.41)$$

Maximizing the middle term of this result is the same as minimizing the least-squares problem (3.38):

$$\max_{\hat{\mathbf{R}} \in \text{SO}(3)} \sum_{i=1}^N s \tilde{\mathbf{b}}_i^T \hat{\mathbf{R}} \tilde{\mathbf{a}}_i \quad (3.42)$$

The simplified problem can then be expressed in a quadratic form, which does not depend on the scaling factor, by representing the optimal rotation with a unit quaternion $\hat{\mathbf{q}}$:

$$\max_{\mathbf{q} \in \mathbb{H}^*} \sum_{i=1}^N \left(\hat{\mathbf{q}}^T \begin{bmatrix} 0 & \tilde{\mathbf{a}}_i \end{bmatrix}^T \hat{\mathbf{q}}^* \right)^T \begin{bmatrix} 0 & \tilde{\mathbf{b}}_i \end{bmatrix}^T = \hat{\mathbf{q}}^T \mathbf{N} \hat{\mathbf{q}} \quad (3.43)$$

Where \mathbb{H}^* is the group of unit quaternions (3.23), \mathbf{q}^* is the quaternion conjugate $[\eta \ -\epsilon^T]^T$, and \mathbf{N} is a symmetric 4×4 matrix:

$$\begin{bmatrix} S_{xx} + S_{yy} + S_{zz} & S_{yz} - S_{zy} & S_{zx} - S_{xz} & S_{xy} - S_{yx} \\ S_{yz} - S_{zy} & S_{xx} - S_{yy} - S_{zz} & S_{xy} + S_{yx} & S_{zx} + S_{xz} \\ S_{zx} - S_{xz} & S_{xy} + S_{yx} & -S_{xx} + S_{yy} - S_{zz} & S_{yz} + S_{zy} \\ S_{xy} - S_{yx} & S_{zx} + S_{xz} & S_{yz} + S_{zy} & -S_{xx} - S_{yy} + S_{zz} \end{bmatrix} \quad (3.44)$$

The trace of this matrix $\text{Tr}(\mathbf{N})$ is equal to zero, and its elements are given by:

$$S_{mn} = \sum_{i=1}^N \tilde{\mathbf{a}}_{m,i}^T \tilde{\mathbf{b}}_{n,i}, \quad m, n \in \{x, y, z\} \quad (3.45)$$

Solving (3.43) is similar to the problem of solving (3.34), and the optimal solution is the eigenvector of \mathbf{N} that corresponds to the largest eigenvalue. The optimal rotation $\hat{\mathbf{R}}(\mathbf{q})$, as given by (3.25), can then be used to solve for the translation $\hat{\mathbf{t}}$ in (3.40). Horn et al. [21] also presented an optimal scaling factor, which does not depend on the rotation or translation:

$$\hat{s} = \sqrt{\frac{\sum_{i=1}^N \|\mathbf{a}_i\|^2}{\sum_{i=1}^N \|\mathbf{b}_i\|^2}} \quad (3.46)$$

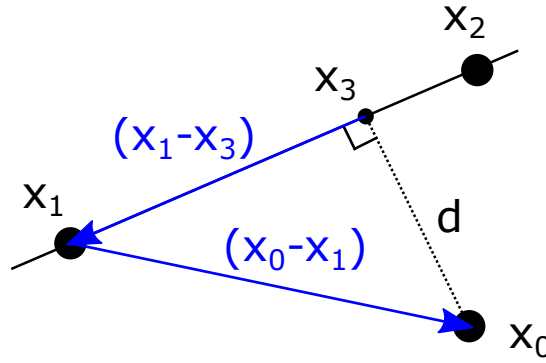


Figure 3.3: A point x_0 with perpendicular distance d from a line that is defined by the points x_1 and x_2 .

3.4 PERPENDICULAR DISTANCE FROM A POINT TO A LINE

The vector from a line to a point in \mathbb{R}^3 , which is perpendicular to the line, has the shortest possible length from the line to the point. Figure 3.3 shows this length as the perpendicular distance d from a point x_3 on the line to a point x_0 . This perpendicular distance and its corresponding vector will be used to define unique geometric primitives in 3D space. The line can be parameterized from two points x_1 and x_2 on the line:

$$t \frac{(x_2 - x_1)}{\|x_1 - x_2\|} + x_1, \quad t \in \mathbb{R} \quad (3.47)$$

The point on this line that has the shortest distance to x_0 , can be found from the projection of vector $(x_0 - x_1)$ onto the line. This projection results in a parameter t as the dot product:

$$t = (x_0 - x_1)^T \frac{(x_2 - x_1)}{\|x_2 - x_1\|} \quad (3.48)$$

Substituting this parameter into equation (3.47) gives the point:

$$x_3 = \left[\frac{(x_0 - x_1)^T (x_2 - x_1)}{\|x_2 - x_1\|} \right] \frac{(x_2 - x_1)}{\|x_1 - x_2\|} + x_1 \quad (3.49)$$

The perpendicular distance vector d is then given by:

$$d = (x_0 - x_3) = \left[\frac{(x_1 - x_0)^T (x_1 - x_2)}{\|x_1 - x_2\|} \right] \frac{(x_1 - x_2)}{\|x_1 - x_2\|} + (x_0 - x_1)$$

(3.50)

Which can be expressed as the composition of vectors $(\mathbf{x}_1 - \mathbf{x}_3)$ and $(\mathbf{x}_0 - \mathbf{x}_1)$, as shown in figure 3.3.

The dot product of a vector with itself gives its squared length:

$$\begin{aligned}
 \|\mathbf{d}\|^2 = \mathbf{d}^T \mathbf{d} &= \left[\frac{(\mathbf{x}_1 - \mathbf{x}_2)^T (\mathbf{x}_1 - \mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \right]^2 + \|\mathbf{x}_0 - \mathbf{x}_1\|^2 \\
 &+ \left[\frac{(\mathbf{x}_1 - \mathbf{x}_2)^T (\mathbf{x}_1 - \mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} \right] 2 (\mathbf{x}_1 - \mathbf{x}_2)^T (\mathbf{x}_0 - \mathbf{x}_1) \\
 &= \frac{\|\mathbf{x}_0 - \mathbf{x}_1\|^2 \|\mathbf{x}_1 - \mathbf{x}_2\|^2 - [(\mathbf{x}_0 - \mathbf{x}_1)^T (\mathbf{x}_1 - \mathbf{x}_2)]^2}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} \\
 &= \frac{\|(\mathbf{x}_0 - \mathbf{x}_1) \times (\mathbf{x}_1 - \mathbf{x}_2)\|^2}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2}
 \end{aligned} \tag{3.51}$$

Where the expression has been simplified with the scalar quadruple product identity:

$$\|\mathbf{a} \times \mathbf{b}\|^2 = \|\mathbf{a}\|^2 \|\mathbf{b}\|^2 - (\mathbf{a}^T \mathbf{b})^2, \quad \mathbf{a}, \mathbf{b} \in \mathbb{R}^3 \tag{3.52}$$

Taking the square root of the resulting expression gives the beautiful equality:

$$\|\mathbf{d}\| = \frac{\|(\mathbf{x}_0 - \mathbf{x}_1) \times (\mathbf{x}_1 - \mathbf{x}_2)\|}{\|\mathbf{x}_1 - \mathbf{x}_2\|} = \frac{\|(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_1)\|}{\|\mathbf{x}_2 - \mathbf{x}_1\|} \tag{3.53}$$

HTC VIVE

The Vive is a room-scale VR system, allowing the user to freely walk around in a diagonal area of up to 5 m and interact with an environment. Steuer [22] defined VR as “a real or simulated environment in which a perceiver experiences telepresence”, where telepresence is defined as “the experience of presence in an environment by means of a communication medium”. In this work, the environment is a virtual representation of a robot cell.

The main medium is in this case a head-mounted display (HMD) that is worn on the user's head. This HMD immerses the user in a visualized environment, by displaying a 3D image through lenses in the HMD. There are also controllers and trackers available that allow the user to interact directly with the environment. The trackers are functionally similar to controllers, but they have a smaller puck-like form factor and there are no connected inputs, such as buttons.¹

Viewing the virtual robot cell through the HMD was not considered as a part of this work, as the main purpose was to explore the positioning capabilities of the tracked devices. However, an open-source plugin already exists to display the 3D view of RViz in the HMD [23].

4.1 LIGHTHOUSE TRACKING

The technology that enables the Vive to track devices in a room-scaled environment is called lighthouse tracking. This outside-in technology sweeps the room horizontally and vertically with 850 nm infrared (IR) laser lines from one or multiple stationary base stations in the room. These base stations contain a pair of DC motors that spin at a rate of 120 Hz, where one motor is turned by 90° and phase shifted by 180° from the other motor. In other words, each motor takes turns sweeping the room horizontally and vertically through a wheel-mounted fresnel lens with a field of view of 120°. The base stations also take turns sweeping the room in a similar manner, where one of the base stations acts as a master (a or b mode) and the other acts as a slave (c mode). This gives the tracking system an update rate of 60 Hz.

The base stations contain an array of 15 IR LEDs, which floods the tracking volume with 1.8 MHz modulated pulses at the start of each sweep. These pulses are used for time synchronization and

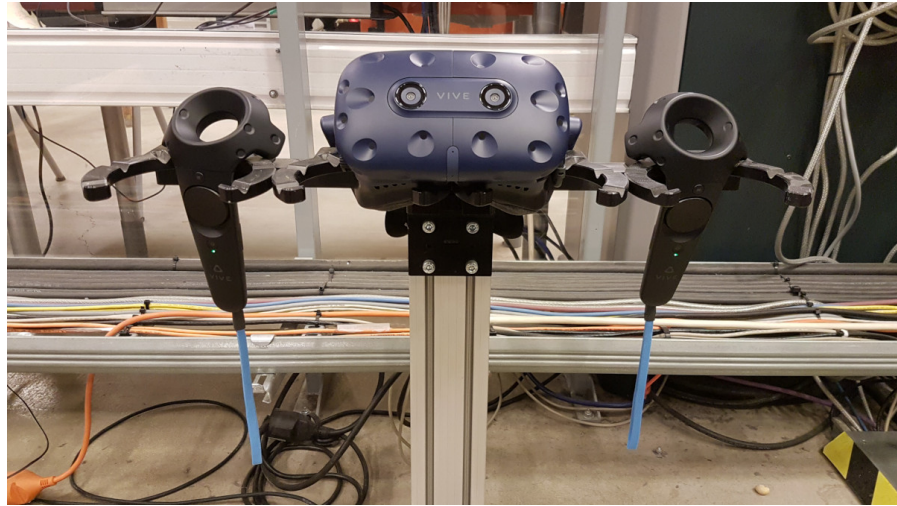


Figure 4.1: Vive Pro head-mounted display (HMD) and one controller for each hand [4].

transmission of *omnidirectional optical transmitter* (OOTX) data. This data provides the tracked devices with identification of the base stations, their factory calibration data and current status [24].

The tracking works by measuring the time difference between synchronization pulses and line sweeps, as perceived from surface-mounted IR photodiodes on the tracked devices. This time difference is used to compute the angle of a base station's motor, and computing both motor angles gives the intersection between two perpendicular planes, a line. The line is projected from a base station towards an IR photodiode on the surface of a tracked device.

In normal operation, multiple IR photodiodes are hit by the laser during a single line sweep, and multiple projected lines are computed. The positions of the IR photodiodes along these lines are unknown, but the geometric relationship between the IR photodiodes are known. Estimating the pose of a tracked device based on this knowledge is similar to the Perspective-n-Point (PnP) problem [25], where n is the number of 3D points (IR photodiode positions). Solving this problem is equivalent to finding the transformation that maps points from a local device frame to a global tracking frame, which respects the constraints that are given by the projected lines.¹

The tracked devices also contains an inertial measurement unit (IMU), which provides faster updates from their relative motion in between the updates from the lighthouse tracking. The poses of the tracked devices are updated at 220-360 Hz depending on the type of device [26]. The measurements are sub-sampled in the Vive Bridge package at a rate of 120 Hz by default.

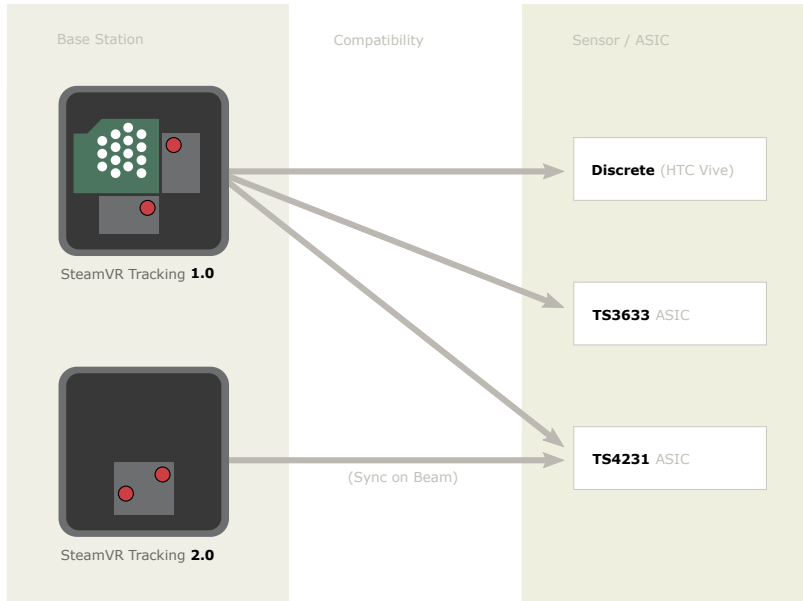


Figure 4.2: Device compatibility between SteamVR 2.0 and 1.0 tracking [27].

4.1.1 Lighthouse tracking 2.0

The description that is provided about the lighthouse tracking in section 4.1, only explains how the first generation of the technology works. There are two major differences with how the tracking works in the second revision:

- There is only one motor with two wheel-mounted fresnel lenses, which are arranged in such a way that their projected laser lines do not intersect with each other.
- The synchronization pulse is removed, and the (OOTX) data and time synchronization is modulated directly on the laser lines instead.

These changes have the consequence that the second revision of the base stations are more reliable (less components), but the tracked devices also have to support the 2.0 tracking. New revisions of the tracked devices (2018) support both versions, as shown in figure 4.2.

Performing synchronization on the laser beam increases the diagonal of the tracking area to 10 meters. It is possible to increase the tracking area even further by introducing more base stations, which also reduces the risk of losing tracking.

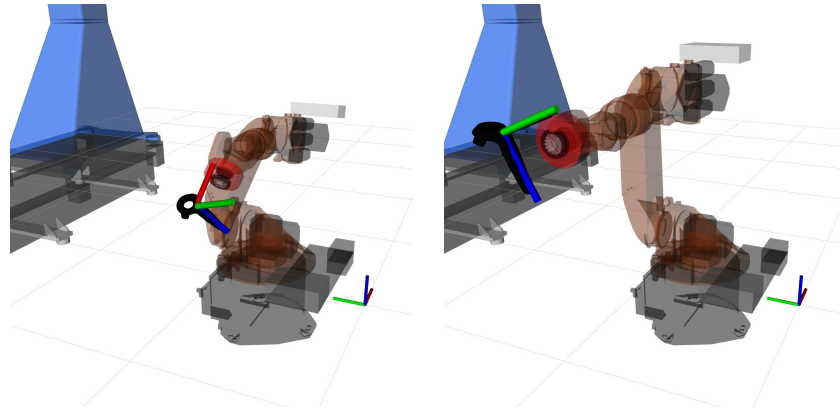


Figure 4.3: The robot was moved quickly between two states in order to test the tracking dynamics.

4.2 ACCURACY AND PRECISION

The lighthouse tracking is advertised with sub-millimeter precision, which is an impressive feat for a consumer-grade product such as the Vive. Niehorster et al. [28] tested the Vive HMD, and compared its accuracy and precision to a research-grade tracking system. They showed that the average positioning error was 17 mm with 9 mm standard deviation, and the RMS noise levels was below 0.2 mm and 0.02°.

Borges et al. [29] showed a similar precision as Niehorster, but also tested the dynamic accuracy and precision for robotics applications. The dynamic accuracy was shown to be in the millimetric to metric range with a best case of 2.36 mm and a worst case of 0.80257 m.¹

4.3 TRACKING ISSUES

Two problems related to specific issues of the Vive's tracking system were identified in the specialization project [4]:

1. Niehorster et al. [28] reported that poses measured with the Vive are provided in a reference frame, which is tilted with respect to the physical ground plane.
2. Borges et al. [29] showed that the Vive's tracking algorithm gives greater weight to its inertial measurements, in order to generate smooth trajectories for VR applications.

This prioritization of inertial measurements can clearly be seen in figure 4.4, where a tracked device was moved quickly between two points as shown in figure 4.3. The figure shows

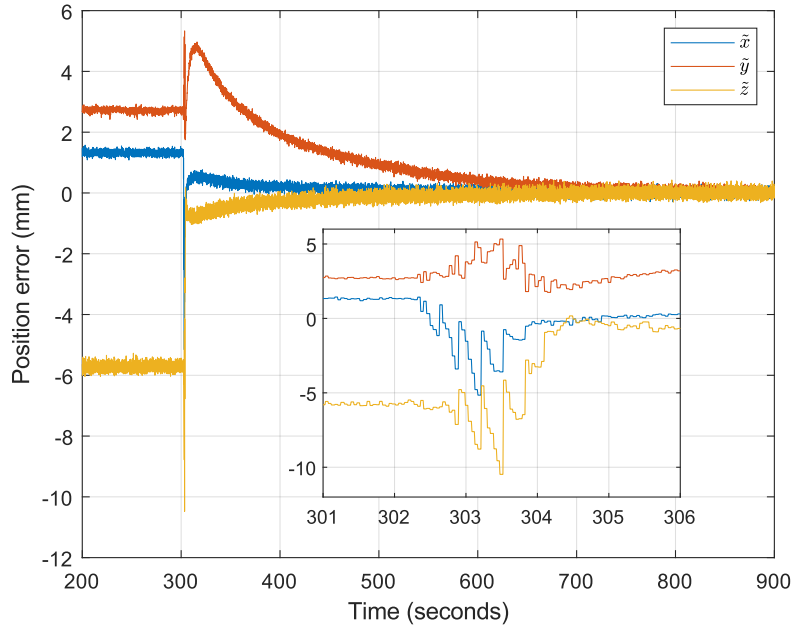


Figure 4.4: Error response from moving a tracked device quickly between two points. The move starts at 302.4 s, and it lasts approximately 2 s.

that the error is converging a lot faster when the tracked device is moving, which then slowly approaches its final value with an overdamped (second order) impulse response. Convergence can take as long as 500 seconds, and causes an error in the millimetric range when measuring the position of a device before it has converged. Figure 4.4 also shows that the steady-state error of a tracked device depends on its location.

4.4 MINOR ISSUES

There are a few minor inconveniences with the default SteamVR settings, and it is not immediately obvious how some of these settings can be changed. It can be especially hard to change settings on the Linux version of SteamVR, where the user interface does not always work as intended. Some settings are also “hidden” within multiple layers of virtual reality menus. Luckily, SteamVR provides a local web console when it is running:

<http://localhost:8998/console/index.html>

Where all the settings can be changed.

4.4.1 Controller timeout

The default behaviour of the Vive controllers is to timeout and turn off after 5 minutes without interaction. This behaviour can be impractical for calibration purposes, where the Vive controller can turn off between or during measurements. The controller has to be turned on manually by the user after a timeout, which is not optimal when there is a safety fence between the user and the Vive controller. However, it is easy to disable this timeout with a single command in the web console:

```
settings power.turnOffControllersTimeout 0
```

4.4.2 Tracker roles

The Vive Trackers are general purpose devices, and can take on different roles. This role can change abruptly with the default behaviour of the trackers (held in hand). A tracker can take on the role of a Vive controller, effectively replacing the spot of an existing controller, if SteamVR thinks you are holding the tracker in one of your hands.

The role switching of Vive trackers is currently not supported within the Vive Bridge package, and results in trackers that behaves like controllers after switching their role. An existing controller may stop working as only two controllers are allowed at a time. This behaviour is undesired, and the easiest fix is to disable the role switching altogether, with the instructions that are shown below:

1. List tracker settings in the web console with the following command:

```
settings trackers./devices/htc/vive_trackerLHR-
```

This command should return the current tracker roles:

```
Mon Apr 08 2019 16:05:39.851 - [Console] trackers./
devices/htc/vive_trackerLHR-4AB25273: "TrackerRole_
Handed,TrackedControllerRole_Invalid"
Mon Apr 08 2019 16:05:39.851 - [Console] trackers./
devices/htc/vive_trackerLHR-9C8FB0EC: "TrackerRole_
Handed,TrackedControllerRole_Invalid"
```

2. Note the serial numbers 4AB25273 and 9C8FB0EC that identify the trackers. The role switching of these trackers is then disabled with the following commands:

```
settings trackers./devices/htc/vive_trackerLHR-4AB25273
TrackerRole_None
settings trackers./devices/htc/vive_trackerLHR-9C8FB0EC
TrackerRole_None
```

This page is intentionally left blank.

VIVE-ROBOT CELL SETUP AND CALIBRATION

In order to use the Vive's tracking system for robot cell calibration, the pose of the tracked devices must be known relative to the robot and its environment. This chapter is primarily concerned with how a calibration procedure can be established for the tracking system.

The base stations have to be securely set up in the robot cell before a calibration can be performed. Tripods could be used to install the base stations in a temporary setup for robot cell calibration tasks. These tripods have to be extended to over two meters above the ground with standard quarter inch UNC threaded camera mounts.

The tripods takes up space and could be moved around unintentionally in the robot cell, invalidating the last calibration. Tripods are unsuited for experimental purposes, where it is important that the setup does not change between experiments. The base stations were for this reason firmly mounted in a fixed setup for Thivaldi, as shown in figure 5.1. A setup should follow the recommendations of the official Vive support [30], where the base stations should be:

- Above head height, ideally more than 2 m above ground
- Angled down between 30 and 45 degrees

These recommendations were followed for Thivaldi by mounting a pair of first-generation base stations three meters above the ground, facing 45° down towards the center of the robot cell. The base stations have a field of view of 120° , leaving $30\text{--}45^\circ$ for adjustment. The optimal setup is not strict in the sense that the base stations have to be placed perfectly. Most importantly, the base stations should be placed such that their view of each other and the robot cell is unobstructed. It is also important that their field of view overlaps as much as possible within the intended tracking volume.

Figure 5.2 shows the tracking area of the current setup for Thivaldi, which covers most of the robot cell. This setup has no problems tracking the area under and in front of the gantry.

5.1 INTERNAL VIVE CALIBRATION

The setup has to be internally calibrated with SteamVR. This calibration is done via the room setup tool, where a room-scale setup is

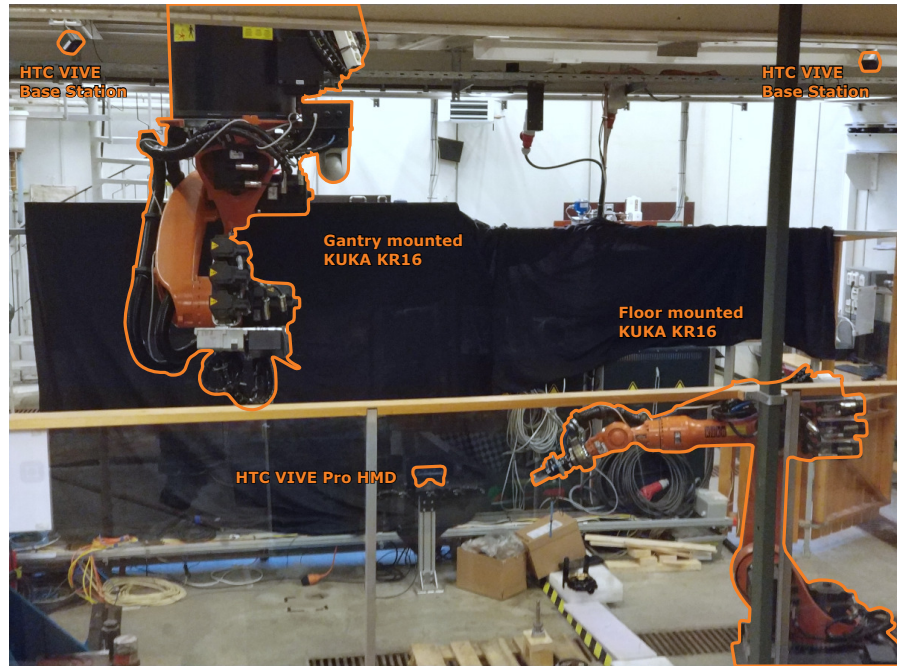


Figure 5.1: Setup of the Vive’s tracking system in a robot cell. The base stations are rigidly mounted on aluminum brackets below h-beams in the roof structure.

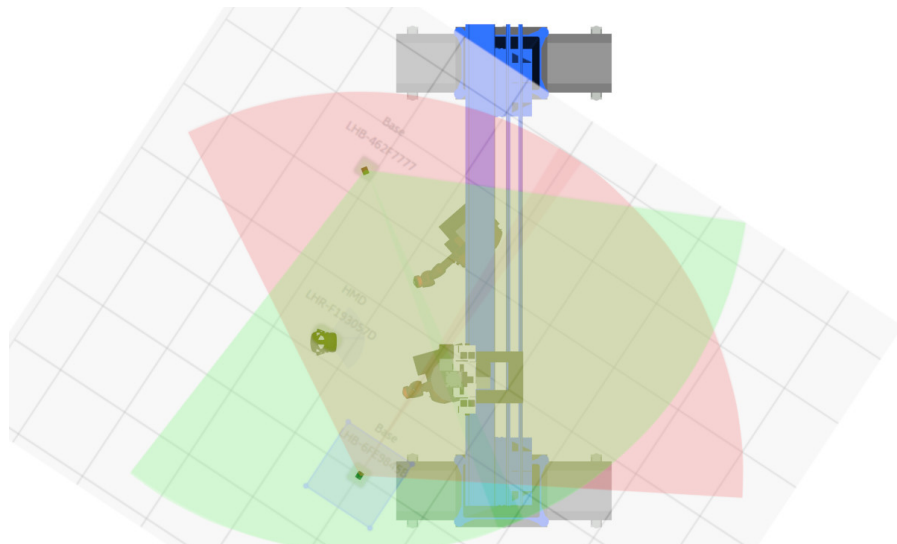


Figure 5.2: Plan view of the robot cell, where the base stations’ field of view is shown in red (master) and green (slave). Each square corresponds to one square meter [4].

performed. The calibration process is completed by following simple on-screen instructions and prompts. Both controllers are put on the ground to calibrate the floor, and the boundary of the tracking area is traced with a Vive controller. The traced boundaries are used in a system called Chaperone, in order to warn the user about physical obstructions. This system displays a grid within the HMD whenever the user approaches the boundary. It is important for user safety in VR applications using the HMD.¹

5.2 EXISTING CALIBRATION PROCEDURE

One-to-one mapping from a robot cell to its virtual counterpart was established in [4] by finding a spatial relationship between the inertial frames of robot cell and tracking system. This relationship was found by employing hand-eye calibration; in order to estimate the rigid transformation \hat{X} between a tracked device that was firmly attached to the robot's gripper and an arbitrary tool frame of the robot.

The hand-eye calibration was solved by using a closed form solution by Park and Martin [12]. Their method was derived in section 3.1 as algorithm 1, where the input is at least three measured pairs of homogeneous transformation matrices $(A_i, B_i) \in SE(3)$. These pairs are defined in (3.2) as the deviation between consecutive samples of tool and sensor poses.

5.2.1 Generating sample poses for calibration

Tool poses are generated for sampling the necessary poses with a robot. Their position is selected at random within a spherical volume element, and their orientation is picked from the normal vector at this position as from the surface of a sphere. The positions can take on any value within the spherical volume element that is parameterized as shown in figure 5.3. Each tool pose can then be represented as a homogeneous transformation from the robot's base {b} to tool {t} frame:

$$\mathbf{T}_b^t = \begin{bmatrix} \mathbf{R}_{z,\theta}\mathbf{R}_{x,\phi} & \mathbf{R}_{z,\theta}\mathbf{R}_{x,\phi} \begin{bmatrix} 0 & 0 & r \end{bmatrix}^T \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix} \quad (5.1)$$

Where $\mathbf{R}_{x,\phi}$ and $\mathbf{R}_{z,\theta}$ are basic rotations about the x-axis and z-axis by an angle ϕ and θ respectively, and r is the radius of a sphere.

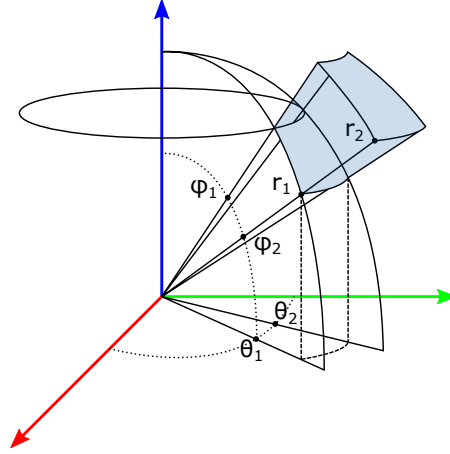


Figure 5.3: Spherical volume element that is defined by the spherical coordinates ($r \in [r_1, r_2]$, $\theta \in [\theta_1, \theta_2]$, $\phi \in [\phi_1, \phi_2]$) in a right-handed coordinate system.

Tsai et al. [31] observed that the rotation between consecutive sensor poses, and the translation between consecutive tool poses should be maximized and minimized respectively; in order to improve the accuracy of the hand-eye calibration. Thus, the translations and rotations are generated from two sets of parameters in smaller and larger ranges about the same point. This method of randomly generating tool poses for sampling within a range is flexible and generalizes well for different setups.

5.2.2 Computing the mapping

The mapping from robot cell $\{rc\}$ to tracking system $\{vr\}$ is computed from the final samples of tool and sensor poses, and the estimated solution \hat{X} from solving the hand-eye calibration:

$$\mathbf{T}_{rc}^{vr} = (\mathbf{T}_t^{rc})^{-1} \hat{X} \mathbf{T}_s^{vr}, \quad \hat{X} = \hat{\mathbf{T}}_t^s \quad (5.2)$$

The resulting transformation is then used to calibrate the system by automatically updating the corresponding relationship in the transform tree. This relationship is updated by changing x , y , z , roll, pitch and yaw parameters in the Vive bridge node through its dynamic reconfigure interface.

5.2.3 Performing the calibration for Thrivaldi

A Vive controller was firmly attached to the gripper of the floor mounted robot, as shown in figure 5.4. And 51 tool poses were



Figure 5.4: Vive controller firmly attached to the robot's gripper.

generated in the range ($r \in [1.4, 1.6]$, $\theta \in [-5\pi/16, -3\pi/16]$, $\phi \in [\pi/8, 3\pi/16]$) for positions and range ($r \in [1.4, 1.6]$, $\theta \in [0, -\pi/2]$, $\phi \in [5\pi/16, 11\pi/16]$) for orientations, with origin at the robot base. This range corresponds to sampling sensor poses in close proximity to the tool pose of the floor mounted robot in figure 5.5. The synthetic tests in [12] suggests that this number of samples should result in a solution that is close to convergence. A wait time of 20 seconds was used in between each sample, in order for the tracking dynamics to settle within a reasonable range of a few millimeters.

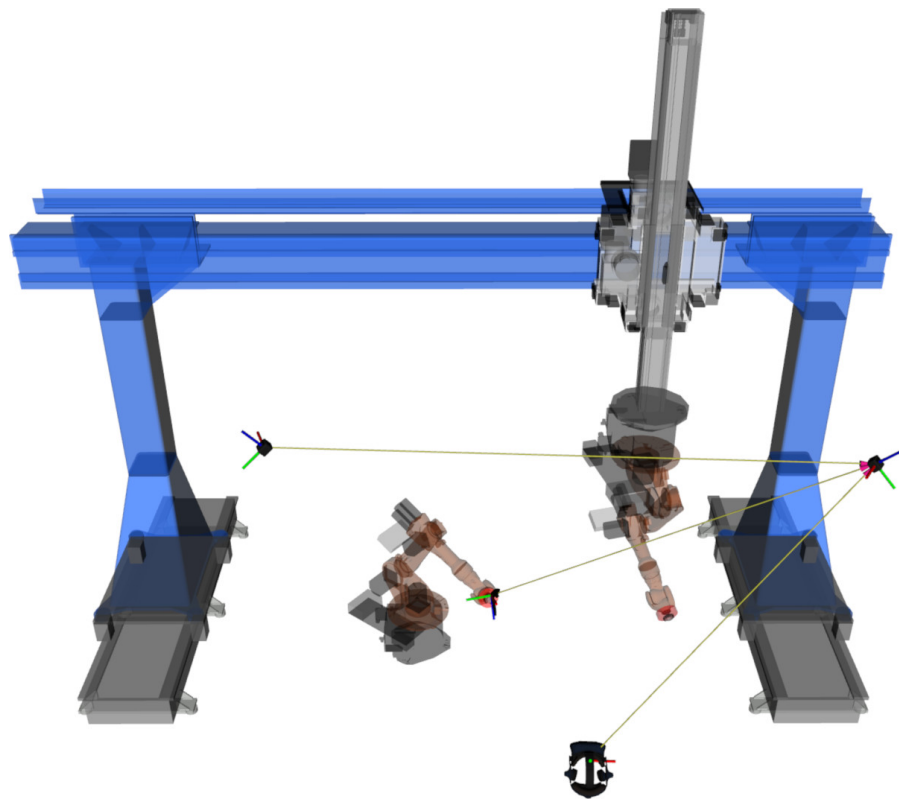


Figure 5.5: Virtual representation of the robot cell after calibration, as visualized in RViz.

IMPROVING THE CALIBRATED SYSTEM

The calibrated system has a few problems that are related to specific issues of the lighthouse tracking. This chapter will focus on trying to understand these issues, and come up with potential solutions that could improve the tracking performance in the robot cell. A special emphasis will be placed on the spatially dependant bias, as it is the biggest hurdle to overcome for sub-centimetric accuracy.

Three different approaches were tested in order to improve the accuracy and precision of the calibrated system:

1. Improve the existing calibration procedure.
2. Map the spatially dependant bias within the tracking volume, and investigate the possibility of correcting this error.
3. Implement the libsurvive library as an alternative to Valve's SteamVR and OpenVR SDK.

6.1 IMPROVING THE EXISTING CALIBRATION PROCEDURE

6.1.1 *Sampling procedure*

The measured poses of a tracked device are subject to noise in the sub-millimetric and sub-degree range. This noise may cause a small but visible discrepancy in the mapping between the robot cell and tracking system. It is easy to remove this discrepancy under a few assumptions about the measurements:

- The device is stationary when measuring its pose, such that the zero-frequency component (mean) and inherent noise are the only signal properties that are measured.
- The sampling frequency is constant, such that the weight of each measurement is equal.

A simple filter to remove the noise under these assumptions is the average of multiple measurements. Averaging the position of a tracked device is trivial, but how the orientation should be

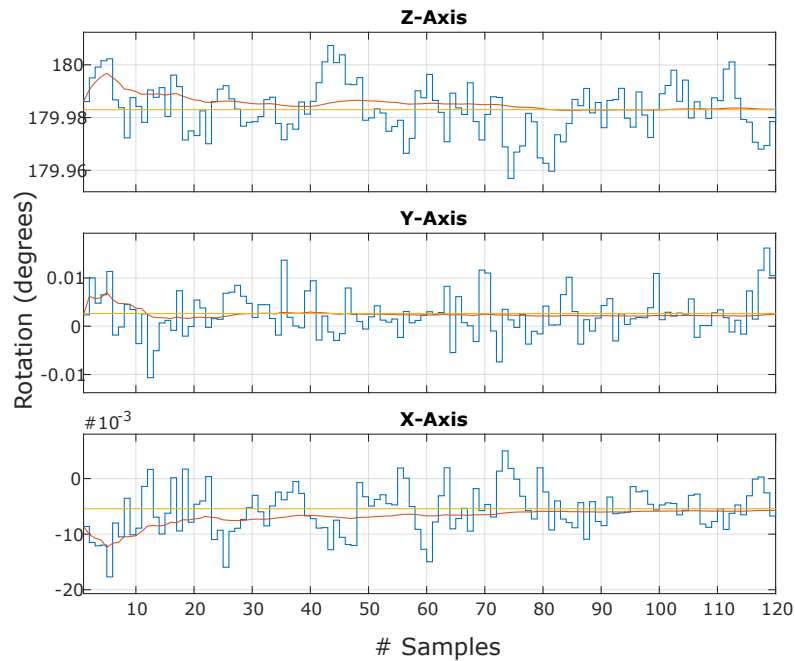


Figure 6.1: An example that shows the stationary orientation of a Vive controller, which has been converted into the blue ZYX Euler angles. The average of this orientation is shown in red with an increasing number of samples to the right, and the yellow lines are the result of averaging 1000 samples.

averaged is not obvious. There are many different representations of an orientation, and many different methods of averaging a set of orientations exists. Quaternion averaging was chosen because of the fact that quaternions are the de-facto method of representing orientations in ROS.

A method of averaging quaternions was derived from a paper by Markley et al. [17] in section 3.2. The method results in figure 6.1, where the noise is reduced by almost two orders of magnitude with 120 samples. Employing this method on the measurements removed the discrepancy when calibrating the system.

6.1.2 Reducing the mapping error

It was noted in [4] that the steady-state error of a tracked device depends on its location. This spatial dependency makes the mapping from the existing calibration procedure accurate at only one location: the final sensor pose, where system was calibrated. It should be possible to reduce this error by computing an average of the mapping from multiple poses.

A transformation between the inertial frames of robot cell {rc} and tracking system {vr} can be computed for each of the measured poses from the hand-eye calibration:

$$\mathbf{T}_{rc}^{vr} = (\mathbf{T}_t^{rc})^{-1} \hat{\mathbf{X}} \mathbf{T}_s^{vr}, \quad \hat{\mathbf{X}} = \hat{\mathbf{T}}_t^s \quad (6.1)$$

These transformations can then be averaged in the same way as the sampled poses in section 6.1.1. For simplicity, the measurements from the hand-eye calibration were used to compute the averaged mapping. This average resulted in a minor accuracy improvement in the millimetric range.

Measurements from the hand-eye calibration are not suited for computing an average, as the translations between consecutive tool poses should be minimized in order to improve the accuracy of the solution [31]. It is possible that a larger sampling volume could additionally reduce the error. However, this was not tested due to time constraints and will have to be assessed in future work.

6.1.3 Nonlinear optimization step

The closed form solution to the hand-eye calibration is based on a decoupled solution, where the rotation is assumed to be decoupled from the translation. Although algorithm 1 is robust with many measurements, solving the rotational part first before using it to solve the translational part propagates an error to the translation. An extra optimization step was added to reduce this error, where the following cost function was minimized:

$$\min_{\mathbf{X} \in \text{SE}(3)} \sum_{i=1}^N \rho_i, \quad \rho_i = \log \left((\mathbf{A}_i \mathbf{X})^{-1} \mathbf{X} \mathbf{B}_i \right) \in \text{se}(3) \quad (6.2)$$

Here, N is the number of measured pairs (A_i, B_i) , $(\cdot)^{-1}$ is the $\text{SE}(3)$ group inverse and $\log(\cdot)$ is the matrix logarithm, which maps elements in the group of rigid transformations $\text{SE}(3)$ into elements of its tangent space $\text{se}(3)$. The residual is represented as a vector with 6 elements.

The cost function can be defined in a straightforward way by using the Sophus library. A functor is an object that behaves like a function by overloading its $()$ -operator, and it is defined as shown in listing 6.1 to evaluate the residual ρ_i in (6.2) for the non-linear least squares Ceres solver [9].

Note that the residuals in listing 6.1 are computed directly from their definition in (6.2), and Sophus uses the $SE(3)$ group operations to compute them and their derivatives. A least squares problem is then defined by adding a residual block for each of the N measured pairs to a Ceres problem object, as shown in listing 6.2.

The residual blocks are defined from the cost functor with measured pairs $(A_i, B_i) \in SE(3)$ as its input, and the closed form solution to the hand-eye calibration is used as an initial guess for the solver. The problem is then solved by the solver, and details about the termination state of the solver are printed to the console. A comparison between the solutions before and after the optimization step are also printed to the console as homogeneous transformation matrices:

```
[ INFO] [1554998264.773617329]: Before optimization:
-0.0664468  -0.384709   0.920643  -0.0703492
-0.00455076  0.92279   0.385277  -0.0167372
  -0.99778   0.0214108 -0.0630671   0.224237
           0           0           0           1

[ INFO] [1554998264.814470110]: After optimization:
-0.0665727  -0.384724   0.920628  -0.0703145
-0.00395554  0.922769   0.385333  -0.0168156
  -0.997774  0.0220111 -0.062953   0.224193
           0           0           0           1
```

Although the resulting change is small (within a sub-millimetric range for the position); the optimization step significantly reduced the error in (6.2) when using fewer than 11 samples. This finding indicates that the implemented hand-eye calibration is robust, and it does not cause the spatially dependent error that is observed with the calibrated system.

6.2 MAPPING THE ERROR WITH A ROBOT

An automated sampling procedure was created in order to measure the offset error within a section of the tracking volume. This procedure uses the same setup as the calibration procedure, with a tracked device firmly attached to the robot.

The forward kinematics of the robot was used as a ground truth for the measurements, as the robot was assumed to be at least two orders of magnitude more accurate than the lighthouse tracking. This assumption is not necessarily true, since the specification that is provided for the robot is a measure of its repeatability, not a guarantee that its Denavit–Hartenberg

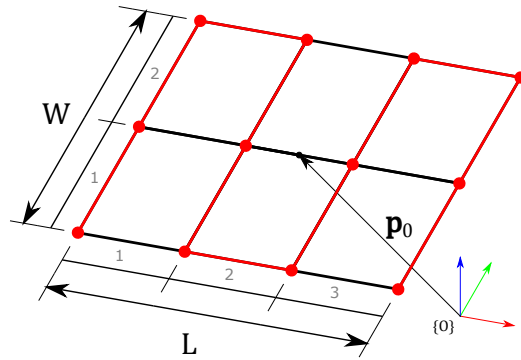


Figure 6.2: A 2×3 point lattice with total length L and total width W , where the center of the lattice has been offset by an amount $p_0 \in \mathbb{R}^3$ with respect to an arbitrary reference frame $\{O\}$.

parameters are correct. The Denavit–Hartenberg parameters that are used in the KRC appears standard from the design with nominal values, and not individually calibrated to each robot.

6.2.1 Generating a set of sampling poses

The sampling procedure consists of a set of points that are generated from $m \times n$ point lattices, as shown in figure 6.2. Each lattice is defined by its total length L and total width W , which are split into a grid of $m \times n$ rectangles. The points are then located at intersections between the perpendicular grid lines of these rectangles. Each rectangle has sides of length L/m and width W/n , which defines the distances between consecutive points in the lattice.

The points are generated iteratively from left to right, alternating between up and down. It is also possible to reverse the order of these points, such that points in layers of alternating order can be stacked in a contiguous manner. This stacking is accomplished by moving each layer a given amount (x_0, y_0, z_0) relative to a shared reference frame. The algorithm for generating a single layer of such a point lattice can then be summarized as algorithm 2.

Each point defines a sensor position that should be sampled with the robot, which has to approach the positions with a reachable orientation. The positions have to be mapped to a tool pose that the robot can reach, in order to utilize the robot's workspace. Poses are defined such that the tracked device is radially aligned with the robot facing it, and horizontally aligned

with the point lattice. That is, the poses are aligned with the robot base but orientated for best reachability.

The desired sensor orientation can be found by simply rotating a predefined orientation of the tracked device, which corresponds to the robot base at a zero degrees angle. This orientation depends on how the tracked device is attached to the robot, and the calibration setup for Thrivaldi results in a downwards facing z -axis relative to the robot base. The desired rotation for this setup is then given by a basic rotation about the z -axis by an angle:

$$\theta = -\text{atan2}(y, x) \quad (6.3)$$

Here, it is assumed that the 2D-coordinates (x, y) of the sensor position is given with respect to the robot base. The desired rotation is represented internally in the node as a quaternion:

$$q(\theta) = \left[\cos\left(\frac{\theta}{2}\right) \quad 0 \quad 0 \quad \sin\left(\frac{\theta}{2}\right) \right]^T \in \mathbb{H}^* \quad (6.4)$$

The desired sensor orientation can then be computed by rotating the predefined (zero degrees) orientation with this quaternion. This orientation and its corresponding position can be represented as a homogeneous transformation matrix $T_b^s \in SE(3)$, from the robot's base frame $\{b\}$ to the desired sensor frame $\{s\}$. The desired tool pose of the robot is then given by:

$$T_t^b = T_s^b T_t^s, \quad T_t^s = \hat{X} \quad (6.5)$$

Where $\{t\}$ is the robot's tool frame, and \hat{X} is the estimated solution from the hand-eye calibration.

A tool pose is computed for each of the generated lattice points. The computed poses are then used to plan the necessary robot trajectories for sampling the points with MoveIt. The sensor pose is sampled between each trajectory, and the program waits a predefined time before sampling, in order for the robot- and tracking dynamics to settle down. Each sample is compared with an ideal sensor pose, which is computed from the forward kinematics of the robot and the estimated solution \hat{X} from solving the hand-eye problem:

$$\tilde{T}_{rc}^s = T_{rc}^{vr} T_{vr}^s - (T_t^{rc})^{-1} \hat{X} \quad (6.6)$$

The wait time can be as large as 500 seconds for the tracking dynamics to settle down, and running the sampling procedure with 84 lattice points takes 12 hours to finish. The robot is not moving during most of the procedure, as it is mostly waiting to sample a point.

Algorithm 2: A simple lattice point generator

Input: $x_0, y_0, z_0, L, W \in \mathbb{R}, m, n \in \mathbb{N}, f_{\text{rev}} \in \{0, 1\}$

Output: $p \in \mathbb{R}^{3 \times (m+1)(n+1)}$

```

// Initialize constants:
L2 = L/2, W2 = W/2;
Ln = L/m, Wn = W/n;
// Reverse order if the reverse flag is 1
rev = 1 - 2 frev;
// Set positive y-direction
dir = 1;

// Generate lattice points
for i = 0 to n do
  for j = 0 to m do
    p [1, 2, 3] [(m + 1) i + j] =
      
$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + \text{rev} \begin{bmatrix} (L_m i - L_2) \\ (W_n j - W_2) \text{ dir} \\ 0 \end{bmatrix}$$

  end
  // Reverse y-direction
  dir = (-1) dir;
end
return p;

```

6.2.2 Running the sampling procedure

A tracking volume of $L \times W \times H = 1.0 \text{ m} \times 3.0 \text{ m} \times 1.0 \text{ m} = 3.0 \text{ m}^3$ in the center of the robot cell was split into $(m + 1)(n + 1)(o + 1) = (3 + 1)(6 + 1)(2 + 1) = 84$ distributed points for sampling. The same points were sampled eight times, but the measurements from the tracking system were offset during the second and seventh run. This offset caused a systematic error with multiple modes, as shown in figure 6.3.

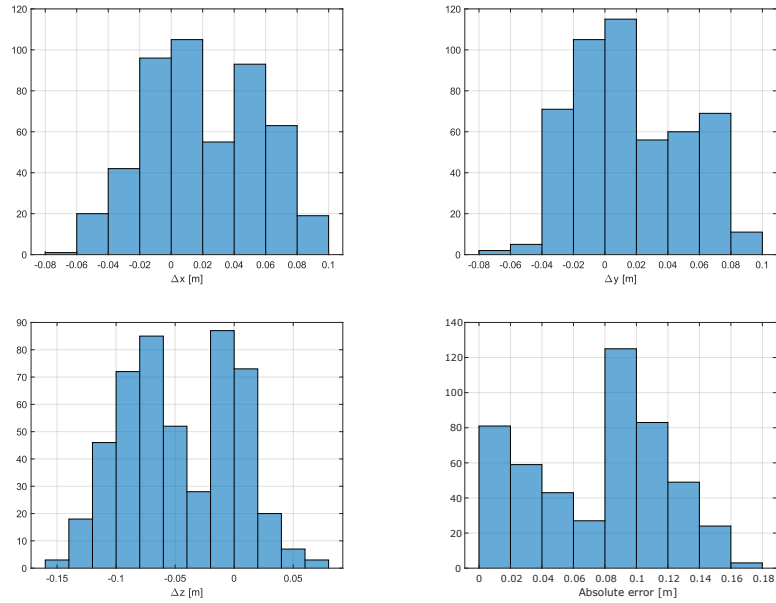


Figure 6.3: Frequency distribution of the measurement error along the x-, y-, z-axis and their absolute values with the robot's forward kinematics (FK) as a ground truth. The y-axis shows the number of samples in each bin.

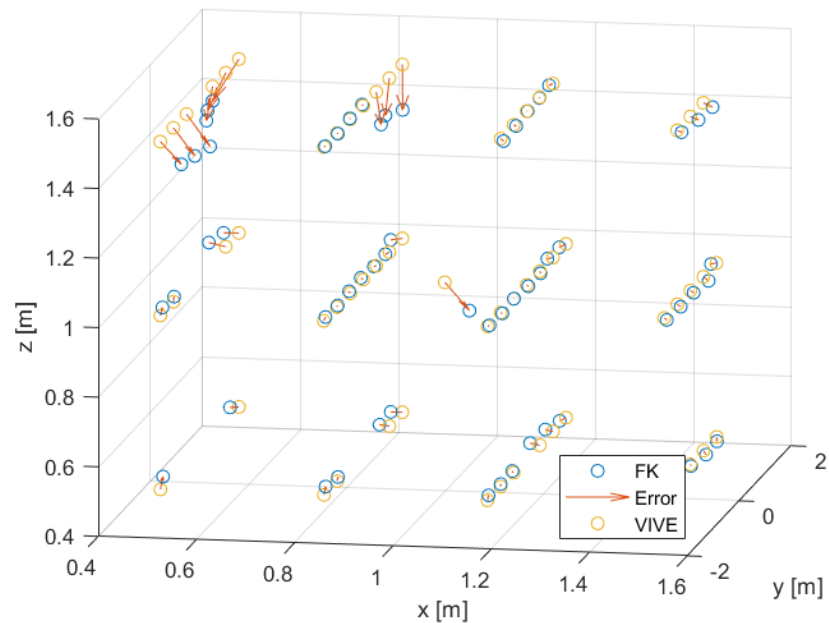


Figure 6.4: Comparison of the sampled sensor positions from the second run, which was sampled with the Vive tracker and the robot's forward kinematics (FK) as a ground truth. The positions are expressed relative to the robot's base frame, and the error is visualized as red arrows.

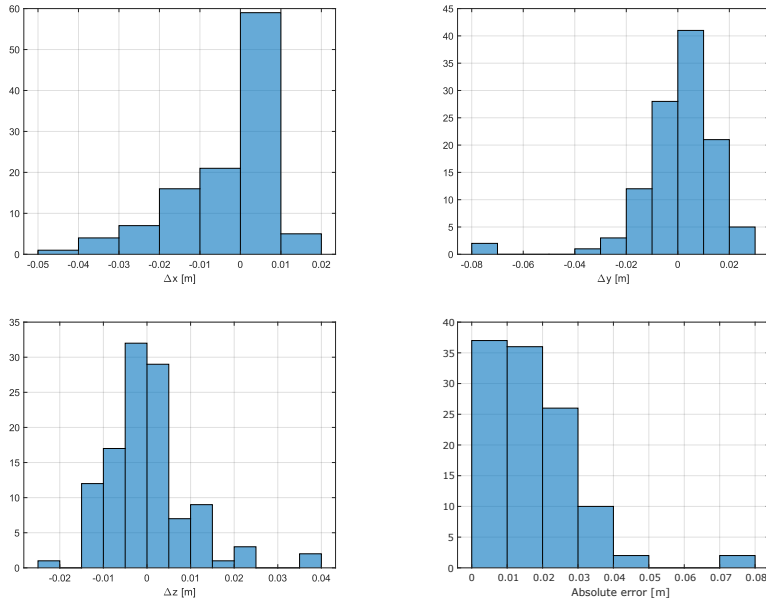


Figure 6.5: Frequency distribution of the measurement error from the first two runs along the x -, y -, z -axis and their absolute values with the robot's forward kinematics (FK) as a ground truth. The y -axis shows the number of samples in each bin.

It was easy to identify if an offset had occurred, as a large change in the error was observed between two consecutive points. The offset that occurred during the second run is recognized as the red arrows with a relatively large magnitude in figure 6.4. Sampled points with an offset error were considered outliers and excluded from the samples, leaving almost every point from the first two runs. Excluding these points resulted in a more sensible frequency distribution of the measurement error, as shown in figure 6.5.

The resulting means along the axes were found to be $[-0.003363, 0.0002183, 0.0003022]$ meters with standard deviation $[0.0119, 0.01468, 0.009321]$ meters. The cause of the negative bias along the x -axis was not well-understood, but it is small and the tracking dynamics suggests that it is caused by a larger drift along the x -axis. These results indicate a random measurement error in the centimetric range, and the maximum absolute error was 8 cm.

The resulting mean absolute error of the calibrated system was 17 mm with 13 mm standard deviation. This error is equal to the average positioning error that was reported by Niehorster et al. [28], but the standard deviation is 4 mm higher than their findings. The similar results indicates that the calibrated system

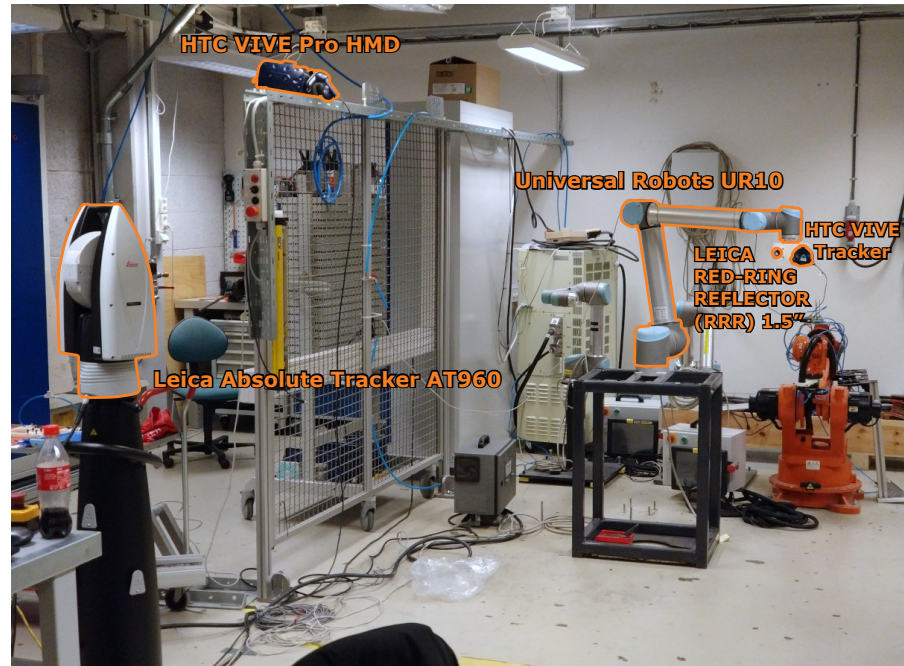


Figure 6.6: Experimental setup for sampling sensor positions with a Leica and Vive Tracker. The y-axis shows the number of samples in each bin.

performs comparably to the Vive. However, the measurement error was found to be nonlinear within the tracking volume.

6.3 MAPPING THE ERROR WITH A LASER TRACKER

A similar sampling procedure was run with a Leica Absolute Tracker AT960, as a sub-millimetric ground truth for the position of a tracked device. This sampling was coordinated with SINTEF Manufacturing, and was performed at their laboratory with the experimental setup that is shown in figure 6.6 and 6.9. The setup equipped a Universal Robots UR10 robot with a 1.5 inch red ring reflector and a Vive tracker.

A tracking volume of $L \times W \times H = 1.8 \text{ m} \times 0.9 \text{ m} \times 0.2 \text{ m} = 0.324 \text{ m}^3$ in front of the robot was split into $(m + 1)(n + 1)(o + 1) = (6 + 1)(3 + 1)(2 + 1) = 84$ distributed points, which were sampled with both the Leica and the Vive. The tool orientation was constrained towards the Leica tracker, such that the reflector was visible at all the sampling poses. However, MoveIt did not always satisfy the constraint during execution of a trajectory.

The sampled positions have to be expressed in a shared reference frame in order to compare them. A transformation from the robot's base frame to an arbitrary reference frame of the Leica

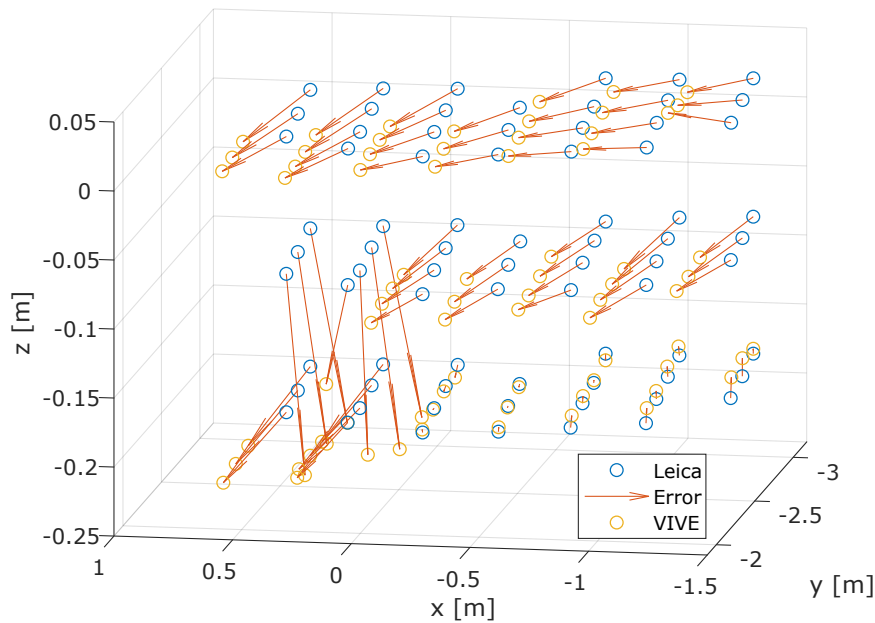


Figure 6.7: Comparison of the sensor positions that was sampled with the Leica reflector and Vive tracker. The positions are expressed relative to an arbitrary reference frame of the Leica tracker, and the errors are visualized as red arrows.

tracker is required. Horn’s quaternion-based method [21] was used to find the optimal transformation between these frames. The method is described in section 3.3, and takes two sets of correspondent 3D-points in their respective frames as an input.

Ideal Vive tracker positions from the forward kinematics of the robot were used to find the required transformation. The mean absolute error of the transformed positions relative to the Leica tracker was 1.6 mm with a maximum error of 3.4 mm. This transformation was then used to transform the sampled Vive tracker positions to the Leica’s reference frame, resulting in the comparison that is shown in figure 6.7.

A large error was observed during the sampling, which seemingly switched its value at random whenever the robot was moving. This type of error was described by Niehorster et al. [28], as a large change in offset whenever the tracking was briefly lost. This brief loss of tracking can be recognized as the spikes in figure 6.7.

The resulting measurement error is an order of magnitude worse than the previous results, and shows the importance of avoiding this error. Although the large changes in offset makes it hard to assess the accuracy of the calibrated system, it still

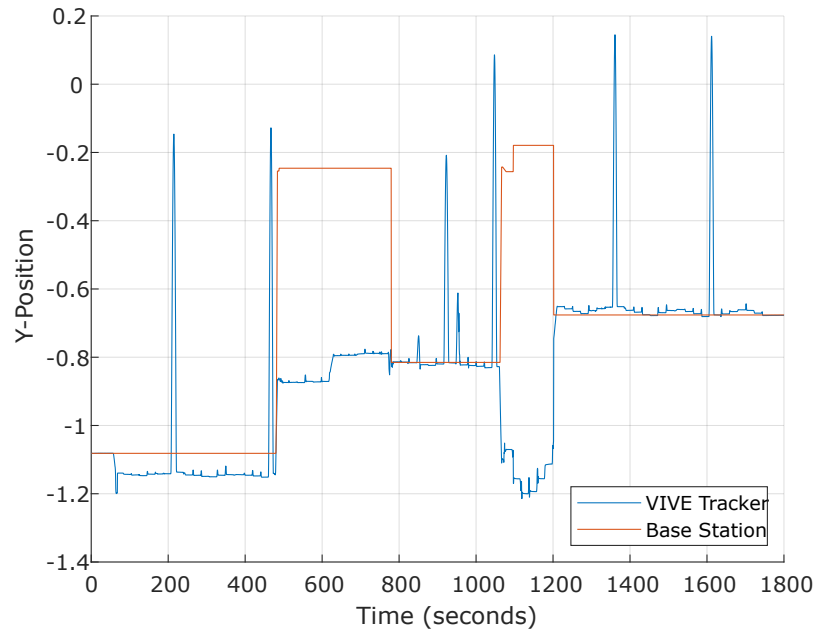


Figure 6.8: Y-position of the Vive tracker and one of the base stations during the sampling procedure. The position of the base station has been scaled and offset to make it easier to discern an instantaneous change in the offset of both devices.

provides insights into the inner workings of the Vive and its tracking system.

6.3.1 *Reasons for the large changes in offset*

The large changes in offset that happened during the sampling were not immediately well-understood. It was noted that the changes occurred instantaneously for all of the tracked devices between two consecutive samples, including one of the base stations. This behaviour can be seen in figure 6.8, where the y-positions of the Vive tracker and one of the base stations are shown.

The poses of the base stations were initially thought to be constant at all times after the internal Vive calibration, but this is clearly not the case. According to the inventor of lighthouse tracking, Alan Yates (Reddit user vk2zay in the provided reference); the error occurs whenever the base stations disagree with each other by a large amount [32]. The error is caused by a recalibration of the base stations, in order to reduce the discrepancy between them. This recalibration shows up as a bootstrapping of one of the base stations in the web console:

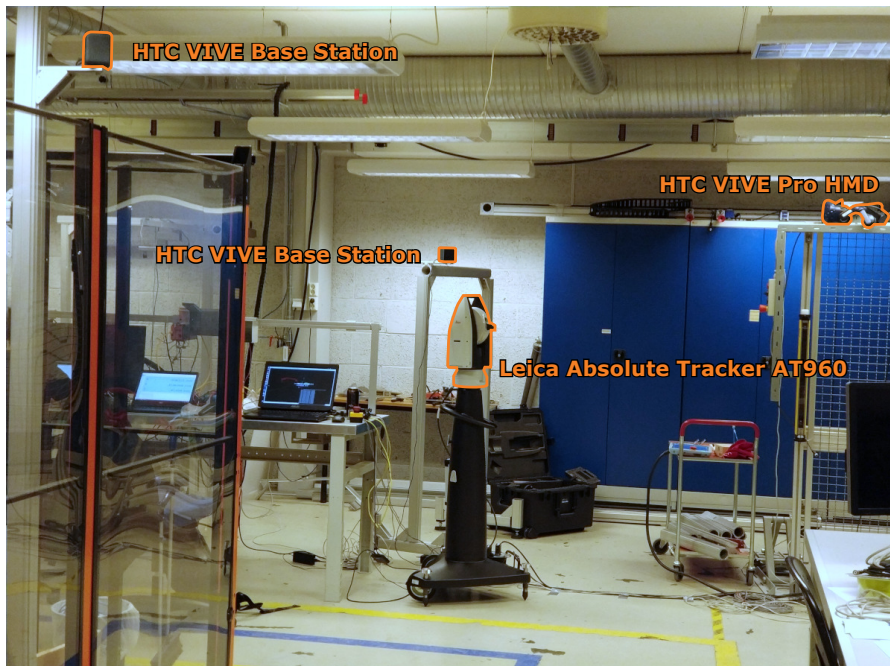


Figure 6.9: Experimental setup for sampling sensor positions with a Vive- and Leica tracker, as seen from the side.

```

lighthouse: LHR-F7EF5940 C: ----- BOOTSTRAPPED base 591F485E (
    immediate) distance 2.20m velocity 0.01m/s recorded pitch
    ~-38.9 deg roll ~-4.7 deg -----
lighthouse: LHR-F7EF5940 C: ----- CALIBRATED base 591F485E at
    pitch 39.29 deg roll -4.58 deg -----
lighthouse: LHR-F7EF5940 C: ----- SECONDARY base A9C7E2D4
    distance 2.51m -----
lighthouse: LHR-F7EF5940 C: ----- RELATIONSHIP bases 591F485E <->
    a9c7e2d4 distance 3.37m, angle 178.29 deg -----

```

The large changes in offset were likely caused by the grating in figure 6.6, which partially concealed the light sensors on the HMD and Vive tracker for at least one of the base stations. This cause fits with the fact that a recalibration almost never occurred with the Thrivaldi setup, where the HMD was kept visible to both base stations at all times.

From the layout of the experimental setup, it seems probable that the location of the base station behind the Leica tracker in figure 6.9 was the problem. The HMD should have been moved to the front of the grating, and the base station should have been moved to the other side of the Leica tracker.

Internal recalibration of the base stations invalidates the configuration of the Vive Bridge package, which has to be recalibrated after such an event. A monitor was added to the Vive Bridge package to continuously check if the position of a base

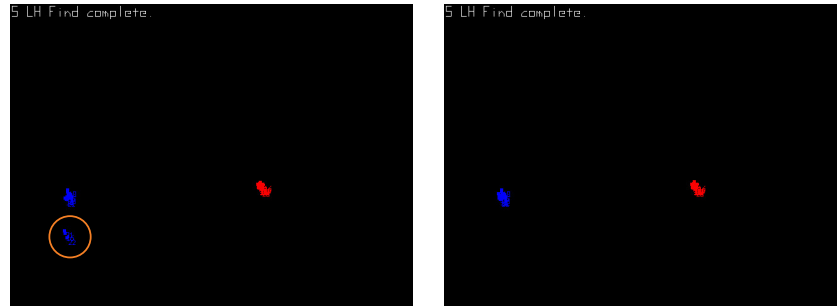


Figure 6.10: The LibSurvive calibration tool before (left) and after (right) the reflections in the orange ring was removed with a black piece of fabric, where the points should be clustered together.

station has changed, and warn the user if a recalibration has occurred:

```
[LIGHTHOUSE MONITOR] Pose of lighthouse_LHR-F7EF5940 has changed.
Tracked devices have likely also changed their pose.
```

If the tracked device that was used to calibrate the Vive Bridge package to a robot cell, still is fixed to the robot; the transformation from the previous hand-eye calibration is valid. It is then possible to quickly recalibrate the Vive Bridge package without performing a new hand-eye calibration, by passing a true/false argument to the calibration package.

6.4 LIBSURVIVE

The LibSurvive library was tested as an alternative to Valve's SteamVR and OpenVR SDK. It was implemented by wrapping its C API in an object with helper functions. These functions return or write data with the same structures that the OpenVR SDK uses. It was then possible to reuse the node portion of the Vive Bridge package, in order to expose the same functionality as with SteamVR to the ROS environment.

The LibSurvive API initially lacked some of the existing functionality in the Vive Bridge package. This missing functionality was requested in a pull request on the software repository for LibSurvive [33], and the functionality was added by the developers after some dialogue. It was then possible to implement the library in the Vive Bridge package with minor workarounds.

An error in the way that the button inputs were handled was found during the implementation of LibSurvive. This error

was reported in its own issue on the software repository for LibSurvive [34], and was promptly fixed by the developers.

LibSurvive was initially unable to internally calibrate the Vive setup. After a lengthy discussion with the developers and some trial and error, it was noted that the calibration worked with any other tracked device than the HMD. The Vive Pro HMD was untested with LibSurvive up until this point, and the developers found that the IMU scaling was wrong. This scaling was fixed by the developers, and the internal calibration of LibSurvive worked but not flawlessly.

Although the internal calibration of LibSurvive worked, it would randomly fail from time to time and the tracking was affected by jitter. The fencing system that encloses the robot cell was suspected to be the cause, as it consists of clear polycarbonate that could cause reflections. This suspicion was confirmed with the calibration tool for LibSurvive, which is able to visualize the reflections in a 2D map.

Figure 6.10 shows the situation before and after the black piece of fabric in figure 5.1 was added to the robot cell. This change fixed the tracking jitter with LibSurvive, and the SteamVR tracking was also more robust in general. The resulting tracking dynamics of LibSurvive was a lot faster than SteamVR, and converged within tens of milliseconds for fast movements between two points.

The LibSurvive tracking uses an arbitrary reference frame at the tracked device that was used for internal calibration, and it has to be calibrated separately from the SteamVR tracking. The calibration was performed in the same way as shown in section 5.2.3, and the hand-eye calibration returned a solution that was approximately equal to solutions from SteamVR with the same setup.

The spatially dependant bias of the calibrated system was about an order of magnitude worse than that of the SteamVR tracking. Although the tracking dynamics was a lot faster, the larger error made LibSurvive a nonviable alternative to SteamVR for the purposes of robot cell calibration.

Listing 6.1: Defining a functor for the Ceres solver

```

struct HandEyeCostFunctor {
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    // Constructor
    HandEyeCostFunctor(Sophus::SE3d A, Sophus::SE3d B) :
        A(A), B(B) {}

    // Overload ()-operator
    template <class T>
    bool operator()(T const* const sX, T* sResiduals) const
    {
        // Map input arrays to Sophus and Eigen objects
        Eigen::Map<Sophus::SE3<T> const> const X(sX);
        Eigen::Map<Eigen::Matrix<T, 6, 1>> Residuals(sResiduals);

        // Compute residuals
        Residuals =
            ( (A.cast<T>() * X).inverse() *
              (X * B.cast<T>()) ).log();

        return true;
    }

    Sophus::SE3d A, B;
};

```

Listing 6.2: Defining a Ceres problem from the functor

```

// Instantiate a Ceres problem
ceres::Problem problem;

for (int i = 0; i < N; i++) {
    // Define cost function from hand-eye functor with pair (A_i,
    // B_i) as input
    ceres::CostFunction* cost_function =
        new ceres::AutoDiffCostFunction<HandEyeCostFunctor,
            Sophus::SE3d::DoF,
            Sophus::SE3d::
                num_parameters>
        ( new HandEyeCostFunctor(A[i].cast<double>(),
            B[i].cast<double>()) );

    // Add residual block for each pair (NULL: no loss function)
    problem.AddResidualBlock(cost_function, NULL, X.data() );
}

```

RAPID ROBOT CELL CALIBRATION

A framework was created in order to define the virtual representation of objects or features of objects in the robot cell. This framework allows the user to register points on the surface of these objects with a tracked device, and interactively define the objects with geometric primitives. The framework was implemented in its own ROS node, which gets the registered points from the transform tree in the coordinates of the robot cell.

7.1 DEFINING GEOMETRIC PRIMITIVES FROM POINTS

The framework consists of multiple methods that are used to define geometric primitives such as boxes, spheres, cylinders and cones. These methods are general in the sense that they do not depend on the type of device that is used to register the surface points, and can be defined by using the Vive controller, tracker or even the HMD.

The choice of which primitives to implement was partially inspired by Miller et al. [35], whose paper presented a method of automatic grasp planning for robotic hands from primitives. Their grasping simulator, GraspIt, is available as a ROS package and presents the best grasps to the user based on provided primitives. This simulator was envisioned as a part of the framework to test a grasping scenario for an assembly use-case. However, it was not implemented due to time constraints and will have to be assessed in future work.

7.1.1 *Plane of finite size*

A plane of finite size can be uniquely defined in 3D space from three points $x_0, x_1, x_2 \in \mathbb{R}^3$. These points are used to define the orientation of the plane from basis vectors:

$$\mathbf{b}_x = \frac{x_1 - x_0}{\|x_1 - x_0\|}, \quad \mathbf{b}_y = \frac{x_2 - x_0}{\|x_2 - x_0\|}, \quad \mathbf{b}_z = \mathbf{b}_x \times \mathbf{b}_y \quad (7.1)$$

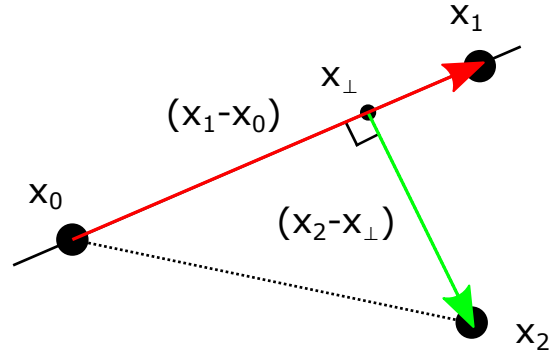


Figure 7.1: Defining a unique plane from three 3D points.

Where x_{\perp} is the point on the line from x_0 to x_1 that is closest to x_2 , as shown in figure 7.1:

$$\mathbf{x}_{\perp} = \mathbf{x}_0 - \left[(\mathbf{x}_0 - \mathbf{x}_2)^T \mathbf{b}_x \right] \mathbf{b}_x \quad (7.2)$$

This point is defined such that the basis vectors are orthogonal, which can be used to form the rotation matrix:

$$\mathbf{R} = \begin{bmatrix} \mathbf{b}_x & \mathbf{b}_y & \mathbf{b}_z \end{bmatrix} \in SO(3). \quad (7.3)$$

The translation to the center of the plane is then given by:

$$\mathbf{t} = \mathbf{x}_{\perp} + 1/2 (L \mathbf{b}_x + W \mathbf{b}_y) \in \mathbb{R}^3 \quad (7.4)$$

Where L and W is the length and width of the plane respectively:

$$L = \|\mathbf{x}_1 - \mathbf{x}_0\|, \quad W = \|\mathbf{x}_1 - \mathbf{x}_{\perp}\| \quad (7.5)$$

7.1.1.1 Angular error

This method has the nice property that the angular error $\Delta\theta$ is reduced with distance d :

$$d = \frac{h + \Delta h}{\tan(\theta + \Delta\theta)} = \frac{1 - \tan(\theta) \tan(\Delta\theta)}{\tan(\theta) + \tan(\Delta\theta)} (h + \Delta h) \quad (7.6)$$

As shown in figure 7.2. Solving this expression for $\tan(\Delta\theta)$ gives:

$$\tan(\Delta\theta) = \frac{\Delta h}{d + (h + \Delta h) \tan \theta} \quad (7.7)$$

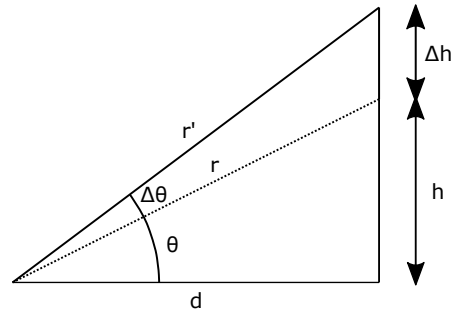


Figure 7.2: The effect of a height error Δh at a distance d on the angular error $\Delta\theta$.

Which for small-angle approximation results in:

$$\Delta\theta \approx \frac{\Delta h}{d}, \quad \theta, \Delta\theta \approx 0 \quad (7.8)$$

This approximation acts as an upper bound on the angular error $\Delta\theta$ that is caused by the height error Δh . It also implies that the angular error $\Delta\theta$ is (at least) inversely proportional to the distance d . Another implication of this property is that the longest side of the plane should be measured first, as an error in the x -axis also affects the y -axis.

7.1.1.2 Defining an arbitrary coordinate frame

This method of defining a plane could be used as an alternative to an automated alignment and correction method by Peer et al. [36]. They proposed the use of three Vive Trackers arranged on a frame, in order to align the virtual space with the physical ground and fix the tilt that was reported by Niehorster et al. [28]. Similarly, it is possible to define the ground plane with the method that is described here, and fix the issue with three points and one tracked device instead.

7.1.2 Box

A box can be uniquely defined in 3D space from four points $x_0, x_1, x_2, x_3 \in \mathbb{R}^3$. The first three points are used to define a plane of the box in the same way as shown in section 7.1.1. This plane gives the orientation of the box as a rotation matrix, and it is possible to compute the height H of the box by introducing a fourth point x_3 :

$$H = (x_3 - x_{\perp})^T \mathbf{b}_z. \quad (7.9)$$

The translation to the center of the box is then given by:

$$\mathbf{t} = \mathbf{x}_\perp + 1/2 (L \mathbf{b}_x + W \mathbf{b}_y + H \mathbf{b}_z) \in \mathbb{R}^3 \quad (7.10)$$

This method of defining a box, where the x- and y-axes are locked in place with the second and third points respectively, is intuitive to use. The framework visualizes a point, line, plane and box in that order for every point that is defined by the user.

7.1.3 Sphere

A sphere can be uniquely defined in 3D space from four non-coplanar points $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3 \in \mathbb{R}^3$ on its surface. The distance between these points and the center point $\mathbf{c} \in \mathbb{R}^3$ of the sphere should be equal to its radius r :

$$(\mathbf{x}_i - \mathbf{c})^\top (\mathbf{x}_i - \mathbf{c}) - r^2 = \mathbf{x}_i^\top \mathbf{x}_i + \begin{bmatrix} \mathbf{x}_i^\top & 1 \end{bmatrix} \begin{bmatrix} -2\mathbf{c} \\ \mathbf{c}^\top \mathbf{c} - r^2 \end{bmatrix} = 0 \quad (7.11)$$

A system of linear equations can be defined from this expression and four points:

$$\begin{bmatrix} \mathbf{x}_0^\top & 1 \\ \mathbf{x}_1^\top & 1 \\ \mathbf{x}_2^\top & 1 \\ \mathbf{x}_3^\top & 1 \end{bmatrix} \begin{bmatrix} -2\mathbf{c} \\ \mathbf{c}^\top \mathbf{c} - r^2 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0^\top \mathbf{x}_0 \\ \mathbf{x}_1^\top \mathbf{x}_1 \\ \mathbf{x}_2^\top \mathbf{x}_2 \\ \mathbf{x}_3^\top \mathbf{x}_3 \end{bmatrix} \quad (7.12)$$

The center point \mathbf{c} and radius r of the sphere is then found from the solution to this system. This method is not robust and it is very important the points are non-coplanar. Therefore, an extra optimization step with N points was added; where the following cost function was minimized with the solution to (7.12) as an initial guess for the Ceres solver:

$$\min_{\mathbf{c} \in \mathbb{R}^3, r \in \mathbb{R}} \sum_{i=1}^N (\mathbf{x}_i - \mathbf{c})^\top (\mathbf{x}_i - \mathbf{c}) - r^2 \quad (7.13)$$

It is then possible for the user to define as many points as needed to fit a sphere to an object.

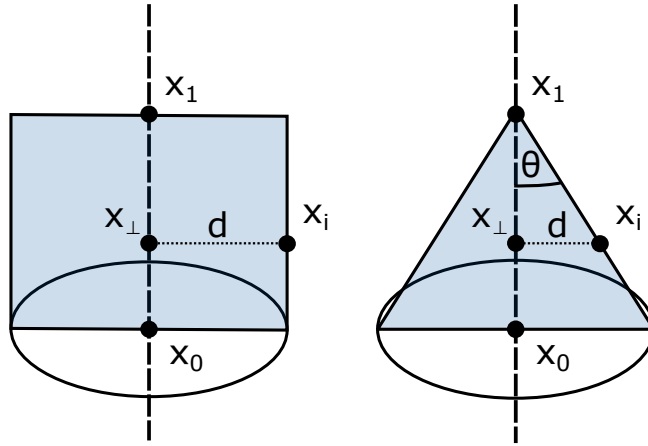


Figure 7.3: Defining a cylinder or cone from surface points x_i with perpendicular distance d from an axis that is defined by the points x_0 and x_1 .

7.1.4 Cylinder and Cone

The cylinder and cone cases are based on the perpendicular distance d between surface points x_i and an axis line that is defined from two points, as shown in figure 7.3. A beautiful formula for this distance was derived in section 3.4. For simplicity, it is assumed that the axis is approximated from the two first points. Each surface point x_i should have a perpendicular distance d that is equal to the radius of the cylinder. The cylinder case can then be given as the minimization problem:

$$\min_{x_0, x_1 \in \mathbb{R}^3, r \in \mathbb{R}} \sum_{i=2}^N \frac{\|(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_i - \mathbf{x}_1)\|^2}{\|\mathbf{x}_i - \mathbf{x}_1\|^2} - r^2 \quad (7.14)$$

Similarly, $(x_i - x_1)$ is the hypotenuse of a right triangle, which together with an angle θ relative to the axis can be used to compute the perpendicular distance d . The cone case can then be given as the minimization problem:

$$\min_{x_0, x_1 \in \mathbb{R}^3, \theta \in [0, \pi/2]} \sum_{i=2}^N \frac{\|(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_i - \mathbf{x}_1)\|^2}{\|\mathbf{x}_i - \mathbf{x}_1\|^2} - \|\mathbf{x}_i - \mathbf{x}_1\|^2 [\sin(\theta)]^2 \quad (7.15)$$

These minimization problems were solved with the Ceres solver, where at least 6 surface points are required. Radius $r_0 = 0.5$ m and angle $\theta_0 = \pi/4$ was used as an initial guess

for the solver, and the solution to both problems converges in general; given that the two first points are registered in such a way that they give the general direction of the axis. An example of such points would be to define the first and second point at the base and apex of a cone object respectively.

Knowing the axis of a cylinder or cone, it is possible to re-define the bottom point \mathbf{x}_0 with an extra point \mathbf{x}_{N+1} that is projected onto the axis:

$$\mathbf{x}_0 = \mathbf{x}_1 + \left[\frac{(\mathbf{x}_{N+1} - \mathbf{x}_1)^T (\mathbf{x}_0 - \mathbf{x}_1)}{\|\mathbf{x}_0 - \mathbf{x}_1\|} \right] \frac{(\mathbf{x}_0 - \mathbf{x}_1)}{\|\mathbf{x}_0 - \mathbf{x}_1\|} \quad (7.16)$$

This expression can also be used to redefine the top point \mathbf{x}_1 of the cylinder with an extra point \mathbf{x}_{N+2} . The translation to the center of the cylinder is then given by:

$$\mathbf{t}_{\text{cylinder}} = 1/2(\mathbf{x}_0 + \mathbf{x}_1) \quad (7.17)$$

And the translation to the cone is defined at its apex:

$$\mathbf{t}_{\text{cone}} = \mathbf{x}_1 \quad (7.18)$$

The orientation is defined similarly to the plane case in section 7.1.1 with an arbitrary rotation about the cylinder or cone axis, which is chosen as the x-axis:

$$\mathbf{x}_\perp = \mathbf{x}_1 + \left[(\mathbf{x}_2 - \mathbf{x}_1)^T \mathbf{b}_x \right] \mathbf{b}_x \quad (7.19a)$$

$$\mathbf{b}_x = \frac{\mathbf{x}_0 - \mathbf{x}_1}{\|\mathbf{x}_0 - \mathbf{x}_1\|}, \quad \mathbf{b}_y = \frac{\mathbf{x}_2 - \mathbf{x}_\perp}{\|\mathbf{x}_2 - \mathbf{x}_\perp\|}, \quad \mathbf{b}_z = \mathbf{b}_x \times \mathbf{b}_y \quad (7.19b)$$

$$\mathbf{R} = \begin{bmatrix} \mathbf{b}_x & \mathbf{b}_y & \mathbf{b}_z \end{bmatrix} \in \text{SO}(3) \quad (7.19c)$$

Where the first surface point \mathbf{x}_2 is used to compute the basis vector of the y-axis. The complete procedure to define a cylinder or cone can then be summarized as follows:

1. Define an approximation of the axis with the two first points from \mathbf{x}_0 to \mathbf{x}_1 , as shown in figure 7.3
2. Register at least 6 surface points or more in order to compute the necessary parameters
3. Define the bottom point \mathbf{x}_0 of the cylinder or cone
4. Define the top point \mathbf{x}_1 of the cylinder

7.2 REPRESENTING A VIRTUAL ROBOT CELL IN ROS

The objects are defined internally in the ROS node with their pose and parameters, as defined in section 7.1. These objects and their constituent parts such as: points, lines and planes for the box case, points and spheres for the sphere case, and points and lines (axes) for the cylinder or cone case, are visualized in real-time as they are being defined. The primitives are displayed to the user by publishing them to RViz through the RViz Visual Tools wrapper.

The virtual representation of the robot cell has to be saved to an appropriate file format, in order to make use of it for other applications. In order of importance, the format should be:

1. Standardized with its own specification
2. Suitable for describing objects and environments for robotics applications
3. Parseable with a C++ or Python interface
4. Preferably supported by MoveIt and RViz
5. Human-readable

Universal Robot Description Format (URDF) has historically been the format that is used to describe a robot in ROS. This format satisfies most of the requirements, but it is not suited for describing complex environments with multiple robots and sensor systems. It has several shortcomings that was documented by Chitta [37] in the URDF 2.0 specification.

URDF was designed to describe a single robot, not complex environments. For instance, the robots and gantry in Thrivaldi are described as a single entity in the URDF. Therefore, the Simulation Description Format (SDF) was chosen instead, as it checks all the requirements except for MoveIt and RViz support.

7.2.1 *Simulation Description Format (SDF)*

SDF is a human readable Extensible Markup Language (XML) format that describes objects and environments for robot simulators, visualization, and control. This format was created to solve the shortcomings of URDF, and provides a complete and scalable description for robots and their environment. It was originally developed as a part of the Gazebo robot simulator and also contains additional simulation-specific elements.

The format was implemented by wrapping its C++ API in an object with helper functions. These functions were defined to add the implemented primitives to an SDF file, as per the SDF specification [38]. Primitives such as a box can then be added to an SDF file with a single function call:

```
AddBox(-0.75243, -1.12828, 0.146555, // x, y, z - position
        3.09398, -0.003191, 0.028503, // R, P, Y - orientation
        1.19128, 0.12283, 0.016055, // L, W, H - dimensions
        "root") // reference frame
```

Which results in the following output to the SDF file:

```
<model name='box_o'>
  <static>1</static>
  <!-- x, y, z, roll, pitch, yaw relative to frame -->
  <pose frame='root'>-0.75243 -1.12828 0.146555 3.09398 -0.003191
    0.028503</pose>
  <link name='link'>
    <collision name='collision'>
      <geometry>
        <box>
          <!-- length, width, height -->
          <size>1.19128 0.12283 0.016055</size>
        </box>
      </geometry>
    </collision>
    <visual name='visual'>
      <geometry>
        <box>
          <!-- length, width, height -->
          <size>1.19128 0.12283 0.016055</size>
        </box>
      </geometry>
    </visual>
  </link>
</model>
```

7.3 CALIBRATION TOOL

The initial implementation of the ROS node used the position of a Vive controller, which was offset from its reference frame as shown in figure 7.4. It was then possible to press the flat face of the controller against a surface, and register a point in the center of the torus shaped controller head with the grip button.

The offset was measured from a CAD model of the controller, which could introduce a millimetric error due to manufacturing tolerances. A larger error is introduced by the user, as the center of the torus shaped controller head was used as a scope to

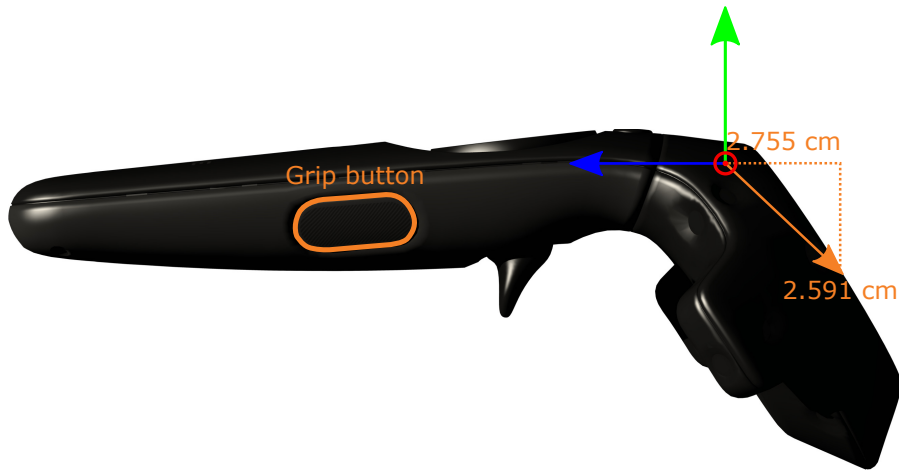


Figure 7.4: Reference frame of the Vive controller, which was offset to its flat top surface. The grip button was used to register the surface points when pressed.

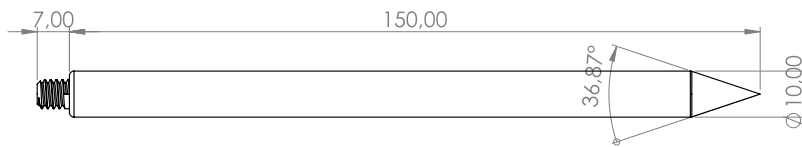


Figure 7.5: Drawing of the spike probe, where the dimensions are millimetric and the threads are standard $\frac{1}{4}$ " UNC.

crudely aim at and register the desired points. The bulky exterior of the controller also makes it hard to register points in small places. Additionally, the reference frame is located inside the controller, making it difficult to validate the offset with direct measurements. Therefore, a calibration tool was made from a Vive tracker with a spike probe attached to it, as shown in figure 7.5 and 7.6. The spike probe was fabricated from a nylon rod by the mechanical workshop at the institute.

The spike probe was not completely straight when screwed into the Vive tracker, so a calibration procedure is required. The reference frame of the Vive tracker is located at its camera mount, which makes it easy to calibrate and validate the probe tip.

A calibration procedure was implemented in its own ROS node, where the probe tip is held at a fixed point while the user rotates the Vive tracker around this point. The Vive tracker positions that are measured in this configuration should all be located on a sphere with center at the probe tip. It is then possible to use the method described in section 7.1.3 to find the parameters of this sphere. Knowing the center point c of the



Figure 7.6: Calibration tool based on a Vive tracker with a 15 cm long and 1 cm thick spike probe screwed into its ¼" UNC threaded camera mount.

sphere, the translation from the Vive tracker to the probe tip can be computed as an average from N measured points x_i on the sphere:

$$\mathbf{t} = \frac{1}{N} \sum_{i=1}^N \mathbf{R}^{-1} (\mathbf{c} - \mathbf{x}_i), \quad \mathbf{R} \in \text{SO}(3), \quad \mathbf{c}, \mathbf{x}_i \in \mathbb{R}^3 \quad (7.20)$$

Where \mathbf{R}^{-1} rotates the points into the Vive tracker frame, and the points are averaged in order to reduce the measurement error. A console command for sending the tool frame to the transform tree is then printed to the console:

```
roslaunch tf2_ros static_transform_publisher 0.00284715 0.00161493
0.150659 0 0 0 1 tracker_LHR_9C8FB0EC_tool0
tracker_LHR_9C8FB0EC
```

This command defines a transformation from the Vive tracker to its attached probe tip in the transform tree. The calibration tool can then be used to define objects in the same way as with the controller.

7.4 ASSEMBLY SCENARIO

The framework was tested on an assembly scenario provided by Mjøs metallvarefabrikk that is shown in figure 7.7, where a mock-up for inserting a rotor into a motor frame was mapped with a tracked device. The calibration tool was used to register the necessary points to define collisions in the assembly scenario, by pointing the probe tip at edges of the wood and pressing the grip button on a Vive controller.



Figure 7.7: Assembly scenario.

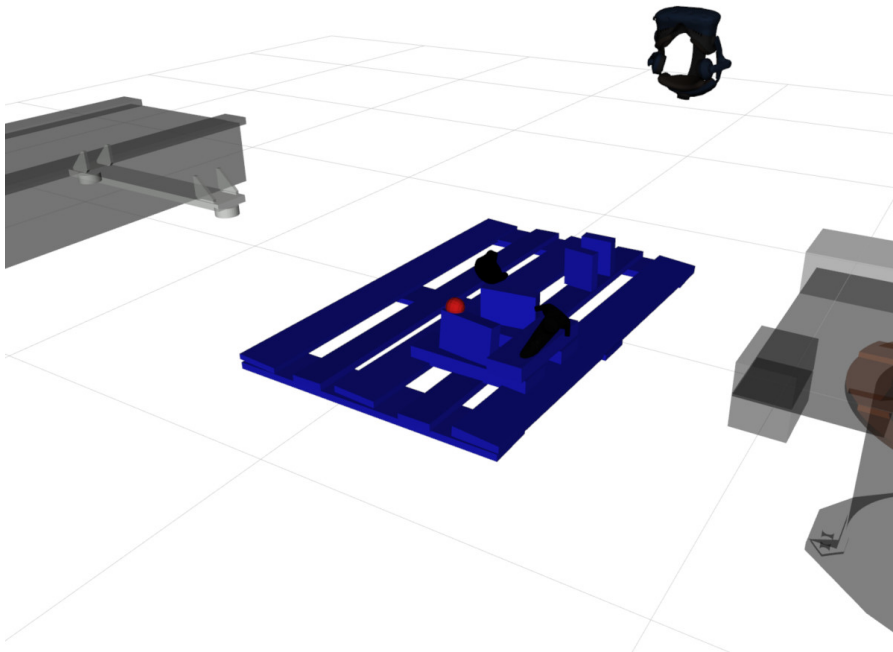


Figure 7.8: Virtual representation of the assembly scenario. The red sphere represents the tip of the calibration tool.

Figure 7.8 shows the virtual representation of the scenario, which was meticulously defined in about five minutes. The dimensions of the Euro-pallet in this figure were defined with centimetric accuracy and a millimetric deviation between similar parts. This accuracy is similar to that of the calibrated system, which limits the potential use-cases of the framework and depends on the required tolerances.

DISCUSSION, FUTURE WORK AND CONCLUSION

8.1 DISCUSSION

The calibration procedure presented in this thesis relies on a method of hand-eye calibration that does not depend on the choice of coordinates. The procedure was implemented in its own ROS node, which can be run with any ROS-Industrial supported robot through the hardware-agnostic MoveIt library. It generalizes well for different setups, and was successfully tested with the floor and gantry mounted KUKA KR16-2 robots, and a Universal Robots UR10 robot in another robot cell.

The calibrated system has a few problems that are related to specific issues of the lighthouse tracking. These issues have been prevalent throughout this project, and a disproportionate amount of time has been spent on trying to understand them. This effort has been important in order to perform reliable measurements with the Vive and understand its intricacies and limitations.

The tracking issues are mentioned in literature about the Vive and its tracking system, but the documentation on troubleshooting and correcting them is sparse. The findings of this project will hopefully rectify some of this sparsity by outlining the issues, their cause, and potential fixes in a concise manner. These findings were also summarized in the article of appendix B.

8.1.1 *Summarizing the tracking issues*

8.1.1.1 *Switching Bias*

Niehorster et al. [28] observed a large systematic error that switched its value whenever tracking was briefly lost. According to the inventor of lighthouse tracking, Alan Yates (Reddit user vk2zay in the provided reference); the error occurs whenever the base stations disagree with each other by a large amount [32]. The error is caused by a recalibration of the base stations, in order to reduce the discrepancy between them. This recalibration

shows up as a bootstrapping of one of the base stations in the web console of SteamVR.

The resulting error is nonlinear in Euclidean space, as the pose of the base stations is changed internally in the tracking system. It was noted that this change occurs instantaneously for all devices, and a monitor was added to the ROS node in order to warn the user if a recalibration has occurred.

The recalibration can be avoided, for the most part, by always keeping a tracked device in a location that is visible to both base stations without risk of concealment. The central and out of the way placement of the HMD in figure 5.1 was done for this purpose.

8.1.1.2 *Prioritizing inertial measurements*

Borges et al. [29] showed that the Vive's tracking algorithm gives greater weight to its inertial measurements, in order to generate smooth trajectories for VR applications. This weighting can clearly be seen in figure 4.4, where a tracked device was moved quickly between two points. The figure shows that the error is converging a lot faster when the tracked device is moving, which then slowly approaches its final value with an overdamped (second order) impulse response.

Convergence of the tracking dynamics can take as long as 500 s, and causes an error in the millimetric range when measuring the position of a tracked device before it has converged. The only known way of avoiding this error is the use of a third party tracking algorithm, which is exactly what Borges et al. introduced in their article [29].

An open-source back-end such as LibSurvive has to be used in place of SteamVR, in order to use a third party tracking algorithm. The tracking dynamics of LibSurvive was a lot faster than SteamVR, and converged within tens of milliseconds for fast movements between two points.

8.1.1.3 *Tracking Jitter*

The final and perhaps most common issue is tracking wobble and jitter, which is caused by reflections from the environment. Robot cells, for instance, are often enclosed by a fencing system with clear polycarbonate for safety reasons. This enclosure may cause reflections that have a negative impact on the robustness of the tracking.

The LibSurvive library is able to visualize reflections in a 2D map through its calibration tool, as shown in figure 6.10. This figure shows the situation before and after the black piece of fabric in figure 5.1 was added to the robot cell. Removing the reflections resulted in more robust tracking for SteamVR, and the LibSurvive tracking would not work properly without this change.

8.1.1.4 *Tilted reference frame*

Niehorster et al. [28] reported that poses measured with the Vive are provided in a reference frame that is tilted with respect to the physical ground plane. This issue is caused by the fact that the reference frame is aligned with the gravity vector, which is estimated with an IMU in the tracked device.

The tilted reference frame is a symptom of sensor bias in the IMU [39], and the solution is to either return the device or recalibrate the IMU [40]. Access to the calibration tools requires a SteamVR tracking license. The tilted floor is not an issue for the calibration procedure that is presented in this thesis, as it relies on an external calibration that does not depend on the choice of coordinates.

8.1.2 *Improving the calibration procedure*

Three different approaches were tested in order to improve the accuracy and precision of the calibration procedure. Of these, trying to improve the hand-eye calibration with a nonlinear optimization step had the least impact. The implemented hand-eye calibration method by Park and Martin [12] is robust with the number of samples that was used in this project (51), but the optimization step significantly reduced the error in (6.2) when using fewer than 11 samples.

A method of averaging quaternions by Markley et al. [17] was also tested, in order to remove a small but visible discrepancy in the mapping between the robot cell and tracking system. The discrepancy was caused by the fact that mapping (5.2) from the existing calibration procedure was based on only one location. This mapping was not robust, and the sub-millimetric and sub-degree noise in the pose of the tracked devices propagated to the mapping. The quaternion averaging reduced this noise by almost two orders of magnitude and removed the discrepancy.

Quaternion averaging was also tested to compute an average of mapping (5.2) from the hand-eye calibration measurements. This average resulted in a minor accuracy improvement in the millimetric range. The spatially dependent error was found to be a random error in section 6.2.1, and it is possible that averaging the mapping over the intended tracking volume gives the best possible solution for the current calibration procedure.

It is also possible that the Euclidean parameterization of mapping (5.2) is not optimal. The base stations are similar to camera systems in the sense that both of them can be represented with a projective model. However, there are no canonical camera models to reference when modeling the base stations [41].

Calibrating the tracking system on a lower level with a projection model could yield better results, but would likely require a considerable effort to implement. An open-source back-end such as LibSurvive would have to be used to access the required low-level components of the tracking system.

8.2 FUTURE WORK

The following points were recognized as potential directions for future work:

- Explore the use of second-generation (2.0) base stations. This revision has several improvements, which are mentioned in section 4.1.1, over the first-generation base stations that was used in this project. The Vive Bridge package should support the use of 2.0 base stations through SteamVR without any changes. The LibSurvive community is currently working on support for the 2.0 base stations.
- Implement a tracking algorithm that is more suited for robotics. The tracking dynamics of SteamVR is very slow (500 s), and the default LibSurvive tracking results in a large spatially dependant bias.
- Find a more optimal mapping from the robot cell to its virtual representation to reduce the spatially dependent error. Perhaps through the use of the projection model in [41]?
- Implement augmented reality by projecting RViz over a video feed from the front-facing HMD camera. An open-source plugin already exists to display the 3D view of RViz in the HMD [23].

- Implement support for other sensor systems. For instance, the Vive could be used for intuitive interaction with a vision system in the robot cell.
- Expand the feature set of the framework to support grasping scenarios. This would require that the underlying control algorithm exhibits sufficient robustness to positioning errors.

8.3 CONCLUSION

The work on this project culminated in a set of ROS nodes for: interfacing the Vive with ROS, calibrating its tracking system to a robot cell or an arbitrary reference frame, running an automated sampling procedure of uniformly distributed points with a robot, calibrating the position of a probe tip relative to a tracked device and positioning of objects or features of objects in the robot cell. All of the methods presented in this thesis generalizes well for different setups.

The calibration procedure for the Vive's tracking system exhibited a centimetric positioning error within the robot cell. This error is larger than the goal of sub-centimetric accuracy, and several approaches of little or no avail were tested while trying to reach this goal. However, a substantial amount of knowledge, which the writer would personally have loved to know from the get go, has been gathered on the Vive and its tracking system. Hopefully, this knowledge will prove to be useful for others working with the Vive in the future.

The robot cell calibration procedure consists of a framework with multiple methods, which are used to define geometric primitives such as boxes, spheres, cylinders and cones in the coordinates of the robot cell. This framework was tested on a mock-up for inserting a rotor into a motor frame, where the outline of the mock-up was defined with a spike probe that was attached to a Vive Tracker.

The dimensions of the mock-up were defined with centimetric accuracy and a millimetric deviation between similar parts. This accuracy is similar to that of the calibrated system, which limits the potential use-cases of the framework and depends on the required tolerances. The calibrated system could be used for crude positioning of objects, such as for collision avoidance or high-level planning.

This page is intentionally left blank.

APPENDIX



VIVE BRIDGE ROS PACKAGE README

This appendix contains the documentation for the Vive Bridge ROS package as it is given in its online repository. Vive Bridge is an open-source ROS package under the MIT License, which was created during a summer job at SINTEF Digital and further developed throughout this project. It is freely available from https://github.com/mortaas/vive_rrcc.

This page is intentionally left blank.

README.md

6/10/2019

Vive Bridge

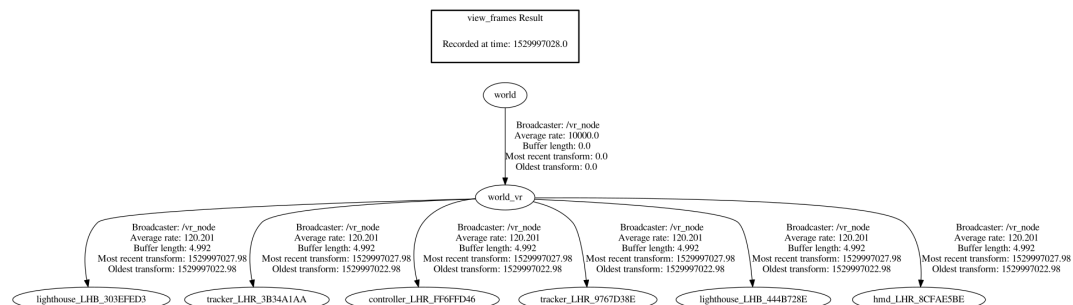
Vive Bridge is a [Robotic Operating System \(ROS\)](#) package that utilises the [OpenVR SDK](#) by Valve, or alternatively [LibSurvive](#), to make VR devices such as the HTC VIVE available in a ROS environment. The package is inspired by an existing [vive_ros](#) package by RoboSavvy, and it exposes a lot of the same functionality. The essentials of the [OpenVR SDK](#) is explained in great detail in the [OpenVR Quick Start](#) guide by Kevin Kellar. The guide is also saved locally in this package under the [doc/CassieVrControls.wiki](#) folder.

Features

The package supports the following types of devices:

- HMD (Head-Mounted Display)
- Controller
- Tracker
- Lighthouse

The package exposes the position and orientation (pose) of each device as coordinate frames relative to the *world_vr* frame in the tf tree, which is configurable relative to the *world* frame. The naming scheme of each coordinate frame follows the following structure: <device type>_<serial number>, e.g. *controller_LHR_FF6FFD46*. The serial number is used both internally and externally (in the package) to uniquely identify tracked devices. This results in a structure similar to the tf tree example that is shown below:



The package can also publish the linear and angular velocities (twists) of tracked devices as a *geometry_msgs/TwistStamped* message on the */vive_node/twist/<device type>_<serial number>* topic, e.g. */vive_node/twist/controller_LHR_FF6FFD46*. Axes and buttons on controllers can also be published as a *sensor_msgs/Joy* message on the */vive_node/joy/<device type>_<serial number>* topic, e.g. */vive_node/joy/controller_LHR_FF6FFD46*. Joy messages are only published when the controllers are interacted with. These publishers are not enabled by default, but they are easily enabled during runtime by using the [rqt_reconfigure](#) package.

Visualization

It is also possible to visualize the tracked devices by using a *MarkerArray* display in RViz. The mesh files are defined in the [launch/vive_node.launch](#) file as parameters for each type of device:

README.md

6/10/2019

- hmd_mesh_path
- controller_mesh_path
- tracker_mesh_path
- lighthouse_mesh_path

The mesh files has to be supported by RViz, i.e. *.stl*, *.mesh* (Ogre) or *.dae* (COLLADA).

The tracked devices are then visualized by adding `/vive_node/rviz_mesh_markers` as *Marker Topic* in a *MarkerArray* display.

The following warning message is normal when starting the node:

```
[ WARN] [1552396105.439729800]: Topic '/vive_node/rviz_mesh_markers' unable to connect to any subscribers within 0.5 sec. It is possible initially published visual messages will be lost.
```

It just indicates that no nodes received the mesh markers that are published when the vive node starts.

Requirements

OpenVR SDK

The package requires the [OpenVR SDK](#), which has to be built from the newest available source. It is possible to download and build the source in the correct folder by utilising the following commands:

```
cd ~
mkdir lib
cd lib
git clone https://github.com/ValveSoftware/openvr.git
cd openvr
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make
```

It is also possible to specify which folder the [OpenVR SDK](#) should be located in, by changing the *CMakeLists.txt* file in the package directory:

```
set(OPENVR "$ENV{HOME}/lib/openvr")
```

LibSurvive

The package requires [LibSurvive](#), which is an open-source and reverse engineered driver, API and tools for the HTC VIVE.

README.md

6/10/2019

```
cd ~
mkdir lib
cd lib
git clone https://github.com/cnlohr/libsurvive.git
cd libsurvive
make
```

It is also possible to specify which folder [LibSurvive](#) should be located in, by changing the `CMakeLists.txt` file in the package directory:

```
set(OPENVR "$ENV{HOME}/lib/libsurvive")
```

You also have to install the udev rules that comes with [LibSurvive](#):

```
cd lib/libsurvive
sudo cp useful_files/81-vive.rules to /etc/udev/rules.d/
sudo udevadm control --reload-rules && udevadm trigger
```

Steam and SteamVR

SteamVR is available through [Steam](#), which is utilised for configuration and room setup. It is also required for running this package by itself, as it depends on the `vrserver` process running in the background. This is a requirement because the OpenVR part of the package runs as a background application ([OpenVR API Documentation](#)):

VRApplication_Background - The application will not start SteamVR. If it is not already running the call with `VR_Init` will fail with `VRInitError_Init_NoServerForBackgroundApp`.

[Steam](#) is installed by following the *Getting Started* guide on their [Steam for Linux](#) tracker. SteamVR should be installed automatically by Steam if there is any VR devices present on your computer. It is also important to meet the **GRAPHICS DRIVER REQUIREMENTS** and the **USB DEVICE REQUIREMENTS** on their [SteamVR for Linux](#) tracker. A complete guide on getting the HTC VIVE up and running in SteamVR is available from: [HTC Vive Installation Guide](#).

Installation

The package is built by cloning this repository into your catkin workspace (`catkin_ws/src` directory) and then making it with `catkin_make`

Usage

The package is simply run by launching the following launch file: `roslaunch vive_bridge vive_node.launch`

README.md

6/10/2019

Alternatively, the package can be run with [LibSurvive](#) by launching the following launch file: `roslaunch vive_bridge survive_node.launch`. The package should expose the same functionality with [LibSurvive](#).

You may have to change the directory paths for Steam and your Catkin workspace in the `/scripts/launch.sh` shell script depending on their location. The package assumes that the directories are in their default locations.

```
STEAM_RUNTIME=$HOME/.steam/steam/ubuntu12_32/steam-runtime
CATKIN_WS=$HOME/catkin_ws
```

Applications are generally run through the Steam runtime by running the `run.sh` script. The script is in the `steam-runtime` folder, and takes the application as an argument for the script.

```
~/ .steam/steam/ubuntu12_32/steam-runtime/run.sh
~/catkin_ws/devel/lib/vive_bridge/vive_bridge_node
```

Interacting with the node

The `vive_node` publishes information about the currently tracked devices to the `/vive_node/tracked_devices` topic. This topic uses a custom `vive_bridge/TrackedDeviceStamped.msg` message that contains information about:

- `frame_id` - Fixed VR frame (within the message header)
- `uint8 device_count` - Number of tracked devices
- `uint8[] device_classes` - Classes of tracked devices (classes are defined within the message):

```
uint8 HMD=1
uint8 CONTROLLER=2
uint8 TRACKER=3
uint8 LIGHTHOUSE=4
```

- `string[] device_frames` - Child frames associated with each tracked device

The frame names within the `device_frames` field can then be used to find the joy and twist topics of each tracked device, e.g. the twist topic of the first tracked device could be:

```
"/vive_node/twist/" + msg_.device_frames[0]
```

It is also possible to request this information from the `/vive_node/tracked_devices` service, which requests a `std_msgs/Empty` message, and responds with the same format as the `vive_bridge/TrackedDeviceStamped.msg`. The `frame_id` is however included as its own field in the response, instead of being included in the message header.

Controller axes and buttons

The package currently supports all inputs from the HTC VIVE controller, and the `sensor_msgs/Joy` messages have the following format:

- `axes[0]` - Trackpad x

README.md

6/10/2019

- axes[1] - Trackpad y
- axes[2] - Trigger
- buttons[0] - Menu
- buttons[1] - Grip
- buttons[2] - Trackpad
- buttons[3] - Trigger

The package also emulates a numpad when pressing different points on the trackpad button:

- buttons[4] - 1 Left down
- buttons[5] - 2 Center down
- buttons[6] - 3 Right down
- buttons[7] - 4 Left center
- buttons[8] - 5 Center
- buttons[9] - 6 Right center
- buttons[10] - 7 Left up
- buttons[11] - 8 Center up
- buttons[12] - 9 Right up

The x and y values from touching the trackpad is used to find the corresponding numpad key. The header also contains the frame_id associated with the tracked device.

Haptic Feedback

The package supports activating the rumble feature of the HTC VIVE controller. Rumble is activated by sending a `sensor_msgs/JoyFeedback` message to the `/vive_node/joy/haptic_feedback` topic of type `TYPE_RUMBLE`. The id of the HTC VIVE controller corresponds with the order from the `/vive_node/tracked_devices` topic, and the intensity is the duration of the vibration.

Configuration

The configuration window for the `/vr_node` contains the following settings:

- `publish_joy`:
- `publish_twist`:
- `x_offset`: Slider from -5.0 to 5.0, value: 0.0
- `y_offset`: Slider from -5.0 to 5.0, value: 0.0
- `z_offset`: Slider from -5.0 to 5.0, value: 2.0
- `yaw_offset`: Slider from 14159265359 to 3.141592653, value: 0.0
- `pitch_offset`: Slider from 14159265359 to 3.141592653, value: 1.57079632679
- `roll_offset`: Slider from 14159265359 to 3.141592653, value: 0.0

The position and orientation (pose) of each device is defined relative to the `world_vr` frame, which has the same position as one of the lighthouses. This frame has to be defined relative to some defined `world` frame to

README.md

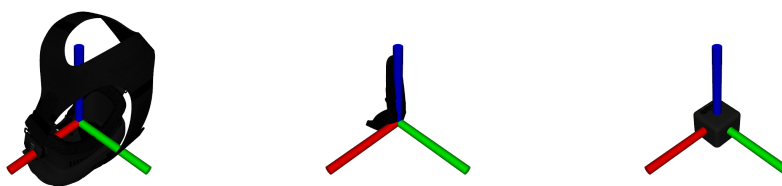
6/10/2019

make sense of the environment. The transformation between these frames are exposed as x-, y-, z- and roll-, pitch-, yaw- (RPY) offset parameters by the [dynamic_reconfigure](#) package. This package provides a standard way to change the offset parameters at any time without having to restart the node, and also provides a graphical user interface (GUI) to change these parameters by using [rqt_reconfigure](#):

```
roslaunch rqt_reconfigure rqt_reconfigure.
```

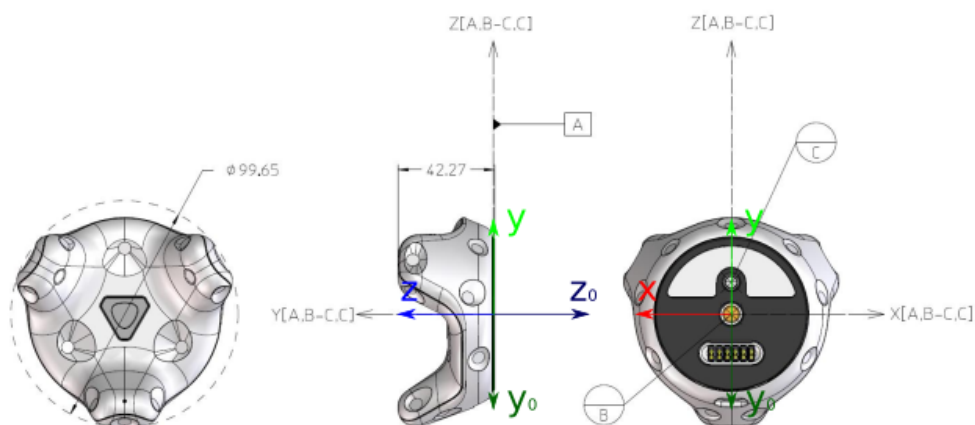
The parameters from [dynamic_reconfigure](#) are saved and loaded automatically from the `/cfg/dynparam.yaml` file.

Coordinate systems



Tracked devices follows the following coordinate system conventions:

- X-axis equates to pitch
- Y-axis is up and equates to yaw (except for the VIVE Tracker, which has Z-axis down)
- Z-axis is opposite of approach direction and equates to roll



The VIVE Tracker coordinate system is rotated 180° around the x-axis such that the z-axis points upwards. This is because we want the tracker to match the orientation of our reference frame (world), when it is placed horizontally on the ground.

Compatibility

The package was tested with:

README.md

6/10/2019

- HTC VIVE with OpenVR SDK 1.2.10 and Ubuntu 16.04 LTS running ROS Kinetic Kame (1.12.13)

To-do list

This page is intentionally left blank.

SUBMITTED CONFERENCE PAPER TO ICCMA
2019

This appendix contains the pre-review version of an article that was submitted to the 7th International Conference on Control, Mechatronics and Automation (ICCMA 2019). The article summarizes some of the results that is presented in this work.

This page is intentionally left blank.

Vive for Robotics: Rapid Robot Cell Calibration

Morten Andre Astad¹, Mathias Hauan Arbo¹, Esten Ingar Grøtli², and Jan Tommy Gravdahl¹

Abstract—The use of an HTC Vive; a virtual reality (VR) system and its innovative tracking technology is explored in order to create an approximate one-to-one mapping to the virtual representation of a robot cell. The mapping is found by performing hand-eye calibration, establishing a spatial relationship between the inertial frames of the robot cell and the tracking system. One of the main contributions of this article is the development of an open-source robotic operating system (ROS) package for VR devices such as the HTC Vive. The package includes automated calibration procedures such that the devices gives a centimetric measurement error in the robot cell. The calibrated system has problems that are related to specific issues of the tracking technology. This article outlines these issues, their cause, and potential fixes in a concise manner. A simple assembly scenario is presented, where the outline of objects in the robot cell are defined by registering points with the HTC Vive tracker. The potential use cases of the calibrated system are limited by its accuracy, and depends on the required tolerances.

I. INTRODUCTION

Industrial robots are often too inflexible for the current market demands of small- and medium-sized enterprises (SMEs). As part of the EU funded research project SMERobotics, [12] suggest that one of the main challenges preventing the adoption of industrial robots in SMEs is that current robot programming techniques are not suitable for frequent changes of often highly customized products manufactured in small batches.

This article explores the use of an HTC Vive, a virtual reality (VR) system codeveloped by Valve and HTC, for rapid robot cell calibration. By creating an approximate one-to-one mapping to the virtual representation of a robot cell, one can quickly place objects and obstacles as necessary. The innovative technology that allows for positioning in a room-scaled environment is called lighthouse tracking. This technology is able to track the user's hands, head, or other objects in real-time through tracked devices. The devices have sub-millimeter precision within an area whose diagonal is up to 5 meters in length.

The outside-in tracking system of the Vive sweeps the room horizontally and vertically with 850 nm infrared (IR) laser lines at a fixed frequency, from one or multiple stationary base stations in the room. Light sensors on the tracked devices are hit periodically by the laser lines, and their position and orientation (pose) is reconstructed by solving a problem that is similar to the Perspective-n-Point (PnP)

problem [5]. The tracked devices also contain an inertial measurement unit (IMU) that provides faster updates than the ones from the lighthouse tracking. This gives the devices low frequency measurements of absolute position and orientation, and faster updates of the relative motion of the devices.

In [9] the accuracy and viability of an HTC Vive are described for scientific research. They concluded that the Vive was unsuited for scientific experiments if loss of tracking was likely, as a large systematic error was observed that changes whenever the tracking was briefly lost. This error makes it difficult to establish a calibration procedure that aligns the real and virtual coordinate space. However, it is possible to avoid this error by taking measures against its cause, which will be presented together with a calibration procedure that does not depend on the choice of coordinates.

The main contributions of this article are: elaborating on the issues related to the HTC Vive, development of open-source software for calibration with respect to an industrial robot, development of open-source software for defining points, planes and boxes in a virtual environment, as well as presenting an assembly use-case example.

The article is split into 3 main parts. Section II describes how the Vive was integrated in a software environment for robots, and includes the theory and methods that was used to automatically calibrate the tracking system. Section III gives an overview on how the tracking system was set up, calibrated and evaluated in a robot cell, and also presents a simple assembly scenario that was defined with collidable objects using a tracking device. Section IV discusses specific issues of the lighthouse tracking, as they are prevalent, and getting the Vive to work properly without understanding these issues and how to fix them can be quite challenging.

The work is based on the master thesis of Astad [1], which we refer the reader to for more in-depth implementation details.

II. SYSTEM INTEGRATION AND THEORY

A. Hand-Eye Calibration

The standard hand-eye calibration problem was formulated in [15], where the problem was stated as an equation of homogeneous transformations:

$$\mathbf{A}\mathbf{X} = \mathbf{X}\mathbf{B}, \quad \mathbf{A}, \mathbf{B}, \mathbf{X} \in \text{SE}(3) \quad (1)$$

where \mathbf{A} depicts a change in the robot's tool pose, \mathbf{B} represents the resulting sensor displacement from changing the tool pose, and \mathbf{X} is an unknown transformation relating the tool frame to the sensor frame. The unknown transformation \mathbf{X} is constant under the assumption that the sensor is rigidly

¹Morten Andre Astad (mortaas@stud.ntnu.no), Mathias Hauan Arbo, and Jan Tommy Gravdahl are with the Department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU), Trondheim, Norway

²Esten Ingar Grøtli is with SINTEF Digital, Trondheim, Norway

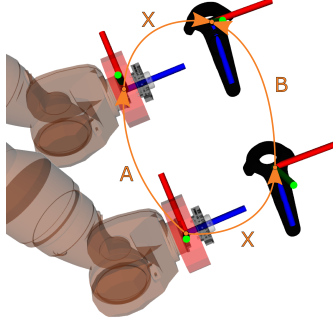


Fig. 1. Geometric interpretation of the $\mathbf{AX} = \mathbf{XB}$ problem, showing two different robot states.

attached to the robot and its tool frame. Fig. 1 shows a geometric interpretation of this problem.

The hand-eye calibration was solved by utilising the closed form solution in [10]. The input to this method is measured pairs of transformations $(\mathbf{A}_i, \mathbf{B}_i) \in \text{SE}(3)$, as defined by the deviation between consecutive samples of tool $\{t\}$ and sensor $\{s\}$ poses:

$$\mathbf{A}_i = \mathbf{T}_{t_i}^{-1} \mathbf{T}_{t_{i+1}}, \quad \mathbf{T}_{t_i}, \mathbf{T}_{t_{i+1}} \in \text{SE}(3) \quad (2a)$$

$$\mathbf{B}_i = \mathbf{T}_{s_i}^{-1} \mathbf{T}_{s_{i+1}}, \quad \mathbf{T}_{s_i}, \mathbf{T}_{s_{i+1}} \in \text{SE}(3) \quad (2b)$$

and a solution requires at least three of these pairs. The method solves the rotational part first before using it to solve the translational part, which propagates an error from rotation to translation. An extra optimization step was added to reduce this error, where the following cost function was minimized with the closed form solution to the hand-eye calibration as a seed for the solver:

$$\min_{\mathbf{X} \in \text{SE}(3)} \sum_{i=1}^N \log((\mathbf{A}_i \mathbf{X})^{-1} \mathbf{X} \mathbf{B}_i) \quad (3)$$

Here, N is the number of measured pairs $(\mathbf{A}_i, \mathbf{B}_i)$, $(\cdot)^{-1}$ is the $\text{SE}(3)$ group inverse and $\log(\cdot)$ is the matrix logarithm, which maps elements in the group of rigid transformations $\text{SE}(3)$ into elements of its tangent space $se(3)$. This step significantly reduced the error when using fewer than 10 measured pairs.

B. Generating Sample Poses

For automatic calibration, tool poses are generated for sampling the necessary poses with a robot. Their position is selected at random within a spherical volume element, and their orientation is picked from the normal vector at this position as from the surface of a sphere. The positions can take on any value within the spherical volume element that is parameterized as shown in Fig. 2. Each tool pose can then be represented as a homogeneous transformation from the robot's base, $\{b\}$, to tool, $\{t\}$, frame:

$$\mathbf{T}_b^t = \begin{bmatrix} \mathbf{R}_{z,\theta} \mathbf{R}_{x,\phi} & \mathbf{R}_{z,\theta} \mathbf{R}_{x,\phi} \begin{bmatrix} 0 & 0 & r \end{bmatrix}^T \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix} \quad (4)$$

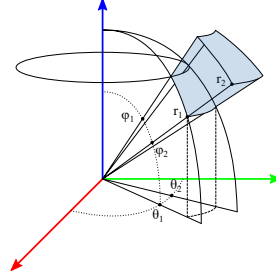


Fig. 2. Spherical volume element that is defined by the spherical coordinates $(r \in [r_1, r_2], \theta \in [\theta_1, \theta_2], \phi \in [\phi_1, \phi_2])$ in a right-handed coordinate system.

where $\mathbf{R}_{x,\phi}$ and $\mathbf{R}_{z,\theta}$ are basic rotations about the x -axis and z -axis by an angle ϕ and θ respectively, and r is the radius of a sphere. [17] observed that the rotation between consecutive sensor poses, and the translation between consecutive tool poses should be maximized and minimized respectively; in order to improve the accuracy of the hand-eye calibration. Therefore, the translations and rotations are generated from two sets of parameters in smaller and larger ranges about the same point. This method of randomly generating tool poses for sampling within a range is flexible and generalizes well for different setups. It was tested with the floor and gantry mounted KUKA KR16 robots presented in this article, and a Universal Robots UR10 robot that was mounted on a freestanding frame.

C. Robot Operating System Package

The Robotic Operating System (ROS) is an open-source middleware solution for robotics. At its core it offers a communication system, which provides a message passing interface between distributed nodes. One of the main contributions of the work that is presented in this article was the development of a ROS package named `vive_rrcc`. This package makes VR devices such as the Vive available in a ROS environment by utilising SteamVR through the OpenVR SDK by Valve.

The package exposes the pose of each tracked device as a coordinate frame with respect to an inertial tracking frame. These frames and their relationships are maintained in a distributed tree structure that is buffered in time with the `tf2` transform library for ROS [6]. This library allows the user to transform vectors, quaternions, poses and so forth between any two frames in the tree structure. It also acts as a buffer for the poses of the tracked devices, which are available in any frame of the transform tree, and to all nodes in the ROS environment.

Other features of the package includes controller inputs, haptic feedback, linear and angular velocities (twists) and 3D visualization of the tracked devices, and a standard interface to interact with and calibrate the node in realtime. The `LibSurvive` library [3] was also implemented as an alternative to SteamVR and OpenVR. Unlike OpenVR, this library allows for access to the low-level components of

the lighthouse tracking and supports the use of different community implemented tracking algorithms.

The generated sample poses can be automatically realized on the robot system. Robot trajectories are planned from the generated tool poses with the MoveIt library [4], a motion planning framework that is integrated with ROS. Tool and sensor poses are then sampled from the transform tree between each executed trajectory, and the program waits a predefined time before sampling, in order for the robot and tracking dynamics to settle. This hardware-agnostic approach can be used on any ROS-Industrial supported robot with a MoveIt package.

The sampled poses of a tracked device are subject to noise in the sub-millimetric and sub-degree range, which may introduce a small error in the calibrated system. This noise was reduced by almost two orders of magnitude by using the quaternion averaging method in [8] and 120 samples.

The ROS package is open-source under the MIT License and is freely available from https://github.com/mortaas/vive_rrcc.

D. Calibrating the Tracking System

One-to-one mapping from a robot cell to its virtual representation is established by finding a spatial relationship between the inertial frames of the robot cell and tracking system. This relationship was found by employing hand-eye calibration, in order to estimate the rigid transformation $\hat{\mathbf{X}}$ between a tracking device that is firmly attached to the robot and an arbitrary tool frame of the robot.

A transformation between the inertial frames of robot cell $\{rc\}$ and tracking system $\{vr\}$ is computed for each of the measured sample poses, with the estimated solution $\hat{\mathbf{X}}$ from solving the hand-eye problem:

$$\mathbf{T}_{rc}^{vr} = (\mathbf{T}_t^{rc})^{-1} \hat{\mathbf{X}} \mathbf{T}_s^{vr}, \quad \hat{\mathbf{X}} = \hat{\mathbf{T}}_t^s \quad (5)$$

where \mathbf{T}_{rc}^{vr} is the transformation from the inertial frame of the robot cell to the inertial frame of the Vive system according to the calibration, \mathbf{T}_{vr}^s is the transformation to the tracking device frame relative to the Vive system, and $(\mathbf{T}_t^{rc})^{-1}$ is the tool frame relative to the robot cell.

These transformations are then averaged in the same way as the sampled poses, in order to reduce a small nonlinear and spatially dependent error of the tracking system. The resulting average is used to calibrate the system by automatically updating the corresponding relationship in the transform tree. Figure 5 shows the virtual representation of the robot cell after performing this calibration.

E. Rapid Obstacle Placement

A simple framework was created in order to define collidable objects with geometric primitives, such as boxes, spheres, cylinders, and cones, in the virtual robot cell using the tracking devices. This section presents the box case.

A box can be uniquely defined in 3D space from four points $x_0, x_1, x_2, x_3 \in \mathbb{R}^3$. The first three points are used

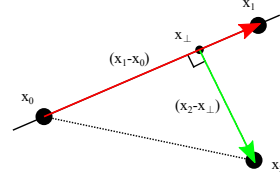


Fig. 3. Defining a unique plane from three 3D points.

to define the orientation of the box from basis vectors:

$$\mathbf{b}_x = \frac{x_1 - x_0}{\|x_1 - x_0\|}, \quad (6a)$$

$$\mathbf{b}_y = \frac{x_2 - x_\perp}{\|x_2 - x_\perp\|}, \quad (6b)$$

$$\mathbf{b}_z = \mathbf{b}_x \times \mathbf{b}_y \quad (6c)$$

where x_\perp is the point on the line from x_0 to x_1 that is closest to x_2 , as shown in Fig. 3:

$$x_\perp = x_0 - [(x_0 - x_2)^T \mathbf{b}_x] \mathbf{b}_x. \quad (7)$$

This can be used to form the rotation matrix:

$$\mathbf{R} = [\mathbf{b}_x \quad \mathbf{b}_y \quad \mathbf{b}_z] \in \text{SO}(3). \quad (8)$$

It is now possible to define the length L , width W and height H of the box by introducing the fourth point x_3 :

$$L = \|x_1 - x_0\|, \quad W = \|x_1 - x_\perp\|, \quad H = (x_3 - x_\perp)^T \mathbf{b}_z. \quad (9)$$

The translation to the center of the box is then given by:

$$\mathbf{t} = x_\perp + 1/2 (L \mathbf{b}_x + W \mathbf{b}_y + H \mathbf{b}_z) \in \mathbb{R}^3 \quad (10)$$

This method of defining a box is intuitive, and the framework visualizes a point, line, plane and box in that order for each point that is defined by the user. The recorded collidable objects are saved as Simulation Description Format (SDF) files, a human readable XML format that describes objects and environments for robot simulators, visualization, and control.

The planar part of this method could be used as an alternative to the automated alignment and correction method in [11]. Where three Vive Trackers affixed to a frame is utilised in order to align the virtual space with the physical ground, and fix the tilt that was reported in [9]. Similarly, it is possible to define the ground plane with the method that is described here, and fix the issue using only one tracked device and defining three points instead.

III. EXPERIMENTAL SETUP

A. Vive-Robot Cell Setup

The system was tested on the robot cell with an approximate size of $6m \times 4m \times 4m$, shown in Fig. 4. The robot cell consists of two KUKA KR 16 industrial robots; one of which is mounted on the floor, and the other is mounted on a gantry system from Güdel. Both base stations are mounted below H-beams in the roof structure, following the recommendations from HTC. The base stations have a field of view of 120° ,

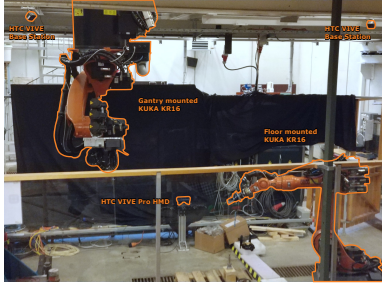


Fig. 4. Setup of the Vive's tracking system in a robot cell.

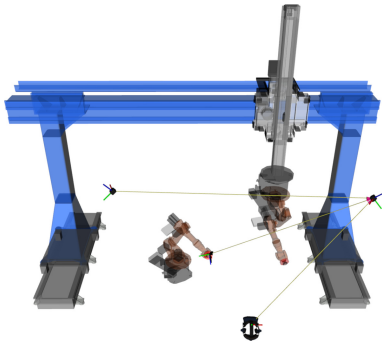


Fig. 5. The virtual representation of the robot cell after calibration, as visualized in RViz.

leaving 30-45° for adjustments. The base stations should be placed such that their view of each other and the robot cell is unobstructed. It is also important that their field of view overlaps as much as possible within the intended tracking volume. A Vive Pro Starter Kit with first-generation base stations was used, which provides updates at a rate of 220-370 Hz depending on the type of tracked device [7]. The measurements are sub-sampled in the ROS package at a rate of 120 Hz by default.

B. Calibration of the Mapping

A tracked device was firmly attached to the gripper of the floor robot, and 51 tool poses was generated in the range ($r \in [1.4, 1.6]$, $\theta \in [-5\pi/16, -3\pi/16]$, $\phi \in [\pi/8, 3\pi/16]$) for positions and range ($r \in [1.4, 1.6]$, $\theta \in [0, -\pi/2]$, $\phi \in [5\pi/16, 11\pi/16]$) for orientations, with origin at the robot base. This range corresponds to sampling sensor poses in close proximity to the tool pose of the floor mounted robot in Fig. 5. And the synthetic tests in [10] suggests that this number of samples should result in a solution that is close to convergence, which is further refined by solving the minimization problem in (3). A wait time of 20 seconds was used in order for the tracking dynamics to settle within a reasonable range of a few millimeters. An RViz visualization of the robot cell after calibration is given in Fig. 5.

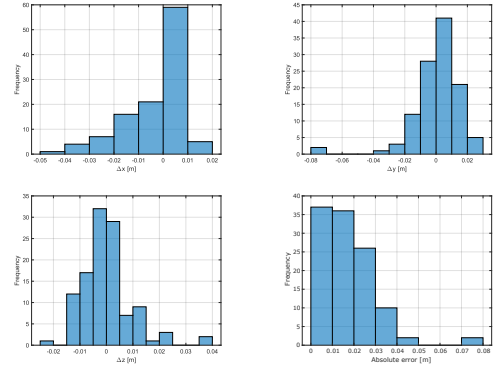


Fig. 6. Frequency distribution of the measured error along the x -, y -, z -axis and their absolute values with the forward kinematics of the robot as a ground truth.

C. Testing the Calibrated System

A volume of $1.0\text{ m} \times 3.0\text{ m} \times 1.0\text{ m}$ in the center of the robot cell was sampled with the calibrated system at $4 \times 7 \times 3$ distributed points, in order to show an indication of its accuracy. The sampling was performed with the same setup as the calibration, with a tracked device firmly attached to the robot. Each sample was compared with an ideal sensor pose, which was computed with the forward kinematics of the robot and the estimated solution $\hat{\mathbf{X}}$ from solving the hand-eye problem:

$$\tilde{\mathbf{T}}_{rc}^s = \mathbf{T}_{rc}^{vr} \mathbf{T}_{vr}^s - (\mathbf{T}_t^{rc})^{-1} \hat{\mathbf{X}} \quad (11)$$

with transformations defined as in 5.

The sampling procedure was run twice in order to validate the runs against each other, and resulted in the frequency distribution of Fig. 6. This distribution has a small negative bias along the x -axis, where the resulting mean along the axes was found to be $[-0.003363, 0.0002183, 0.0003022]$ meters with standard deviation $[0.0119, 0.01468, 0.009321]$ meters. The cause of this bias is not well-understood, but the tracking dynamics seems to imply that it is caused by a larger drift along the x -axis. These results indicates a measurement error in the centimetric range, and the maximum absolute error was 8 cm.

D. Assembly Use-Case

The calibrated system was tested on a simple assembly scenario that is shown in Fig. 7, where a mock-up for inserting a rotor into a motor housing was mapped with a tracking device. A simple tool was made from a tracked device with a spike probe attached to it, as shown in Fig. 8. The tool was used to register the necessary points to define collisions in the assembly scenario, by pointing the spike at points and pressing a button. Figure 9 shows the virtual representation of the scenario, which was meticulously defined in about five minutes. The dimensions of the Euro-pallet in this figure was defined with centimetric accuracy and a millimetric deviation between similar parts.



Fig. 7. Assembly scenario.



Fig. 8. Simple tool based on an HTC Vive Tracker with a 15 cm long and 1 cm thick spike probe screwed into its quarter inch UNC threaded camera mount.

IV. DISCUSSION

The calibrated system has a few problems that are related to specific issues of the lighthouse tracking. These issues are mentioned in literature about the Vive and its tracking system, but the documentation about troubleshooting and correcting them is sparse. This article hopes to rectify some of this sparsity by outlining the issues, their cause, and potential fixes in a concise manner.

1) *Prioritizing Inertial Measurements*: [2] showed that the Vive's tracking algorithm gives greater weight to its inertial measurements in order to generate smooth trajectories for VR applications. This weighting can clearly be seen in Fig. 10, where a tracked device was moved quickly between two points. The error is converging a lot faster when the tracked device is moving, which then slowly approaches its final value with an overdamped (second order) impulse response. Although the wait time that was used for calibration is low relative to the tracking dynamics and causes a millimetric error, the hand-eye calibration is robust.

Convergence can take as long as 500 seconds, and causes an error in the millimetric range when measuring the position

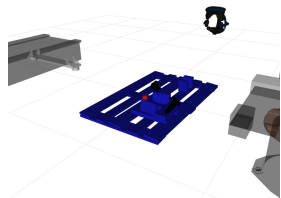


Fig. 9. Virtual representation of the assembly scenario, as visualized in RViz. The red sphere is the tip of the spike probe.

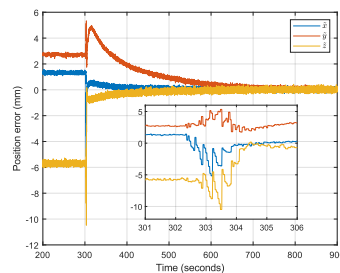


Fig. 10. Error response from moving a tracked device quickly between two points. The move starts at 302, 4 seconds, and it lasts approximately 2 seconds.

of a device before it has converged. The only known way of avoiding this error is the use of a third party tracking algorithm, which is exactly what Borges et al. introduced in their article [2]. An open-source back-end such as LibSurvive has to be used in place of SteamVR, in order to use a third-party tracking algorithm.

2) *Tilted Reference Frame*: [9] reported that poses measured with the Vive are provided in a reference frame that is tilted with respect to the physical ground plane. This issue is caused by the fact that the reference frame is aligned with the gravity vector, which is estimated with an IMU in the tracked device. The tilted reference frame is a symptom of sensor bias in the IMU [16], and the solution is to either return the device or recalibrate the IMU [14]. Access to the calibration tools requires a SteamVR tracking license. The tilted floor is not an issue for the calibration procedure that is presented in this article, as it relies on an external calibration that does not depend on the choice of coordinates.

3) *Switching Bias*: [9] also observed a large systematic error that switched its value whenever tracking was briefly lost. According to the inventor of lighthouse tracking, Alan Yates (Reddit username: vk2zay), the error occurs whenever the base stations disagree with each other by a large amount [13]. The error is caused by a recalibration of the base stations, in order to reduce the discrepancy between them. This recalibration shows up as a bootstrapping of one of the base stations in the web console of SteamVR. The resulting error is nonlinear in Euclidean space, as the pose of the base stations is changed internally in the tracking system. It was noted that this change occurs instantaneously for all devices, and a monitor was added to the ROS node in order to warn the user if a recalibration has occurred.

This recalibration can be avoided for the most part, by always keeping a tracked device in a location that is visible to both base stations without risk of concealment. The head-mounted display (HMD) in Fig. 4 was used for this purpose.

4) *Tracking Jitter*: The final and perhaps most common issue is tracking wobble and jitter, which is caused by reflections from the environment. Robot cells, for instance, are often enclosed by a fencing system with clear polycarbonate for safety reasons. This enclosure causes reflections that may

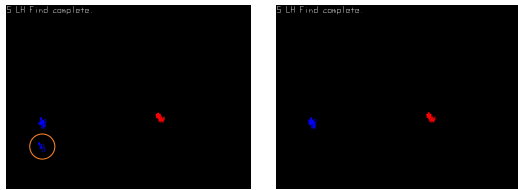


Fig. 11. The LibSurvive calibration tool before (left) and after (right) the reflections in the orange ring was removed with a black piece of fabric, where the points should be clustered together.

have a negative impact on the robustness of the tracking.

The LibSurvive library is able to visualize the reflections in a 2D map through its calibration tool, as shown in Fig. 11. And this figure shows the situation before and after the black piece of fabric in Fig. 4 was added to the robot cell. Removing the reflections resulted in more robust tracking for SteamVR, and the LibSurvive tracking would not work properly without this change.

V. CONCLUSION

In this article a set of ROS packages are presented that were developed for calibration of an HTC Vive with respect to a robot cell, and rapid placement of collidable objects and identifying relevant points in the robot cell. The procedure is hardware-agnostic and can run on any system with ROS-Industrial and MoveIt! plugin. The calibration was tested using a KR16 and an assembly use-case was presented. The calibration showed a centimetric positioning error, which suggests that the system can be used for crude positioning of objects, such as for collision avoidance or high-level planning, or if the underlying control algorithm exhibits sufficient robustness to positioning errors. The article outlines some of the most common tracking issues, and gives a description of how to resolve them.

ACKNOWLEDGMENT

The work reported in this paper was partially funded by the Research Council of Norway through the projects SFI Manufacturing (contract number: 237900) and Dynamic Robot Interaction and Motion Compensation (contract number: 270941). The authors would like to thank the LibSurvive community for their troubleshooting and advice, and Rune Sandøy of SINTEF Manufacturing and Mjøs Metallvarefabrikk for the assembly usecase.

REFERENCES

- [1] M. A. Astad. “VIVE for Robotics.” Master Thesis. Department of Engineering Cybernetics, NTNU, 2019.
- [2] M. Borges, A. Symington, B. Coltin, T. Smith, and R. Ventura. “HTC Vive: Analysis and Accuracy Improvement.” In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2018, pp. 2610–2615.
- [3] C. Lohr et al. *Lightweight HTC Vive Library*. [Online; accessed 27-May-2019]. 2019. URL: <https://github.com/cnlohr/libsurvive>.
- [4] S. Chitta, I. Sukan, and S. Cousins. “MoveIt! [ROS Topics].” In: *IEEE Robotics Automation Magazine* 19.1 (Mar. 2012), pp. 18–19.
- [5] M. A. Fischler and R. C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography.” In: *Commun. ACM* 24.6 (June 1981), pp. 381–395.
- [6] T. Foote. “tf: The transform library.” In: *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. Apr. 2013, pp. 1–6.
- [7] O. Kreylos. “Lighthouse tracking examined.” In: *URL: http://doc-ok.org* (2016). [Online; accessed 25-May-2019].
- [8] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman. “Averaging quaternions.” In: *Journal of Guidance, Control, and Dynamics* 30.4 (2007), pp. 1193–1197.
- [9] D. C. Niehorster, L. Li, and M. Lappe. “The Accuracy and Precision of Position and Orientation Tracking in the HTC Vive Virtual Reality System for Scientific Research.” In: *i-Perception* 8.3 (2017), p. 2041669517708205.
- [10] F. C. Park and B. J. Martin. “Robot sensor calibration: solving $AX=XB$ on the Euclidean group.” In: *IEEE Transactions on Robotics and Automation* 10.5 (Oct. 1994), pp. 717–721.
- [11] A. Peer, P. Ullrich, and K. Ponto. “Vive Tracking Alignment and Correction Made Easy.” In: *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. Mar. 2018, pp. 653–654.
- [12] A. Perzylo et al. “SMERobotics: Smart Robots for Flexible Manufacturing.” In: *IEEE Robotics Automation Magazine* 26.1 (Mar. 2019), pp. 78–90.
- [13] Reddit. *Controllers jump when changing base station line of sight*. [Online; accessed 13-May-2019]. 2017. URL: https://www.reddit.com/r/Vive/comments/5tafa5/controllers_jump_when_changing_base_station_line/.
- [14] Reddit. *Fix for Slanted Floor Issue - IMU Recalibration*. [Online; accessed 15-May-2019]. 2017. URL: https://www.reddit.com/r/Vive/comments/6tzthx/fix_for_slanted_floor_issue_imu_recalibration/.
- [15] Y. C. Shiu and S. Ahmad. “Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form $AX=XB$.” In: *IEEE Transactions on Robotics and Automation* 5.1 (Feb. 1989), pp. 16–29.
- [16] Steam. *Floor Tilt a Hardware or Software Issue?* [Online; accessed 15-May-2019]. 2016. URL: <https://steamcommunity.com/app/358720/discussions/0/353916981477560813/?tscn=1490201536>.
- [17] R. Y. Tsai and R. K. Lenz. “A new technique for fully autonomous and efficient 3D robotics hand/eye calibration.” In: *IEEE Transactions on Robotics and Automation* 5.3 (June 1989), pp. 345–358.

BIBLIOGRAPHY

- [1] J. Kaeser. "Why Robots Will Improve Manufacturing Jobs." In: *Why Robots Will Improve Manufacturing Jobs* (Sept. 2017). URL: <http://time.com/4940374/joe-kaeser-siemens-robots-jobs/>.
- [2] G. Papadopoulos, S. Rikama, P. Alajääskö, A. Airaksinen Z. Salah-Eddine (Eurostat Structural business statistics), and H. Luomaranta (Statistics Finland). "Statistics on small and medium-sized enterprises." In: *Eurostat Statistics Explained* (2018). ISSN: 2443-8219. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Statistics_on_small_and_medium-sized_enterprises.
- [3] A. Perzylo et al. "SMERobotics: Smart Robots for Flexible Manufacturing." In: *IEEE Robotics Automation Magazine* 26.1 (Mar. 2019), pp. 78–90. ISSN: 1070-9932. DOI: [10.1109/MRA.2018.2879747](https://doi.org/10.1109/MRA.2018.2879747).
- [4] M. A. Astad. *VIVE-Workcell Setup and Calibration*. Department of Engineering Cybernetics, NTNU, 2018.
- [5] I. Eriksen. "Setup and Interfacing of a KUKA Robotics Lab." Master Thesis. Department of Engineering Cybernetics, NTNU, 2017.
- [6] Department of Engineering Cybernetics, NTNU. *Thrivaldi Documentation*. [Online; accessed 24-April-2019]. URL: <https://github.com/itk-thrivaldi/Documentation>.
- [7] Open Source Robotics Foundation. *ROS Core Components*. [Online; accessed 5-May-2019]. URL: <http://www.ros.org/core-components/>.
- [8] T. Foote. "tf: The transform library." In: *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. Apr. 2013, pp. 1–6. DOI: [10.1109/TePRA.2013.6556373](https://doi.org/10.1109/TePRA.2013.6556373).
- [9] Sameer Agarwal, Keir Mierle, et al. *Ceres Solver*. <http://ceres-solver.org>.
- [10] O. Egeland and J. T. Gravdahl. *Modeling and simulation for automatic control*. Vol. 76. Marine Cybernetics Trondheim, Norway, 2002.

- [11] Y. C. Shiu and S. Ahmad. "Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form $AX=XB$." In: *IEEE Transactions on Robotics and Automation* 5.1 (Feb. 1989), pp. 16–29. ISSN: 1042-296X. DOI: [10.1109/70.88014](https://doi.org/10.1109/70.88014).
- [12] F. C. Park and B. J. Martin. "Robot sensor calibration: solving $AX=XB$ on the Euclidean group." In: *IEEE Transactions on Robotics and Automation* 10.5 (Oct. 1994), pp. 717–721. ISSN: 1042-296X. DOI: [10.1109/70.326576](https://doi.org/10.1109/70.326576).
- [13] H. H. Chen. "A screw motion approach to uniqueness analysis of head-eye geometry." In: *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. June 1991, pp. 145–151. DOI: [10.1109/CVPR.1991.139677](https://doi.org/10.1109/CVPR.1991.139677).
- [14] K. Daniilidis. "Hand-Eye Calibration Using Dual Quaternions." In: *The International Journal of Robotics Research* 18.3 (1999), pp. 286–298. DOI: [10.1177/02783649922066213](https://doi.org/10.1177/02783649922066213).
- [15] M. Feuerstein. *Hand-Eye Calibration*. Nov. 2009. URL: <http://campar.in.tum.de/Chair/HandEyeCalibration>.
- [16] D. Condurache and V. Martinusi. "Computing the Logarithm of Homogenous Matrices in $SE(3)$." In: 1st International Conference "Computational Mechanics and Virtual Engineering", COMEC 2005, Brasov, Sept. 2005. ISBN: 973-635-593-4.
- [17] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman. "Averaging Quaternions." In: *Journal of Guidance, Control, and Dynamics* 30.4 (2007), pp. 1193–1197. DOI: [10.2514/1.28949](https://doi.org/10.2514/1.28949).
- [18] G. Wahba. "A Least Squares Estimate of Satellite Attitude." In: *SIAM Review* 7.3 (1965), pp. 409–409. DOI: [10.1137/1007077](https://doi.org/10.1137/1007077).
- [19] F. L. Markley and D. Mortari. "Quaternion attitude estimation using vector observations." In: *Journal of the Astronautical Sciences* 48.2 (Apr. 2000), pp. 359–380.
- [20] I. Y. Bar-Itzhack and Y. Oshman. "Attitude Determination from Vector Observations: Quaternion Estimation." In: *IEEE Transactions on Aerospace and Electronic Systems* AES-21.1 (June 1985), pp. 128–136. ISSN: 0018-9251. DOI: [10.1109/TAES.1985.310546](https://doi.org/10.1109/TAES.1985.310546).

- [21] B. K. P. Horn. "Closed-form solution of absolute orientation using unit quaternions." In: *J. Opt. Soc. Am. A* 4.4 (Apr. 1987), pp. 629–642. DOI: [10.1364/JOSAA.4.000629](https://doi.org/10.1364/JOSAA.4.000629).
- [22] J. Steuer. "Defining Virtual Reality: Dimensions Determining Telepresence." In: *Journal of Communication* 42.4 (1992), pp. 73–93. DOI: [10.1111/j.1460-2466.1992.tb00812.x](https://doi.org/10.1111/j.1460-2466.1992.tb00812.x).
- [23] A. Gilerson. *RVIZ Plugin for the HTC Vive*. [Online; accessed 09-June-2019]. URL: https://github.com/AndreGilerson/rviz_vive_plugin.
- [24] GitHub user: nairol. *Lighthouse Reverse-Engineered Documentation*. [Online; accessed 5-May-2019]. URL: <https://github.com/nairol/LighthouseRedox>.
- [25] M. A. Fischler and R. C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: [10.1145/358669.358692](https://doi.org/10.1145/358669.358692).
- [26] O. Kreylos. "Lighthouse tracking examined." In: URL: <http://doc-ok.org> (2016). [Online; accessed 25-May-2019]. URL: <http://doc-ok.org/?p=1478>.
- [27] Steam. *SteamVR Tracking Technology Update*. [Online; accessed 15-May-2019]. 2017. URL: <https://steamcommunity.com/games/steamvrtracking/announcements/detail/1264796421606498053>.
- [28] D. C. Niehorster, L. Li, and M. Lappe. "The Accuracy and Precision of Position and Orientation Tracking in the HTC Vive Virtual Reality System for Scientific Research." In: *i-Perception* 8.3 (2017), p. 2041669517708205. DOI: [10.1177/2041669517708205](https://doi.org/10.1177/2041669517708205).
- [29] M. Borges, A. Symington, B. Coltin, T. Smith, and R. Ventura. "HTC Vive: Analysis and Accuracy Improvement." In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2018, pp. 2610–2615. DOI: [10.1109/IROS.2018.8593707](https://doi.org/10.1109/IROS.2018.8593707).
- [30] HTC Corporation. *Tips for setting up the base stations*. [Online; accessed 05-May-2019]. URL: https://www.vive.com/eu/support/vive-pro-hmd/category_howto/tips-for-setting-up-the-base-stations.html.

- [31] R. Y. Tsai and R. K. Lenz. "A new technique for fully autonomous and efficient 3D robotics hand/eye calibration." In: *IEEE Transactions on Robotics and Automation* 5.3 (June 1989), pp. 345–358. ISSN: 1042-296X. DOI: [10.1109/70.34770](https://doi.org/10.1109/70.34770).
- [32] Reddit. *Controllers jump when changing base station line of sight*. [Online; accessed 13-May-2019]. 2017. URL: https://www.reddit.com/r/Vive/comments/5tafa5/controllers_jump_when_changing_base_station_line/.
- [33] M. A. Astad. *Moved API structs from source to header file #143*. [Online; accessed 07-June-2019]. 2019. URL: <https://github.com/cnlohr/libsurvive/pull/143>.
- [34] M. A. Astad. *button_process: SurviveObject is 0 when pressing the grip button or when (first) touching the touchpad #141*. [Online; accessed 07-June-2019]. 2019. URL: <https://github.com/cnlohr/libsurvive/issues/141>.
- [35] A. T. Miller, S. Knoop, H. I. Christensen, and P. K. Allen. "Automatic grasp planning using shape primitives." In: *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*. Vol. 2. Sept. 2003, 1824–1829 vol.2. DOI: [10.1109/ROBOT.2003.1241860](https://doi.org/10.1109/ROBOT.2003.1241860).
- [36] A. Peer, P. Ullich, and K. Ponto. "Vive Tracking Alignment and Correction Made Easy." In: *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. Mar. 2018, pp. 653–654. DOI: [10.1109/VR.2018.8446435](https://doi.org/10.1109/VR.2018.8446435).
- [37] S. Chitta. *URDF 2.0: Update the ROS URDF Format*. [Online; accessed 07-June-2019]. URL: <http://sachinchitta.github.io/urdf2/>.
- [38] Open Source Robotics Foundation. *SDF specification*. [Online; accessed 08-June-2019]. 2019. URL: <http://sdformat.org/spec>.
- [39] Steam. *Floor Tilt a Hardware or Software Issue?* [Online; accessed 15-May-2019]. 2016. URL: <https://steamcommunity.com/app/358720/discussions/0/353916981477560813/?tscn=1490201536>.
- [40] Reddit. *Fix for Slanted Floor Issue - IMU Recalibration*. [Online; accessed 15-May-2019]. 2017. URL: https://www.reddit.com/r/Vive/comments/6tzthx/fix_for_slanted_floor_issue_imu_recalibration/.

- [41] J. D. Bergman. *BSD Calibration Values*. [Online; accessed 09-June-2019]. URL: <https://github.com/cnlohr/libsurvive/wiki/BSD-Calibration-Values>.

