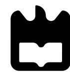


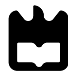
Universidade de Aveiro Departamento de Engenharia Mecânica

2021

Miguel Duarte Rocha Pina

Development of an interface based on ROS-RVIZ for calibration systems

Desenvolvimento de uma interface baseada em ROS-RVIZ para sistemas de calibração



Universidade de Aveiro Departamento de Engenharia Mecânica
2021

Miguel Duarte Rocha Pina

Development of an interface based on ROS-RVIZ for calibration systems

Desenvolvimento de uma interface baseada em ROS-RVIZ para sistemas de calibração

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica do Doutor Miguel Armando Riem de Oliveira, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro, e do Doutor Eurico Farinha Pedrosa, Investigador Doutorado (nível 1) do Departamento de Eletrónica, Telecomunicações e Informática do Instituto de Engenharia Eletrónica e Telemática de Aveiro .

o júri / the jury

presidente / president	**Professor Doutor José Paulo Oliveira Santos** Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro
vogais / committee	**Doutor João Manuel Leite da Silva** Investigador Sénior na empresa *VORTEX-CoLab* (arguente)
	Professor Doutor Miguel Armando Riem de Oliveira Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Começo por deixar um agradecimento especial ao meu orientador, Professor Miguel Riem de Oliveira, pelo conhecimento que me foi transmitindo e pelo apoio dado, sobretudo na fase inicial desta dissertação, tendo-me dado a motivação para superar um início algo atribulado sem nunca me deixar perder o rumo ao objetivo final. Deixo também um agradecimento ao meu co-orientador, Professor Eurico Farinha Pedrosa, pelo acompanhamento e pelas dicas que foi dando no decorrer deste trabalho.

Um agradecimento ao Pedro, ao Ruben e ao Zé, não só por todas as vezes que me ajudaram no decorrer do curso, mas também pelos anos de amizade que lhe antecederam e os que ainda hão de vir. Um agradecimento também ao Ricardo, ao Nuno e ao Daniel, que desde o 1º ano foram a minha companhia em praticamente todos os projetos e trabalhos de grupo e em quase todas as aulas, tendo criado laços de amizade que certamente se manterão. Fica ainda um agradecimento aos restantes colegas cujos caminhos se cruzaram com o meu no decorrer de todo este trajeto, e que sempre mostraram um grande sentido de companheirismo e entreajuda.

Por último, deixo um grande agradecimento à minha família. Ao meu pai, por tudo o que fez para me proporcionar todas as oportunidades que tive. Ao meu irmão, que sempre me ajudou e serviu de exemplo para mim. E à minha mãe, por todo o amor e carinho e por, ainda que tenha partido antes de me poder ver a começar esta jornada, sempre ter sido um grande apoio para mim e ser a base de tudo aquilo que sou e alcancei.

keywords Robotics; Extrinsic Calibration; ATOM; ROS; RViz plugin; Interface

abstract Sensor calibration is an essential prerequisite for many applications in the field of robotics. For complex robotic systems with several sensors of different modalities, the way to perform an extrinsic calibration is usually through sequential pairwise calibrations, which comes with its problems. The major shortcoming of sequential pairwise approaches is related to the fact that the transformation between two sensors is estimated only considering the error between the data from the selected pair of sensors. This leads to a lack of accuracy in the calibration procedure, given that, despite the fact that the pose of each sensor in relation to the other is well estimated, their pose relatively to the rest of the sensors and the robot is compromised. ATOM is a calibration framework for multi-sensor, multi-modal robotic systems, based on the optimization of atomic transformations, that offers a solution for this problems and it is integrated with the ROS framework. However, like most calibration systems, it lacks a graphical interface to facilitate the execution of the several steps of the calibration procedure. This dissertation's goal was to create that interface and have it integrated with the ROS 3D visualization tool, RViz, as a plugin. This interface allows its users to interact with the ATOM calibration system in a more dynamic way, through graphical icons and visual indicators. Some calibrations using the developed interface were then conducted for different robotic systems to test its usability, with satisfying results, offering a more user-friendly experience for the calibration procedure.

palavras-chave Robótica; Calibração Extrínseca; ATOM; ROS; RViz plugin; Interface

resumo

A calibração de sensores é um pré-requesitio essencial para muitas aplicações na área da robótica. Para sistemas robóticos complexos com vários sensores de diferentes modalidades, a maneira como é feita uma calibração extrínsica é através de calibrações par a par sequenciais, o que apresenta os seus problemas. O maior problema que este tipo de abordagem apresenta é o facto da transformação entre dois sensores ser estimada considerando apenas o erro entre os dados dos dois sensores. Isto leva a uma baixa precisão no processo de calibração, visto que, apesar da posição e orientação de cada sensor em relação ao outro ser bem estimada, as suas posição e orientação relativamente ao robô e aos restantes sensores ficam comprometidas. O ATOM é uma estrutura de calibração para sistemas robóticos com múltiplos sensores de múltiplas modalidades, baseada na otimização de transformações atómicas, que oferece uma solução para estes problemas e está integrada com o ROS. No entanto, tal como a maioria dos sistemas de calibração, não tem uma interface gráfica que facilite a execução dos vários passos relativos ao procedimento da calibração. O objetivo desta dissertação foi criar esta interface e integrá-la sob a forma de plugin com a ferramenta de visualização 3D do ROS, o RViz. Esta interface permite que os seus utilizadores interajam com o sistema de calibração do ATOM de uma forma mais dinâmica, através de objetos gráficos e indicadores visuais. Depois de implementada, foram feitas algumas calibrações usando a interface desenvolvida para diferentes sistemas robóticos para testar a sua usabilidade, obtendo resultados satisfatórios, com uma experiência de calibração mais agradável para o utilizador.

Contents

1 Introduction **1**
1.1 Problem Definition 2
1.2 Objectives 3
1.3 Document Structure 3

2 Related Work **5**
2.1 Graphical Interfaces for Robot Applications 5
2.2 Graphical Interfaces for Calibration 8
2.3 ATOM Calibration Framework 18
2.3.1 Parameters Configuration 19
2.3.2 Initial Estimate 20
2.3.3 Data Collection 21
2.3.4 Optimization Procedure 22
2.4 Summary 23

3 Software Infrastructure **25**
3.1 ROS - Robot Operating System 25
3.1.1 ROS packages 25
3.1.2 ROS nodes, topics, messages and services 25
3.1.3 ROS Master 26
3.1.4 ROS Parameter Server 26
3.1.5 ROS launch files 26
3.1.6 ROS bag files 27
3.1.7 ROS introspection tools 27
3.2 RViz 28
3.2.1 Robot Model 29
3.2.2 Tf 29
3.2.3 Point Cloud 30
3.2.4 Image 31
3.2.5 Camera 31
3.2.6 Markers 32
3.2.7 Interactive Markers 32
3.2.8 Custom Plugins 34
3.3 Qt 35

4 Approach **37**
4.1 RViz plugin implementation 37
4.2 Calibration procedure 40
4.2.1 Parameters Configuration 42
4.2.2 Set Initial Estimate 50
4.2.3 Data Collection 55
4.2.4 Calibration 61

5 Results **67**
5.1 Interface overview 67
5.2 Testing the interface with other robotic systems 68
5.3 Comparison of the interface with the HandEye Calbration GUI from MoveIt! 73

6 Conclusions **77**
6.1 Future Work 78

References **78**

List of Figures

2.1 Software Architecture of MoveIt! 6
2.2 MoveIt! Setup Assistant GUI. 7
2.3 Motion Planning panel from MoveIt! 7
2.4 Example of the calibration procedure using the easyhandeye interface. . . 9
2.5 RQt plugin for industrial extrinsic calibration. 10
2.6 Visual representation of both cases of the hand-eye calibration problem. . 11
2.7 Automatic Robot Movements Interface 12
2.8 Calibration Interface for the Easy Hand-Eye package. 13
2.9 Selection of the HandEye calibration panel. 14
2.10 First tab widget of the HandEye Calibration interface. 15
2.11 Second tab widget of the HandEye Calibration interface. 16
2.12 Third tab widget of the HandEye Calibration interface. 17
2.13 Menu when right-clicking on interactive marker. 20

3.1 rqt screen showing a simple node graph. 27
3.2 Default RViz window. 28
3.3 Robot Model Display Type in RViz. 29
3.4 Tf Display Type in RViz. 30
3.5 PointCloud Display Type in RViz. 30
3.6 Image Display Type in RViz. 31
3.7 Camera Display Type in RViz. 31
3.8 Different markers displayed on RViz. 32
3.9 4 Interactive Markers displayed on RViz. 33
3.10 Communication of interactive markers with RViz. 33
3.11 Screenshot of the RViz environment with the added plugin in it. 34
3.12 Screenshot of a panel being designed using Qt Designer. 35

4.1 Screenshot of a simple button being added to Qt Designer. 38
4.2 Simple Panel with just a button printing to the terminal. 39
4.3 mmtbot in the Gazebo Environment. 40
4.4 Transformations Tree (TF tree) of the mmtbot robotic system. 41
4.5 Layout of Configuration Tab Widget: First tab. 45
4.6 Layout of Configuration Tab Widget: Second tab. 46
4.7 Layout of Configuration Tab Widget: Third tab. 46
4.8 ROS package name inserted in the line edit. 47
4.9 Parsing of file to the Configuration Tab Widget: First tab. 47
4.10 Parsing of file to the Configuration Tab Widget: Second tab. 48

4.11 Parsing of file to the Configuration Tab Widget: Third tab. 48
4.12 Combo box of the sensors tab selected (all sensors shown). 49
4.13 Layout of the Initial Estimate Tab Widget. 50
4.14 Initial Estimate Tab: Selecting a sensor from the table of sensors. 51
4.15 Initial Estimate Tab: Sensors Visibility and Scale (default). 51
4.16 Initial Estimate Tab: Sensors Visibility and Scale (changed). 52
4.17 Initial Estimate Tab: world_camera out of place. 53
4.18 Initial Estimate Tab: world_camera sensor positioned with sliders. 53
4.19 Flowchart for the communication architecture of the initial guess phase. . 54
4.20 Layout of the Data Collection Tab widget. 56
4.21 Data Collection Tab: Interactive marker out of place. 57
4.22 Data Collection Tab: Interactive marker being positioned. 57
4.23 Data Collection Tab: 4 Collections saved. 58
4.24 Data Collection Tab: collections tree. 59
4.25 Data Collection Tab: Delete functionality - No collection selected. 59
4.26 Data Collection Tab: Delete functionality - Confirmation Dialog Box. . . 60
4.27 Flowchart for the communication architecture of the data collection phase. 60
4.28 Layout of the Configuration Tab Widget. 63
4.29 ”Help” Dialog Box that pops up everytime the ”Help” button is clicked. . 63
4.30 Configuration Tab Widget: Command-line Text Box changing. 64
4.31 Configuration of the command line text box by changing table cells. . . . 65
4.32 Visual result of running the command configured in the panel. 65

5.1 Sensors on board of the ATLASCAR2. 68
5.2 ATLASCAR2: LiDAR sensors circled in yellow and cameras in blue. . . . 69
5.3 Transformations Tree of the ATLASCAR2 robotic system. 69
5.4 ATLASCAR2 robot model represented in the RViz environment. 70
5.5 AGROB robotic system. 71
5.6 AGROB robot model: cameras circled in blue and LiDAR sensor in green. 71
5.7 Transformations Tree of the Agrob robotic system. 72
5.8 Agrob robot model represented in the RViz environment. 72
5.9 Representation of the eihbot robot model. 73
5.10 Using the Motion Planning plugin from MoveIt! 74

Listings

2.1 General structure of a robot description .xacro file 19
4.1 config.yml - Yaml file to describe the calibration parameters of the mmtbot robotic system 42
4.2 Tree View of the JSON File for the data collection 55
4.3 List of all command-line arguments on the calibrate script 61

Chapter 1

Introduction

The field of robotics has been an area of great development in this modern era and its use in the industry will only continue to grow, as robots have shown themselves to be important tools without which we could hardly achieve the quality of life that we have today. With the developments in technology, there will be increasingly more jobs to be performed by robots which would translate in many processes becoming automatic, quicker and more efficient [1].

As this field keeps evolving, there are progressively more autonomous robotic systems that rely on a broad number of sensors with different modalities, such as LiDAR (Light Detection And Ranging) and vision based sensors that can require a high level of precision. It is then crucial that the output data from these sensors is as precise as possible in order to give the robot a clear perception of its surroundings, thus allowing the robots to perform their tasks more accurately. For that to happen, these robotic systems need to go through a procedure that goes by the name of sensors calibration, which is the process of estimating both the intrinsic parameters of the sensor (e.g., focal length, image center, etc) and the extrinsic ones, i.e., the position and orientation (pose) of the sensor in respect to the world or to another sensor [2].

This task is an essential prerequisite for many applications in robotics, computer vision and augmented reality. For instance, in the field of robotics, in order to fuse measurements from different sensors, all the sensors' measurements must be expressed with respect to a common frame of reference, which requires knowing the relative pose of the sensors in order to establish a spatial relationship between them, which can be achieved by performing an extrinsic calibration [2]. In this calibration procedure, it is established an association between the incoming data from each sensor to be calibrated. An optimization procedure is then formulated to estimate the parameters of the transformation between the sensors to minimize the discrepancies between associations. Seeing that the accuracy of these associations is critical, the data for this estimation procedure is usually collected by placing a pattern that can be detected independently of the sensor modality (i.e., objects that are robustly and accurately detected) at multiple poses in the common field of view of the sensors [3, 4].

Even though the topic of sensors calibration has been tackled multiple times, there is no straightforward solution for the calibration of multiple sensors and multiple modalities in robotic systems [5]. This is due to the fact that most of the studies on calibration focus on sensor to sensor pairwise calibrations, either being between two cameras [6–10], or between cameras and LiDARs [11–15]. These approaches present some problems when

it comes to operating with more complex robotic systems that have multiple sensors of different modalities. To solve these problems, there are a few works [16–18] that tackle calibration from a multi-sensor, simultaneous optimization point of view.

However, the focus of this dissertation will be on ATOM [3–5], a project that is being carried out at the University of Aveiro that proposes a new approach to this problem, somewhat similar to [18], given that both employ a bundle adjustment-like optimization procedure, with the difference that this approach does not focus on just one robotic platform, rather it is a general approach that is applicable to any robotic system, which also relates it with [17].

1.1 Problem Definition

ATOM[1] (Atomic Transformation Optimization Method) is a calibration framework that provides tools for the calibration of multi-sensor, multi-modal robotic systems, based on the optimization of atomic transformations (geometric transformations that are not aggregated, i.e., are indivisible).

It is integrated within the Robot Operating System (ROS), which is a framework that has become the standard for the development of robotic solutions. Since this calibration approach requires the creation of a transformation tree from which the atomic transformations are optimized, ROS is ideal, as it provides a tree graph referred to as *tf tree*[2], a tool that helps defining the desired transformation. Plus, the Robot Operating System Visualization (RViz) tool supports additional functionalities, such as robot visualization, collision detection, etc. In fact, this visualization procedure is interactive, in that if any transformation between two links change, the robotic platforms and sensors that are affected by these links change their poses accordingly. This interactive procedure is possible given that the cost function of the optimization always recomputes the aggregate transformations. Hence, a change in one atomic transformation in the chain affects the global sensor pose, and consequently, causes the error to minimize [4].

This ROS calibration framework, implemented in this approach, can be divided in four main components: configuration of the calibration parameters, initial estimate, data collection and the optimization procedure. These components shall each be described in more detail in Chapter 2. For now, it is important to point out the problem at hands, which is the lack of an interface in RViz that is capable of integrating all these calibration phases. As of right now, the calibration procedure is being executed using RViz in a way that might not be as straightforward for the people that might want to use it, with the results of each phase usually having to be followed by just watching the terminal. However, it is fair to assume that the majority of people would prefer to have a more user-friendly interface so that they could configure the calibration and even see its results more easily, rather than just having to look at the command line.

Therefore, the opportunity arises to develop a simple and user-friendly yet powerful graphical interface with the sole purpose of allowing its users to interact with ATOM in a more dynamic way, through graphical icons and visual indicators.

[1] `https://github.com/lardemua/atom`
[2] `http://wiki.ros.org/tf`

1.2 Objectives

Having established that there is an opportunity to implement an interface that would facilitate the calibration procedure, the primary goal of this dissertation is to achieve precisely that. In order to do so, the Qt and ROS frameworks will be combined to create a tab based GUI (Graphical User Interface) that will be integrated as a plugin for RViz, with each tab of this interface handling each step of the calibration procedure:

- Configuration of the Calibration Parameters;
- Set Initial Estimate;
- Data Collection;
- Optimization Procedure.

For each of the objectives mentioned, there will be several functionalities that will be shown and explained later in this dissertation.

1.3 Document Structure

This document is comprised of 6 chapters.

Chapter 1 (*Introduction*) provides some context on the main subjects of this work, followed by a definition of the problem at hands and the opportunity that arises from it and lists the objectives that are expected to be achieved by the end of this project.

Following that, Chapter 2 (*Related Work*) gives an insight on the current calibration process of the ATOM Calibration project and looks into other graphical interfaces that may exist within the field of robotics and then more specifically for calibration, with a focus on *MoveIt!*, being that it has one of the most evolved interfaces (albeit for motion planning of the robot instead of sensors calibration) and one of the only interfaces used for the purpose of robot sensors calibration.

Software Infrastructure is in Chapter 3, where the software used during this dissertation is presented, namely the ROS (Robot Operating System) framework and its fundamental concepts, the ROS 3D visualization tool, RViz, for which the interface plugin will be built, and Qt, the platform in which the graphical interface was created.

The *Approach*, in Chapter 4, describes the solutions developed during the dissertation, explaining the process of what was done and the functionalities added for each step of the ATOM calibration - parameters configuration, initial estimate of the sensors pose, data collection and the calibration itself.

Afterwards, in Chapter 5, a section is dedicated to present the *Results*, consisting on giving an overview of the interface developed, testing the interface with different robotic systems to validate its usability and providing a comparison of the experience between calibrating a robot's sensors using the ATOM panel implemented in this dissertation and the other existing calibration panel, from MoveIt.

To round up the document, in Chapter 6, a chapter giving the final remarks is presented (*Conclusions*), reviewing the work as a whole, including an insight on future work that can be done to improve this interface.

Chapter 2

Related Work

In this chapter, some of the work related to the topic of this dissertation will be presented, to serve as a baseline for what will be implemented in the interface that is going to be developed and to understand in what way it will differ from the interfaces that already exist. It will be shown work on graphical interfaces created for ROS to support a particular robot functionality, followed by an insight on the work done on interfaces that are specifically targeted for calibration. Lastly, with the already existing work in mind, a brief explanation on the current calibration procedure using the ATOM calibration framework will be given in order to give a better understanding of what should be implemented in its interface.

2.1 Graphical Interfaces for Robot Applications

The use of a GUI (Graphical User Interface) in ROS to help solve existing problems in a certain robot's functionality is not new. In fact, there have been several works that integrated graphical interfaces with ROS [19–23]. However, ROS also has its own GUI-based tool, RQt, which is integrated with Qt, that can be used for this exact purpose, without the need of any other softwares.

RQt is a graphical user interface framework that implements various tools and interfaces in the form of plugins for analyzing and controlling ROS systems. It allows graphical representations of ROS nodes, topics, messages and other information. As its name indicates, the platform in which these interfaces are built is Qt, which is a framework to build graphical interfaces that is supported by ROS. Even though RQt already has several built-in interfaces that can be added to the rqt screen as plugins[1], it is possible to create a custom GUI to be added as a plugin as well, as it can be seen in these works [24–27].

However, despite it being a great tool for a quick and straightforward use, RQt is not ideal for highly customized UIs with a lot of different functionalities. For that, and taking advantage of it being supported by ROS, Qt's platform could still be used to integrate an interface as a Qt application or as a plugin for RViz. Qt applications have been used with ROS in a few works [28, 29] and could have potentially been a viable option for the interface developed in this work, if not for the fact that it runs separately from RViz, which strays away from the goal of actually having the intended plugin within RViz to

[1] `http://wiki.ros.org/rqt/Plugins`

provide a more intuitive and close to real-time calibration experience using the interface, while still being able to visualize and interact with the robotic system's components.

Therefore, the attention is shifted to plugins for RViz, with the most prominent platform for this sort of interfaces being *MoveIt!*[2]. *MoveIt!* is an open-source robotic manipulation platform with an extensible plugin architecture that allows the development of complex manipulation applications using ROS, such as motion planning, environment monitoring, trajectory control of robot manipulators, hand-eye calibration, amongst others. It builds on the ROS messaging and build systems and utilizes some of the common tools in ROS, like the ROS Visualizer (RViz) and the ROS robot format (URDF). Figure 2.1 shows a representation of the *MoveIt!* interaction between nodes/actions.

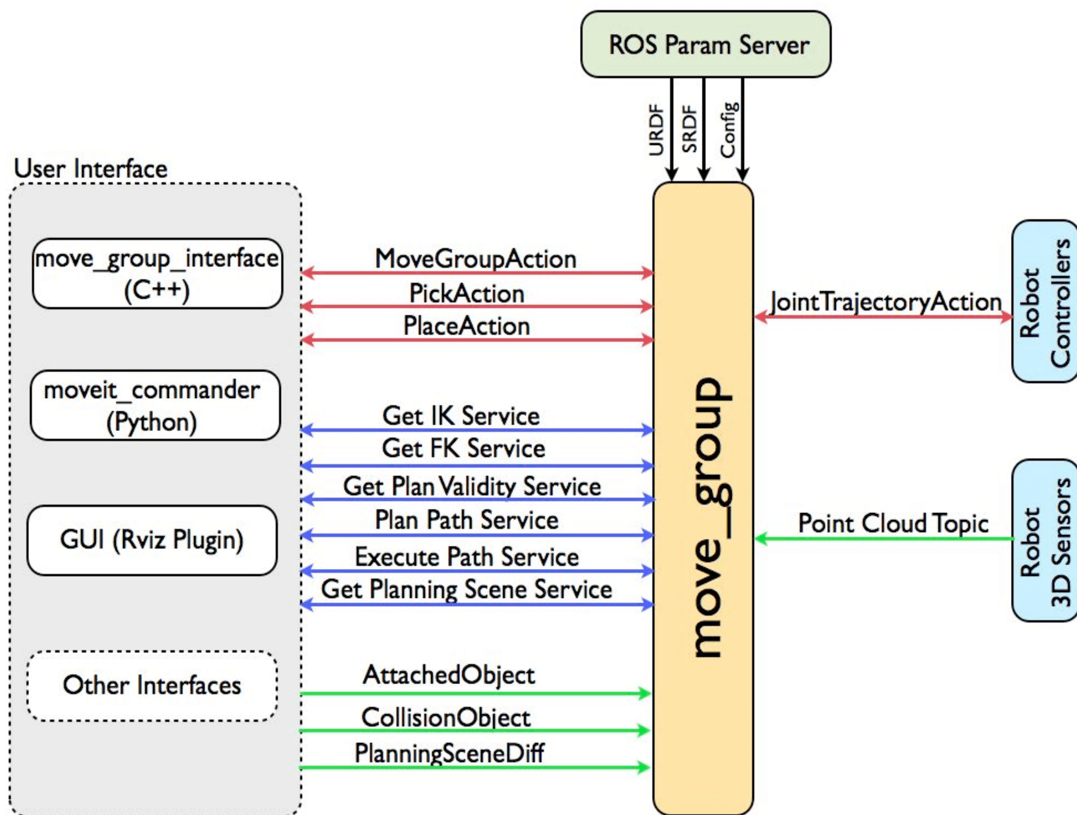


Figure 2.1: Software Architecture of MoveIt![3]

MoveIt! provides a graphical user interface (GUI) application called *MoveIt! Setup Assistant*[4] (Figure 2.2) that can be used to generate a *MoveIt!* configuration package for any robot to be used with this platform [30].

[2]https://moveit.ros.org/

[3]https://moveit.ros.org/documentation/concepts/

[4]https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html

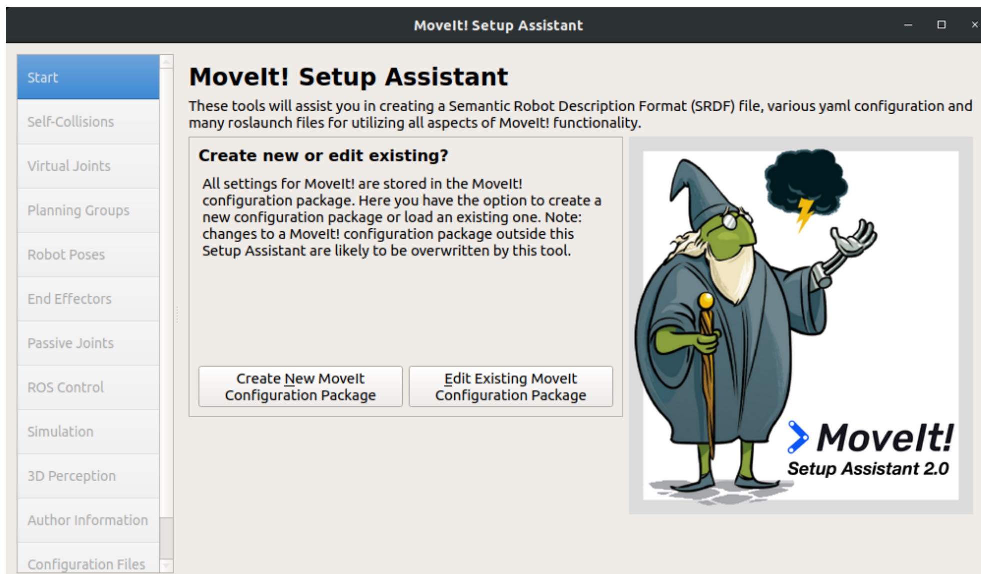


Figure 2.2: MoveIt! Setup Assistant GUI.

The plugins provided by *MoveIt!* are often used without any other interface in a variety of works [31–35], although there are also works that have integrated these plugins with their own custom interfaces [36, 37]. Amongst all these plugins, the *Motion Planning* [5], seen in figure 2.3, seems to be the most evolved existing interface that is used as a plugin for RViz.

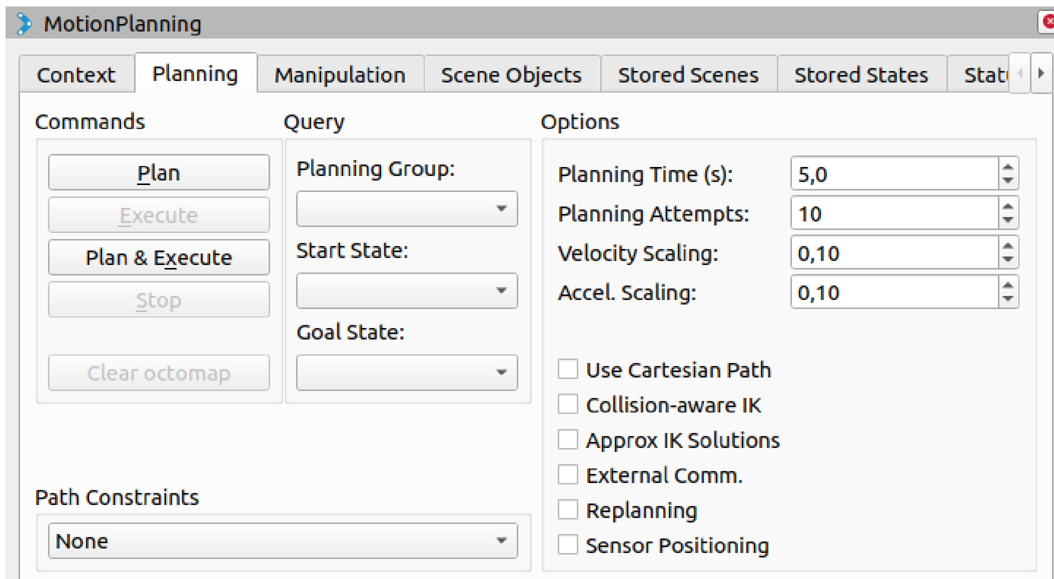


Figure 2.3: Motion Planning panel from MoveIt!

[5] https://ros-planning.github.io/moveit_tutorials/doc/motion_planning_pipeline/motion_planning_pipeline_tutorial.html

Therefore, before moving on to the main topic, which is sensors calibration, it was first given a closer look into the interface shown above, to get a grasp of how it was built and how it was integrated with RViz in preparation for the GUI being developed in this dissertation. In fact, the first step of creating the RViz plugin (just a simple panel, without any complex items or functionalities added to it) was done by taking a look into the source code of the Motion Planning package[6], since it added the plugin from the .ui file directly, instead of adding it from the Qt code as it is done in the RViz plugin tutorials[7] provided by ROS.

2.2 Graphical Interfaces for Calibration

Having talked about the use of graphical interfaces in ROS for different robot applications, it is time to discuss its use in the one robot application for what this dissertation is actually aimed for, and that is sensors calibration.

There are plenty of ROS packages created for calibration, be it intrinsic or extrinsic [37–45]. However, it is difficult to understand the working mechanism of the majority of these packages due to their lack of maintenance and detailed information available to allow a straightforward use for an appropriate test and evaluation of their performance. Amongst the ones that do provide enough information, some do not use a graphical user interface and therefore they are also not tested here, since the interest in this chapter is to study how an interface can be used for calibration. Therefore, in this section, the only packages that will be discussed are the ones that were well maintained enough to test them [37–40].

As it was previously mentioned in Chapter 1, there are two types of calibration, intrinsic and extrinsic. For the intrinsic calibration, the only interface found was a relatively simple one for the calibration of monocular or stereo cameras using a checkerboard calibration target [38].

The calibration using this interface is done by showing multiple and representative (i.e., they are not redundant) shots of a checkerboard pattern, with known dimensions, to the camera (an example is demonstrated on Figure 2.4). After enough samples are gathered, the algorithm calculates the best fitting values for the focal length, image sensor format and principal point, as well as distortion, rectification and projection coefficients[8].

[6] `https://github.com/ros-planning/moveit/tree/master/moveit_ros/visualization/motion_planning_rviz_plugin`

[7] `http://docs.ros.org/en/kinetic/api/rviz_plugin_tutorials/html/`

[8] `https://github.com/IFL-CAMP/easy_handeye`

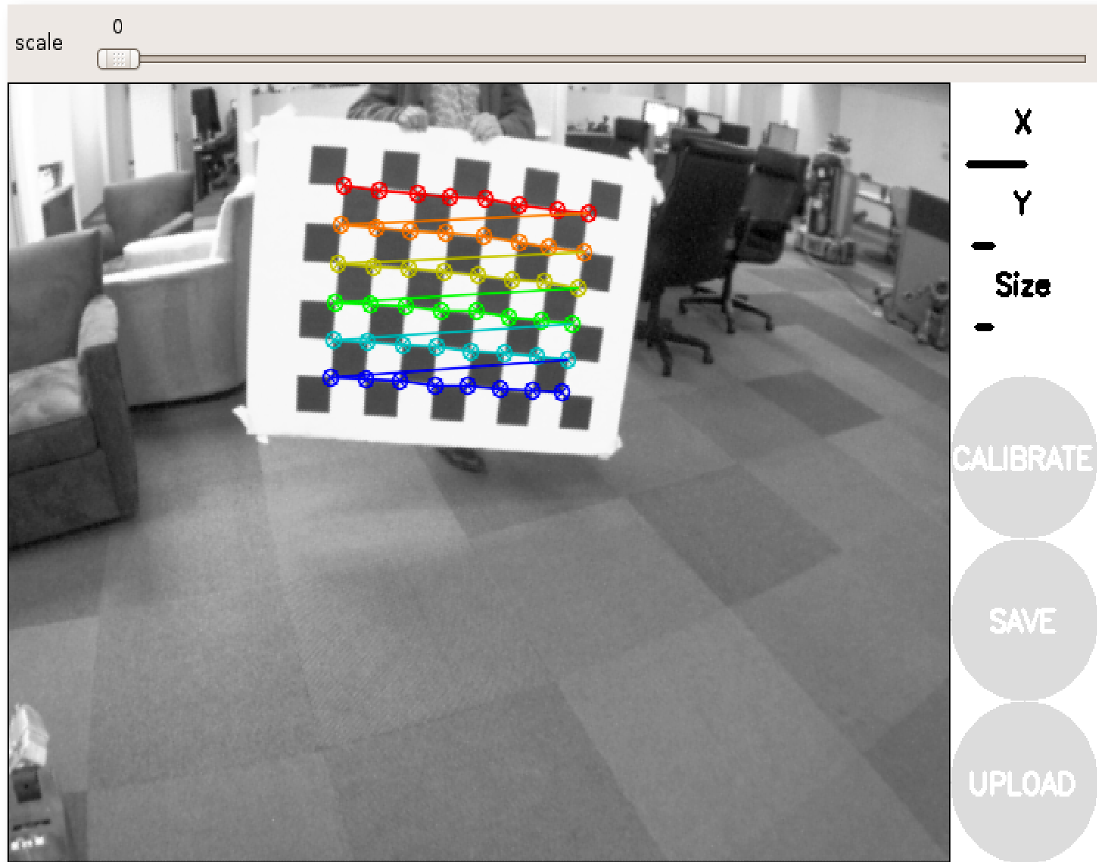


Figure 2.4: Example of the calibration procedure using the easyhandeye interface for monocular or stereo camera calibration.

However, this work is more focused on the extrinsic calibration of the robot's sensors and the interface that is going to be developed is expected to be slightly more complex than this one. For that type of calibration (extrinsic), only three graphical interfaces were found [37, 39, 40].

The graphical interface that is shown in Figure 2.5 is part of the work developed at [39], and it consists of a simple rqt plugin that was implemented in a package that provides a generic tool for calibrating sensors to a known reference frame, aimed for the industrial extrinsic calibration of a robot's sensor.

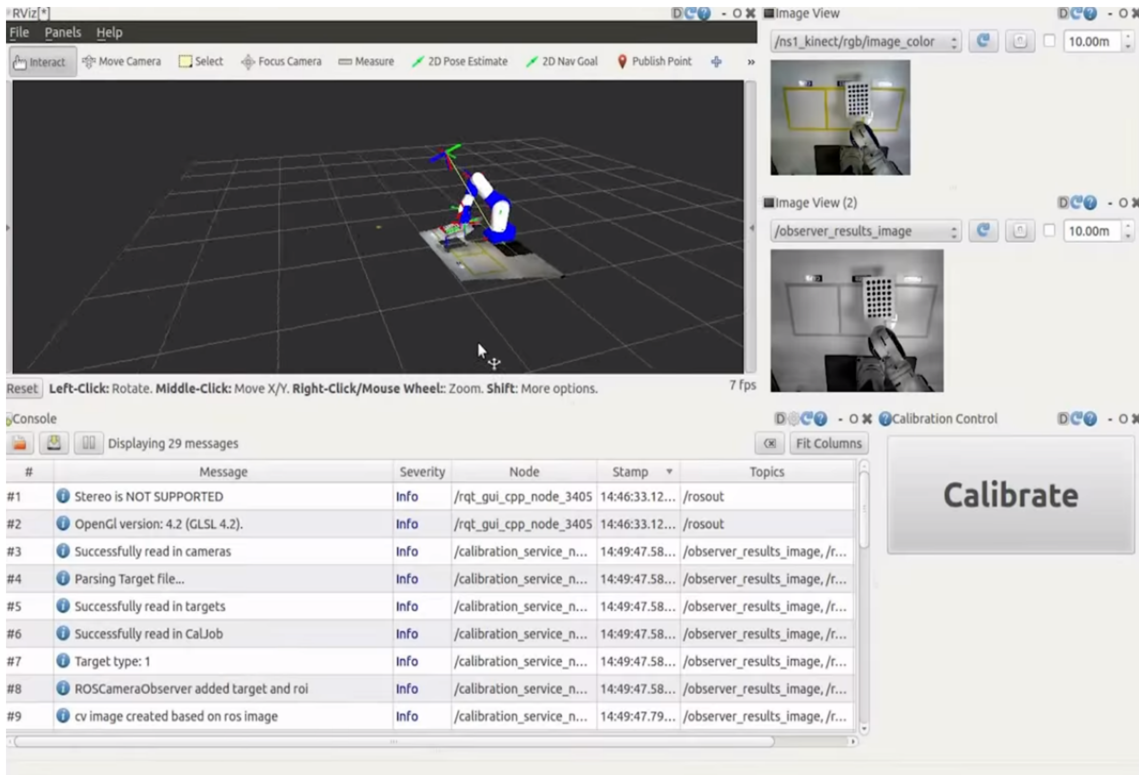


Figure 2.5: RQt plugin for industrial extrinsic calibration.

Beyond the fact that it is an RQt plugin and not an RViz plugin as it is intended for this dissertation, the above interface is limited, as it does not leave much room for a lot of configuration of the calibration parameters, nor does it allow the calibration of sensors of different modalities.

As for the graphical interfaces that are used in [37] and [40], both of them were created for performing and managing Hand-Eye calibrations in a simpler manner. However, before showing and explaining these interfaces, it is important to give a general idea of what the hand-eye calibration problem is, seeing that it also relates to the ATOM calibration project, as it presents an approach on how to solve this problem, as per explained in the article [5].

The hand–eye calibration [46–49] is a well-known calibration problem, that consists of determining the homogeneous transformation matrices (HTMs) between a robotic arm's end-effector (the *hand* of the robot), to its camera (the *eye* of the robot), as well as the transformation of the robot base to the world coordinate system [48]. In many applications, especially robot driven, hand-eye calibration is a must, as it is the binding between the robot and the camera, which makes it easy to understand why having an accurate hand-eye calibration is essential to solving the automation task at hands.

The hand-eye calibration problem can be divided in two different cases: *eye-to-hand*

and *eye-in-hand*, and both are depicted in Figure 2.6. For the eye-to-hand case, the camera is positioned in a stationary place next to a robotic arm. In this case, the transformation needs to be found from the camera's coordinate system to the coordinate system of the base of the robot. As for the eye-in-hand case, the camera is mounted on the robotic arm itself so, in this case, the transformation has to be found from the robot end-effector to the camera.

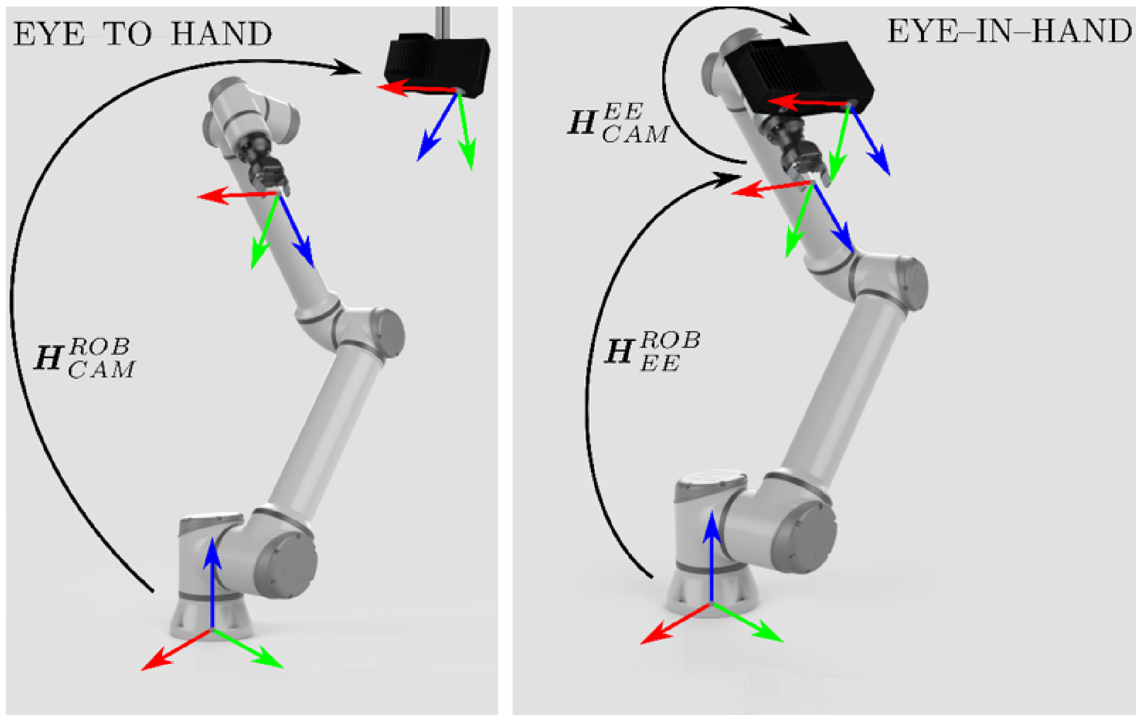


Figure 2.6: Visual representation of both cases of the hand-eye calibration problem: the *eye-to-hand* (on the left) and the *eye-in-hand* (on the right)[9]

Having established what this problem consists of, it is time to move on to the presentation of the previously mentioned interfaces.

Starting with [37], it consists of a package created to perform Hand-Eye calibration and it includes three interfaces, the first being for moving the robot around its starting position, and the GUI used for that is the *MoveIt's Motion Planning* plugin that was already mentioned in Section 2.1 and another one is a custom RQt plugin to command the calibrator script included in that package. Additionally, there is another RQt plugin that was created to guide its users through the calibration process and it can be seen in the following page (Figure 2.7).

[9]`https://blog.zivid.com/importance-of-3d-hand-eye-calibration`

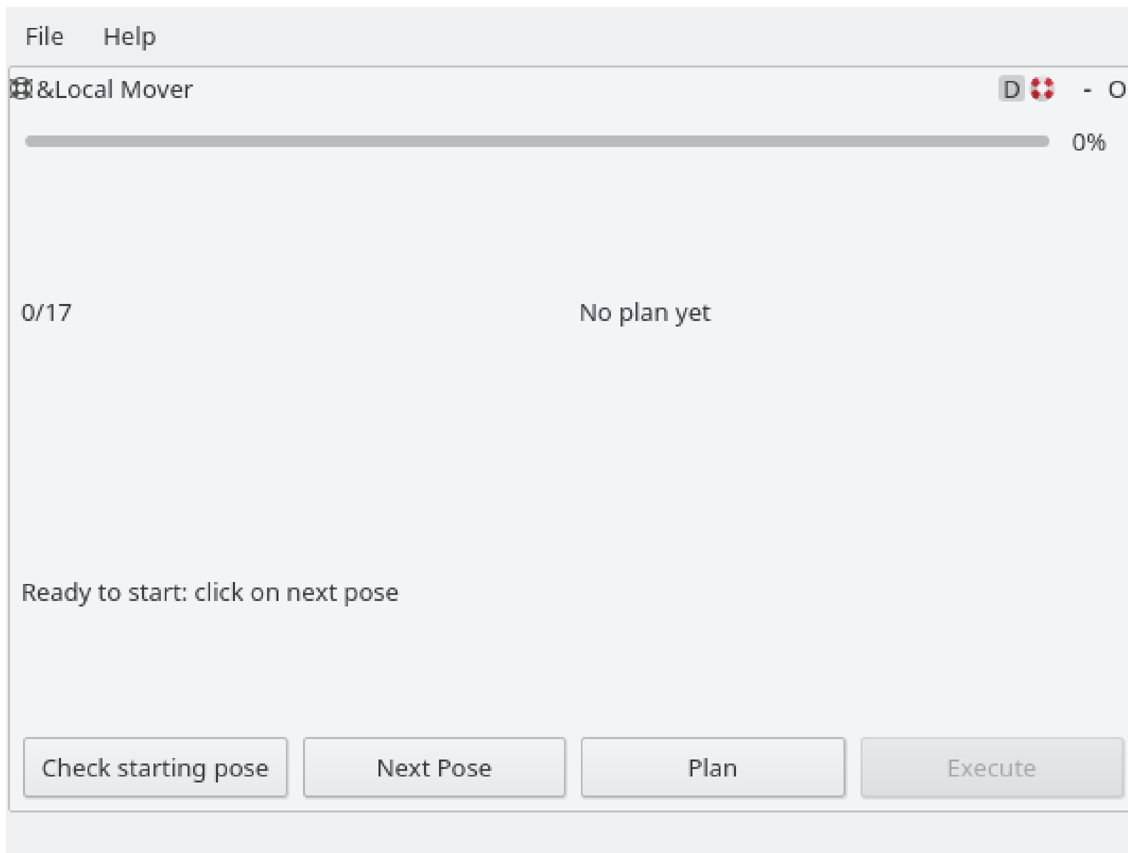


Figure 2.7: Automatic Robot Movements Interface implemented in the Easy Hand-Eye Calibration package.

The panel shown above allows its users to check if it is possible to rotate the robot's end effector around all axes, and translate it in all directions (the rotation and translation ranges can be passed as parameters). This avoids interrupting the calibration process in the middle because the robot cannot move in a certain direction due to joint limits or collisions.

This interface also allows the planning of the motion of the robot: the user can review the trajectory in RViz, provided that the motion planning plugin is activated (and correctly configured). If the joints of the robot stay within a certain range from the initial position, *MoveIt!* can be used to execute the motion plan; otherwise the point can be skipped. The joint range is also configurable as a parameter.

Once the robot has completed the motion, the user can go to the custom calibration interface shown in Figure 2.8, take a sample and proceed to the next pose. Afterwards, the robot will go back to the initial position and the process is then repeated until all samples are taken[10]. Besides taking samples, this interface also gives its users the option to review all the samples that were taken, remove a sample, compute the calibration from the current samples and save the calibration to a file.

[10] https://github.com/IFL-CAMP/easy_handeye/blob/master/rqt_easy_handeye/README.md

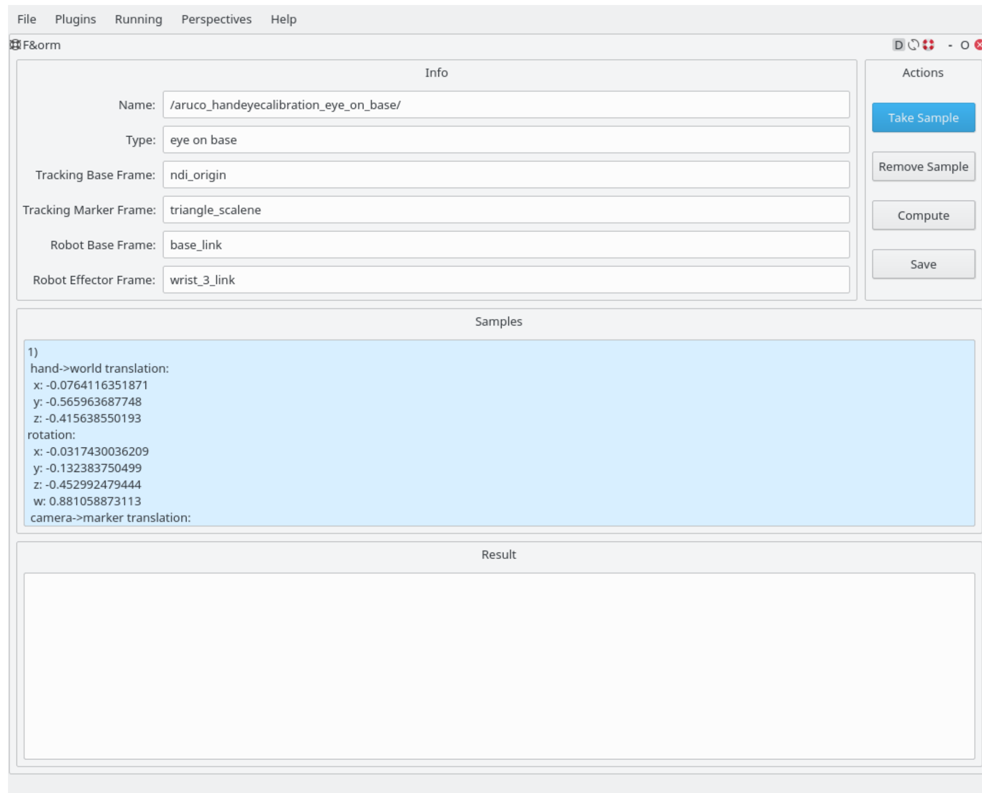


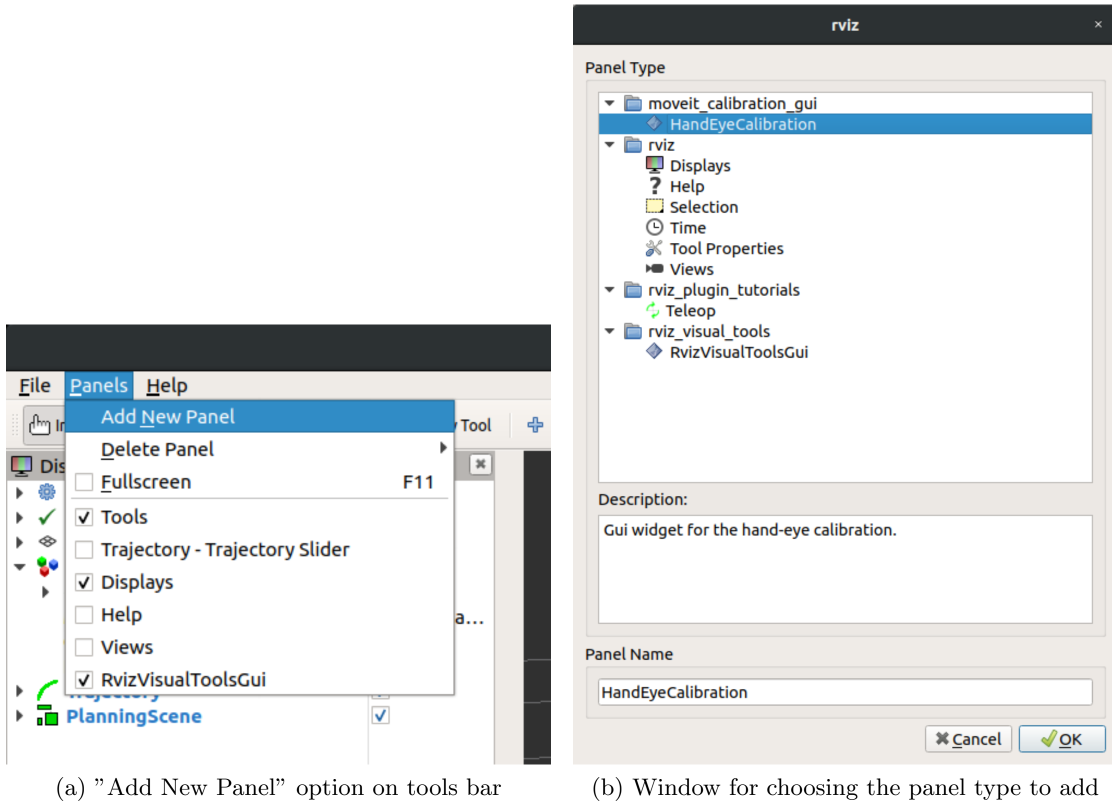
Figure 2.8: Calibration Interface for the Easy Hand-Eye package.

Having gone through the process of calibration using this package, these interfaces seemed to present some problems. First of all, as it has been mentioned several times for the previous interfaces, the goal is to have a graphical interface that is integrated within RViz and this interface is not. The second problem it presented was the fact that all throughout the process you would have to jump from interface to interface between the MoveIt! Motion Planning Interface (Figure 2.3) and the graphical interfaces seen at figures 2.7 and 2.8, which makes it a not so practical set of tools to use. The fact that the interface does not provide a lot of options for the configuration of the calibration parameters and it only calibrates one sensor at a time could also make this process last longer than it needs to be.

Finally, moving on to the Hand-Eye Calibration plugin from *MoveIt!* [40], it is perhaps the most evolved calibration interface that is known, as of now. For that reason, and being that this one is a plugin for RViz, which is what is expected with the present work, the calibration procedure for this interface will be carefully looked at, as it will serve as a good term of comparison for what is intended to be achieved.

The *MoveIt! Hand-Eye Calibration* package provides plugins and a graphical interface for conducting a hand-eye camera calibration that can be performed for cameras rigidly mounted in the robot base frame (eye-to-hand) and for cameras mounted to the end effector (eye-in-hand).

After looking into the discussion of the process for building this plugin[11], going through the necessary steps described in the tutorial[12] of this package and setting up the robotic arm to work with *MoveIt!*, the demo launchfile of the robot to be calibrated can be launched. Then, the calibration GUI can be added by going to the RViz "Panels" menu on the tools bar, choosing "Add New Panel" and selecting the "HandEyeCalibration" panel type, as seen in figure 2.9.



(a) "Add New Panel" option on tools bar (b) Window for choosing the panel type to add

Figure 2.9: Selection of the HandEye calibration panel.

The added Rviz GUI plugin contains three tab widgets, with each containing separate functions.

The first tab widget, seen in the figure 2.10 below, labeled "Target", is where the user is able to create a visual calibration target with the necessary parameters. This target has distinctive patterns that are easy to identify in the image data, and by providing a measurement of the target's real size, the pose of the target in the camera's coordinate frame can be estimated.

When conducting the hand-eye calibration, the target's precise location is not needed, because as long as the target is stationary in the robot's base frame, the hand-eye calibration can be estimated from a sequence of 5 or more poses. After the target is created, it is displayed in the panel and it can also be saved, in order to be printed out

[11] https://github.com/ros-planning/moveit/issues/1070?fbclid=IwAR18Iwtxd77-cwJfnVUULTfJqK_ml-5j32K0pHFaWEn9UVMUISDL2E6CxI4

[12] https://github.com/JStech/moveit_tutorials/blob/new-calibration-tutorial/doc/hand_eye_calibration/hand_eye_calibration_tutorial.rst

and placed near the robot, where it can be easily seen by the camera. For the target detection, the camera image topic and camera info topic can be selected from the topic filtered drop-down menus on the bottom left of the interface.

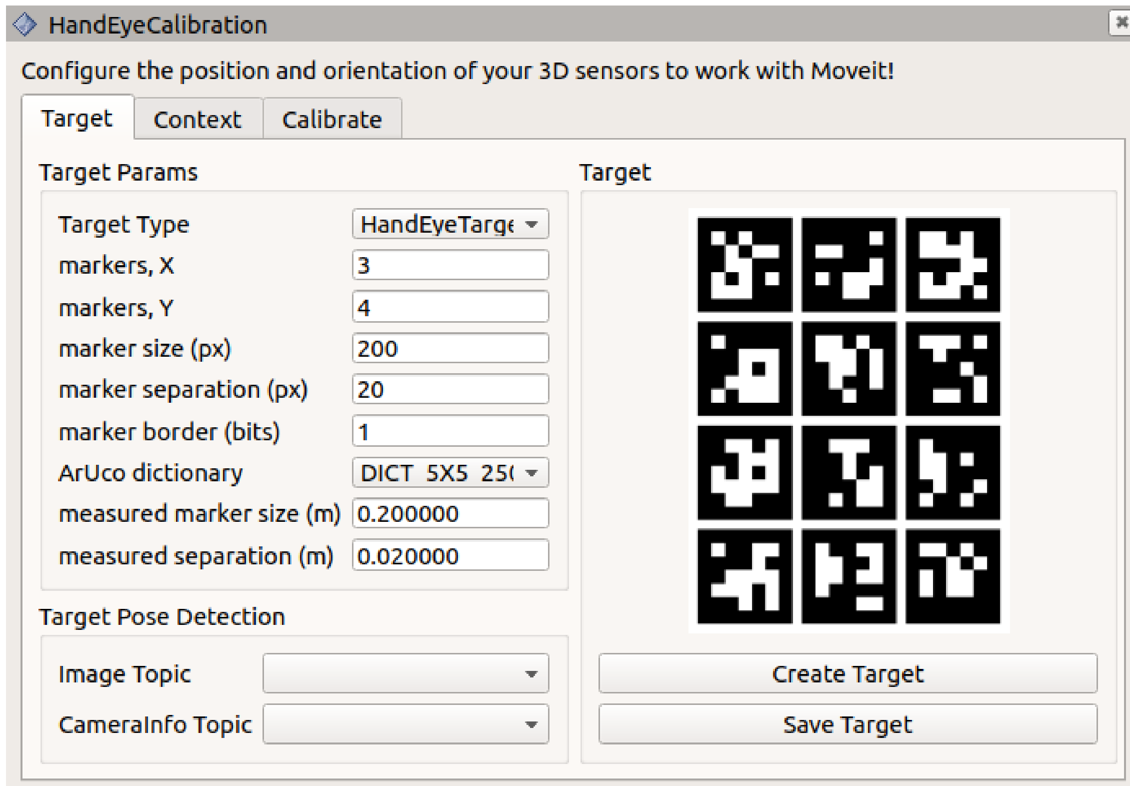


Figure 2.10: First tab widget of the HandEye Calibration interface.

The second tab widget of this GUI appears in Figure 2.11, with the name "Context", and it contains the geometric information necessary to conduct the calibration.

Its user is able to select the sensor mount type, "eye-in-hand" or "eye-to-hand", in the "general setting" groupbox. Then, in the "frames selection" groupbox, the four frame names necessary for a hand-eye calibration can be selected from available TF frames: the "sensor frame" is the camera optical frame, the "object frame" is the frame defined by the calibration target, the "end-effector frame" is the robot link that is rigidly attached to the camera (for the eye-in-hand case) and the "robot base frame" is the frame in which the calibration target is stationary. The "FOV" section controls the rendering of the camera's field of view in RViz and is generated from the received CameraInfo message. In order to see the FOV, the user needs to add a "MarkerArray" display, and set it to listen to the "/rviz_visual_tools. Lastly, on the right side of the panel, the user can set the initial pose of the camera.

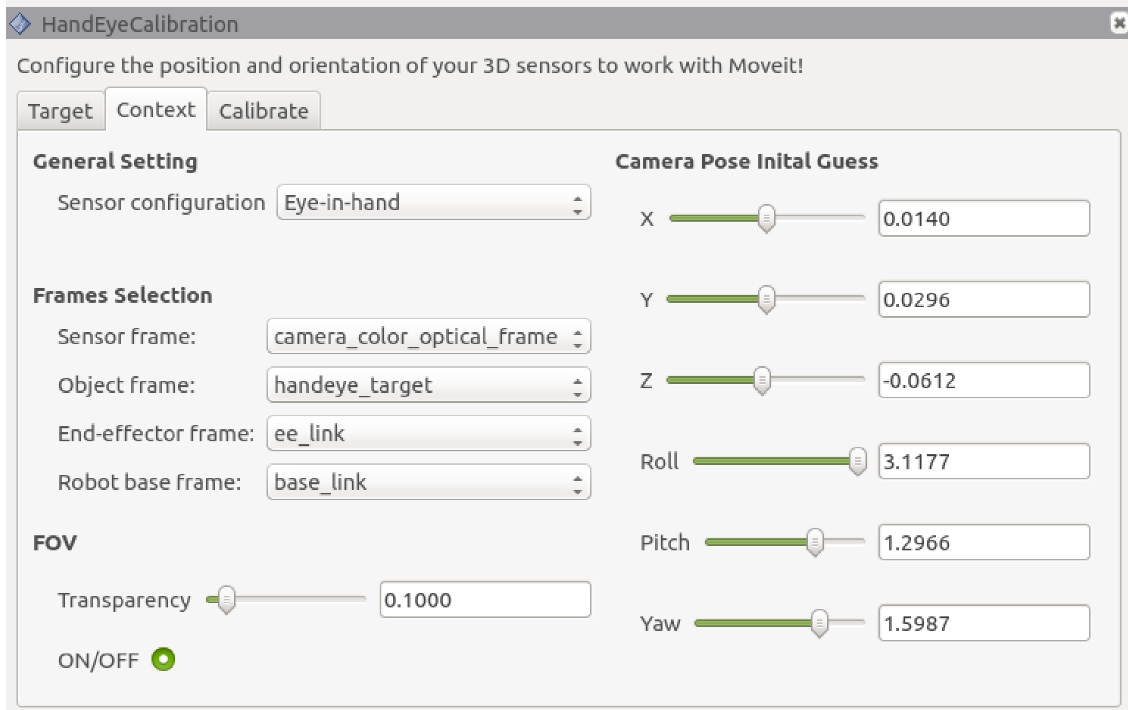


Figure 2.11: Second tab widget of the HandEye Calibration interface.

The third and last widget (”Calibrate”), shown in figure 2.12, provides the tools to collect the dataset and calculate and export the calibration. During this phase, it is helpful to add an image panel to the RViz display to see the target detection in the camera view.

On this tab, in the ”Settings” groupbox, the user can select the AX=XB solver from the loaded solver plugins. Still in the same groupbox, the user can choose the ”Planning Group” which is the joint group that will be recorded, and there is also a button to save in a .yaml file the joint states at which the samples are taken that can then be loaded back so that the same poses can be used again to recalibrate in the future.

In the ”Manual Calibration” groupbox, when the target is visible in the arm camera and the axis is rendered on the target in the target detection image, the user is ready to take the first calibration sample by clicking in the ”Take Sample” button, adding a new sample to the ”Pose samples” list on the left side of the panel. The ”Clear Samples” button deletes all samples, should the user want to retake them.

The process of taking samples is repeated until five samples are taken, from which point a calibration will be performed automatically, and then updated every time a new sample is added, with the calibration being better with every sample added (until about 12-15 samples, where it reaches a *plateau*). The calibration result can then be exported by clicking on the ”Save camera pose” button in the ”Settings” groupbox, with the result of the camera-robot pose being saved into a .launch file, which can publish the static transform tf.

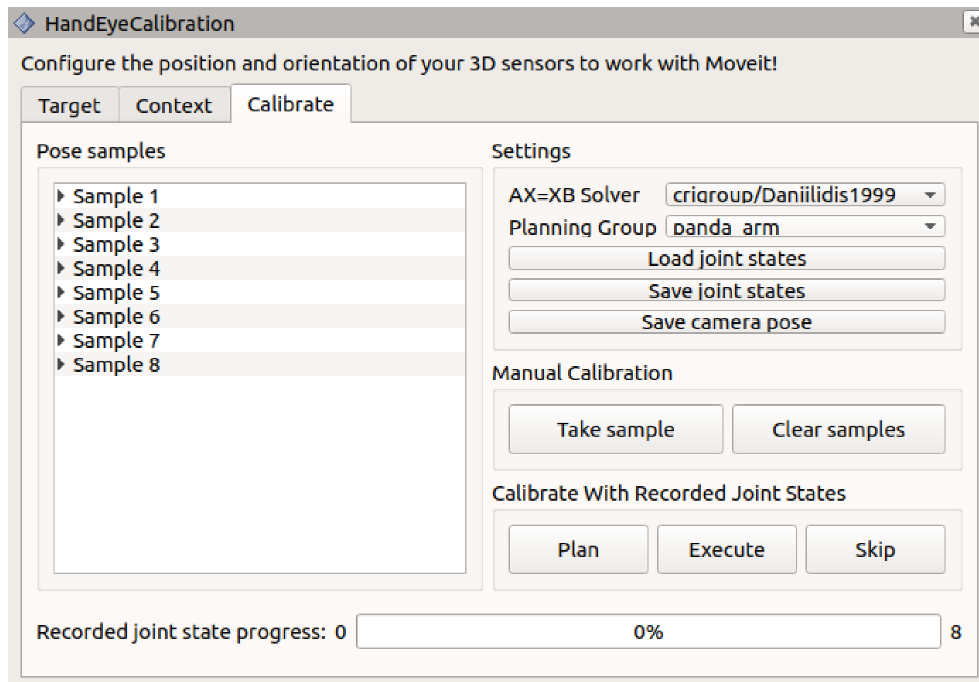


Figure 2.12: Third tab widget of the HandEye Calibration interface.

Overall, this is a relatively well organized interface, although a lot of necessary steps are needed for it to be used. With the similarities of the steps of the calibration procedure and the fact that this is an Rviz GUI plugin, this will serve as a good baseline for what will be developed in this work.

However, this plugin is not without its problems. First of all, it is an interface that is used specifically for the hand-eye problem, therefore it only calibrates cameras and it can only do it one at a time. This would be fine for calibrating systems that only use one camera, but for calibrating complex robotic systems with several cameras, having to repeat the calibration procedure for each camera could become a prolonged process. Therefore, seeing that ATOM is targeting the calibration of robotic systems that have multiple sensors that are not exclusively cameras, but sensors of different modalities as well, such as LiDARs, this interface could not be used for the ATOM framework.

One other problem of this interface is that, in the first tab, it would be useful to load the target settings from a file, along with other configuration parameters that could be useful for the calibration. Although it is possible to save the rviz configuration file (.rviz) with the settings in all the tab widgets saved, if for some reason the panel is closed or if the user wants to set up a new rviz file, all the parameters that were already set would be lost. Ideally, all the configuration parameters necessary for the calibration (target settings, frames, etc) would be in a yaml file in the robot package that could be loaded to the plugin, with its content being able to be modified within the panel. That way, the parameters would not be dependent on the rviz configuration file.

Additionally, in the "Calibrate" tab, the entire process of collecting the dataset and performing the calibration was not as straightforward at first, as it seems that a lot is happening in this widget. It would make sense to split this tab in two, one for collecting the data and the other to actually perform the calibration.

This interface also presents the problem that, after the samples are taken, it is unclear how to actually calibrate the camera, since there is no "Calibrate" button or something of that nature. Through the tutorial, it becomes clear that the final button is the "Save Camera Pose", but neither its position, name or size imply that. Lastly, one other problem with this tab is that the user only as the option to clear all samples. This could be very inconvenient if several samples had already been taken and the user wanted to delete just one of them, so it would make sense to add one extra button to do that.

Having gone through the primary works that have been developed when it comes to ROS GUI applications, it is time to go through the current ATOM calibration procedure, that will be described in the following section, in order to better understand its current state and give an idea of the functionalities that an interface created specifically for this framework could have.

2.3 ATOM Calibration Framework

As it was already mentioned before, the ATOM (Atomic Transformation Optimization Method) is a framework that is integrated with ROS and it provides the necessary tools for the calibration of complex robotic systems. All the details for setting up the environment to use ATOM are carefully explained in its package[13].

Firstly, in order to be able to perform the calibration with this set of tools, the users need to define their robotic system (e.g. <your_robot>) and have a system description in the form of an URDF file that is usually stored in a ROS package named <your_robot>_description.

The URDF[14] (Unified Robot Description Format) file is a specific format that is used in ROS and it allows you to describe all of the robot's physical properties in an XML (eXtensible Markup Language) file. An XML language that is very well known and commonly used in ROS is the *xacro*[15], which stands for *XML macro*, meaning that, with this language, it is possible to construct shorter and more readable XML files by using macros that expand into larger XML expressions.

A robot description in a URDF file uses several elements, with the robot element always being the root element of the file. Besides this element, the description usually consists of a set of links that are connected by a set of joints. There can also be other URDF elements (like transmission, gazebo, sensor, and so on) and each of these elements have their own attributes and elements to be specified. For instance, a joint element has a 'name' and a 'type' as its attributes and parent link, child link and origin as its elements, with the origin being the geometric transformation from the parent link to the child link [50].

In the listing 2.1 presented in the following page is a simple example of the general structure of a robot's description file.

[13]https://github.com/lardemua/atom

[14]http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch

[15]http://wiki.ros.org/xacro

```xml
<?xml version="1.0"?>
<robot name="any_robot_name">
  <link name="name_of_link1">
    <visual>
      <geometry>
        < ... />
      </geometry>
    </visual>
  </link>

  <link name="name_of_link2">
    <visual>
      <geometry>
        < ... />
      </geometry>
    </visual>
  </link>

  <joint name="name_of_link1_to_name_of_link2" type="...">
    <parent link="name_of_link1"/>
    <child link="name_of_link2"/>
    <origin rpy="..." xyz="..."/>
  </joint>

</robot>
```

Listing 2.1: General structure of a robot description .xacro file

After having the robot description, ATOM also requires a bagfile with a recording of the data from the sensors that are going to be calibrated. When everything is set correctly, the user can then proceed to the beginning of the calibration procedure.

2.3.1 Parameters Configuration

Along with the above mentioned URDF files that are required for the description of the robotic system, ATOM's approach aims to extend these files in order to also provide the information that is necessary for the correct configuration of the calibration to be carried out.

All this information is defined in a calibration configuration file saved as a YAML[16] file (config.yml). This file should be saved within a 'calibration' folder in a ROS package named <your_robot>_calibration and it contains all of the parameters needed for the calibration, namely: the description file of the robot, the bagfile to extract the necessary data for the calibration, the frame of reference for the optimization process, a discrimination of the sensors that will be part of the calibrations, the properties of the pattern used in the calibration, the possibility to anchor one of the sensors and the maximum time between sensor data messages when creating a collection. All this will create a set of files for launching the system, configuring rviz, etc.

[16] http://yaml.org/spec/current.html#id2502311

However, ATOM currently does not have any interactive way of modifying all these parameters. Every time there is a need for a change in one of them, it has to be changed the old fashioned way, by actually going into the directory of the file and changing them, which could be a tedious process.

This leads to what would ideally be the first phase of the interface to be created: a section of the plugin dedicated to load the contents of this configuration file inside RViz and making the appropriate changes to it, with the possibility of saving them in that same yaml file.

2.3.2 Initial Estimate

Iterative optimization procedures usually suffer from a problem that is known by the name of local minima, a problem that occurs when the initial solution is far from the optimal parameter configuration, which may lead to failure in finding adequate parameter values [4].

According to the ATOM's proposed approach for this problem, to make sure that the optimization will converge into the optimal solution, the setup of a plausible initial guess for the entire parametric optimization system is essential. If the first estimate of the parameters is near the optimal solution, then, intuitively, it is less likely to run into the local minima issue. Thus, the goal of this step is to ensure a reliable first sensors' pose configuration.

ATOM provides an interactive way of setting the sensors' poses that will be considered in the first iteration of the optimization procedure. By parsing the robot description files, it is possible to know the number and location of the sensors in consideration and associate an interactive marker in RViz for each sensor. Given that each interactive marker has the exact same pose as the associated sensor, the user could freely translate and rotate the marker using just the mouse cursor, allowing a close to real time visual feedback provided by the observation of the bodies of the robot model. The way to save a sensor's pose is by right-clicking on the respective interactive marker and choose that option on the menu that appears, as seen in figure 2.13 below.

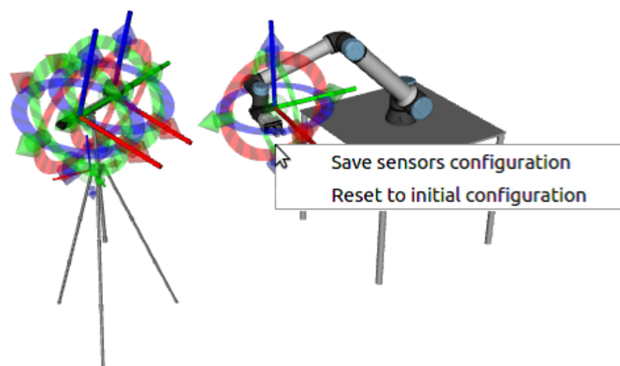


Figure 2.13: Menu when right-clicking on interactive marker.

Having gone through the details of the second step of this calibration procedure, a second phase for the interface is in place.

In this new tab, the interface should be able to showcase a list of all of the robotic system's sensors. Additionally, as it was mentioned before, each sensor can be moved by dragging their respective interactive marker, which is a very helpful tool, but using the mouse cursor to drag the interactive marker a lot of times would only get the sensors to its general pose, therefore, having a way to perform micro movements of each interactive markers' xyz and rpy (position and orientation) would also be a good functionality to have in the panel to help improve the accuracy of the estimation. Also, the way of saving and resetting a sensor's pose is not very intuitive nor practical and, for that reason, the interface should also allow the user to save a sensor's pose, or reset it to the previously saved one.

Additionally, other functionalities could be added, if seen fit, upon the implementation of the interface, for instance, the possibility of resetting all sensors to their respective previous poses all at once, instead of having to do it individually, and the ability to change the visibility and scale of each sensor's interactive marker, both of which are functionalities that could be quite useful in cases where the robotic systems have a large number of sensors.

2.3.3 Data Collection

To run a system calibration, it is necessary to collect information about the sensor data at different time instants. However, this data needs to be labeled first. The labeling of data is the annotation of the portions of data that views the calibration pattern. A common calibration pattern used for labeling data is the chessboard pattern, particularly for RGB and RGB-D cameras, and they can be easily labeled using any of the available image-based chessboard detectors. As for the data of a 2D LiDAR, it is not possible to robustly detect the chessboard because of the multiple planes in the scene derived from other structures. ATOM presents a solution to this, by using once again an interactive approach with the rviz interactive markers, where the user drags the marker to indicate where in the data the chessboard is observed [5].

After labeling the sensors' data, providing the information about which measurements are concerned with the chessboard detection, all the data must now be gathered and saved in an accessible format, in order to be used afterwards in the optimization procedure. To do this, temporal synchronization is required. Differently from other approaches, that need hardware synchronization to operate, ATOM solves this by collecting data (and the respective label) at moments defined by the user in which the scene has remained static for a certain period of time. In static scenes, the problem of data desynchronization is not observable, warranting the assumption that for each captured collection the sensor data is synchronized. These snapshot recordings of multi-sensor data are referred to as data collections. [5].

All this information, necessary for the optimization procedure, is then saved in a JSON file that will be accessed by the optimizer. This file contains important sensor

information, such as the sensor transformation chain and specific information about each collection, i.e., sensor data, partial transformations and data labels. It is important to note that, in order to ensure an accurate pose estimation, the calibration pattern should be detected in as many distinct positions and orientations as possible, so that the calibration becomes more reliable. As such, the pattern should be moved around and the user should save several collections. This is a concern that most calibration procedures take into account.

The third widget of the interface should concern the calibration procedure explained in this section. First, there should be a list where the important information present in the dictionary from the json file is showcased in the panel. Also, similar to the initial estimate widget, it would be helpful to have a way to perform micro movements of the interactive marker's pose in the panel. Plus, right now it is only possible to save a collection in a similar way to what was done in the initial estimate, by right clicking on the interactive marker and clicking on the save option, which as it was said for the previous case, it is not very intuitive. That would be the one other functionality to add to the plugin, along with the option of deleting a collection, something that is not possible to do, as of now.

2.3.4 Optimization Procedure

Finally, in the last phase, a system calibration is called through, where the goal is to provide the user with some visual feedback to give an insight into the calibration procedure as it is progressing. Seeing that most calibrations usually just print some information on the screen during the procedure, ATOM approaches this limitation by increasing both the quantity and the quality of the information provided to the framework. The data from all collections is published simultaneously, as if those time instants were collected and processed all at the same time. This makes it possible to visualize in RViz images with the reprojection, displaying the robot meshes, the position of the reference frames, etc. [3].

In the fourth and last widget, apart from a button to start the calibration, there are not many functionalities left to add, as this is the phase of the calibration procedure where everything done thus far is processed to perform the calibration. Something that the interface can indeed offer, is the visual part of the results. Two visual feedback items to be added are the plots of the value of the optimization residuals and the total error versus the number of iterations, both of which would be shown in the panel in a dynamic way, i.e., constantly updating during the procedure. Further functionalities could then be added upon the implementation of the interface, if deemed necessary.

2.4 Summary

Throughout this chapter, it was presented some work on GUIs used in the field of robotics in general, followed by a closer look into the main existing graphical interfaces for calibration systems, becoming clear that the HandEye Calibration plugin from *MoveIt!* is the most evolved panel so far and its step by step procedure gave some ideas of what to do and what could be done better in the interface developed in this dissertation. Additionally, it was given an explanation of the current calibration procedure of the ATOM framework, giving a general idea of the functionalities that could be helpful to implement for each of the four phases of the calibration procedure.

This chapter has hopefully helped justify the need for an interface, given that, as it was seen, ATOM is lacking one and none of the presented interfaces offered all the desired tools and functionalities for a graphical interface of this sort.

Chapter 3

Software Infrastructure

Robot Operating System (ROS) and Qt are two powerful tools used in the field of robotics and Graphical User Interface (GUI) development. When combined, these tools provide the possibility of creating a simple and easy to use GUI that would allow its users to interact with RViz in a more dynamic way, through graphical icons and visual indicators.

Before going into the approach on how to solve the problem at hands, it is important to recognize and give some insight on these tools that were used to build the GUI that was implemented in this dissertation.

3.1 ROS - Robot Operating System

ROS is an open source framework that is widely used in robotics and the philosophy behind it is to have a piece of software that enables the development of collaborative software for robots so that even the largest and most complex robots can be easily manipulated. What we get with this idea is to create functionalities that can easily be shared and used in other robots in such a way that we do not have to reinvent the wheel [51].

The GUI implemented in this dissertation aims to create a visual component of the ATOM's calibration procedure for RViz, which is a 3D visualizer tool of the ROS framework that will be explained in more detail below, in Section 3.2. But first, it is important to give a brief explanation on some other important ROS concepts.

3.1.1 ROS packages

The basic building blocks of the ROS software framework are ROS packages. This packages contain a file describing the package and stating any dependencies and can also contain various types of images, data, configuration files, programs (written in any of the compatible programming languages, like python or C++, for example) and so on [52].

3.1.2 ROS nodes, topics, messages and services

One of the primary purposes of ROS is to facilitate communication between the ROS modules, called nodes. ROS nodes are processes that perform computation. They are independent modules that can interact with other nodes in the system using the ROS communication capabilities [53]. Nodes can independently execute code to perform their

task, but they can also communicate with other nodes by sending or receiving messages under specific topics.

A node that is interested in a certain kind of data will subscribe to the appropriate topic. ROS topics are named buses over which nodes exchange messages[1]. It is possible for a node to have multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence.

ROS messages are strictly typed data structures that can be listened to by ROS nodes by subscribing to a ROS topic or they could be sent by publishing to a ROS topic. These messages can support standard primitive types (integer, floating point, boolean, etc), arrays of primitive types and constants [54].

Lastly, nodes also have the capability to provide a service[2]. Even though the publish/subscribe model is a very flexible communication paradigm, its one-way transport is not appropriate for RPC request/reply interactions. Those type of interactions can be dealt with using ROS services, being that they provide a two-way communication between the nodes with two messages: one for the request and the other for the reply.

3.1.3 ROS Master

The communication between the ROS nodes is established by the ROS Master[3]. The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. Its role is to enable individual ROS nodes to locate one another, in order for them to communicate with each other peer-to-peer.

The Master also provides the Parameter Server.

3.1.4 ROS Parameter Server

When a ROS Master is created, it creates a ROS parameter server, which is essentially a dictionary containing global variables that are accessible from anywhere in the current ROS environment. Those global variables are called ROS Parameters.

ROS parameters are named using the normal ROS naming[4] convention, which means that they have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to prevent the collision of parameter names. The hierarchical scheme also allows parameters to be accessed individually or as a tree[5].

3.1.5 ROS launch files

ROS launch files[6] are ROS tools that help launching multiple ROS nodes in addition to setting parameters on the ROS Parameter Server using just one file. These files are written in XML and usually end in a .launch extension.

[1] `http://wiki.ros.org/Topics`
[2] `http://wiki.ros.org/Services`
[3] `http://wiki.ros.org/Master`
[4] `http://wiki.ros.org/Names`
[5] `http://wiki.ros.org/Parameter%20Server`
[6] `http://wiki.ros.org/roslaunch`

3.1.6 ROS bag files

A ros bag file is a file format in ROS for storing ROS message data. These bags are usually created by subscribing to one or more ROS topics and storing the received message data in an efficient file structure. Afterwards, this .bag extension files can be played back with the information within them being published in the same ROS topics or even in remapped ones.

3.1.7 ROS introspection tools

One of the strongest features that ROS has is its powerful development toolset, that supports introspecting, debugging, plotting, and visualizing the state of the system being developed. The underlying publish/subscribe mechanism allows a spontaneously introspection of the data flowing through the ROS system, making it easy to comprehend and debug issues as they occur. The ROS tools take advantage of this introspection capability through an extensive collection of graphical and command line utilities that simplify development and debugging[7].

Most of these tools are command-line tools, where there is no use of a GUI, making all core functionality and introspection tools accessible via the terminal. There are over 45 command-line tools that can be used for a variety of actions, from launching groups of nodes, introspecting topics, services and actions, recording and playing back data, amongst many others[8]. As for tools that have a graphical component, ROS also has rqt[9], which is a Qt-based framework of ROS that implements various GUI tools in the form of plugins and RViz[10], which is perhaps the most well-known tool in ROS and it provides three-dimensional visualization of many sensor data types and any URDF-described robot. The latter will be discussed in more detail in the next section, as it will be the focus of this work, while a representation of the former can be seen in figure 3.1, with one of the many rqt plugins available, in this case the rqt_graph, where a simple node graph is displayed by the RQt screen.

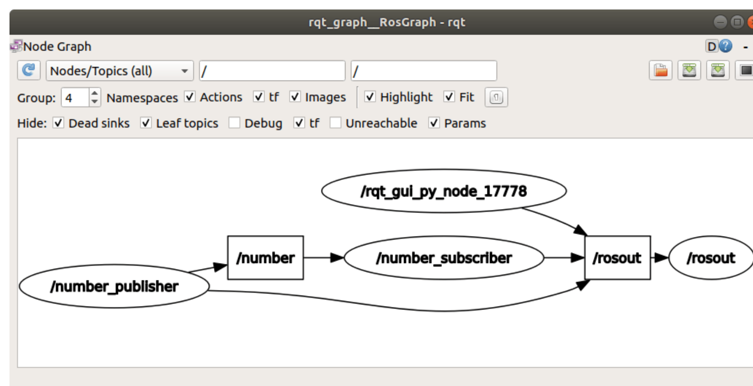


Figure 3.1: rqt screen showing a simple node graph: each rectangle represents a ROS topic and each ellipse a ROS node.

[7]https://www.ros.org/core-components/
[8]http://wiki.ros.org/ROS/CommandLineTools
[9]http://wiki.ros.org/rqt
[10]http://wiki.ros.org/rviz

3.2 RViz

RViz, abbreviation for ROS Visualization, is a powerful 3D visualization tool for ROS that allows the user to view the simulated robot model, log sensor information from the robot's sensors and replay the logged sensor information. By visualizing what the robot is seeing, thinking and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions. This way, it provides much needed insight into an application's state and view of the world [55].

The screenshot below (Figure 3.2) shows the RViz default window, when it is first opened, with the sections of it that were considered relevant properly labeled.

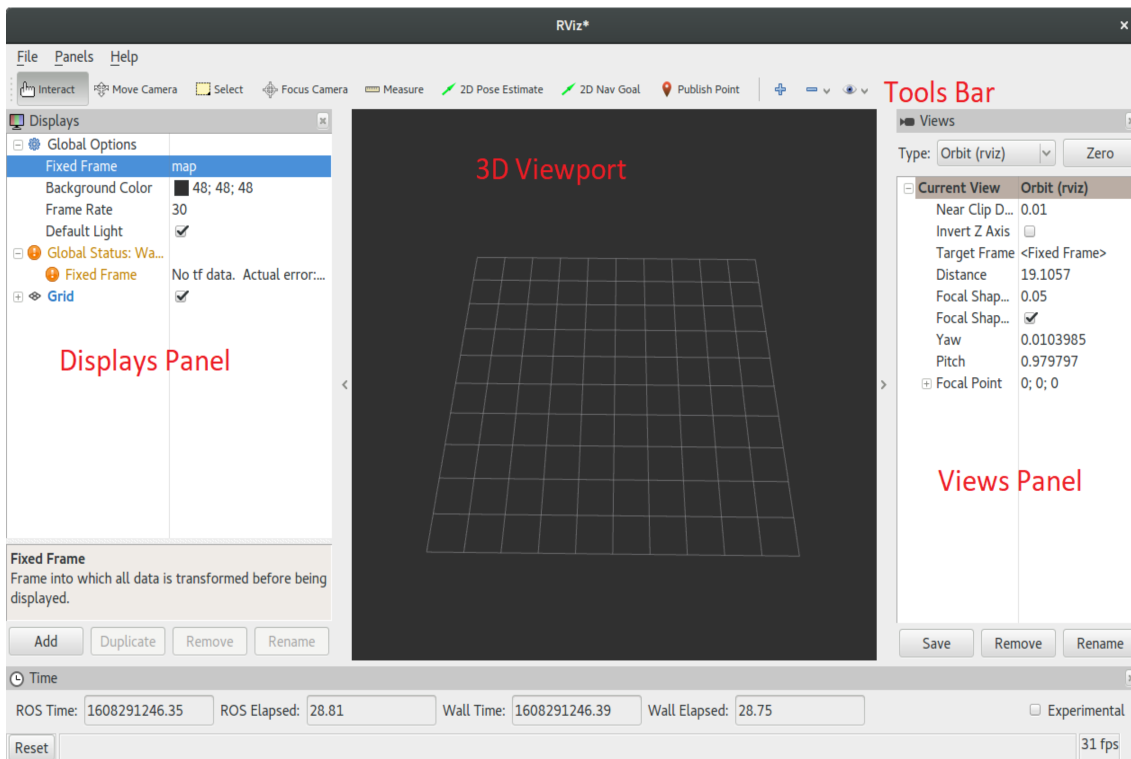


Figure 3.2: Default RViz window.

At this point, it is only visible a "Displays" panel on the left that includes the "Global Options," "Global Status," and a "Grid" display. From these, the key parameter is the fixed frame of the "Global Options", which indicates the name of the frame used as reference for all the other frames and any frame available in the combo box can be selected.

However, several other elements could be added to this window, by clicking on the "Add" button on the bottom left of the "Displays" panel and choosing which visualization to add. In this section, some of the visualizations that were found to be the most relevant during this work will be looked at.

3.2.1 Robot Model

The Robot Model is a display type provided by RViz that shows the links of a robot (defined by a urdf file) in their correct poses according to the tf transform tree. When added, the robot might not be visible at the beginning, because the user first needs to tell rviz which fixed frame to use.

The key parameters of this display are the "Visual Enabled", to enable/disable the 3D visualization of the model and the "Robot Description", which is for the topic on which the robot description is published. By expanding the "Links" parameter, it is possible to see the whole model tree, with all the joints and the links available and the relative position and orientation in the space relative to the fixed frame[11]. Figure 3.3 shows this display type added to RViz.

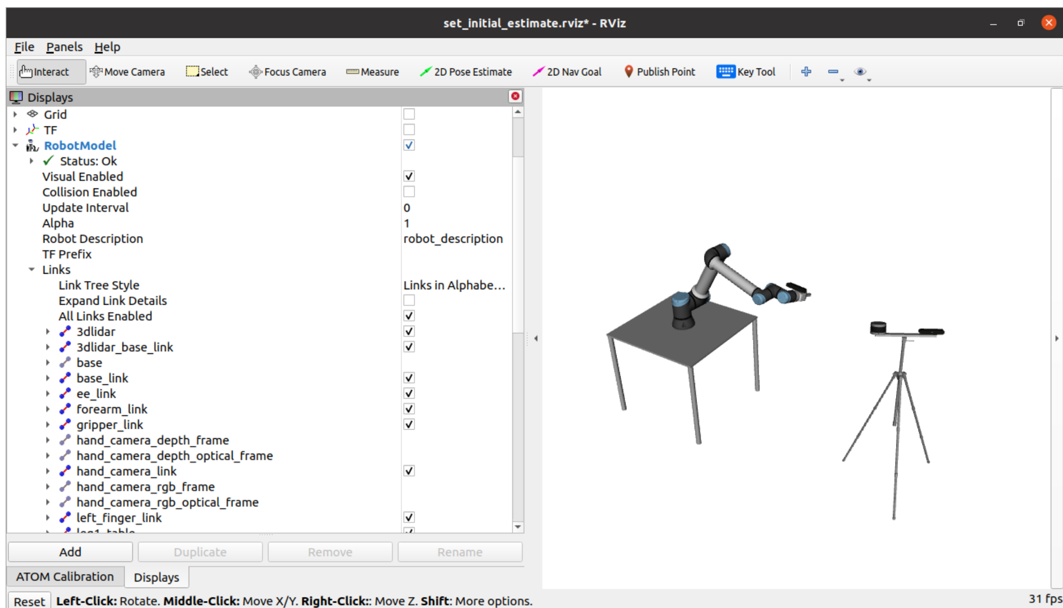


Figure 3.3: Robot Model Display Type in RViz.

3.2.2 Tf

The TF display (Figure 3.4) allows the visualization of the position and the orientation of all the frames that compose the TF Hierarchy. There are three optional pieces of data to display: the frame name, the frame axes, and an arrow from the frame to its parent. If the axes are displayed, the X axis is indicated in red, the Y axis is indicated in green, and the Z axis is indicated in blue[12]. Critical to using this display is the ability to enable/disable the visualization of individual frames. This allows the users to concentrate only on the parts that are most important for their current task.

[11]https://www.stereolabs.com/docs/ros/rviz/
[12]http://wiki.ros.org/rviz/DisplayTypes/TF

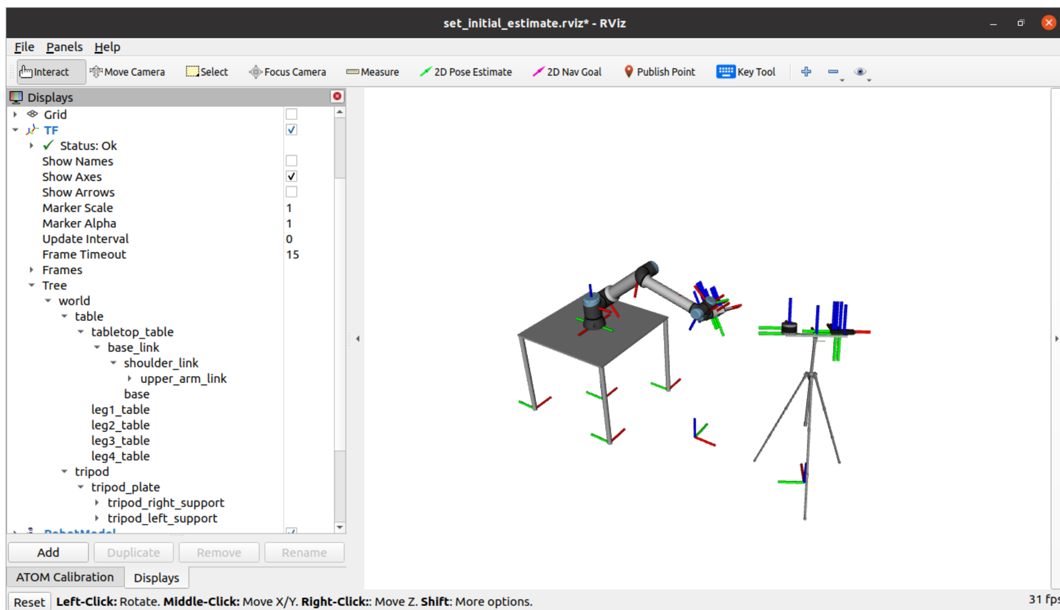


Figure 3.4: Tf Display Type in RViz.

3.2.3 Point Cloud

A point cloud is a set of data points in 3D space. Such data is usually derived from time-of-flight, structured light or stereo reconstruction. This display type can be represented in RViz using either the sensor_msgs/PointCloud2 or pcl::Pointcloud data type. An example of this is shown in Figure 3.5.

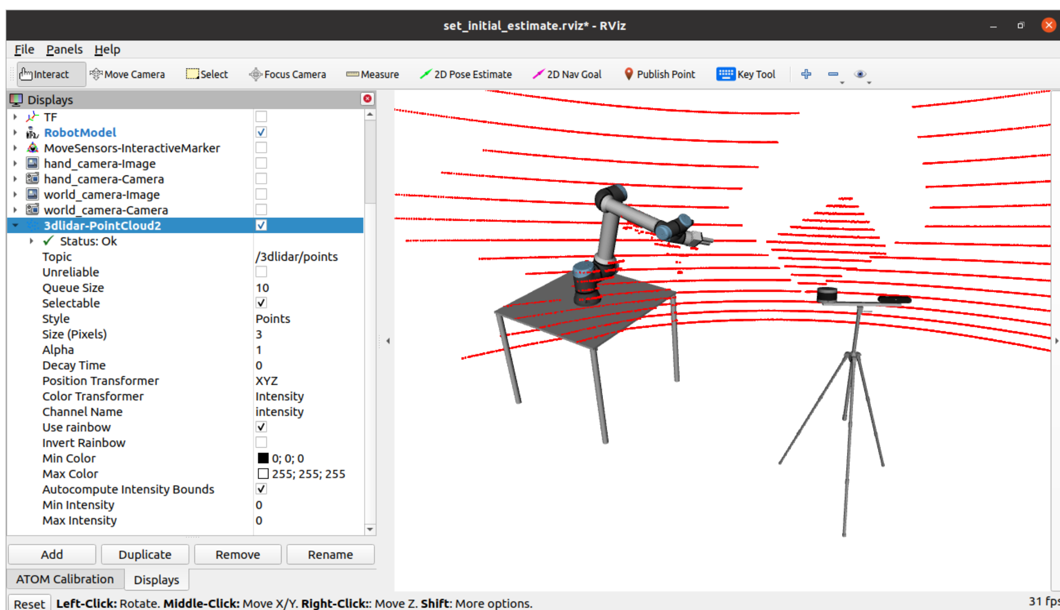


Figure 3.5: PointCloud Display Type in RViz.

3.2.4 Image

The Image display type (Figure 3.6) creates a new rendering window for the camera image. To be able to see the image, the topic sensor_msgs/Image needs to be subscribed to (bottom left shows the image from the camera).

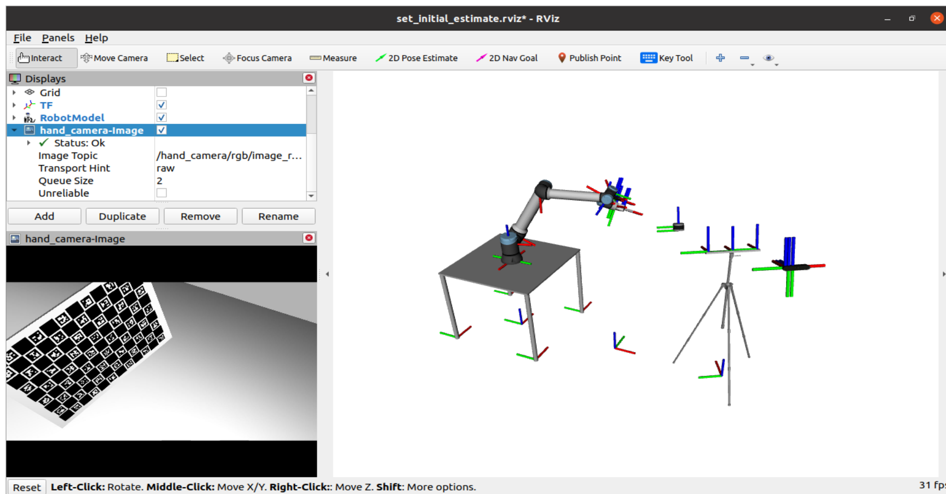


Figure 3.6: Image Display Type in RViz.

3.2.5 Camera

The Camera display creates a new rendering window from the perspective of a camera, and overlays the image from the camera on top of it. For this display to work properly the sensor_msgs/Image topic subscribed to must be part of a camera, and must have a sensor_msgs/CameraInfo topic named camera_info alongside it[13]. Figure 3.7 exhibits an example of this camera display being used in RViz.

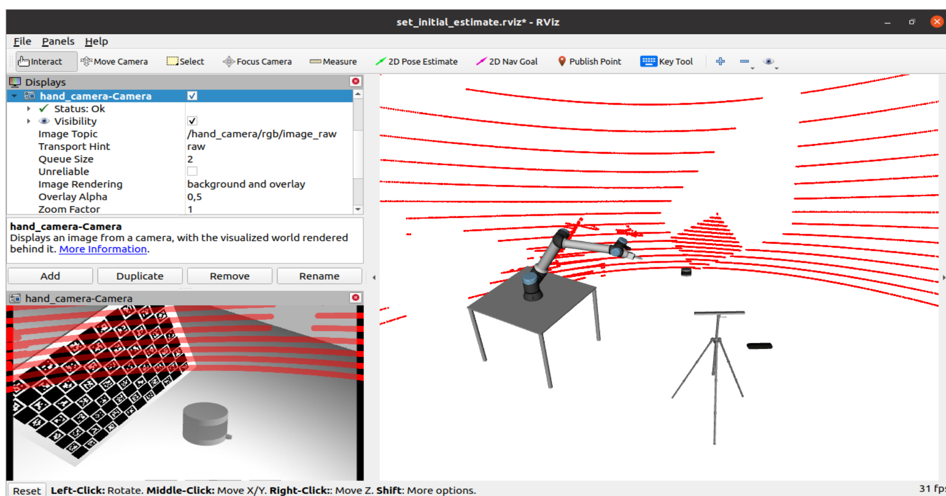


Figure 3.7: Camera Display Type in RViz.

[13] http://wiki.ros.org/rviz/DisplayTypes/Camera

3.2.6 Markers

There are many ROS applications that rely on *Markers*[14] for visualization. They allow programmatic addition to the 3D view at specified location and can contain a single primitive shape such as a sphere, a box, or an arrow, a list of points or triangles, or can point to a 3-D model that is stored on disk. This way, they are able to provide some insight into a robotic application's state and view of the world. Figure 3.8 shows some of the many markers that can be displayed.

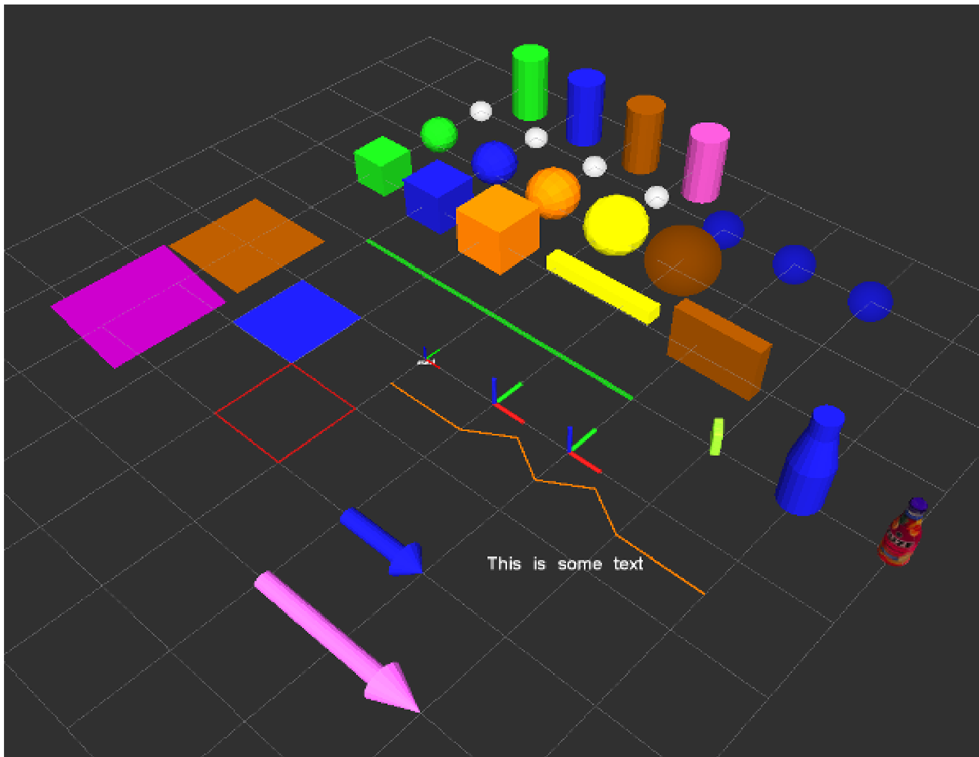


Figure 3.8: Markers displayed on RViz[15]

3.2.7 Interactive Markers

As said above, the Marker Display can be a very useful visualization tool to use in rviz. However, if instead of simply visualizing, the users want to control their applications using a 3-D interface or communicate with their robot and become an active participant rather than just an observer, *Markers* are not able provide that functionality. That is exactly what Interactive Markers are used for [56].

Interactive markers (Figure 3.9) work in a similar way to the markers, as in they can also be represented with primitive shapes (arrows, boxes, spheres and lines) that are displayed in RViz and can be used to represent a given item or location to serve the purpose of visualization, having, however, the ability to be interacted with as well[16]. The

[14] `http://wiki.ros.org/rviz/DisplayTypes/Marker`
[15] `https://github.com/DavidB-CMU/rviz_tools_py`
[16] `http://wiki.ros.org/rviz/Tutorials/Interactive%20Markers%3A%20Getting%20Started`

way this interaction works is by the user using the mouse cursor to drag them or right clicking on them, selecting something from a context menu assigned to each marker. ROS nodes can then respond to the actions of the user.

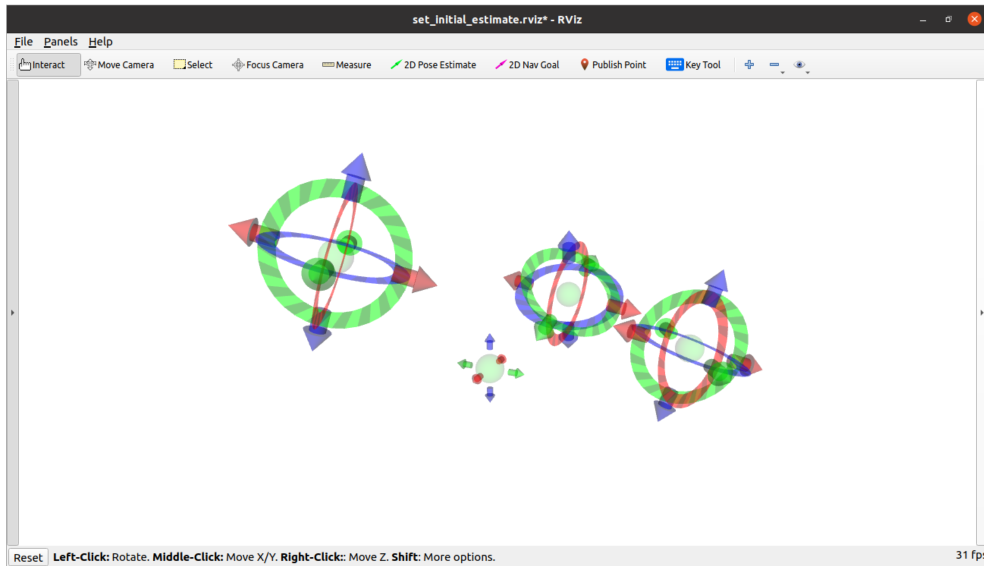


Figure 3.9: 4 Interactive Markers displayed on RViz.

In order to create a node providing a set of interactive markers, the user needs to instantiate an InteractiveMarkerServer object. This will handle the connection to the client (that is usually RViz) and make sure that all changes made are being transmitted and that the application is being notified of all the actions the user performs on the interactive markers[17]. Below, Figure 3.10 shows a simple scheme of the explained interactive network of this tool.

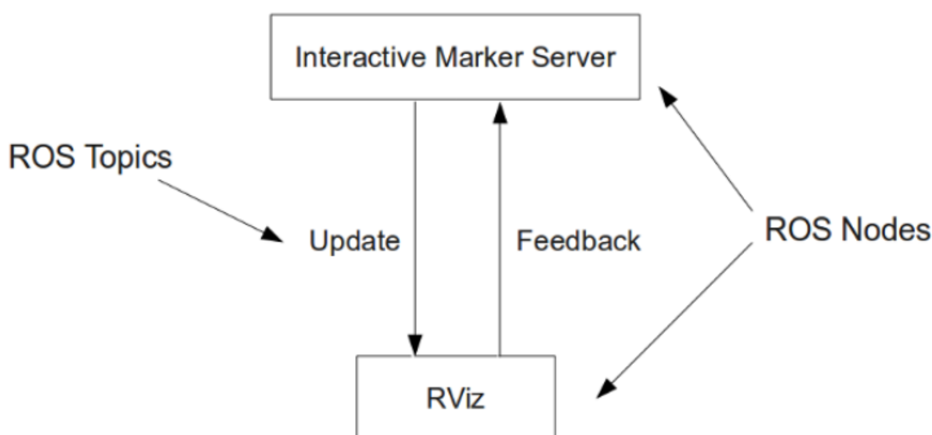


Figure 3.10: Communication of interactive markers with RViz.

[17] https://zongweizhou1.github.io/2019/06/26/rviz-interactive-markers/

3.2.8 Custom Plugins

Even though RViz already has its own built-in displays, panels and tools, it is also possible to integrate a custom GUI as a plugin. A tool plugin is a class that determines how mouse events interact with the visualizer and they are added to the RViz's toolbar. As for the Display plugins, they are added to the Displays Panel and the main goal of this type of tools is to visualize different types of ROS messages, mainly sensor data in the RViz 3D viewport. These plugins can be added by clicking on the 'Add' button on the bottom left of the Displays Panel. Lastly, the panel plugin in RViz is a GUI widget which can be docked in the main window. It does not show properties in the "Displays" panel like a Display, but it could show other things in the 3D scene. A panel can be a useful place to put a bunch of application-specific GUI elements with different functionalities. These plugins can be added by clicking on 'Panels' on the menu bar, and selecting 'Add New Panel', choosing then the created plugin.

The latter will be the one developed in this dissertation with the help of a cross-platform GUI library called Qt, which will be presented in the following section (section 3.3).

In figure 3.11, it is possible to see a screenshot of the RViz environment with an example of a customized GUI plugin in it (in this case, it is the the Teleop Panel from the RViz plugin tutorials[18]).

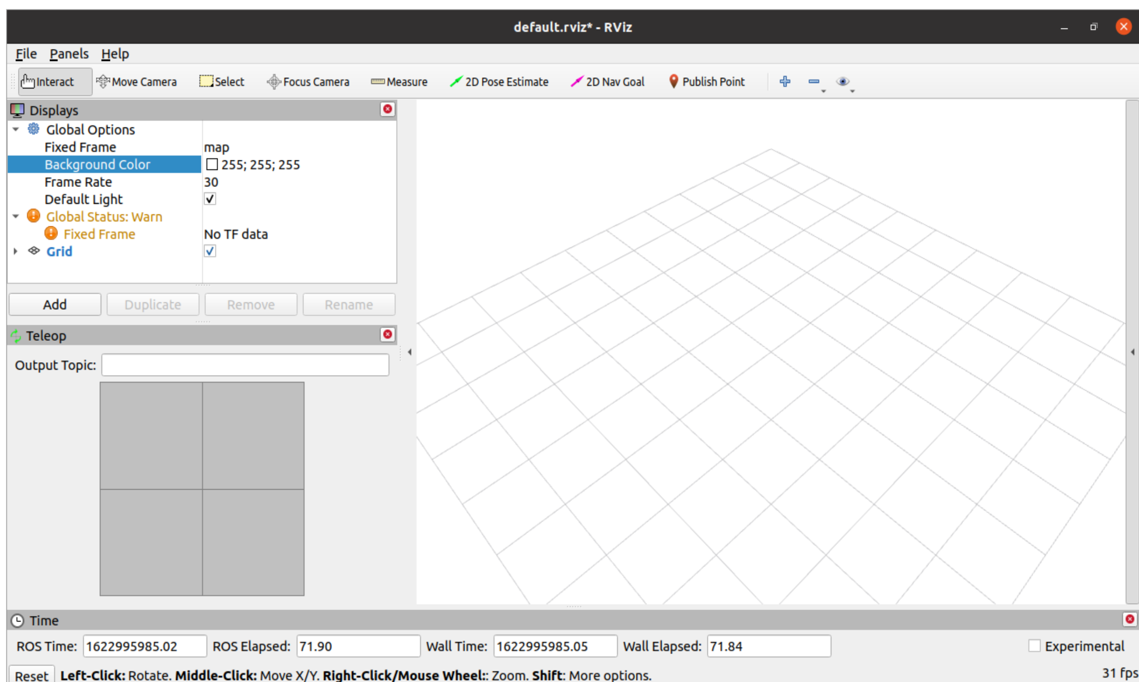


Figure 3.11: Screenshot of the RViz environment with the added plugin in it.

[18] http://docs.ros.org/en/kinetic/api/rviz_plugin_tutorials/html/panel_plugin_tutorial.html

3.3 Qt

A graphical user interface, or GUI, is a user interface that includes graphical elements such as widgets, icons and buttons that aims to facilitate the users' experience with whatever application they are using it in. In fact, there are multiple tools that can be used in Linux or Ubuntu to make a GUI, such as Qt, Gambas, GTK+, Perl and many others.

In order to save developers' time and energy and help them focus on the original technical topic, which is the functionalities of the panel itself, Qt highlights itself from the rest of the softwares listed above, for its use in a variety of platforms and its plenty of open-sourced resources.

For that reason, and being that Rviz is implemented using Qt[19], this was the tool used in this project to deal with all the visual parts of the panel, with the functionalities of every object added to this panel being given later, using C++, upon their integration with ROS.

Below, in figure 3.12, is a screenshot of the interface being designed in Qt Designer with each section properly labeled, to help give a better understanding of the environment of this software.

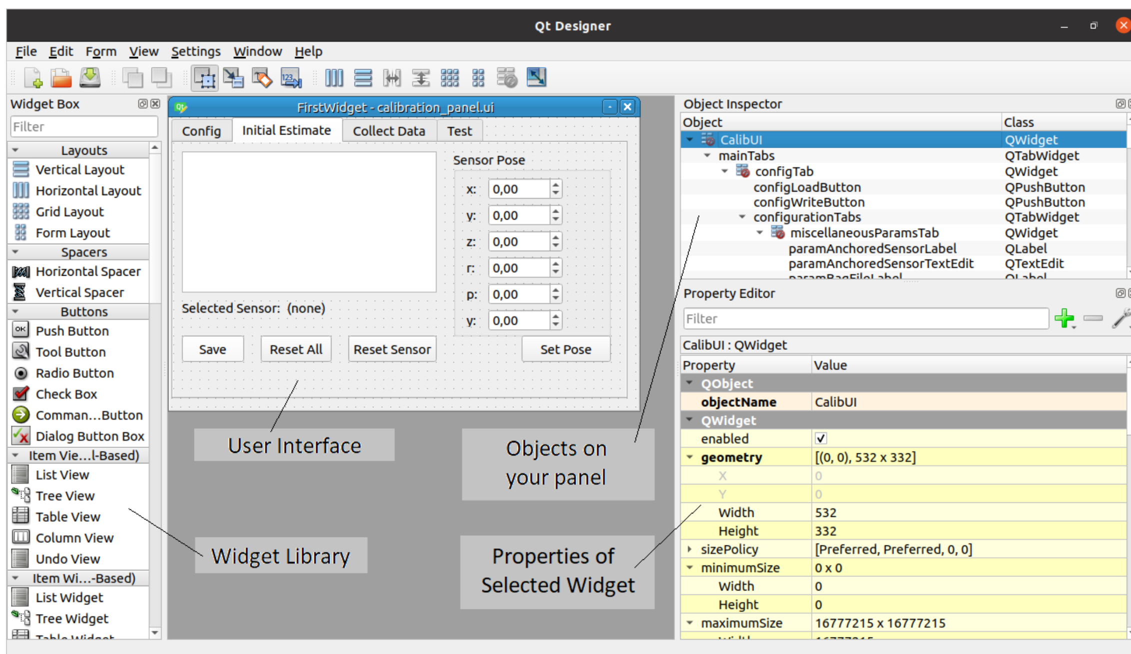


Figure 3.12: Screenshot of a panel being designed using Qt Designer.

This software does not need much explanation on how it works, since it can be used very intuitively. In short, the elements on the left (Widget Library), can be placed in the window (User Interface) and its names and properties can be changed in the bottom right panel.

[19] `https://doc.qt.io/`

Chapter 4

Approach

This chapter discusses the approach taken for the implementation of the interface developed in this dissertation. Initially, an explanation on the process of creating the plugin for RViz will be given. Following that is a step by step of the calibration procedure with the created GUI, explaining all the functionalities that were added to the panel for each phase. The entire work was done using C++ as its programming language.

4.1 RViz plugin implementation

Before starting to add the desired functionalities to the RViz plugin, a panel must be created and integrated with the ROS Visualization tool. There are a few ways to do so, the main one being the one described in the Rviz Plugin tutorials[1], which consists on implementing the interface with Qt, using only code. The advantage of this method of implementation is the amount of information available on how to write panel layouts in the source code. However, seeing that the interface that is being developed will have several tabs with a variety of items in each of them, having to create the entire interface from a programming file would be a long and complex process that would just add more code to a project that is already expected to have a large amount of code for the panel's functionalities.

Therefore, the focus was on creating a panel design directly from the Qt Designer and making it an Rviz Plugin. This approach would give more visual feedback during the process of creating the interface, with a much less abundance of code. This is not a commonly used method, although it was the method used by *MoveIt!* for the *Motion Planning* plugin, as it was mentioned in the beginning of Chapter 2, which served as a guideline for the current approach.

Initially, it was implemented to RViz just a simple panel with only one button that prints some information to the terminal. After that is done and the panel could already be added to RViz, some more items and functionalities could start being added to achieve the desired goals.

[1] `http://docs.ros.org/en/kinetic/api/rviz_plugin_tutorials/html/panel_plugin_tutorial.html`

The first step was to create the .ui file with Qt Designer, a tool provided by QT Creator (Qt's IDE). This file describes the design of the panel and the use of this tool is fairly simple. In short, the user can intuitively create the layout of the panel by just placing the elements in the window. Figure 4.1 shows a button being added to Qt.

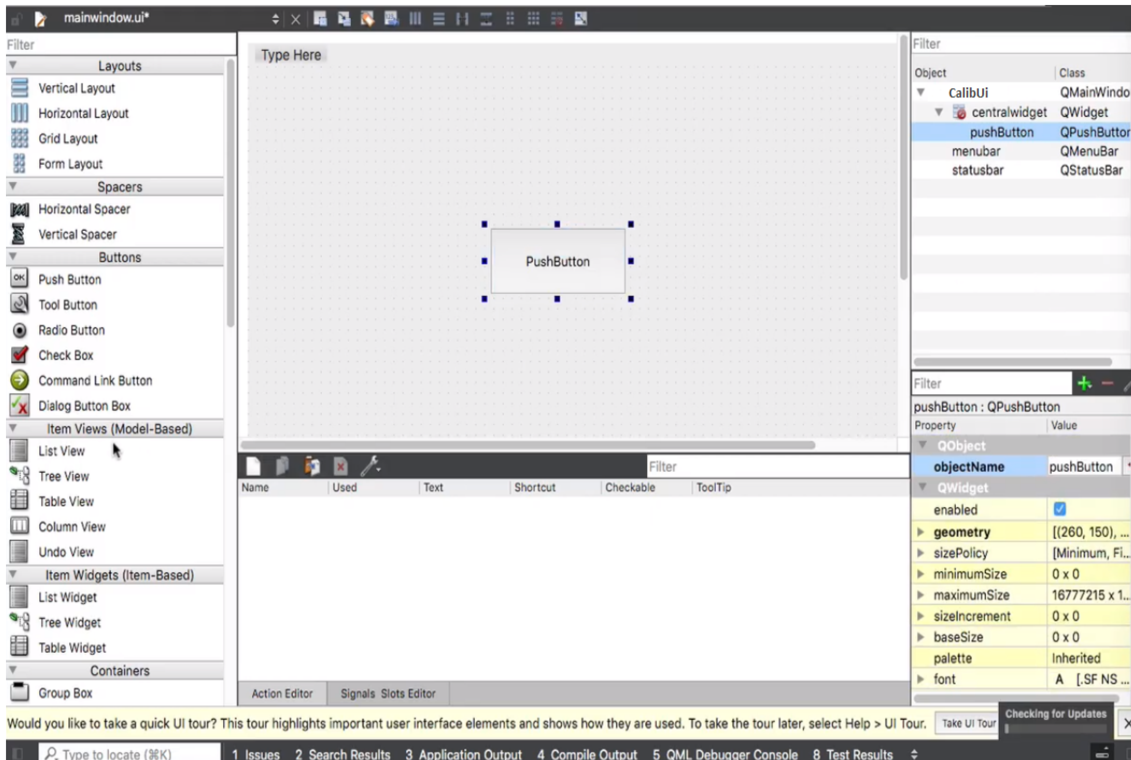


Figure 4.1: Screenshot of a simple button being added to Qt Designer.

The .ui file automatically creates a class and header file at compile time. This class then needs to be included in the header file, with the namespace being fixed at Ui, while the class can have any name, as long as it matches the QWidget name given at the .ui file. Still in the header file, under a different namespace, the rviz::Panel[2] class needs to be declared and then it is done the standard procedure in a C++ header file (constructor, deconstructor, and so on), followed by the pointer declaration of Ui::MainWindow, that is secured by the constructor of the class and will be used later throughout the source files to be able to recognize the QObjects added to the panel.

After that, there are still few steps that need to be followed. In short, what still needs to be done is: properly setting up the CMakeLists.txt file, which is the file that contains a set of directives and instructions describing the project's source files and targets (executables, libraries, and so on); write the main source file, where the GUI is set up, and we can connect the signals from actions performed on the objects from

[2]http://docs.ros.org/en/indigo/api/rviz/html/c++/classrviz_1_1Panel.html

the panel to the slots, which are the callback functions for each action[3]; create the plugin_description.xml[4], which is an XML file that serves to store all the important information about a plugin in a machine readable format (contains information about the library the plugin is in, the name of the plugin, the type of the plugin, the icon of the interface, etc); export the created description file of the plugin in the package.xml file of the ROS package (it is necessary to pass the setting value of to the plugin setting of Rviz).

This completes all the necessary file descriptions. The ROS package where all these steps were followed can then be rebuilt and, when running RViz, the new panel can now be added (Figure 4.2 shows this first panel implemented in RViz printing to the terminal when the button is clicked).

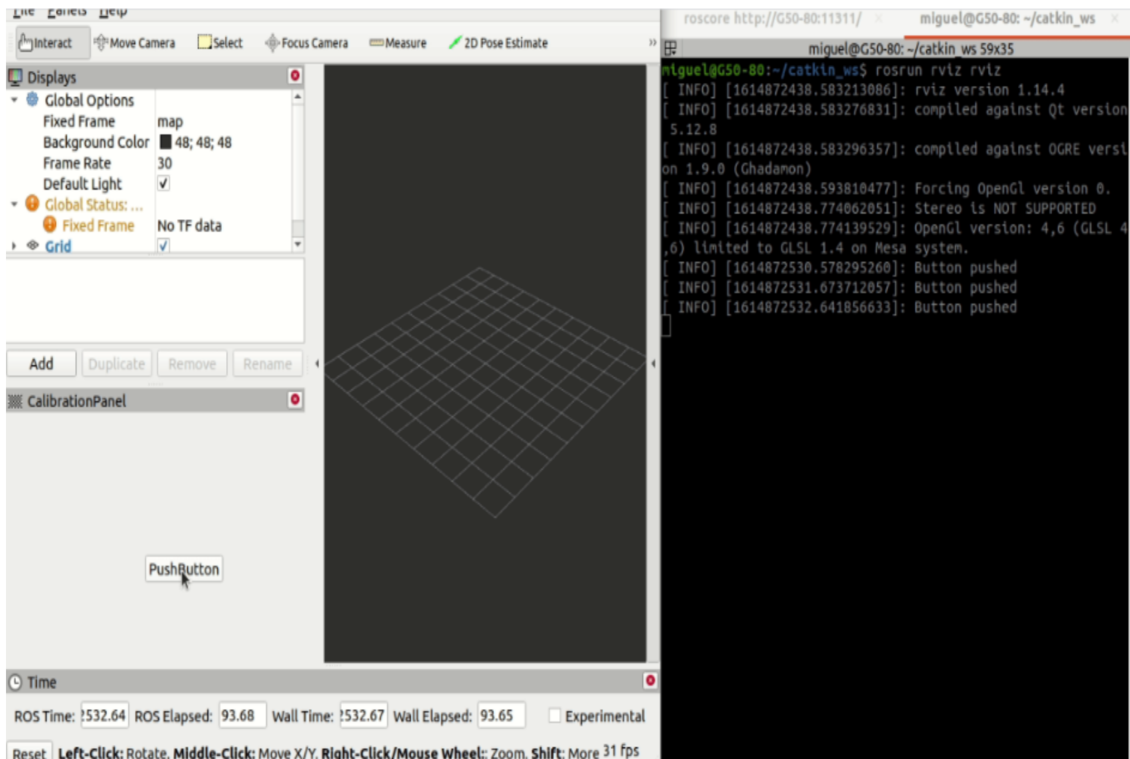


Figure 4.2: Simple Panel with just a button printing to the terminal.

Having achieved this preliminary step, with a fully functional panel already being able to be added to RViz as a plugin and an action occurring upon clicking the button, what remains to be done is start adding more items to the interface and their respective functionalities in order to meet the goals that were set for each phase of the calibration procedure.

[3] https://doc.qt.io/qt-5/signalsandslots.html
[4] http://wiki.ros.org/pluginlib

4.2 Calibration procedure

The ATOM calibration consists of four phases, all of which already described in Chapter 2. In this section, the approach taken for each of these phases will be presented.

However, to do so, a robotic system to perform the appropriate testing throughout this work is needed. That robotic system will be the mmtbot[5]. The Multi-Modal Test Bot (mmtbot) is a conceptual robot designed to test advanced calibration methodologies. This system contains the following sensors:

- hand_camera - An RGB-D camera mounted on the manipulator's end effector link
- 3dlidar - A 3D LiDAR mounted on a tripod;
- world_camera - An RGB-D camera mounted on the same tripod

This robotic system was designed with Gazebo[6]. Gazebo is an open-source 3D robotics simulator. Its objective is to simulate a robot, mimicking a robot's behavior in a real-world physical environment. It can compute the impact of forces (such as gravity) and provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can also model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle), Kinect style sensors, etc.

Figure 4.3 shows the mmtbot robotic system in the Gazebo environment with each sensor circled with a different colour: the hand_camera in red, the 3dlidar in yellow and the world_camera in blue. As for Figure 4.4, it shows the transformations tree of this robotic system.

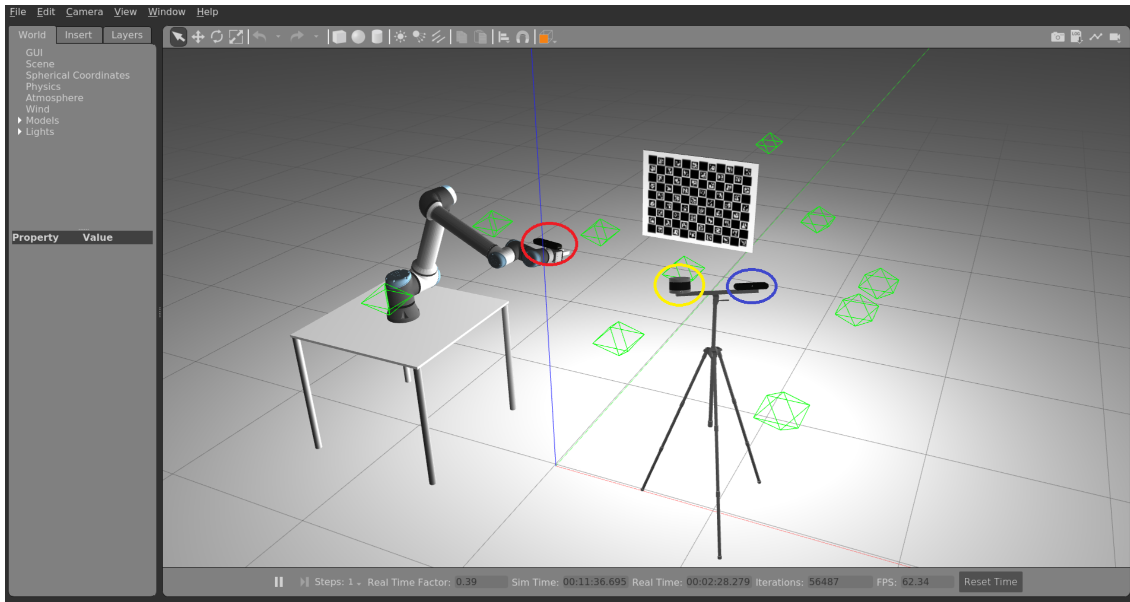


Figure 4.3: mmtbot in the Gazebo Environment.

[5] https://github.com/miguelriemoliveira/mmtbot
[6] http://gazebosim.org/

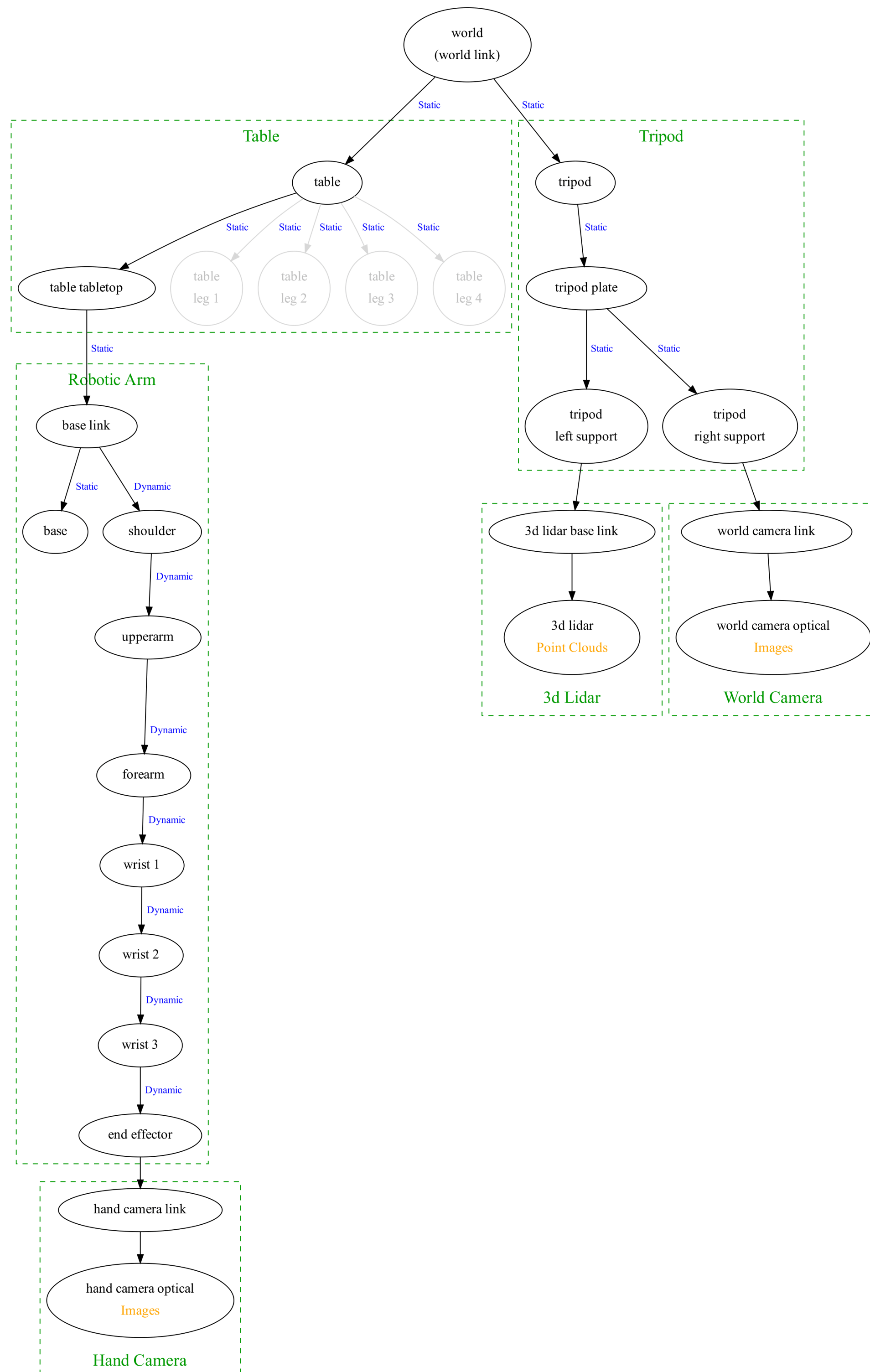


Figure 4.4: Transformations Tree (TF tree) of the mmtbot robotic system.

4.2.1 Parameters Configuration

As stated in Chapter 2, Section 2.3.1, all the information in regards to the configuration of the calibration is stored in config.yml, a yaml file that is under a 'calibration' folder inside a ROS Package named *mmtbot_calibration*, with *mmtbot* being the name of the robotic system used during this work, as stated in the previous section, and it should be changed to the name of the user's robotic system. In the listing 4.1 below, it is possible to see an example of this file for the robotic system used, along with a description of each of its parameters.

```
1 #  ATOM FRAMEWORK
2 #  https://github.com/lardemua/atom
3 # This yaml file describes your calibration!
4
5 # You can start by defining your robotic system.
6 # This is the URDF file (or xacro) that describes your robot.
7 # Every time a path to a file is requested you can use
8 #   - Absolute Path
9 #       Example 1: /home/user/ros_workspace/your_package/urdf/description.urdf.
    xacro
10 #       Example 2: file://home/user/ros_workspace/your_package/urdf/description
    .urdf.xacro
11 #
12 #   - Path Expansion
13 #       Example 1: ${HOME}/user/${YOUR_VARIABLE}/your_package/urdf/description.
    urdf.xacro
14 #       Example 2: ~/user/ros_workspace/your_package/urdf/description.urdf.
    xacro
15 #
16 #       NOTE: It is up to you to guarantee the environment variable exists.
17 #
18 #   - ROS Package Reference
19 #       Example: package://your_package/urdf/description.urdf.xacro
20 description_file: "package://mmtbot_description/urdf/larcc.urdf.xacro"
21
22 # The calibration framework requires a bagfile to extract the necessary data
    for the calibration.
23 bag_file: "$ROS_BAGS/mmtbot/11_03_2021.bag"
24
25 # You must define a frame of reference for the optimization process.
26 # It must exist in the transformation chains of all the sensors which are being
     calibrated.
27 world_link: "world"
28
29 # ATOM will calibrate the extrinsic parameters of your sensors.
30 # In this section you should discriminate the sensors that will be part of the
    calibrations.
31 sensors:
32   # Each key will define a sensor and its name, which will be use throughout
    the calibration.
33   # Each sensor definition must have the following properties:
34   #       link:
35   #           The frame of the sensor's data (i.e. the header.frame_id).
```

```
36   #
37   #       parent_link:
38   #           The parent link of the transformation (i.e. link) to be
      calibrated.
39   #
40   #       child_link:
41   #           This is the transformation (i.e. link) that we be optimized.
42   #
43   #       topic_name:
44   #           Name of the ROS topic that contains the data produced by this
      sensor.
45   #           If you are calibrating an camera, you should use the raw image
      produced by the
46   #           sensors. Aditionally, it the topic is an image it will
      automatically use the
47   #           respective `camera_info` topic.
48   # Example:
49   hand_camera:
50     link: "hand_camera_rgb_optical_frame"
51     parent_link: "ee_link"
52     child_link: "hand_camera_link"
53     topic_name: "/hand_camera/rgb/image_raw"
54
55   world_camera:
56     link: "world_camera_rgb_optical_frame"
57     parent_link: "tripod_right_support"
58     child_link: "world_camera_link"
59     topic_name: "/world_camera/rgb/image_raw"
60
61   3dlidar:
62     link: "3dlidar"
63     parent_link: "tripod_left_support"
64     child_link: "3dlidar_base_link"
65     topic_name: "/3dlidar/points"
66
67 # The calibration requires a detectable pattern.
68 # This section describes the properties of the calibration pattern used in th
      calibration.
69 calibration_pattern:
70
71   # The frame id (or link) of the pattern.
72   # This link/transformation will be optimized.
73   link: "pattern_link"
74
75   # The parent frame id (or link) of the pattern.
76   # For example, in hand-eye calibration the parent link
77   # of the pattern can be the end-effector or the base of the arm
78   parent_link: "world"
79
80   # Defines if the pattern link is the same in all collections (i.e. fixed=true
      ),
81   # or each collection will have its own estimative of the link transformation.
82   fixed: false
```

```
83
84  # The type of pattern used for the calibration.
85  # Supported pattern are:
86  # - chessboard
87  # - charuco
88  pattern_type: "charuco"
89
90  # If the pattern type is "charuco" you need to define
91  # the aruco dictionary used by the pattern.
92  # See https://docs.opencv.org/trunk/dc/df7/dictionary_8hpp.html
93 #    dictionary: "DICT_5X5"
94  dictionary: "DICT_5X5_100"
95
96  # Mesh file (collada.dae or stl) for showing pattern on rviz. URI or regular
     path.
97  # See: description_file
98 #    mesh_file: "package://atom_calibration/meshes/charuco_5x5.dae"
99  mesh_file: "package://mmtbot_gazebo/models/charuco_800x600/charuco_800x600.dae"
100
101  # The border width from the edge corner to the pattern physical edge.
102  # Used for 3D sensors and lidars.
103  # It can be a scalar (same border in x and y directions), or it can be {'x':
     ..., 'y': ,,,}
104  border_size: {'x': 0.04, 'y': 0.03}
105
106  # The number of corners the pattern has in the X and Y dimensions.
107  # Note: The charuco detector uses the number of squares per dimension in its
     detector.
108  # Internally we add a +1 to Y and X dimensions to account for that.
109  # Therfore, the number of corners should be used even for the charuco pattern
     .
110  dimension: {"x": 11, "y": 8}
111
112  # The length of the square edge.
113  size: 0.06
114
115  # The length of the charuco inner marker.
116  inner_size: 0.045
117
118 # Miscellaneous configuration
119
120 # If your calibration problem is not fully constrained you should anchored one
      of the sensors.
121 # This makes it immovable during the optimization.
122 # This is typically referred to as gauge freedom.
123 anchored_sensor: ""
124
125 # Max time delta (in milliseconds) between sensor data messages when creating a
      collection.
126 max_duration_between_msgs: 1000
```

Listing 4.1: config.yml - Yaml file to describe the calibration parameters of the mmtbot robotic system

As of now, any modifications required to any of these parameters need to be done by going directly to the file and modify them by hand. The file can then be loaded and the ROS Parameters are set, either by using the rosparam load command-line tool or by integrating it in the launchfile that launches the rviz configuration with the robot model and all other visual displays.

Seeing that this is not very practical, the first step was to create an interactive way to load this file, see all its parameters in the interface and update the file with any changes made within the interface.

The arrangement of the elements inserted in this tab was thought out to have two buttons at the bottom of the panel, one to load the configuration file, the other one to update the content of the file.

As for the parameters of the configuration file, they would appear in editable QLineEdit objects (text boxes from Qt), which would allow to both read and write in them. Since there are a lot of parameters in the file, it would take a lot of the interface's space to have them all in one tab. Therefore, these parameters were separated in three tabs: the first one for miscellaneous configurations (since the next two could be grouped better), the second was destined to the configuration of the calibration pattern, with the last one being for the sensors. The next three figures show the entire configuration tab layout developed (Figures 4.5, 4.6, 4.7).

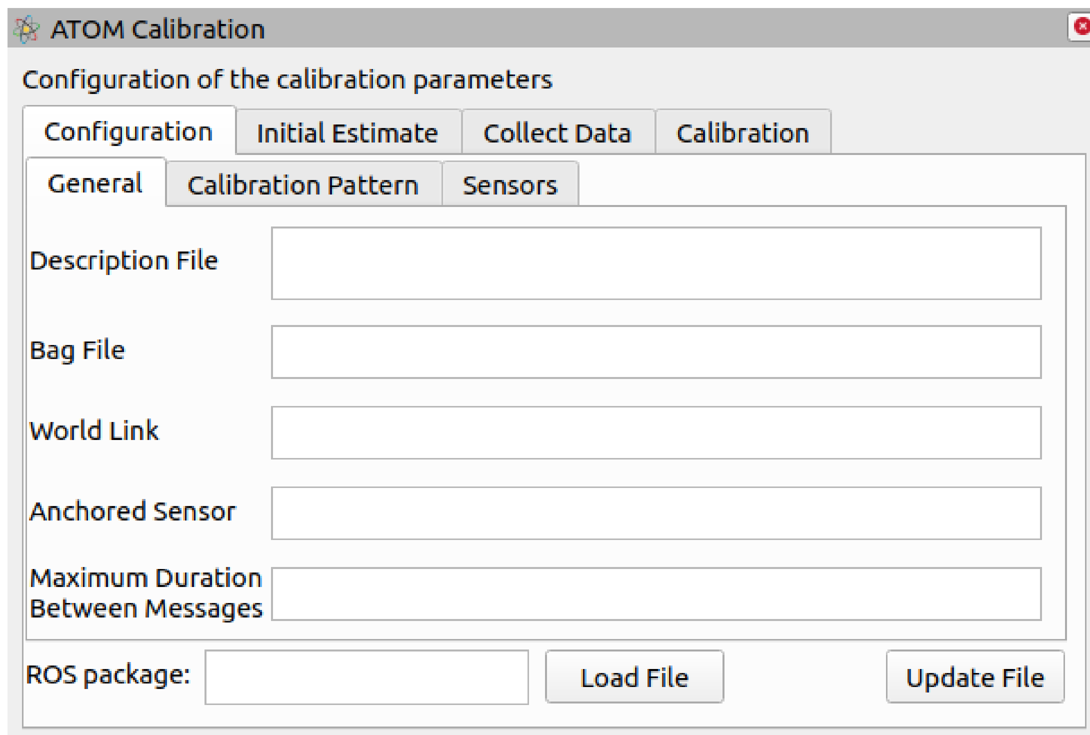


Figure 4.5: Layout of Configuration Tab Widget: First tab.

Figure 4.6: Layout of Configuration Tab Widget: Second tab.

Figure 4.7: Layout of Configuration Tab Widget: Third tab.

Having dealt with the visual part of the panel, some functionalities were then added to it. Initially, the configuration file must be loaded. For that, a line edit object was added next to the "Load File" button to allow its user to write the ROS package name of the robotic system currently being calibrated (Figure 4.8). The file is then located in the code, since the setup of the package requires this file to be in the "calibration" folder inside this package, making the directory easy to find within the code, using one of ROS tools (ros::package::getPath()[7]).

Figure 4.8: ROS package name inserted in the line edit: mmtbot_calibration for the case of this approach.

After pressing the button, if the package name inserted is not valid (either by it not existing or not containing the file in the right directory), a message shows up with the warning that no calibration file was found with the provided name. If, on the other hand, the ROS package name is valid, it means the configuration file was found and it can be parsed to the code to then set the line edits in the interface.

Figures 4.9, 4.10 and 4.11 show the interfaces after pressing the 'Load File' button, with all the parameters shown in listing 4.1 set in the appropriate line edit box.

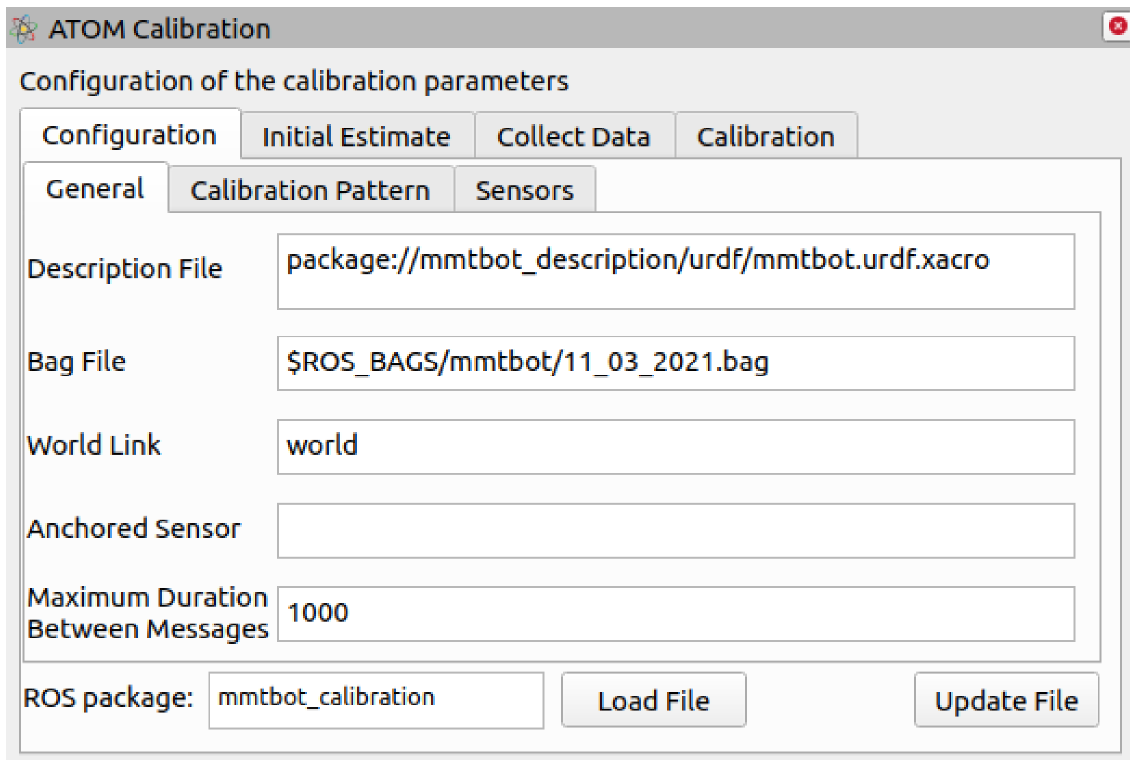


Figure 4.9: Parsing of file to the Configuration Tab Widget: First tab.

[7] https://wiki.ros.org/Packages#C.2B-.2B-

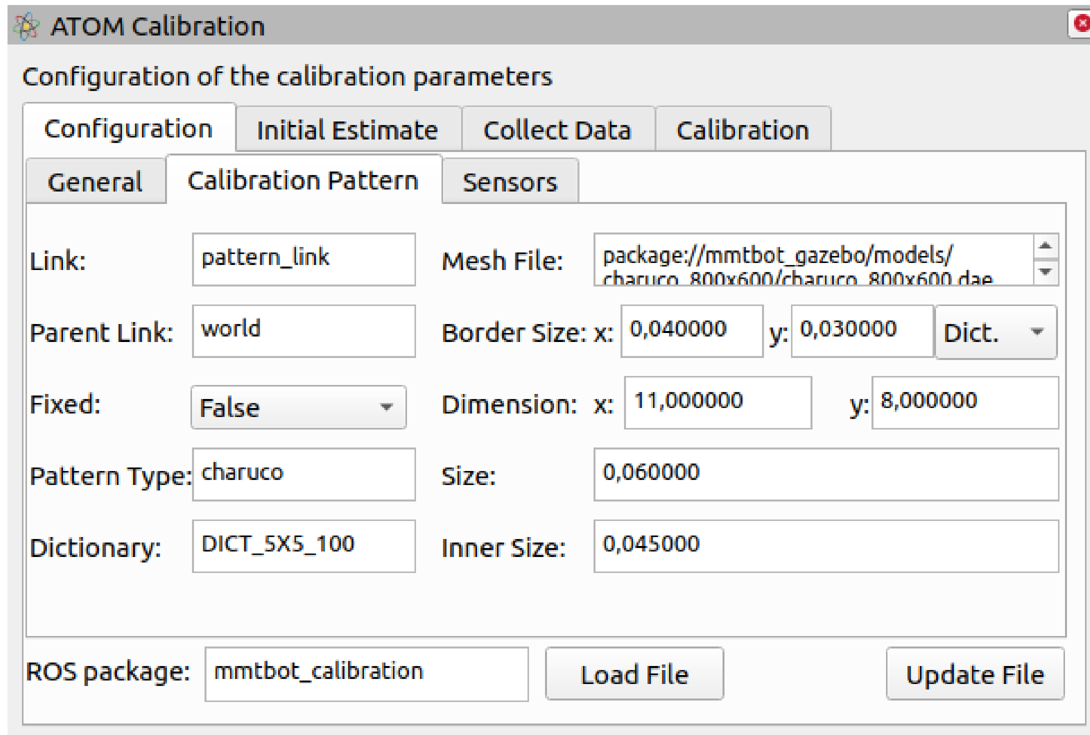


Figure 4.10: Parsing of file to the Configuration Tab Widget: Second tab.

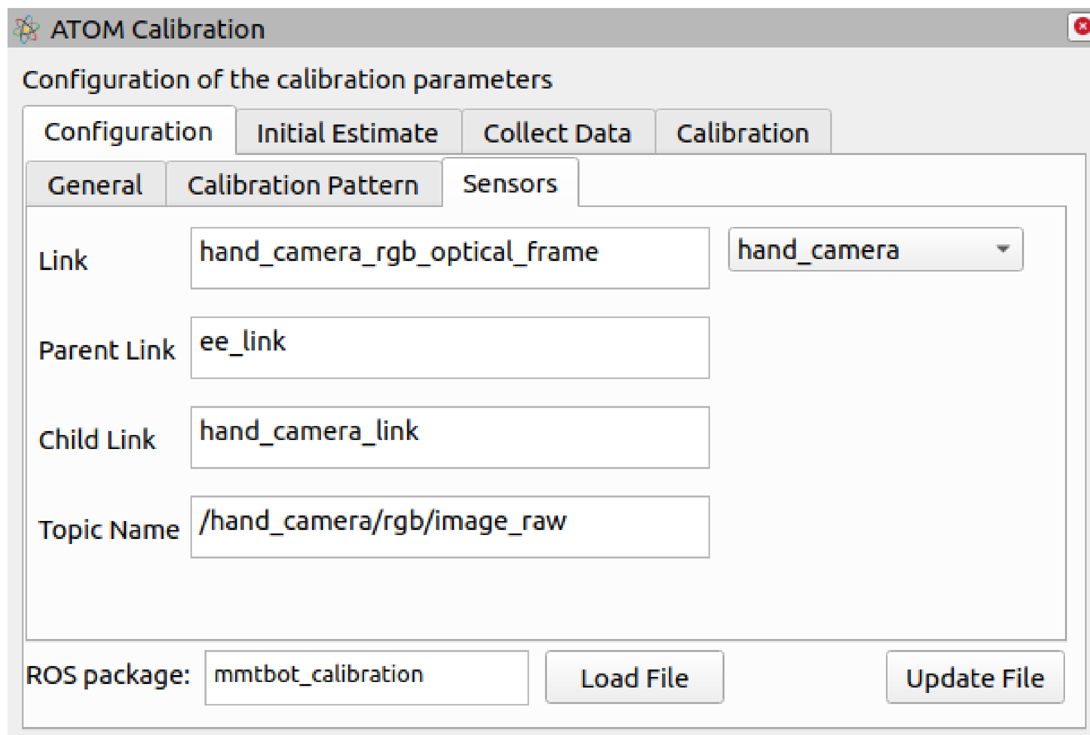


Figure 4.11: Parsing of file to the Configuration Tab Widget: Third tab.

The only tab that differs from the others is the sensors tab, since there in no way of knowing how many sensors the robotic system would have and so it would not be possible to have line edits for each sensor. Therefore, in this tab, the sensors are all put on a combo box (Figure 4.12). and, as the user alternates between the sensors in this combo box, the parameters change accordingly.

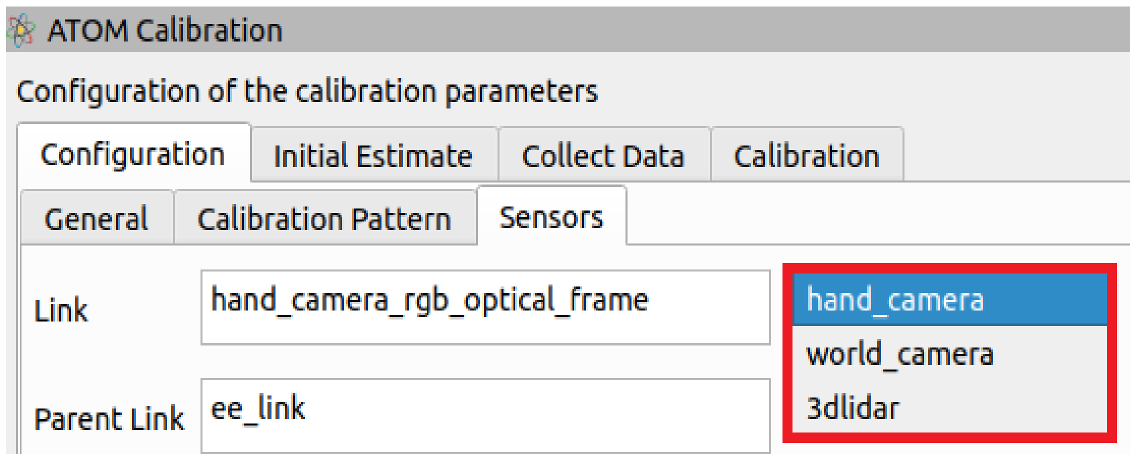


Figure 4.12: Combo box of the sensors tab selected (all sensors shown).

The line edits of the parameters in all three tabs can be modified in the interface by the user, with the 'Update File' button, as its name indicates, updating the configuration file with all the parameter values that are on those line edit boxes at the time the button is clicked.

The loading and updating of the file were both done using yaml-cpp[8], which is a YAML parser and emitter in C++. Using this tool, when the file was parsed (i.e., the "Load file" button was clicked), it was possible to retrieve all the values of each parameter from the file to the code, set the parameters in the ROS Parameter Server and write them on the panel. The reverse process was done when the "Update file" button was pushed, where the values from the panel would be retrieved by the code and then the parameters would be set in the ROS Parameter Server and emitted in the yaml file, updating its content.

This concludes the approach for the first tab widget of the implemented plugin. The following section will concern the second tab widget developed, aimed for setting the initial estimate of the sensors' pose.

[8] `https://github.com/jbeder/yaml-cpp`

4.2.2 Set Initial Estimate

This subsection is aimed at developing the interface of the initial estimate phase of the calibration procedure. This phase and the ideal functionalities for its interface were already addressed in Section 2.3.2.

In short, an initial estimate of the sensors' pose is necessary to avoid the local minima problem, assuring that the optimization will converge into its optimal solution. ATOM provides an interactive way of moving the sensors to perform this first guess, using interactive markers. The fact that it is possible to change the sensor configuration in accordance to the interactive marker's pose, defined by the user, allows a real time comparison with the real world scene.

Figure 4.13 shows the end result of the tab widget developed. The approach for each element and its functionality will be explained throughout the rest of this section.

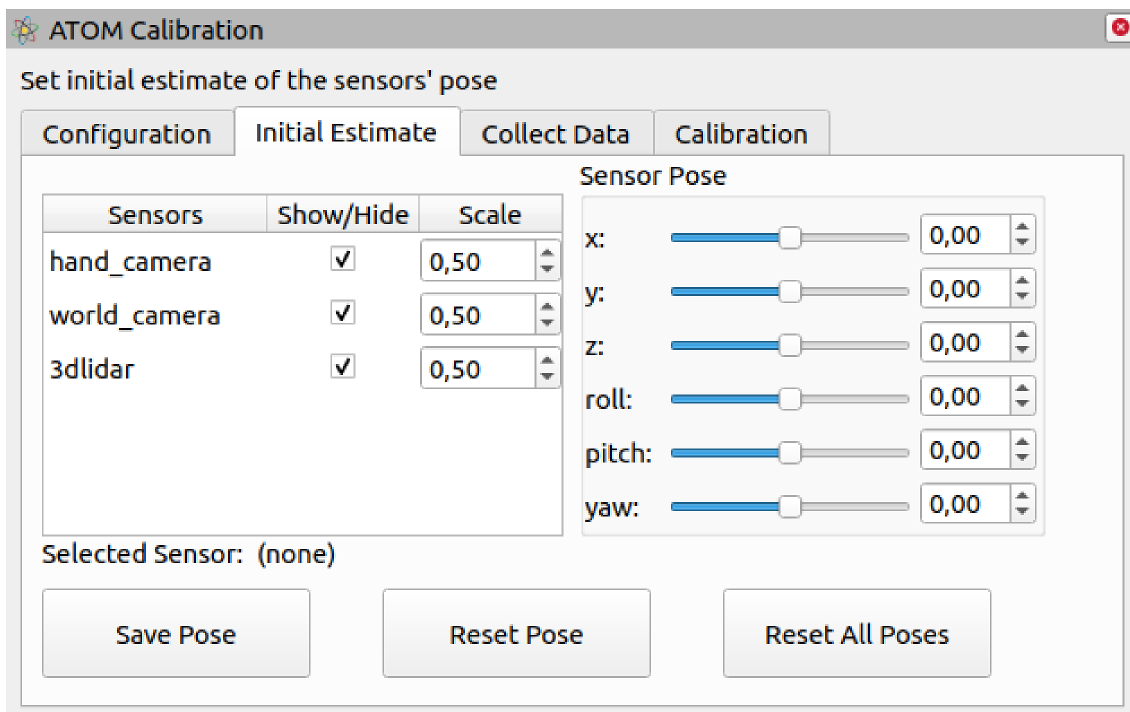


Figure 4.13: Layout of the Initial Estimate Tab Widget.

When creating the layout of the panel, the first functionality that was thought was the ability to have all the sensors of the robotic system listed. This was done using the sensors' names retrieved from the configuration file previously loaded and adding them to a QTableWidget object from Qt.

A text label was then added below the table that would change depending on which sensor is selected, in order to tell the user in which sensor the other functionalities seen in the panel are being performed. In the case of the figure shown above, no sensor is selected, but Figure 4.14 shows an example of one of the sensors (*hand_camera*) being selected.

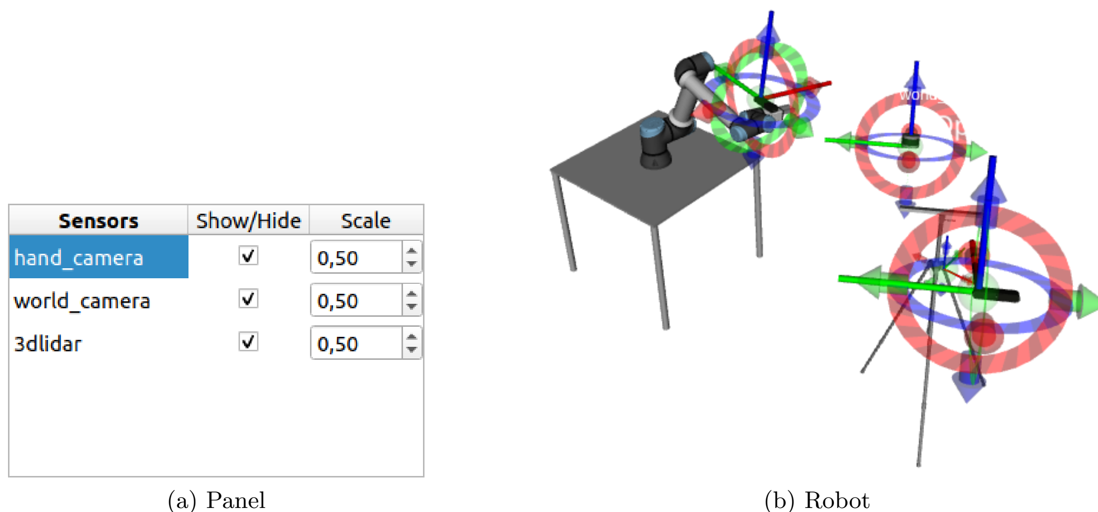
Sensors	Show/Hide	Scale
hand_camera	✓	0,50
world_camera	✓	0,50
3dlidar	✓	0,50

Selected Sensor: hand_camera

Figure 4.14: Initial Estimate Tab: Selecting a sensor from the table of sensors.

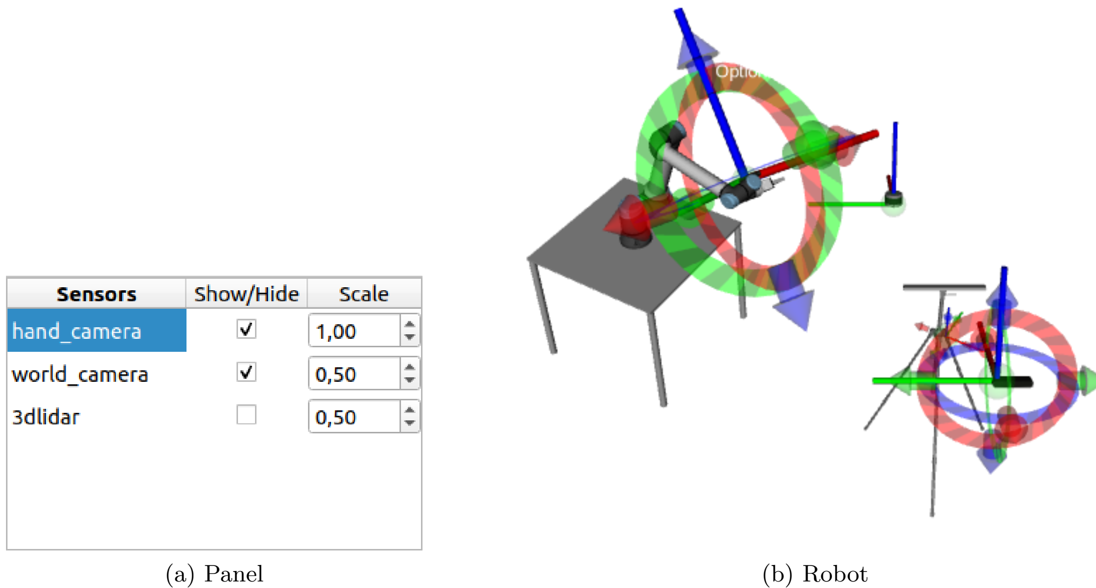
Still in this table, there are also two other columns that have not been talked about: the check boxes in the "Show/Hide" column and the spin boxes in the "Scale" column. These were two additional functionalities that were added especially with robotic systems that have a large number of sensors in mind, since the RViz environment could become overcrowded with interactive markers.

These interactive markers (that represent each sensor) were all previously set at a default scale and visibility, and these status could only be changed collectively in the displays panel. However, the user might not want to hide all sensors or have all interactive markers represented at the same scale. The added columns provide the users the possibility to change these properties individually. Figures 4.15 and 4.16 show, respectively, the properties before and after being changed in the panel and its effect on the interactive markers of the robot.



(a) Panel (b) Robot

Figure 4.15: Initial Estimate Tab: Sensors Visibility and Scale (default).



(a) Panel (b) Robot

Figure 4.16: Initial Estimate Tab: Sensors Visibility and Scale (changed).

As evident by both figures, by unchecking the check box for the 3dlidar sensor, the respective interactive marker loses its visibility. As for the *hand_camera* sensor's scale, it is possible to see that it doubled in size.

These functionalities were implemented resourcing to ROS services. There were two service types that were implemented in the ATOM ROS package, the GetSensorInteractiveMarker[9] and the SetSensorInteractiveMarker[10].

The GetSensorInteractiveMarker is a service type that takes no argument and returns two variables, a boolean for the scale and a float for the visibility. In the code, it is only called one time for each sensor, when the sensors are first inserted in the table, under the name */set_initial_estimate/<sensor>/get_sensor_interactive_marker* (with <sensor> representing the name of each sensor). When it is called, it establishes a communication with ATOM, returning the current state of these variables, that are then used to set all of the elements of the "Show/Hide" and "Scale" columns of the table accordingly.

As for the SetSensorInteractiveMarker service type, it takes two arguments, for the visibility (boolean) and the scale (float), and returns a message to verify if the service was correctly called. Every time the state of any of the check boxes or spin boxes on the table is changed, the service */set_initial_estimate/<sensor>/set_sensor_interactive_marker* is called and each sensor is iterated, with each iteration passing the state of the second and third columns of the respective sensor as arguments of this service, changing the state of the interactive markers appropriately.

[9] `https://github.com/lardemua/atom/blob/noetic-devel/atom_msgs/srv/GetSensorInteractiveMarker.srv`

[10] `https://github.com/lardemua/atom/blob/noetic-devel/atom_msgs/srv/SetSensorInteractiveMarker.srv`

On the right side of the tab widget, it is also possible to see some sliders in a group box titled "Sensor Pose". This is part of another functionality added to this interface. Using the mouse cursor to drag the interactive markers to their estimated pose is very helpful, but it only gets them to their general pose. Therefore, having a way to perform the micro movements of each interactive markers' xyz and rpy (position and orientation) would also be a good functionality to add to the panel, improving the accuracy of the estimation.

The approach in implementing this functionality was that whenever a sensor is selected, the values of x, y, z, roll, pitch and yaw from the group box would change to match that sensor's pose, which was retrieved from the robot's transforms. Then, every time one of the sliders or spin boxes in that group box was changed, that sensor would move accordingly. Figures 4.17 and 4.18 show an example of this, with one of the sensors (*world_camera*) being selected and the slider being moved to put the sensor in its place.

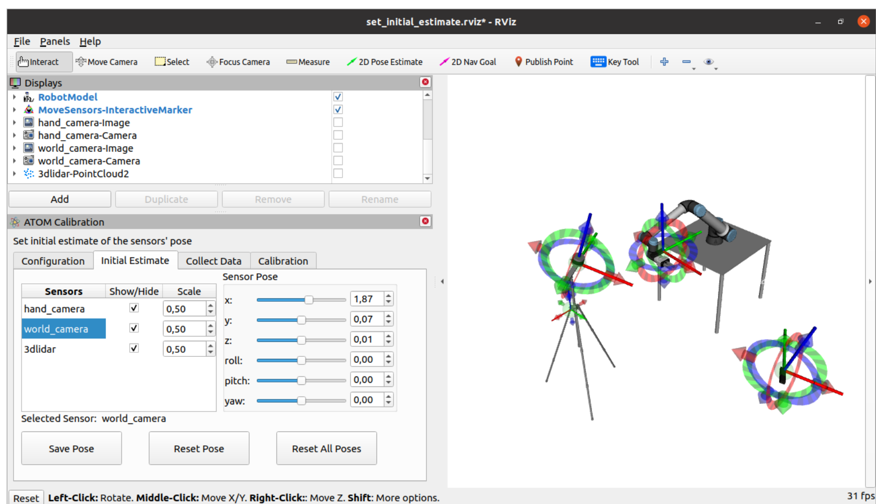


Figure 4.17: Initial Estimate Tab: world_camera out of place.

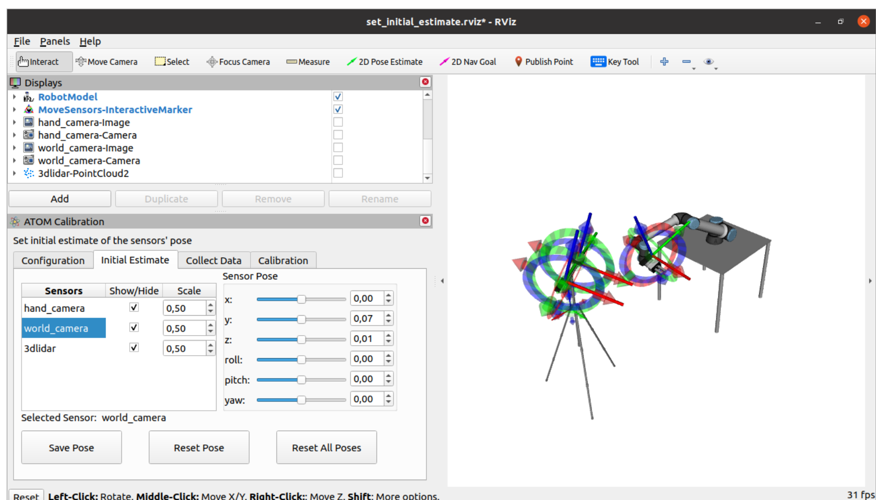


Figure 4.18: Initial Estimate Tab: world_camera sensor positioned with sliders.

The way the interactive markers are able to move by changing the sliders or spin boxes on the panel is through the use of a ROS message[11]. The message type is visualization_msgs/InteractiveMarkerFeedback[12]. These messages are published under the topic /set_initial_estimate/feedback every time an interactive marker is moved. So, by publishing a message of this type from the code under this same topic every time any element of the group box on the panel is changed, the interactive markers change as well.

Lastly, when the sensors are being positioned, the users will need to save that sensor pose configuration or they might want to reset one particular sensor or even reset all sensors to their last saved poses. The three buttons on the lower part of the panel are used to do that, in a very similar way to how the micro movements functionality previously implemented. Every time any of these buttons is clicked, a message of the same type as the above one is published under the same topic. What determines which message is destined for each action is the message parameters, namely the menu_id, marker_name and event_type, that vary depending on what triggered that message (movement, save pose, reset pose or reset all).

Figure 4.19 shows a flowchart for the messages and services for this calibration phase.

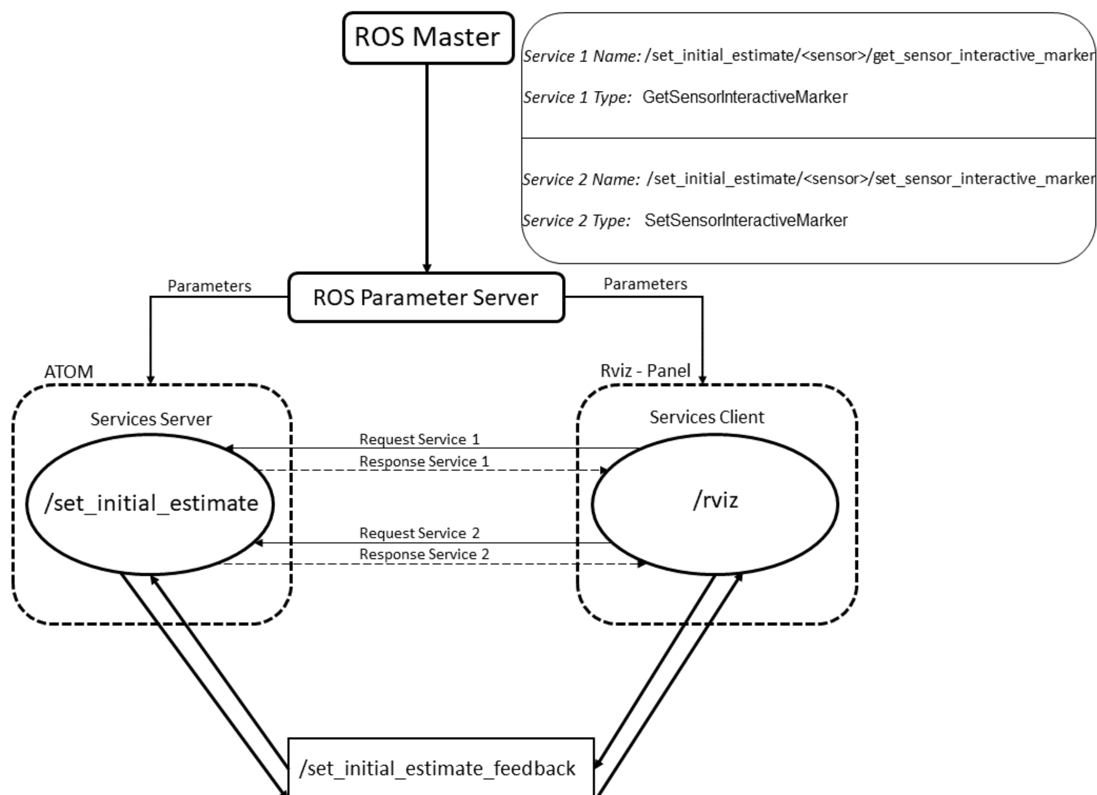


Figure 4.19: Flowchart for the communication architecture of the initial guess phase: the ellipses represent the nodes; the box represent the message topic; the 'request' and 'response' arrows represent the services' communication, properly identified in top right.

[11] http://wiki.ros.org/Messages

[12] http://docs.ros.org/en/melodic/api/visualization_msgs/html/msg/InteractiveMarkerFeedback.html

4.2.3 Data Collection

The last step before the calibration can be done is the data collection. As already explained in Section 2.3.3, this step comes after the appropriate labeling of the sensors' data, where all the data must then be gathered and saved in an accessible format, in order for it to be used afterwards in the optimization procedure. The way ATOM does this is by collecting the data and respective label of the sensors at moments defined by the user in which the scene has remained static for a certain period of time. All this information is then saved in a JSON file that will be accessed by the optimizer. The listing 4.2 below shows a tree view of a dataset for the current robotic system (mmtbot).

```
1  — additional_sensor_data {0}
2  — calibration_config {7}
3          — anchored_sensor :
4          — bag_file : $ROS_BAGS/mmtbot/11_03_2021.bag
5          — calibration_pattern  {10}
6              — border_size  {2}
7              — dictionary : DICT_5X5_100
8              — ...
9          — description_file : package://mmtbot_description/urdf/mmtbot.urdf.xacro
10         — max_duration_between_msgs : 1000
11         — sensors  {3}
12         — world_link : world
13 — collections  {27}
14          — 0  {4}
15          — 1  {4}
16              — additional_data  {0}
17              — data  {3}
18              — labels  {3}
19                  — 3dlidar  {3}
20                      — detected :  true
21                      — idxs  [756]
22                      — idxs_limit_points  [24]
23                  — hand_camera  {2}
24                      — detected :  true
25                      — idxs  [88]
26                  — world_camera  {2}
27              — transforms  {35}
28          — 2  {4}
29          — ...
30 — sensors  {3}
31     — 3dlidar  {7}
32         — _name : 3dlidar
33         — calibration_child : 3dlidar_base_link
34         — calibration_parent : tripod_left_support
35         — chain  [5]
36         — msg_type : PointCloud2
37         — parent : 3dlidar
38         — topic : /3dlidar/points
39     — hand_camera  {9}
40     — world_camera  {9}
```

Listing 4.2: Tree View of the JSON File for the data collection

This file contains important information about the sensors, such as the sensor transformation chain and specific information about each collection, i.e., sensor data, partial transformations and data labels, along with the information about the configuration of the calibration that was already talked about in the previous sections.

However, during the process of the calibration with the interface, the user would only need a few of these parameters, as most of this information is only important for the backend of the ATOM calibration, since the optimizer will need this information for the optimization process.

Therefore, allowing the users to see the collections made by showing the parameters of the json file that may be the most useful to them was the first step of the interface layout concerning this calibration phase.

These collections would be made by attempting to position an interactive marker in the place of the calibration pattern, that could be identified by the pointcloud cluster in the RViz environment for each lidar sensor. Even though each collection contains information for every sensor, the reason this positioning only needs to be done for LiDAR sensors is because ATOM already provides the automatic labeling of cameras, since the pattern can be easily recognized from the camera image. To be able to do this, the approach was to implement a group box similar to the one implemented on the previous phase, with a combo box to select the LiDAR sensor and with sliders to change the position of the interactive marker.

Additionally, there should be a button in the panel to save each collection, along with a button to delete a saved collection.

Figure 4.20 shows the developed tab with the appropriate panel layout to match the description of the functionalities mentioned above.

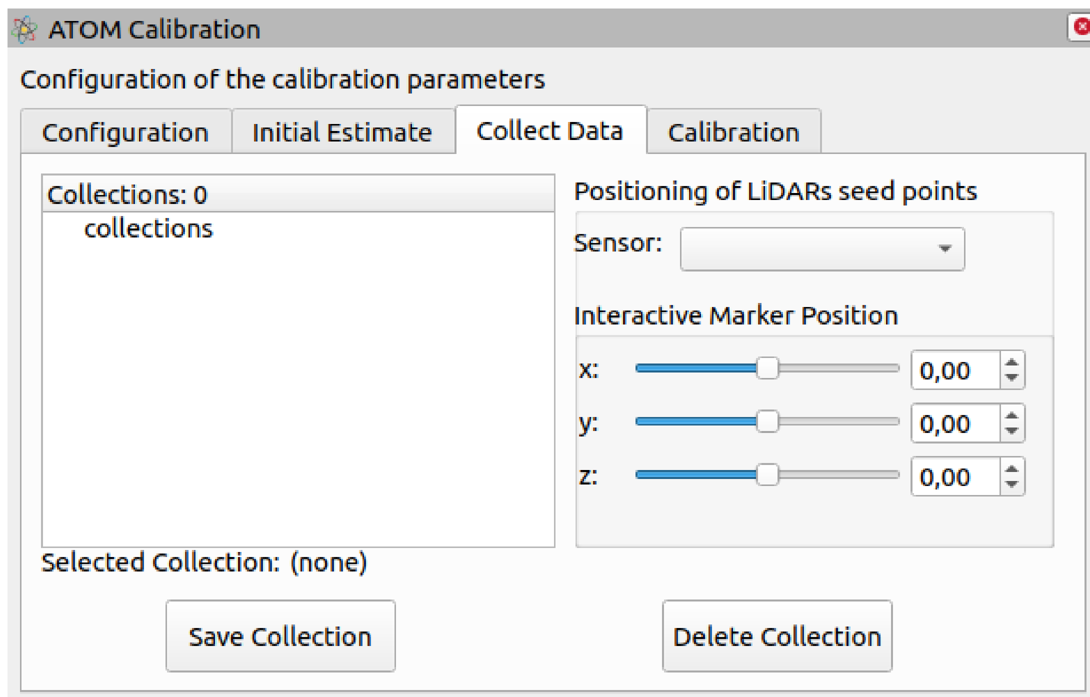


Figure 4.20: Layout of the Data Collection Tab widget.

As mentioned above, the approach taken for the micro movements was similar to the one used on the Initial Estimate tab. However, in this case, instead of the interactive markers being used to estimate the sensors' pose, they were used to position them as close to the calibration pattern as possible. Also, since it only matters to position the marker on top of the pattern, the orientation is not important for these movements, so the group box only moves the position (x, y and z) and not the Euler angles (roll, pitch and yaw), like the previous one. This process would have to be repeated for each lidar sensor (changed in the "Sensor" combo box), but for this robotic system, there is only one of these sensors, the *3dlidar*. Figure 4.21 shows the interactive marker out of place and Figure 4.22 shows the marker positioned in the calibration pattern after moving the sliders.

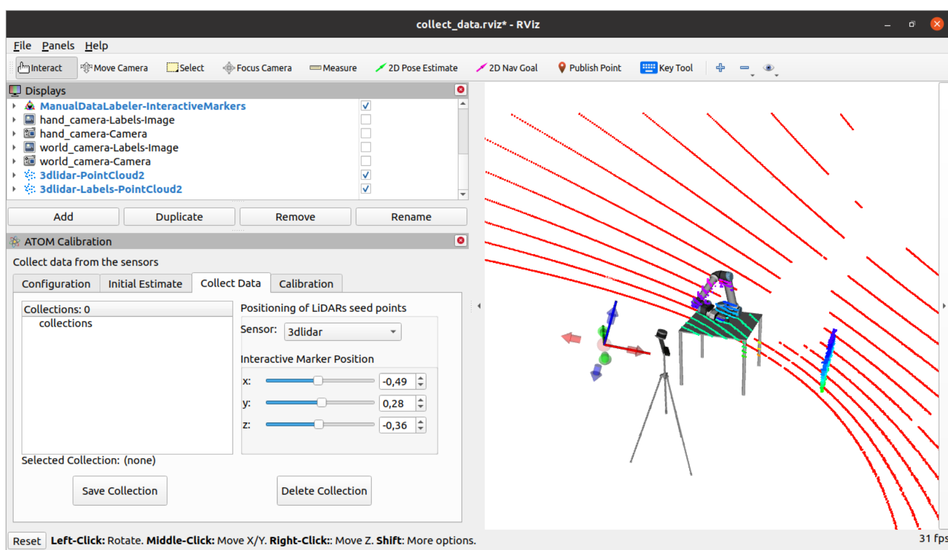


Figure 4.21: Data Collection Tab: Interactive marker out of place.

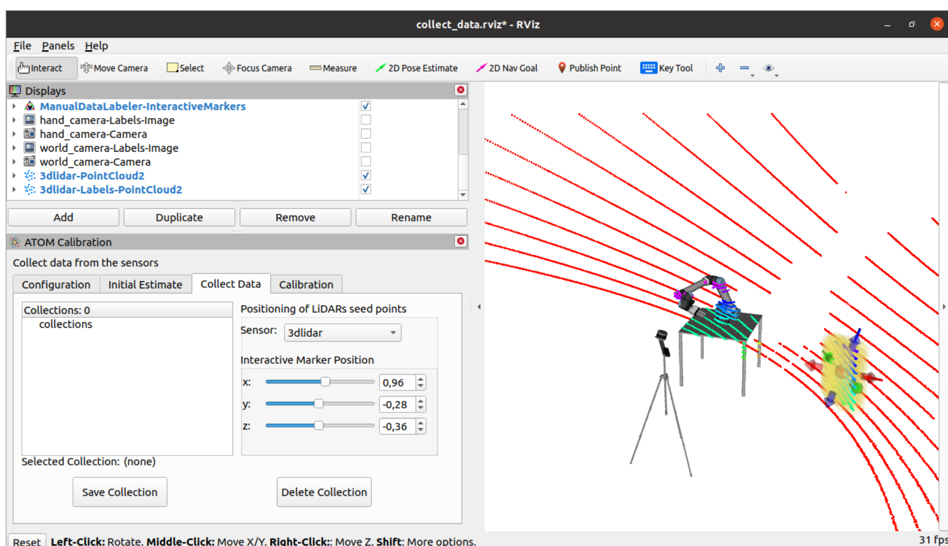


Figure 4.22: Data Collection Tab: Interactive marker being positioned.

Just like the previous tab, the movements are also performed using a ROS message of the same type, with the difference that these messages are published under a different topic (/data_collect/feedback).

The "Save Collection" button allows the user to save the collections after all the interactive markers are properly positioned in the calibration pattern. By clicking on this button, a SaveCollection[13] service type (implemented in the ATOM ROS package), by the name /collect_data/save_collection, is called. This service takes no arguments and what it does is add all the information of the collection made to the json file of that dataset. Then, a function in the code is called, where another service (/collect_data/get_dataset) is called. This service, GetDataset[14], also implemented in the ATOM ROS package, takes no arguments and returns the json file with all the information up until the point of clicking the button. Using a json parser[15], this file's content can then be retrieved within the code and added to the tree widget object in the panel. Figure 4.23 shows the result of four collections being saved.

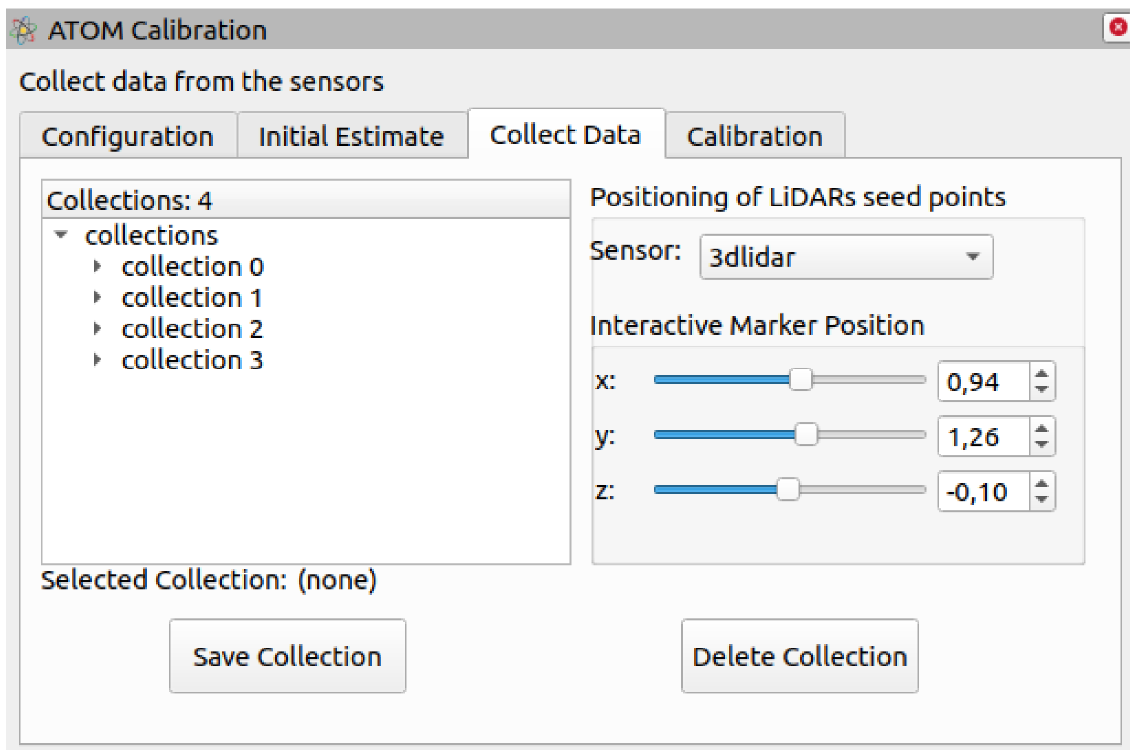


Figure 4.23: Data Collection Tab: 4 Collections saved.

From all the information in the json file showed in the listing 4.2, the only one that seems relevant for the user to see in the interface is whether the sensors were labeled or not. The rest, as mentioned before, is used by the optimizer during the calibration, but it does not seem important for the user to see in the interface. Figure 4.24 shows the tree view of what can be seen in the panel for each collection.

[13] https://github.com/lardemua/atom/blob/noetic-devel/atom_msgs/srv/SaveCollection.srv
[14] https://github.com/lardemua/atom/blob/noetic-devel/atom_msgs/srv/GetDataset.srv
[15] https://github.com/nlohmann/json

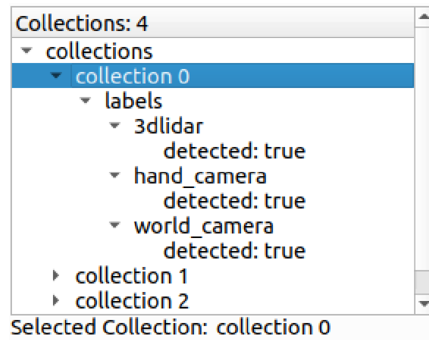


Figure 4.24: Data Collection Tab: collections tree.

These collections can also be deleted. By selecting a collection and clicking on the "Delete Collection" button, a DeleteCollection[16] service type is called, under the name /collect_data/delete_collection. This service takes one argument, which is a string with the name of the collection to be deleted and communicates with the ATOM package, where the information regarding that collection is removed from the json file. Afterwards, it is done the same process as the save button: the /collect_data/get_dataset service is called, returning the updated json file (with the collection already deleted) and then the file is parsed in the code and the tree widget is changed accordingly.

If no collection is selected, clicking on the button to delete will warn the user of that fact (Figure 4.25).

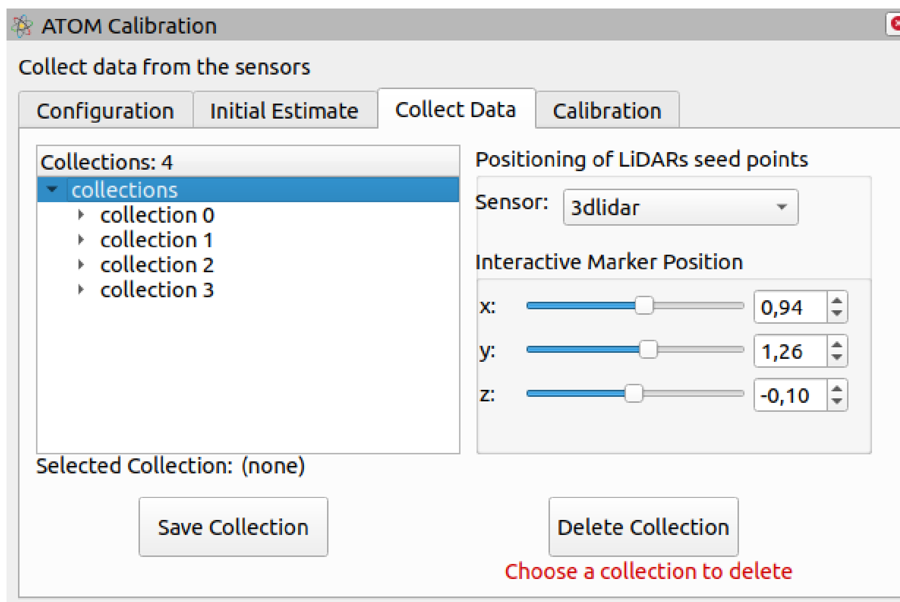


Figure 4.25: Data Collection Tab: Delete functionality - No collection selected.

If, on the other hand, the collection was selected, by clicking on the button a dialog box shows up, for the user to confirm that action (Figure 4.26).

[16] https://github.com/lardemua/atom/blob/noetic-devel/atom_msgs/srv/DeleteCollection.srv

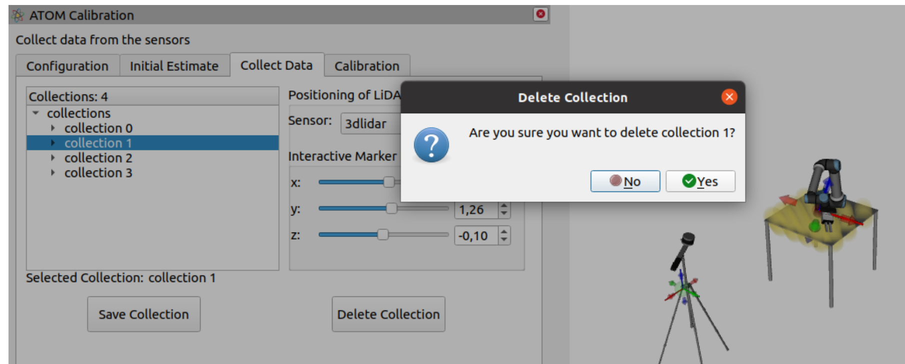


Figure 4.26: Data Collection Tab: Delete functionality - Confirmation Dialog Box.

Figure 4.27 shows the flowchart for the message topics and the services for this calibration phase.

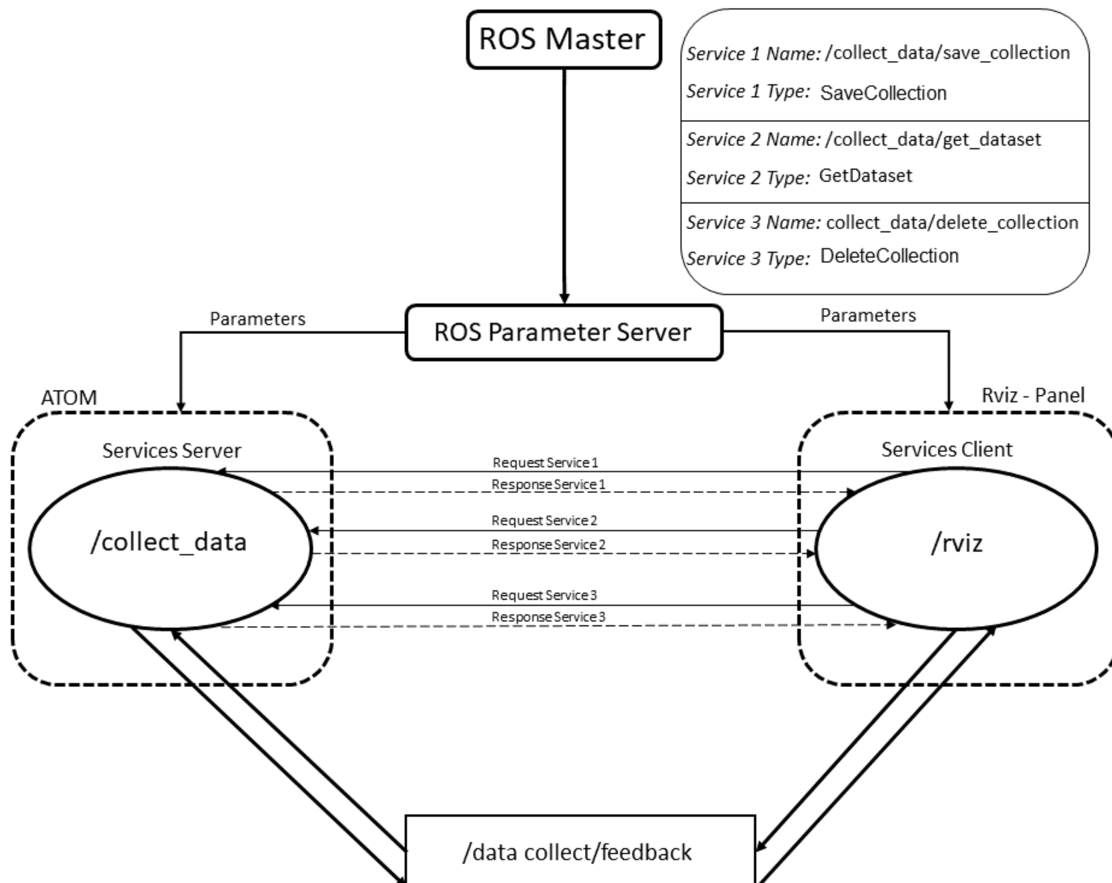


Figure 4.27: Flowchart for the communication architecture of the data collection phase: the ellipses represent the nodes; the box represents the message topic; the 'request'/'response' arrows represent the services' communication, properly identified in the top right.

4.2.4 Calibration

This last phase is dedicated to the optimization procedure, where all the information gathered in the previous phases is used to perform the calibration. Consequently, there are not a lot of functionalities needed for the calibration itself, since all of them were already addressed and properly set in the previous sections.

Therefore, this phase is more focused on seeing the results of said calibration, and the way ATOM currently does that is by calling a system calibration with a launch file that calls the 'calibrate' script, along with several other scripts, with the appropriate arguments to configure the calibration procedure.

However, ATOM also presents an alternative way, instead of calling the launch file with several scripts at the same time, that allows for the debugging of the calibrate script. This can be done by calling the launch file with the argument *run_calibration* set on false, which launches everything except the 'calibrate' script, and then run the script in standalone mode. This way, the debugging of the calibration file could be performed, by calling one or several of its command-line arguments.

To get a detailed information about the command-line arguments and how they can be called, the *rosrun atom_calibration calibrate -h* command can be executed and this information is shown on the terminal, which gives an extensive list of these arguments, exactly like the one showed in the listing 4.3 below.

```
usage: calibrate [–h] [–sv SKIP_VERTICES] [–z Z_INCONSISTENCY_THRESHOLD] [–vpv]
                 [–vo] –json JSON_FILE [–v] [–rv] [–si] [–oi] [–pof] [–ss SAMPLE_SEED]
                 [–sr SAMPLE_RESIDUALS]  [–ajf] [–uic] [–rpd] [–nig translation rotation]
                 [–ssf SENSOR_SELECTION_FUNCTION] [–ox OUTPUT_XACRO] [–ipg]
                 [–csf COLLECTION_SELECTION_FUNCTION] [–oj OUTPUT_JSON]
                 [–phased]

optional arguments:
  –h, ––help       Show this help message and exit
  –sv SKIP_VERTICES, ––skip_vertices SKIP_VERTICES
                   Skip vertices. Useful for fast testing
  –z Z_INCONSISTENCY_THRESHOLD, ––z_inconsistency_threshold
     Z_INCONSISTENCY_THRESHOLD
                   Threshold for max z inconsistency value
  –vpv, ––view_projected_vertices
                   Visualize projections of vertices onto images
  –vo, ––view_optimization
                   Displays generic total error and residuals graphs
  –json JSON_FILE, ––json_file JSON_FILE
                   Json file containing input dataset.
  –v, ––verbose    Be verbose
  –rv, ––ros_visualization
                   Publish ros visualization markers.
  –si, ––show_images
                   Shows images for each camera
```

```
25 -oi, --optimize_intrinsics
26                     Adds camera instrinsics to the optimization
27 -pof, --profile_objective_function
28                     Runs and prints a profile of the objective function, then exits.
29 -ss SAMPLE_SEED, --sample_seed SAMPLE_SEED
30                     Sampling seed
31 -sr SAMPLE_RESIDUALS, --sample_residuals SAMPLE_RESIDUALS
32                     Samples residuals
33 -ajf, --all_joints_fixed
34                     Assume all joints are fixed and because of that draw a single robot mesh.
35                     Overrides automatic detection of static robot.
36 -uic, --use_incomplete_collections
37                     Remove any collection which does not have a detection for all sensors.
38 -rpd, --remove_partial_detections
39                     Remove detected labels which are only partial. Used or the Charuco.
40 -nig translation rotation, --noisy_initial_guess translation rotation
41                     Percentage of noise to add to the initial guess atomic transformations set
42                     before.
43 -ssf SENSOR_SELECTION_FUNCTION, --sensor_selection_function
     SENSOR_SELECTION_FUNCTION
44                     A string to be evaluated into a lambda function that receives a sensor name
45                     as input and returns True or False to indicate if the sensor should be loaded
46                     (and used in the optimization). The Syntax is lambda name: f(x), where f(x)
47                     is the function in python language. Example: lambda name: name in
48                     ["left_laser", "frontal_camera"] , to load only sensors left_laser and
49                     frontal_camera.
50 -ox OUTPUT_XACRO, --output_xacro OUTPUT_XACRO
51                     Full path to output xacro file.
52 -ipg, --initial_pose_ghost
53                     Draw a ghost mesh with the systems initial pose. Good for debugging.
54 -csf COLLECTION_SELECTION_FUNCTION, --collection_selection_function
     COLLECTION_SELECTION_FUNCTION
55                     A string to be evaluated into a lambda function that receives a collection
56                     name as input and returns True or False to indicate if the collection should
57                     be loaded (and used in the optimization). The Syntax is lambda name: f(x),
58                     where f(x) is the function in python language. Example: lambda name:
59                     int(name) > 5 , to load only collections 6, 7, and onward.
60 -oj OUTPUT_JSON, --output_json OUTPUT_JSON
61                     Full path to output json file.
62 -phased, --phased_execution
63                     Stay in a loop before calling optimization, and in another after calling the
64                     optimization. Good for debugging.
```

Listing 4.3: List of all command-line arguments on the calibrate script

With all this in mind, instead of adding just a simple button to perform a calibration with no extra configuration whatsoever, as initially planned in Section 2.3.4, the thought process for the last tab was allowing users to set the command-line within the panel that could be used to ran the calibration with any of the arguments listed above.

Figure 4.28 shows the proposed layout for this tab widget. It is possible to see a table with multiple rows, one for each of the command-line arguments previously showed, along with a text box for the command-line to run the calibration and some buttons.

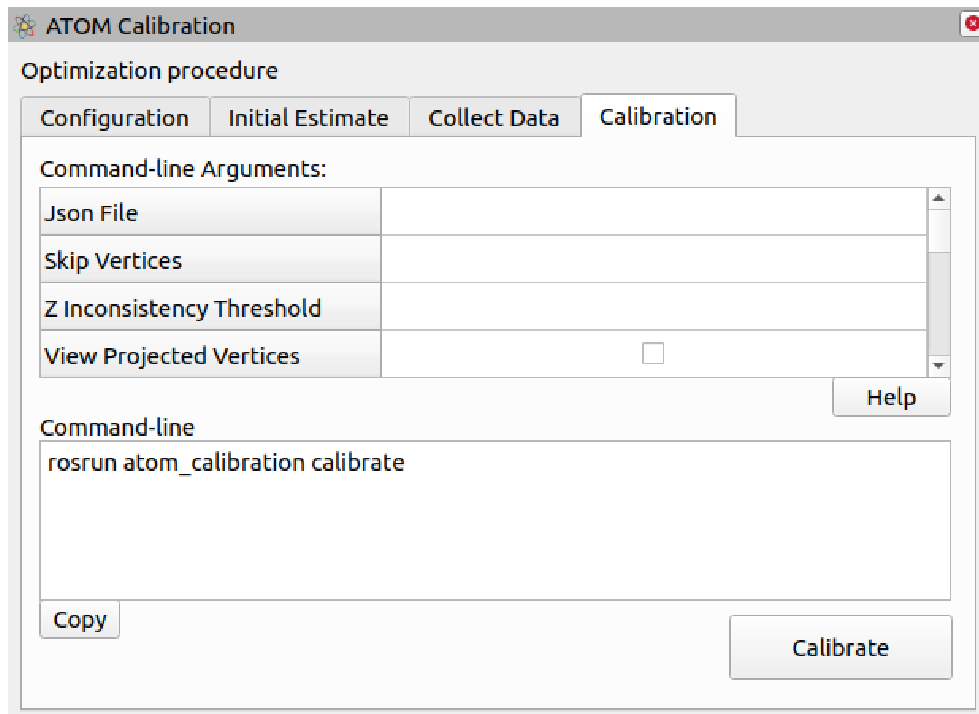


Figure 4.28: Layout of the Configuration Tab Widget.

The "Help" button opens a QMessageBox, which is an object from Qt that can be configured as a dialog box of several types. In this case, when the button is clicked, it opens an information dialog box, as it can be seen in Figure 4.29 below, containing a brief explanation of each of the arguments present on the table.

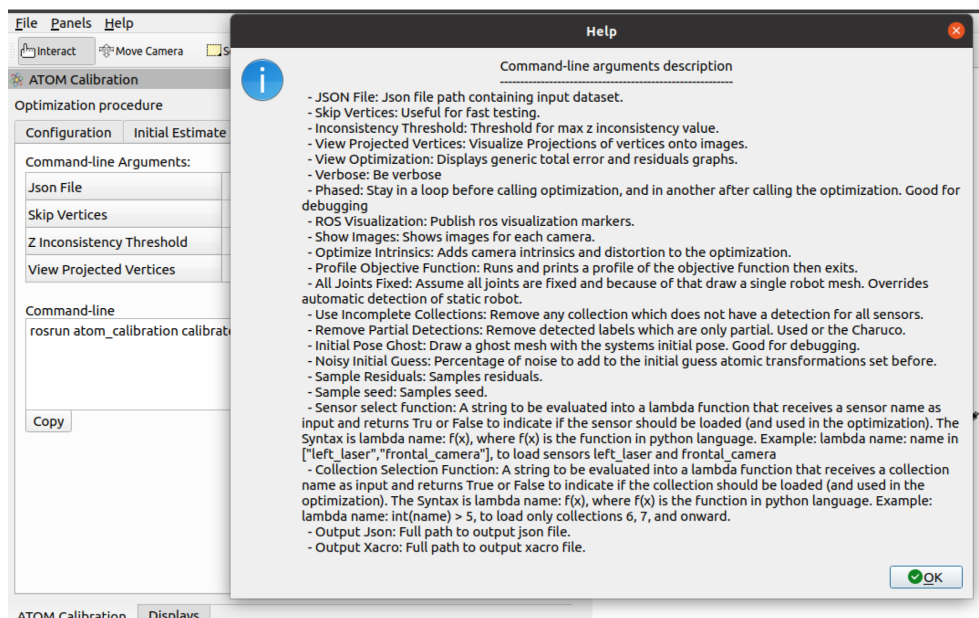


Figure 4.29: "Help" Dialog Box that pops up everytime the "Help" button is clicked.

Each cell of the table can be edited, and each change made in one of these cells is connected to an event that properly adds the respective argument to the "Command-line" text box shown below the table. This way, the user can easily create the command that is needed to be run without having to type it all in and knowing exactly the command for each argument.

Figure 4.30 shows some cells changed in the table and the respective arguments automatically appearing on the text box.

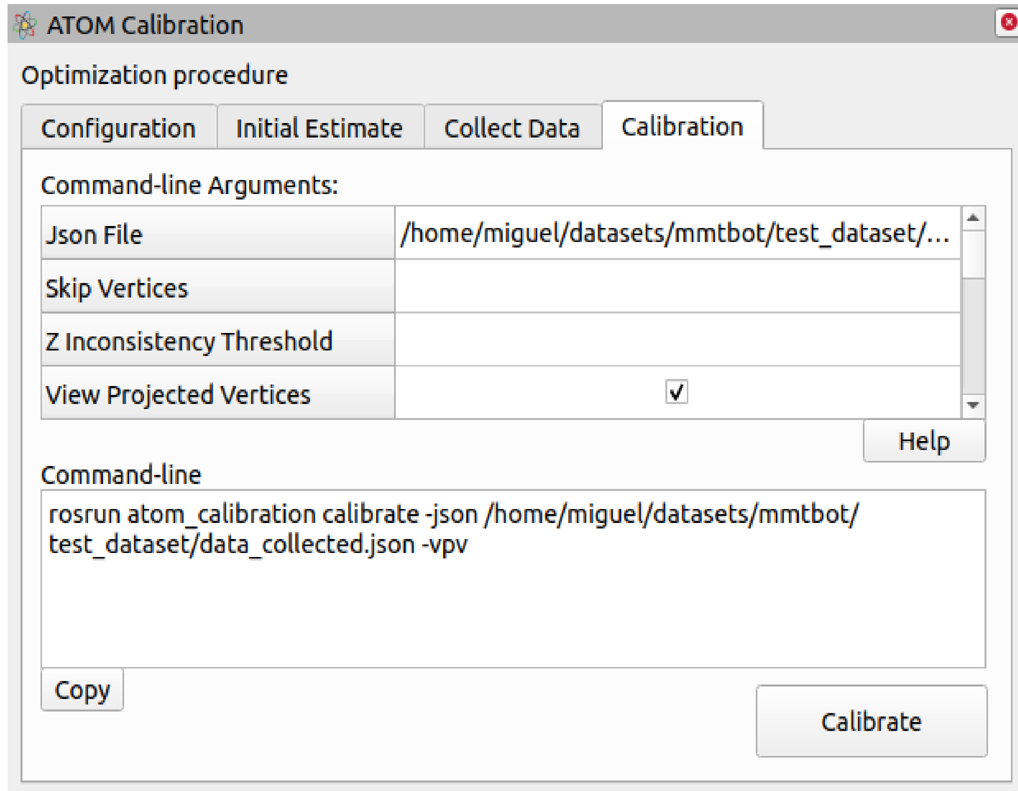


Figure 4.30: Configuration Tab Widget: Command-line Text Box changing according to the changes made on the table.

After all the elements on the table are properly set and the command-line changed accordingly, the user can then either click on the "Copy" button or the "Calibrate" button.

The "Copy" button copies the content of the command-line text box to the clipboard, that can then be pastes on the terminal and the command can be ran to start the calibration.

However, if the user does not need any feedback from the terminal and just wants to perform the calibration, the "Calibrate" button initiates the optimization procedure directly by reading the command-line string from the text box and directly running within the code, using the system()[17] function.

[17] https://www.tutorialspoint.com/system-function-in-c-cplusplus

Figure 4.31 shows the panel being configured with the command-line text box changing accordingly and Figure 4.32 shows the visual result of running that command, with the optimization function running in the terminal.

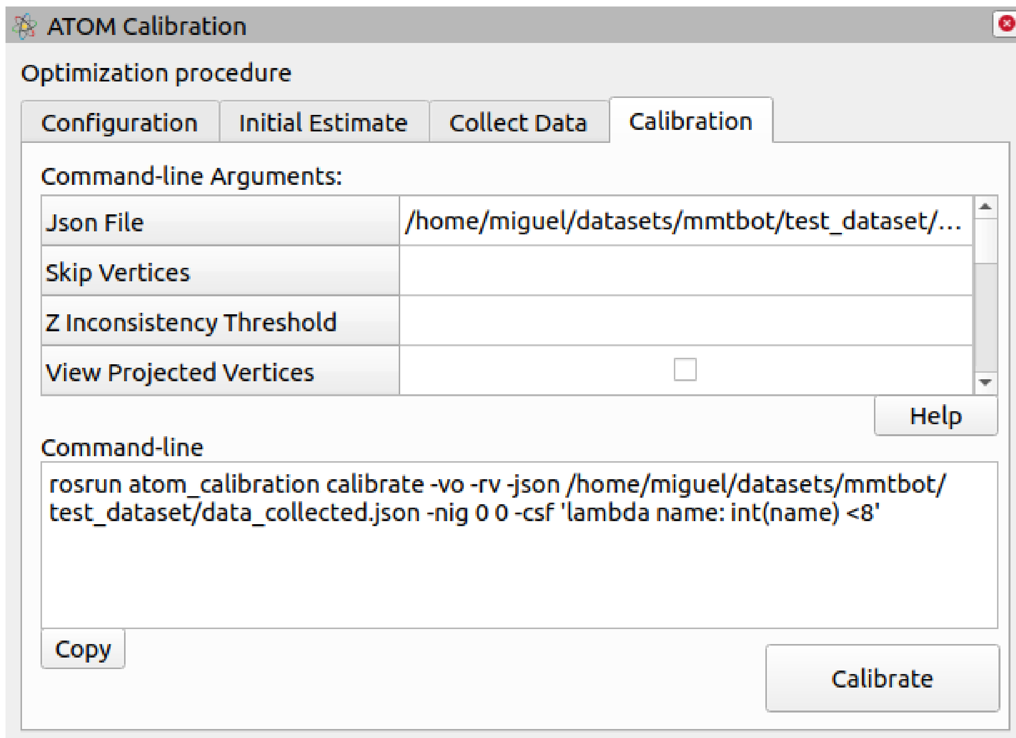


Figure 4.31: Configuration of the command line text box by changing table cells.

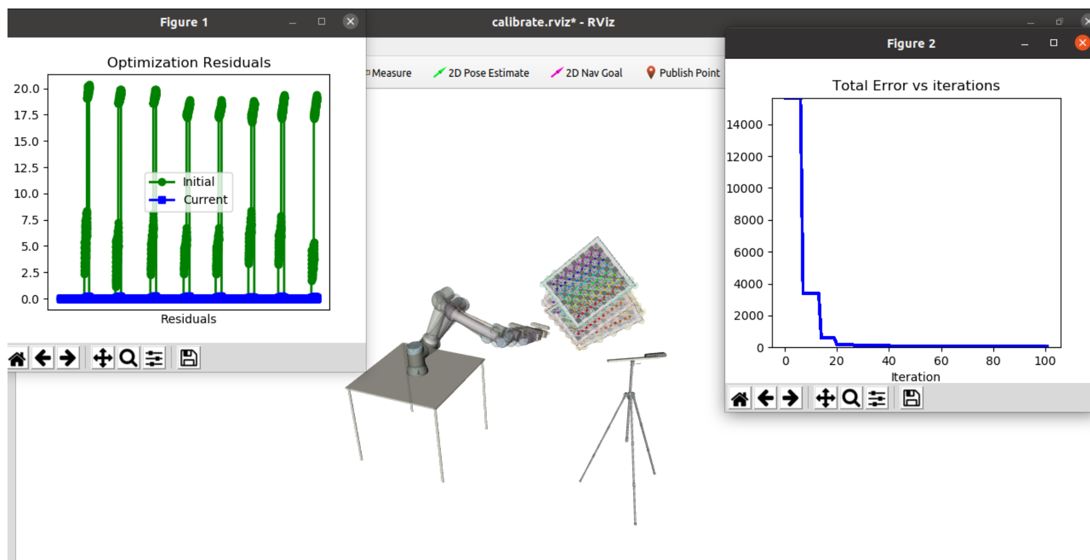


Figure 4.32: Visual result of running the command configured in the panel.

Chapter 5

Results

This chapter gives an overview on the developed interface, discussing its results and providing links of videos of a calibration being performed using this plugin. Following that, since the work was done using only one robotic system (mmtbot), some tests of this calibration with other robotic systems were then made, to help validate the work done. Finally, it is made a comparison of the calibration experience between the developed interface and the HandEye Calibration Interface from MoveIt!.

5.1 Interface overview

In order to test and validate the performance of the proposed approach, it was conducted a calibration of the simulated robotic system addressed in Chapter 4 (mmtbot) using the developed interface and the ATOM framework.

When it comes to the first calibration phase, the possibility of seeing the calibration parameters within the RViz environment during the calibration process and allowing the users to make the necessary changes to them, that can then be saved back in the configuration file, is a great improvement to how this was previously done. Before, every time there was a need to make any change to this file, it had to be changed by hand, going to the directory of the file, which was not very practical. A video with an example of a configuration of the mmtbot calibration parameters using this interface is provided in the following link: `https://youtu.be/KSUbFCFNd08`.

In the second phase, regarding the initial estimate of the sensors' pose, the ATOM framework was already providing a few interactive tools, such as the interactive markers representing the sensors and the ability to save a pose by right clicking on the marker. However, these were not so intuitive, and the implemented plugin was able to offer a lot of advantages, such as listing all the sensors, changing the markers' scale and visibility, performing movements that would help make the estimates more precise instead of just dragging the markers to their general pose, and the ability to save the estimate pose or reset them, either individually or collectively. The following video shows the initial estimate for the sensors in the mmtbot robotic system being made using the ATOM interface: `https://youtu.be/VtHhpO57Sgo`.

As for the data collection phase, the ATOM interface was able to provide the users with a list of the collections as they were being saved, with some information of each collection saved in a tree view. The positioning of the markers to make the collections was also improved by the RViz plugin, by allowing the user to perform micro movements, similarly to the previous phase. This link shows an example of data being collected in the mmtbot robotic system: `https://youtu.be/ZOnPDOnsc5w`.

Finally, the interface's tab that was created for the calibration phase provided a way to easily prepare the command-line to perform the debugging of the optimization procedure. By calling a system through a launch file, launching only the visualization, it was possible to separately run the 'calibrate' script with a variety of arguments, each with their own unique way of being called. The interface facilitated this process, by showcasing all the possible arguments for this script in a table and creating the necessary command-line as the table cells were changed. In the following link, a video of this part of the interface: `https://youtu.be/LOdk748x_lM`.

5.2 Testing the interface with other robotic systems

Having obtained satisfactory results, with the implemented interface being able to be used to calibrate the mmtbot robotic system, which is a simulated robot, some tests were then made for the calibration of real robotic systems, in an attempt to further validate the usability of this interface.

The first robot to be used was the ATLASCAR2, which is an intelligent vehicle developed at the Department of Mechanical Engineering of the University of Aveiro, in Portugal. This vehicle is one more prototype of many autonomous cars developed in the the ATLAS project[1].

The ATLASCAR2 is considered a complex robotic system, since it contains multiple sensors of different modalities: two 2D LiDARs (LMS1xx lasers) and two RGB cameras (pointgrey flea3).

Figure 5.1 shows those sensors, while Figure 5.2 shows said sensors on board of the ATLASCAR2 and Figure 5.3 shows the transformations tree of this vehicle.

(a) LMS1xx LiDAR

(b) point grey flea3 RGB camera

Figure 5.1: Sensors on board of the ATLASCAR2.

[1]`http://atlas.web.ua.pt/`

Figure 5.2: ATLASCAR2 with the LiDAR sensors circled in yellow and the cameras in blue.

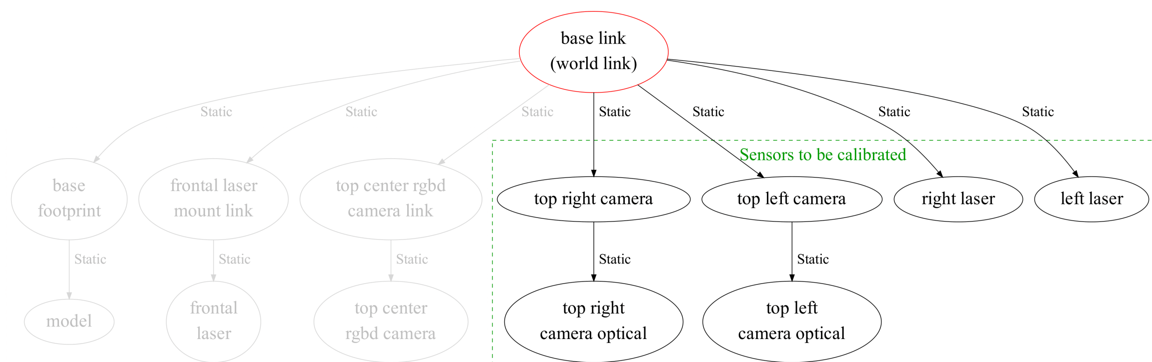


Figure 5.3: Transformations Tree of the ATLASCAR2 robotic system.

Since the ATLAS and ATOM are both projects being conducted in the same University, the package[2] for this robotic system is already modeled and configured correctly for its use with ATOM. Therefore, by installing this package, and playing back the bag file with the recorded information from each sensor, it is possible to conduct the calibration following the same steps as the ones followed for the mmtbot robotic system. Figure 5.4 shows the initial estimate phase of the calibration being performed using the ATOM interface.

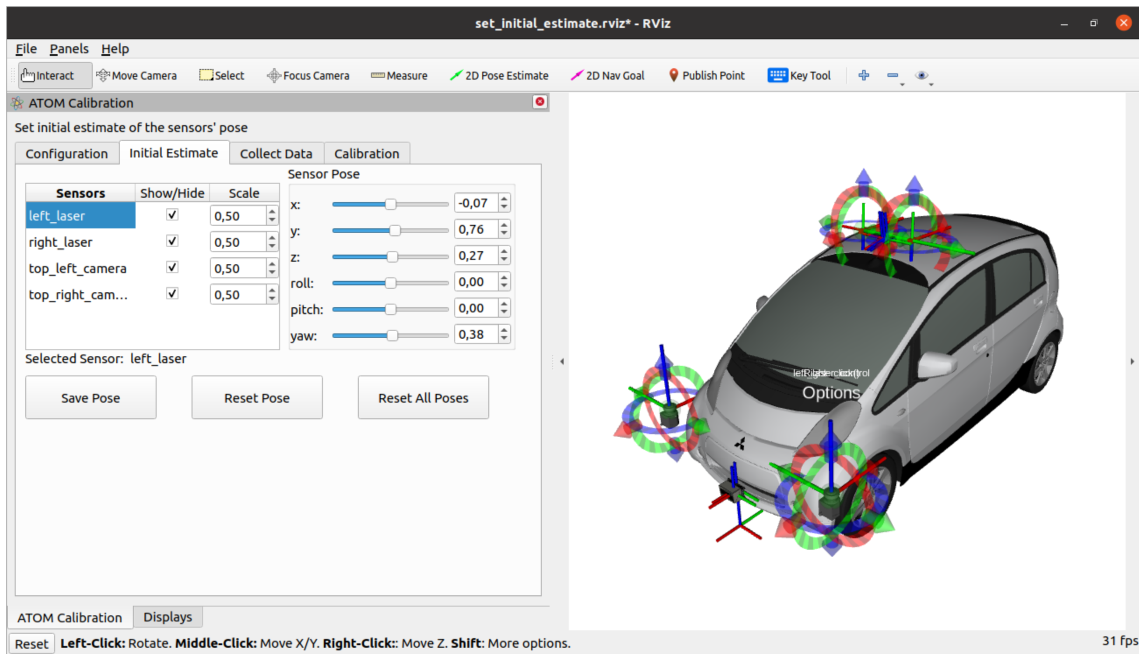


Figure 5.4: ATLASCAR2 robot model represented in the RViz environment with a calibration being performed using the ATOM interface.

Besides this robot, the interface was also tested with the Agrob robotic system, which is an agricultural robot that has three sensors on board: two cameras (3D Zed Cameras) and a velodyne LiDAR sensor (vlp16). Once again, it is a complex robotic system due to it having multiple sensors of different modalities.

Similar to the ATLASCAR2, there is also a ROS package for this robotic system correctly configured to match the prerequisites required by the ATOM framework. Therefore, the RViz plugin created in this interface could also be used for this robot.

Figure 5.5 shows the Agrob v16 robot. Figures 5.6 and 5.7 show, respectively, the robot model, with its sensors duly identified, and its respective transformations tree. As for Figure 5.8, it shows the Agrob in the RViz Environment during the set initial estimate phase of the calibration that is being performed using the ATOM interface.

[2] https://github.com/lardemua/atlascar2

Figure 5.5: AGROB robotic system.

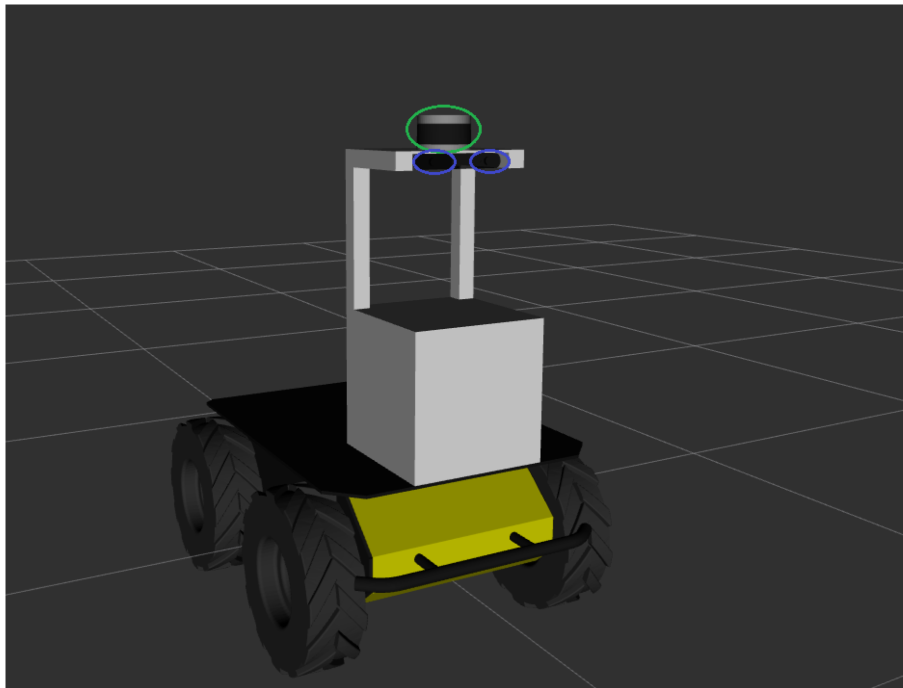


Figure 5.6: AGROB robot model with the cameras circled in blue and the LiDAR sensor in green.

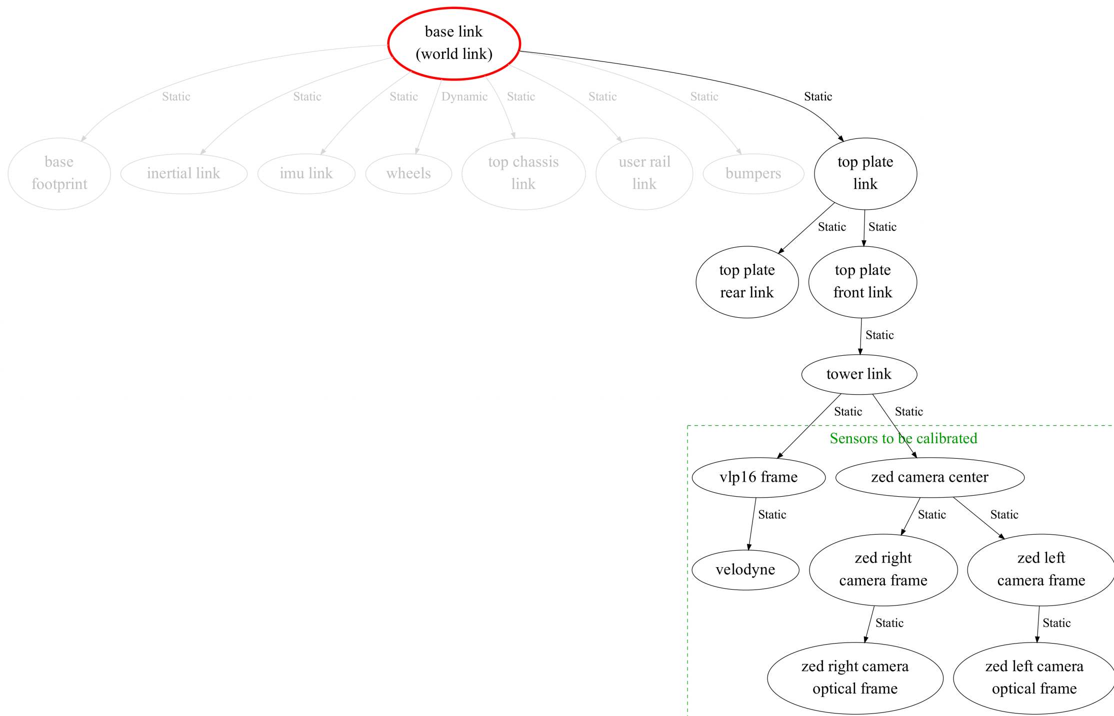


Figure 5.7: Transformations Tree of the Agrob robotic system.

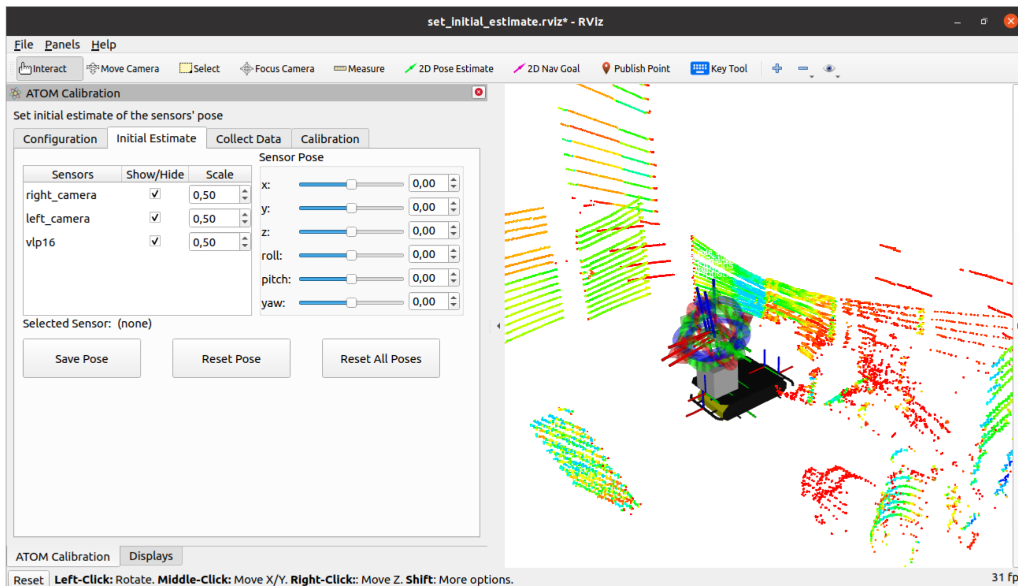


Figure 5.8: Agrob robot model represented in the RViz environment with a calibration being performed using the ATOM interface.

Overall, the tests using the presented robotic systems were deemed to be successful, with the interface allowing a smoother calibration, providing a more interactive and user-friendly procedure, as it was intended in the start of this dissertation.

5.3 Comparison of the interface with the HandEye Calibration GUI from MoveIt!

In order to further validate the usability of the plugin created in this work, this section was thought out to compare the calibration experience between the developed interface and the HandEye Calibration interface from MoveIt! that was already addressed in Section 2.2.

To do that, it was attempted to create similar conditions for both calibrations, so that the only variable to differ would be the interface being used.

However, despite several attempts, a calibration using the HandEye plugin could not be carried out until the end, due to the fact that MoveIt! does not provide enough information nor material to test the interface in a simulated environment and there were no possible conditions to use a real robotic arm for these tests.

Nonetheless, in the midst of the several attempts made and going through all the available information and discussions around this interface, it was possible to get a good idea of the experience using it.

For the attempts made, the robotic system used was the eihbot. The Eye In Hand Bot (eihbot) is a simulated robot that was created specifically for these tests. This robotic system, shown in Figure 5.9, was obtained by removing two sensors of the previously used mmtbot (the LiDAR sensor and the world camera), leaving just the hand camera, which is the camera on the end effector of the robot.

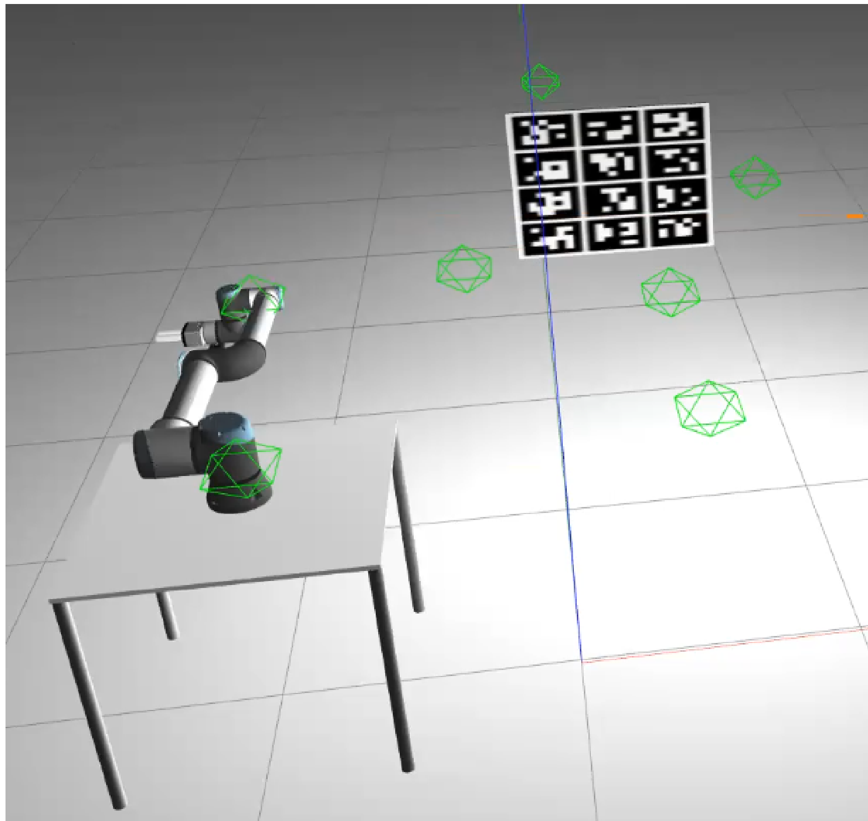


Figure 5.9: Representation of the eihbot robot model.

This was done with the intent of having a robotic arm with just one camera to be able to perform a hand eye calibration. For that, a calibration pattern like the one shown in the above figure was created and positioned in a way that the camera in the robotic arm could see it. Following that, using the Motion Planning plugin from MoveIt!, as depicted in Figure 5.10, the robotic arm could be moved around while recording the data from its camera in a bag file.

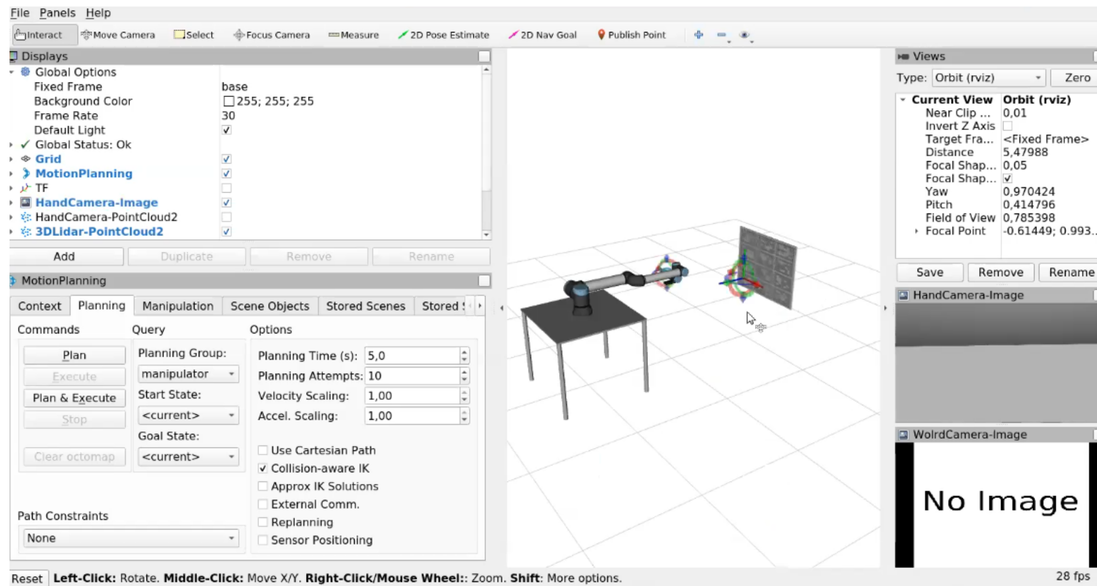


Figure 5.10: Using the Motion Planning plugin from MoveIt! to move the eihbot and record the camera image data.

This bag file, containing the data from the camera of the robotic arm, could then be played back and a calibration using both interfaces could start.

For the ATOM interface, the experience of calibrating is no different from all the robots previously used (mmtbot, ATLASCAR2 and Agrob), having to go through the same steps already explained in Chapter 4, with the only change being this robot only having one sensor in the panel to configure.

For the HandEye Calibration plugin, as already mentioned, MoveIt! does not provide the ability to fully calibrate a simulated robot with their interface, although it is still possible to go through some of its steps. The full calibration procedure using this panel and all the necessary steps to perform a calibration were already thoroughly explained in Section 2.2 and can be seen online in the provided tutorials[3].

Based on the attempts that were made to perform the calibration and the calibrations[4] and discussions[5] that are publicly available online from different users of this panel reporting on their experience, it was possible to compare this interface with the one developed in this work.

[3] `https://ros-planning.github.io/moveit_tutorials/doc/hand_eye_calibration/hand_eye_calibration_tutorial.html`

[4] `https://www.youtube.com/watch?v=xQ79ysnrzUk&t=451s&ab_channel=PickNikRobotics`

[5] `https://github.com/ros-planning/moveit/issues/1070?fbclid=IwAR18Iwtxd77-cwJfnVUULTfJqK_ml-5j32K0pHFaWEn9UVMUISDL2E6CxI4`

Firstly, even before starting to compare what each panel offers, it is important to reinforce the lack of enough tools necessary to perform a full calibration using the Hand-Eye interface. Even though MoveIt! does provide some robot packages, it could help to provide some test bag files with already recorded data and the possibility to perform the calibration in simulated environments until the end, which is something that ATOM does offer.

As for the panels, the way the configuration of the calibration pattern is done in the HandEye Calibration plugin is not very practical, because the configuration is fully made in the panel. This means that whenever the user wants to launch a new system or if the panel is closed, all the configured parameters would be lost and would have to be set again. The way the ATOM interface performs this configuration is by loading the configuration from a yaml file that is inside the ROS package of the robotic system. All the parameters inside that file can then be modified within the panel and the file could update them accordingly. This way, all the parameters that are set in that file remain intact even when the panel is closed or the system is relaunched. One interesting detail that the HandEye panel does offer in detriment of the ATOM panel is the possibility to see and save the outlook of the calibration target that is being configured, although it is a functionality that is not working flawlessly (it is only able to show the design of Aruco boards, even though there is also an option to choose Charuco boards).

The yaml configuration file of the ATOM framework, mentioned above, is not exclusively for the configuration of the pattern used, but rather for the configuration of the entire calibration parameters, having several other important information, such as the bag file with the sensors' data, the main link of the transformations tree, information about the frames of each sensor, and so on. This seems to be more intuitive, since the entire configuration is grouped in only one tab, unlike the HandEye Calibration panel, that only does some of this other configurations, and they are done in a different tab widget (the second one).

Lastly, in the HandEye Calibration plugin, the calibration and collection of data was done in the same tab widget, having a lot happening simultaneously. Plus, there were a few ambiguous elements in this part of the panel, with the main one being the actual calibration, since users would most likely not be able to figure out by themselves what button of the panel would initiate the calibration without having to check the tutorials. The ATOM plugin separates the data collection phases, each offering more functionalities than the other panel and with a clearer arrangement, making it much more intuitive to use.

Overall, both interfaces presented a nice calibration experience. However, even though the panel from MoveIt! served as a good baseline for this work, the plugin developed in this dissertation does present a more natural flow of the calibration procedure, with each tab addressing a specific goal and with more functionalities in comparison to the HandEye Calibration panel. This makes for an easier and more user-friendly usage of the RViz plugin. Adding to that, the ATOM interface is also more broad when it comes to the robotic systems that can use it, due to the fact of it being able to calibrate systems with multiple sensors of different modalities, not being exclusively for hand-eye calibrations.

Chapter 6

Conclusions

The main purpose of this work was to solve the problem of the lack of an interface in the ROS Visualization tool (RViz) for the ATOM Calibration system. Like the majority of the existing calibration systems in ROS, whose calibration procedures are executed with the help of just the terminal or by resourcing to RViz in ways that might not be as straightforward for most people, ATOM did not provide a simple and user-friendly graphical interface to help with this process.

Therefore, this work aimed to explore the full potential of RViz for the ATOM Calibration framework, by creating an interface that could be added to the ROS Visualization tool as a plugin, that would help during the entire process of the calibration.

Having that ambition in mind, the developed work was successful, since this plugin is able to provide an interactive experience that facilitates the calibration process for the users of the ATOM framework.

The usability of this interface was also validated through its ability of being used to calibrate the sensors of several robotic systems, provided that the installation of ATOM and configuration of the robotic system's ROS package are done correctly. A calibration using other existing interfaces was also conducted, with the ATOM interface presenting several advantages and improvements in comparison to them.

The ROS Package of the interface developed in this work, as well as the ROS Package of the ATOM Calibration Framework, are publicly available at:

- ATOM RViz Plugin: `https://github.com/lardemua/atom_rviz_plugin`
- ATOM Calibration Framework: `https://github.com/lardemua/atom`

Below is a list of videos of the ATOM interface being used for the calibration of the simulated robot that was mainly used throughout this dissertation:

- Configuration of Calibration Parameters: `https://youtu.be/KSUbFCFNd08`
- Initial Estimate: `https://youtu.be/VtHhpO57Sgo`
- Data Collection: `https://youtu.be/ZOnPDOnsc5w`
- Calibration: `https://youtu.be/LOdk748x_lM`

6.1 Future Work

Although the main goal of this dissertation, as well as the inherent milestones, were satisfactorily achieved, there is still room for improvement in this plugin.

First of all, further tests should be conducted. Although the interface was indeed tested with a few robotic systems, running more tests with different types of robotic systems with as many sensors as possible could be a good way to fully assure its efficiency. Additionally, since the tests were only made internally, there was no outside feedback of this interface. Therefore, allowing other people to try to use the created RViz plugin to calibrate their own robotic systems could also help validate this work even further.

As for improvements in the interface, in the last tab widget (Calibration tab), the table containing the parameters that will create the command-line to run the calibration is divided in two categories: string parameters and boolean parameters. The boolean parameters are the ones with a checkbox and they are the ones that do not require any additional information, the user either wants to run the calibration with them or not. On the other hand, the string parameters require an additional argument, and a nice improvement for the interface would be to personalize most of these additional arguments. For instance, the '-json' argument requires the path of the json file with all the information regarding the collected data of the sensors. As for the '-csf' argument, it requires a string with a specific syntax to be added that selects a group of collections to be used for the optimization. As of now, these strings are required to be written by hand. However, it would be nice to have, for example, a file dialog box to search for the file to be used by the '-json' argument and a pair of spin boxes for the '-csf' argument to define the minimum and maximum number of the collections' interval to be used in the optimization procedure, with the necessary string being created according to the information from those objects.

References

[1] R. D. Atkinson, "Robotics and the Future of Production and Work," tech. rep., Information Technology and Innovation Foundation, Oct. 2019.

[2] F. M. Mirzaei, "Extrinsic and intrinsic sensor calibration," Dec. 2013. Accepted: 2014-02-12T14:36:06Z.

[3] E. Pedrosa, M. Oliveira, N. Lau, and V. Santos, "A General Approach to Hand–Eye Calibration Through the Optimization of Atomic Transformations," *IEEE Transactions on Robotics*, pp. 1–15, 2021. Conference Name: IEEE Transactions on Robotics.

[4] A. S. Pinto de Aguiar, M. A. Riem de Oliveira, E. F. Pedrosa, and F. B. Neves dos Santos, "A Camera to LiDAR calibration approach through the optimization of atomic transformations," *Expert Systems with Applications*, vol. 176, p. 114894, Aug. 2021.

[5] M. Oliveira, A. Castro, T. Madeira, E. Pedrosa, P. Dias, and V. Santos, "A ROS framework for the extrinsic calibration of intelligent vehicles: A multi-sensor, multimodal approach," *Robotics and Autonomous Systems*, vol. 131, p. 103558, Sept. 2020.

[6] G. Mueller and H.-J. Wünsche, "Continuous stereo camera calibration in urban scenarios," *undefined*, 2017.

[7] L. Wu and B. Zhu, "Binocular stereovision camera calibration," in *2015 IEEE International Conference on Mechatronics and Automation (ICMA)*, pp. 2638–2642, Aug. 2015. ISSN: 2152-744X.

[8] R. Su, J. Zhong, Q. Li, S. Qi, H. Zhang, and T. Wang, "An automatic calibration system for binocular stereo imaging," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 896–900, Oct. 2016.

[9] Y. Ling and S. Shen, "High-precision online markerless stereo extrinsic calibration," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1771–1778, Oct. 2016. ISSN: 2153-0866.

[10] V. Q. Dinh, T. P. Nguyen, and J. W. Jeon, "Rectification Using Different Types of Cameras Attached to a Vehicle," *IEEE Transactions on Image Processing*, vol. 28, pp. 815–826, Feb. 2019. Conference Name: IEEE Transactions on Image Processing.

[11] F. Vasconcelos, J. P. Barreto, and U. Nunes, "A Minimal Solution for the Extrinsic Calibration of a Camera and a Laser-Rangefinder," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, pp. 2097–2107, Nov. 2012. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[12] M. Pereira, D. Silva, V. Santos, and P. Dias, "Self calibration of multiple LIDARs and cameras on autonomous vehicles," *Robotics and Autonomous Systems*, vol. 83, pp. 326–337, Sept. 2016.

[13] M. Almeida, P. Dias, M. Oliveira, and V. Santos, "3D-2D Laser Range Finder Calibration Using a Conic Based Geometry Shape," in *Image Analysis and Recognition* (A. Campilho and M. Kamel, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 312–319, Springer, 2012.

[14] C. Guindel, J. Beltrán, D. Martín, and F. García, "Automatic extrinsic calibration for lidar-stereo vehicle sensor setups," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–6, Oct. 2017. ISSN: 2153-0017.

[15] A. Geiger, F. Moosmann, O. Car, and B. Schuster, "Automatic camera and range sensor calibration using a single shot," in *2012 IEEE International Conference on Robotics and Automation*, (St Paul, MN, USA), pp. 3936–3943, IEEE, May 2012.

[16] Y. Liao, G. Li, Z. Ju, H. Liu, and D. Jiang, "Joint kinect and multiple external cameras simultaneous calibration," in *2017 2nd International Conference on Advanced Robotics and Mechatronics (ICARM)*, pp. 305–310, Aug. 2017.

[17] J. Rehder, R. Siegwart, and P. Furgale, "A General Approach to Spatiotemporal Calibration in Multisensor Systems," *IEEE Transactions on Robotics*, vol. 32, pp. 383–398, Apr. 2016. Conference Name: IEEE Transactions on Robotics.

[18] V. Pradeep, K. Konolige, and E. Berger, "Calibrating a Multi-arm Multi-sensor Robot: A Bundle Adjustment Approach," in *Experimental Robotics: The 12th International Symposium on Experimental Robotics* (O. Khatib, V. Kumar, and G. Sukhatme, eds.), Springer Tracts in Advanced Robotics, pp. 211–225, Berlin, Heidelberg: Springer, 2014.

[19] P. Cieślak, "Stonefish: An Advanced Open-Source Simulation Tool Designed for Marine Robotics, With a ROS Interface," in *OCEANS 2019 - Marseille*, pp. 1–6, June 2019.

[20] Y. Hold-Geoffroy, M.-A. Gardner, C. Gagné, M. Latulippe, and P. Giguère, "ros4mat: A Matlab Programming Interface for Remote Operations of ROS-Based Robotic Devices in an Educational Context," in *2013 International Conference on Computer and Robot Vision*, pp. 242–248, May 2013.

[21] R. L. Avanzato, "Development of a MATLAB/ROS Interface to a Low-cost Robot Arm," June 2020.

[22] D. Zolett and A. Ramirez, "Desenvolvimento de uma Interface de Monitoração Remota para o Sistema Robótico ROBIX, Integrando o Protocolo MQTT e oROS," Sept. 2020. Pages: 412.

[23] L. Pfotzer, J. Oberlaender, A. Roennau, and R. Dillmann, "Development and calibration of KaRoLa, a compact, high-resolution 3D laser scanner," in *2014 IEEE International Symposium on Safety, Security, and Rescue Robotics (2014)*, pp. 1–6, Oct. 2014. ISSN: 2374-3247.

[24] I. Rodriguez, A. Astigarraga, E. Jauregi, T. Ruiz, and E. Lazkano, "Humanizing NAO robot teleoperation using ROS," in *2014 IEEE-RAS International Conference on Humanoid Robots*, pp. 179–186, Nov. 2014. ISSN: 2164-0580.

[25] R. Kaestner, "Rqt plugin for visualizing numeric values in multiple 2d plots," May 2021. `https://github.com/ANYbotics/rqt_multiplot_plugin` Accessed: 2021-05-24.

[26] Franz, "Rqt plugin for ros to control turtles in turtlesim," May 2021. `https://github.com/fjp/rqt-turtle` Accessed: 2021-05-24.

[27] R. Navarro, "Rqt plugin to control and monitor the barrethand," Mar. 2021. `https://github.com/RobotnikAutomation/barrett_hand` Accessed: 2021-05-24.

[28] Y. Nasri, E. D. Rouvray, V. Vauchey, E. D. Rouvray, C. R. France, R. Khemmar, E. D. Rouvray, C. R. France, N. Ragot, E. D. Rouvray, C. R. France, K. Sirlantzis, and J. Building, "ROS-based Autonomous Navigation Wheelchair using Omnidirectional Sensor,"

[29] S. S. Velamala, D. Patil, and X. Ming, "Development of ROS-based GUI for control of an autonomous surface vehicle," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 628–633, Dec. 2017.

[30] S. Chitta, I. Sucan, and S. Cousins, "MoveIt! [ROS Topics]," *IEEE Robotics Automation Magazine*, vol. 19, pp. 18–19, Mar. 2012. Conference Name: IEEE Robotics Automation Magazine.

[31] S. Meng, Y. Liang, and H. Shi, "The Motion Planning of a Six DOF Manipulator Based on ROS Platform," *Shanghai Jiaotong Daxue Xuebao/Journal of Shanghai Jiaotong University*, vol. 50, pp. 94–97, July 2016.

[32] C. P. Quintero, O. Ramirez, and M. Jägersand, "VIBI: Assistive vision-based interface for robot manipulation," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4458–4463, May 2015. ISSN: 1050-4729.

[33] H. Deng, J. Xiong, and Z. Xia, "Mobile manipulation task simulation using ROS with MoveIt," in *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pp. 612–616, July 2017.

[34] R. K. Megalingam, N. Katta, R. Geesala, P. K. Yadav, and R. C. Rangaiah, "Keyboard-Based Control and Simulation of 6-DOF Robotic Arm Using ROS," in *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, pp. 1–5, Dec. 2018. ISSN: 2642-7354.

[35] S. Hernandez-Mendez, C. Maldonado-Mendez, A. Marin-Hernandez, H. V. Rios-Figueroa, H. Vazquez-Leal, and E. R. Palacios-Hernandez, "Design and implementation of a robotic arm using ROS and MoveIt!," in *2017 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*, pp. 1–6, Nov. 2017. ISSN: 2573-0770.

[36] L. Chen, Z. Wei, F. Zhao, and T. Tao, "Development of a virtual teaching pendant system for serial robots based on ROS-I," in *2017 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pp. 720–724, Nov. 2017. ISSN: 2326-8239.

[37] M. Esposito, "Easy-handeye: automated, hardware-independent hand-eye calibration," May 2021. `https://github.com/IFL-CAMP/easy_handeye` Accessed: 2021-05-26.

[38] J. Bowman, "Intrinsic calibration of monocular or stereo cameras." `http://library.isr.ist.utl.pt/docs/roswiki/camera_calibration.html` Accessed: 2021-05-26.

[39] C. Lewis, "industrial_extrinsic_cal - ROS Wiki." `http://wiki.ros.org/industrial_extrinsic_cal` Accessed: 2021-05-26.

[40] Y. Yan, "ros-planning/moveit_calibration," May 2021. original-date: 2020-06-02T14:59:11Z.

[41] J. Meyer, "Robot Calibration Tools," May 2021. `https://github.com/Jmeyer1292/robot_cal_tools` Accessed: 2021-05-26.

[42] A. Hundt and F. Jonathan, "ROS + CamOdoCal Hand Eye Calibration," May 2021. `https://github.com/jhu-lcsr/handeye_calib_camodocal` Accessed: 2021-05-26.

[43] F. Furrer, M. Fehr, T. Novkovic, I. Gilitschenski, and R. Siegwart, "Hand-Eye-Calibration," May 2021. `https://github.com/ethz-asl/hand_eye_calibration` Accessed: 2021-05-26.

[44] F. S. Ruiz, "handeye," Apr. 2021. `https://github.com/crigroup/handeye` Accessed: 2021-05-26.

[45] A. Dhall, "LiDAR-Camera Calibration using 3D-3D Point correspondences," May 2021. `https://github.com/ankitdhall/lidar_camera_calibration` Accessed: 2021-05-26.

[46] Y. Shiu and S. Ahmad, "Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form AX=XB," *IEEE Transactions on Robotics and Automation*, vol. 5, pp. 16–29, Feb. 1989. Conference Name: IEEE Transactions on Robotics and Automation.

[47] R. Tsai and R. Lenz, "A new technique for fully autonomous and efficient 3D robotics hand/eye calibration," *IEEE Transactions on Robotics and Automation*, vol. 5, pp. 345–358, June 1989. Conference Name: IEEE Transactions on Robotics and Automation.

[48] A. Tabb and K. M. A. Yousef, "Solving the Robot-World Hand-Eye(s) Calibration Problem with Iterative Methods," *Machine Vision and Applications*, vol. 28, pp. 569–590, Aug. 2017. arXiv: 1907.12425.

[49] A. Li, L. Wang, and D. Wu, "Simultaneous robot-world and hand-eye calibration using dual-quaternions and Kronecker product," *International journal of physical sciences*, vol. 5, Sept. 2010.

[50] A. Castro, *Multi-modal sensor calibration on board the ATLASCAR2.* PhD thesis, 2019. Accepted: 2020-08-27T09:02:55Z.

[51] A. Martinez and E. Fernández, *Learning ROS for Robotics Programming.* Packt Publishing Ltd, Sept. 2013. Google-Books-ID: 2ZL9AAAAQBAJ.

[52] L. Joseph and J. Cacace, *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System.* Packt Publishing Ltd, 2018.

[53] C. Fairchild and D. T. Harman, *ROS Robotics By Example.* Packt Publishing, June 2016.

[54] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," p. 6.

[55] D. Chikurtev, I. Rangelov, N. Chivarov, E. Markov, and K. Yovchev, "Control of Robotic Arm Manipulator Using ROS," p. 10.

[56] D. Gossow, A. Leeper, D. Hershberger, and M. Ciocarlie, "Interactive Markers: 3-D User Interfaces for ROS Applications," *IEEE Robot. Automat. Mag.*, vol. 18, pp. 14–15, Dec. 2011.