Master's thesis

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

Henning L. Frestad

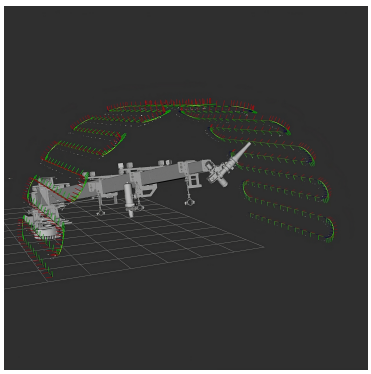
Software Development for a 5-DOF Shotcrete Robot Using ROS, MoveIt, and Gazebo

Master's thesis in Cybernetics and Robotics
Supervisor: Anton Shiriaev
Co-supervisor: Konstantinos Alexis
June 2023

NTNU
Norwegian University of
Science and Technology

Henning L. Frestad

Software Development for a 5-DOF Shotcrete Robot Using ROS, MoveIt, and Gazebo



Master's thesis in Cybernetics and Robotics
Supervisor: Anton Shiriaev
Co-supervisor: Konstantinos Alexis
June 2023

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

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis offers an exploration of the Robot Operating System (ROS) as a viable option for AMV in developing robotic software for their portfolio of machines. It specifically documents the process of implementing ROS for their 4200 Shotcrete robot. The objective is to provide a comprehensive knowledge base that enables AMV to make an informed decision regarding the adoption of ROS for their robotic software development.

The AMV 4200 Shotcrete vehicle is designed for spraying shotcrete on rocky surfaces. It is equipped with a 5-degree-of-freedom (DOF) robotic arm, originally designed to be operated manually. However, AMV has initiated development to automate the spraying operation, enhancing efficiency and quality.

A comprehensive introduction to ROS is included in this report, focusing on its relevant features for the AMV 4200 Shotcrete. It explores the integration of MoveIt and Gazebo, which complement ROS by offering essential functionalities. MoveIt provides tools for motion planning and kinematics, while Gazebo serves as a simulation environment. A robot model was created based on the design files of the AMV 4200 Shotcrete, encompassing the necessary properties for simulation, kinematics, and motion planning. The successful configuration and initial testing of MoveIt and Gazebo with the AMV 4200 Shotcrete have been documented, accompanied by a discussion on overcoming practical challenges associated with utilizing ROS, Gazebo, and MoveIt for this particular robot.

The motion planning capabilities of MoveIt were tested with a typical use case for the AMV 4200 Shotcrete, and the results demonstrated great promise. Several useful features of Gazebo have been presented and discussed, but not enough time was set aside to test them with the AMV 4200 Shotcrete.

Based on the results and accumulated experience, the utilization of ROS, MoveIt, and Gazebo has the potential to substantially reduce the development time required to create high-quality robotics software for AMV.

Sammendrag

Denne avhandlingen utforsker Robot Operating System (ROS) som et alternativ for utviklingen av robotprogramvare for AMV sin maskinporteføljе. Den dokumenterer spesifikt prosessen med å implementere ROS for 4200 Shotcrete roboten deres. Målet er å tilby et omfattende kunnskapsgrunnlag som gjør det mulig for AMV å ta en informert beslutning angående bruk av ROS for utviklingen av deres robotprogramvare.

AMV 4200 Shotcrete er designet for sprøyting av sprøytebetong på bergoverflater. Den er utstyrt med en robotarm med 5 frihetsgrader som opprinnelig er designet for manuell styring. Imidlertid har AMV startet utviklingen av systemer som skal automatisere sprøyteoperasjonen for å øke effektiviteten og kvaliteten.

Avhandlingen inneholder en grundig introduksjon til ROS med spesifikt fokus på relevante funksjoner for AMV 4200 Shotcrete. Implementeringen av MoveIt og Gazebo blir utforsket som komplementære verktøy til ROS. MoveIt tilbyr verktøy for bevegelsesplanlegging og kinematikk, mens Gazebo fungerer som et simuleringmiljø. Det ble laget en robotmodel basert på designfilene til AMV 4200 roboten. Den omfatter egenskaper til roboten som er nødvendig for simulering, kinematikk og bevegelsesplanlegging. Konfigurasjonen og testing av MoveIt og Gazebo med AMV 4200 er dokumentert, og praktiske utfordringer knyttet til bruk av ROS, Gazebo og MoveIt for denne spesifikke roboten er diskutert.

Bevegelsesplanleggingen i MoveIt ble testet med en typisk sprøyteopreasjon, og resultatene var svært lovende. Flere nyttige funksjoner i Gazebo ble presentert og diskutert, men det ble ikke tid til grundig testing av disse funksjonene med AMV 4200 roboten.

Basert på resultatene og opparbeidet erfaring på bruken av ROS, MoveIt og Gazebo, har disse verktøyene potensial til å drastisk redusere tiden som kreves for å utvikle robotprogramvare av høy kvalitet.

Acknolegments

The completion of this project marks the end of my 2-year master's program in cybernetics and robotics at NTNU. I would like to thank NTNU and my lecturers for equipping me with the knowledge needed to call myself an engineer. I also want to thank my peers for making my time at NTNU truly memorable.

I extend my sincere thanks to AMV for providing me with this assignment, and to my supervisor from AMV, Jonas Tønnessen, for his continuous availability and valuable guidance. I am delighted to know that this thesis will contribute value to the industry, and grateful for the experience I have gained throughout the project.

I am also grateful to the dedicated developers who voluntarily contribute to the open-source projects of ROS, MoveIt, and Gazebo. Their valuable contributions have greatly enhanced the functionality and usability of these platforms. I would like to extend a special thanks to Josh Newans of the ROS community for his exceptional tutorials.

Contents

List of Figures

List of Tables

Acronyms

AMV	Andersens Mekaniske Verksted AS
DOF	Degrees of Freedom
LiDAR	Light Detection and Ranging
PLC	Programmable Logic Controller
ROS	Robot Operating System
SDF	Simulation Description Format
URDF	Unified Robotics Description Format

Glossary

joint space	a space that encompasses all valid angular or translational displacements for each joint of the robot.
motion planning	"Motion planning is the process by which you define the set of actions that you need to execute to follow the path you planned. We consider things such as the speed at which we want the robot to move, if we need to speed up or slow down, how to approach a change in direction, etc." [1]
path	describes a route from A to B in task space or configuration space "with position and orientation information, but without timing considerations, i.e. without considering velocities and accelerations"[2] along the route.
path planning	"to determine a path in task space (or joint space) to move the robot to a goal position while avoiding collisions with objects in its workspace." [2]
task space	the space in which the task of the robot can be naturally expressed [3]. Position and orientation parameters that describe a transformation between the robots base frame and the desired end-effectors frame.
trajectory	"determine the time history of the manipulator along a given path or between initial and final configurations."[2]

Chapter 1

Introduction

1.1 Background

The project problem is provided by Andersens Mekaniske Verksted AS (AMV). AMV was founded in 1860 by Gabriel T. Andersen in Flekkefjord, Norway. At the time of writing, the company has roughly 235 employees, where the vast majority are based in Flekkefjord. AMV design, manufacture and deliver a range of large machines for the tunneling and mining industry, among them is the 4200 Shotcrete Robot which plays a pivotal role in this project [4]. The robot is designed to spray shotcrete onto rocky surfaces, with the purpose of binding, smoothing, and securing the surface [5]. Shotcrete is concrete that is being sprayed at high velocity onto a surface. The hardening properties of conventional concrete and shotcrete are similar. However, the high velocity makes the shotcrete bond to the surface almost instantly [6]. The long and sturdy telescopic boom of the AMV 4200 Shotcrete Robot makes it ideal for applying shotcrete to the inside of tunnels and mines. Illustrations of the robot can be seen in Figure 1.1. A remote controller can be used to manually control the robotic arm at the back of the truck [5]. AMV are developing means of automating the robotic arm movement, and this project is designed to help to achieve that objective.

Figure 1.1: The AMV 4200 Shotcrete Robot has a robotic arm mounted at the back of a trailer. Shotcrete is sprayed from the end effector of the robotic arm. Illustrations made by AMV [5].

1.1.1 Rationale for automation

The decision to automate the 4200 Shotcrete is motivated by the expectation that it will result in a more efficient and predictable operation of the spraying robot. This is relevant as the results of the spraying operation can be substantially influenced by the operator's skill set. The primary objective is to achieve a more uniform and consistent concrete thickness, that is closer to the desired thickness, compared to manual operation. This would reduce the amount of concrete required to secure and level the tunnel surface, thereby decreasing concrete production and the associated CO2 emissions. Additionally, the integration of robot control and camera systems could enable the operator to control the machine from the drives cabin, thereby eliminating the need to be in hazardous areas. According to AMV, increased automation in the 4200 Shotcrete robot is well-received by their current and potential customers.

1.1.2 Status of the automation project

Currently, AMV is developing a robotic motion control (RMC) system for the 4200 Shotcrete on a Beckhoff Programmable Logic Controller (PLC). Modes that replicate common driving patterns of operators have been created. To support these modes, algorithms for common robotic applications have been built from scratch, including path planning, kinematics, setpoint generation, and controllers. Furthermore, other fundamental functionalities, such as communication, alarm systems, and module management have also been developed from the ground up. Hopefully, the system is ready for sale, within a few months. Possibilities of using a Light Detection and Ranging (LiDAR) and a camera are being explored as additional features. Implementing the feedback of these types of sensors is outside the project scope. Nonetheless, the project report will include discussions of how ROS can be utilized to incorporate LiDAR and camera data.

1.1.3 Why Robot Operating System (ROS)?

AMV's engineers have recognized the potential of utilizing ROS for the development of their robotic systems and sought to investigate its potential for automating their robots. The use of Beckhoff PLC presents certain limitations that could potentially be addressed by implementing ROS. The arguments presented below are based on the AMV engineers' experience with the Beckhoff PLC and their impression of ROS before the project started.

Firstly, all the software has been developed from scratch in the Beckhoff PLC because few libraries for robotics are available. In contrast, ROS and its active community offer numerous software packages and tools that could be leveraged for common robotic applications, potentially leading to substantial savings in development time. Although the system is ready for sale soon on the 4200 robot, this argument is relevant because AMV wishes to develop equivalent systems for similar machines in their portfolio, and are considering to continue the development of the 4200 robot with ROS.

Secondly, the engineers at AMV would prefer to use a widely used programming language like C++ for software development, which is compatible with ROS. In contrast, PLCs require specialized programming languages that are exclusive to PLCs. This creates difficulties for developers who are not familiar with these programming languages. Given that there is a significantly larger pool

of C++ developers compared to Beckhoff developers, finding and hiring experienced developers is likely to be easier.

Furthermore, obtaining support for Beckhoff programming can be challenging due to the small size of the community. In contrast, ROS has a vast, active community of contributors and users, with forums providing a valuable platform for troubleshooting and knowledge-sharing.

1.2 Thesis scope

The master project can be described as a feasibility study, where the purpose is to answer the following problem statement:

> **Problem statement:**
> Should the engineers at AMV use the Robot Operating System as the foundation for software development on the AMV 4200 Shotcrete robot and other robots in their portfolio?

The problem statement is very open, this is mostly due to the limited prior knowledge about ROS among the AMV engineers and the author. However, after acquiring a fundamental understanding of ROS, the following project goals were established in collaboration with Jonas Tønnessen (supervisor from AMV):

- Integrate the AMV 4200 Shotcrete robot with Gazebo, a physics-based simulator.
- Implement motion planning algorithms for the AMV 4200 Shotcrete using MoveIt, a motion planning framework.

These goals were selected based on their potential to bring significant value to the automation project for the AMV 4200 Shotcrete. By pursuing these objectives, the aim is to gain valuable experience in overcoming the challenges associated with implementing ROS, MoveIt, and Gazebo. The knowledge and insights gained from this endeavor are expected to be relevant not only to the development of ROS-based software for the AMV 4200 Shotcrete but also to other robots in AMV's portfolio. In addition, the software developed during this thesis will be shared with AMV and provide a building block for further development if they decide to use ROS.

1.3 Report outline

This report is tailored for the robotic engineers at AMV. Given their expertise in robotics, this report will not cover robotics theory but rather focus on showcasing the practical implementation of ROS in robot applications, specifically highlighting the use of ROS in the context of the AMV 4200 Shotcrete robot. While detailed code implementation is not covered extensively, the report provides valuable insights and guidance for using ROS and related software for AMV's robot applications.

- **Chapter 2** - *Robot Description* - offers a robotics-focused overview of the AMV 4200 Shotcrete and its spraying operation.
- **Chapter 3** - *Introduction to the software frameworks* - provides a comprehensive introduction to the relevant software tools for the project, including ROS, Gazebo, and MoveIt.
- **Chapter 4** - *Software Implementation* - provides insight into the implementation process for motion control of the AMV 4200 Shotcrete robot. The chapter is structured into sections, each focusing on a specific aspect of the implementation, from the installation of ROS to the integration of motion planning algorithms using MoveIt.
- **Chapter 5** - *Results* - prensent the motion planning capabilities of the final software.
- **Chapter 6** - *Discussion and Future Work* - analyzes the results from the previous chapter, shares the author's user experience with ROS and MoveIt, and identifies future research directions.
- **Chapter 7** - *Conclusion* - Finally, an answer to the problem statement based on the results and experience gathered throughout the project.

Chapter 2

Robot Description

This chapter provides an overview of the AMV 4200 Shotcrete and its spraying operation from a robotics standpoint. It will include a discussion on the task space, and briefly touch upon analytical solutions for kinematics, although a detailed analysis falls outside the scope of this thesis. The aim is to provide the reader with a sufficient understanding of the robot, enabling a better comprehension of the subsequent chapters. Note that the vehicle is stationary during the spraying operation, hence the machine can be interpreted as a robot arm manipulator.

2.1 The task space

For the task of applying shotcrete, only the direction of the nozzle is relevant, not the full orientation. When adding the positional coordinates as well, the task space can be described with 3 coordinates and 2 rotation angles. This can be shown by inspecting the homogeneous transform matrix,

$$T = \begin{pmatrix} \mathbf{R} & \mathbf{d} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{2.1}$$

where the rotation matrix **R** and the translation vector **d** define the orientation and position of the end-effector respectively, relative to the base frame. By defining the rotation matrix as a sequence of rotation about z, then y, then x we get the following transfrom matrix:

$$T = \begin{pmatrix} \cos(\alpha_z)\cdot\cos(\alpha_y) & \cos(\alpha_z)\cdot\sin(\alpha_y)\cdot\sin(\alpha_x) & -\sin(\alpha_z)\cdot\cos(\alpha_x) & d_x \\ -\sin(\alpha_z)\cdot\cos(\alpha_y) & -\sin(\alpha_z)\cdot\sin(\alpha_y)\cdot\sin(\alpha_x) & -\cos(\alpha_z)\cdot\cos(\alpha_x) & d_y \\ \sin(\alpha_y) & -\cos(\alpha_y)\cdot\sin(\alpha_x) & \cos(\alpha_y)\cdot\cos(\alpha_x) & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

The elements highlighted in red fully define the x-axis of the new frame, and hence the x-axis is determined by solely α_z and α_y. Assuming that the x-axis represents the nozzle direction, the y- and z-axis (marked green and blue respectively) are not relevant to the task. Hence, there is no need to include α_x in the task space, and the task space can be defined using 2 angles

and 3 position coordinates. Throughout the thesis, the 6-Degrees of Freedom (DOF) space of 3 coordinates and 3 rotation angles will also be relevant and will be referred to as the *full task space*.

2.2 Kinematics

The AMV 4200 was initially developed for manual operations. With its straightforward kinematic structure consisting of 5 individually controlled joints, the machine can be intuitively controlled by an operator. However, due to the absence of the sixth DOF, a general solution to the inverse kinematics problem does not exist when the task space is defined with 6 DOF [7]. Auen and Lyngroth [8] addressed this challenge in their master thesis, attempting to derive a closed-form analytical solution for the robot's inverse kinematics by defining the task space with 5 DOF, following a similar approach as described in section 2.1. However, their efforts were unsuccessful, and no conclusive evidence was presented regarding the absence of a closed-form solution. On the other hand, Auen and Lyngroth successfully addressed the forward kinematics problem and documented their findings in their thesis. In this project, the analytical solutions for the forward and inverse kinematics received minimal attention, as the primary focus was on exploring software-based approaches to address these problems rather than pursuing analytical solutions.

2.3 Joints and actuation

Figure 2.1 provides a visual representation of the joints of the robot. The first joint enables the boom's rotation about the base and is driven by a slewing mechanism actuated by two hydraulic worm gears. Revolution about the Jib axis is achieved through two hydraulic cylinders. The prismatic joint consists of a telescopic arm with two stages. A series of two cylinders actuate the total extension of the boom, in other words, the stages of the telescope cannot be controlled individually. Hydraulic rotary actuators on the final two joints enable control over the roll and pitch of the nozzle.

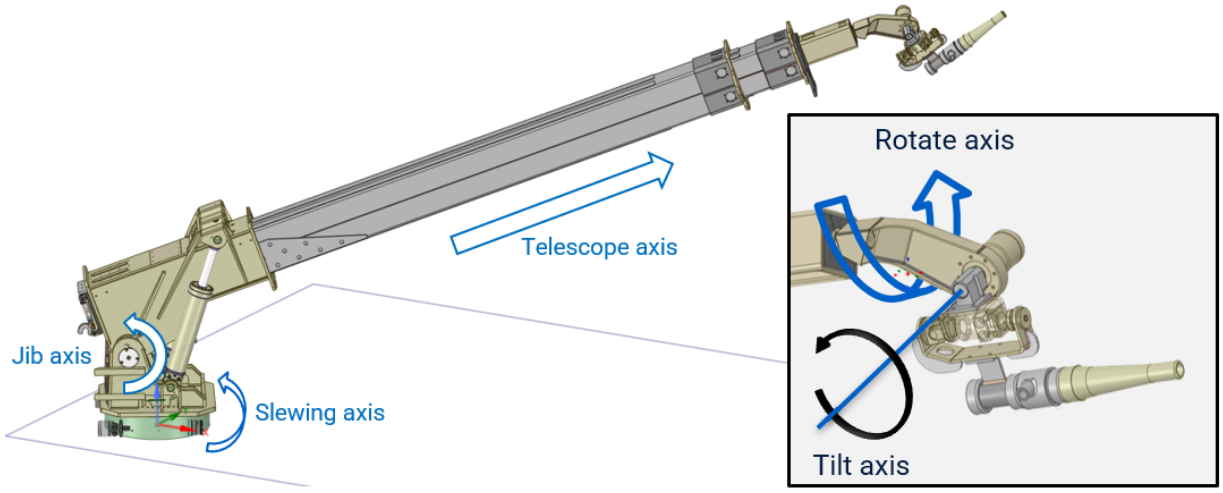
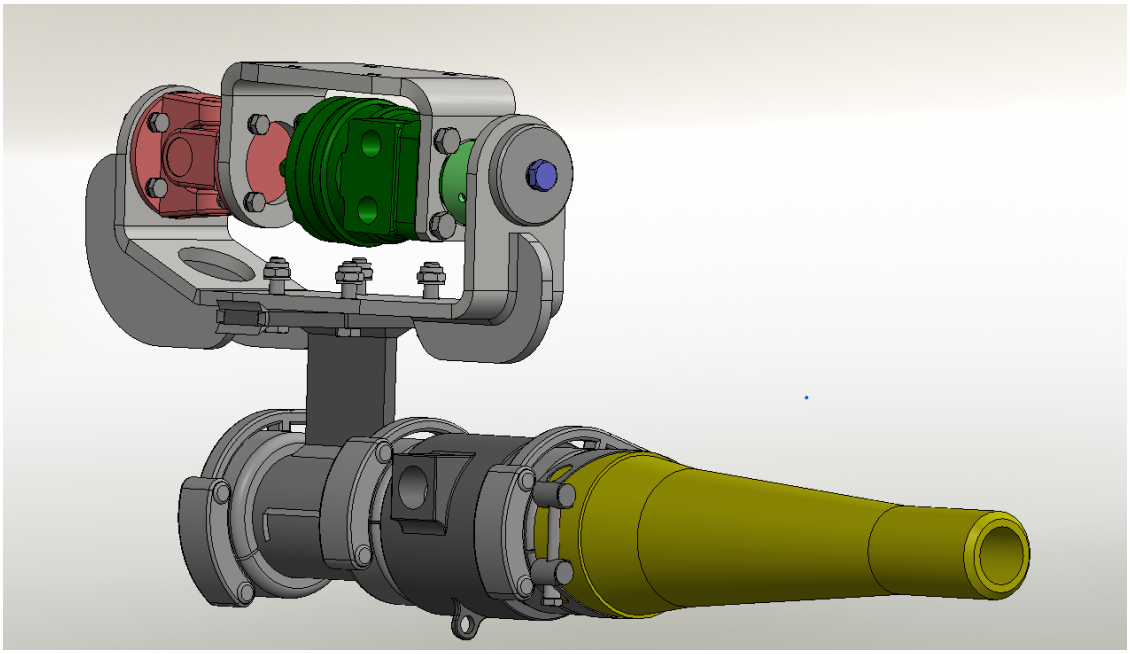


Figure 2.1: AMV 4200 Shotcrete's axis of motion.

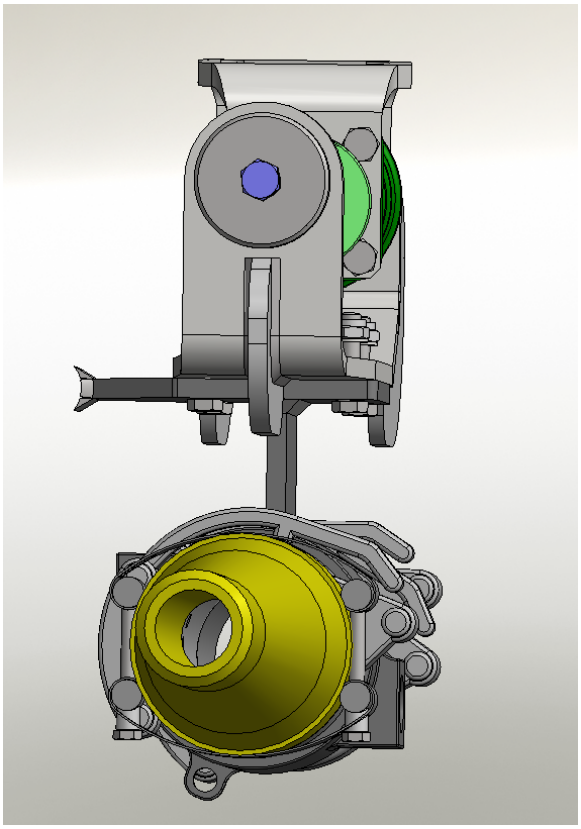
Joint	**Range**
Slewing	[-65, 65]°
Jib	[-16, 57]°
Telescope	[7, 15] m
Rotate	[-180, 180]°
Tilt	[-62, 118]°

Table 2.1: Joint ranges

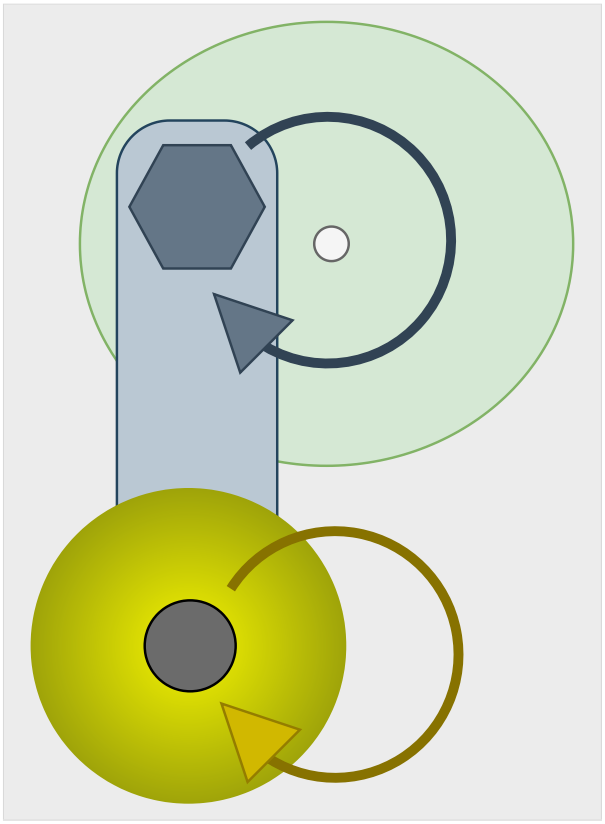
The nozzle is attached to an eccentric rotary mechanism called the "nikker", which ensures that the shotcrete is evenly dispersed on the surface by applying a special periodic motion to the nozzle. Figure 2.2c illustrates that a bolt (blue) is mounted at an offset to the center of the rotor (green). This causes a circular motion to the end of the nozzle. The simplified figure fails to illustrate that the nikker also affects the angle of the nozzle, which can be observed in Figure 2.2b. The nozzle angle will always be tilted outwards, making the overall motion of the nozzle cone-shaped. Thus, shotcrete is spread out more evenly as a result of the motion created by the nikker. Note that this mechanism is not subject to position control, as it does not feature a feedback sensor. Moreover, due to the nature of its function, the nikker is not suitable as a controllable joint.



(a) A side view of the nikker and nozzle.



(b) The nikker and nozzle viewed directly from the front. Notice the angle offset of the nozzle.



(c) A simplified version viewed from the front. The figure illustrates how the eccentric mechanism results in a circular motion for the nozzle tip.

Figure 2.2: The figures show the nikker and nozzle of the 4200 Shotcrete robot and illustrates how the nikker affects the nozzle pose.

Chapter 3

Introduction to the software frameworks

This chapter aims to provide a fundamental understanding of ROS, MoveIt, and Gazebo, covering the basics of these frameworks while emphasizing their potential contributions to the development of a more autonomous AMV 4200 Shotcrete robot.

3.1 Robot Operating System

The Robot Operating System (ROS) is a software development kit specifically designed for building robotic systems. It offers a comprehensive set of tools and libraries that serve as the foundation for developing custom robotics applications. Developed by Open Robotics, a non-profit organization, ROS is freely available and distributed as an open-source platform. This means that anyone can access the source code, contribute to its improvement, and introduce new functionality to enhance its capabilities. Contrary to what the name suggests, ROS is not an operating system and needs an actual operating system to run on top of. Instead, it is often described as middleware because it enables communication between different parts of a robotic system. This makes it easier to divide the program into separate modules running in parallel, which is crucial for a real-time system, like a robot, to function properly.

The challenges faced when making a robotic system are often alike across a wide range of robotic applications. Developers should avoid creating new software components or libraries from scratch if there are existing ones available that can be used instead. This is the main purpose of ROS. In addition to communication, ROS also provides a large collection of software packages and libraries that cover a wide range of functionality, from low-level drivers for sensors and actuators to high-level algorithms for motion planning and control. By using existing ROS packages and libraries, developers can save time and effort and focus on building the unique and specific functionality of their robotic system. Due to the well-defined interface and structure of ROS, a ROS package can be easily shared among developers and reused for new robot applications.

3.1.1 Distributions and operating systems

Open Robotics releases a new distribution of ROS every year. A ROS distribution is a collection of ROS packages with a specific version number. The purpose of ROS distributions is to provide a stable codebase for developers to work with, limiting changes to bug fixes and non-disruptive improvements for the core packages. Humble Hawksbill is the latest released distribution at the time of writing and was hence chosen for this project. It was released in May 2022 and is supported until May 2027. Humble Hawksbill works best for Windows 10 and Ubuntu Jammy Jellyfish. Installing it on other operating systems is possible, but not recommended for beginners [9]. A full list of supported operating systems for the different ROS 2 distributions can be found at https://www.ros.org/reps/rep-2000.html.

There are two main versions of ROS, namely ROS 1 and ROS 2, each with its own annual distribution release. ROS 2 was specifically developed to cater to the industry's needs, focusing on resolving limitations present in ROS 1, such as scalability, real-time capabilities, language support, and security. These improvements required substantial modifications to the system's core components, resulting in a clear distinction between ROS 1 and ROS 2. The latest and final distribution of ROS 1, known as Noetic Ninjemys, was introduced in 2020 and will receive support until May 2025. After this date, there will be no further support for ROS 1 [10]. Consequently, ROS 2 becomes the preferred choice for new projects. Be aware that ROS is often used interchangeably with ROS 1. During this report, ROS 2 will be used interchangeably with ROS.

3.1.2 ROS as a middleware

This section will explain how a distributed system works in ROS. The system can be divided into smaller parts called nodes, and ROS provides tools for communication between them. This network of nodes is defined as the ROS graph.

Nodes

Distributed systems in ROS consist of several nodes running concurrently. Nodes can communicate with each other through topics, services, actions, and parameters. Each of these communication methods will be described in dedicated subsections. A node should be complete, meaning it should be responsible for doing a single, modular task [11]. Throughout this section, sensors, actuators, and controllers will be used as example nodes to help illustrate otherwise abstract concepts. However, it's worth noting that running separate nodes for these components is not common in practice, and we will revisit this topic in section 3.4.

Nodes are designed to be language agnostic, enabling communication between nodes written in various programming languages. The standard installation of ROS offers *client libraries* for developing nodes using both Python and C++. Each node is an instance of the Node class, which is included in these libraries. It is recommended to assign a separate file for each node, although it is feasible to define multiple nodes within the same file. Communities of dedicated developers provide and maintain client libraries for supporting additional programming languages. [11]

Topics

Topics are an essential component of the ROS graph, as they enable communication between nodes. As depicted in Figure 3.1, nodes can communicate with one another by publishing and subscribing to topics. Nodes can publish messages to a topic, and nodes that are interested in data on that topic can subscribe to it. "/encoderData" is an example topic that an encoder would naturally publish its data to. A controller node could subscribe to this topic to receive the encoder data. The controller node might process the encoder data and publish a motor input for a motor node. This system provides a versatile communication mechanism that extends beyond one-to-one interactions, allowing for one-to-many, many-to-one, and many-to-many communication. A node may publish to any number of topics and may subscribe to any number of topics. [12]

In practice, a node publish and subscribe to a topic by creating publishers and subscribers. These are instantiated using the Node class provided by the client libraries. A node might contain any number of subscribers and publishers. The topic name and message type must be defined when creating a publisher or a subscriber. Both parameters must match in order for them to communicate. The topic is automatically created when a publisher starts publishing to it. [13]

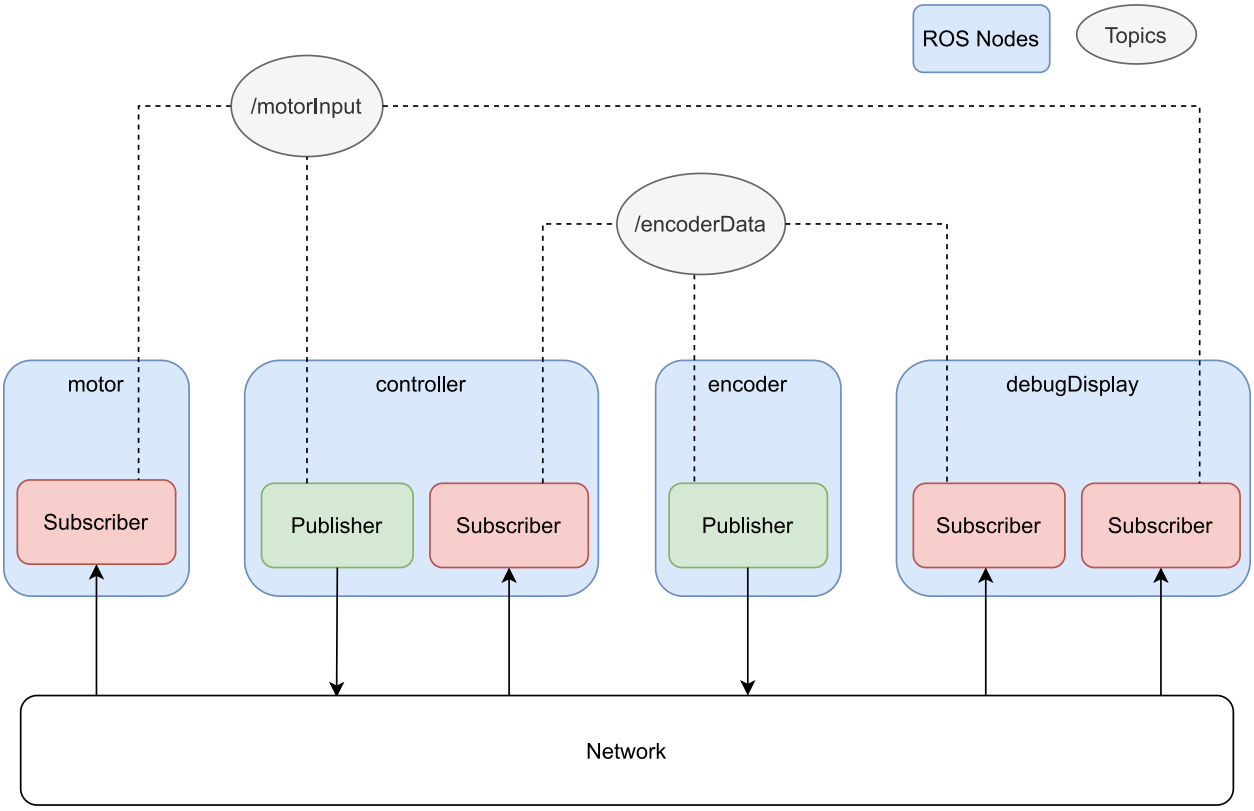


Figure 3.1: An illustration of how topic are utilized to communicate between nodes.

Note that the publisher does not care who, if anyone, receives its messages. In the case of the encoder, its responsibility is only to publish new data on the topic. It does not care how it is used, and hence the encoder node could easily be reused in other projects. Likewise, subscribers do not care where a message comes from, only that it is on the correct topic, with the correct message type.

In summary, topics are an integral part of the ROS communication framework, enabling nodes to publish and subscribe to data. This flexible communication mechanism allows for different types of interactions and facilitates the reuse of nodes across multiple projects.

Services

In a distributed system, ROS services provide a request-response pattern for one-to-one communication between nodes. As illustrated in Figure 3.2, service clients and service servers are created as part of a node, just like publishers and subscribers. The client initiates the sequence by sending a request message, and the server responds with an appropriate message. The service block in the middle represents the underlying communication layer, which is abstracted away for the ROS user. The request and response messages are predefined in a srv-file. Each message can contain many data types, such as floats, strings, or other custom types, or in some cases be empty. [14]

While multiple service clients can utilize the same service server, each client can only be associated with one server. A node can have any number of service clients and servers. However, it's important to note that while a client waits for a response from the server, it will block the client's node. Similarly, the server will block its node while generating a response for the client. As a result, services are not suited for continuous calls, and the response should be generated quickly by the server. Typical uses for ROS services include starting, resetting, and stopping the server's nodes, or requesting specific data from server nodes. [15]

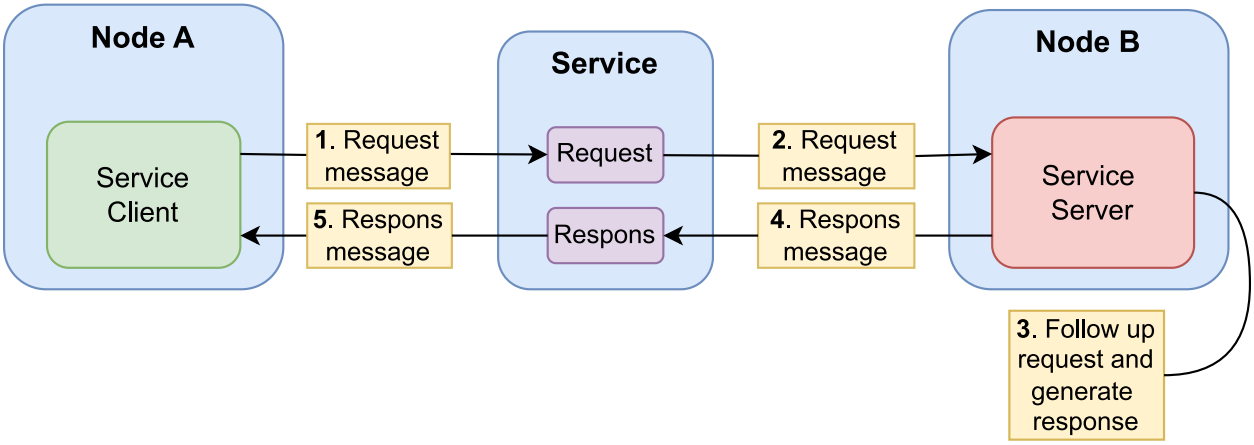


Figure 3.2: Illustration of the request and response pattern of ROS services. The sequence of the pattern is given by the numbers.

Actions

ROS 2 utilizes actions as a form of communication, designed specifically for lengthy tasks. An action is composed of services and topics, mainly the goal service, the feedback topic, and a result service. The server's progress can be monitored asynchronously by the client, and the request can be canceled at any time.

The typical flow of an action will be explained through an illustrative example. Consider the scenario where a robotic arm is required to be moved to a new position. Figure 3.3 will be frequently referenced to illustrate the process. In this scenario, Node B has control over the robotic arm, while Node A requests the movement. The sequence begins with the action server (of Node A) sending a goal request specifying the new position of the arm. The goal request is processed by the goal service server, where the user can specify conditions for accepting the goal. Upon acceptance, the action server will execute a user-defined method to fulfill the goal, while simultaneously handling incoming requests asynchronously. While the method is being executed, feedback messages can be published on the feedback topic. In the robot arm example, Node A could publish the current position of the robot arm, while the movement is being executed. These messages allow the action client to monitor the server's progress. [16]

After the action server has received a goal response, it can send a result request through the result service. The result response must contain whether the goal was successfully reached but may contain extra data such as the robot arm's final position. A crucial feature of the action design is that Node A is not blocked while waiting for a result response. Instead, the user defines a callback function that handles the response when it arrives. [16]

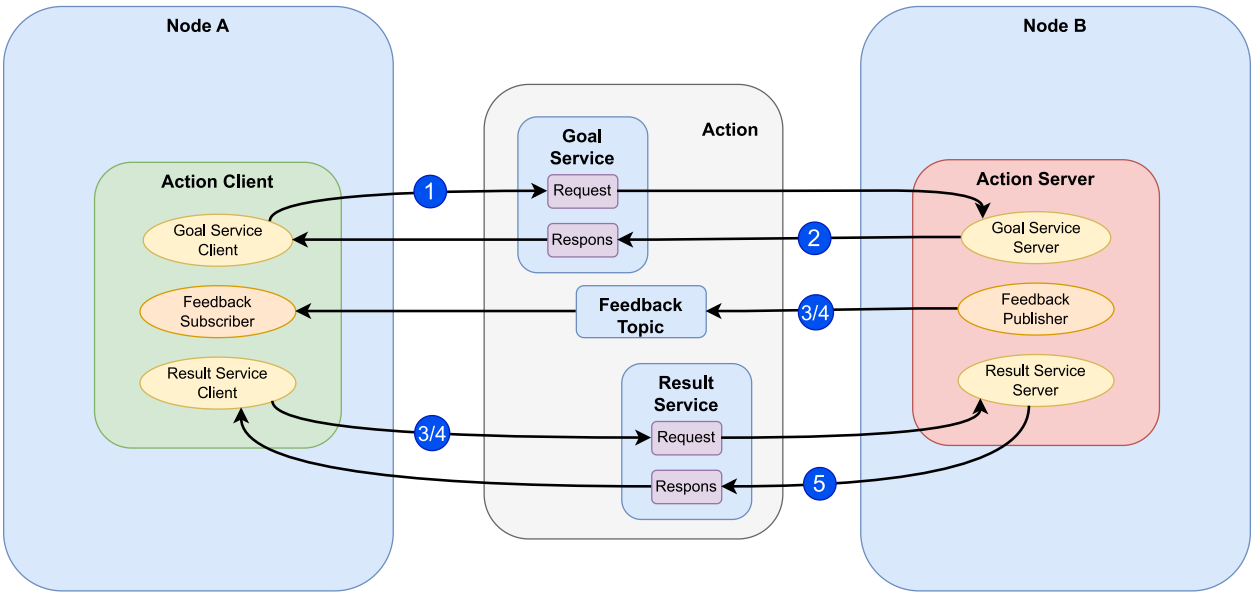


Figure 3.3: Illustration of the interactions between an action server and an action client. The numbers give the sequence.

To minimize clutter, certain details were omitted from the explanation and Figure 3.3. However, it is worth noting that there is an additional service called the Cancel Goal Service. This service allows the action client to cancel the goal prematurely. Moreover, the goal goes through different states during its lifespan, such as "Canceled" and "Succeeded." Whenever the goal transitions from one state to another, the action server publishes the new state on the Goal Status Topic. The action client does not subscribe to this topic. It is mainly used for introspecting the action server, typically for debugging. [17]

Actions are highly customizable. The user can define custom callbacks for all incoming messages to the action server and action client. For example, the user can define how incoming goal requests should be processed. To ensure goal feasibility, the user may choose to validate it before accepting it. If the server is already executing a previously received goal, incoming goals may be queued, replace the current goal, or declined entirely, depending on the user's preference. This level of customization allows the user to adapt the action system to their specific needs. The asynchronous execution and handling of incoming requests by the action server enable it to serve several action clients concurrently. [17]

Overall, actions provide a powerful and flexible method of interaction between ROS2 nodes and can be a key tool in building complex robotic systems.

Parameters

Parameters are values used to configure a node's behavior, enabling the modification of how a specific node operates, such as tuning parameters for a controller or setting the operating frequency for a sensor. Each node in ROS 2 maintains its own set of parameters. They provide a set of "get/set parameter services", which can be accessed, and utilized by other nodes or external programs. Parameters can be changed while the node is running, thanks to these services. The parameters can also be set when starting up a node. YAML-files may be used to store a set of node parameters in one place, making it easier for users to switch between common configurations for the node. [18, 19]

ROS provides two types of callback functions for parameters that allow users to exercise more control. The first type is triggered right before a parameter change is executed, allowing the user to reject the change or prepare for it. For example, changing controller parameters during runtime could result in large spikes in the controller output, so the user may want to reject the change or prepare for it accordingly. The second type of callback is triggered after the parameter has changed, enabling the user to respond in a desired way. [19]

Overall, parameters provide users with greater flexibility and control over the behavior of nodes without the need to modify and recompile the code of the node.

3.1.3 Launch files

A typical way to start a ROS node is by running it from the command line with specific parameters. However, as the robot application becomes more complex and the number of nodes in a project grows, the process of manually launching the nodes can become cumbersome and error-prone. Fortunately, ROS provides a more efficient and scalable way to launch nodes using launch files. Launch files allow users to launch multiple nodes, and specify their parameters, and other configurations within one file, which can be executed with a single command.

For instance, users can create separate launch files for launching their robot application in a simulation and on actual hardware. Launch files can also launch other launch files and take in arguments. This allows for the creation of a macro launch file that, based on an argument, runs the simulation launch file or the hardware launch file. Another use case is to bundle necessary nodes for a LiDAR sensor in one launch file, which can then be called as part of a macro launch file responsible for starting up the entire robot. This enables developers to separate the project into smaller modules.

In summary, launch files make it easier to start up and manage complex robot applications with multiple nodes. By reducing the manual effort required to launch individual nodes and manage their parameters, launch files enable developers to focus on building the functionality of the robot application.

3.1.4 Packages and Workspaces

In ROS, a package is the fundamental unit of the software structure. It contains code, libraries, data, and configuration files that provide functionality for a specific task. Packages can depend on other packages, allowing for modular development and reuse of code. The purpose of packages in ROS is to provide a standardized way of organizing and sharing code between projects.

Packages in ROS are organized within a designated folder known as a ROS workspace. A command line tool called *Colon build* is utilized to build and install the packages into dedicated folders in the workspace. Among other things, *Colon build* creates executables from the source code that can be run in the command line. Once the build process is complete, the workspace can be sourced into a command-line instance, which makes all the packages of the workspace available. In practice, sourcing a workspace entails a setup file (.bash) created by *colcon build* that informs the command line instance where all the packages and their executables can be found. Hence, users do not have to provide the path when an executable is to be run from the command line. The following listing illustrates a simplified ROS2 workspace.

```
workspace_folder/
   install/
     Output from colcon build (setup.bash is located here)
   build/
     Output from colcon build
   src/
     package_1/
       src/
         nodeA.cpp
         nodeB.cpp
       launch/
         launchNodes.launch.py
       ...
     package_2/
     ...
     package_n/
```

3.2 Unified Robotics Description Format (URDF)

URDF, or Unified Robot Description Format, is a widely used markup language in the field of robotics. The main purpose of URDF is to provide a comprehensive description of a robot's physical properties, such as its geometric shape, visual appearance, kinematic structure, and dynamic properties. A variety of ROS tools rely on the URDF to provide simulation, visualization, and motion planning among other things. Hence, the URDF is a crucial component when using ROS.

URDF is based on XML (eXtensible Markup Language), which is a widely used format for describing complex structures and hierarchical data. The URDF file follows a hierarchical structure and consists of various elements that represent different components of the robot (see listings below). Elements that are relevant to the project will be covered in the following subsections, but note that other elements exist and are covered in the official documentation by Open Robotics [20].

Listing 3.1: The example code illustrates the hierarchical structure of the URDF format. Note that the example is not complete, necessary elements are not fully defined.

```
<robot name="example_robot">
  <link name="base_link">
    <visual>
      <origin>
      <geometry>
      <material>
    </visual>
    <collision>
      <origin>
      <geometry>
    </collision>
    <inertial>
      <mass>
      <inertia>
    </inertial>
  </link>
  <link name="arm_link">
    <--! Similar to base_link -->
  </link>
  <joint name="shoulder_joint" type="revolute">
    <origin>
    <parent link="base_link"/>
    <child link="arm_link"/>
  </joint>
  <gazebo>
    <--! Specify parameters used for simulation in Gazebo -->
  </gazebo>
</robot>
```

3.2.1 Robot

The robot element encapsulates all other elements in a URDF and is hence the root element. Link, Joint, and Gazebo elements are hierarchical one level below the robot element. *Links* and *Joints* represent the same concepts as in traditional robotics/kinematics, where the manipulator is composed of links connected by joints to form a kinematic chain.

3.2.2 Link

Rigid bodies are represented as link elements in URDFs. It defines the geometric and physical properties of a specific body, such as its visual appearance, collision properties, and inertial properties. These properties are defined in separate sub-elements. See Figure 3.4 for a visual representation of the sub-elements.

The visual element defines the geometric shape and material of the link. The geometry can be defined by basic Cartesian shapes or by specifying the file path to a triangle mesh, which represents the surface of a 3D object as a mesh of triangles. There are several file formats for triangle meshes, with Open Robotics recommending the use of the *Collada .dae* format, although other formats can also be used with good results. The material element in URDF defines the surface color and texture of the visual element. Note that several visual elements may be defined within one link. The union of these elements will be used to represent the robot visually.

Graphical interfaces, such as Rviz and Gazebo utilize the visual element to present a visual representation of the robot. In most cases, the visual element does not affect the physical simulation of the robot. However, it's worth noting that some visual sensors, such as cameras, may "see" the visual elements in Gazebo. Therefore, proper consideration should be given to accurately defining the visual properties for applications with such visual sensors.

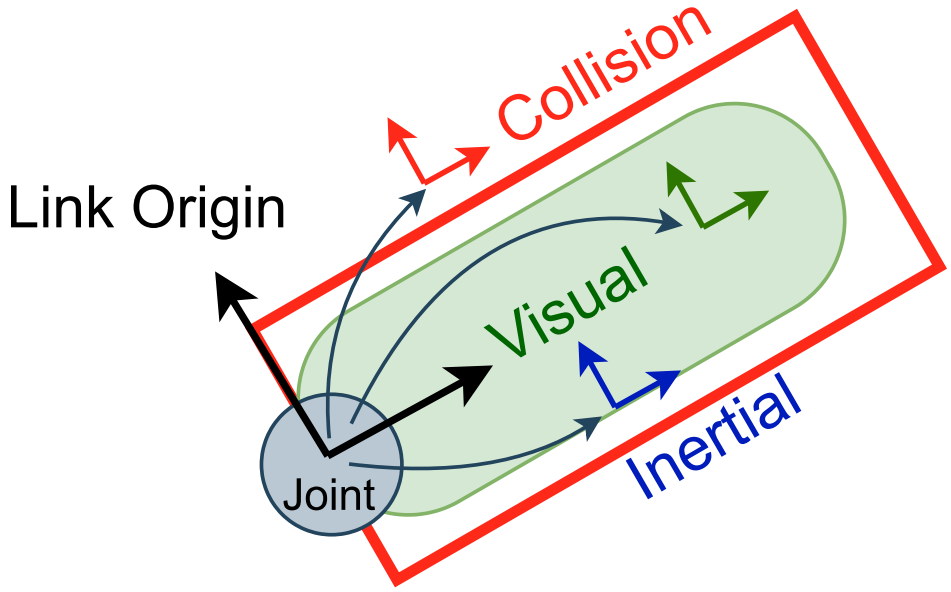


Figure 3.4: A visual representation of the link elements in a URDF. The elements can have separate origins defined relative to the link origin.

The collision element is defined just like the visual element, except that it does not contain the material property. It is used to detect contact between links or with the robot's environment, letting physical simulators simulate realistic forces during collisions and motion planning software generate collision-free trajectories.

In many cases, the collision element may be a direct copy of the corresponding visual element. However, note that complex collision geometry will increase the number of calculations required to check for collision between objects, affecting the simulation speed among other things. This is seldom an issue when making the collision model from basic geometric shapes, but must be considered if the geometry is defined from triangular meshes. Open Robotics recommends that triangle meshes used for collision should contain less than 1000 triangles [21], while other sources recommend even fewer triangles or avoiding the use of triangle meshes for collision geometry [22]. The complexity of the collision model should be chosen carefully based on the application requirements, as some applications may require high accuracy in the collision model (e.g., hand manipulators), while others may not.

The inertial element defines the total mass, center of mass and inertia of the link. It represents the inertia using an inertia matrix that is defined with respect to the center of mass. Note that the origin of the inertial element is used to represent the center of mass.

The inertial element is essential for physics simulations in Gazebo. It is necessary to properly define the inertial element for Gazebo to accept the URDF. Accurate inertial properties are crucial for accurate physics-based simulations of the robot's dynamics. It ensures that the link behaves realistically in simulation, which is important when software for robot control and path planning is to be tested in the simulator.

3.2.3 Joint

Just like in traditional robot kinematics, joints represent the connection between two links in a robot model. URDF supports six types of joints, including revolute joints for rotation and prismatic joints for linear motion. The joint element encapsulates the physical properties of the joint, such as its type, position, limits, dynamics, and the joint's axis of movement. Additionally, it includes properties that are relevant for applying controllers, such as safety limits and calibration information. Only sub-elements that are relevant to this project will be covered below. Refer to the URDF documentation by Open Robotics [23] to read about excluded elements and other details, such as the joint types not mentioned here.

The joint's **parent link** and **child link** are specified in separate elements, indicating the links that are connected by the joint. Figure 3.5 illustrates the relationship between the parent link and child link. Although several child links may share the same parent link, a child link may only have one parent link. This limits the URDF to robots that can be represented as tree structures.

The origin frame of the joint is defined relative to the origin of the parent link. The joint's origin also defines the origin of the child link, and consequently where the child link is connected to the parent link. In other words, the origin element is the transform from the parent link to the child link.

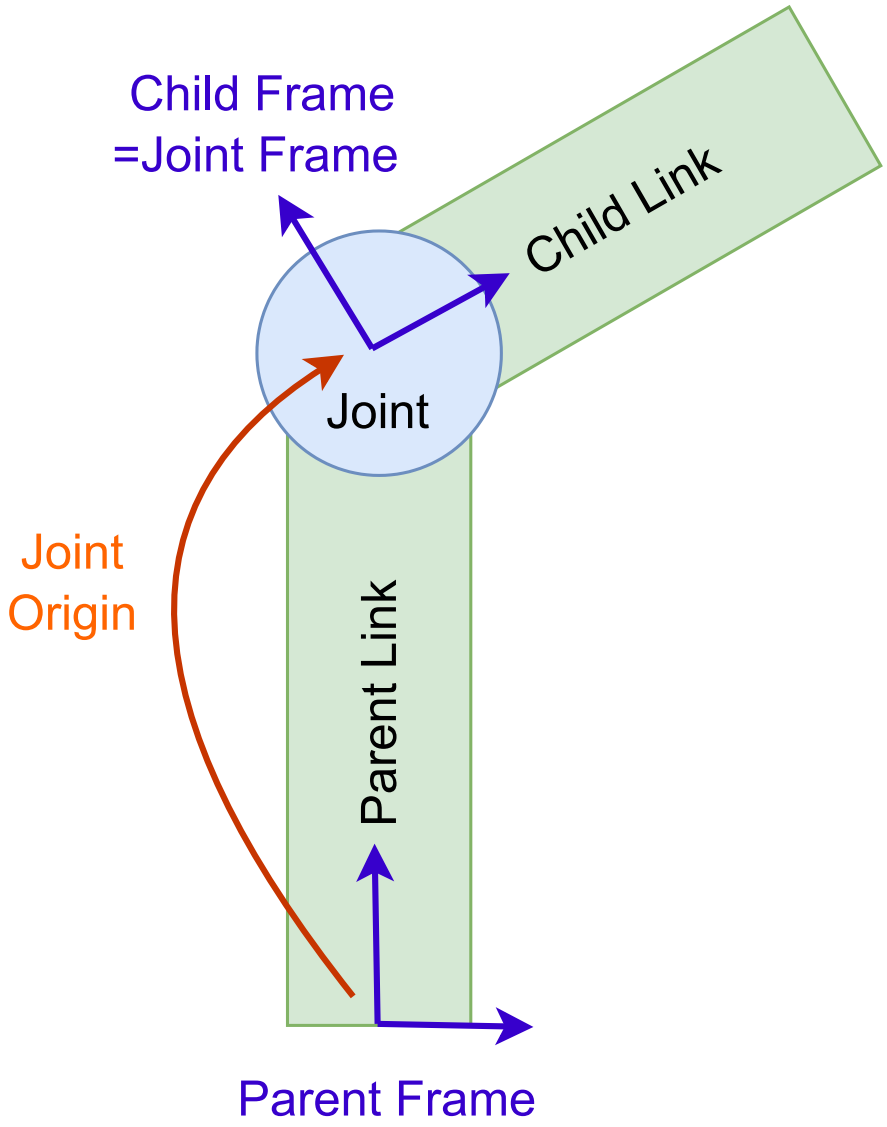


Figure 3.5: A visual representation of how links are connected by joint elements.

The axis element determines the movement of the joint. Depending on the joint type, the axis element will represent which axis the child link can rotate about or move along. Additional restrictions to the movement are defined by the **limit** element, which defines the upper and lower limits of the movement. The limits are expressed as radians for revolute joints and in meters for prismatic joints.

The **dynamics** element include damping and friction coefficients for the joint. These coefficients mainly affect the simulation.

The mimic element allows a joint to mimic the movement of another joint. The displacement value of the joint is determined using the formula: $value = multiplier \times other\_joint\_value + offset$. The offset and multiplier parameters can be specified in their respective element fields.

3.2.4 Gazebo elements

Gazebo is a software for simulating robots with ROS (described in Section 3.5). The URDF format has crucial shortcomings in regard to simulation, and hence Gazebo uses another XML-based format called Simulation Description Format (SDF). Fortunately, Gazebo is able to automatically convert a URDF file to SDF. However, SDF is a more comprehensive format compared to URDF, and certain properties necessary in SDF are not included in URDF. For this reason, Gazebo elements are provided as an extension to the URDF. These elements can be included in the URDF, just like the other elements, but are only recognized by Gazebo and ignored by other software. They provide all the necessary information for Gazebo to conduct a proper physical simulation. There are three different types of Gazebo elements, each providing additional information about the robot, link, and joint element respectively. For example, friction coefficients for the surface of a certain link can be specified in a Gazebo element. [24]

3.2.5 URDF Limitations

Aside from the shortcomings mentioned above, URDF has other limitations that are worth pointing out. Because URDF is a crucial building block in any ROS-driven software, these limitations may have a huge impact. Firstly, only tree-structured robots can be described in a URDF [25]. This rules out robots with parallel kinematic chains, such as *delta robots* and *SCARA robots*. However, it is still possible to simulate such robots in Gazebo because they can be described in the SDF. Rico Ruotong Jia describes how to utilize Moveit as well for parallel robots [26]. Note that workarounds like these could increase the complexity of the overall software. An additional limitation is that only rigged links are supported [25]. No viable method for implementing flexible links in URDF, or ROS in general, was found during the project.

3.3 Rviz

Rviz (ROS visualization) is a powerful tool used in the Robot Operating System (ROS) ecosystem. It allows users to visualize the robot's state and environment, and display sensor data in a user-friendly manner. Rviz is a vital tool for debugging and developing robot applications, as it provides a visual representation of what the robot precepts.

Rviz uses the URDF to create a 3D model of the robot. Recall that the URDF describes the robot's physical characteristics, such as its joints, links, sensors, and geometry. Rviz retrieves the URDF by subscribing to the *robot_description* topic. Hence, another ROS node is required to publish the URDF content to this topic. By visualizing the robot's URDF, users can verify that the robot's joints and links are defined correctly, and identify any issues that need to be addressed.

In addition to displaying the robot's state, Rviz can also display sensor data in the visualization. It can subscribe to topics that publish sensor data, such as point clouds and laser scans, and displays them in a variety of ways, such as color maps and 3D point clouds. This allows users to see what the robot's sensors are "seeing" and verify that they are working correctly. By visualizing the sensor data in Rviz, users can identify issues with the sensor readings and improve the robot's perception capabilities. Figure 3.6 illustrates an example of how a robot and its sensor data can be visualized in Rviz.

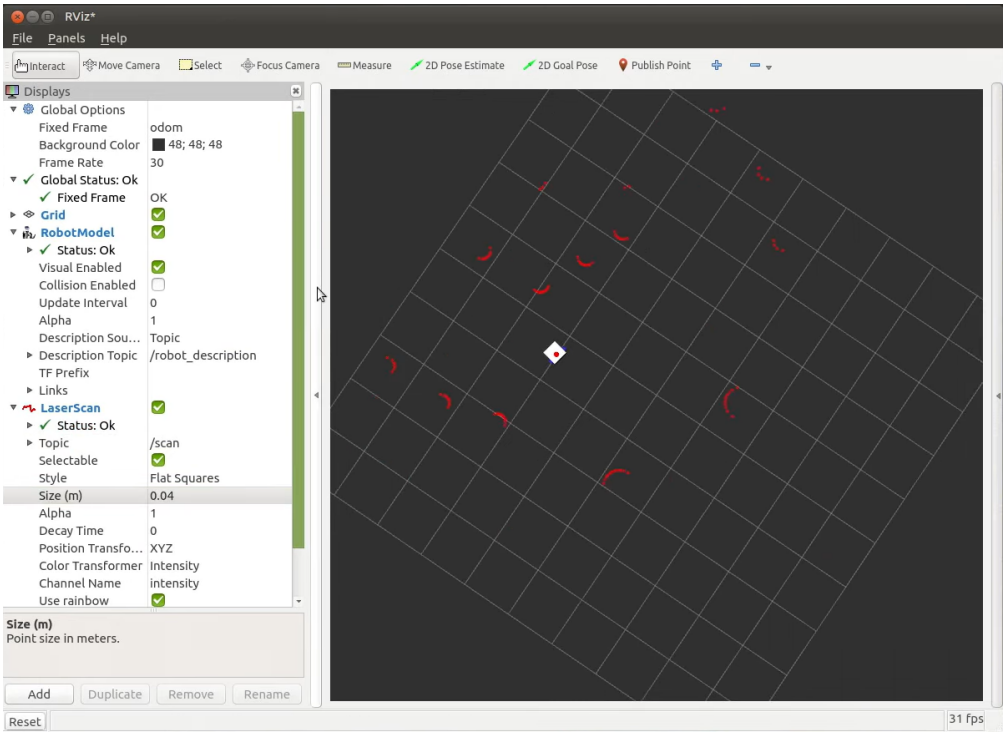


Figure 3.6: Screenshot of the Rviz interface. This example shows a two-wheeled robot (white and blue) with a 2D LiDAR (red) mounted on top. The robot is navigating in an environment containing several traffic cones, and the data from the LiDAR is represented with red dots. Picture taken from Articulated Robotics by Josh Newans [27].

3.4 ROS2_control

ROS2_control is a standardized framework designed for real-time control of robots using ROS 2. One of its most important features is the ability for users to define hardware interfaces and controllers in a modular way. This modularity allows for easy modification of the controllers or change of hardware components with minimal modifications to the software. Furthermore, the modular design makes the code reusable between projects, which allows developers to share software more easily. This approach is in line with the core philosophy of ROS2_control and ROS in general, which emphasizes not wasting time on solving a problem that has already been solved. For instance, if someone has already developed a hardware interface for the specific sensor you are using, you can be confident that it can easily be integrated into your robot application. The rest of the section will briefly explain the design of ROS2_control through the use of the diagram of Figure 3.7.

Hardware interfaces are responsible for abstracting away the hardware details, and providing standardized interfaces of controllable inputs, called *command interfaces*, and readable states, called *state interfaces*. By separating the abstracting hardware from the control logic, ROS2_control provides a clear separation of concerns and allows for easier code maintenance and reusability. Hardware interfaces might be provided by the manufacturer of the hardware components or have to be written by the user.

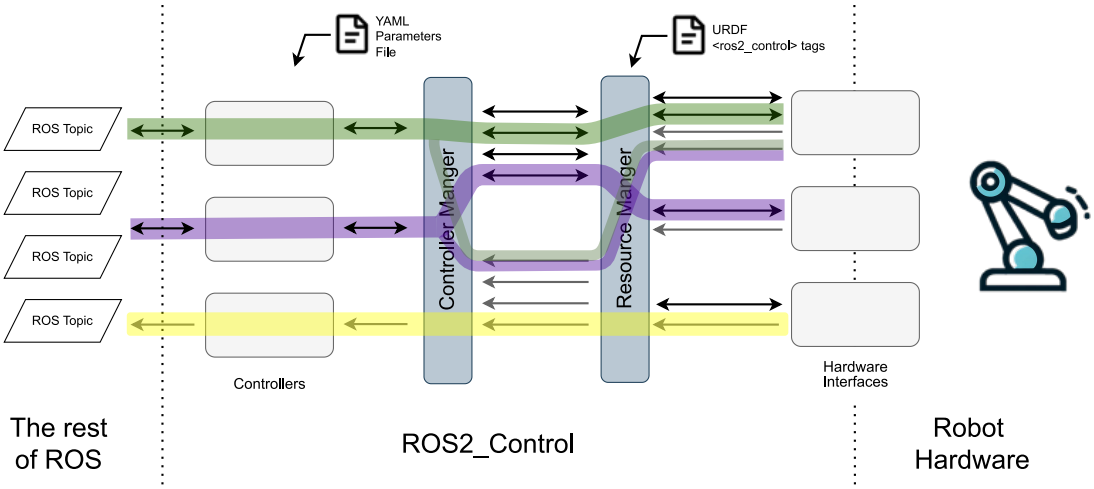


Figure 3.7: Overview of ROS2_control. Colors represent examples of how the Controller Manager could connect controllers and hardware interfaces. Double arrows represent command interfaces, while single arrows represent state interfaces. Image inspired by an illustration of Josh Newans [28].

Robot applications often have multiple hardware interfaces. To deal with this, the controller manager uses a resource manager to pipeline all of these interfaces into a singular interface. The resource manager read the URDF to figure out which hardware interfaces to load.

The controllers are located on the left-hand side. They typically take in one or several reference values by subscribing to ROS topics, and interact with the hardware interface, through the Controller manager, to create a controlled feedback loop. ROS2_controll comes with a variety of standard controllers designed for robot control. These range from simple feedback PID controllers for controlling effort, velocity, or position, to more sophisticated controllers that can control a set of joints to follow a trajectory. Note that controllers do not have to be used for feedback control. They can be used for simple feedforward, or just publish state values from sensor data to ROS topics (illustrated by the yellow path). It is possible to have multiple controllers for one robot, as long as they don't try to control the same state.

The controller manager plays a crucial role in tying everything together. It is responsible for loading controllers and matching them with the correct command and state interfaces exposed by the resource manager. A YAML file is used to specify which controllers to load and the parameters for the controller. Once the controllers are successfully loaded, they can be started and stopped at runtime by the user through the controller manager.

It's important to note that the different blocks shown in Figure 3.7 are not separate nodes. Instead, the controllers and hardware interfaces are implemented as plugins that are loaded by the Controller Manager. This means that all the components are actually run in one node, which is more efficient than having separate nodes communicating through topics, actions, and services. Plugins can be thought of as libraries that can be loaded at runtime, adding functionality to the Controller Manager. This approach is a key factor in enabling ROS2_control to meet the strict deadlines required for real-time applications.

3.5 Gazebo

Simulating robots and testing new algorithms and software on actual robot hardware can be costly, time-consuming, and even dangerous. Having a good simulation environment can reduce the risks and expenses associated with testing on real hardware. Gazebo is a powerful open-source tool used for physics-based simulation of robots and their environments. It is provided by Open Robotic, the same organization that provides ROS. Gazebo is not dependent on ROS, but integration with ROS is provided as a plugin. [29]

In Gazebo, users can design virtual environments, called worlds, that the robot can be tested in. These worlds can be modeled after real-world environments or created from scratch using the built-in world editor. A custom robot can easily be spawned into the virtual world by providing a URDF to Gazebo. This can be achieved by publishing the URDF to the */robot_description* topic. The standard ROS installation comes with a node called *robot_state_publisher*, which can be used to publish the URDF. [29]

Sensors and actuators can be described in the URDF, enabling realistic feedback control of the simulated robot. Gazebo offers plugins for all sorts of sensors and actuators, but users are free to create custom plugins if required. Actuators will be affected by physical properties such as friction, damping, and mass, creating a realistic simulator. Hence, Gazebo can be used to test and tune controller algorithms before applying them to actual hardware. Gazebo uses hardware interface plugins for actuators and sensors making it compatible with *ROS2_control*. Because the rest of the ROS system only interacts with the sensors and actuators through *ROS2_control*, switching between simulation in Gazebo and employing the software on the actual robot requires only minimal changes to the system. This point is illustrated in 3.8. Here, MoveIt is used for motion control and can be thought of as a high-level controller in a feedback loop. It reads sensor data from the topics: */joint_sensor* and */lidar_sensor* and publishes a controller input which Gazebo applies to the virtual actuators. From MoveIt's point of view, it does not matter if it is Gazebo or a real robot on the other side. Note that *ROS2_control* is usually incorporated between the Gazebo plugins and the sensor and controller topics. What happens inside the MoveIt block in the figure will be explained in Section 3.6. [29]

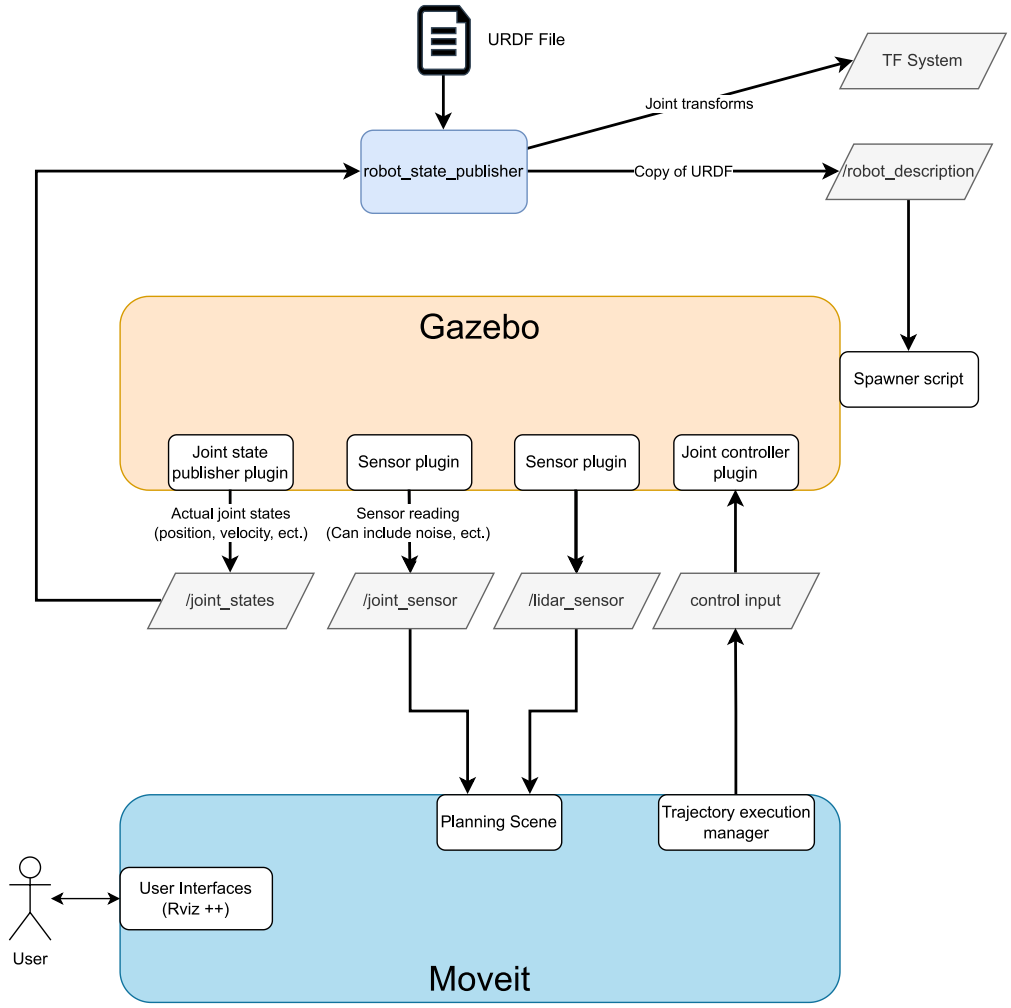


Figure 3.8: Illustration of a typical interface between MoveIt and Gazebo. Inspired by an image created by Josh Newans [24]

3.6 MoveIt

Spraying concrete on tunnel walls autonomously is a challenging task that requires precise and coordinated motion planning of the robotic arm in a constrained environment. The robot arm needs to maintain a safe distance from various obstacles, such as other machines and the walls, to avoid collisions. It also needs to ensure that it can reach its target positions while satisfying various constraints, such as joint limits and workspace limitations. Moreover, the shotcrete spraying process involves spraying the shotcrete mixture on the wall using a nozzle attached to the end of the robot arm, requiring precise control of the nozzle's orientation, position, and velocity to achieve the desired thickness of the resulting concrete layer. Solving the motion planning problem is essential for deploying autonomous robotic systems for shotcrete spraying in tunnel construction projects. It requires efficient and robust algorithms and tools that can handle the complexity of the problem.

MoveIt is an open-source software framework that provides a set of packages for motion planning, manipulation, and perception for robots. It is designed to work with the Robot Operating System (ROS) and provides a flexible and modular framework for developing various robotic applications. Rviz can be utilized as a user interface, providing an interactive and visual 3D environment. In addition, Gazebo can be implemented as well, providing physical simulations. Figure 3.8 illustrates the interface between Gazebo and MoveIt. A description of the MoveIt elements will follow, but the takeaway here is that Gazebo works seamlessly together with MoveIt. Picknik Robotics is the lead organization for the development of MoveIt, but all developers are welcome to contribute to the project. MoveIt comes in different versions that correspond to the ROS distributions. Note that the newer MoveIt releases, for ROS 2 distributions, are sometimes referred to as MoveIt2.

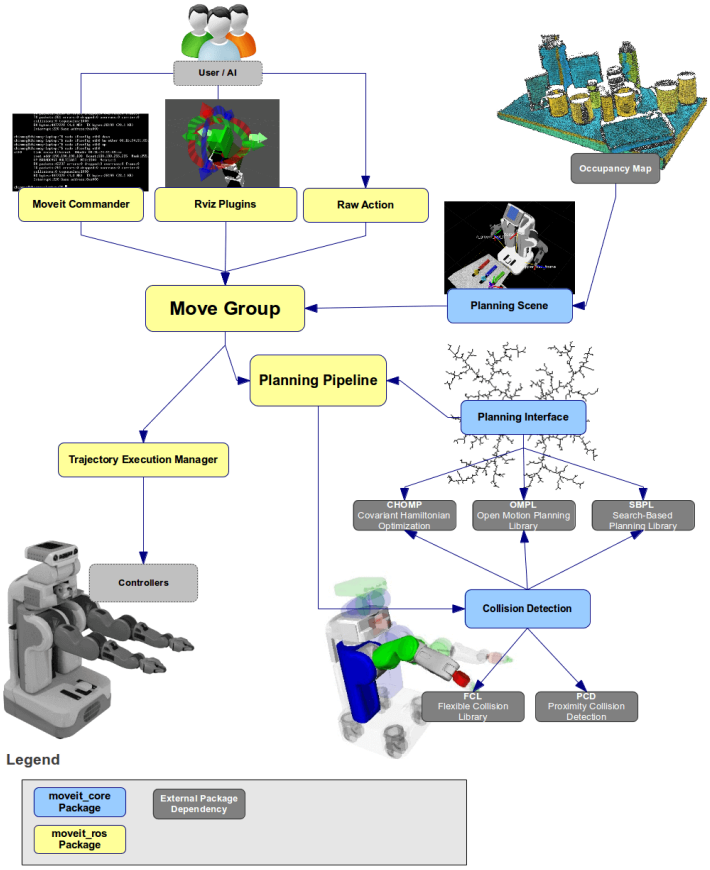


Figure 3.9: The MoveIt pipeline. Picture created by Open Robotics [30]

Figure 3.9 shows the move_group node at the core of MoveIt. This node brings together all the individual components of MoveIt to offer a comprehensive set of ROS actions and services, available for users. In addition, MoveIt comes with extra interfaces for the move_group node that abstract the services and actions away. The Rviz plugin, lets the user interact with the move_group node through Rviz. Among other things, the user can request motions by dragging the robot to a desired state, and clicking "plan". The resulting plan will be shown as an animation of the robot moving to the requested state. In addition, there are code interfaces for *C++* and *Python* that provide more advanced capabilities and customization.

All information about the robot and its environment is stored in the *planning scene*. The *planning scene* is continually updated through sensor messages, such as joint states and image data from cameras, and point clouds obtained from LiDARs. During motion planning, the *planning scene* is utilized to generate collision-free trajectories.

The move_group uses the planning pipeline to create joint trajectories that satisfy motion requests from the user. The planning pipeline encompasses *Collision Detection* and a *Planning Interface* (bottom right of Figure 3.9). In simple terms, these components are interfaces to collision and motion planning libraries. Which libraries MoveIt uses is determined in dedicated configuration files. The default and most commonly used motion planning library is the Open Motion Planning Library (OMPL). This library includes algorithms such as PRM, RRT, SPARS, and variants of these. More information about available algorithms can be found on the home page of OMPL [31]. Choosing the optimal algorithm will depend on the use case, and require knowledge about the algorithms and a good understanding of the use case. Luckily, the planning pipeline will automatically choose an algorithm based on the planning scene, that should give decent performance. Read more about how OMPL and other planning libraries are implemented in MoveIt in the MoveIt documentations [32].

Planning adapters can be included in the planning pipeline, and can be categorized into two types: those that adjust the motion request before planning and those that adjust the motion response after planning (see Figure 3.10). The former type of adapter primarily aims to correct invalid requests or fill in any missing information. On the other hand, the latter type of adapter is used to refine the output trajectories after planning. One example of an adapter that modifies the output trajectory is the "AddTimeParameterization" adapter. Planning libraries typically generate paths in joint space without a time parameter, making them unsuitable for direct execution. The "AddTimeParameterization" adapter solves this problem by parameterizing the path with time while adhering to the velocity and acceleration constraints on the joints.

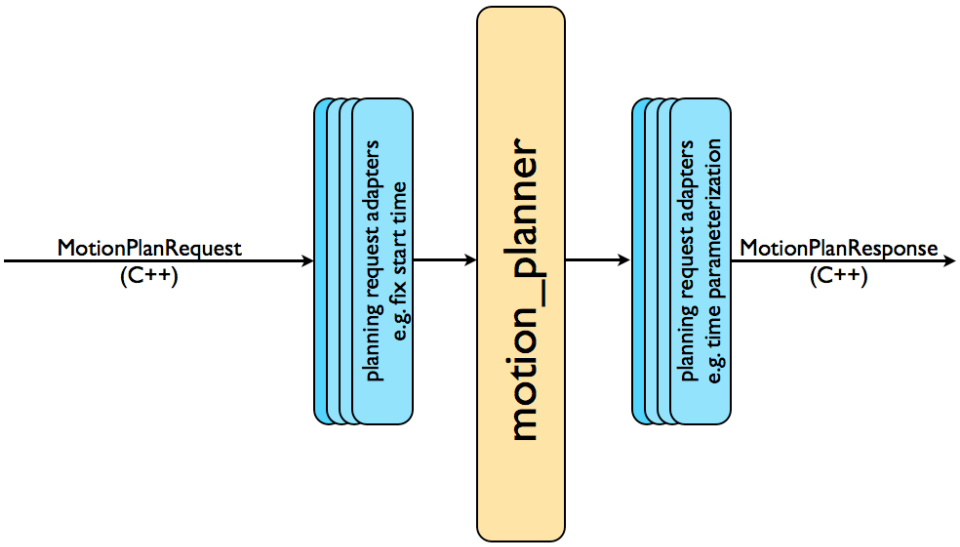


Figure 3.10: The adapters can be utilized to alter the input to and output from the motion planner. The picture was taken from MoveIt documentations [30].

Kinematics is a crucial component of motion planning. While motion requests are typically defined in task space, the resulting motion trajectory is represented in joint space. To convert from task space to joint space is called inverse kinematics, and MoveIt solves this through kinematic plugins. Different approaches to inverse kinematics exist and MoveIt provides a variety of different kinematic solvers, but users may also implement custom solvers. The default inverse kinematics solver is a numerical jacobian-based solver, called KDL. However, other solvers like the analysis-based solver IKFast are available, which can derive inverse kinematic equations based on the URDF [33]. Forward kinematics, which determines the end effector position and orientation based on the variables of the robot [2], is also available in MoveIt.

As mentioned in recent paragraphs, MoveIt offers a lot of configurations for its various components. All of these configurations are defined in a dedicated MoveIt configuration package. However, setting up all of these configurations correctly can be daunting for beginners, as they are spread across several files and follow specific naming conventions. To simplify the process, MoveIt offers a graphical user interface called MoveIt Setup Assistant.

The Setup Assistant uses the robot's URDF file to define its physical properties and generates a complete robot model that can be used for motion planning and control. When the URDF is provided, the Setup Assistant guides the user through the remaining configurations that need to be set, making the process more manageable for users with limited experience in MoveIt configuration. As a bonus, the Setup Assistant generates a bundle of useful launch files, including a demo that can be used to test the configurations with Rviz. However, note that not all configurations can be altered with the Setup Assistant. For advanced customization, the user may need to manually edit the configuration files. [34]

Chapter 4

Software Implementation

This chapter will describe the process of developing the ROS software for the AMV 4200 Shotcrete robot. It will include everything from installing the core ROS packages to simulating the dynamics of the robot in a virtual environment, and motion planning. This chapter will cover how ROS, Gazebo, and MoveIt were utilized in the project.

The sections will describe one step in software implementation. They will consist of three main parts. The first is a *Method* part that concisely describes what was done to reach the final outcome. This is followed by a part that describes the process in more detail, explains why the method was chosen, and discusses alternative approaches that were considered. The final part will be a *conclusion* which includes some recommendations based on experience gathered during the project. The recommendations are targeted toward AMV and are meant to guide them in developing similar software for their other robots or to improve the software built during this project. However, the recommendation might be useful for others, as the methods used during the project are applicable to a large set of different robots.

4.1 ROS installation

This section will not include a step-by-step on how to install ROS, but rather justify the choice of ROS distribution and operating system to run it on. The conditions for these choices will change over time, hence tips on how to make these choices at a later date are also provided. When the operating system and distribution are chosen, follow the appropriate installation guide provided by Open Robotics[35].

4.1.1 Method

ROS was installed on Ubuntu 22.04 using Debian, as described in the official guide [35]. Humble Hawksbill was chosen as the distribution.

4.1.2 Choosing Distribution

As described in Subsection 3.1.1, Open Robotics releases a new distribution of ROS every year. Humble Hawksbill was recommended by Open Robotics at the time of writing [36], and was hence chosen. However, there are pros and cons to the different distributions. Here are some tips for choosing the most suitable distribution:

- Choosing the latest distribution will give access to the newest features.
- All distributions have an end-of-life date. The distribution will be supported by Open Robotics up until this date. Be aware that "newest release" is not equivalent to "latest end-of-life" [37].
- Each distribution is available to a limited selection of operating systems.
- Third-party software, such as MoveIt, is usually supported for a limited selection of distributions.

Humble Hawksbill was an easy choice at this point in time. According to Open Robotics [37], it is both the latest distribution, offering the newest features, and has the latest end-of-life date (May 2027). Additionally, it has been available for almost a year, giving the ROS community plenty of time to adapt. This means that most third-party software and packages are available for Humble Hawksbill. Finally, note that it is fully possible to migrate from one distribution to another. This is normal procedure when upgrading from an older to newer distribution.

4.1.3 Choosing Operating system

Open Robotics highly recommends using either Windows 10 or Ubuntu 22.04, as they are described as tier 1 operating systems for Humble Hawksbill [36]. Because Windows 10 was familiar, and already installed on the PC provided by the university, ROS was initially installed on the Windows 10 operating system. However, it was decided to switch to Ubuntu 22.04 for several reasons. ROS is built for Ubuntu, and third-party programs must be utilized to make ROS work on Windows 10. As a result, the installation process is several times more complex on Windows compared to Ubuntu. This can be seen by comparing the recommended installation methods for Windows and Ubuntu [38, 35]. The dependencies on the third-party programs add more potential for bugs if installed incorrectly. Additionally, the majority of the ROS community utilizes Ubuntu, making it more difficult to find support when using Windows. At the time of writing, Windows 10 is not a recommended platform for the simulation tool used in this project, Gazebo [39]. The tool used for path planning and kinematics, MoveIt, is only available in an older distribution of ROS called ROS 2 Foxy for Windows 10 [40].

4.1.4 Conclusion

AMV should install the Humble Hawksbill distribution on Ubuntu 22.04, regardless of prior experience with Ubuntu. Essential third-party software, such as MoveIt and Gazebo, are either exclusively available or best supported on Ubuntu. Using Ubuntu will ensure that engineers have access to these powerful tools and save them significant development time.

4.2 Learning the essentials in ROS

ROS is a fantastic tool with a seemingly endless list of capabilities, which can feel overwhelming at first. Spending time getting a fundamental understanding of what ROS is and how it works, is crucial before taking on a big project.

4.2.1 Method

The beginner and intermediate tutorials by Open Robotics [41] were used to grasp the ROS basics. The YouTube videos and blogs of Josh Newans [42] were used as a supplement.

4.2.2 Tutorials

Open Robotics [41] have created a series of tutorials that teaches the ROS essentials. They are designed to be beginner-friendly and provide step-by-step instructions with examples and exercises to help users learn ROS in a hands-on manner. It is recommended to follow the tutorial series linearly, as the later tutorials build on the previous tutorials. Only the beginner and intermediate tutorials were completed during the project, as the advanced tutorials were not deemed relevant. An exception was the Gazebo tutorial, which is a part of the advanced tutorials. The estimated time to complete all the beginner and intermediate tutorials sums up to 630 minutes [41].

Josh Newans [42] is one of many active members of the ROS community. He has produced a comprehensive series of educational blogs and videos explaining how to build a robot from scratch, using ROS to build the software. Essential ROS components and advanced tools such as LiDARs and cameras are covered, all within the context of building a single robot. The tutorials provide a deep understanding of the interactions between the different ROS components and other compatible software, demonstrating how they can be combined to create software capable of executing complex tasks. Source code is also provided throughout the series, encouraging the reader to follow along and experiment with the code.

4.2.3 Conclusion

The ROS tutorials by Open Robotics are a great way to get started with the basics. The content of Josh Newans gives a much better understanding of how all the different components of ROS interact to create a distributed robotic system. For this reason, it is warmly recommended as a supplement to the official ROS tutorials.

4.3 Making the URDF

One of the most central aspects of any ROS software is the URDF because it describes all physical properties of the robot, including geometry, visuals, actuators and much more. Nearly all tools and packages utilized in the project require information from the URDF to fulfill their purpose. There are different approaches to making a URDF, and this section will include discussions on the advantages and drawbacks of some alternatives to the method that was chosen.

4.3.1 Method

The URDF is based on a CAD model of the 4200 Shotcrete robot, which was provided as a STEP file from AMV. To generate a URDF from the robot model, a SolidWorks tool called sw_urdf_exporter [43] was utilized. Autodesk Inventor was used to acquire inertia values, which were then manually typed into the URDF after it was generated. MeshLab was used to generate the collision mesh for the URDF, by simplifying the visual mesh generated by sw_urdf_exorter.

4.3.2 Challenges of Manually Writing the URDF

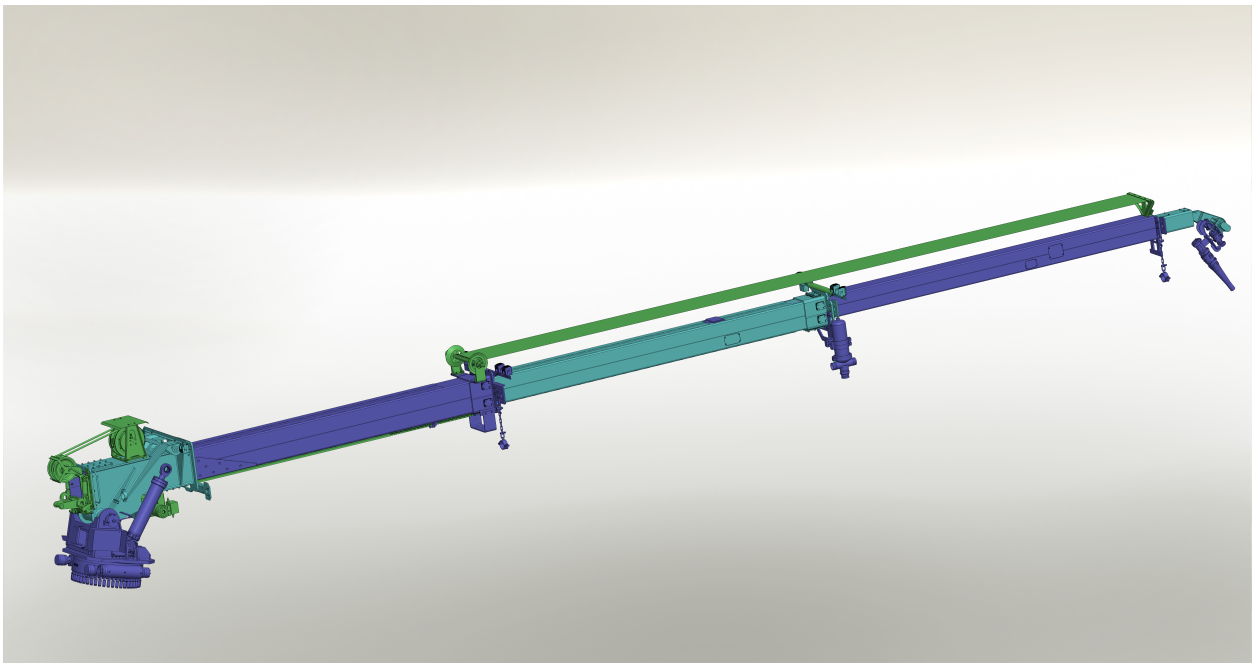
In the early stages of this project, an attempt was made to write the URDF manually. This process provided valuable insight into the challenges of creating a URDF that accurately represents the 4200 Shotcrete. Accurate inertia and other physical properties were important for the physical simulation. Estimating these properties could be difficult and time-consuming. Though not crucial, having a model that visually resembles the actual robot is desired. However, this is difficult to achieve when writing the URDF manually because the model must be created with basic shapes (boxes, spheres, and cylinders). Most importantly, the geometry of the robot must be accurate. If not, the trajectories generated in the software would not be applicable to the real robot. AMV shared a Computer-Aided Design (CAD) model of the 4200 robot, which provides the necessary geometry and physical properties to create the URDF. As described in 4.3.4 the CAD model turned out to be even more useful than initially expected.

Josh Newans' blog post on URDFs [44] was very helpful in understanding the concepts and syntax of the URDF format.

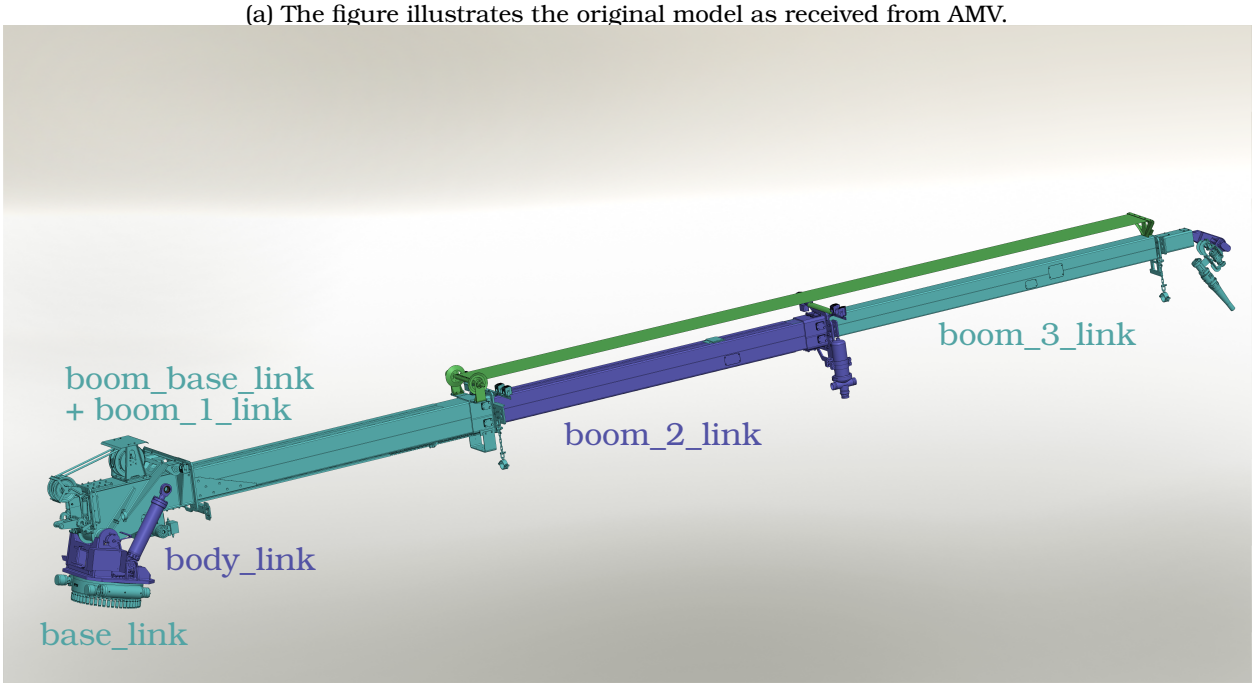
4.3.3 The CAD Model

Before proceeding, it's important to have some basic knowledge of Computer-Aided Design (CAD) models. A CAD model is a digital representation of a physical object or system created using specialized computer software. These virtual 3D models are used to design, visualize and simulate products or systems before they are manufactured or constructed in the physical world. CAD models can include details such as geometry, dimensions, material properties, and surface finishes. SolidWorks and Inventor are examples of software used to create and view CAD models; both were utilized during this project. Figure 4.1 shows the 4200 Shotcrete robot in SolidWorks.

The complete model is constructed from thousands of smaller components such as bolts, nuts, and cables. In SolidWorks and Inventor, the most basic component is called a *part*. The next level of complexity is an assembly, which consists of parts and/or other assemblies. In Figure 4.1 the large assemblies are marked with alternating colors. *Mates* can be used to define relationships between components, by constraining the movement between them. Assemblies and mates are conceptually very similar to links and joints in a URDF. Keep that in mind for the following section.

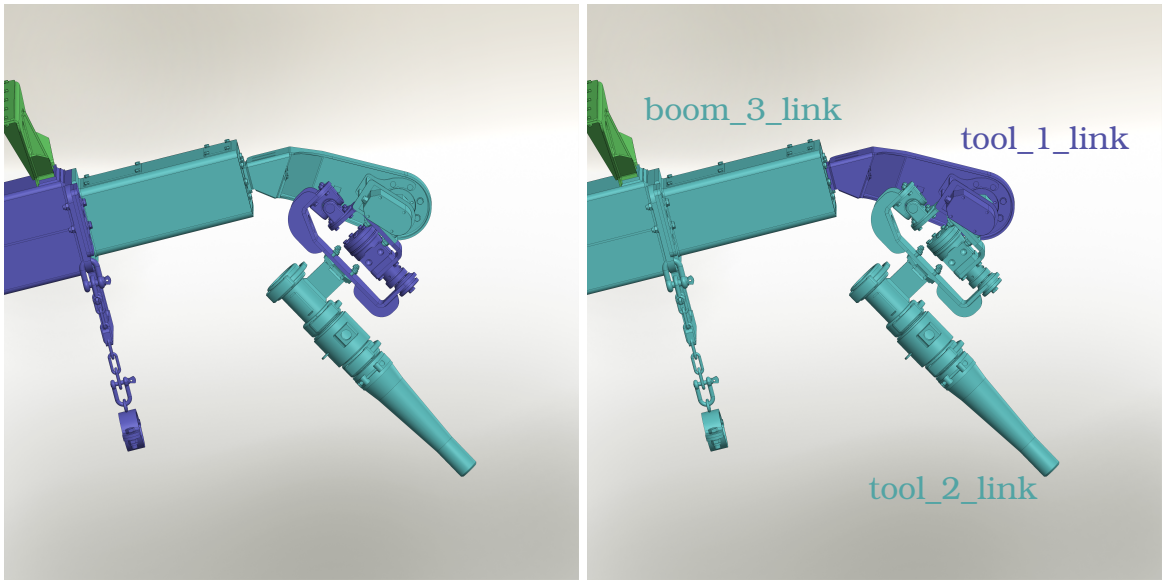


(a) The figure illustrates the original model as received from AMV.



(b) The figure illustrates how the model was altered to be compatible with sw_urdf_exorter. The protective cover (marked green) is not included in the URDF.

Figure 4.1: The AMV 4200 Shotcrete in SolidWorks. Alternating colors are used to differentiate between the main assemblies of the model.



(a) The figure illustrates the original model as received from AMV.

(b) The figure illustrates how the model was altered to be compatible with sw_urdf_exorter.

Figure 4.2: The tool of the AMV 4200 Shotcrete in SolidWorks. Alternating colors are used to differentiate between the main assemblies of the model.

4.3.4 Generating a URDF from a CAD model

A tool for Soildworks called sw_urdf_exporter was used to generate a URDF file from the robot models. sw_urdf_exporter allows the user to define the links for the URDF by selecting assemblies in the model. The joint type between the links is automatically detected, but can be altered by the user. From these initial configurations, the tool is able to properly define the geometry of the robot, generate visual mesh files, and add physical properties to the URDF. This makes the tool a huge time saver if the robot model is available in SolidWorks. Unfortunately, the model provided by AMV had some issues had to be solved before the sw_urdf_exporter could be utilized.

Links and Joints

The sw_urdf_exporter utilizes assemblies in SolidWorks to generate links in the URDF. Likewise, the SolidWorks mates constitute the URDF joints. It is imperative that each link corresponds to either a single SolidWorks assembly or a collection of assemblies. This was not the case with the original model, as some assemblies would cross over joints and would consequently be a part of two links. From Figure 4.1 it can be seen that the base and body links were originally one assembly. Modifications were also required for the tool, as depicted in Figure 4.2.

Furthermore, the assemblies must be mated together in a way that resembles the actual robot. In practice, this means you should be able to move links of the robot model in SolidWorks just like in reality. The original model did not meet this requirement. All the assemblies were fixed, meaning their position is locked relative to the global origin. Hence, mates that resemble the desired URDF joints had to be applied.

Inertia and mass properties

Unfortunately, exporting the inertia and mass properties with sw_urdf_exporter was not straightforward either. AMV used a file format called STEP (Standard for the Exchange of Product Data) to share the CAD model. This file format does not contain material information, which SolidWorks needs to calculate the mass and inertia of parts and assemblies. When opening STEP in SolidWorks, a default material is assigned to all parts and assemblies of the imported model. This material was very different from the actual material, hence the mass and inertia properties were completely wrong.

It was discovered that inertia and mass were available when opening the STEP file in Inventor, which is the CAD software utilized by AMV. Inventor did seemingly not need the material information, presumably because the STEP file was exported from Inventor in the first place.

A handful of different approaches to the issue were explored, but the final solution was of the simple kind. Get the inertia and mass for each assembly using Inventor, and manually type it into the respective link in the URDF. However, a couple of challenges had to be overcome first.

The assemblies did not correspond properly to the URDF links. This is the same problem as described above in *Links and Joints*, but for Inventor this time.

All assemblies have a local origin that is defined within the assembly. These origins were not intuitively placed in the original model. In the URDF, a link's origin is placed at the origin of the joint that connects the link to its parent link. Hence, the assembly origins were redefined to match the joint origins in SolidWorks before exporting to URDF. Refer to Figure 3.4 of section 3.2 for a visual illustration of the origins of a URDF. The inertia matrix in a URDF is defined with respect to a specified Cartesian point, which by convention is the center of mass for the respective link. This point is again defined relative to the origin of the link. Inventor can provide the inertia with respect to a few different points, the center of mass of the relevant assembly being one of them. Just like in the URDF the center of mass is defined relative to the local origin. Hence, the local origins of the assemblies were redefined in Inventor to match the SolidWorks model and URDF format.

If the origins were refined in Inventor before exporting SolidWorks, the origins would not have to be redefined in SolidWorks. This would halve the total time spent on this issue.

An alternative way to get inertia properties would be to manually define the material of all assemblies and parts in SolidWorks. How time-consuming this process would be is entirely dependent on the desired accuracy of the inertia properties. Only defining material for larger parts could result in sufficiently accurate inertia.

Simplifications

The robot is equipped with a protective cover to prevent damage to the boom from small rocks and other debris. This cover is depicted in green in Figure 4.1b. However, it was not included in the URDF model due to its flexible nature and the limitations of URDF in handling non-solid objects. Additionally, connecting the cover to boom_3 would have resulted in boom_3 having two

parents (boom_2 and the cover), which is not allowed in URDF as a link can only have one parent. Despite this simplification, the accuracy of the model was not significantly affected, as the cover is relatively light compared to the rest of the links.

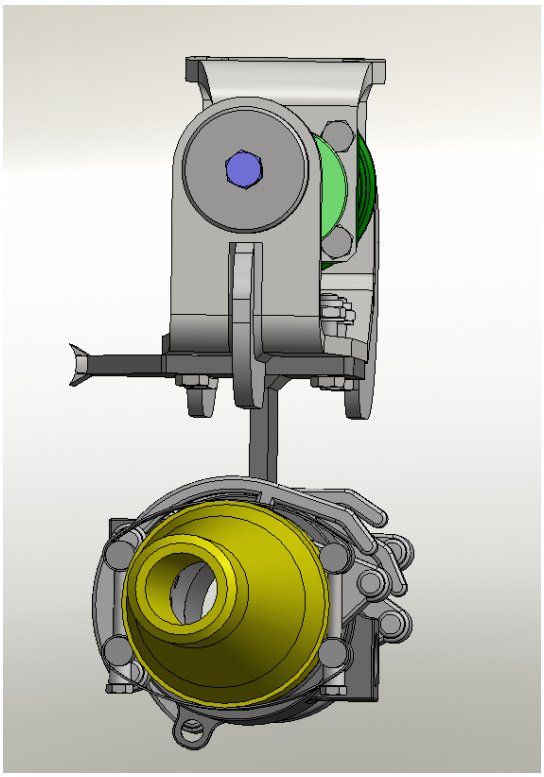
All links included in the URDF were treated as rigid bodies without considering flexibility or vibrations. The simplification of ignoring these factors was made without conducting physical tests to validate its accuracy. However, the engineers at AMV approved this approximation, and as a result, no research was conducted on how non-rigid bodies could be modeled within the framework of ROS. It should be noted that the URDF format does not support the representation of non-rigid bodies. This could present challenges if such models are to be incorporated as alternative approaches or workarounds would need to be explored.

Boom_2_link and boom_3_link were modeled as two separate links in the URDF. However, their joints cannot be actuated independently. To address this the parent joint of boom_3_link, referred to *ptBoom3*, was configured to mimic the parent joint of boom_2_link, referred to *ptBoom2*. Consequently, the linear displacement of *ptBoom2* is copied to *ptBoom3*. Hence, the total displacement of the boom will be two times the displacement of *ptBoom2* (or *ptBoom3*). In this setup, only *ptBoom2* can be directly actuated, while *ptBoom3* will be indirectly actuated. It was discovered that this model does not accurately represent the actual behavior of the boom. In reality, the links are extended more or less sequentially, with boom_3 starting extension after boom_2 is fully extended. This inaccuracy in the model will affect the accuracy of the simulation as the inertia of the boom links will be misplaced.

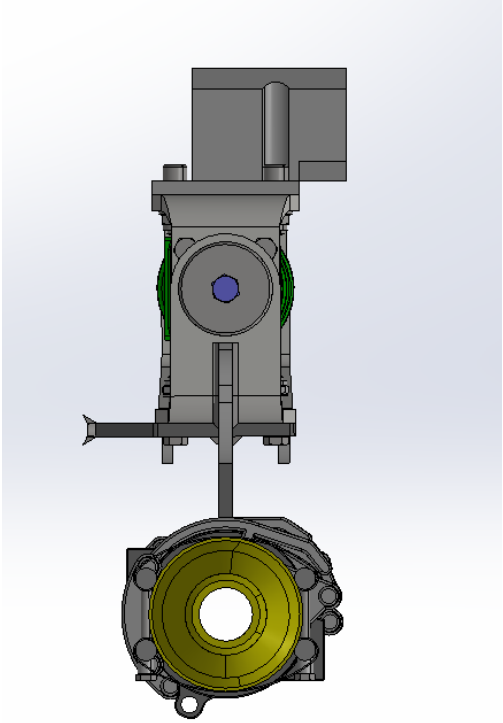
Section 2.3 describes how the nikker applies a special motion to the nozzle, improving the robot's ability to spread shotcrete evenly on a surface. Due to the complexity of this mechanism, it was not included in the URDF. Instead, the nozzle and the nikker were combined into one link. As depicted in Figure 4.3 the bolt (blue) has been aligned concentrically with the nikker's rotor (green). The new nozzle pose is fixed relative to the nikker and can be thought of as an average of the pose space the nikker would generate.

Notes on sw_urdf_exporter and alternative tools

sw_urdf_exporter is open source and maintained by members of the ROS community. The available documentation [45] is targeted at ROS 1 and has not been updated since 22nd December 2021. One notable exception was the launch file generated for testing the URDF in Gazebo, which required modifications to ensure compatibility with newer versions of Gazebo and ROS. Similar tools for other CAD software, such as Inventor, were found on GitHub as open-source projects. These were generally poorly documented and had not been updated for several years, but could be worth testing out if one wishes to use an alternative to Solidworks.



(a) Original nikker and nozzle. The bolt (blue) is mated **eccentrically** to the nikker's rotor (green), resulting in an offset angle and position for the nozzle.



(b) Modified nikker and nozzle. The bolt (blue) is mated **concentrically** to the nikker's rotor (green), aligning the nozzle with nikker's base.

Figure 4.3: Original vs modified nikker and nozzle mates

4.3.5 Making a collision model

When creating a collision model, there is a balance to strike between the accuracy of the model and performance. An accurate collision model will let the robot interact more realistically with the simulated world. On the contrary, a complex model will require more computations to determine contact between objects. How to balance accuracy and performance will depend on the robot application. The geometry of the 4200 Shotcrete ensures that it cannot collide with itself (collision between links), given that joint limits are properly defined in the URDF. Hence, the only purpose of the collision model is to detect and preferably avoid collisions with the environment. This means that the model can be very simple without compromising its function.

A visual mesh for each link was generated by the sw_urdf_exporter. As described in Section 3.2.2 these meshes are used by visual tools, such as Rviz and Gazebo, to present a visual model of the robot in the user interface. For some applications, a copy of the visual model can be used for the collision model. This is usually fine if the visual model is made up of simple shapes. However, visual meshes generated from sw_urdf_exporter were extremely detailed, resulting in horrendous performance. This means that the meshes cannot be used as collision elements in their current state. Three solutions to this problem were explored:

1: Defining collision geometry manually

This option involves editing the URDF file to specify the collision geometry as a combination of basic shapes such as boxes, cylinders, and spheres instead of using meshes. A drawback of this approach is that there is no quick and easy way to get visual feedback while editing the URDF file. As a result, checking that the model turns out as intended requires a lot of testing.

This option requires modifying the URDF file to define the collision geometry with basic shapes such as boxes, cylinders, and spheres instead of using meshes. A drawback of this approach is that there is no suitable way to get visual feedback while editing the URDF file. As a result, checking that the model turns out as intended requires a lot of testing.

2: Making new meshes for collision in SolidWorks

Another option is to make new assemblies in SolidWorks that represent the collision model. The model should use basic shapes to minimize complexity and improve performance. By designing the collision model in SolidWorks, one can easily compare it with the actual model to make sure it turns out as intended. Then, the sw_urdf_exporter can be used as in Section 4.3.4, but with the collision assemblies instead of the robot assemblies. This will generate new mesh files and a new URDF where the collision elements created in SolidWorks will be placed in the visual tag. The content of the visual tags can simply be copied and pasted into the collision tag of the original URDF. With sw_urdf_exporter it is easy to make sure that collision meshes are defined with respect to the same origins as the visual tags, and hence the meshes can be used in the original URDF without modifications.

Making meshes for collision in SolidWorks is a good option for people who are familiar with SolidWorks, but were not chosen for this project due to the lack of SolidWorks experience. Readers interested in this option should check out Shay Sackett's [46] YouTube series, SolidWorks to URDF Using the SW2URDF.

3: Mesh simplification with MeshLab

This is the solution that was implemented in the project. Adam Conkey [22] illustrates how to simplify visual meshes for collision detection using MeshLab. The collision meshes should have as few faces as possible, ideally less than 100. Figure 4.4 shows the resulting simplified meshes from Conkey's method, which is summarized below.

MeshLab was used to generate a convex hull for each mesh. A convex hull is the minimal surface that encloses the mesh entirely. Think of it as covering the mesh with plastic, while the goal is to minimize plastic use. The resulting surface of plastic enclosing the mesh represents the convex hull.

The mesh was further simplified with MeshLab using *Quadratic Edge Collapse Decimation*. This method reduces the number of faces while preserving the shape. It does this by collapsing edges in the mesh based on how they affect the geometry and topology of the mesh.

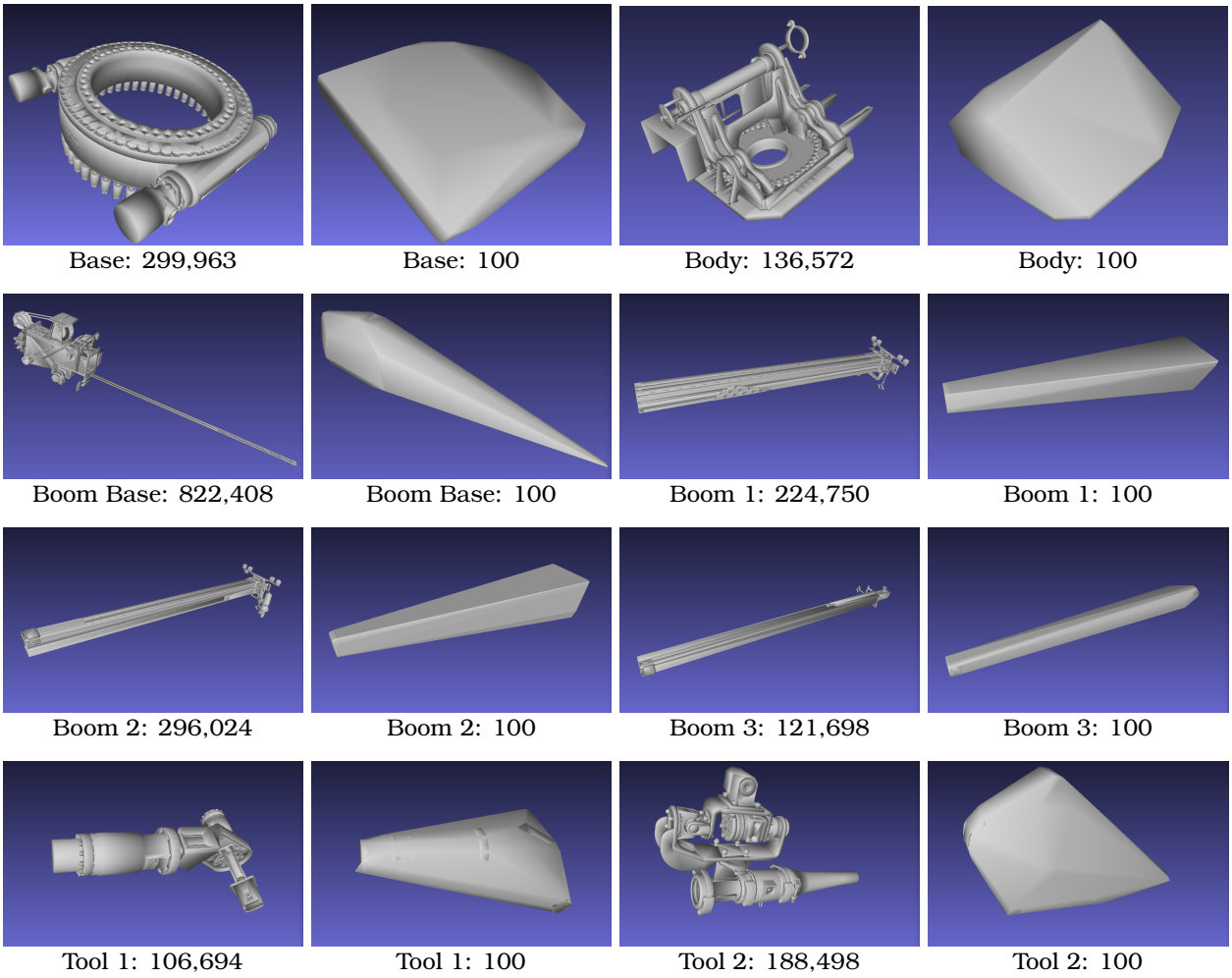


Figure 4.4: This figure shows the original and simplified meshes of each link, obtained using MeshLab. The number of faces in each mesh is indicated below the corresponding link.

The main advantage of this method is development speed. Generating simplified versions of the visual meshes is done in no time using Meshlab, given that the visual meshes are available. However, this method has drawbacks. With the other two methods, the collision meshes could be more accurate with fewer faces. This is evident in the Boom Base link (see Figure 4.4). Utilizing one of the other methods, two boxes could form the collision model, one for the cables and one for the rest of the link. This model would be more precise than the simplified visual mesh and have only $2 \times 6 = 12$ faces instead of 100. Figure 4.5 shows plenty of collisions between the links, which is not ideal for many robot applications. However, these limitations are not significant in the case of the 4200 Shotcrete. The collision meshes are accurate enough to avoid contact with the environment, and simple enough for Gazebo to simulate the robot in real-time.

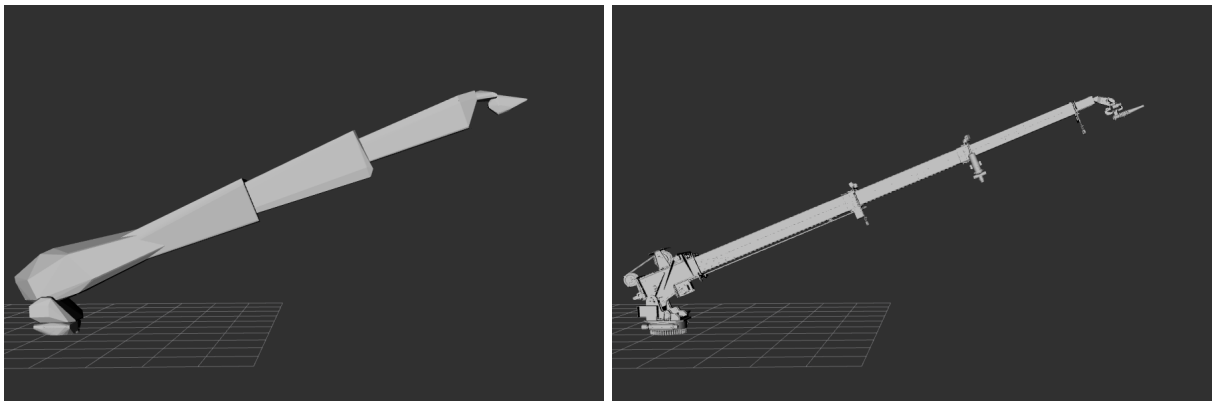


Figure 4.5: A comparison of the visual model and the final collision model made based on mesh simplifications.

4.3.6 Conclusion

Exporting the URDF from a CAD model is highly recommended for the engineering team at AMV. The CAD models are already available, and sw_urdf_exporter is an excellent tool for this purpose. However, the model had to be modified in Solidworks before the exporter could be utilized. Additionally, sw_urdf_exporter was unable to retrieve the inertia properties from the CAD model. Inventor was used for obtaining these properties. These issues made the workflow from the CAD model to URDF more cumbersome than expected. Some suggestions to improve the workflow were presented in 4.3.4.

When it comes to the collision model, there is no definitive best approach. Simplifying the visual meshes requires less effort, but generally results in a more complex and less accurate collision model compared to making the collision model manually. These pros and cons should be considered when deciding on the approach. However, using simplified visual meshes as collision meshes satisfy the requirements for the AMV 4200 Shotcrete Robot.

4.4 MoveIt Setup

To utilize MoveIt with a custom robot, it is necessary to create a MoveIt configuration package that includes all the information required for MoveIt to function effectively in a given environment. This section outlines the process of generating a configuration package using the MoveIt Setup Assistant, based on the URDF created in Section 4.3. Additionally, this section addresses challenges encountered when setting up MoveIt, and the corresponding solutions.

4.4.1 Method

MoveIt for Humble was installed following the installation instructions provided in the "Getting started" tutorial by PickNik Robotics [47]. The MoveIt configuration package was generated with MoveIt Setup Assistant, following the official guide [34] with some notable modifications: Self-collision was disabled between all links, the LMA kinematic plugin was used instead of KDL, and no end effector group was defined. Additionally, a virtual link was added to represent the tip of the nozzle. The configuration was tested using the generated *demo.launch* file from the Setup Assistant, with *LC_NUMERIC=en_US.UTF-8* appended to the launch command to avoid conflicts with local settings and MoveIt.

4.4.2 Installation

MoveIt was installed as described in the tutorial, "Getting started", by PickNik robotics [47]. Be aware that this tutorial describes how to install a version of MoveIt that is compatible with the ROS distribution, Humble.

4.4.3 Setup Assistant

In order to generate the configuration package for the AMV 4200 Shotcrete, the Setup Assistant was utilized following the tutorial by PickNik Robotics [34]. However, as the tutorial uses a different robot as an example, certain settings needed to be adjusted to match the AMV 4200 Shotcrete. The upcoming paragraphs will discuss the configurations that differ from the tutorial. This is not intended to replace the tutorial of the Setup Assistant, but rather to provide readers with enough information to recreate the configuration themselves and to justify the choices made in the setup process.

Self Collision

During motion planning, MoveIt checks for collisions between robot links (self-collision) and the environment. Collision checking is a computationally expensive operation and can account for nearly 90% of the total cost [48]. MoveIt Setup Assistant simplifies the process by generating a collision matrix that defines which links should be checked for collision. Minimizing the number of link pairs to be checked can significantly reduce the planning time. Thanks to the robot's geometry, none of the links are capable of colliding with each other. This presupposes that the joint limits in the URDF are set correctly. As a result, MoveIt was configured to not check for self-collision between any links whatsoever.

Planning group

Planning groups are defined by a collection of joints that must be taken into consideration during motion planning. For the purpose of controlling the pose of the nozzle tip, all the joints must be considered. Hence, all joints are included in the planning group. The *kdl_kinematics_plugin* was initially chosen as the kinematic solver and used the default settings for search resolution (0.005) and search timeout (0.005 sec). However, the plugin was replaced with the *lma_kinematics_plugin* at a later stage because it performed significantly better when implemented in the script described in Section 4.5.

End effector

No valid reason to define an end effector was found during the project. Regular planning groups seem to work fine, despite the Setup Assistant throwing a warning message when generating a configuration package without end effectors. However, defined end-effector groups might be useful in some unforeseen way.

The remaining configurations follow the default suggestion from the tutorial by PickNik Robotics. Note that readers with access to the project's source code can open the configuration package with the Setup Assistant to inspect the configurations through the GUI.

4.4.4 Problems

After the configuration package was generated, there were some errors and issues that had to be addressed before continuing.

Locale settings

When attempting to run the demo launch file generated by the Setup Assistant, the robot would not be loaded. The same problem was faced when going through the tutorial, MoveIt Quickstart in RViz by PickNik Robotics [49]. The listing below shows the relevant error message that was returned when attempting to run the demo.

```
Exception caught while processing action 'loadRobotModel':
parameter 'panda_arm.kinematics_solver_timeout' has an invalid type:
Wrong parameter type, parameter {panda_arm.kinematics_solver_timeout}
is of type {double}, setting it to {string} is not allowed.
```

Listing 4.1: Error message caused by wrong locale settings

The error was caused by a locale setting of the terminal being incompatible with MoveIt. More specifically, the *LC_NUMERIC* setting was set to *nb_NO.UTF-8*, which defines decimal numbers (like doubles and floats) using a comma. In contrast, the *loadRobotModel* action expected a period as the decimal separator, resulting in the action interpreting the *kinematics_solver_timeout* parameter as a string rather than a double.

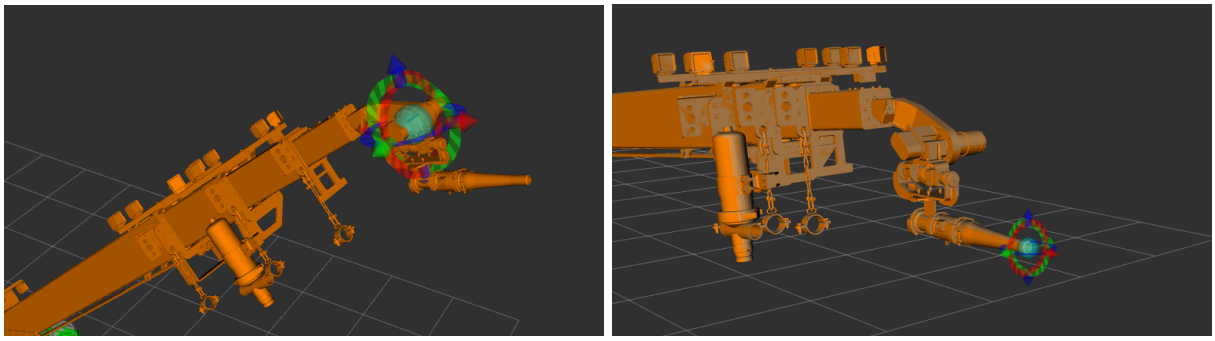
Locale settings can be appended when running a command in the Ubuntu terminal. The first command in the listing below is an example of how this can be done when launching a launch file. This will address the error described above. Alternatively, the second command will make sure *LC_NUMERIC=en_US.UTF-8* for all new instances of a terminal. Hence, the setting does not have to be appended every time MoveIt-related software is called from the terminal.

```
1: LC_NUMERIC=en_US.UTF-8 ros2 launch <package_name> <launch_file_name>
2: echo "export LC_NUMERIC=en_US.UTF-8" >> $HOME/.bashrc
```

Listing 4.2: Two ways of setting the LC_NUMERIC setting on Ubuntu using the terminal.

Unintuitively placed motion request marker

MoveIt was initially configured to plan motions for the origin pose of *tool_2_link* (as shown in Figure 4.6a). However, the tip of the nozzle should be the point of interest for motion planning. To address this, a new link was added to the URDF, fixed to the tip of *tool_2_link*. This new link acts as a coordinate frame and does not affect the robot model in any way, since it does not have any elements such as visual or inertia properties. By including this link in the configurations, MoveIt can now accept motion planning requests for the tip of the nozzle, making the program more intuitive to work with.



(a) The goal pose marker is placed is at the origin of *tool_link_2*.

(b) The goal pose marker is moved to the origin of the new link/frame *nozzle_tip_frame*

Figure 4.6: The interactive marker in Rviz was moved to better to a more intuitive location

4.4.5 Running the Demo

The Setup Assistant generates a demo launch file, that can be used to test the configurations with Rviz. Figure 4.7 shows a snapshot of the demo in action. A dedicated panel for MoveIt called MotionPlanning can be seen on the left side of Rviz. This panel offers a lot of functionality and options. A trajectory between the start state (in green) and the goal state (in orange) can be generated and executed using the corresponding buttons. The user can define a goal pose and start pose for the nozzle by moving the interactive markers located at the nozzle tip. MoveIt will continuously solve the inverse kinematics to display the robot in a state that fulfills the desired nozzle position defined by the markers. Importantly, the setting "Approx IK Solutions" is enabled, due to the robot having 5 degrees of freedom, which prevents it from physically achieving many

nozzle poses defined in the full task space. Unfortunately, this aspect was not addressed in the documentation, exemplifying the overall lack of guidance for implementing robots with fewer than 6 degrees of freedom in MoveIt.

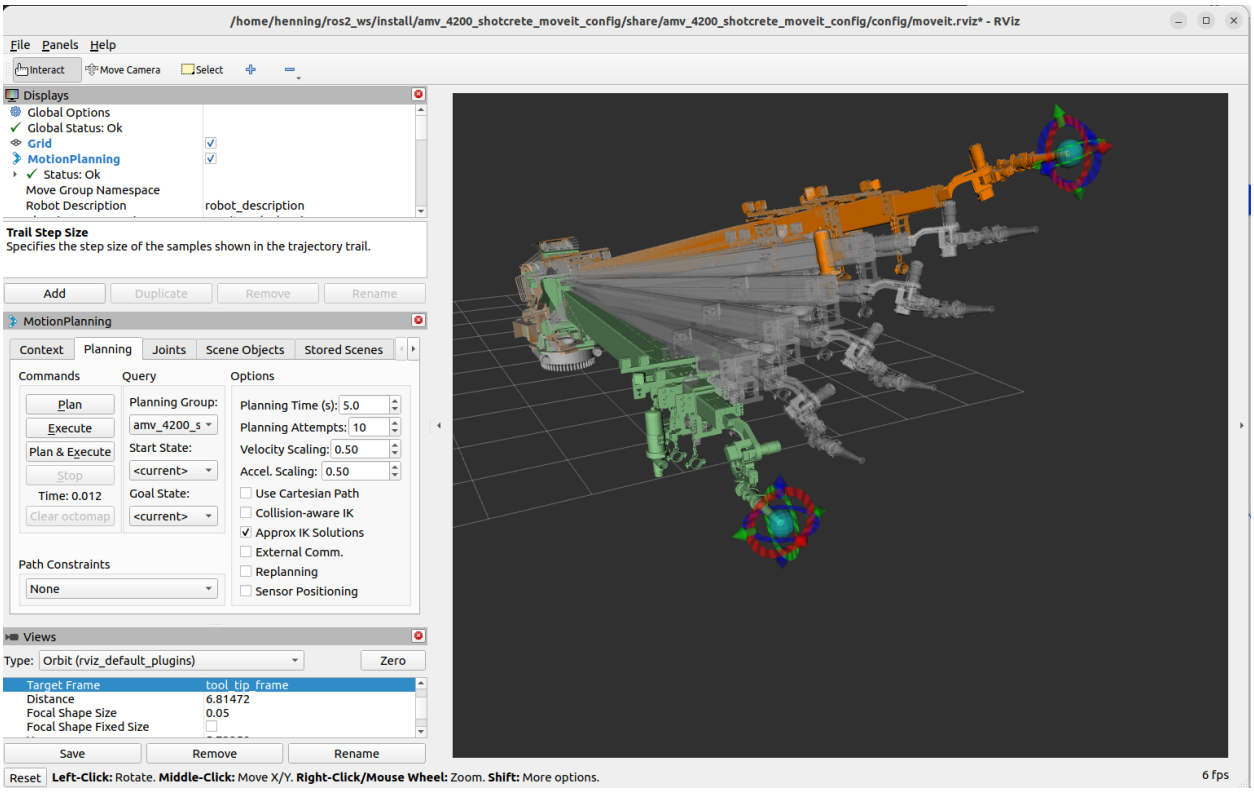


Figure 4.7: Motion planning with MoveIt using the Rviz interface. Orange represents the goal state and green is the start state. The planned path is shown in white as intermediate states between the start and goal state. Markers at the nozzle tip can be dragged to redefine the start and goal state.

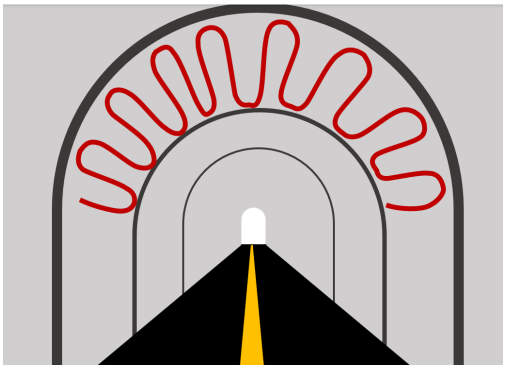
4.4.6 Conclusion

The integration of the AMV 4200 Shotcrete with MoveIt was accomplished by generating a MoveIt configuration package for the robot using the Setup Assistant and following the official tutorials. The Setup Assistant played a crucial role in streamlining the setup of the necessary configuration package for MoveIt. To validate the configurations, simple motion planning was performed using the *demo.launch* file generated by the Setup Assistant.

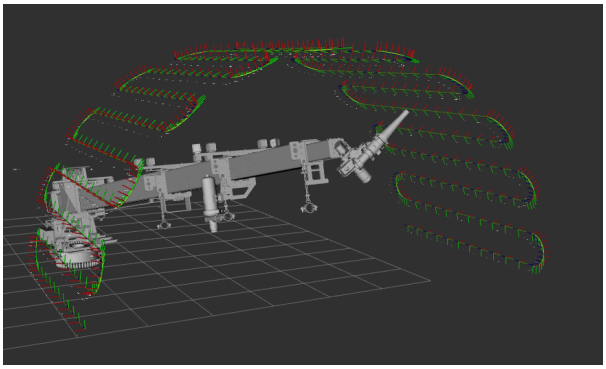
The overall process of integrating the AMV 4200 Shotcrete robot with MoveIt proceeded smoothly, largely due to the valuable guidance provided by the tutorials. However, it is important to highlight that certain issues arose during the implementation that was not addressed in the MoveIt documentation. Specifically, there was a lack of guidance for robots with less than 6 degrees of freedom and an issue with local language settings on the operating system. The advice to AMV engineers is to utilize the community through the MoveIt Github forums when facing such problems.

4.5 Motion Planning in MoveIt

This section explores the utilization of MoveIt for a typical use case of the 4200 Robot, which involves following a path that resembles a spraying motion, depicted in Figure 4.8a. MoveIt offers multiple interfaces for scripting robot motion planning, including the Move Group Interface and C++ API, both of which were evaluated during the course of this project.



(a) Illustration of a typical path the nozzle needs to follow during a spraying procedure.



(b) The test waypoints (430 in total) illustrated in Rviz. Notice the x-axis (red) is always pointing outwards.

Figure 4.8: The test path represents a spraying procedure.

4.5.1 Method

The final software utilizes the C++ API of MoveIt and is inspired by the C++ API tutorials. RRT-Connect of the Open Motion Planning Library (OMPL) is used for motion planning and the LMA kinematics plugin for inverse kinematics. The motion planner generates separate paths between each waypoint, which are subsequently merged into a single path. In order to meet the specified velocity constraints defined in the configuration package, the TimeOptimalTrajectoryGeneration class incorporates a time parameter into the path, resulting in the final trajectory.

A virtual degree of freedom was implemented in the URDF to let the motion planner and kinematics interpret the robot to have 6-DOF. The virtual DOF is ignored when executing the planned trajectory.

4.5.2 Generating a test path

Prior to creating the path-following script, it was necessary to define a test path in a format compatible with the MoveIt interfaces. This was accomplished by developing a custom function that accepts tunnel parameters and generates a set of waypoints defined as poses (position and orientation). These waypoints are illustrated in Figure 4.8b using Rivz. The direction of the nozzle is indicated by the x-axis (red), which can be confirmed by referring to Figure 4.6 or 4.7. It is essential that the nozzle is pointing outwards, and hence the direction of the x-axis is crucial. However, the y- and z-axis direction does not matter. This means the desired orientation could be sufficiently defined by two rotations. However, MoveIt expects orientations to be defined as quaternions, and no alternative method was found. As a result, the orientation of the waypoints had to be fully defined, leading to several issues that will be discussed below.

4.5.3 The Limitation with a 5-DOF Arm

As mentioned in Section 2.2 the AMV 4200 Shotcrete is a 5-degrees-of-freedom robot, indicating that not all poses defined in the full task space are physically attainable for the end effector. Consequently, a series of 6-DOF waypoints would inevitably include several poses that are unreachable.

Note that any rotation about the x-axis of the nozzle would still result in a pose that satisfies the requirement of the nozzle pointing outwards. Additionally, it was hypothesized that there exists a rotation about the nozzle's x-axis that results in a reachable pose. Taking this into consideration, various approaches were contemplated to address this problem:

The first option involved carefully selecting a valid rotation about the x-axis for each waypoint prior to providing them to the motion planner. However, determining this rotation value is not a straightforward process and would likely involve performing inverse and forward kinematics calculations, which would undermine the primary purpose of utilizing MoveIt. Consequently, this option was not pursued further.

The second alternative was to disregard any discrepancy between the x-axis rotation of the waypoints and the planned trajectory. MoveIt's motion planners utilize an input parameter called Goal Tolerance, which defines the maximum permissible error between the requested pose (Goal Pose) and the final pose of the generated trajectory. In our case, each waypoint serves as a goal pose. By setting the Goal Tolerance for the x-axis rotation to a value greater than $\pm\pi$, any rotation would be deemed acceptable. This approach was tested using the Move Group Interface and the C++ API of MoveIt.

The third alternative involved introducing a virtual degree of freedom to the robot model, enabling rotation about the x-axis in the URDF. This modification allows MoveIt to process requests in the full task space and generate valid trajectories. When applying the resulting trajectory to the controllers, the virtual joint added in the URDF would simply be ignored. This approach was only explored using the C++ API.

4.5.4 Goal Tolerance

This alternative did not yield satisfactory results. However, an analysis of the underlying reasons for these poor outcomes will be discussed in this subsection. Additionally, several different approaches to addressing the issues will be presented.

Move Group Interface

As outlined in the 3.6, the Move Group Node is a central part of the MoveIt pipeline and acts as an interface between the user and the various components of MoveIt. One way for a user to interact with the Move Group Node is through ROS actions, services, and topics. The Move Group Interface encapsulates the actions, services, and topics, abstracting them away from the user, hence making it an excellent choice for beginners as it is relatively simple to use.

The Move Group Interface provides a useful function called *computeCartesianPath*, which can plan a trajectory that ensures the nozzle travels through a series of provided waypoints. Unfortunately, the function would always fail in our case because the waypoints were unreachable.

In an attempt to resolve this issue, increasing the goal tolerance for the rotation of the x-axis was considered. This would have allowed a trajectory to be deemed valid even if the nozzle's x-axis was rotated compared to the waypoints. However, it was discovered that the Move Group Interface only permits the same tolerance value to be set for all rotations that define the orientation. Applying a tolerance to all rotations would result in a nozzle that does not align with the wall, making the interface unsuitable for this approach.

It is worth mentioning that a proposal has been put forward to introduce a new feature in the Move Group Interface that would allow the adjustment of tolerance for individual axes [50]. As of now, this feature is set to be added to MoveIt soon. However, it should be noted that this is for a version of MoveIt designed for ROS 1 distributions. It remains unclear whether/when this feature will be incorporated into the new version of MoveIt for Humble.

C++ API

Using the C++ API of MoveIt, the user can directly access various components of MoveIt, such as the Planning Pipeline, rather than communicating with the Move Group Node through ROS actions. This grants more intricate control and better performance but at the cost of being less beginner-friendly than the Move Group Interface.

In contrast to the Move Group Interface, the C++ API offers the capability to set goal tolerances for each individual axis of rotation. However, the *computeCartesianPath* function, or equivalent functions, are not available in this interface. Consequently, an alternative approach was necessary to generate a trajectory through waypoints. The implemented solution involved planning paths between each waypoint in joint space individually and subsequently "stitching" the paths together. The combined path through all waypoints is then time-parametrized to create a trajectory that can be used as a reference for joint controllers. It is important to acknowledge that this solution is not optimal and could be significantly improved in terms of performance and quality. Nonetheless, it served as an investigation into whether setting a high goal tolerance for the x-axis rotation would enable MoveIt to plan suitable trajectories for the AMV 4200 robot.

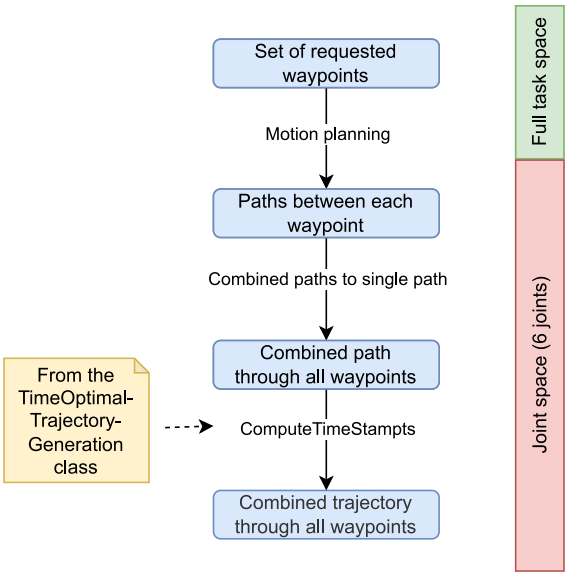


Figure 4.9: Illustration of the trajectory generation process.

The goal tolerance was set to $\pm\pi$, allowing a trajectory to be considered valid even if the nozzle's x-axis rotation deviates by π in both directions. Consequently, any rotation about the x-axis would yield a valid trajectory. Using this approach, the motion planner usually generated valid trajectories between individual waypoints, although occasional failures occurred. When trying to plan a combined trajectory through the set of test waypoints, the motion planner would seldom reach far. Table 4.1 presents the results of 10 planning attempts using the same script and conditions. Despite this, a significant variability can be observed in the outcomes.

The data presented in Table 4.1 clearly demonstrates the ineffectiveness of this approach. Out of 10 attempts to generate a trajectory through 430 waypoints, the most successful attempt only reached 9 waypoints. The average planning time further highlights another issue. Even if the full trajectory were successfully planned, it would take an estimated time of $8.2sec \times 430 \approx 59min$. Such a long execution time is unacceptable for this application.

Attempt	Waypoints reached	Total Time	Average planning time (excluding the failed attempt)
1	9	100	8.778
2	1	34	13
3	4	45	6
4	1	24	3
5	3	42	7
6	2	53	16
7	1	26	5
8	0	21	-
9	2	33	6
10	0	21	-
Total	23	399	8.217

Table 4.1: The result of 10 attempts at planning a trajectory through all 430 test waypoints. The planner was terminated if it did not reach the next waypoint within 20 seconds. These 20 seconds were not included in the average planning time for successful plans in the rightmost column.

What is the issue?

When the planner failed, it consistently returned one of the two error messages listed below. The source of the error can be traced back to the OMPL library, specifically its implementation of the RRTConnect algorithm, which is a variation of the RRT algorithm. The RRT (Rapidly-Exploring Random Tree) algorithm efficiently explores the joint space by incrementally growing a tree of randomly generated configurations. Each new configuration is connected to the existing tree, biasing sampling towards unexplored areas. This iterative process generates a tree structure that represents a potential path for the robot to follow. Eventually, the RRT selects a robot configuration that positions the end-effector within the goal tolerance, enabling the traversal of the tree to find a path from the start to the goal. RRTConnect, an extension of RRT, simultaneously grows two trees, one rooted at the start state and the other at the goal state. This approach often outperforms the classical RRT algorithm [31]. While there is much more that can be said about these algorithms, the crucial point to understand is that the RRTConnect needs a valid goal state to build one of the two trees.

```
[motion_planning_api-1] [ERROR] [ompl]:
./src/ompl/geometric/planners/rrt/src/RRTConnect.cpp:252 -
amv_4200_shotcrete/amv_4200_shotcrete:
Unable to sample any valid states for goal tree

[motion_planning_api-1] [ERROR] [ompl]:
./src/ompl/geometric/planners/rrt/src/RRTConnect.cpp:213 -
amv_4200_shotcrete/amv_4200_shotcrete:
Insufficient states in sampleable goal region
```

Listing 4.3: Error messages from during motion planning

Motion planning algorithms like RRTConnect commonly operates in joint space, while the goal is typically specified in the full task space, defining the desired position and orientation of the end effector. To address this, kinematic solvers are utilized to convert the task space input into a suitable joint space goal for the planner. Given the wording of the error messages, it is plausible that the problem stems from the kinematic solver. If the solver fails to accurately convert the goal into valid configurations in joint space, it could explain the occurrence of these errors.

Figure 4.10 illustrates the role of the kinematic solver in regard to this issue. A subset of valid goal poses is stochastically selected from the entire set of valid goal poses (goals within the tolerance). These poses are then provided to the kinematic solver for inverse kinematics computation. Successful computation yields valid goal states that can be utilized by RRTConnect. Even if the high goal tolerance of the x-axis rotation guarantees that there exists a goal pose that has a valid state representation (the inverse kinematic solution exists), there is still no guarantee that such a goal pose is chosen in a finite number of attempts. Hence, there is no guarantee that a valid goal state is found. This interpretation of the problem is derived from the examination of the source code of MoveIt's OMPL interface [51] and should be regarded as a tentative understanding rather than a definitive conclusion.

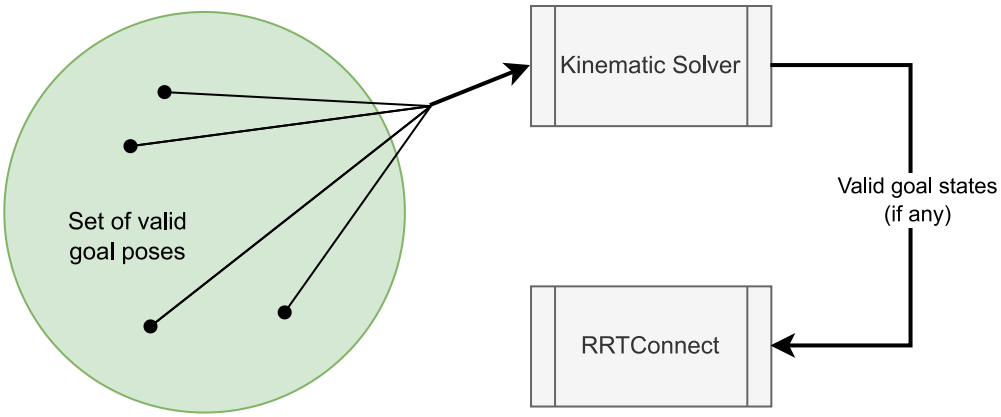


Figure 4.10: A set of goal states that are within the defined tolerance are provided to the kinematic solver. The kinematic solver finds the joint space equivalent of the goal poses (if possible) and sends them to RRTConnect.

Several forum posts discussing similar problems have highlighted the issue of kinematic solvers when dealing with arms that have fewer than 6 degrees of freedom. Michael Görner (username v4hn), a core contributor to the MoveIt development project, claims that KDL kinematic plugin struggles with such robots and recommends using a solver called IK-Fast instead [52]. Another forum post confirms the limitations of KDL for robots with 5 degrees of freedom or less, emphasizing the necessity of exploring alternative solutions [53]. Felix von Drigalski (username fvd), an active contributor, acknowledges the widespread difficulties users encounter with inverse kinematics in MoveIt when working with a low degree of freedom arms [54].

The Kinematics and Dynamics Library (KDL) is the default kinematics plugin for MoveIt, and as emphasized, does not work well with the 5-DOF shotcrete robot. This was experienced firsthand when the C++ API script was initially tested with KDL. This was never observed to reach even one waypoint. Hence, the LMA plugin was used instead with the results discussed above. This indicates that the kinematic plugin plays a pivotal role in the poor results of the motion planner. Other kinematic plugins were not tested during the project.

Alternative solutions

One option is to utilize a kinematic plugin that can effectively handle goal poses defined in 5 degrees of freedom (DOF). Examples of such plugins include IK-Fast and bio_ik. The selection of an appropriate solver requires careful consideration of its advantages and disadvantages, which will not be included in the scope of the thesis. Additionally, implementing a custom solver as a kinematics plugin is another viable option worth considering.

Another alternative is to explore different motion planner algorithms. As previously mentioned, the errors arise due to the requirement of a valid goal state before initiating planning with RRT-Connect. Switching to a unidirectional planner that only necessitates the start state might help mitigate this issue. However, it is important to note that this approach primarily addresses the symptoms rather than directly resolving the underlying cause.

The last presented option is to add a virtual joint, which introduces an additional degree of freedom, to the URDF. This approach offers a distinct solution that is not reliant on adjusting the goal tolerance. Therefore, it is discussed separately in its dedicated section.

4.5.5 Adding a virtual DOF

As described in Subsection 4.4.4 a virtual link called nozzle_tip_frame was introduced into the URDF to represent the tip of the nozzle's pose. Recall that this virtual link serves as the end effector used for motion planning, and hence is the link that should pass through the waypoints. Initially, this link was connected to its parent, tool_link_2, through a fixed joint. However, to address the 5-DOF problem, a workaround was implemented by converting this joint to a revolute joint that allows rotation about the x-axis. This modification effectively provides an additional degree of freedom to the robot model. Upon integrating the updated URDF into MoveIt, the kinematics plugin is able to solve the inverse kinematics as if the robot had 6-DOF. As a consequence of making the extra virtual joint revolute, the planned trajectory will contain states for the virtual joint. However, these can easily be discarded.

This software version was implemented with the C++ API using the same "stitching" approach discussed in Section 4.5.4. Although there was not enough time left to test the virtual DOF approach with the Move Group Interface, there is no reason to assume that it would not work.

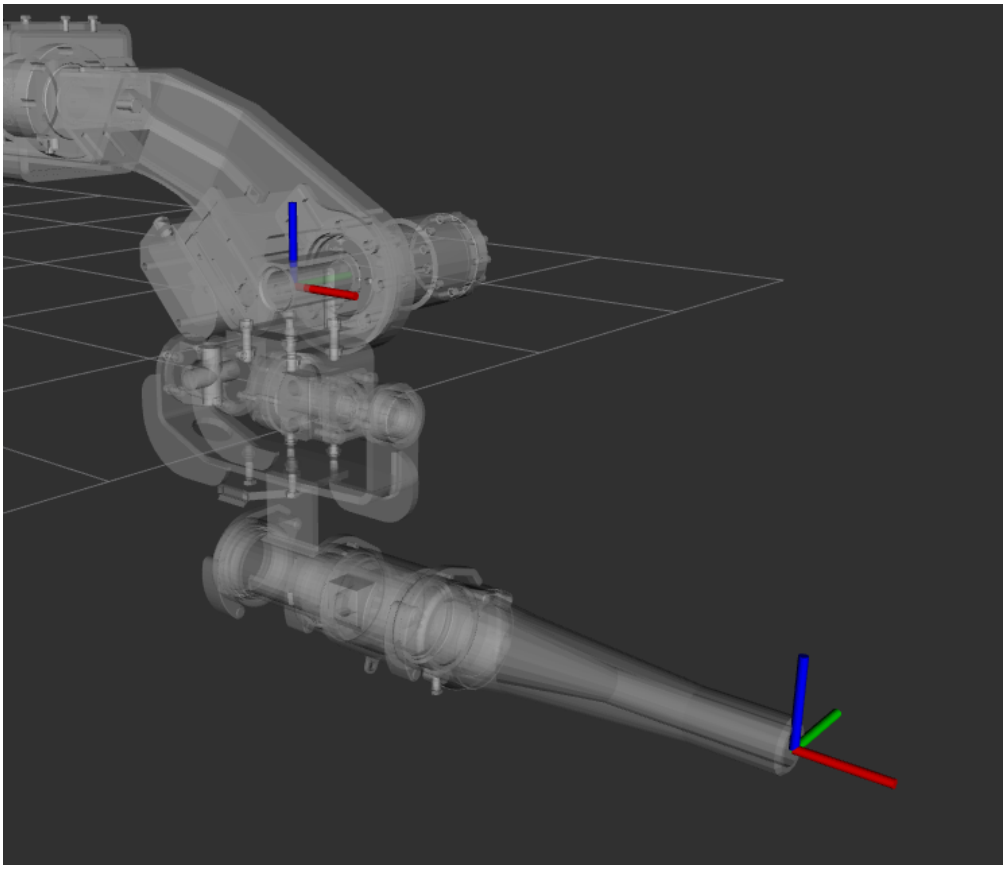


Figure 4.11: The origin frame of the virtual link and the virtual joint is shown in the bottom right, while the origin frame of the *tool_link_2* is shown in the top left corner.

4.5.6 Conclusion

Developing a motion planning script for the AMV 4200 Shotcrete robot proved to be more challenging than anticipated due to the lack of guidance in the official MoveIt documentation regarding the implementation of robots with fewer than 6 degrees of freedom. The available tutorials use a 7-DOF example robot and hence did not address the specific issues encountered during this project.

MoveIt expects the desired end effector pose to be defined in 6 DOF. However, the 4200 Shotcrete is a 5-DOF robot, meaning poses defined with 6 degrees of freedom are not generally reachable. As a result, difficulties arose with the default motion planner and kinematic plugin of MoveIt, which are designed to handle 6-DOF inputs.

Two primary approaches were explored to address this issue:

1. Setting a high goal tolerance for rotation about the insignificant axis, effectively allowing the motion planner to ignore this axis. This approach does not address the kinematic plugin. Some different options for dealing with the kinematic plugin problem have been presented but not tested, including changing to a plugin better suited for 5-DOF applications.

2. The other approach presented is to implement a virtual degree of freedom into the robot model, specifically allowing rotation about the insignificant axis. This modification allows MoveIt to process requests in 6-DOF and generate trajectories in 6-DOF, effectively bypassing the issue. The resulting trajectory will include positions for the virtual joint, but these values can easily be discarded. This is the recommended approach for the AMV engineers, as it requires no changes to default settings, was simple to implement, and allows for a wider range of kinematics plugins and motion planners to be utilized.

MoveIt provides multiple interfaces, two of which were tested during the project. While the Move Group Interface is beginner-friendly, the C++ API offer significantly more control and performance. Only the C++ API was tested with the final script, demonstrating acceptable results. Although it is assumed that the Move Group Interface is also capable of delivering satisfactory outcomes, there was insufficient time to test it thoroughly. Getting familiar with the C++ API is recommended despite the steeper learning curve, due to the extra capabilities it provides. It is worth noting that there is also a Python interface available which was not tested during this project.

4.6 Gazebo

This section will not follow the structure of the previous sections, as the work done toward implementing Gazebo was very limited. The robot can be spawned into a Gazebo world with a launch file generated by the MoveIt Setup Assistant generated, meaning a user can interact with the robot in a physical simulation. However, many key features of Gazebo were not utilized, as Motion Planning in MoveIt was prioritized. Figure 3.8 in section 3.5 illustrates how MoveIt and Gazebo could be integrated to form a closed-loop control system. However, instead of using simulated sensors and actuators from Gazebo, the current implementation of MoveIt just publish and subscribes directly to the *joint_state* topic, effectively ignoring all dynamics of the model.

Unlocking the full capabilities of Gazebo would require a few updates to the current state of the URDF. Realistic dynamics for the joint would have to be specified. Actuator control plugins must be added for each joint, and sensor plugins must be added for relevant sensors, this could include the camera and LiDAR as well as joint sensors. If the camera and LiDAR sensors on the robot are to be utilized for high-level control and motion planning, the access to a simulator that can easily be integrated with the motion planning software could be immensely valuable for testing.

To unlock the full capabilities of Gazebo, updates to the URDF would be required. This includes specifying realistic dynamics for the joints, defining actuator control plugins for each joint, and adding sensor plugins for the joints, sensors, and potential camera and LiDAR. The desired accuracy of the simulator would naturally impact the amount of work that has to be put into these updates. The engineers at AMV will have to decide if the value of a physical simulator is worth the extra effort.

Chapter 5

Results

This chapter presents the motion planning capabilities of the final software utilizing MoveIt. A test path was defined by a series of waypoints (position and orientation). It imitates a typical spraying pattern that operators of the AMV 4200 exploit. The motion planner incorporates inverse kinematics and a path planning algorithm to generate a path in joint space through the waypoints. Finally, the path is time parametrized to create a trajectory in joint space. The test path is visualized in Figure 5.1 and is described in more detail in section 4.5 together with the implemented motion planner. Planning took 15.18 seconds, followed by 1.56 seconds for post-processing. The resulting trajectory consisted of 3986 intermediate states with a time interval of 20 ms between each state, making the trajectory 79.7 seconds long.

The resulting motion of the trajectory was visualized using RViz, and it was observed that the nozzle tip accurately followed the desired path.

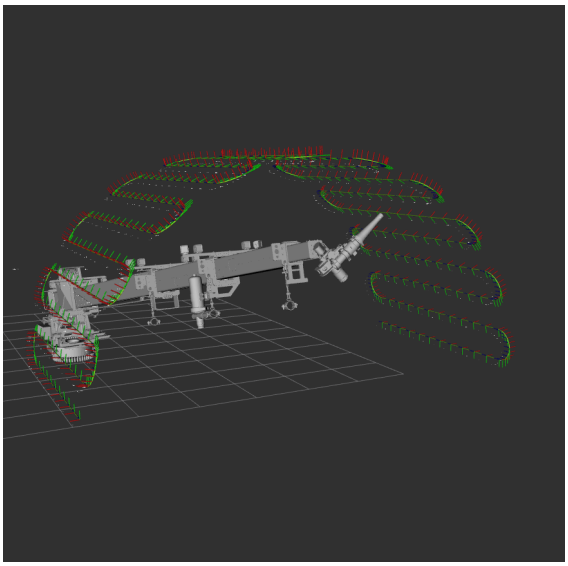


Figure 5.1: The test path resembles a half cylinder with a radius and length of 2 meters. It comprises 430 waypoints denoted by position and orientation. The first and last rows of waypoints are positioned 2 meters above the ground. Axis colors: x (red), y (green), z (blue).

5.1 Joint trajectories

The resulting joint trajectory from the motion planning is presented in Figure 5.2. Instead of a single trajectory in joint space, a separate trajectory is presented for each joint. Note that ptTip is a virtual joint and hence does not represent an actual joint of the physical robot. The ptBoom2 and ptBoom3 were combined into ptBoom, as the two original joints were constrained to have the same value. Upper and lower limits of 0.5 $rad/s = 28.65\ ^\circ/s$ were put on velocity for rotational joints and 0.5 m/s for ptBoom2 and ptBoom3, resulting in 1.0 m/s for ptBoom. It is important to note that these limits were chosen for the sake of simplicity and are not recommended for actual hardware applications. Tables 5.1 and 5.2 present the upper and lower values for each joint.

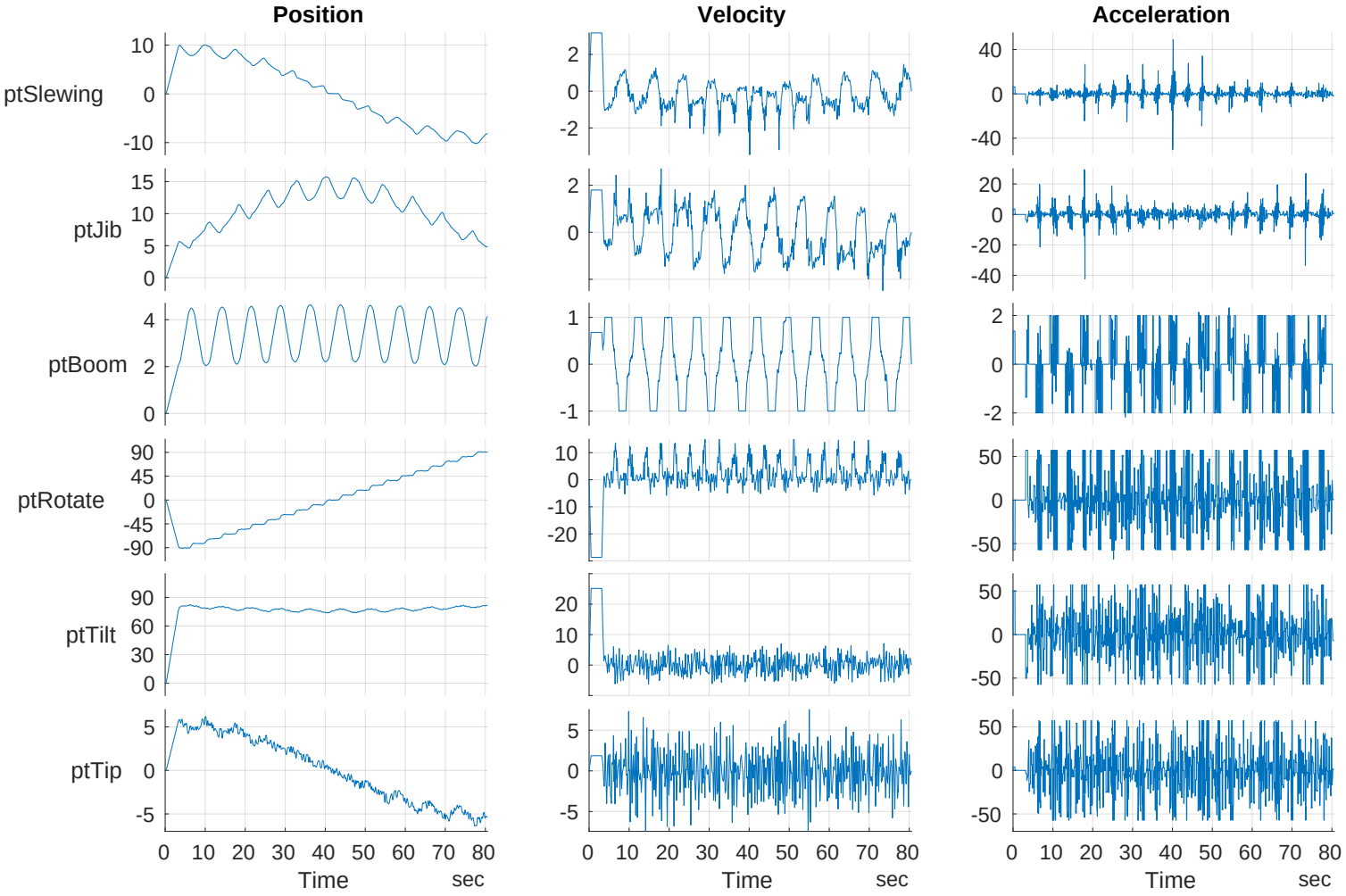


Figure 5.2: Joint trajectories

	Min Value	**Max Value**	
ptSlewing	-0.060341	0.05527	rad/s
ptJib	-0.043295	0.047135	rad/s
ptBoom	-1	1	m/s
ptRotate	-0.5	0.33167	rad/s
ptTilt	-0.10855	0.43909	rad/s
ptTip	-0.12974	0.1317	rad/s

Table 5.1: Min and max velocities of the joint trajectories.

	Min Value	**Max Value**	
ptSlewing	-0.87815	0.85144	rad/s^2
ptJib	-0.74084	0.51372	rad/s^2
ptBoom	-2.1698	2.3187	m/s^2
ptRotate	-1.1842	1.0047	rad/s^2
ptTilt	-1.0092	1.003	rad/s^2
ptTip	-1.0039	1.0038	rad/s^2

Table 5.2: Min and max accelerations of the joint trajectories.

5.2 The Nozzle in Task Space

Figure 5.3 shows the pose of the nozzle_tip_frame throughout the trajectory. The plotted values are based on the forward kinematics of the joint trajectory, calculated with MoveIt, and the time aspect of the trajectory is ignored. The trajectory orientations are originally expressed in quaternions, but have been converted to Tait–Bryan angles for easier interpretation. Hence, the orientation and angular velocity of the nozzle in Figure 5.3 are represented using roll (x-axis), pitch (y-axis), and yaw (z-axis) rotations. To avoid singularities at the relevant orientations, the sequence of rotations is as follows: Yaw, Roll, Pitch.

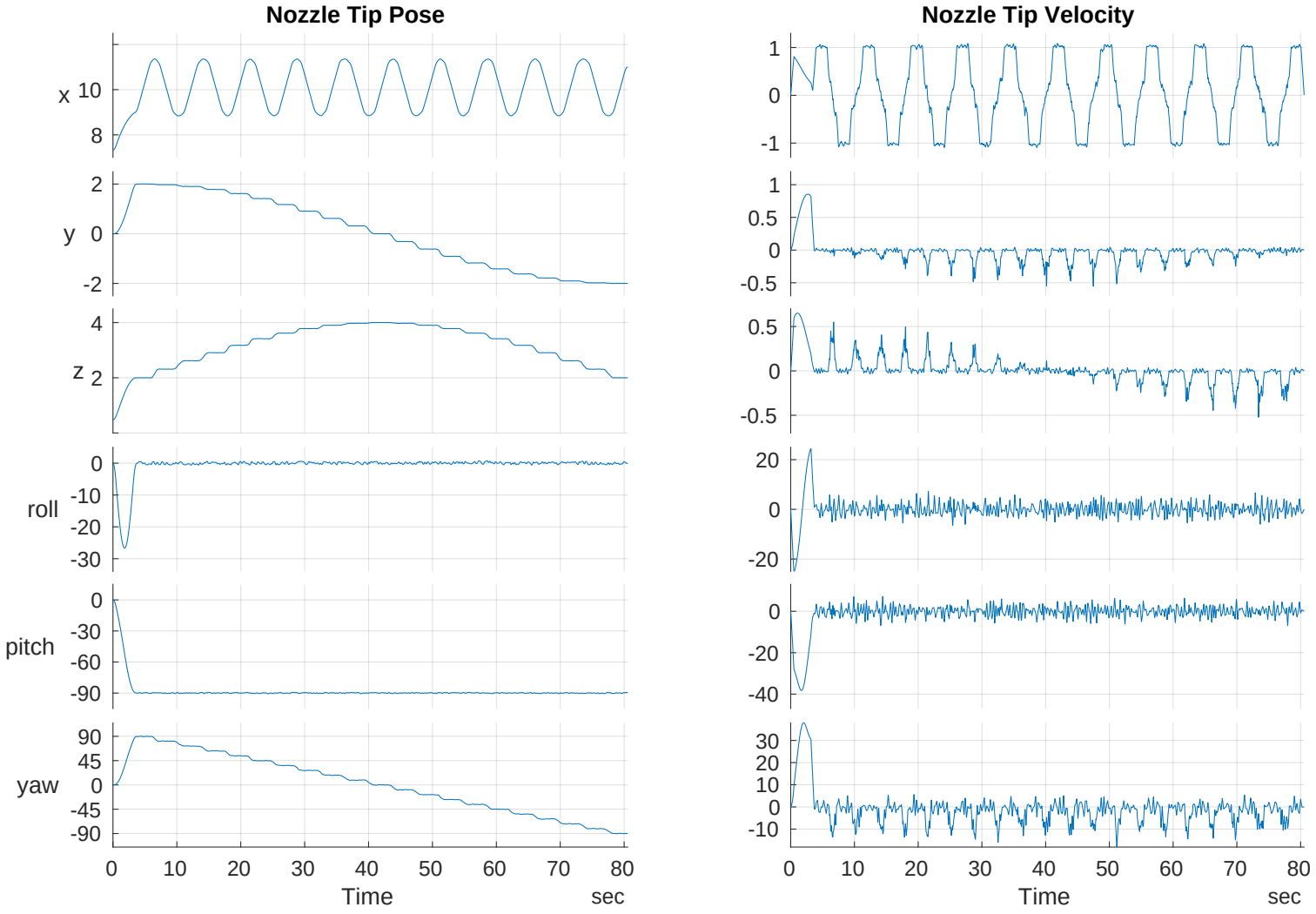


Figure 5.3: Pose and velocity of *nozzle_tip_frame*

5.3 Error between the desired path and generated trajectory

For the test, the motion planner was configured to tolerate a discrepancy between the waypoints and the trajectory of *0.01* meters for each positional coordinate and *0.01* radians for each rotation angle. This tolerance was successfully maintained for all waypoints, except for the first one. In the case of the first waypoint, the y-coordinate error was measured to be *0.01346* meters, and the yaw-angle orientation error was found to be *0.0147* radians.

Figure 5.5 shows the positional error between the trajectory and the desired path. Defining the discrepancy between them was not straightforward, as the trajectory consisted of 3986 intermediate states compared to the 430 waypoints of the desired path. How the error was defined is illustrated in Figure 5.4. The plotted positional error represents the Euclidean distance between a waypoint and a straight line connecting the two closest trajectory points of the waypoint. The orientation error plotted in Figure 5.6 is defined as the discrepancy between the waypoint orientation and an estimated orientation between the trajectory points (as illustrated in Figure 5.4). This estimate is based on linear interpolation using the orientation of the two waypoints. The total angle error is presented as $\sqrt{\text{error}_{\text{roll}}^2 + \text{error}_{\text{pitch}}^2 + \text{error}_{\text{yaw}}^2}$.

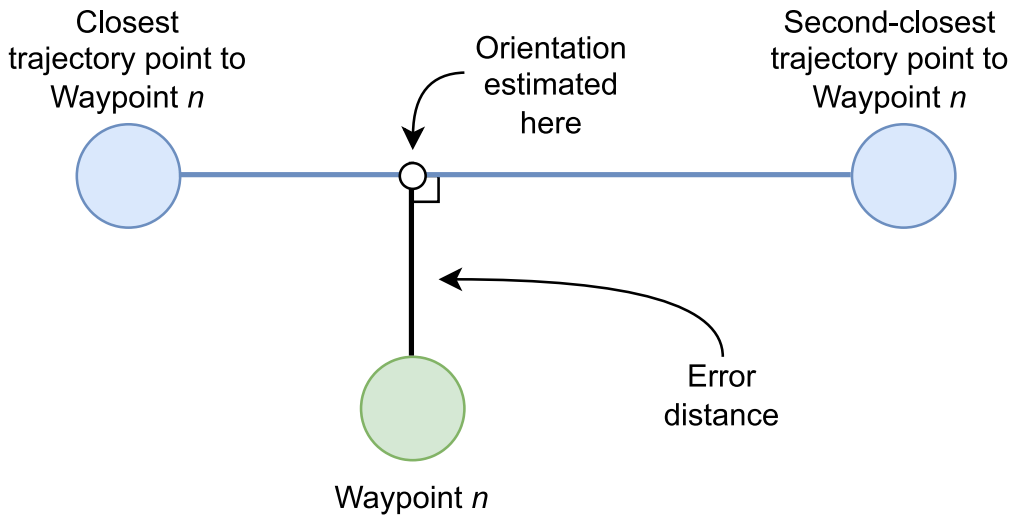


Figure 5.4: Illustration of how the error between the desired path (green) and the output trajectory (blue) is defined. Note that this error is calculated for each of the 430 waypoints in the desired path.

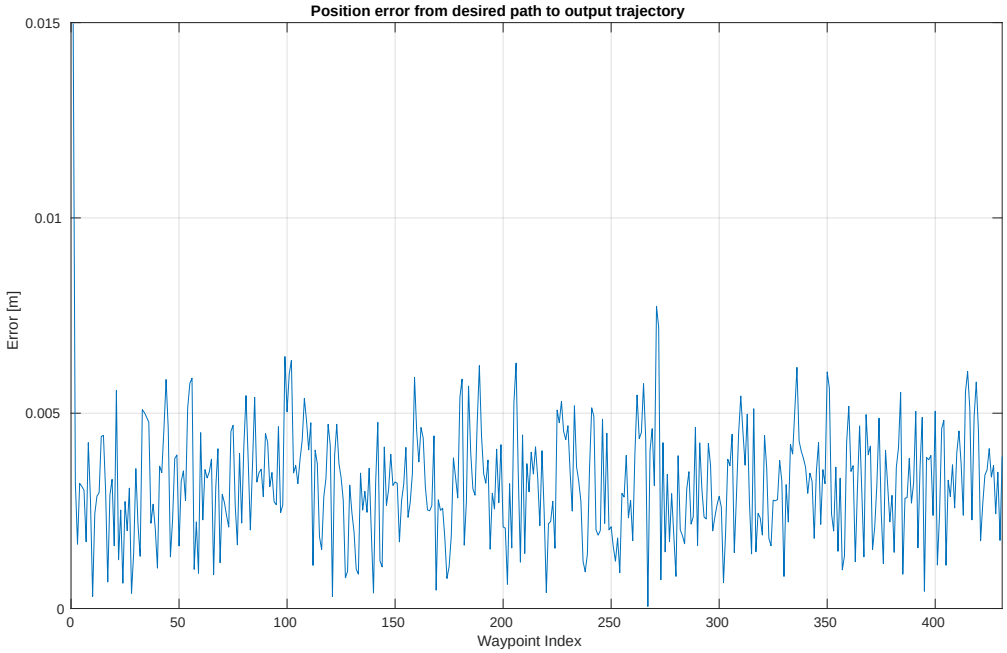


Figure 5.5: Positional error between the desired path of waypoints to output trajectory.

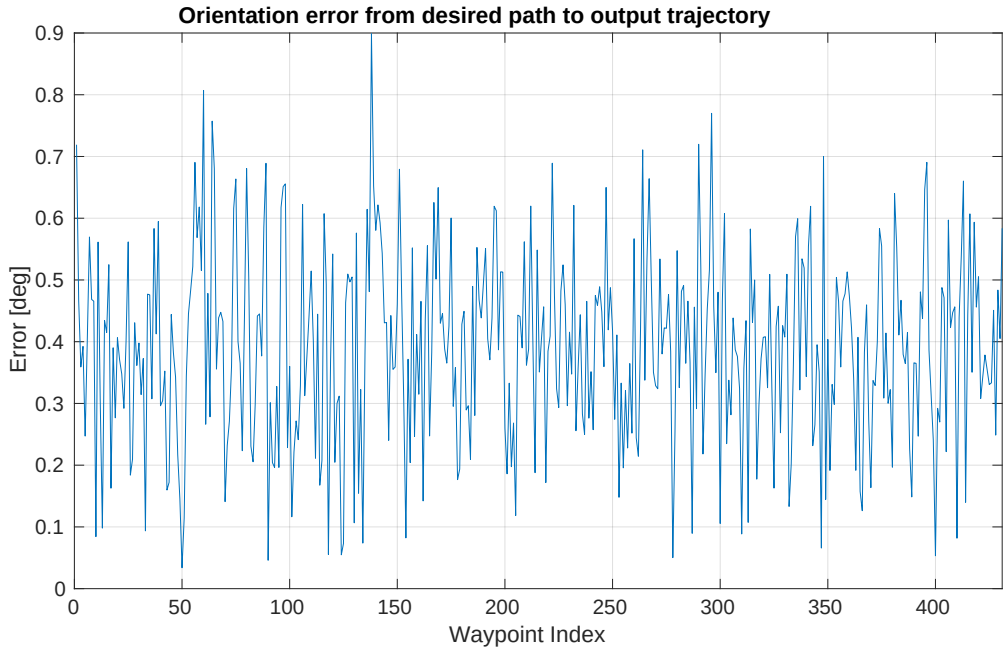


Figure 5.6: Orientation error between the desired path of waypoints and output trajectory.

Chapter 6

Discussion and Future Work

This chapter aims to present an analysis and discussion of the results obtained in the previous chapter. Additionally, it will incorporate insights regarding the author's personal experience with ROS and MoveIt in terms of user experience. Furthermore, this chapter will present areas for future research and development.

6.1 Discussion of Results

The results demonstrate that the planned trajectory generally adheres to the desired path with high accuracy. However, it was observed that the specified error tolerance of 0.01 meters and 0.01 radians was exceeded for the first waypoint. This discrepancy may be attributed to the post-processing stage, where the planned path is time-parametrized and the error tolerance is not considered. It is worth noting that the error tolerance was only exceeded for the first waypoint, which involves a significant change in direction. Overall, the attained accuracy is deemed more than satisfactory for the intended applications of the AMV 4200. In the context of the actual robot, where factors such as sensor precision, actuator limitations, and imperfect modeling come into play, the trajectory itself will not pose a significant constraint in terms of accuracy.

The final trajectory obeys the velocity limits that were set. However, the velocity trajectories exhibit significant noise since no limits were imposed on acceleration. The current joint controllers of the AMV 4200 utilize position and velocity as references for each time step. However, using the generated velocity trajectories as references may result in excessive wear on the actuators due to the rapid changes in velocity. Limiting acceleration would lead to smothering velocity trajectories, and would therefore be desirable. Fortunately, incorporating acceleration limits would require minimal changes in the software. It is worth noting that on April 22nd, 2023, it was announced that MoveIt now supports jerk limitation capabilities, enabling even smoother trajectories [55].

Although there was not enough time to fine-tune the configurations of MoveIt to best fit the AMV 4200, the results and the final software serve as proof of concept that MoveIt can be used to plan motions for the AMV 4200. The planning time of less than 17 seconds and other aspects of

the trajectory can be further enhanced by exploring alternative planning algorithms and inverse kinematics algorithms.

PickNik Robotics, the company responsible for the maintenance and development of MoveIt, offers consulting services to assist companies in utilizing MoveIt in their robotics projects [56]. They provide a small fixed-price consulting engagement that produces a comprehensive feasibility study report, typically ranging from 15 to 30 pages. This report includes a technical analysis discussing PickNik's proposed approach, an assessment of project risks including technology readiness level (TRL), software development milestones, and a roadmap for implementing the project. Additionally, the report includes architecture diagrams illustrating the high-level layout of the proposed solution. By engaging with PickNik, AMV can benefit from their expertise, and the assurance of having a trusted advisor throughout their robotics project.

6.2 Personal user experience with ROS and MoveIt

To provide context, it is helpful to share some background information about my experience before the project start. I am currently pursuing a master's degree in cybernetics and robotics, which has provided me with a solid foundation in robotics concepts. While I am familiar with programming in C++, I would consider myself more proficient in other programming languages. However, during this project, C++ was the primary language utilized. Additionally, my experience with simulation is limited, and although I have studied some theoretical aspects of motion planning, I had limited practical exposure to implementing numerical methods for motion planning, such as those utilized in MoveIt. Prior to this project, I had bearly heard about ROS and had no hands-on experience.

Starting out with ROS can be daunting due to its steep learning curve, requiring a thorough understanding of key concepts and hours of tutorials. ROS has a rich ecosystem of packages, libraries, and tools which can be utilized to accelerate development. However, navigating and understanding the available resources felt somewhat overwhelming in the beginning, and it took a while to identify and utilize the most relevant components. I found the video tutorials by community member Josh Newans [42] to be particularly helpful in understanding how different ROS components work together as a coherent system. Online forums were also a valuable resource, with experienced developers readily sharing their knowledge and offering assistance. In summary, the high expectations for the ROS community were met.

While the documentation and tutorials for ROS were comprehensive and helpful, the same level of guidance was not available for MoveIt. Specifically, there was a lack of information on implementing low degrees of freedom robots in the official MoveIt tutorial and documentation. As a result, a considerable amount of time was invested in configuring the AMV 4200 robot to work with MoveIt. It is worth noting that some of the available tutorials were still oriented toward ROS 1. However, with the impending end-of-life for ROS 1's last distribution in May 2025, it is expected that MoveIt 2 tutorials will receive more attention.

On multiple occasions, during my interactions with my supervisor at AMV, I have presented the progress of my work, only to receive the response, "That is exactly what I am currently developing

from scratch." This recurring scenario highlights the significant value that ROS, MoveIt, and other related software offer in the development of robots of AMV. Furthermore, it emphasizes the core principle of ROS: "Don't reinvent the wheel".

While there have been challenges in utilizing ROS and MoveIt, I believe that the task of building equivalent software from scratch would be even more demanding. Furthermore, through my experience working with these tools, I have come to realize the immense potential they hold in enabling proficient ROS users to swiftly develop software for diverse robotic systems. Based on the knowledge I have gained throughout this project, I am confident that I could easily construct software for a similar robot within a matter of a few weeks, as opposed to several months. This includes essential components such as kinematics, motion planning with collision avoidance, a physical simulator, and visualization, among others. The sheer scope of these tasks makes it impractical to achieve the same results by starting from scratch.

6.3 Further work

The following list summarizes the suggested steps for further work in the continued development of the project:

- Contact PickNik Robotics to discuss the services they can provide.
- Further specify limits on the joints and trajectory planning. Most notably, set an acceleration limit and perhaps also a jerk limit on the trajectory from the motion planner.
- Explore the use of different planning and inverse kinematic algorithms in MoveIt to find what is best suited for the AMV 4200, perhaps guided by PickNik Robotics.
- Connect MoveIt to Gazebo to enable testing of output trajectory in physics-based simulation.
- Further explore the use of LiDAR and cameras for motion planning and control. Test ideas and algorithms in a simulator by implementing these sensors as Gazebo plugins.

Chapter 7

Conclusion

This thesis aimed to recommend whether AMV should adopt ROS for developing robotics and automation software, specifically focusing on the AMV 4200 Shotcrete robot. Based on the research and testing of ROS, along with complementary software like MoveIt and Gazebo, it is evident that utilizing these tools would yield significant benefits in terms of development speed and software quality. This is particularly true when considering the development of software for multiple robots.

ROS is an open-source framework that serves as the foundation for building robotic systems. It provides a flexible platform with software libraries and tools for creating modular robotic software of high quality. MoveIt is a motion planning framework that integrates with ROS to enable manipulation capabilities, offering tools for tasks such as motion planning, collision checking, kinematics, and control. Gazebo, a simulation environment, also integrates with ROS to provide a physics-based platform for testing and validating the software before deploying it in the physical world. Together, ROS, MoveIt, and Gazebo form a cohesive toolkit that can empower the robotics engineers at AMV to design robotic software for the AMV 4200 Shotcrete and potentially their other machines as well.

The results presented show great promise in using MoveIt for motion planning with the AMV 4200. With Gazebo there is also the possibility of testing the motion plans in a simulator and implementing virtual LiDAR and cameras to provide a safe environment for developing and testing more advanced automation. The community of ROS has proven to be a valuable resource in terms of providing assistance and support, which contrasts with the support for Beckhoff PLC currently used by AMV.

The ROS framework has a steep learning curve due to its abundance of packages and resources. This thesis serves as an introductory guide, assisting AMV in exploring ROS by highlighting the relevant resources. As developers gain proficiency, the benefits of using ROS and its accompanying software become more pronounced. Hence, the experience gained from building ROS-based software for the AMV 4200 Shotcrete would significantly accelerate the development of future robotic software as well.

Bibliography

[1] Marcelina Krowinska. *Motion planning vs Path planning*. URL: https://www.shaderobotics.com/posts/motion-planning-vs-path-planning.

[2] Mark W Spong, Seth Hutchinson and M Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: John Wiley & Sons, 2006.

[3] Northwestern University. *Task space and workspace*. URL: https://modernrobotics.northwestern.edu/nu-gm-book-resource/2-5-task-space-and-workspace/#:~:text=The%20task%20space%20is%20a,space%20is%20the%20Euclidean%20plane..

[4] AMV. *AMV Home Page*. URL: https://www.amv-as.no/ (visited on 13/02/2023).

[5] AMV. *AMV 4200 Shotcrete Robot*. URL: https://www.amv-as.no/4200-shotcrete-robot (visited on 13/02/2023).

[6] American Concrete Institute. *What is shotcrete?* URL: https://www.concrete.org/tools/frequentlyaskedquestions.aspx?faqid=746.

[7] Deepak Tolani, Ambarish Goswami and Norman I. Badler. 'Real-Time Inverse Kinematics Techniques for Anthropomorphic Limbs'. In: *Graphical Models* 62.5 (2000), pp. 353–388. ISSN: 1524-0703. DOI: https://doi.org/10.1006/gmod.2000.0528. URL: https://www.sciencedirect.com/science/article/pii/S1524070300905289.

[8] Andreas K. Auen and Tomas S. Lyngroth. 'Modelling, Identification and Control of a 5-DOF Shotcrete Robot : Development of a Framework for Automatic Application of Shotcrete for AMV 4200H'. MA thesis. University of Agder, 2019.

[9] Open Robotics. *Getting Started*. URL: https://www.ros.org/blog/getting-started/.

[10] Open Robotics. *Noetic Ninjemys: The Last Official ROS 1 Release*. URL: https://www.openrobotics.org/blog/2020/5/23/noetic-ninjemys-the-last-official-ros-1-release.

[11] Open Robotics. *Understanding Nodes*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

[12] Open Robotics. *Understanding Topics*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html.

[13] Open Robotics. *Writing a Simple C++ Publisher and Subscriber*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html.

[14] Open Robotics. *Understanding Services*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html.

[15] Open Robotics. *Writing a Simple Service and Client*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Service-And-Client.html.

[16] Open Robotics. *Understanding Actions*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html.

[17] Open Robotics. *Actions*. URL: http://design.ros2.org/articles/actions.html.

[18] Open Robotics. *Understanding Parameters*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html.

[19] Open Robotics. *About parameters in ROS 2*. URL: https://docs.ros.org/en/humble/Concepts/About-ROS-2-Parameters.html.

[20] Open Robotics. *URDF*. URL: http://wiki.ros.org/urdf.

[21] Open Robotics. *Link*. URL: http://wiki.ros.org/urdf/XML/link.

[22] Adam Conkey. *Collision Meshes for simulating robots in Gazebo*. URL: https://adamconkey.medium.com/collision-meshes-for-simulating-robots-in-gazebo-ccc647d8b87d.

[23] Open Robotics. *Joint*. URL: http://wiki.ros.org/urdf/XML/joint.

[24] Open Robotics. *Using a URDF in Gazebo*. URL: https://classic.gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[25] Open Robotics. *XML Robot Description Format (URDF)*. URL: http://wiki.ros.org/urdf/XML/model.

[26] Rico Ruotong Jia. *I like to Moveit!* URL: https://ricoruotongjia.medium.com/s-d65d8ffcc73d.

[27] Josh Newans. *Making a Mobile Robot #8 - Adding a Lidar*. URL: https://articulatedrobotics.xyz/mobile-robot-8-lidar/.

[28] Josh Newans. *Making a Mobile Robot #12 - ros2_control Concept & Simulation*. URL: https://articulatedrobotics.xyz/mobile-robot-12-ros2-control/.

[29] Josh Newans. *Getting Ready for ROS Part 8: Simulating with Gazebo*. URL: https://articulatedrobotics.xyz/ready-for-ros-8-gazebo/.

[30] PickNik Robotics. *Concepts*. URL: https://moveit.picknik.ai/humble/doc/concepts/concepts.html.

[31] Rice University Kavraki Lab Department of Computer Science. *Available Planners*. URL: https://ompl.kavrakilab.org/planners.html.

[32] PikNik Robotics. *MoveIt Configurations*. URL: https://moveit.picknik.ai/humble/doc/examples/examples.html#configuration.

[33] PikNik Robotics. *IKFast Kinematics Solver*. URL: https://moveit.picknik.ai/humble/doc/examples/ikfast/ikfast_tutorial.html.

[34] PikNik Robotics. *MoveIt Setup Assistant*. URL: https://moveit.picknik.ai/humble/doc/examples/setup_assistant/setup_assistant_tutorial.html.

[35] Open Robotics. *Ubuntu Install Debian*. URL: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html.

[36] Open Robotics. *Getting Started*. URL: https://www.ros.org/blog/getting-started/.

[37] Open Robotics. *Distributions*. URL: https://docs.ros.org/en/humble/Releases.html.

[38] Open Robotics. *Windows Install Binary*. URL: https://docs.ros.org/en/humble/Installation/Windows-Install-Binary.html.

[39] Open Robotics. *Getting Started with Gazebo?* URL: https://gazebosim.org/docs.

[40] PickNik Robotics. *MoveIt 2 Binary Install On Windows*. URL: https://moveit.ros.org/install-moveit2/binary-windows/.

[41] Open Robotics. *Tutorials*. URL: https://docs.ros.org/en/humble/Tutorials.html.

[42] Josh Newans. *Articulated Robotics Home*. URL: https://articulatedrobotics.xyz/index.html.

[43] Open Robotics. *SolidWorks to URDF Exporter*. URL: http://wiki.ros.org/sw_urdf_exporter.

[44] Josh Newans. *Getting Ready for ROS Part 7: Describing a robot with URDF*. URL: https://articulatedrobotics.xyz/ready-for-ros-7-urdf/.

[45] Open Robotics. *sw_urdf_exporter*. URL: http://wiki.ros.org/sw%5C_urdf%5C_exporter.

[46] Shay Sackett. *SolidWorks to URDF Using the SW2URDF Plugin*. URL: https://www.youtube.com/watch?v=Id8zVHrQSlE.

[47] PikNik Robotics. *Getting Started*. URL: https://moveit.picknik.ai/humble/doc/tutorials/getting_started/getting_started.html.

[48] PikNik Robotics. *Kinematics*. URL: https://moveit.picknik.ai/humble/doc/concepts/kinematics.html.

[49] PikNik Robotics. *MoveIt Quickstart in RViz*. URL: https://moveit.picknik.ai/humble/doc/tutorials/quickstart_in_rviz/quickstart_in_rviz_tutorial.html.

[50] Gauthier Hentz. *Set Goal tolerance per coordinate in move_group_interface*. URL: https://github.com/ros-planning/moveit/pull/2780.

[51] Kavraki Lab. *Moveit OMPL Planning Interface*. URL: https://github.com/KavrakiLab/moveit_ompl_planning_interface.

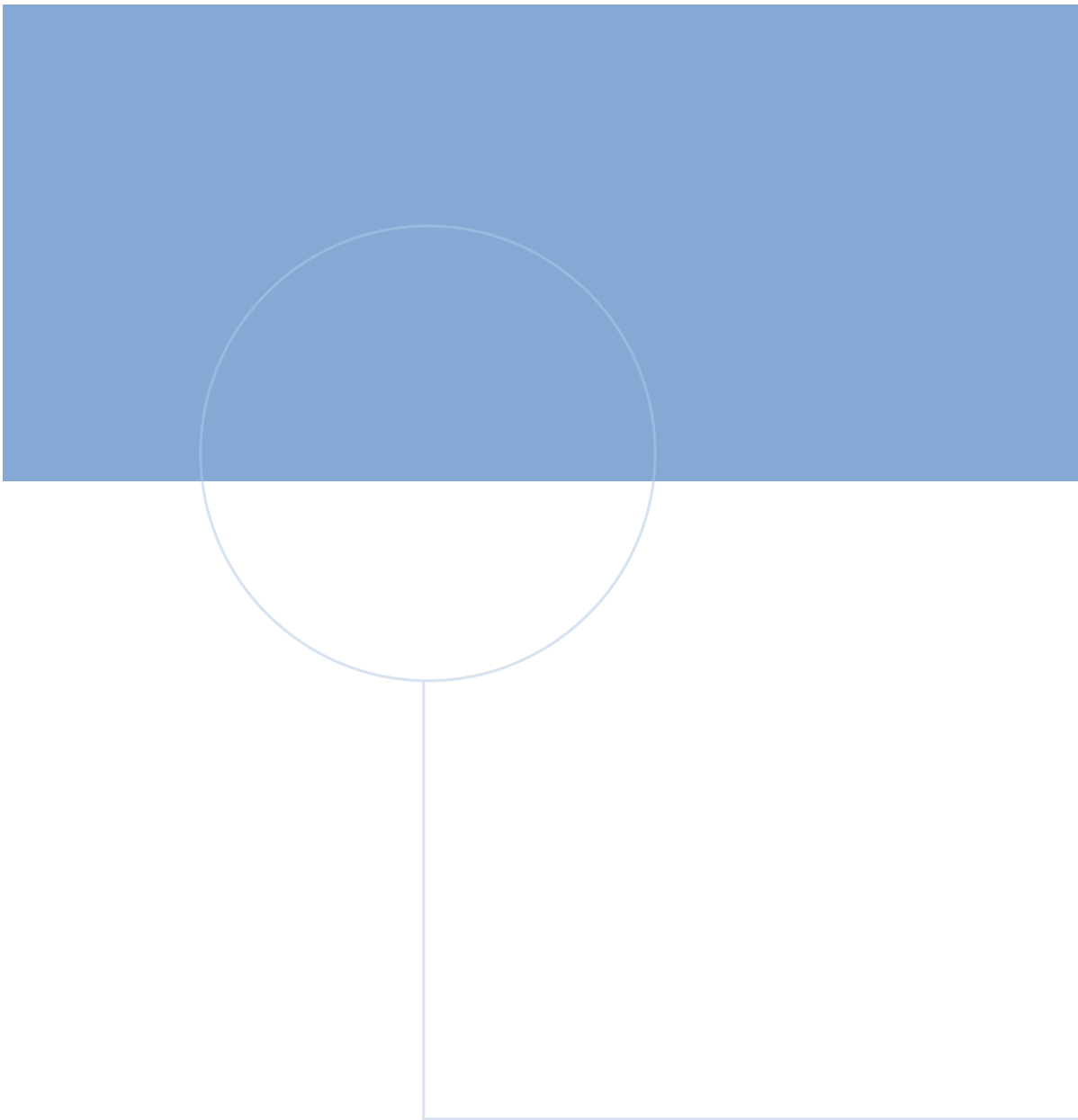
[52] shixinli. *Unable to sample any valid states for goal tree*. URL: https://github.com/ros-planning/moveit/issues/343.

[53] Ryan P. *MoveIt Motion Planning Failed: Unable to Sample Any Valid States for Goal Tree (Python Interface)*. URL: https://answers.ros.org/question/352387/moveit-motion-planning-failed-unable-to-sample-any-valid-states-for-goal-tree-python-interface/.

[54] Aaron MV. *MOVEIT + 4DOF*. URL: https://answers.ros.org/question/342490/moveit-4dof/.

[55] AndyZe. *Jerk-limited trajectory smoothing in MoveIt2!* URL: https://discourse.ros.org/t/jerk-limited-trajectory-smoothing-in-moveit2/25089.

[56] PickNik Robotics. *Services*. URL: https://picknik.ai/services/.



NTNU

Norwegian University of
Science and Technology