Johan Olaf Løe

Multi-Robot Control with MotoROS2

Enabling ROS2 for Welding with the GP25-12

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of Science and Technology

Johan Olaf Løe

Multi-Robot Control with MotoROS2

Enabling ROS2 for Welding with the GP25-12

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
June 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Multi-Robot Control with MotoROS2

Enabling ROS2 for Welding with the GP25-12

Johan Olaf Løe

2023-06-10

Preface

This thesis marks the journey's end at the Norwegian University of Science and Technology, NTNU. I want to express my sincere gratitude to Lars Tingelstad for his invaluable guidance as my supervisor, whose expertise and support have been instrumental in completing this research. Additionally, I am grateful to NTNU for allowing me to utilize their lab, Manulab, which has provided a crucial foundation for my experimental work.

Johan Olaf Løe

2023-06-10

Summary

This thesis addresses the challenges associated with intelligent robot programming, particularly in the context of the Manulab robot system at NTNU. The existing control method for the system was inadequate for efficient and intelligent programming, so the goal was to develop a user-friendly software package that simplifies the programming process.

To achieve this goal, the thesis outlines several sub-objectives, including identifying necessary constraints and variables, developing a simulation system representation, enabling external communication capabilities with MotoROS2, designing and implementing software for controlling the robot system and validating the proposed solution through physical tests.

Tests were conducted to evaluate the system's performance in various scenarios, including straight lines, curves, movements inside workpieces, and welding tasks. The outcomes of the tests were successful, with motion planners generating accurate trajectories that followed the desired paths and velocities. The result is a system that makes it easy to program the robot system and implement sensors into the robot program, relative to the existing solution.

Sammendrag

Denne oppgaven tar for seg utfordringene knyttet til intelligent robotprogrammering, spesielt i sammenheng med Manulab-robotsystemet ved NTNU. Den eksisterende kontrollmetoden for systemet var utilstrekkelig for effektiv og intelligent programmering, så målet var å utvikle en brukervennlig programvarepakke som forenkler programmeringsprosessen.

For å oppnå dette målet beskriver oppgaven flere delmål, inkludert identifisering av nødvendige begrensninger og variabler, utvikling av en simulert systemrepresentasjon, muliggjøring av ekstern kommunikasjonsevne ved bruk av MotoROS2, design og implementering av programvare for styring av robotsystemet, og validering av den foreslåtte løsningen gjennom fysiske tester.

Testene ble gjennomført for å evaluere systemets ytelse i ulike scenarier, inkludert rette linjer, kurver, bevegelser nær arbeidsstykker og sveiseoppgaver. Resultatene fra testene var vellykkede, med baneplanleggere som genererte nøyaktige baner som fulgte ønskede bane og hastigheter. Resultatet er et system som gjør det enkelt å syre robotsystemet, og implementere sensorer til robotprogrammet, relativt til den løsningen som var tilstede.

Contents

Preface i

Summary iii

Sammendrag v

1. Introduction 1
- 1.1. Background and Motivation . . . 1
- 1.2. Outline of the Thesis . . . 2

2. Fundamentals 3
- 2.1. Robot Controller . . . 3
- 2.2. Robot Manipulators . . . 4
- 2.3. Robotic System . . . 4
- 2.4. Sensors . . . 4
- 2.5. Middleware . . . 4
  - 2.5.1. Nodes . . . 5
  - 2.5.2. Services and Messages . . . 5
  - 2.5.3. Colcon . . . 6
- 2.6. URDF . . . 6
- 2.7. Robotics . . . 7
  - 2.7.1. Frames . . . 7
  - 2.7.2. Degrees of Freedom . . . 8
  - 2.7.3. Rotations . . . 8
  - 2.7.4. Quaternions . . . 10
  - 2.7.5. Transformations . . . 11
  - 2.7.6. Twist . . . 12
  - 2.7.7. Kinematics . . . 13
  - 2.7.8. Path and Trajectory . . . 14
  - 2.7.9. Online and Offline Programming . . . 15
- 2.8. Welding . . . 16
- 2.9. Robotic welding . . . 16
- 2.10. Existing Solutions and Previous Work . . . 17

2.11. Welding Parameters . . . 18
2.12. Welding Equipment . . . 19

3. Hardware and System Description **21**
3.1. Robot Controller . . . 21
3.2. Robot Manipulators . . . 22
3.3. Extra Modules . . . 22
3.4. Welding Equipment . . . 24
3.5. The Robot Cell . . . 27

4. Approach for Achieving the Stated Objectives **31**
4.1. Communication . . . 31
4.2. Simulation . . . 32
4.3. Calibration . . . 34
4.4. Motion Planning . . . 35
4.4.1. Velocity Control . . . 36

5. Implementation **39**
5.1. MotoROS 2 . . . 39
5.1.1. Installation . . . 39
5.2. MoveIt 2 . . . 42
5.2.1. Creating System Model . . . 42
5.2.2. Creating Moveit package . . . 45
5.3. Ros2 Main Package - planner_node . . . 47
5.3.1. Planning Functionality . . . 47
5.3.2. Implementing End Effector In-Position Publisher . . . 48
5.4. Ros 2 Interface Package . . . 49
5.5. Usage . . . 50

6. Experiments **53**
6.1. Case . . . 53
6.2. Definition of Object Origin . . . 54
6.3. Tests . . . 54

7. Results **59**
7.1. The Planning Package . . . 59
7.2. The Virtual Environment . . . 60
7.3. The In-Position Publisher . . . 61
7.4. The Velocity Limiter . . . 62
7.5. Test 1: The Linear Motion Test . . . 62
7.6. Test 2: The Circular Motion Test . . . 62
7.7. Test 3: The Inside Weld Test . . . 70
7.8. Test 4: Weld Test . . . 70

8. Discussion **77**
8.1. Virtual Environment 77
8.2. The Tests 77
8.3. The velocity Limiter 78
8.4. The Controlling Interface 80
8.5. The Planning Implementation 82
8.6. MotoROS2 85
8.7. Further Limitations 86

9. Conclusion and Further Works **89**
9.1. Conclusion 89
9.2. Further Works 89

A. Yaskawa Motoman GP25-12 Datasheet **95**

B. Yaskawa Motoman TSL600 Datasheet **97**

C. Yaskawa Motoman MT1 Datasheet **99**

D. Description of the Folders Included in the MotoROS2 Package **103**

E. Structure of the Robot Model Packages **105**

F. Moveit Controller Changes **107**

G. The Launchfile for Planning **109**

H. utilities.h **111**

I. utilities.cpp **113**

J. planner_node.cpp **123**

K. jacobian_generator.py **135**

L. waypoint_publisher.py **145**

M. Joint Velocities for Test Cases 1, 2, 3, and 4 **153**

List of Figures

2.1. Visualization of two frames, where frame {b} is located in the y-direction of frame {a}, while frame {a} is located in the z-direction of frame {b}. 8
2.2. A transformation between the two frames, {w} and {1}. 11
2.3. Representation of twist. A mobile robot rotates around the point r with an angular velocity w and linear velocity v. Figure from [18]. 12

3.1. The Motoman YRC1000 robot controller with the teach pendant (top right corner of the controller). 23
3.2. Netgear R2610 wireless router used for establishing network connectivity in the setup. 24
3.3. Yaskawa Motoman GP 25-12 industrial robot arm equipped with TIG torch as end effector. 25
3.4. Yaskawa Motoman GP 25-12 industrial robot arm mounted on Yaskawa Motoman TSL600 linear module, equipped with MIG gun as end effector. 26
3.5. Yaskawa Motoman MT1 with workpiece. 26
3.6. The Fronius MagicWave 3000 welding source. 27
3.7. The Fronius TPS400i welding source. 28
3.8. The image captures the complete robot cell, which includes two robot arms, a linear module (TSL600), a workpiece positioner module (MT1), and two welding apparatus. 28
3.9. The image showcases the world frame, the frame of the TSL600 base, and the frame of the mounting pendant within the system. . 29

4.1. System overview and functionality. 32

5.1. The figure displays that the IP address has been manually set to the value described. 40
5.2. Configuration of the agent IP for establishing a connection. 41
5.3. Configuration of joint names. 41
5.4. Alignment of the model and origin. 44

6.1. The coordinates defining the start point (left) and end point (right) of the weld are extracted directly from CAD software. 58
6.2. The offset of the end effector. 58

7.1. The schematics of the node waypointlistener. 60
7.2. The figure shows the virtual environment created in this thesis, visualized with Rviz2. 60
7.3. The figures illustrate the variation in data as the end effector approaches the waypoint set for welding, indicating the activation of the welding torch. 61
7.4. The trail of the end effector represents the trajectory for straight-line motion. 63
7.5. Linear velocity for straight-line motion. Untouched (top) and twist method (bottom). 64
7.6. Linear velocity for straight-line motion. Iterative time parameterization (top) and twist method (bottom) 65
7.7. Initial movement. 66
7.8. The trail of the end effector represents the trajectory. for circular motion 66
7.9. The trail of a low sampled arc. 67
7.10. The generated path lengths for a circle with resolutions $n = 15$ and $n = 1000$. 67
7.11. Linear velocities for circular path Untouched (top) vs twist (bottom) 68
7.12. Linear velocity for circular path, Iterative time parameterization (top) and twist (bottom). 69
7.13. The trail of the end effector representing the trajectory for test 3. . 70
7.14. Linear velocity for inside motion. Untouched (top) vs twist method (bottom). 71
7.15. Linear velocity for inside motion. Iterative time parameterization (top) vs twist method (bottom). 72
7.16. The trail of the end effector representing the trajectory for test 4. . 73
7.17. The end effector can be traced through the edge defined as the weld. 73
7.18. Linear velocity for test 4. Untouched (top) vs twist manipulation (bottom). 74
7.19. Linear velocity for test 4. Iterative time parameterization (top) vs twist method (bottom). 75

8.1. Desired path vs generated path 80
8.2. Representation of how the end effector may reach the desired waypoint inbetween publishing joint states, failing to update the kinematics in the node. 81

8.3. Visualization of the ready signal. Here, the signal is TRUE from waypoint X (left) untill updated at waypoint Y (right) which is FALSE. 82
8.4. The structure of the URDF. 83
8.5. The figure shows the resulting trajectory for a small Cartesian change when planning in joint space. 84
8.6. The figure shows the resulting trajectory for a small Cartesian change when planning in joint space with constraints. 85
8.7. If no Ethernet connection is detected after MotoROS2 is installed on the controller, a warning will show. 87

D.1. The structure of a JBI file. Collected from Yaskawa DX100 INSTRUCTIONS FOR RELATIVE JOB FUNCTION manual 104

M.1. Joint velocities for straight line motion. Untouched (top) and twist method (bottom) 154
M.2. Joint velocities for straight line motion. Iterative time parameterization (top) and twist method (bottom) 155
M.3. Joint velocities for circular path. Untouched (top) and twist (bottom). 156
M.4. Joint velocities for ciruclar path. Iterative time parameterization (top) and twist method (bottom). 157
M.5. Joint velocities for inside motion. Untouched (top) vs twist method (bottom). 158
M.6. Joint velocities for inside motion. Iterative time parameterization (top) vs twist (bottom). 159
M.7. Joint velocities for test 4. Untouched (top) vs twist method (bottom).160
M.8. Joint velocities for test 4. Iterative time parametrization (top) vs twist method (bottom). 161

List of Tables

6.1. Waypoints for test 1 55
6.2. Waypoints for test 2 56
6.3. Waypoints for test 3 56
6.4. Waypoints for test 4. 57

7.1. Distance and time for the tests. 62

Chapter 1.

Introduction

1.1. Background and Motivation

Robotics has witnessed significant growth and has emerged as a vital component of the modern industrial complex over the past decade [10], with its importance expected to increase further in the future. Robots offer precision, speed, long working hours, and the ability to operate in hazardous environments where human presence, even with safety equipment, is not feasible [36]. However, despite their numerous advantages, programming robots remains a complex and challenging task, requiring knowledgeable and experienced programmers to achieve the desired functionality. Moreover, the time required for robot programming is often impractical for rapidly changing jobs [15]. The advent of Industry 4.0 has introduced new industrial principles that emphasize small-batch manufacturing with frequent production changes [12]. Consequently, smart solutions are essential for efficient robotic programming in line with these evolving manufacturing requirements.

This thesis aims to address the challenges associated with intelligent robot programming, specifically focusing on the robot system available in Manulab at NTNU. At the time of this investigation, the existing control method for the system fell short of meeting the requirements for efficient and intelligent programming. A key sub-goal is to develop a user-friendly software package that puts the planning in an abstract layer, such that developers can focus on intelligent solutions such as edge detection or acquire points of interest with a camera. The proposed solution can benefit from its modularity by leveraging open-source software, allowing for seamless future modifications and enhancements. The thesis specifically focuses on establishing a comprehensive framework for seamless offline programming of the Motoman GP25-12 industrial general-purpose arms equipped with welding gear, and thus, welding jobs will be in focus for the development. To accomplish this overarching goal, the following sub-objectives have been identified:

- Identify the necessary constraints and variables for being able to solve the case.
- Enable external communication capabilities with the robot system for efficient programming and control.
- Develop a virtual representation of the system to facilitate planning, testing, and analysis.
- Design and implement software that can effectively control and manage the robot system.
- Validate the proposed solution by conducting tests on a physical system.

1.2. Outline of the Thesis

This thesis will first provide the preliminary knowledge needed to understand the methods and ideas discussed within it, as well as identify some of the challenges within the case. The following chapter will propose solutions for each problem and describe the hardware and software used to achieve the stated goals. Subsequently, the results will be presented and discussed, followed by a conclusion and suggestions for further development.

Chapter 2.

Fundamentals

This chapter aims to provide readers with the necessary knowledge to understand the concepts and methods utilized in this thesis. Before delving into the preliminaries, it is important to establish a clear definition of the term used. In this thesis, the term *robot* specifically refers to open-loop robots, commonly known as robot arms. It is important to note that the term robot can encompass a wide range of mechanical and electronic assemblies capable of manipulating their environment, such as steward robots and wheeled robots.

2.1. Robot Controller

The robot controller serves as the central computer that governs the operations of the mechanical arm. A robot arm is capable of carrying out various functions, such as acting as a force source, executing specific motions, interacting with the environment, and performing a combination of these tasks. To enable these functionalities, the robot controller is responsible for effectively controlling the individual joints of the robot. By coordinating the movements of these joints, the robot controller enables the arm to accomplish its intended tasks [18].

The teach pendant, also known as the robot pendant or programming pendant [44], is a handheld device that is connected to the robot controller. It serves as a user interface, allowing for monitoring, programming, and interaction with the robot controller. The teach pendant provides a convenient means for users to interface with the robot system and manage its operations effectively.

2.2. Robot Manipulators

A robot manipulator or robot arm is a physical product that can be configured to achieve different configurations. This allows the robot to manipulate physical objects, allowing for the automation of complex tasks. A robot manipulator consists of a series of links, which are connected by joints, that allow the arm to move and assume different configurations.

The joints are typically driven by electric motors or hydraulic actuators, which provide the necessary force and control to move the arm. The end of the arm may also have a tool or gripper attached to it, allowing it to pick up and move objects. This tool is often referred to as the *end effector*.

The arm's configuration can be controlled through various means, including manual programming or computer algorithms. Being programmable, the robot can perform virtually any task that can be programmed. As a result, the arm is capable of a wide range of operations, from simple holding tasks to intricate assembly and manufacturing processes like welding.

2.3. Robotic System

In this thesis, the term "robotic system" refers to the comprehensive integration of hardware and software components utilized to enable the robot to execute the desired task. It encompasses all the necessary elements and subsystems that contribute to the functionality and operation of the robot.

2.4. Sensors

A sensor is a component that is capable of detecting and measuring physical phenomena from the surrounding environment. It converts the sensed data into a format that can be used in the digital world. Sensors can range from simple devices like photo-resistors to more advanced technologies such as camera sensors and Light Detection and Ranging (LiDAR) sensors.

2.5. Middleware

Middleware is a vital communication architecture that enables different applications to interact with one another. In the realm of robotics, this component plays a crucial role, particularly when applications are written in different programming languages. Middleware facilitates seamless communication between these applications, bridging the language barrier and allowing them to exchange information effectively [6].

ROS 2 (Robot Operating System 2), is a robotics middleware that provides a collection of tools, libraries, and conventions for developing complex robotic systems. It is an evolution of the original ROS, with the aim of addressing some of the limitations and shortcomings of the original system.

ROS 2 supports multiple programming languages, including C++, Python, and others, making it easier for developers to use the language they are most comfortable with. Additionally, it provides a rich set of tools for debugging and visualizing the behavior of robotic systems [19].

Certain computers, such as microcontrollers, often have limited resources in terms of available RAM and computational power. As a result, running the ROS 2 framework on these devices becomes challenging. However, a solution called Micro-ROS addresses this limitation by enabling seamless communication between resource-constrained computers and the ROS 2 framework [1]. Micro-ROS offers a lightweight implementation of ROS 2 that can effectively operate on devices with limited resources, making it possible to leverage the core functionalities of ROS 2 in such environments.

2.5.1. Nodes

A ROS node is a process that processes data. Typically, a ROS node is responsible for performing specific tasks such as calculations, reading and sharing sensor data, or activating external components such as light-emitting diodes. These nodes are designed to have a narrow scope of responsibility, which offers the advantage of simplifying error detection and adding fault tolerance to the system since nodes can continue to operate even when one node fails. Nodes are essential components of the ROS framework and can communicate with each other using messages and services [29].

2.5.2. Services and Messages

ROS messages are a way for different nodes in a ROS system to communicate with each other by sending data. Messages define the structure and content of the data being sent and are defined using a simple message definition language. Messages can contain various types of data, including strings, numbers, and arrays. Once a message is defined, it can be published by one node and subscribed to by another node, allowing for asynchronous communication between different parts of the system.

ROS services, on the other hand, provide a way for nodes to communicate in a synchronous manner. Instead of sending data continuously like messages, services allow nodes to request specific tasks to be performed by other nodes. Services are defined using a similar language to messages and consist of a request message and a response message. When a node makes a request to a service, it waits for a response before proceeding with its task.

ROS services and messages are published and subscribed to over topics. Topics are designed to be asynchronous and loosely coupled. Nodes can publish messages to a topic without knowing if there are any subscribers, and subscribers can receive messages without knowing who published them. This makes it easy to add or remove nodes from the system without affecting the overall communication flow [30].

In ROS 2, both ROS messages and ROS services are critical for building complex robotic systems that require coordination between different nodes. They provide a standardized way of exchanging data and commands, enabling the seamless implementation of custom messages and services [28].

2.5.3. Colcon

Colcon is a build tool specifically designed for ROS 2, aiming to facilitate the management and construction of large-scale robotic systems. It streamlines the process of building and packaging ROS 2 packages, providing developers with a convenient means of organizing their projects.

One of the key advantages of using Colcon is its support for independent package building. Each package within the workspace is treated as an autonomous unit, allowing developers to rebuild only the specific package that requires modifications. This approach significantly reduces build times, leading to improved efficiency during development.

After building a workspace using Colcon, it is essential to source the workspace in order to access the packages from the command line. Sourcing the workspace ensures that the terminal recognizes the packages and their associated dependencies, enabling seamless interaction and execution of commands related to the packages within the workspace [39].

2.6. URDF

The Unified Robotics Description Format (URDF) is a file format used to describe robot scenes. URDF allows for the specification of various components such as links, joints, sensors, and actuators. This comprehensive representation enables a detailed description of the robot's physical structure and interactions with the environment.

URDF incorporates kinematic and dynamic properties essential for accurate simulation and control of the robot. It allows the specification of joint limits, joint types (such as revolute or prismatic), and inertial properties (such as mass, center of mass, and moments of inertia). These properties determine the robot's motion, stability, and response to external forces.

To tackle the challenges associated with managing large and complex URDF files, a macro language for XML called xacro is employed. Xacro enables a more transparent and organized approach to creating URDFs. By utilizing xacro, each robot can be described in its own file, and a more comprehensive system consisting of multiple robots and objects can be created by importing each component into a joint xacro file, which can be used to generate the URDF [8].

2.7. Robotics

2.7.1. Frames

A frame is a coordinate system. Frames are assigned to reference points and parts in a robot. they are useful for describing the orientation between points of interest relative to a given frame. As different objects may change orientation, frames can be used to express how the object is oriented as well as how other objects are positioned when observed from a specific frame. A visualization of frames can be seen in Figure 2.1.

In robotics, the body-frame, often denoted as {b}, is a local coordinate system attached to the end effector of the robot. This frame is used to describe the orientation of the robot's end effector relative to the robot's base frame or another fixed reference frame, such as the space-frame {s}. The body-frame moves with the end effector, and its position and orientation can change as the robot moves and performs tasks.

The space-frame {s}, also known as the world frame or global frame, is a fixed reference frame attached to a stationary specific point in the robot's environment. It serves as a common reference for all frames within the robot system and is crucial to determine its position and orientation in the environment and navigating to different locations [18].

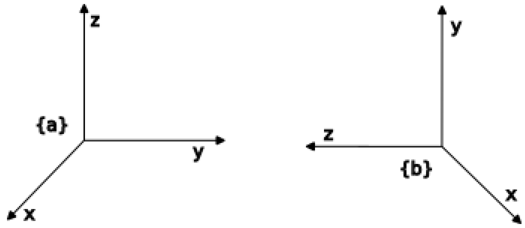


Figure 2.1.: Visualization of two frames, where frame {b} is located in the y-direction of frame {a}, while frame {a} is located in the z-direction of frame {b}.

2.7.2. Degrees of Freedom

In robotics, degrees of freedom refers to the number of independent configurations a frame can have. A frame is said to have more degrees of freedom if it can be configured in more independent ways. For instance, if a frame can have any orientation and exist within the space defined by the three dimensions x, y, and z, it has six degrees of freedom. In Euclidean space, six degrees of freedom are the maximum possible, as the frame cannot be configured independently in any more ways. Degrees of freedom plays a critical role in robotic systems as they determine the robotic system's range of motion and flexibility.

2.7.3. Rotations

Rotations are fundamental in the field of robotics and mechanics as they are used to describe the orientation of one coordinate frame relative to another. A rotation can be described by a 3x3 rotation matrix or a quaternion.

When considering two frames {a} and {b}, the rotation matrix that describes the rotation from frame {a} to frame {b} is denoted as R_{ab}. Similarly, the rotation matrix that describes the rotation from frame {b} to frame {a} is denoted as R_{ba}.

Rotation matrices possess a significant property that their transpose is equal to their inverse, i.e., $R^T = R^{-1}$. This property ensures that the transpose of a rotation matrix is equivalent to rotating in the opposite direction, meaning $R_{ab}^T = R_{ba}$. Furthermore, rotations can be used to express a point described in frame {b}, in frame {a}. If a point is represented in frame {b}, the corresponding coordinates in frame {a} can be obtained utilizing the following:

$$t_a = R_{ab} t_b. \tag{2.1}$$

The properties described above are beneficial in many robotics applications, where frames of reference constantly change, and switching between frames is necessary.

Therefore, understanding the properties of rotation matrices and their relationships between frames is essential in the field of robotics and mechanics. This knowledge is particularly critical for applications involving object manipulation, where the accurate interpretation of orientation significantly impacts task performance and success.

A rotation matrix R can be mathematically represented as follows:

$$R = \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix} \in SO(3) \tag{2.2}$$

Here, the elements $x_x, y_x, z_x, x_y, y_y, z_y, x_z, y_z, z_z$ define the orientation of the frame. The notation $SO(3)$ denotes the special orthogonal group of dimension 3, which represents the set of all rotation matrices in three-dimensional space.

The rotation matrix R can be represented using exponentials. If a frame is rotated around the unit axis $\hat{\omega}$ by an angle θ, the corresponding rotation matrix is given by:

$$R = e^{[\hat{\omega}]\theta} = I + \sin(\theta)[\hat{\omega}] + (1 - \cos(\theta))[\hat{\omega}]^2 \tag{2.3}$$

Here, $[\omega]$ represents the skew-symmetric matrix associated with the 3-dimensional rotation axis ω:

$$[\omega] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{2.4}$$

In Equation 2.3, I is the identity matrix. The matrix exponential $e^{[\omega]\theta}$ provides a compact representation of the rotation matrix R using the axis-angle parameters ω and θ.

Rotation matrices can be multiplied together to form a new rotation matrix. By defining three distinct rotations around the orthogonal axes x, y, and z, the resulting composite rotation can be represented as $R = R_{x,\theta_1} R_{y,\theta_2} R_{z,\theta_3}$, where θ_1, θ_2, and θ_3 denote the rotation angles around each respective axis. This composition of rotations enables the description of any desired orientation by combining the individual rotations around the axes [18].

2.7.4. Quaternions

A quaternion is a four-dimensional vector consisting of a real component, denoted as w, and three complex components, denoted as i, j, and k. In the field of robotics, quaternions are commonly used to describe rotations. The real component represents the magnitude of rotation, while i, j, and k are orthogonal vectors that define a three-dimensional axis around which the rotation occurs.

Mathematically, a quaternion can be expressed as:

$$q = w + ai + bj + ck, \tag{2.5}$$

where a, b, and c are coefficients and i, j, and k represent imaginary units. If the norm of the quaternion is equal to 1, meaning that the magnitude is 1, the quaternion can be formatted as:

$$q = \sin(\theta/2) + \cos(\theta/2) \cdot \omega, \tag{2.6}$$

where θ and ω is the angle and unit axis of rotation, similar to Equation 2.3.

Hamilton product, also known as quaternion multiplications, is a convention to multiply two quaternions together, forming a new quaternion. If the two quaternions are unit quaternions, then the product is also a unit quaternion. Hamilton product is defined as follows:

$$\begin{aligned} q_1 \circ q_2 =& (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ &+ (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i \\ &+ (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2) j \\ &+ (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) k \end{aligned} \tag{2.7}$$

To rotate a vector or a point with quaternions, the vector needs to be rewritten as a quaternion, which can be done by writing the point r $= x, y, z$ as $\bar{r} = 0 + xi + yj + zk$, where 0 corresponds to the w. The point can then be multiplied with another quaternion:

$$\hat{r} = q \circ \bar{r} \circ q*, \tag{2.8}$$

where $q*$ is the conjugated of the quaternion defined as $q* = w - xi - yj - zk$, and $\hat{r}$ is the rotated point [5].

2.7.5. Transformations

Transformations are a fundamental mathematical concept that describes the relative position and orientation of coordinate frames in a given space. In robotics, transformations are used extensively to specify the position and orientation of a robot's end-effector, as well as the location and orientation of objects in its environment. Mathematically, a transformation can be represented as a 4x4 matrix, commonly referred to as a transformation matrix:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \in SE(3). \tag{2.9}$$

Here, $0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$, and t is the translation between the two frames $t = \begin{bmatrix} x & y & z \end{bmatrix}^T$. The notation $SE(3)$ denotes the special Euclidean group consisting of all homogeneous transformation matrices in $\mathbb{R}^3$ [18].

A transformation between two frames; frame {a} and frame {b} is noted as T_{ab}. Because of the unique properties of transformation matrices, the transformation from frame {b} to frame {a} can be found by taking the inverse of the transformation T_{ab}: $T_{ba} = T_{ab}^{-1}$. The transformation between two frames is visualized in Figure 2.2.

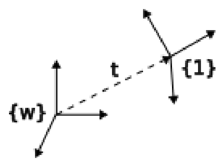


Figure 2.2.: A transformation between the two frames, {w} and {1}.

A pose refers to the transformation between the global frame and another frame of interest within a robotic system. It can be represented using different mathematical representations, such as a transformation matrix or a combination of a translation vector and a quaternion. The transformation matrix provides a comprehensive representation of the pose, including both translation and rotation information. On the other hand, using a translation vector and a quaternion allows for a more compact representation while still capturing the necessary spatial information. Both representations are widely used in robotics to describe the position and orientation of objects or coordinate frames in a precise and consistent manner.

2.7.6. Twist

In robotics, a twist is a mathematical concept that describes the combined linear and angular motion of a rigid body or a frame in space. A twist is typically represented as a six-dimensional vector, denoted by ν, which includes both a three-dimensional angular velocity vector ω and a three-dimensional linear velocity vector v.

$$\nu = \begin{pmatrix} \omega \\ v \end{pmatrix} = \begin{pmatrix} \hat{\omega}\dot{\theta} \\ -\hat{\omega}\dot{\theta} \times q + h\hat{\omega}\dot{\theta} \end{pmatrix}. \tag{2.10}$$

Here, $\hat{\omega}$ is the rotation axis which the frame is rotated around, $\dot{\theta}$ is the magnitude of the change in rotation, q is the vector describing the distance from $\hat{\omega}$ to the frame evaluated and h is the speed at which the frame is moving along $\hat{\omega}$. The angular velocity vector ω describes the rate of change in the orientation of the frame, while the linear velocity vector v describes the rate of change in the position of the frame.

There are two types of twists that are commonly used in robotics: the spatial twist and the body twist. The spatial twist ν_s describes the motion of a frame relative to the fixed reference frame {s}. The body twist, denoted as ν_b, represents the local motion experienced by a frame. A visual representation of the twist can be seen in Figure 2.3.

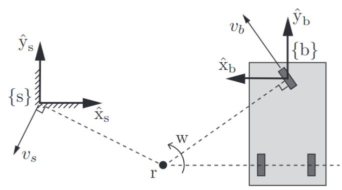


Figure 2.3.: Representation of twist. A mobile robot rotates around the point r with an angular velocity w and linear velocity v. Figure from [18].

In robotics, a twist can be interpreted as rotation about an axis known as the "screw axis" S with the angular velocity $\dot{\theta}$:

$$\nu = S\dot{\theta} \tag{2.11}$$

The screw axis provides crucial information about how a rigid body moves when it undergoes a motion consisting of translation and rotation. Because the physical properties of robot arms are a series of rigid links and joints, so all motions can be represented as a screw. This proves helpful, as it can be shown that any configuration of a robot arm can be represented with constant S.

The screw axis is defined as the normalized twist, denoted by S. It captures the direction and magnitude of the motion and is defined as:

$$S = \begin{matrix} \nu/\|\omega\| \text{ if } \omega \neq 0 \\ \nu/\|v\| \text{ if } \omega = 0. \end{matrix} \tag{2.12}$$

The screw axis plays a vital role in kinematics and dynamics analysis, motion planning, and control of robotic systems [18].

2.7.7. Kinematics

Forward Kinematics

Forward kinematics describes the transformation between a reference frame and the frame of a given link, usually the end effector. As this transformation changes as the joint values θ i.e the motors in the robot turn, forward kinematics are essential for knowing where in space the robot is:

$$T_{sb} = T(\theta). \tag{2.13}$$

Product-of-exponentials is a convention used to describe the forward kinematics of a robot. The convention utilizes the screws S_i of each joint, as well as the corresponding joint angles θ_i. This convention has the advantage of not requiring the definition of a frame for each joint, except for specifying the rotation axis ω_i and the position q_i of each joint relative to the base frame of the robot:

$$T(\theta) = \Pi_{i=1}^{n} e^{[s_i]\theta_i} M. \tag{2.14}$$

Here, n is the number of joints in the system and $M = T(0)$ represents the transformation from the coordinate frame s to the coordinate frame b when all the joints are at their zero configuration [18].

Inverse Kinematics

Inverse kinematics plays a crucial role in effectively controlling robots. It is a mathematical process that aims to calculate the joint positions required to achieve a specific goal or pose. In contrast to forward kinematics, which determines the resulting pose based on input positions, inverse kinematics focuses on determining the joint positions necessary to reach a desired pose.

Velocity Kinematics

Velocity kinematics, also known as differential kinematics, provides information about the speed at which the robot is moving, given a specific joint configuration and velocity.

The Jacobian is a crucial tool for understanding the velocity kinematics of a robot. The Jacobian is a matrix that relates the velocities or rate of change of a set of variables to another set of variables. More specifically, the Jacobian matrix represents the gradient or derivative of a vector-valued function concerning its input variables. In the context of robotics, the Jacobian matrix describes the relationship between a robot's joint velocities and its end-effector velocities or the velocities of any other relevant frames.

For a robot arm, the Jacobian is obtained with the following equation:

$$J(\theta) = \begin{bmatrix} J_1 & J_2 & \dots & J_n \end{bmatrix}, \tag{2.15}$$

and

$$J_i = Ad_{(\Pi_{j=1}^{i-1} e^{[S_j]\theta_j})} S_i. \tag{2.16}$$

The Ad(x) represent the adjoint of the resulting 4×4 matrix:

$$Ad(T) = \begin{bmatrix} R & 0 \\ [t]R & R \end{bmatrix} \in \mathbb{R}^{6\times 6}.$$

The velocity of the end effector is defined as the twist ν_{ee} (Equation 2.10), which can be obtained by multiplying the Jacobian matrix $J(\theta)$ with the vector of joint velocities $\dot{\theta}$ [18]:

$$\nu = J(\theta)\dot{\theta}. \tag{2.17}$$

2.7.8. Path and Trajectory

In robotics, a path refers to a time-independent and purely geometric sequence of positions and orientations that a robot is required to follow in order to accomplish a specific task or achieve a desired goal. To define the starting and ending points of a path, it is commonly parameterized using the parameter s. The parameter s is defined within the range of [0, 1], where 0 represents the beginning of the path and 1 represents the end. This parameterization enables to establish a clear reference for the progression along the path. In robotics, a path is denoted as $\theta(s) = \theta_1(s), \theta_2(s), \ldots$, where $\theta_i(s)$ represents the value of the i-th joint at the position s.

A trajectory is a path that is time-dependent, where each point in the path is reached at a specific time. In contrast to a path, a trajectory is time-parameterized, and the scalar path parameter s becomes a function of time denoted as $s(t)$. Therefore, a trajectory is denoted as $\theta(s(t))$ in robotics to represent its dependence on time.

By manipulating the time parameterization function $s(t)$, the velocity profile of the robot can be adjusted to achieve specific behaviors, such as minimizing the robot's acceleration, known as jerk, maximizing the smoothness of the trajectory, and minimizing the total time required to complete the trajectory, known as time optimization [18].

2.7.9. Online and Offline Programming

A robot is a mechanical and electronic product that requires programming to achieve its desired behavior. Programming a robot involves configuring the values for its actuators, which control its movement and actions. In the field of robotics, there are two main methods for programming a robot: offline programming and online programming.

Offline programming is a programming method that does not require direct interaction with the physical robot. Instead, it involves developing and testing programs using a digital representation or simulation of the robot system. This approach offers several advantages, including the ability to rapidly iterate and develop complex programs, leverage advanced sensors, and employ sophisticated algorithms. By relying on digital simulations, collision risks with the physical robot can be easily avoided and mitigated. However, it is important to note that offline programming heavily relies on accurate algorithms and calculations to ensure the program's effectiveness and safety. By utilizing software tools and digital representations of the system, potential collisions and other safety concerns can be proactively addressed, resulting in efficient and reliable program development.

On the other hand, online programming involves using a physical robot to achieve the desired behavior. This is achieved by manually controlling the robot through actions such as jogging or physically guiding its movements. Online programming allows the programmer to directly record and teach the robot specific actions and sequences of movements, enabling precise execution of tasks. However, one limitation of this approach is that it often requires taking the robot out of production or temporarily halting its normal operations for programming purposes. This interruption can disrupt workflow and reduce productivity during the programming phase. Additionally, utilizing sensors effectively may be challenging. Nevertheless, online programming offers the advantage of immediate feedback and real-time interaction with the robot, allowing for explicit and, most importantly, simple control [11].

2.8. Welding

Welding is a widely used process in manufacturing and construction industries, where two or more metal parts are fused. Welding creates strong, durable, and permanent joints between metal parts and is a critical process in producing many structures, such as buildings, bridges, pipelines, and vehicles.

Gas Tungsten Arc Welding (GTAW), also known as Tungsten Inert Gas (TIG) welding, and Gas Metal Arc Welding (GMAW), also known as Metal Inert Gas (MIG) welding, are two standard welding techniques. TIG welding uses a tungsten electrode and a shielding gas to create a precise, high-quality weld, making it ideal for welding thin materials like aluminum. MIG welding, on the other hand, uses a wire electrode that is continuously fed through a welding gun and a shielding gas to protect the weld from contamination [40].

The choice of welding technique depends on the specific application and the properties of the metals being welded. Each technique has its own advantages and disadvantages, and selecting the correct technique for the job requires careful consideration of factors such as material thickness, joint configuration, and desired weld quality.

2.9. Robotic welding

Robotic welding is an advanced and automated technique that utilizes robot manipulators to perform welding tasks, eliminating the need for manual labor. This technology has gained significant traction, particularly in industries that require a high volume of welds, such as the automotive industry. By employing robots, the quality of welds can be improved, along with enhancements in flexibility, cost-effectiveness, and overall manufacturing productivity [14].

The value and significance of robotic welding are further emphasized by the shortage of skilled welders in the manufacturing industry [21, 41, 2]. The declining number of skilled welders has created a scarcity, which necessitates exploring alternative solutions to ensure uninterrupted welding operations and maintain productivity.

Moreover, certain metals, such as aluminum, pose specific challenges for human welders. Aluminum has higher thermal conductivity compared to other metals, resulting in a narrower window for achieving proper fusion between parts. Additionally, when exposed to air, aluminum is prone to oxidation, leading to the formation of oxide layers on the metal surface. These oxide layers can interfere with the welding process and contribute to the formation of impurities in the weld, resulting in weaker and inferior welds [13].

To address these challenges and ensure high-quality welds, the use of robotic welding systems offers distinct advantages. Robots can execute precise and consistent welds, reducing the risk of human error and mitigating the difficulties associated with welding aluminum or other challenging materials.

While robotic welding offers numerous advantages, there are also several issues that need to be considered when substituting humans with robotic applications [14]. These include the substantial time and effort required to develop robotic applications for low-batch size jobs, such as repairing. Additionally, in the absence of sensors and advanced control systems, the robot system is unable to make corrective adjustments when unforeseen changes occur. Moreover, the initial cost of implementing robotic welding solutions can be high, potentially surpassing the return on investment.

One of the most crucial aspects of robotic welding is the ability of robots to make corrective decisions, which heavily relies on advanced control systems and sensors. However, implementing advanced sensors and programs with industrial arms presents a challenge due to the low-level nature of the robot controller. While robots typically come with simplified programming options like a teach pendant and some online programming capabilities, integrating sensors necessitates the acquisition and programming of an external PLC (programmable logic controller), introducing additional complexities. Moreover, developing programs in this low-level language is demanding, and robot controllers often have limited resources in terms of memory and computational power.

2.10. Existing Solutions and Previous Work

Because of the apparent need and the many benefits of utilizing robotic welding, ongoing research efforts are focused on advancing robotic welding technologies, refining control algorithms, exploring innovative welding techniques, and integrating advanced sensing systems. There are multiple solutions have been developed.

ROSWELD, developed by PPM ROBOTICS AS, is a comprehensive planning, monitoring, and control software suite specifically designed for heavy industrial robot applications [25]. While frameworks like ROSWELD offer valuable functionalities, there are certain considerations to be aware of. As ROSWELD is built on the ROS framework, more specifically the ROS 1 version, it is not automatically compatible with newer versions. For instance, if an application is created with the newer ROS 2 framework, it can lead to compatibility issues.

Additionally, as an all-in-one solution, the framework aims to encompass various tasks such as changing welding parameters, visualizing welding processes, and planning welding paths. However, this comprehensive approach can result in a larger program size, potentially compromising transparency and making it less suitable for prototyping and implementing advanced algorithms for new solutions.

Path Robotics is a company that specializes in providing solutions for robotic welding. Their solution includes the concept of “truly autonomous welding” and the ability to have robots scan, position, and weld parts independently without the need for skilled welders or robot programmers [31].

The functionality of their robots is based on a process where the model is scanned and recreated in 3D. Through the utilization of artificial intelligence (AI), the system comprehends the characteristics of the part, enabling the generation of optimal robotic paths and precise part positioning, ultimately leading to the production of high-quality welds. This results in a robot system that eliminates the need for traditional programming.

However, it is essential to consider that one limitation of this approach is its incompatibility with existing robot systems. Implementing Path Robotics’ solution necessitates using a completely new robot cell, making it challenging to retrofit onto other robot systems. As a result, manufacturers interested in adopting this technology would need to replace their current robot cells, which can be a significant expense. Additionally, the new cell would likely be limited to performing a specific task, such as welding, reducing its versatility for other applications.

2.11. Welding Parameters

Welding is a complex process that requires careful consideration of multiple variables to achieve optimal results. The quality of the weld depends on various factors, including but not limited to feeding speed, movement speed, angle of the welding gun, the direction of the weld, the type and thickness of the material being welded, and the specific welding tool used. Developing a comprehensive framework that accounts for all these parameters is a formidable task.

However, it is important to note that many of these parameters are typically determined by the welding source and do not necessarily need to be explicitly considered by the mechanism controlling the robot or welding tool. Assuming that only the parameters directly related to the movement of the end effector are taken into consideration, the following factors remain:

- Direction: The path or trajectory along which the welding torch moves.

- Orientation: The alignment or position of the welding torch with respect to the workpiece during welding.
- Travel speed: The velocity at which the welding torch moves during the welding process. Depending on the technique, the torch should move steadily in the range from 1 to 10 cm/s [20].

2.12. Welding Equipment

The term "welding equipment" encompasses all the components required for being able to weld. These components include the welding gun, the welding source, and the wire feeder.

The welding gun or welding torch is an essential component of welding equipment, serving a crucial role in the welding process. It fulfills multiple functions, including generating the necessary heat or arc to melt the metal pieces together, guiding and directing the filler material into the weld, and directing shielding gas to protect the weld zone. The design of the welding gun can vary depending on the specific welding technique employed, taking into account factors such as the welding method, materials being welded, and desired welding parameters [42].

The wire feeder is a crucial component responsible for supplying filler material to the welding gun. There are two types of wire feeders commonly used: cold feeders and electrode wire feeders.

Cold feeders are designed to provide filler material to the weld without the material being part of the electrical circuit. In this case, the material needs to be heated separately before joining the metal pieces. Cold feeders are commonly used in TIG welding processes.

On the other hand, electrode wire feeders are specifically designed for the wire that is part of the electrical circuit during welding. The wire acts as both the filler material and the electrode, carrying the current necessary for the welding process [43].

The welding source or power source is the component that provides the necessary voltage and amperage to facilitate welding. These sources may include computers that allow for job creation, parameter setting, and external communication through an interface.

Chapter 3.

Hardware and System Description

This chapter describes the hardware that is used and exists within the robot cell.

3.1. Robot Controller

The Yaskawa Motoman YRC1000 is a robot controller Yaskawa which is compatible with multiple arms within the Motoman family. The YRC1000 can control up to 8 robots and external axes for control of a total of 72 joints [44]. The YRC1000 controller can be seen in Figure 3.1.

The robot controller plays a critical role in the operation of industrial robots. Functioning as a computer with its own operating system, it provides a platform for running various applications that enable robots to carry out their designated tasks. Developing third-party applications for robot controllers can be challenging, especially when the operating system is closed source. However, the YRC1000 controller offered by Yaskawa Motoman provides a distinct advantage with the MOTOPLUS SDK (software development kit) [23], a toolkit enabling developers to create custom applications which work seamlesly with the robot controller. This capability opens up significant potential by allowing the utilization of libraries such as ROS2, thereby enhancing the capabilities of industrial robots.

One notable feature of the YRC1000 controller is its extensive range of communication options, enabling seamless interaction with the controller. These options include Ethernet and RS232 connections, utilizing protocols such as TCP/IP [44]. These conventional communication methods provide convenient connectivity with external computers. In this thesis, to establish communication with the robot controller, a generic wireless router (as depicted in Figure 3.2) was employed, connected to the controller via an Ethernet cable.

3.2. Robot Manipulators

The Yaskawa Motoman GP25-12 is a type of robot manipulator designed for industrial automation applications. It features six revolute axes, which provide a high degree of flexibility and control over the arm's motion. The joints are named by the manufacturer as S, L, U, R, B, and T, representing swivel, lower, upper, rotation, bend, and twist respectively.

The six axes of the GP25-12 give the end effector a 6 degrees of freedom, meaning that the end effector can move and orient itself in six different directions. This allows the arm to reach any position within its work space and manipulate objects with great precision and accuracy.

In a robot system, each joint of the robot arm may have a "joint limit" that defines the maximum and minimum values that the joint can move within. Joint limits are typically given in radians or degrees, except in the case of linear joints where they are specified in meters. They are predetermined based on the design of the hardware components, such as servo motors used in the robot, or by physical constrains. The specifications of the GP25-12 can be found in Appendix A. However, it is important to note that in some cases, software-defined limits may be implemented to further restrict the range of movement. This can be particularly relevant when the robot is equipped with specialized equipment that cannot tolerate excessive twisting or bending. Therefore, while the physical limits of the robot must be respected, the software-defined limits may impose additional restrictions to ensure safe and precise operation.

The system has two GP25-12s. One arm is mounted on a pendant elevating the robot from the floor (Figure 3.3), and one arm is mounted on a the linear module TSL600 (Figure 3.4).

3.3. Extra Modules

In addition to the robot controller and the industrial arms, there are other modules within the workspace that enhance the system's capabilities. These modules provide extended reach and additional degrees of freedom, enabling the execution of more complex tasks.

The Motoman TSL600 (shown in Figure 3.4) is a linear motion base that allows movement along a single axis. By mounting a robot manipulator on the TSL600,

Figure 3.1.: The Motoman YRC1000 robot controller with the teach pendant (top right corner of the controller).

Figure 3.2.: Netgear R2610 wireless router used for establishing network connectivity in the setup.

the robot's workspace can be significantly increased. The TSL600 comes in different lengths, with versions available in 2, 3, and 4 meters, which correspond to the length of the linear motion base. This feature enables a single robot arm to perform tasks over a larger work area and perform for example welding tasks on structures or objects which would not be possible otherwise. The specifications of the TSL600 can be found in Appendix B.

The Motoman MT1 (shown in Figure 3.5) is a workpiece station that serves as a platform for holding a workpiece. The station enables rotation and positioning of the workpiece, even when a robot arm is stationary. This allows for operations on the workpiece that would not otherwise be reachable, expanding the range of possible tasks that can be performed by the robot system. See Appendix C for specifications.

3.4. Welding Equipment

The robot cell was equipped with two different welding systems, both manufactured by Fronius. The first system utilized the Fronius TPS400i as its power source, as seen in Figure 3.7. A MIG welding torch is connected to this system for the welding process, and a electrode wire feeder is mounted on the arm.

Figure 3.3.: Yaskawa Motoman GP 25-12 industrial robot arm equipped with TIG torch as end effector.

Figure 3.4.: Yaskawa Motoman GP 25-12 industrial robot arm mounted on Yaskawa Motoman TSL600 linear module, equipped with MIG gun as end effector.

Figure 3.5.: Yaskawa Motoman MT1 with workpiece.

The second arm of the robot system was equipped with the Fronius MagicWave 3000 as its welding power source (shown in Figure 3.6). A TIG welding torch is connected to the robot arm, and a wire feeder is mounted on the arm to supply filler material for the welding process.

Figure 3.6.: The Fronius MagicWave 3000 welding source.

3.5. The Robot Cell

The entire system can be observed in Figure 3.8. The system is contained within a transparent glass enclosure, forming a robot cell. The cell is equipped with a sliding door and a lock mechanism that is connected to the robot controller. This ensures that programs cannot be executed remotely when the door is unlocked, enhancing safety and security measures.

Figure 3.9 illustrates two significant transformations, accompanied by their corresponding transformation matrices as shown in Equations 3.1 and 3.2. These transformations were measured using basic tools, and it is important to acknowledge the possibility of some errors in the measurements.

Figure 3.7.: The Fronius TPS400i welding source.

Figure 3.8.: The image captures the complete robot cell, which includes two robot arms, a linear module (TSL600), a workpiece positioner module (MT1), and two welding apparatus.

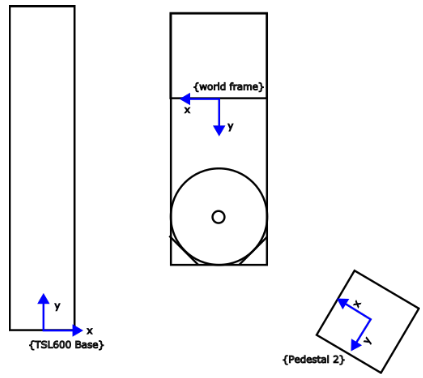


Figure 3.9.: The image showcases the world frame, the frame of the TSL600 base, and the frame of the mounting pendant within the system.

$$T_{worldframe,pedestal2} = \begin{bmatrix} Rot(z, -30.5\,\text{deg}) & t_1 \\ 0 & 1 \end{bmatrix}, t_1 = \begin{bmatrix} -1.10719347 \\ 2.225587 \\ 0 \end{bmatrix} \quad (3.1)$$

$$T_{worldframe,TSL600base} = \begin{bmatrix} Rot(z, 180\,\text{deg}) & t_2 \\ 0 & 1 \end{bmatrix}, t_2 = \begin{bmatrix} 1.58 \\ 3 \\ 0 \end{bmatrix} \quad (3.2)$$

Chapter 4.

Approach for Achieving the Stated Objectives

This chapter is dedicated to describing the methods that have been considered and employed to achieve the goals outlined in the introduction.

4.1. Communication

Establishing communication with the robot controller is vital to control and interact with the robot system effectively. This section outlines the methodology employed in this thesis to achieve communication with the robot controller, enabling precise command execution and real-time monitoring of the robot's state.

A fundamental requirement for successful communication is establishing a reliable and efficient communication channel. This allows for the transmission of control commands, receipt of sensor data, and exchange of information between the robot controller and external systems. By enabling bidirectional communication, advanced control algorithms can be implemented, sensors can be integrated for perception, and collaborative operations with other robotic components can be facilitated.

To fulfill these requirements, the MotoROS2 application was utilized in this thesis. MotoROS2, developed by Yaskawa America, Inc. and the Delft University of Technology [45], allows micro-ROS to run directly on the robot controller. This enables control over the robot system using ROS2 services and actions. Leveraging ROS2 provides the ability to program the robot arm using conventional programming languages, thereby enabling the creation of highly sophisticated programs incorporating advanced sensors without the need for extra hardware such as PLCs.

It is essential to acknowledge that MotoROS2 was in a closed beta phase at the time of this thesis, which means it was subject to potential changes and had limited documentation available. The thesis supervisor provided the necessary files for the implementation.

An overview of the MotoROS2 architecture can be seen in Figure 4.1, illustrating the integration of ROS2 with the robot controller for communication and control. From the figure, it can be seen that MotoROS2 enables the action "follow joint trajectory". This allows the robot controller to execute a preplanned trajectory. By leveraging this functionality, the details of how the robot initiates and moves can be abstracted, allowing for a focus on implementing ways for intuitive path description and sensor implementation.

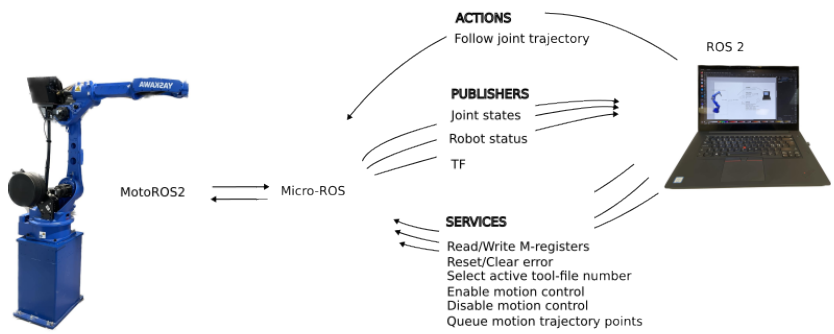


Figure 4.1.: System overview and functionality.

4.2. Simulation

To ensure efficient programming and operation of the robot system, it was crucial to implement a simulation environment. The simulation provided visual feedback, allowing the user to verify the correctness of the system and observe the desired behavior. Numerous simulators were available, each with its own set of features, such as physics simulation and collision detection. Therefore, a careful analysis was conducted to determine the required elements for the software selection.

Two types of simulators were considered, which can be described as kinematic simulators and dynamic simulators. Kinematic simulators focus on replicating the arm's movement, while dynamic simulators incorporate the study of motion caused by forces. Dynamic simulation is particularly valuable for tasks involving deformable objects or when movement and positioning within the work envelope are necessary. These simulators find applications in machine learning problems, such as peg-in-hole tasks.

For this thesis, the main important aspects were kinematics and collision detection. Collision detection plays a vital role in identifying and notifying the presence of collisions within the simulator. This feature assists in preventing collisions within the existing work system and minimizing potential damage.

Further consideration of software options can be classified into two categories: manufacturer-provided and non-manufacturer software.

Manufacturer-provided software, often referred to as robot simulation software or robot programming software, is specifically designed to support the programming and operation of industrial robots. These software products are typically developed by robot manufacturers themselves, ensuring seamless integration with their hardware. They offer accurate geometry and kinematic properties of the robotic systems and provide advanced visualization tools, optimization features, and control modules for additional equipment, such as conveyors. However, these solutions can be expensive and have restricted support for programming languages and customization options, as they primarily focus on the manufacturer's own robot models.

Examples of manufacturer-provided simulators include motoSIM by Yaskawa Motoman [22] and KUKA.sim by KUKA [16], with estimated prices of $8,000$ and $2,000$, respectively [11].

On the other hand, non-manufacturer software is created by third parties who may not necessarily produce physical products but offer virtual tools or services to enhance existing products or processes. These solutions are not tied to specific products and offer high customizability.

Robotsuite is a modular simulation framework designed for machine learning applications on robotic arms. It provides flexibility in choosing physics engines and includes pre-built robotic arm models. However, its focus on machine learning limits the customization of robot cells and systems [46].

ISAAC Sim, developed by NVIDIA, is a scalable robotics simulation application and synthetic data-generation tool. Powered by Omniverse, it creates photorealistic and physically accurate virtual environments, enabling faster development, testing, and management of AI-based robots [24].

Gazebo is a widely used open-source simulation software tool that facilitates the behavior and performance simulation of robots in various environments. It supports complex robotic system modeling and simulation, offering features such as 3D visualization, physics simulation, and sensor simulation. Gazebo supports multiple programming languages and benefits from an active community contributing to its plugins and models [7].

Rviz2 is a tool for ROS2, provides a 3D visualization environment for robotic data and models. While not strictly a simulator, it can function as a kinematic simulator with the help of external packages. Its modular nature enables its application as a simulation tool depending on the visualized data type, making it valuable for robotics research and engineering.

These are just some of the many tools that enable the simulation of the robot. To select simulation software, an analysis of the resources available and requirements necessary to achieve the goals described in the description of this thesis were conducted. Considering these factors, Rviz2 was deemed suitable for the simulation needs of the robot system, providing an efficient and effective environment for offline programming and visualization of the desired robot behaviors and interactions.

It is important to note that the selection of Rviz2 is based on the specific requirements and considerations of the thesis project. Other simulation environments mentioned earlier may also be appropriate depending on the project's objectives, constraints, and specific needs.

4.3. Calibration

To ensure precise control of the robot system, calibration becomes necessary. Calibration involves two main aspects: controller calibration and system description calibration. Controller calibration refers to the process of calibrating the robot controller itself. It involves determining the transformation of each joint in the system and providing this information to the software running on the controller. This allows the controller to accurately represent the joint positions and movements on the teach pendant or other user interfaces. In the scope of this thesis, the goals did not require any changes or adjustments to the controller calibration, and thus it was not necessary to modify this aspect.

System description calibration is the calibration of system description, more specifically the URDF. System description calibration is an essential step in achieving an accurate alignment between the virtual representation of the robot and its physical counterpart. In this study, a systematic calibration process was adopted to fine-tune the transformations within the URDF based on the observed deviations between the actual robot and its simulated representation. This was important because no description of the robot cell was found, and the positions and orientations of each robot component were unknown.

The calibration procedure involved jogging each end effector in the robot system into easily measurable positions. Through careful observation and analysis of the disparities between the real-world robot and its virtual counterpart, adjustments were made to the URDF parameters.

This iterative calibration process enabled the progressive refinement of the URDF, ensuring a more precise alignment between the physical robot system and its digital representation in the simulator.

It is important to note that the precision and accuracy of the calibration process outlined above heavily rely on the ability to accurately measure the real-world system. Any inaccuracies or uncertainties in the measurements can introduce deviations between the physical system and the calibrated model.

While every effort was made to ensure accurate measurements, it is important to acknowledge the inherent limitations and potential sources of error in the calibration process. The obtained results should be interpreted in light of these potential inaccuracies, and further research or refinements may be required to improve the precision of the calibration.

4.4. Motion Planning

To achieve precise control of the robot arm's end effector, an efficient and effective motion planning framework is essential. Motion planners play a crucial role in solving the inverse kinematics problem, enabling the robot to reach a desired goal. Over the years, extensive research has been conducted in the field of motion planning, resulting in the development of numerous algorithms.

Traditionally, algorithms like the A* (A-star) algorithm have been widely used for solving shortest path problems. However, these algorithms rely on predefined graph exploration, which becomes impractical for physical robots operating in environments with stringent precision requirements, such as achieving accuracy within 0.1 mm or less. Additionally, due to the high degree of freedom that a robot arm possesses, creating a map of all possible paths, as required by traditional path-finding algorithms, becomes virtually impossible.

To address these limitations, rapidly exploring random trees (RRT) was introduced as a sampling-based alternative to graph-based algorithms like A*. RRT* (RRT-Star) is a variant of RRT that generates more optimal paths between configurations, making it particularly suitable for complex environments [17].

In recent years, machine learning has gained popularity as an approach for controlling robot arms, particularly in dynamic and intricate environments. By training a neural network on a substantial dataset of example motions, the network can learn to predict the necessary joint positions to achieve a desired end effector pose, without relying on pre-defined planners or trajectories. Various machine learning algorithms are utilized for robotic motion planning, including deep reinforcement learning and supervised learning, and imitation learning. Deep reinforcement learning trains the neural network to take actions based on a reward signal, such as a score for successfully completing a task. Supervised learning trains the network using a labeled dataset of example motions, while imitation learning involves learning from an expert demonstration of the desired motion.

Despite the promising results demonstrated by machine learning approaches in specific applications, they also have certain limitations. One major drawback is the requirement of a large dataset of example motions for training the neural network, which can be time-consuming and costly to collect. Additionally, neural networks may struggle to generalize to novel scenarios beyond the training data, necessitating re-training or fine-tuning for each new task or environment. Nonetheless, machine learning holds significant potential for achieving advanced control of robot arms in the future, and its benefits may outweigh its limitations in specific applications [4].

The selection of the motion planner was driven by the goals outlined in the introduction, with a particular emphasis on leveraging open-source resources and the ROS 2 framework. After considering these factors, the MoveIt 2 platform was chosen. MoveIt 2 is an open-source platform that is widely used and well-documented. It provides the latest developments in planning and control, offering an intuitive setup of the robot and efficient planning capabilities. Additionally, MoveIt 2 comes integrated with the OMPL (Open Motion Planning Library) for advanced motion planning [33].

One significant advantage of using MoveIt 2 and OMPL is the comprehensive collection of planning algorithms they provide. This feature enables rapid switching between different planners based on specific requirements or scenarios. The availability of various algorithms in OMPL offers flexibility and adaptability in the planning process, empowering researchers and practitioners to explore multiple planning strategies and experiment with different approaches. This versatility allows for efficient optimization and customization of the planning process, ultimately enhancing the overall performance and effectiveness of the robotic system.

Moveit 2 also incorporates virtual controllers that utilize the aforementioned follow joint trajectory action. This integration enables seamless compatibility with the MotoROS2 application, leading to the development of a robust system.

4.4.1. Velocity Control

In general, the plans generated by a motion planner do not include information about time dependencies. This means that while the desired points to be reached are defined, the timing between these points is not specified. As a result, the velocities required for the robot arm are not determined. While robot controllers may still be able to interpret these plans, the resulting motion will be jerky and detrimental to the health of the robot arm. To address this issue, methods like time parameterization are employed, as mentioned in subsection 2.7.8.

Although there are several algorithms available for automatically post-processing the path with time parameterization to achieve smooth trajectories for robot arms, not all of them prioritize precise control over end-effector velocity. However, for certain tasks such as welding or applications that require slow and controlled movements, having precise control over end-effector velocity is crucial. Some planners include velocity constraints in their planning algorithms. However, it is important to note that MoveIt 2, which is primarily a kinematic motion planning framework, focuses on planning for joint or end effector positions rather than velocity or acceleration [35]. As a result, a post-processing method is necessary if MoveIt 2 and OMPL are used to achieve smooth and controlled robot motion.

Two methods are considered for post-processing the generated trajectory, manipulating the twist and utilizing time parametrization.

The twist tells about how the end effector moves relative to a frame. The v component tells about the change of position, which in turn results from the changes in the joints defining the robot arm, $\dot{\theta}$ as seen in Equation 2.17.

To determine the required joint velocities for achieving a desired end effector twist, the relationship between the twist, the Jacobian matrix, and the joint velocities can be utilized. If the inverse of the jacobian exists, then the joint velocities $\dot{\theta}$ from Equation 2.17, can be found by pre-multiplicating the inverse of the jacobian into the equation resulting in:

$$\dot{\theta} = J^{-1}\nu. \tag{4.1}$$

By knowing the desired end effector twist $\nu_{desired}$, the required joint velocities $\dot{\theta}_{req}$ can be obtained:

$$\dot{\theta}_{req} = J^{-1}\nu_{desired}. \tag{4.2}$$

In scenarios where the robot does not have a 6-axis configuration, the Jacobian matrix may not be square, leading to the absence of a traditional inverse. The pseudoinverse of the Jacobian matrix, a least squares method solution, is employed to address this limitation. The pseudoinverse enables estimation of the inverse even for singular or non-square matrices.

Further analysis of the twist reveals that while the twist captures the instantaneous linear velocity in a given configuration, evaluating the entire twist is necessary to determine the Cartesian speed. The twist, represented in the space frame, can be written as follows:

$$[\nu] = \begin{bmatrix} [w] & v \\ 0 & 0 \end{bmatrix} = \dot{T}T^{-1} = \begin{bmatrix} \dot{R} & \dot{t} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \dot{R}R^T & \dot{t} - \dot{R}R^T t \\ 0 & 0 \end{bmatrix}, \tag{4.3}$$

where $[\nu]$ is the skew representation of the t, $\dot{R}$ denotes the time derivative of the rotation matrix R, $\dot{t}$ represents the time derivative of the translation vector t, and R and t correspond to the rotation and translation components, respectively [18]. To ensure that the Cartesian velocity remains within the desired limit, the twist must be scaled such that $\dot{t}$ remains within the desired velocity limits:

$$\nu_{desired} = \alpha \nu, \tag{4.4}$$

where

$$\alpha = \frac{\dot{t}}{\text{Desired End Effector speed}}. \tag{4.5}$$

The second approach employed for velocity control is iterative time parametrization, which leverages the properties of time-parameterized trajectories. As the trajectory is computed by the planner, all the necessary information describing the path is known.

Iterative time parametrization involves calculating the forward kinematics for each point along the trajectory. This computation allows for determining the spatial difference between the end effector positions at successive time instances, such as between time t and $t+1$. Dividing this spatial difference by the desired velocity yields the corresponding time duration required for the end effector to traverse between the two points.

To ensure smooth and controlled movements, the iterative time parametrization technique is applied to each segment of the trajectory. By iteratively updating the trajectory based on the values obtained from the previous segment and using the computed time durations, it guarantees that the robot's speed is limited throughout the entire trajectory.

It is important to emphasize that both of these methods are designed to limit the speed to a specified value, which may result in trajectories with velocities lower than the desired speed. Additionally, it is crucial to exercise caution when utilizing the method that involves scaling of the twist, as it is an analytic approach that may generate results exceeding the physical capabilities of the robot if used imprudently. Therefore, careful consideration and validation of the resulting trajectories are necessary to ensure that they align with the robot's operational limits and constraints.

Chapter 5.

Implementation

This chapter aims to describe the process and implementation of the methods outlined in the previous chapter.

5.1. MotoROS 2

As previously mentioned, it is important to note that the MotoROS2 package was in a closed beta stage at the time of implementation. Therefore, the following description provides an overview of the version available during the implementation phase and may be subject to changes.

The MotoROS2 package provided consisted of the following five folders:

- config
- example_jobs
- motoros2-beta1-0.0.15
- motoros2-beta1-main
- motoros2_interfaces-beta1-main

Additionally, the package included the object file motoros2_yrc.o, and a readme.txt file. A description of the folders can be seen in Appendix D.

5.1.1. Installation

A generic wireless router was configured with a subnet mask of 192.168.255.x. The subnet mask was chosen because of the preset settings on the robot controller. This was essential for allowing communication between the robot controller and the host computer. Alternatively, the IP address of the robot controller could have been changed to match the router's settings, achieving the same outcome.

On the host computer, a static IP address of 192.168.255.10 was configured (shown in Figure 5.1). This particular address was selected to accommodate multiple clients on the same network while ensuring a consistent and known IP address assignment. It was crucial to define a specific IP address, as it needed to be specified for the MotoROS2 application, particularly the micro-ROS agent running on the application, to establish a connection successfully.

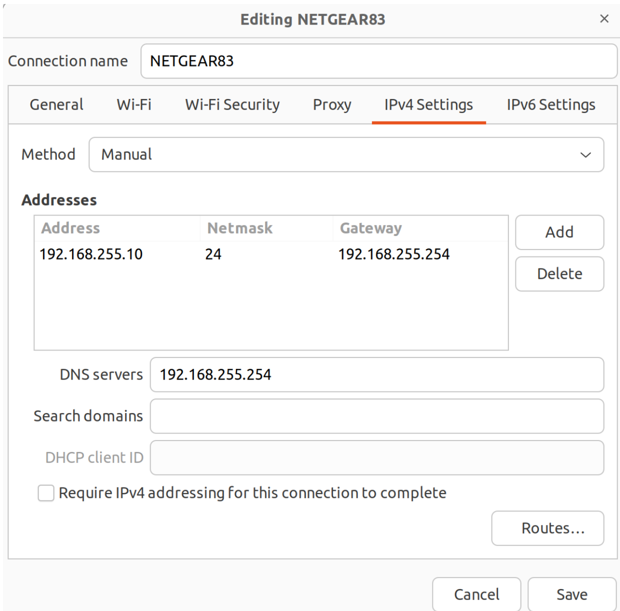


Figure 5.1.: The figure displays that the IP address has been manually set to the value described.

The motoros2_config.yaml file, which was located in the config folder of the MotoROS2 package, served as the configuration file for MotoROS2. This file contained various settings and parameters that defined the behavior of the MotoROS2 application. Specifically, in the motoros2_config.yaml file, the previously mentioned IP address was inserted in the agent_ip_address field as shown in Figure 5.2.

Furthermore, within the configuration file, the custom-defined names of each joint that define the robot system were specified. Since the robot cell comprised two Yaskawa Motoman GP25-12 robot arms, each with 6 joints, they were differentiated by using the prefix names "group_1" and "group_2". Additionally, the extra linear module TSL600 was assigned the prefix "group_3", and the work-

```yaml
#-----------------------------------------------------------------------------
# REQUIRED
# IP address and UDP port number of the Micro-ROS Agent PC. All communication
# to/from MotoROS2 will route through the Agent application.
# (There is no default value for these fields. They must be specified by
# the user.)
agent_ip_address: 192.168.255.10
agent_port_number: 8888


# Any settings that are not specified will be set to their DEFAULT value.


```

Figure 5.2.: Configuration of the agent IP for establishing a connection.

piece positioner module MT1 was assigned the prefix "group_4", as illustrated in Figure 5.3. As emphasized in the configuration, the order of the groups (modules) is important. Additionally, the number of joint names provided must exactly match the number of joints in the corresponding group.

```yaml
# Joints in this configuration file must be listed in the order of Robots,
# Base-axes, Station-axes.
#
# Example: R1+B1+R2+S1 system
#
# joint_names:
# - [r1_s_axis, r1_l_axis, r1_u_axis, r1_r_axis, r1_b_axis, r1_t_axis]
# - [r2_s_axis, r2_l_axis, r2_u_axis, r2_r_axis, r2_b_axis, r2_t_axis, r2_e_axis]
# - [b1_axis]
# - [s1_axis_1, s1_axis_2]
#
# When using a 7 axis robot arm with an elbow joint (E) in the middle of the
# arm, the elbow axis should be listed last (SLURBTE).
#
# DEFAULT: "group_x/joint_y"
joint_names:
 - [group_1/joint_1_s, group_1/joint_2_l, group_1/joint_3_u, group_1/joint_4_r, group_1/joint_5_b, group_1/joint_6_t]
 - [group_2/joint_1_s, group_2/joint_2_l, group_2/joint_3_u, group_2/joint_4_r, group_2/joint_5_b, group_2/joint_6_t]
 - [group_3/joint_1]
 - [group_4/joint_1, group_4/joint_2]


```

Figure 5.3.: Configuration of joint names.

A final modification was made in the field specifying the quality of service (QoS) of the ROS messages. Initially, the QoS profile of the joint_states publisher in the config file was set to "sensor_data", but it was later changed to 'default". This change was motivated by the observation that no data was being published during the initial attempt. Additionally, the config file mentions that certain applications may require the use of default values for QoS in order to ensure proper communication.

The configuration file, object file, and IONAMES.DAT file were uploaded to a freshly formatted USB stick and inserted into the teach pendant. The robot controller was turned on while holding down the "menu" button to start the controller in "maintenance mode". This provided access to the necessary settings for installing MotoROS2. The files were then uploaded and installed according to the instructions provided in the README.md file included with MotoROS2.

It is worth mentioning that the MOTOMAN DRIVER setting was adjusted to "USED" in the "System Info/Setup/Option Function" window within the teach pendant while the robot was in maintenance mode. This adjustment was made to address the expected alarm "8013 [0] Missing MotoROS2 cfg file" that was missing during the controller restart, during installation. As a result of this change, the alarm was triggered as described in the installation guide, indicating a successful installation.

On the host computer used for communication with MotoROS2, micro-ROS was installed directly on the system. Alternatively, micro-ROS could be run in a Docker container, although this approach was attempted but did not work on the current setup. The installation was verified by running the micro-ROS agent, which connected using the UDP4 protocol and the port specified in the config file. After a successful connection, the topics currently active on the network were listed using the following command:

```
ros2 topic list
```

which outputted "motoman_ab_cd_ef", where ab_cd_ef were the last 3 bytes of the mac-address of the controller, signaling that the ROS node on the robot controller was spinning.

Finally, a test was performed by setting the teach pendant to online mode, locking the door to the robot cell, and executing the following command:

```
ros2 service call /start_traj_mode motoros2_interfaces/srv/
StartTrajMode
```

The command returned a successful response, and an audible sound was generated by the servo motors, confirming that the servos on the physical arm were activated. This test verified the proper functioning of the trajectory mode and the activation of the robot arm.

5.2. MoveIt 2

The installation of MoveIt 2 was performed by following the installation guide provided on the official MoveIt 2 webpage [33].

5.2.1. Creating System Model

MoveIt 2 provides a setup assistant tool that assists with the creation of a package for robot control, including a planner and visual modeling. In order to set up a virtual robot system, it is necessary to define the URDF, which requires the 3D models of the system components. The 3D models for the robot system were obtained from online sources. The GP25-12 robot arms were acquired from [9], while the linear module and the workpiece positioner were obtained from [27] and [26], respectively.

A separate package was created for each unique robot model, namely the GP25-12, TSL600, and MT1. These packages were developed based on the existing packages provided by ROS-Industrial [34]. However, since the packages available in ROS-Industrial were designed for ROS 1, they could not be directly used without modifications. The package structure, including the launch and config folders, can be seen in Appendix E. Although the launch and config folders were included in the package structure, they were not utilized in this implementation.

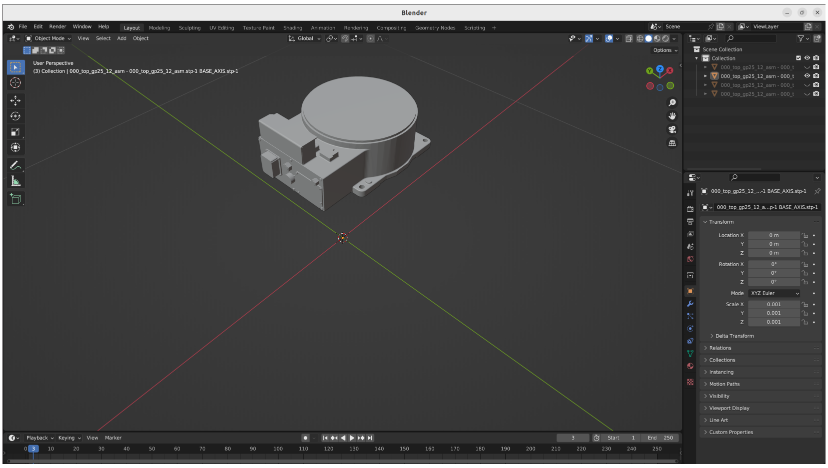
Each link from the 3D models mentioned above was imported into the 3D modeling software Blender [3]. This step was necessary to adjust the scale and position of each link to ensure they had the correct size and alignment. The reason for this adjustment was that different software applications interpret units differently. For example, if a model was created in millimeters [mm], other software applications might interpret it as meters [m].

Additionally, the URDF specification defines the rotation axis about the origin of the model, which may not correspond to the physical center of rotation, as depicted in Figure 5.4a. To address this, each link was moved so that it aligned with the physical center of rotation, resulting in the updated configuration shown in Figure 5.4b.

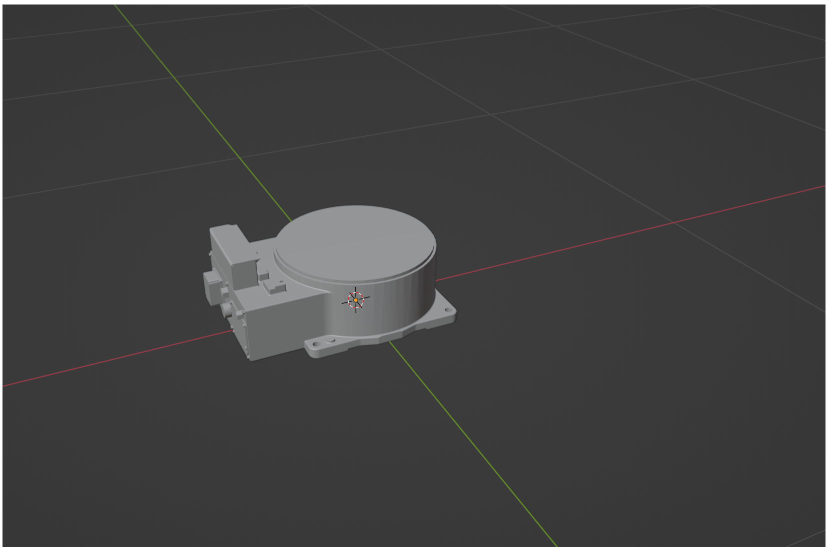
The xacro-files were created from scratch to define the robot's components and their parameters. The macro.xacro file was specifically designed to accept input parameters such as "connectedTo", "translation", "rotation", and "prefix". The "prefix" parameter was used to assign a group name to the robot, for instance, "group_1/" allowing multiple models to be generated from the same file without collision of names. On the other hand, "connectedTo" parameter specifies the parent link to which the base of the robot is connected. The "rotation" and "translation" parameters defined the transformation between the "connectedTo" link and the robot's base. This flexible setup enabled the convenient addition of pedestals and positions within the environment. Additionally, a virtual link and joint were included in the macro.xacro file to describe the transformation between the "connectedTo" link and the robot's base. Finally, a separate xacro file was created specifically for verification and testing purposes, ensuring that the transformations between each link were accurately defined and functioning as intended.

It was crucial to match the joint names defined in the macro.xacro files, with the joint names specified in the config file, which was loaded into the robot controller during the installation of MotoROS2, as a mismatch would cause the mismatched joint not to be found by the system.

After creating support packages for each robot component, a system support folder was created, named "gp25_sys_support", to describe the system as a whole. This



(a) A model of the robotic arm base, where the origin frame (intersection of the two orthogonal lines) is not aligned with the robot base.



(b) The model in the correct position, with the base and origin properly aligned.

Figure 5.4.: Alignment of the model and origin.

folder followed the same structure as the individual robot components described in Appendix E but included some additional models as described below.

To create visual and collision models for the workpiece, end-effectors, and pedestals, three extra folders were added to the "meshes/visual" and "meshes/collision" directories, each with their respective names. These models were designed using the CAD software SolidWorks [37].

In the URDF folder of the system support package, a .xacro file was created to define the robot components. This file imported the macro.xacro file for each component and specified material properties. A world link was defined as a common reference point. Since the robotic arms were mounted on pedestals, links, and joints for this connection were defined to describe their geometry and position. Each robot component was inserted with the appropriate prefix, parent link, and transformation.

The workpiece, used as a test case, had its geometry and transformation defined and connected to the appropriate parent link, in this case, "group_4/link_2". The end-effectors' geometry and transformation were also defined.

The packages were built and sourced. A file with the URDF file extension was generated from the system xacro file using the xacro package included with the ROS2 framework.

5.2.2. Creating Moveit package

In MoveIt Setup Assistant, the robot system URDF file was uploaded, and the setup guide provided on MoveIt's website [32] was followed. The following list outlines the crucial setup steps that enable the real word robot cell to be controlled:

- Collision Matrix: Collision matrix were generated without change of values.
- Virtual Joint: A virtual joint was defined between the world frame and the base link specified in the URDF
- Planning Groups: Planning groups were defined to specify subsets of the robot's joints used for motion planning.
- ROS 2 and MoveIt Controllers: ROS 2 controllers and MoveIt controllers were configured to enable control and motion planning for the robot.

Each robot component was assigned its own planning group, which contained only the joints responsible for the robot's movement. For example, since "group_1" was mounted on "group_3", the joints of "group_3" were included in "group_1". The default planner was set as RRTstar, and the default kinematic solver was set as the default "kdl_kinematics_plugin/KDLKinematicsPlugin," as the system comprised only open-loop robots.

Additionally, a common planning group named "follow_joint_trajectory" was defined, including all the joints in the system. This planning group is necessary because MotoROS2 expects joint values for each joint in the entire system and does not work with individual groups. As the system consists of 15 joints (6 + 6 + 2 + 1), the planning group also includes 15 joints. The name "follow_joint_trajectory" is used because MotoROS2 listens for this action to execute joint trajectories. By matching the name of the system planning group with the action server, the resulting configuration will allow MoveIt 2 to control the physical robot.

After generating the config package using the MoveIt Setup Assistant, the controller names in the file "moveit_controller.yaml" located in "moveit_config/config" were modified. The original controller name, "follow_joint_trajectory_controller," was changed to "follow_joint_trajectory." Additionally, the value of the "action_ns" variable was modified from "follow_joint_trajectory" to an empty string, "". This change can be found in more detail in Appendix F.

The virtual environment could be launched by executing the ROS2 launch file "demo.planning.py". This launch configuration sets up the necessary processes for visualization, planning, and controlling the system. However, it is essential to be aware that the configuration also started a virtual controller, which publishes values on the same topics as MotoROS2, primarily on the topic "/joint_states". This caused conflicts as the virtual robot and the physical robot may be at different positions, causing the system to jump between configurations. Consequently, planning may not be feasible. To address this issue, the frequency in the "ros2_controller.yaml" file located in "moveit_config/config" was modified. Specifically, the frequency was changed from the initial 100 Hz to 0 Hz, effectively preventing the virtual controller from publishing data. This final modification enables the control of the physical robot system using the Rviz2 and MoveIt 2 interface.

An alternative solution was created, and instead of running the "demo.launch.py" configuration with multiple unnecessary processes, the individual launch files "rsp.launch.py" and "move_group.launch.py" could be launched. For visualization purposes, the launch configuration file "moveit_rviz.launch" was used. However, running these launch files individually requires opening three separate terminal windows. To simplify this process, a new launch file was created to start the three desired processes simultaneously, which can be seen in more detail in Appendix G.

5.3. Ros2 Main Package - planner_node

The ROS2 main package consisted of a single node named "planner_node". This node was designed to provide two services: one for planning the robot's trajectory and another for executing the trajectory of the robot. Additionally, the planner_node had the objective of providing information about the robot's position. This functionality was crucial, as explained in section 2.9, to ensure that only robot movements were controlled by this node. By having this capability, the activation of the welding tool could be controlled by other nodes.

The planner_node consisted of a single class named "WaypointListener" with the following structure:

- Class WaypointListener

  Subscriptions:

  - joint_state_subscription

  Publishers:

  - ready_publisher_

  Services:

  - plan_service_
  - execute_service_

  Private core functions:

  - update_thread_function
  - create_plan
  - Callback functions

5.3.1. Planning Functionality

The class implemented a service method to enable the planning service named /plan_group. The callback function associated with this service was responsible for pre-processing the received message defining the planning request, and passing the data to the private function create_plan, which performed the main planning work. Once the plan was created, the callback function notified that the plan was ready for execution and responded with the fraction of the trajectory that was planned. The implementation can be seen in Appendix J.

Using the MoveIt 2 planning interface, a Cartesian path for the given group was calculated based on the waypoints acquired from the message. An algorithm then processed the calculated trajectory to limit the end effector's Cartesian speed to the velocity defined in the message, resulting in a trajectory for the given group with the desired speed.

Since the planning was performed for a single group, an algorithm was developed to manipulate the trajectory. This algorithm ensured that the trajectory contained all the joint values for each joint in the system. The trajectory was then expanded to have the same length as the calculated trajectory for the single group. Finally, the index position of the group in the global trajectory was determined, and the planned trajectory was inserted at the correct position. The implementation of this algorithm can be seen in Appendix I in the function addToPlan.

End Effector Speed Limiter

To limit the speed of the end effector, iterative time parameterization was implemented. The method used was originally developed by Benjamin Scholz and Thies Oelerich [38]. Some minor changes were made so that the method worked standalone with the package presented in this thesis and did not require modification of the external MoveIt 2 framework.

The second suggested method for limiting the end effector's Cartesian velocity involved manipulating the twist of the end effector and calculating the resulting joint velocities required. This implementation was done in Python, allowing for efficient and intuitive development and solution. The numerical math library numpy was utilized for its computational capabilities.

To implement this method, a new ROS2 node named "jacobian_generator" was created. The URDF for the system was loaded, and the kinematics were built following the product of exponential convention. The trajectory was loaded into the script, and the jacobian and the twist were calculated for each point in the trajectory. The desired twist was calculated by scaling the end effector's twist, as described in Equation 4.5.

The rate of change is calculated using the forward kinematics method from [18] and is divided by the time it takes to move between the two points. The desired joint velocities were acquired from Equation 4.2. For each position, the end effector's time to reach the next point was calculated similarly to the iterative time parameterization method. The implementation can be seen in Appendix K.

5.3.2. Implementing End Effector In-Position Publisher

To ensure the continuous calculation of the robot's forward kinematics during plan execution, a dedicated thread was created. This thread performed forward kinematics calculations for the current robot position while the execution was in progress. The use of a separate thread was necessary due to the blocking nature of the trajectory execution function in the MoveIt 2 interface. Additionally, to handle multiple concurrent callbacks, a callback group, and a multithread executor were implemented for the node. This was crucial because, without these, only one callback could be executed at a time, causing issues such as the joint positions needing to be updated. Similarly, a callback group was defined for the ready_publisher to address the same concern.

The forward kinematic poses and the waypoint poses were compared. If the difference between the two poses fell within a specified tolerance (set to 0.01 in this thesis), a private variable named "in_position" was updated to indicate whether the robot was in position. This updated "in_position" variable was then published over its corresponding topic at a predefined frequency of 50 Hz. For detailed implementation information, see Appendix J, specifically the function update_thread.

5.4. Ros 2 Interface Package

An interface package was created to define custom-defined ROS2 services and messages. The services and messages were both created with the goals mentioned in the introduction of this thesis in mind and, as a result, did only consist of data necessary to control the robot.

The main message was a custom made massage defined as the following and included the necessary information required to plan a trajectory:

```
waypoints.msg

    string groupname
    geometry_msgs/Pose[] waypoints
    bool[] is_job
    float32 speed
```

Because the system was made to be able to control multi-arm systems, it was crucial to specify what group which was desired to move. This was done with the variable groupname in the message. If one were to move the arm "group_2", this would be specified in the message such that the program could calculate the trajectory for this group.

The path that the end effector was intended to follow was defined as a list of poses called waypoints. Each waypoint described a point and an orientation that the end effector was meant to visit. The waypoints were ordered such that the first waypoint represented the start of the trajectory, the second waypoint indicated the next point in the trajectory, and so on. The message type geometry_msgs/Pose was chosen to represent the waypoints, as it is used by the Cartesian planner for planning and provides a compact way of conveying the necessary information.

To specify whether the end effector should be activated or deactivated, a variable named “is_job” was defined as a list of booleans. Each index in the is_job list corresponded to a waypoint with the same index such that waypoints[i] = is_job[i]. If, for example, a welding path were defined as waypoints[i] and was finished at waypoints[i+k], the values of is_job[i], is_job[i+1], ..., is_job[i+k-1], were set to true, while the end of the path was set to is_job[k] = false.

Lastly, a speed variable was defined such that the user was able to control the end effector speed used by the methods described above.

The service /execute_plan was defined as follows:

```
Execute.srv
    ---
    bool success
```

The service had no inputs and a single boolean *success* as output. If the service were called, the plan created from the waypoints.msg was executed on the robot system.

The final service definition used to call the service for planning the trajectory is defined as follows:

```
Plan.srv
    Waypoints waypoints
    ---
    float32 trajectory_fraction
```

This service consists of a “Waypoints” message and has a return value of the fraction of the successfully planned trajectory, giving the user or application information if the system was successful with the planning. If the value was equal to 1, then all waypoints were planned for, and thus the executed service can be called safely.

5.5. Usage

The ROS2 package provided in this thesis has dependencies on several external packages. To ensure the package’s successful usage, ensure the following packages are included in the workspace. If they are not included by default in the ROS2 distribution, they will need to be manually installed:

- control_msgs

- indusrial_msgs
- motoros2_interfaces - A part of the MotoROS2 repo
- sensor_msgs
- std_srvs
- tf2_msgs

Usage (assuming Linux):

1. Download, build and source MoveIt 2 framework [32].
2. Build and source the workspace proposed in this thesis:

```
        cd ws_master
        colcon build
        source install/setup.bash

```

   If the physical robot is present, and connected to the same local network as the host computer:

   a) Run the micro-ROS agent with the port defined in the MotoROS2 configuration:

   ```
           ros2 run micro_ros_agent micro_ros_agent udp4 --
       port 8888

   ```

3. launch the systems robot state publisher and move_group launch files

```
        ros2 launch robco rsp.launch.py
        ros2 launch robco move_group.launch.py
        #for visualization
        ros2 launch robco moveit_rviz.launch.py

        # Alternatively, a single command for all three, launch
    the custom launch-file (saves terminal windows)
        ros2 launch robco planning.launch.py
```

   In the move_group window, after successfully loading, it should be printed something along "[move_group-1] You can start planning now!". If micro_ros_agent is connected, the virtual robot in the Rviz2 window should snap to its appropriate place.

4. Run the planner node presented in this thesis

```
        ros2 run moveit_group_planner planner_node

```

The planner_node window should say, "Node is spinning, ready to take waypoints." At this point, waypoints can be sent via the plan_group service. This can be tested with the supplied package waypoint_publisher

```
    ros2 run waypoints_publisher waypoint_publisher
```

After planning, the service should return with a value between 0 and 1, representing the fraction of the trajectory successfully planned.

5. If the physical robot is online, and the host computer running the planner_node is connected to the same network, the servos can be activated by running

```
    ros2 service call /start_traj_mode motoros2_interfaces/srv/
   StartTrajMode
```

An audible click sound should be made. It is then possible to execute the plan on the physical robot with the following command. This is currently only available if the physical robot is connected.

```
    ros2 service call /execute_plan
   moveit_group_planner_interfaces/src/Execute
```

Chapter 6.

Experiments

This chapter describes the setup for testing the solutions. All the implementations of the tests can be seen in Appendix L.

6.1. Case

To simulate real-life scenarios, a workpiece was acquired from the laboratory for the experiments. The selection process for the workpiece was random, as multiple workpieces were available for testing purposes. The chosen workpiece, in this case, is made of aluminum. The workpiece consists of a solid plate measuring 1 meter by 0.8 meters. On top of this plate, there are pieces of U-profiles measuring 13 centimeters by 13 centimeters. This U-profile is positioned in such a way that it creates a cross configuration on the surface of the plate. The workpiece can be seen in the lower left corner in Figure 3.3. The presence of the workpiece adds complexity to the experiments and allows for testing under conditions that closely resemble real-world scenarios.

The robot system will initially be at its "ready" configuration for all tests. This configuration is a custom-defined configuration where all the joint values are at zero, except the linear motion joint on the TSL600, which is set to 1. This configuration allows for quick resetting when the robot is in most poses.

Pose "Ready":

$$0 \text{ for all joints except group 3, which is set to } \theta = 1$$

Group 3 was assigned a value of 1 due to the clustered nature of the robot cell scene. This clutter poses a significant collision risk with other groups, leading to an invalid path being generated by the planner in most configurations when group 3 is set to 0. Furthermore, for post-processing, the trajectories to limit the end effector velocity set to 0.1 m/s or 10 cm/s. This value allowed the robot to finish the movement within a reasonable time while simultaneously allowing the velocity limiter to work.

6.2. Definition of Object Origin

The definition of object origin is essential for allowing efficient description of coordinates in the real world. While the URDF and the robot system described in the previous chapters have their global "world"-frame defined, this may not lie in a place that is intuitive to use for specifying points and tasks. For instance, in the package defining the robot system, the world frame is inside the MT1 positioner, and therefore the application of a transformation is beneficial, such that points can be represented in the frame of the origin of the workpiece. For this system, this transformation was defined as:

$$T_{world,o} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1.532 \\ 0 & 0 & 1 & 0.575 \end{bmatrix},$$

which corresponds to the transformation from the world frame to the top plate, of which objects can be placed on the MT1 platform. The values 0, 1.532, and 0.575 were then defined as cx, cy, and cz, respectively.

6.3. Tests

A series of tests were defined to test the ability of the system to plan and follow desired paths. A total of 4 tests were created for evaluation, each with its own aspects. Test 1 and 2 were tests for simple geometric movements where the main goal was to evaluate the resulting trajectory without the opportunity for collision with the environment. Test 3 and test 4 were created to evaluate the ease of creating welding paths and following edges of the physical workpiece mentioned above. The tests were conducted iteratively with the development to verify the development of the control application.

It is important to note that parallel work was carried out utilizing the Fronius TPS400i welding apparatus. As a result, the focus was shifted to the arm using the same welding apparatus. Therefore, the later tests (Test 3 and Test 4) were not intended to be used for the other arm.

Test 1: Straight line motion

The initial evaluation involved testing the robot to follow a straight path. Specifically, the objective was to move to the center of the workspace, slightly above the rotation platform of the MT1, proceed to the edge of the workpiece, and ultimately to the opposite edge of the workpiece. The end effector's orientation was such that the approach of the end effector pointed downward, resulting in a constant orientation in the defined path. Furthermore, acceleration and deceleration will be necessary to ensure that the trajectory is smooth because of the turn the arm was required to take when the end of the workpiece was reached. If the trajectory then is free of jerk, then the planner is usable and further development can be done.

To achieve this, the service plan_group was called for the points listed in Table 6.1, where rx, ry, rz, and w correspond to the quaternion representing the orientation of the end effector relative to the world frame of the system. The z value was also chosen to ensure a safe distance from potential collisions within the workspace, with an appropriate margin.

Table 6.1.: Waypoints for test 1

Waypoint	**X**	**Y**	**Z**	**rx**	**ry**	**rz**	**w**
Waypoint 1	cx	cy	0.76	1	0	0	0
Waypoint 2	cx + 0.5	cy	0.76	1	0	0	0
Waypoint 3	cx - 0.5	cy	0.76	1	0	0	0

Test 2: Circular Motion

In order to evaluate the system's capability to generate smooth trajectories for paths that involve arcs, an experiment using a circular path was conducted. The path was defined by employing the following formula:

$$\begin{aligned} x &= r \cdot \cos\theta \\ y &= r \cdot \sin\theta, \\ z &= 0.73 \end{aligned}$$

where x, y, and z represent the Cartesian coordinates in space. The variable θ was discretized with a specified resolution of 100 as an input to the system. The z value was set to 0.73 to avoid collisions with the workpiece, similar to test 1. Finally, the r was defined as 0.5 meters. The waypoints defining this test can be seen in Table 6.2. The index i describes the i-th waypoint out of N.

Table 6.2.: Waypoints for test 2
where N = 100 and r = 0.5.

Waypoint	X	Y	Z	rx	ry	rz	w
Waypoint i	cx + $r\cos(\frac{2\pi i}{N})$	cy + $r\sin(\frac{2\pi i}{N})$	0.73	1	0	0	0

Test 3: Inside Job

In the third test of the experimental setup, a straight line motion was performed within the space enclosed by the U-profile of the workpiece. Contrasting the previous two tests that focused on welding operations on the workpiece's outer surface, this specific test aimed to assess the planning and execution of a path under strict constraints regarding movement and poses. The confined space within the U-profile posed a challenge, necessitating precise control and coordination of the robotic system to navigate and execute the desired motion accurately.

The purpose of this test was to assess the system's ability to generate trajectories that adhere to the limited space and pose constraints imposed by the U-profile. The objective was to validate the system's capability to plan and execute movements within restricted areas, ensuring the feasibility and accuracy of the welding process even when confronted with limited freedom of motion. This test was exclusively performed for the robot arm connected to the Fronius TPS400i. This was due to the welding torch of the end effector, which had a 34-degree downward angle, allowing for proper orientation within the enclosed space. The waypoints defining this test can be seen in Table 6.3.

Table 6.3.: Waypoints for test 3

Waypoint	X	Y	Z	rx	ry	rz	w
Waypoint 1	$cx + 0.5$	cy	1.9	-0.5721	0.5721	0	-0.5878
Waypoint 2	$cx + 0.6$	cy	0.75	0.8569	-0.5146	0.0217	-0.0193
Waypoint 3	$cx + 0.55$	cy	0.60	0	$-\frac{\sqrt{2}}{2}$	0	$-\frac{\sqrt{2}}{2}$
Waypoint 4	$cx + 0.50$	cy	0.60	0	$-\frac{\sqrt{2}}{2}$	0	$-\frac{\sqrt{2}}{2}$
Waypoint 5	$cx + 0.35$	cy	0.60	0	$-\frac{\sqrt{2}}{2}$	0	$-\frac{\sqrt{2}}{2}$
Waypoint 6	$cx + 0.55$	cy	0.60	0	$-\frac{\sqrt{2}}{2}$	0	$-\frac{\sqrt{2}}{2}$
Waypoint 7	$cx + 0.6$	cy	0.73	0.8569	-0.5146	0.0217	-0.0193

It should be noted that the values for this test, especially those describing the orientation, were acquired by manually jogging the robot to an orientation that visually appeared appropriate. While this method may not provide precise numerical values, it suffices for the purpose of this specific test, as the main focus is on evaluating the system's performance within the limited space of the U-profile.

Test 4: "Weld" Test

This test aimed to simulate a real-life scenario where the end effector's orientation is aligned with the edge of the workpiece during welding. The weld was performed as a straight line parallel to the y-axis of the world frame. The orientation of the end effector was set by a sequence of rotations: first, a 180-degree rotation around the x-axis, followed by a 45-degree rotation around the z-axis, and finally, a -45-degree rotation around the x-axis, resulting in the end effector pointing downward, and with an angle at 45 degrees in both the direction of the weld and up from the horizon.

A welding path was defined based on the edge of the workpiece, utilizing coordinates extracted from the CAD model using SolidWorks as seen in Figure 6.1. To accommodate for the orientation and diameter of the welding gun, the waypoints needed to be offset in the negative z direction of the end effector's pose by a constant value. Leveraging the equation Equation 2.1 and the known end effector orientation, this offset was easily achieved. A point $t = [0, 0, -\text{offset}]$ was defined, and employing Equation 2.8, the resulting coordinates δ were then added to the coordinates that defined the welding path. The offset was calculated as $D_{\text{ee}}/2 + 5\text{mm}$ tolerance, where $D_{\text{ee}} = 25mm$ represents the diameter of the end effector, asserting the end effector gets as close to the welding path as possible without touching the workpiece, as shown in Figure 6.2. The resulting definition of the path can be seen in Table 6.4.

Table 6.4.: Waypoints for test 4.

Waypoint	X	Y	Z	rx	ry	rz	w
Waypoint 1	cx + 0.5	cy	1.4	-0.5721	0.5721	0	0.5878
Waypoint 2	cx + 0.065 + δ_x	cy + 0.065 + δ_y	0.73 + δ_z	1	0	0	0
Waypoint 3	cx + 0.065 + δ_x	cy + 0.065 + δ_y	0.73 + δ_z	0.8536	-0.3536	-0.1464	0.3536
Waypoint 4	cx + 0.065 + δ_x	cy + 0.065 + δ_y	cz + 0.01 + δ_z	0.8536	-0.3536	-0.1464	0.3536
Waypoint 5	cx + 0.065 + δ_x	cy + 0.4 + δ_y	cz + 0.01 + δ_z	0.8536	-0.3536	-0.1464	0.3536
Waypoint 6	cx + 0.065 + δ_x	cy + 0.4 + δ_y	cz + 0.13 + δ_z	0.8536	-0.3536	-0.1464	0.3536

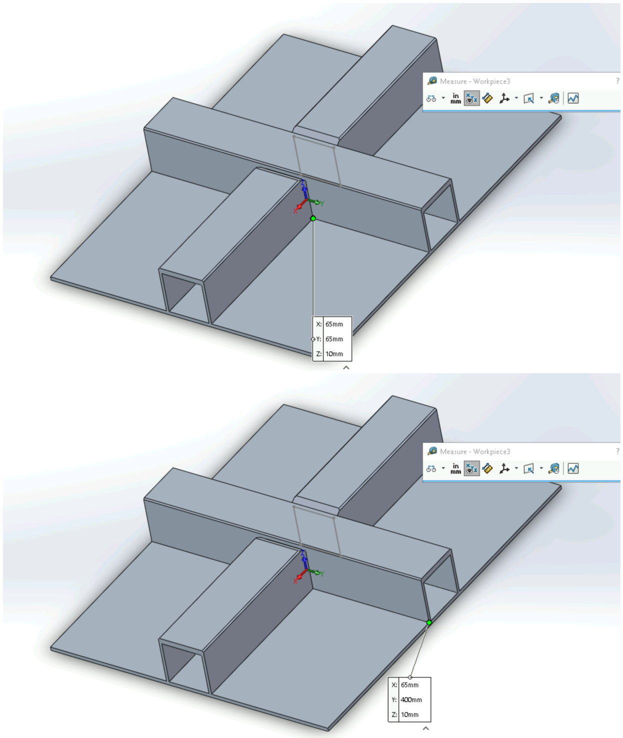


Figure 6.1.: The coordinates defining the start point (left) and end point (right) of the weld are extracted directly from CAD software.

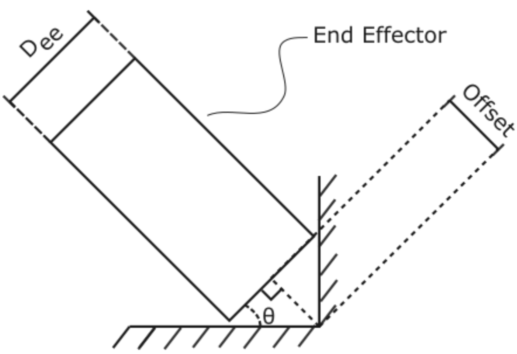


Figure 6.2.: The offset of the end effector.

Chapter 7.

Results

7.1. The Planning Package

This thesis presents a package or, more specifically, a workspace that includes multiple packages for ROS 2. The workspace comprises support packages for the robots used in this thesis, a MoveIt config package for the system used, and the main package *planner_node*. Furthermore, it includes the experimental package jacobian_generator and the testing package waypoint_publisher.

The planner node is the main work. It is a barebone ROS 2 node that takes the custom-defined service *Plan*. Plan has the request data “waypoints” which is a custom-defined message, and a response trajectory_fraction which is the fraction of the total plan which was followed. The message waypoints defined a list of waypoints that the end effector must reach, a list of booleans that correspond to each waypoint, defining if the waypoint is part of the welding trajectory, which group should be planned for, and the speed limit. The node then uses MoveIt 2 and OMPL to create a trajectory for the given group. If the speed limit is set, iterative time parameterization slows the end effector down.

Additionally, the node has a service “execute” that calls for the execution of the plan. During execution, a separate thread listens for joint_states from MotoROS2. It calculates the forward kinematics for the given group to see if the end effector has reached the waypoints defined in the received message. If the waypoint is defined as “is_job”, it will be stored.

Finally, the node publishes the topic “/ready” if the end effector is in the desired position. The purpose of this function is to tell if, for example, the welding gun is in position for activation. By default, it publishes with a frequency of 50 Hz and is set to false. The overall schematics can be seen in Figure 7.1.

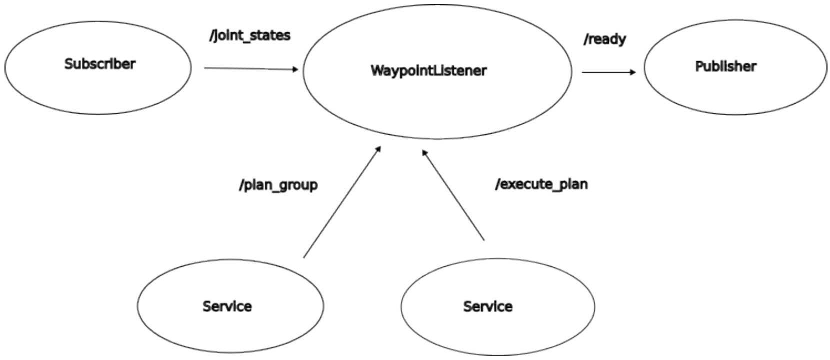


Figure 7.1.: The schematics of the node waypointlistener.

7.2. The Virtual Environment

The virtual environment representing the robot cell is depicted in Figure 7.2. It includes the two robot arms, the workpiece, the workpiece positioner MT1, and the linear module TSL600. The virtual environment incorporates collision and visual models of the robot system, enabling collision detection and planning within the robot cell.

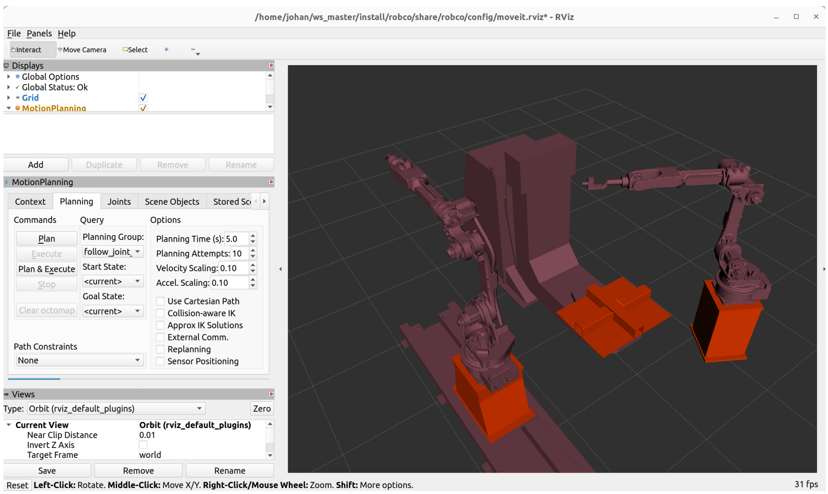
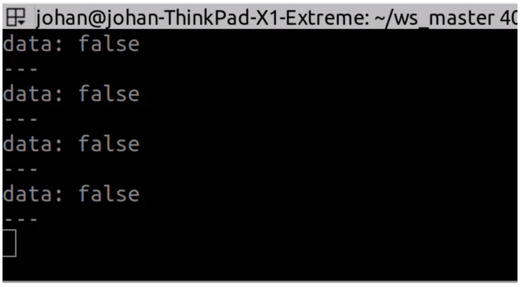


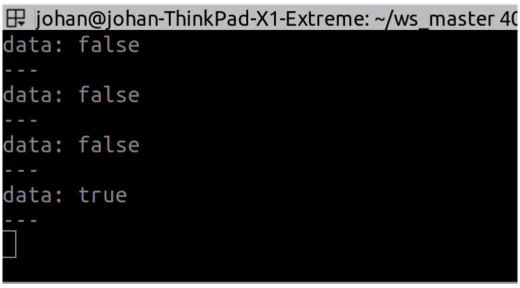
Figure 7.2.: The figure shows the virtual environment created in this thesis, visualized with Rviz2.

7.3. The In-Position Publisher

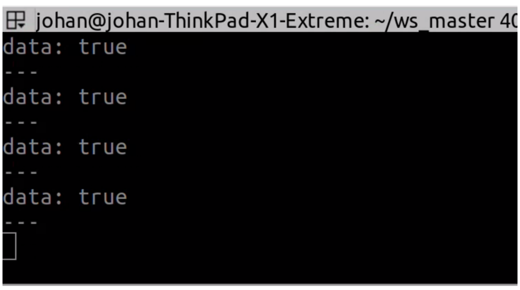
The in-position was successfully implemented, and the result can be seen in Figure 7.3. While the end effector is located in a trajectory segment not designated as a job, the corresponding data is marked as false, as illustrated in Figure 7.3a. Once the end effector reaches a waypoint defined as a job, the data is updated to true, as demonstrated in Figure 7.3b. This updated status remains in effect until the end of the trajectory or until a point is reached where the job is set back to false.



(a)



(b)



(c)

Figure 7.3.: The figures illustrate the variation in data as the end effector approaches the waypoint set for welding, indicating the activation of the welding torch.

7.4. The Velocity Limiter

The result of the velocity limiter implementation can be seen in Table 7.1 for test 1, test 2, test 3, and test 4, as well as in the plots shown. Due to the simple geometry of the paths, the distances of the paths were known, and the estimated time could be calculated by dividing the distance by the velocity specified in chapter 6. The measured times were collected from the plots.

Table 7.1.: Distance and time for the tests.

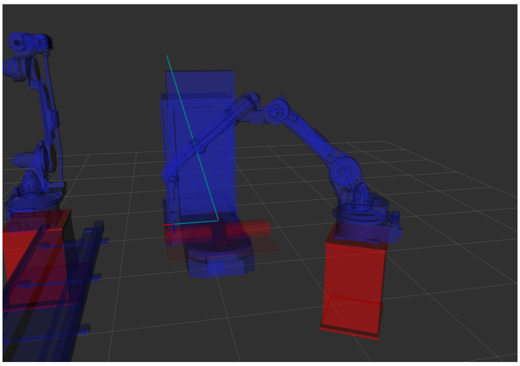
Test	**Distance** [m]	**Estimated Time** [s]	**Measured Time** [s]
Test 1:	3.0687	30.687	32.41
Test 2	4.7137	47.137	49.21
Test 3	3.8750	38.750	42.78
Test 4	3.5934	35.934	39.49

7.5. Test 1: The Linear Motion Test

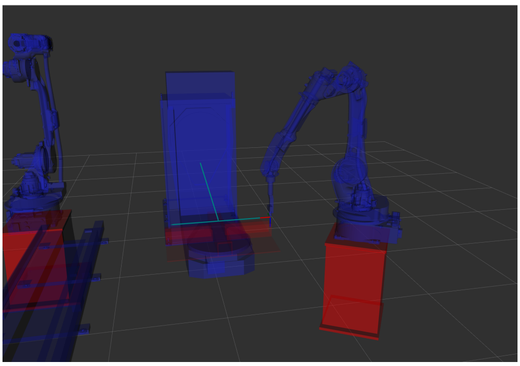
The test demonstrated the system's capability to generate trajectories representing straight lines while maintaining smooth motion and velocity limiting. The resulting path, depicted in Figure 7.4, has been split into two figures due to the software's limitation on the length of the tail. Despite this limitation, the path clearly showcases the successful execution of the straight-line trajectory. The system effectively maintains a consistent direction and accurately and precisely achieves the desired linear motion. The velocity profile of the end effector is shown in Figure 7.5 and 7.6.

7.6. Test 2: The Circular Motion Test

The test results indicate that the system is capable of generating smooth trajectories for arcs, as demonstrated in Figure 7.8, using the implemented Cartesian planner. However, when the arc is represented with low resolution, the trajectory becomes more jagged, as depicted in Figure 7.9. Additionally, the impact of arc resolution is evident in Figure 7.10, highlighting that high resolution of arcs does not negatively affect the trajectory. The velocity profiles of the end effector can be seen in Figure 7.11 and 7.12.



(a) End effector trail, straight line part 1.



(b) End effector trail, straight motion part 2.

Figure 7.4.: The trail of the end effector represents the trajectory for straight-line motion.

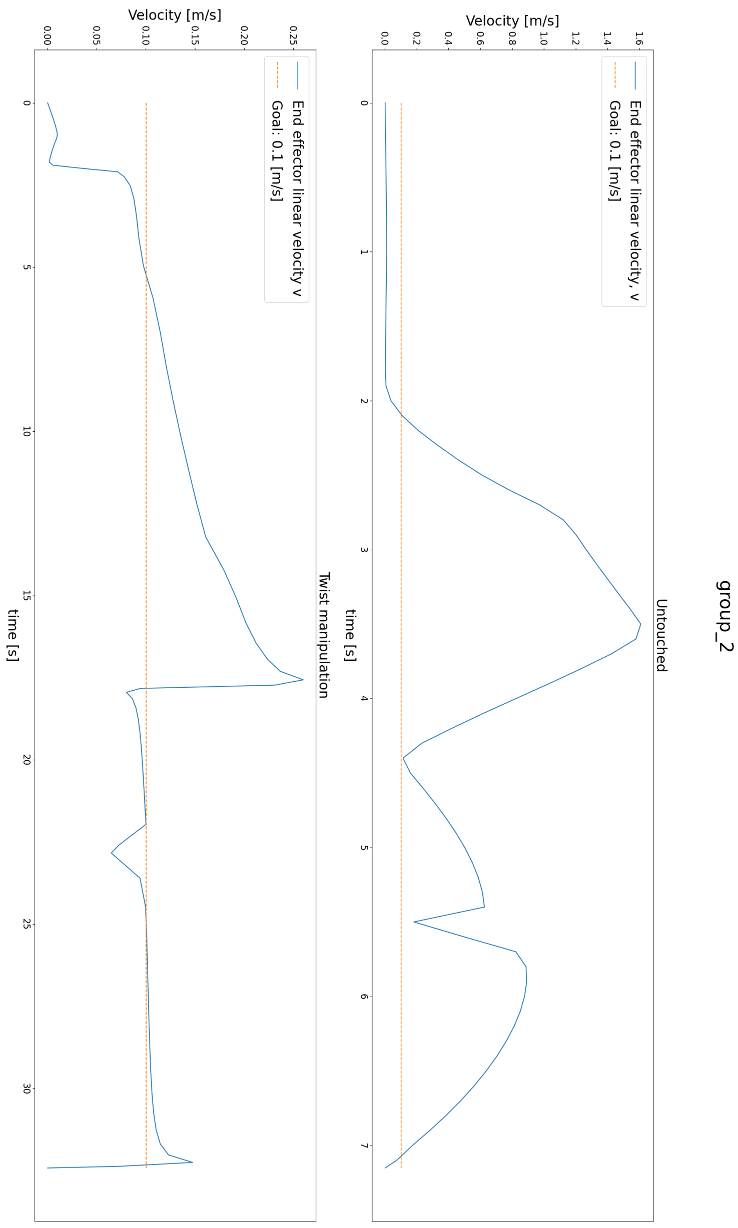


Figure 7.5.: Linear velocity for straight-line motion. Untouched (top) and twist method (bottom).

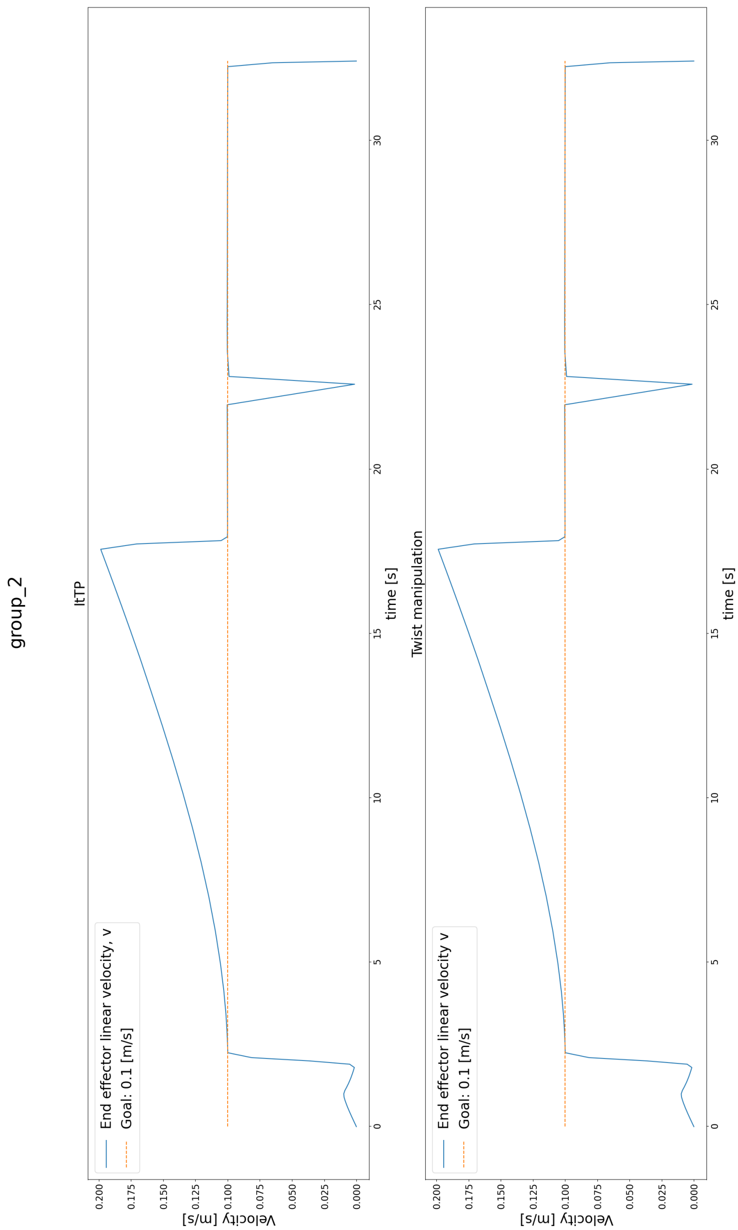


Figure 7.6.: Linear velocity for straight-line motion. Iterative time parameterization (top) and twist method (bottom)
.

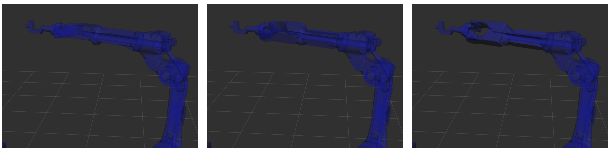
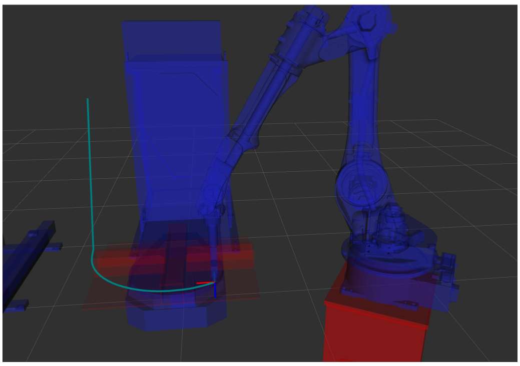
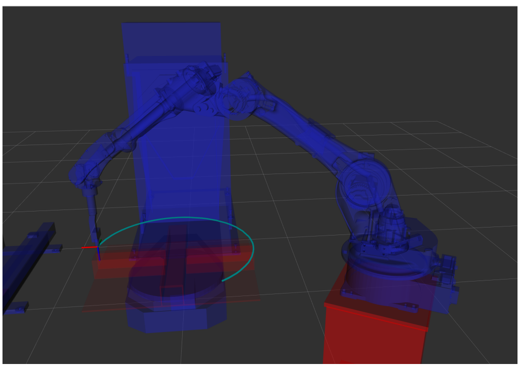


Figure 7.7.: Initial movement.



(a) End effector trail, circular motion part 1.



(b) End effector trail, circular motion part 2.

Figure 7.8.: The trail of the end effector represents the trajectory. for circular motion

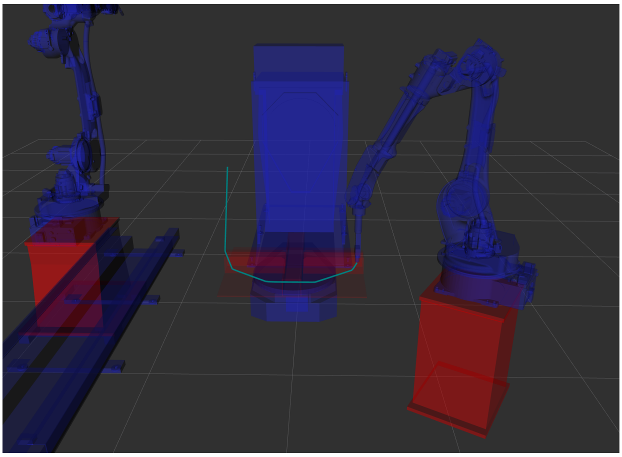
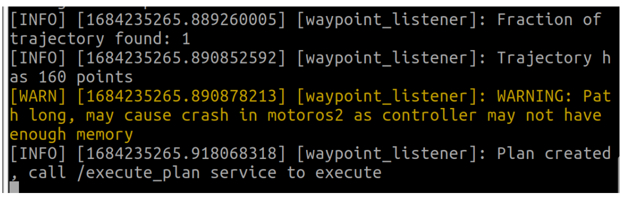
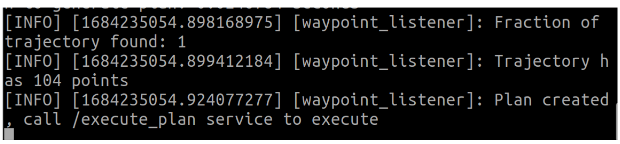


Figure 7.9.: The trail of a low sampled arc.



(a) The generated path with length 160 when a circle is represented with $n = 15$ points.



(b) The generated path with length 104 when a circle is represented with $n = 1000$ points.

Figure 7.10.: The generated path lengths for a circle with resolutions $n = 15$ and $n = 1000$.

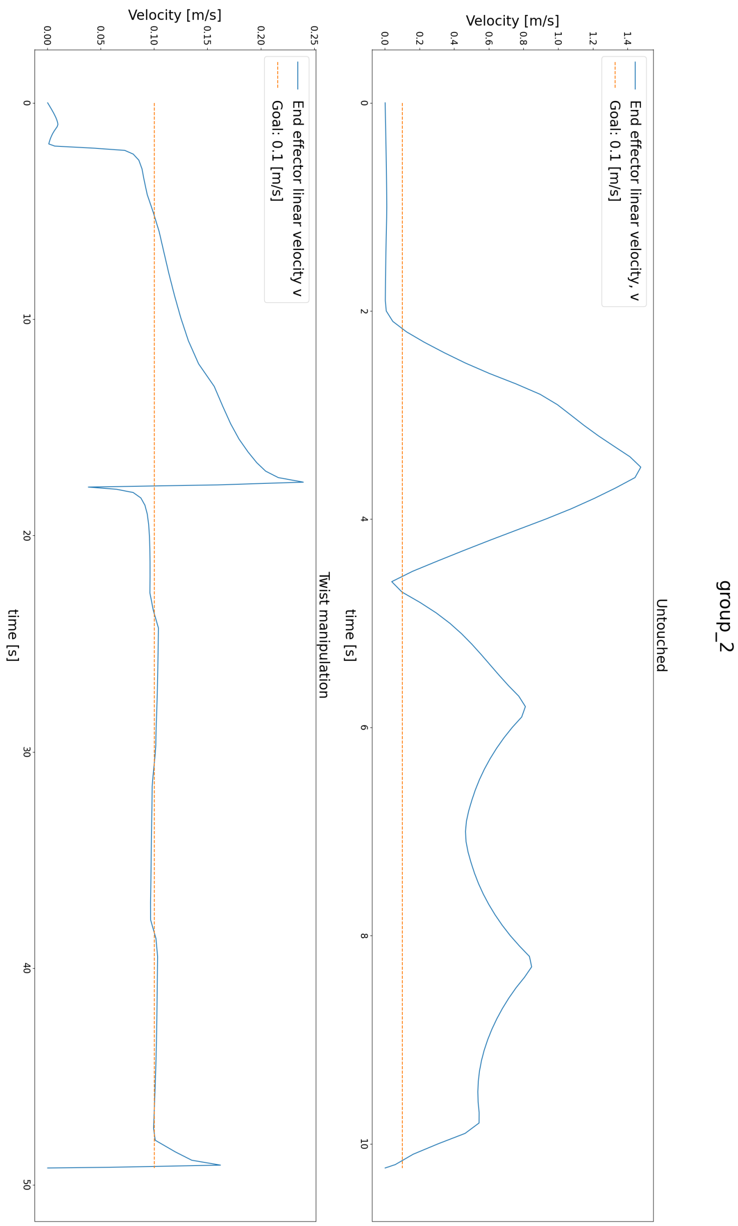


Figure 7.11.: Linear velocities for circular path Untouched (top) vs twist (bottom)

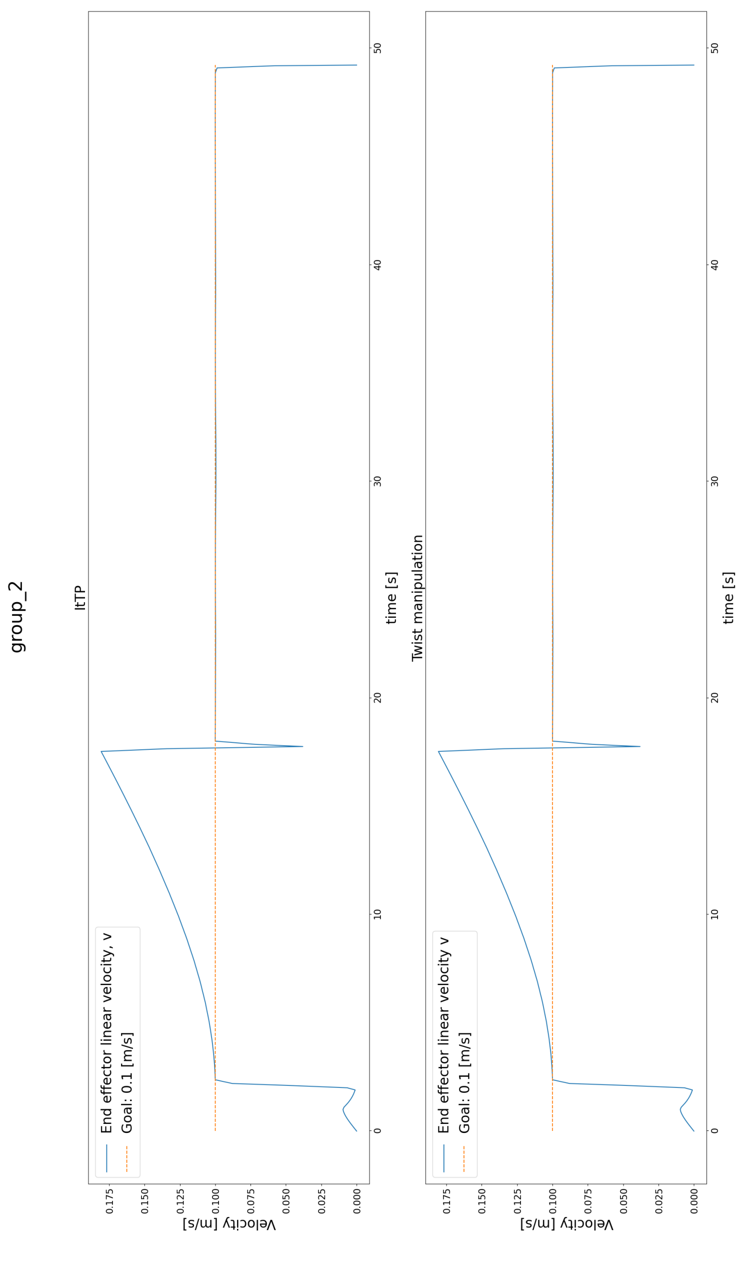


Figure 7.12.: Linear velocity for circular path, Iterative time parameterization (top) and twist (bottom).

7.7. Test 3: The Inside Weld Test

Test 3 demonstrated the system's ability to plan and execute tasks within objects, specifically showcasing its capability to perform welding operations inside objects, provided that the desired goal is reachable. The resulting trajectory is depicted in Figure 7.13, and the velocity profiles of the end effector are shown in Figure 7.14 and 7.15.

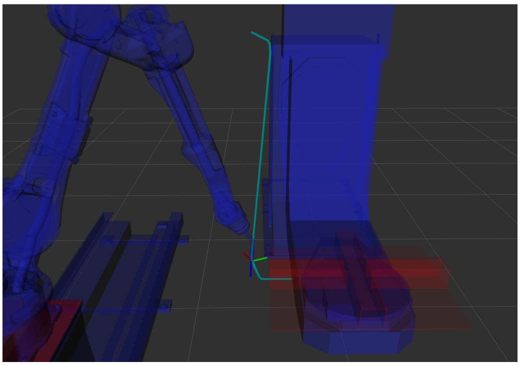


Figure 7.13.: The trail of the end effector representing the trajectory for test 3.

7.8. Test 4: Weld Test

The test demonstrated successful control of the robot and the ease of defining trajectories, as described in chapter 6. The close proximity of the movement to the defined weld edge and the appropriate orientation of the end effector both suggest that the product of this thesis can be effectively utilized for welding tasks. The trajectory trace, depicted in Figure 7.16, provides visual evidence of the robot's path. Furthermore, the intersection of the end effector's z-axis with the weld edge, as shown in Figure 7.17, indicates that the welding process will be successful. The velocity profiles of the end effector for test 4 can be seen in Figure 7.18 and 7.19.

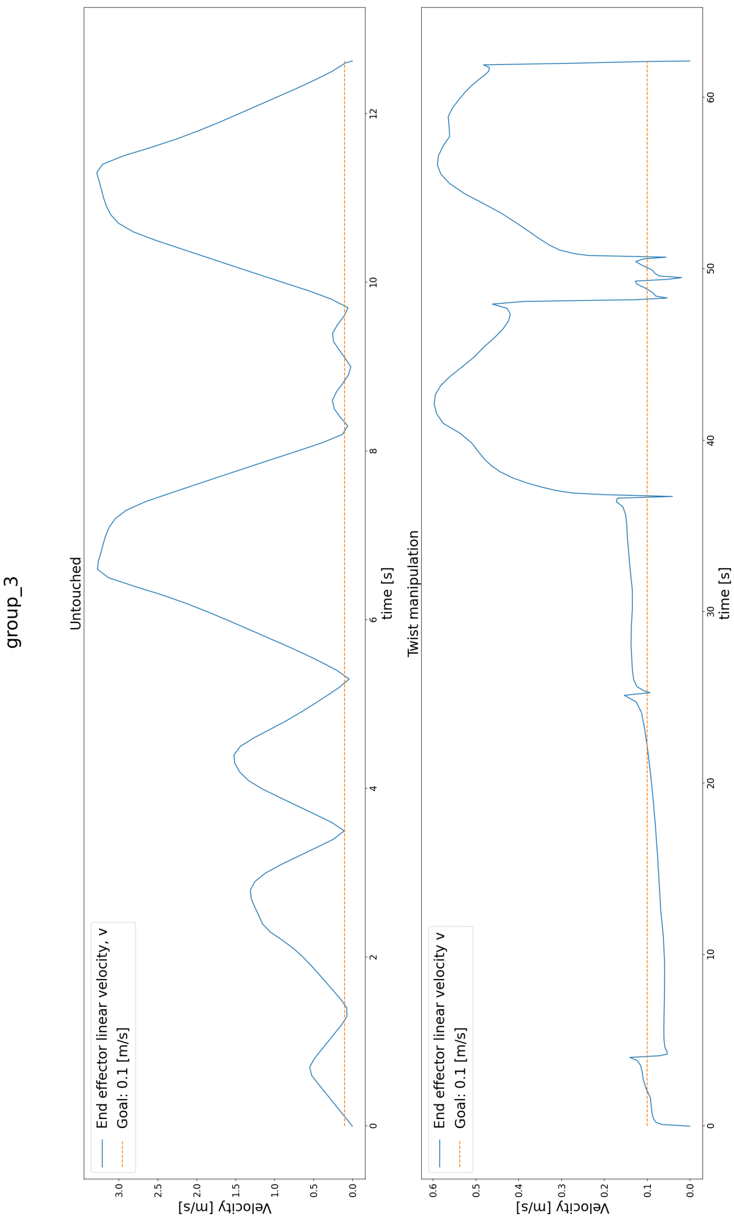


Figure 7.14.: Linear velocity for inside motion. Untouched (top) vs twist method (bottom).

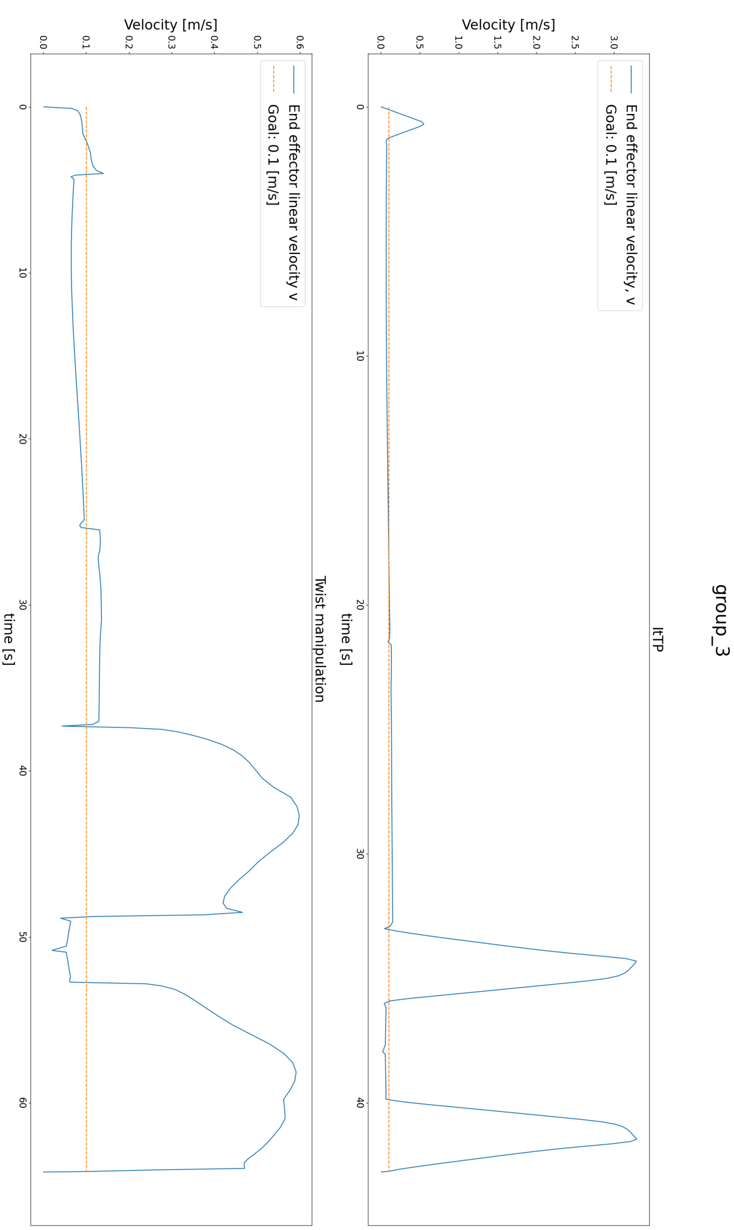
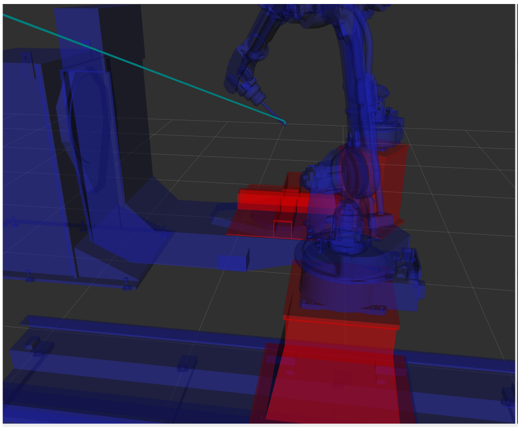
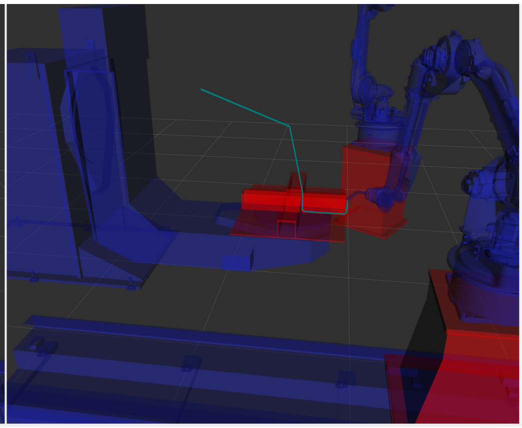


Figure 7.15.: Linear velocity for inside motion. Iterative time parameterization (top) vs twist method (bottom).



(a) End effector trail, test 4 part 1.



(b) End effector trail, test 4 part 2.

Figure 7.16.: The trail of the end effector representing the trajectory for test 4.

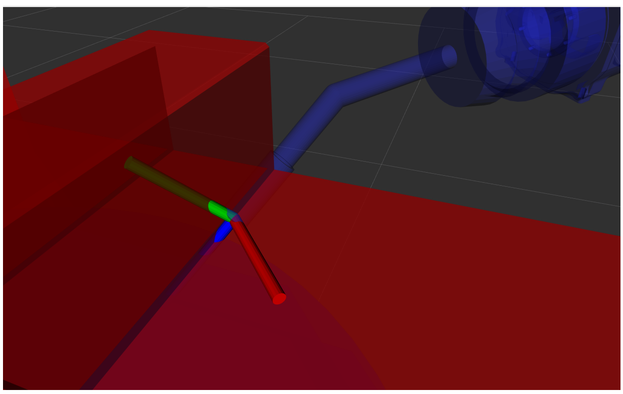


Figure 7.17.: The end effector can be traced through the edge defined as the weld.

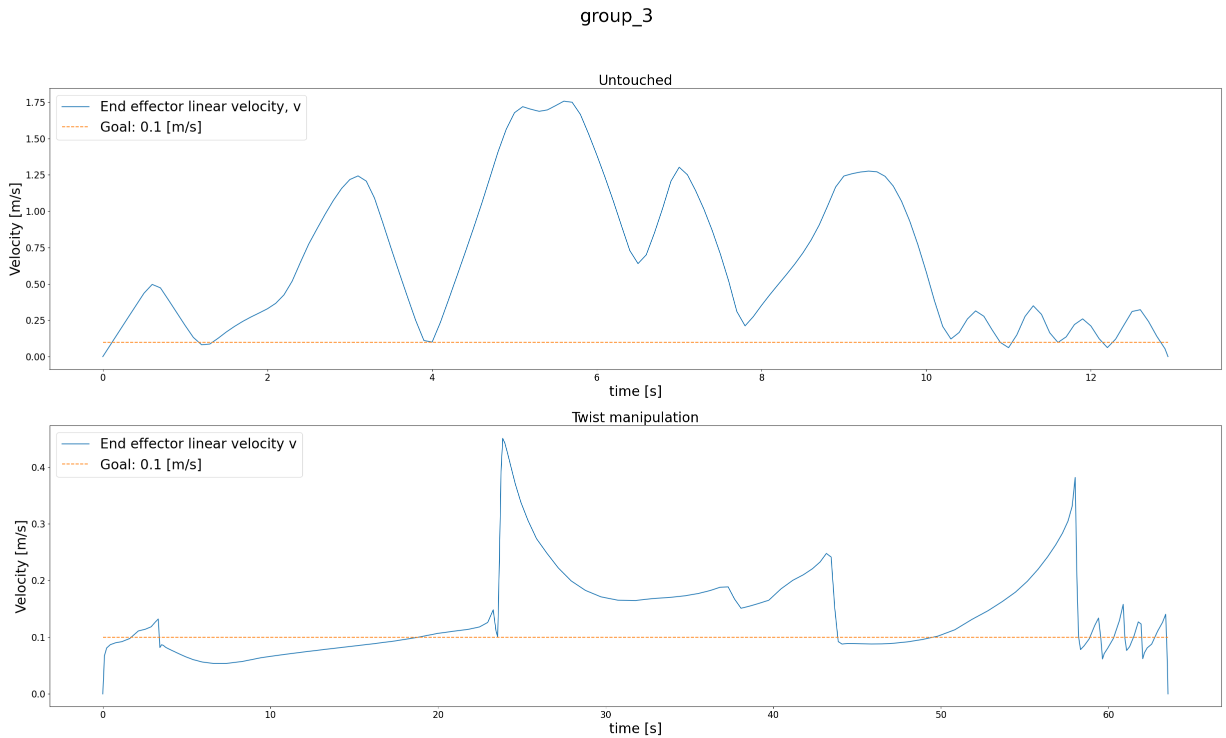


Figure 7.18.: Linear velocity for test 4. Untouched (top) vs twist manipulation (bottom).

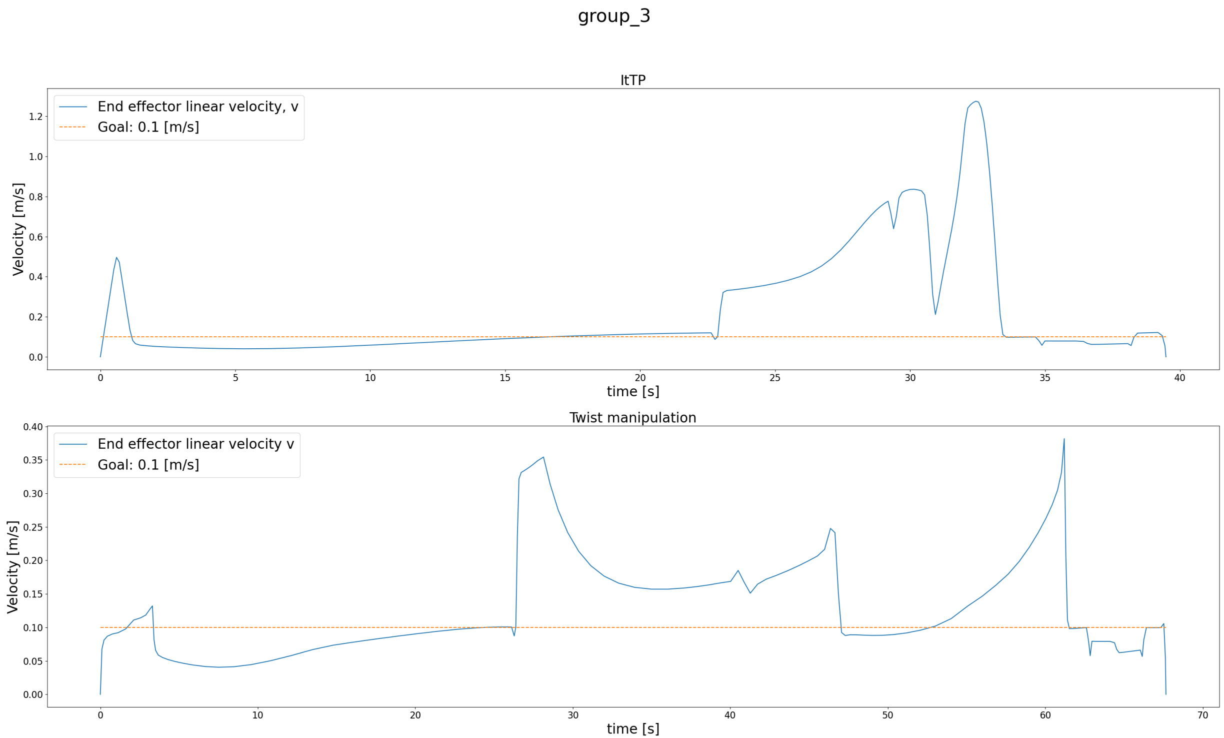


Figure 7.19.: Linear velocity for test 4. Iterative time parameterization (top) vs twist method (bottom).

Chapter 8.

Discussion

8.1. Virtual Environment

The virtual environment depicted in Figure 7.2 employed Rviz2 to visualize the URDF and the virtual robot system. This environment successfully replicated the real-life system, facilitating efficient trajectory planning with the assistance of the MoveIt 2 framework. Although the virtual environment did not encompass every component of the complete robot cell, such as the YRC1000 robot controller, wireless router, welding apparatus, or the surrounding walls, it was a valuable tool for trajectory planning.

The utilization of Rviz2 for this purpose provided extensive customization and flexibility. It offered a wide range of features and options, allowing for comprehensive visualization of the robot system, including its links, joints, and sensors. The ability to customize the display and configure various visual elements within Rviz2 made it easy to verify the trajectory. Additionally, the MoveIt2 module allowed for the positioning of every joint in the system, enabling the control of the robot by graphically setting its pose in the real world.

It is evident that the virtual environment closely approximates the geometry of the real-world system, although it is not an exact replica. The calibration method outlined in section 4.3 proved effective overall, but achieving the exact center of the point of reference (the center hole of the MT1 rotation platform) posed challenges. Nevertheless, no collisions occurred during the development of the planning interface, indicating that the error remains within the millimeter scale.

8.2. The Tests

The conducted tests aimed to evaluate the system's performance in various scenarios, encompassing different types of geometric paths and welding tasks. Test 1 focused on straight lines, while Test 2 involved curves and arcs. Test 3 examined movements close to and inside workpieces, while Test 4 specifically assessed the system's welding capability. Additionally, the tests included objectives related to trajectory smoothness, waypoint detection, and appropriate data publication.

Overall, the tests yielded successful outcomes. The motion planners provided by OMPL generated visually appealing trajectories that precisely followed the desired paths. The trajectory representations can be observed in Figure 7.4, 7.8, 7.13, and 7.16, for test 1, test 2, test 3 and test 4, respectively. Furthermore, videos of robots executing the tests are included in the digital appendix.

Test 4 was primarily designed to simulate an actual welding operation. However, the welding task could not be performed due to the unavailability of shielding gas. Nevertheless, the test was successfully conducted in a "dry" manner, and the results were positive. In Figure 7.17, the approach of the end effector is depicted, demonstrating the precise alignment of the welding torch with the defined weld edge. This suggests that the pieces would be effectively joined together if welding were to take place. However, it is essential to note that without actual welding, the quality of the weld cannot be assessed. Additionally, extracting coordinates from the CAD file validated the effectiveness of utilizing Cartesian coordinates for planning and defining welding and movement areas.

One notable observation is that describing the desired orientation can be challenging and non-intuitive. Although achieving every orientation is generally possible, determining the optimal orientation may require a system that can decide based on factors beyond simple rotation around the world frame. Exploring alternative approaches for orientation calculation could enhance the system's performance in this regard.

8.3. The velocity Limiter

The time the robot arms take to complete the trajectories is a reliable indicator of whether the end effector velocities were appropriately limited. In Table 7.1, the distance covered, estimated time for completion, and actual time used by the arm in the tests are presented. A consistent observation across the tests is that the estimated times and the actual times were similar, with the actual times being slightly slower than the estimated times. This trend is particularly noticeable in Test 3 and Test 4.

The difference between the estimated and actual times can be attributed to two main factors. Firstly, the estimated time does not account for acceleration and deceleration, resulting in a shorter estimated time compared to the actual time taken. As the robot arm starts and stops its motion, the time required for acceleration and deceleration adds to the overall completion time. Secondly, when the end effector undergoes orientation changes while maintaining its position in space, it momentarily pauses during the orientation transition. This pause in motion contributes to a slightly longer actual time compared to the estimated time. In Test 1 and Test 2, where the end effector changes orientation only once, the impact on the overall time is relatively minor. However, in Test 3 and Test 4, where the orientation changes occur multiple times, the cumulative effect is more pronounced.

The similarity between the estimated and actual times implies that the algorithm successfully controlled the robot's velocity to match the desired speed. Despite the minor deviations caused by acceleration, deceleration, and orientation changes, the results demonstrate that the implementation achieved its intended purpose of constraining the end effector velocities within the desired range. This is further evident when inspecting the velocity plots. The four tests yielded similar results, demonstrating a consistent behavior of the end effector's travel speed when using the methods described. In Figure 7.5, 7.11, 7.14, and 7.18, the bottom graphs depict the linear velocities of the end effector obtained from the twist manipulation method applied to the reference trajectory. The top plots depict the reference trajectory of the top plots. In Figure 7.6, 7.12, 7.15, and 7.19, the top plots depict the same test with the iterative time parametrization approach. The bottom is the twist manipulation approach when applied to the iterative time parametrization method.

A notable finding is that the twist manipulation method performed less favorably compared to the results obtained from iterative time parametrization. This difference could be attributed to the fact that the twist manipulation only considers values in the current instance without considering previous and subsequent values. Consequently, the joint values may not be consistent, and compensatory actions may be required if some joints have low velocities while others require compensation. In contrast, the iterative time parametrization approach employs parabolic splining, ensuring consistent velocities with smooth transitions.

Furthermore, an important observation is that all the velocity plots exhibit peaks that exceed the desired velocity limits. This can be explained by considering Equation 4.3, where the end effector's linear velocity component v includes the influence of the end effector's change in orientation $\dot{R}$. This is particularly evident in Figure 7.12, where the end effector undergoes a 90-degree rotation, resulting in the convex portion of the plot. Once the rotation is completed, the end effector's linear velocity remains constant.

In Figure 7.6, 7.12, 7.15, and 7.19, the twist manipulation method is employed on top of the iterative time parameterization method. For this reason, the plots are expected to be the same, as the end effector linear velocity is already limited by the iterative time parameterization method. However, this is not the case, as seen in Figure 7.15 and 7.19. It is evident that one of the methods has been implemented incorrectly. By inspecting the time it is calculated to complete the trajectory in Figure 7.14, it is clear that the twist manipulation does not work as expected, as there is a significant mismatch between the expected time and the time used in the trajectory.

8.4. The Controlling Interface

The proposed solution demonstrated seamless path modification capabilities. If the geometry was known, it was straightforward to define the desired trajectory in Cartesian coordinates and intuitively adjust parameters such as speed and the controlled group. However, the accuracy of the trajectory relies on the planner and the provided waypoints to the ROS node. Cartesian path planners prioritize straight paths between waypoints to minimize the overall distance in cases where the waypoints are sparse. Consequently, the end effector may deviate from the desired path, particularly in arcs, if the resolution is not high enough, as depicted in Figure 8.1. This deviation is illustrated in Figure 7.9, where a circle is represented with only $n = 10$ points. To mitigate this error, increasing the number of waypoints for curves is recommended. An additional benefit of using a higher waypoint resolution for arcs is that the resulting trajectory will be shorter. This occurs because robotic planners aim to minimize jerk, resulting in naturally smooth motions when the waypoints are close to each other, as depicted in Figure 7.10.

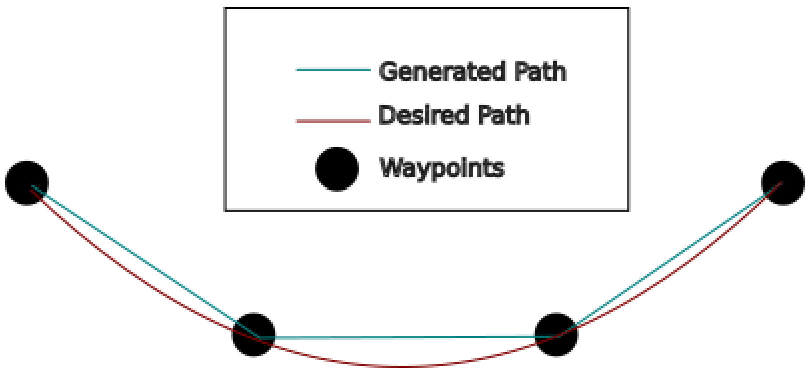


Figure 8.1.: Desired path vs generated path

The joint_states topic provides information about the robot's joint positions and is published by the MotoROS2 node running on the robot controller. Since the robot's joint states change as it moves, it is crucial to publish updates at a sufficiently high frequency to ensure the robot reaches its waypoints when this data is published. Low publishing frequency can cause the robot to surpass the waypoint before the next update, leading the system to perceive that the waypoint was not reached incorrectly. Consequently, the /ready_publisher may not be updated, and the tool may not activate as desired. Figure 8.2 visualizes this phenomenon. During the tests conducted during development, this issue did not occur as the end effector's speed was low, and the publishing frequency was set at the default value of 50 Hz.

Considering the end effector speed used in the tests, which was 10 cm per second, and a refresh rate of 50 Hz, the joint states would be updated for every 2 mm of end effector movement. It is important to note that in practical welding scenarios, where even smaller end effector velocities are typically employed to ensure welding precision, as described in section 2.9, the issue of overshooting waypoints is unlikely to occur.

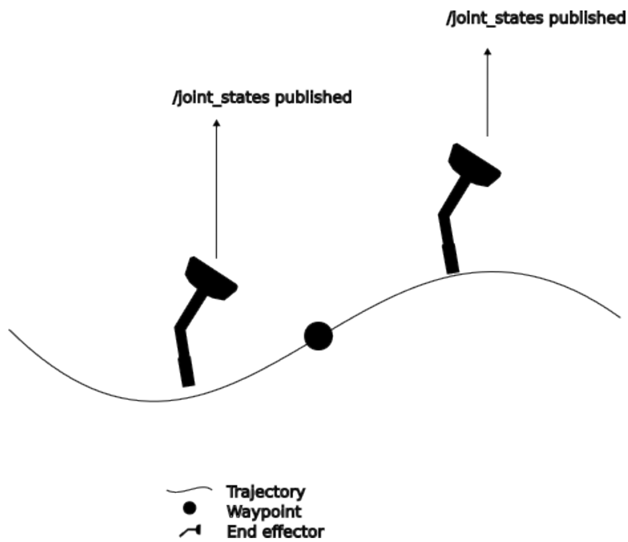


Figure 8.2.: Representation of how the end effector may reach the desired waypoint inbetween publishing joint states, failing to update the kinematics in the node.

The Ready publisher publishes the data from the waypoints message in the "is_job" variable. The data changes when a waypoint is reached and remains unchanged until the next one is reached, as seen in Figure 8.3. This implementation works for welding along edges, where the end effector is constantly on and has a constant effect. However, this approach may not be beneficial for all tasks. It should be considered to publish more information, such as the actual forward kinematics or the number of waypoints reached. Publishing such data improves the system's potential and transparency.

An issue with the current implementation that often occurred was the flawed initialization of the robot in space, resulting in the robot's position not being set correctly within the program developed in this thesis. As a result, the end effector never 'reached' the waypoint within the planner_node, leading to incorrect data being published.

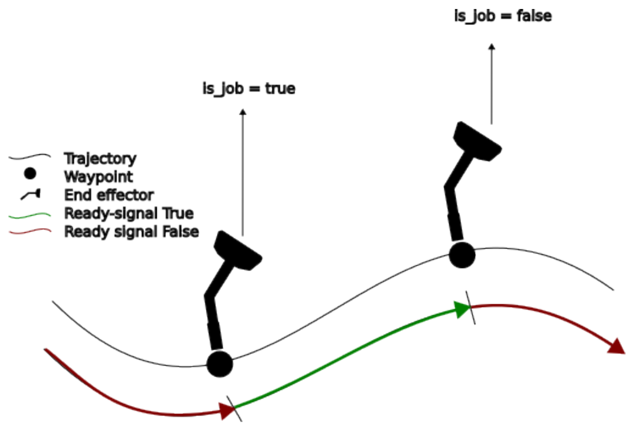


Figure 8.3.: Visualization of the ready signal. Here, the signal is TRUE from waypoint X (left) untill updated at waypoint Y (right) which is FALSE.

8.5. The Planning Implementation

The planning process involved addressing specific challenges related to controlling the robot system using programming and planning a trajectory for the robot. Initially, planning for the whole system was not feasible due to the inability of the MoveIt 2 framework to automatically identify the end effectors. This limitation prevented the creation of a plan based on the desired end goal of the end effector, as the planning interface could not locate the end effector link and thus not knowing the kinematics to solve for. The reason for this was unknown, as the tree defining the robot structure seen in Figure 8.4, clearly shows that end effectors were endpoints in the tree.

To address this challenge, individual planning groups were established, allowing the planner to locate the end effector links and generate Cartesian plans accordingly. However, this approach necessitated extensive manipulation of the planned trajectory due to the robot controller's requirement for data values from all joints within the robot system, even though the planned trajectory focused on a single arm. Despite this complexity, the resulting implementation successfully achieved the desired task.

In order to achieve the desired goal of this thesis, the utilization of Cartesian plan-

Figure 8.4.: The structure of the URDF.

ners proved to be crucial. Planning for a specific set of joint values often resulted in random trajectories. It lacked coherence when attempting to reach the desired goal, as illustrated in Figure 8.5 and 8.6. Such trajectories are unsuitable for tasks requiring precise and stable movements, such as welding. Cartesian planners, however, generate trajectories with coherent points relative to each other, providing trajectories that fulfill these requirements.

A critical limitation of the current implementation is that only groups with at least 6 degrees of freedom can be planned for, as the goal is a 6-degree-of-freedom pose. As a result, even though four groups are present in the system utilized in this study, only 2 (and effectively 3, since one is a subgroup of the other) can be planned for with the current implementation of the planner_node. This is because the planner requires a pose or transformation as input, consisting of a position vector and an orientation quaternion. For instance, group 4 in the system can only rotate and tilt, but not pitch and translation, rendering it incompatible with any Cartesian planner. A potential solution to this limitation could be to check whether the input specifies only an orientation or a position plus orientation. Depending on the input, a different planner can be utilized to enable planning for groups with less than 6 degrees of freedom.

Another significant limitation pertains to collision avoidance. While Cartesian planners incorporate collision detection, they do not inherently provide collision avoidance capabilities. As a result, if the initial waypoint requires the robot arm to move through the workpiece, the generated trajectory will be a straight line toward that point until a collision is detected. At that point, the trajectory will abruptly terminate, resulting in a fraction of the desired path being executed. The MoveIt 2 framework does offer collision avoidance methods, but these methods are exclusively designed for goal-focused planners. In other words, they prioritize reaching the start and end goals while disregarding the arm's orientation and position between them.

In the current implementation, it was essential to manually consider collision avoidance by carefully selecting waypoints that enabled the Cartesian planner to navigate without encountering collisions. This requirement became particularly evident in test 3, where the end effector had to move inside the workpiece, necessitating the definition of additional points, making the programming of the trajectory somewhat more complex. By utilizing target goal planners for the initial waypoint, collision avoidance could be integrated, simplifying the planning process. The remaining trajectory portions could then be planned using the Cartesian planner. Addressing collision avoidance more automatically would significantly enhance the system's usability and flexibility. This can also solve the limitation of low-degree-of-freedom planning mentioned above.

Lastly, it is vital to address the limitation related to the robot arm's initial position relative to the desired start of the job task. In the current implementation, if the robot arm is far from the intended starting point, the velocity of the entire trajectory is limited. This leads to an unnecessarily long trajectory since the movement to reach the start of the path does not require velocity limitation.

To optimize the system further, it would be beneficial to incorporate a more intelligent approach that dynamically adjusts the velocity limitations based on the specific segment of the trajectory. Overall efficiency and time optimization can be achieved by allowing higher velocities for movements that do not involve reaching the start point.

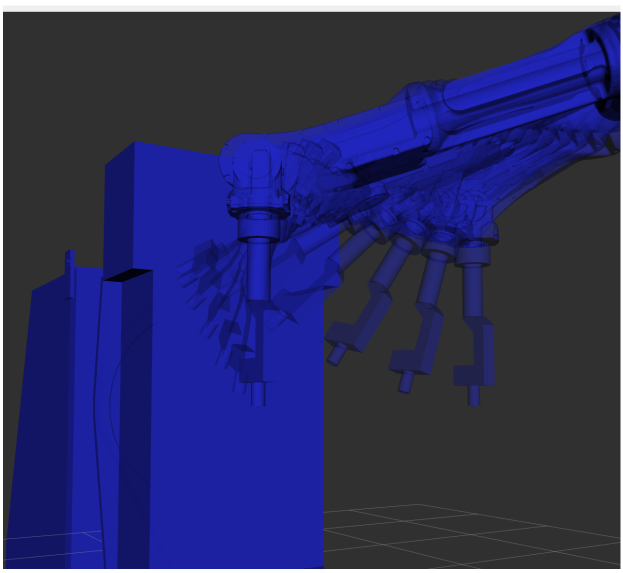


Figure 8.5.: The figure shows the resulting trajectory for a small Cartesian change when planning in joint space.

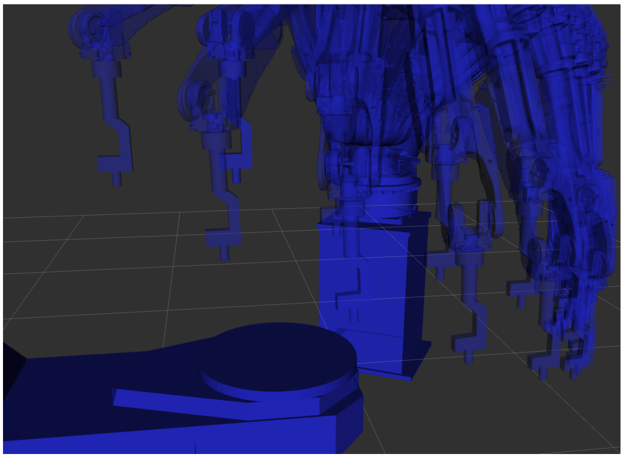


Figure 8.6.: The figure shows the resulting trajectory for a small Cartesian change when planning in joint space with constraints.

8.6. MotoROS2

To establish communication with the robot controller, the MotoROS2 application was employed. This choice facilitated successful communication, and leveraging ROS2 on the controller made it relatively straightforward to develop a robot movement and control program. The utilization of MotoROS2 has thus proven to be highly valuable.

Throughout this thesis, a vision was to implement a camera sensor to detect objects, edges, and changes within the system. For this reason, it was desirable to implement a way to tell the robot to move to a Cartesian coordinate. The main reason for this is that utilizing computer vision and camera sensors provides a transformation from the camera to the pixel evaluated. As this transformation includes a Cartesian coordinate, the transformation can be easily used to tell the robot to move to the point corresponding to the pixel or by following the edge found by an edge-finding algorithm.

The proposed method did facilitate the goal mentioned above. However, it utilized the action "follow joint trajectory". This action required a preplanned trajectory and thus did not offer on-the-go changes in the trajectory, and further explorations of the MotoROS2 application should be considered for alternative, real-time control.

Several challenges were encountered while utilizing MotoROS2 within the setup of this project. One significant issue was the occasional instability in communication. The micro-ROS client experienced frequent dropouts, necessitating the need for reconnection. Consequently, the services and publishers provided by MotoROS2 became temporarily unavailable, leading to the sudden unavailability of certain functionalities. Moreover, this reconnection process often resulted in the emergence of the "TF_OLD_DATA" error, inundating the terminal with error messages. These dropouts may have been caused by a weak wireless network connection, resulting in intermittent communication disruptions. Further investigation and mitigation strategies are necessary to address these connectivity issues effectively.

During the experimentation, a notable limitation was identified regarding the memory capacity of the robot controller and the inadequate support for long trajectories within the MotoROS2 application. When attempting to execute complex paths comprising numerous waypoints, the application would freeze, necessitating a hard reset. Although this limitation was anticipated as a known constraint of the application, it manifested at considerably shorter trajectory lengths. One instance occurred on a trajectory with 166 points, as opposed to the expected 200. This deviation is likely attributed to the inclusion of three additional joints in the system, further exacerbating the memory constraints.

Another limitation observed is the requirement for the robot controller to be connected to Ethernet after installing MotoROS2. Failure to meet this requirement results in an error warning message being displayed, as depicted in Figure 8.7. The warning continues to appear despite attempting to reset the system, disabling most operations, such as jogging the robot. This issue can be resolved by simply connecting the Ethernet cable to establish the required connection.

8.7. Further Limitations

The arm moves continuously between each waypoint, following its defined implementation. Some tasks, like tack welding, require the end effector to stay in one place for a specific duration, which is not feasible when continuously moving. To solve this issue, it is possible to define a waypoint with the exact coordinates but a different orientation. The end effector will move to that spot, stop, change its orientation, and resume motion. However, this method does not allow for controlling the duration of the halt, which can be problematic in some instances. A better approach would be to move the end effector to the designated waypoint, perform the required task, and then calculate a new trajectory based on the updated position of the end effector.

Although the waypoints defined in this thesis were considered representative of the welding path in the case of robotic welding, it is essential to acknowledge certain limitations in the implemented approach of the main package. The current implementation addresses the kinematics required for the end effector to reach the

Figure 8.7.: If no Ethernet connection is detected after MotoROS2 is installed on the controller, a warning will show.

specified waypoints. However, there are potential challenges that may arise. For instance, when dealing with edges in corners, the diameter of the welding gun can prevent reaching the desired waypoints. Additionally, it is necessary to maintain a distance between the welding gun and the arc, further complicating waypoint attainment. In the test case (test 4), a solution was applied by incorporating an offset to account for these challenges. However, this solution requires users to calculate the adjusted points themselves manually. Considering the potential benefits, it could be advantageous to integrate this offset calculation method into the main package.

Chapter 9.

Conclusion and Further Works

9.1. Conclusion

In conclusion, this thesis successfully enabled ROS 2 control for one of the robot cells in Manulab at NTNU and facilitates the integration of advanced sensors like cameras. By configuring the robot controller to communicate with external computers and utilizing the functionalities of ROS 2 through the MotoROS2 application, effective network-based control of the robot was achieved using a laptop running ROS 2.

The proposed planning application and the implementation of end effector velocity control provided an intuitive means for controlling and programming trajectories while ensuring the robot arm's velocity remained within desired limits.

The effectiveness of the implemented solution was evaluated through a series of comprehensive tests, including real-world scenarios in realistic environments. Although limitations with materials prevented actual welding, the system's validation focused on assessing the visual appearance of the robot arm's movements.

The tests' results validated the proposed solution's capabilities and demonstrated its potential for real-world applications. However, to further enhance the system, future work should focus on conducting experiments involving actual welding processes and evaluating the system's performance in a broader range of industrial scenarios.

9.2. Further Works

The primary objective of this thesis is to present a particular concept and enable the system for ROS 2. However, to augment its capability and practicality, it is imperative to fine-tune and amplify specific elements of the thesis.

- Implementing a camera and utilizing its sensor capabilities for edge detection could be valuable for applications such as welding, and such functionality was the main motivator of this thesis. By detecting and recognizing edges, the system could automatically identify and define weld areas, streamlining the planning and execution of welding operations. Furthermore, this camera can provide pose estimation, which would further increase the capability of the system to be more dynamic.
- Finding a more intuitive way to describe the orientation of the end effector would simplify the process of specifying desired orientations for trajectory planning. This could involve exploring alternative representations or methods that make it easier for users to define and understand the desired orientation of the robot's end effector.
- Implement a solution to use a planner that allows for automatic object avoidance to reach the trajectory's desired start. This will make it easier to implement solutions for automatic welding jobs, as it would no longer require manual avoidance.
- Implement low degree of freedom planners. The current solution uses pose planning, requiring at least 6 DOF for solving. This can be solved, as well as the point mentioned above, by planning to achieve a target joint goal, instead of the Cartesian pose used.
- Allowing for planning for multiple groups simultaneously would enhance the solution's versatility and enable coordinated movements of multiple robot groups within a system. This would facilitate more complex tasks and improve overall system efficiency.

References

[1] Kaiwalya Belsare, Antonio Cuadros Rodriguez, Pablo Garrido Sánchez, Juanjo Hierro, Tomasz Kołcon, Ralph Lange, Ingo Lütkebohle, Alexandre Malki, Jaime Martin Losa, Francisco Melendez, Maria Merlan Rodriguez, Arne Nordmann, Jan Staschulat, and Julian von Mendel. "Micro-ROS". In: *Robot Operating System (ROS): The Complete Reference (Volume 7)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2023, pp. 3–55. ISBN: 978-3-031-09062-2. DOI: 10.1007/978-3-031-09062-2_2. URL: https://doi.org/10.1007/978-3-031-09062-2_2.

[2] Marie Bodet. *Robotic Welding with Machine Vision Explained*. Zivid. Sept. 2022. URL: https://blog.zivid.com/robotic-welding-with-machine-vision-explained (visited on 06/03/2023).

[3] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam. URL: http://www.blender.org (visited on 06/08/2023).

[4] Aras Dargazany. "DRL: Deep Reinforcement Learning for Intelligent Robot Control - Concept, Literature, and Future". In: *CoRR* abs/2105.13806 (2021). arXiv: 2105.13806. URL: https://arxiv.org/abs/2105.13806.

[5] James Diebel et al. "Representing attitude: Euler angles, unit quaternions, and rotation vectors". In: *Matrix* 58.15-16 (2006), pp. 1–35.

[6] Ayssam Elkady and Tarek Sobh. "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography". In: *Journal of Robotics* 2012 (May 2012). DOI: 10.1155/2012/959013.

[7] Gazebo. *Open Source Robotics Foundation*. http://gazebosim.org/. (Visited on 04/25/2023).

[8] Stuart Glaser, William Woodall, and Robert Haschke. *xacro*. URL: http://wiki.ros.org/xacro#Macros (visited on 06/01/2023).

[9] Yaskawa Europe GmbH. *Yaskawa GP25-12 Robot*. URL: https://www.yaskawa.eu.com/products/robots/handling-mounting/productdetail/product/gp25-12_698 (visited on 06/03/2023).

[10] Ruchi Goel and Pooja Gupta. "Robotics and Industry 4.0". In: *A Roadmap to Industry 4.0: Smart Production, Sharp Business and Sustainable Development*. Ed. by Anand Nayyar and Akshi Kumar. Cham: Springer International Publishing, 2020, pp. 157–169. ISBN: 978-3-030-14544-6. DOI: 10.1007/978-3-030-14544-6_9. URL: https://doi.org/10.1007/978-3-030-14544-6_9.

[11] Thea Holmedal. *Implementing robotic offline programming with the Yaskawa Motoman GP25-12*. Master's thesis, Norwegian University of Science and Technology. 2021. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2788486.

[12] IBM. *Industry 4.0*. https://www.ibm.com/topics/industry-4-0. 2023. (Visited on 05/26/2023).

[13] Universal Technical Institute. *Aluminum Welding: Tips and Techniques for Success*. Universal Technical Institute. 2023. URL: https://www.uti.edu/blog/welding/aluminum-welding (visited on 06/05/2023).

[14] P. Kah, M. Shrestha, E. Hiltunen, and J. Martikainen. "Robotic arc welding sensors and programming in industrial applications". In: *International Journal of Mechanical and Materials Engineering* 10.1 (2015), p. 13. ISSN: 2198-2791. DOI: 10.1186/s40712-015-0042-y. URL: https://doi.org/10.1186/s40712-015-0042-y.

[15] Danica Kragic, Joakim Gustafson, Hakan Karaoguz, Patric Jensfelt, and Robert Krug. "Interactive, Collaborative Robots: Challenges and Opportunities". In: July 2018, pp. 18–25. DOI: 10.24963/ijcai.2018/3.

[16] KUKA. *KUKA Simulation, Planning & Optimization*. URL: https://www.kuka.com/en-us/products/robotics-systems/software/simulation-planning-optimization/kuka_sim (visited on 05/29/2023).

[17] Steven M. LaValle. "Rapidly-exploring random trees : a new tool for path planning". In: *The annual research report* (1998).

[18] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

[19] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[20] James F. Manji. *Robots Facilitate High-speed Welding*. URL: https://www.automate.org/industry-insights/robots-facilitate-high-speed-welding (visited on 06/08/2023).

[21] ManpowerGroup. *Her er de ti jobbene norske bedrifter sliter mest med å fylle i Norge nå*. 2018. URL: https://www.dn.no/arbeidsliv/manpowergroup/her-er-de-ti-jobbene-norske-bedrifter-sliter-mest-med-a-fylle-i-norge-na/2-1-366384 (visited on 04/04/2023).

[22] Motoman. *Motoman Simulation*. URL: https://www.motoman.com/en-us/products/software/simulation (visited on 05/29/2023).

[23] Motoman. *MotoPlus SDK*. URL: https://www.motoman.com/en-us/products/software/development/motoplus-sdk (visited on 06/06/2023).

[24] NVIDIA. *Isaac Sim*. URL: https://developer.nvidia.com/isaac-sim (visited on 06/07/2023).

[25] PPM-Robotics-AS. *PPM-Robotics-AS*. https://github.com/PPM-Robotics-AS. (Visited on 06/07/2023).

[26] Roboplan. *MT1 Positioner*. URL: https://www.roboplan.pt/en/products/positioners/mt1 (visited on 05/30/2023).

[27] Roboplan. *Yaskawa Motoman Linear Track TSL-TSL*. URL: https://www.roboplan.pt/en/products/positioners/yaskawa-motoman-linear-track-tsl-tsl (visited on 05/30/2023).

[28] Open Robotics. *ROS 2 Concepts - About ROS Interfaces*. 2023. URL: https://docs.ros.org/en/foxy/Concepts/About-ROS-Interfaces.html (visited on 03/03/2023).

[29] Open Robotics. *ROS Wiki: Nodes*. URL: http://wiki.ros.org/Nodes (visited on 05/26/2023).

[30] Open Robotics. *Understanding ROS 2 Topics*. URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html (visited on 05/29/2023).

[31] Path Robotics. *Path Robotics*. https://www.path-robotics.com/. (Visited on 04/14/2023).

[32] PickNik Robotics. *Getting started*. 2023. URL: https://moveit.picknik.ai/humble/doc/tutorials/getting_started/getting_started.html (visited on 04/04/2023).

[33] PickNik Robotics. *MoveIt! by PickNik Robotics*. 2023. URL: https://moveit.picknik.ai/humble/index.html (visited on 04/21/2023).

[34] ROS-Industrial. *ROS-Industrial/motoman*. 2023. URL: https://github.com/ros-industrial/motoman (visited on 05/30/2023).

[35] ROS-Planning. *MoveIt! Tutorials: Time Parameterization*. 2021. URL: https://ros-planning.github.io/moveit_tutorials/doc/time_parameterization/time_parameterization_tutorial.html (visited on 04/21/2023).

[36] Balkeshwar Singh, N Sellappan, and P Kumaradhas. “Evolution of industrial robots and their applications”. In: *International Journal of emerging technology and advanced engineering* 3.5 (2013), pp. 763–768.

[37] *SolidWorks*. URL: https://www.solidworks.com/ (visited on 06/08/2023).

[38] Thieso. *trajectory_processing, MoveIt!* GitHub repository. Cartesian end effector speed implementation. 2021. URL: https://github.com/Thieso/moveit/tree/cartesian_ee_speed/moveit_core/trajectory_processing (visited on 04/21/2023).

[39] Dirk Thomas. *colcon - collective construction*. 2018. URL: https://colcon.readthedocs.io/en/released/ (visited on 06/05/2023).

[40] Klas Weman and Gunnar Lindén. *MIG welding guide*. Woodhead Publishing, 2006.

[41] *Workforce Development*. American Welding Society Foundation. URL: https://www.aws.org/foundation/page/workforce-development (visited on 04/04/2023).

[42] www.robot-welding.com. *Welding torch*. 2001. URL: http://www.robot-welding.com/welding_torch.htm (visited on 06/07/2023).

[43] www.robot-welding.com. *Wire Feeder*. 2001. URL: http://www.robot-welding.com/wire_feeder.htm (visited on 06/08/2023).

[44] Yaskawa. *Overview Functions Packages YRC1000*. Yaskawa Europe GmbH. Aug. 2019. URL: https://www.yaskawa.eu.com/Global%20Assets/Downloads/Brochures_Catalogues/Robotics/software/Overview_Functions_Packages_YRC1000_E_08.2019.pdf.

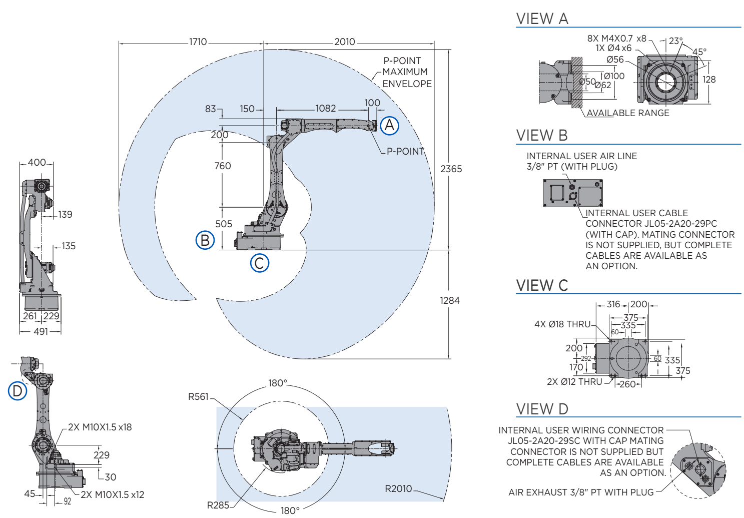
[45] Yaskawa Global. *MotoROS2: Yaskawa-Global/motoros2*. URL: https://github.com/Yaskawa-Global/motoros2 (visited on 05/29/2023).

[46] Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, and Yifeng Zhu. “robosuite: A Modular Simulation Framework and Benchmark for Robot Learning”. In: *arXiv preprint arXiv:2009.12293*. 2020.

Appendix A.

Yaskawa Motoman GP25-12 Datasheet

GP25-12 ROBOT



All dimensions are metric (mm) and for reference only.
Request detailed drawings for all design/engineering requirements.

SPECIFICATIONS

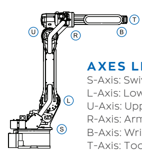
Axes	Maximum motion range	Maximum speed	Allowable moment	Allowable moment of inertia
	degrees	°/sec	N•m	kg•m²
S	±180	210	–	–
L	+155/-105	210	–	–
U	+160/-86	220	–	–
R	±200	435	22	0.65
B	±150	435	22	0.65
T	±455	700	9.8	0.17

Specifications for GP25-12 with XP package may be different.
Mounting Options: Floor, Wall, Tilt or Ceiling
* The MLX300 software option is not available for use with arc or spot welding applications. MLX300 fieldbus cards, I/O cards and vision equipment must be purchased separately from the supplier. All peripherals are programmed using a PLC.

Item	Unit	GP25-12
Controlled axes		6
Maximum payload	kg	12
Repeatability	mm	0.03
Horizontal reach	mm	2,010
Vertical reach	mm	3,649
Weight	kg	260
Internal user I/O cable		17 conductors w/ ground
Internal user air line		(1) 3/8" connection
Power requirements		380-480 VAC
Power rating	kVA	2.0

OPTIONS

- Robot risers and base plates
- Extended length manipulator cables
- Wide variety of fieldbus cards
- External axes
- PLC integration via MLX300 software option*
- Functional Safety Unit (FSU)
- MotoSight™ 2D and 3D vision



AXES LEGEND
S-Axis: Swivel Base
L-Axis: Lower Arm
U-Axis: Upper Arm
R-Axis: Arm Roll
B-Axis: Wrist Bend
T-Axis: Tool Flange

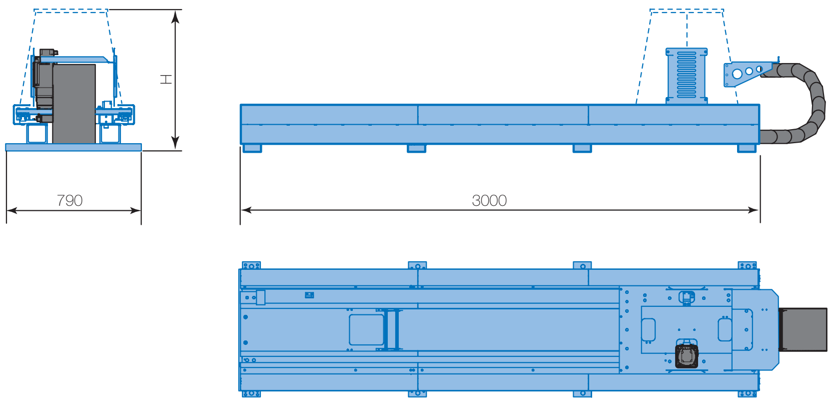
YASKAWA
YASKAWA AMERICA, INC. 100 Automation Way | Miamisburg, OH 45342
937.847.6200 | motoman.com

Technical specifications subject to change without notice. | Motoman is a registered trademark.
All other marks are the trademarks and registered trademarks of Yaskawa America, Inc.
DS-874 ©2021 Yaskawa America, Inc. JANUARY 2021

Appendix B.

Yaskawa Motoman TSL600 Datasheet

TSL-600



All dimensions are for reference only.

Request detailed drawings for design/engineering requirements!

Technical data		TSL-600 SY
Maximum payload		600 kg
Maximum speed		2.14 m/s
ED		50%
Acceleration velocity		2.38 m/s²
Travel	500 mm 1000 mm	0.85 sec 1.16 sec
Repetitive position accuracy		±0.05 mm
Height (H) including robot stand		687, 887, 1087, 1287 or 1487 mm
Standard length (L)		2,3 or 4 meter
Travel length (stroke)		L-850 mm

YASKAWA

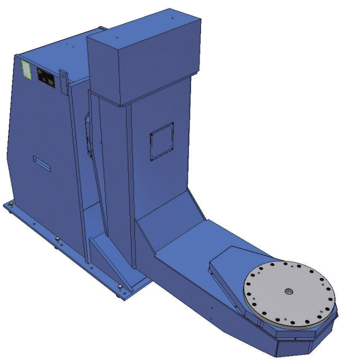
YASKAWA Nordic AB
PO Box 504
SE-385 25 Torsås, SWEDEN

Appendix C.

Yaskawa Motoman MT1 Datasheet

YASKAWA

MT1-1000, MT1-1500



- Rigid design.
- High freedom of positioning.
- Heavy payloads.

This is a one station positioner for workpieces requiring rotation about two axes. Its high freedom of positioning makes it easy to find the optimum position even in complicated workpieces.

This MT1 positioner with its L arm design is able to withstand the strain of handling heavy workpieces.

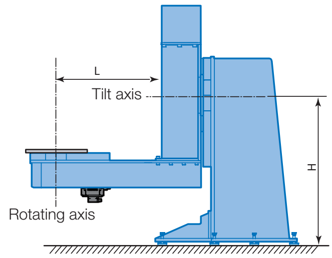
When used together with a Gantry robot it provides the best possible workpiece access.

Technical data		MT1-1000 S2D	MT1-1500 S2D
Max. payload		**1000 kg**	**1500 kg**
Welding capacity	100% duty cycle 60% duty cycle	2x 350 A 2x 460 A	2x 350 A 2x 460 A
Tilt axis torque	dynamic static	4236 Nm 3389 Nm	8290 Nm 6632 Nm
Tilt axis	rated speed maximum speed	0-3.6 rpm 9.6 rpm	0-3.4 rpm 9.1 rpm
Rotating axis torque	dynamic static	2830 Nm 2264 Nm	6480 Nm 5184 Nm
Rotating axis	rated speed maximum speed	0-5.3 rpm 15 rpm	0-4.3 rpm 15 rpm
Rated offset from centre of gravity (COG)		230 mm	352 mm

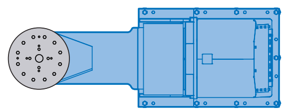
Controlled by Controlled by

MT1-1000, -1500

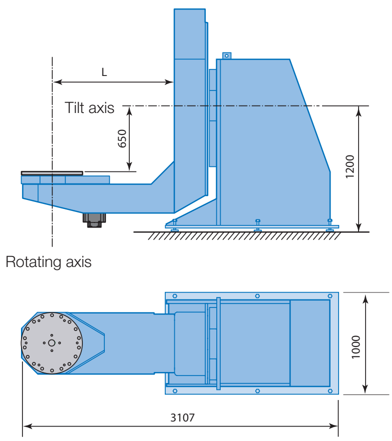
MT1-1000 S2D



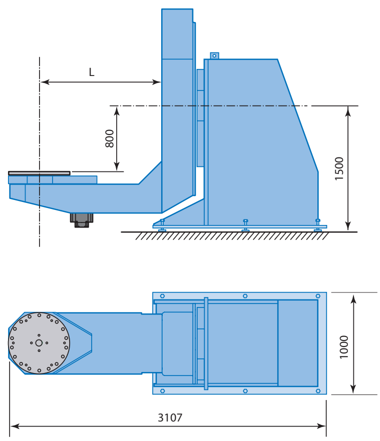
	H (mm)	L (mm)	Weight	Part No.
MT1-1000 S2D	1200	850	1610 kg	124617-102
	1200	1100	1647 kg	124617-103
MT1-1500 S2DL	1200	1195	3484 kg	124618-100
	1500	1615	4010 kg	124619-100



MT1-1500 S2D (H = 1200)



MT1-1500 S2D (H = 1500)



YASKAWA

© YASKAWA Nordic AB · RegNo. 1017EN-03, 2020-09
Technical data may be subject to change without previous notice.

YASKAWA Nordic AB
PO Box 504
SE-385 25 Torsås, SWEDEN

Phone +46 480 41 78 00
info.se@yaskawa.eu.com
www.yaskawa.eu.com

Appendix D.

Description of the Folders Included in the MotoROS2 Package

config - This folder includes a single .yaml file. This file is where the configuration of the motoros2 is being defined by the user.

example_jobs - This folder includes a series of subfolders. The folders were the following:

- sda_dual_arm
- single_arm
- single_arm_with_base_axis
- single_arm_with_ext_axis
- two_arms

Each folder has a IONAME.DAT file and a INIT_ROS.JBI file. The IONAME.DAT file is the same for all folders, while the INIT_ROS.JBI has some different setup. The structure of the INIT_JOB.JBI is seen in Figure D.1. The different folders will set the JOB - INST - GROUP1 and GROUP2 depending if the job is defined for a multiarm/base/station system.

The setup is then followed by the INFORM code:

```
NOP
DOUT OT#(890) OFF
DOUT OT#(889) OFF
TIMER T=0.05
DOUT OT#(889) ON
WAIT OT#(890)=ON
```

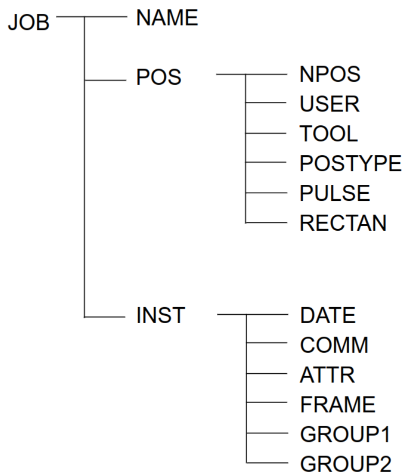


Figure D.1.: The structure of a JBI file. Collected from Yaskawa DX100 INSTRUCTIONS FOR RELATIVE JOB FUNCTION manual

```
DOUT OT#(890) OFF
DOUT OT#(889) OFF
END
```

which is similar for all the folders.

motoros2-beta1-0.0.15 - This folder includes a script which listens for debug messages from motoros2, as well as a README.MD and a CHANGELOG.MD file. The README.MD file describes the installation process and instruction for motoros2 aswell as debugging information and common errors.

motoros2-beta1-main - This folder includes the same files as motoros2-beta1-0.0.15

motoros2__interfaces-beta1-main - This folder is a ros2 package which includes the messages, services and actions used by motoros2.

Appendix E.

Structure of the Robot Model Packages

```
Motoman
    mt1_support
    gp25_12_support
    tsl600_support
```

Each support folder had the following structure:

```
xxx_support
URDF
    - mt1.xacro
    - mt1_macro.xacro
meshes
     visual
        -link_1.stl
        -link_2.stl
        -...
     collision
        -link_1.stl
        -link_2.stl
        -...
launch
config
```

Appendix F.

Moveit Controller Changes

The changes done in , the file moveit_controller.yaml in the moveit_config/config folder were as follows:

From

```
moveit_simple_controller_manager:
  controller_names:
    - group_1_controller
    - group_2_controller
    - group_3_controller
    - group_4_controller
    - follow_joint_trajectory_controller
```

to

```
moveit_simple_controller_manager:
  controller_names:
    - group_1_controller
    - group_2_controller
    - group_3_controller
    - group_4_controller
    - follow_joint_trajectory # <-- changed
```

and from

```
    follow_joint_trajectory:
    type: FollowJointTrajectory
    action_ns: "follow_joint_trajectory"
    default: true
    joints:
      - group_2/joint_1_s
      - group_2/joint_2_l
      - group_2/joint_3_u
      - group_2/joint_4_r
      - group_2/joint_5_b
      - group_2/joint_6_t
```

```
      - group_3/joint_1
      - group_1/joint_1_s
      - group_1/joint_2_l
      - group_1/joint_3_u
      - group_1/joint_4_r
      - group_1/joint_5_b
      - group_1/joint_6_t
      - group_4/joint_1
      - group_4/joint_2
    action_ns: "follow_joint_trajectory"
    default: true
```

to

```
    follow_joint_trajectory:
    type: FollowJointTrajectory
    action_ns: "" # <-- changed
    default: true
    joints:
      - group_2/joint_1_s
      - group_2/joint_2_l
      - group_2/joint_3_u
      - group_2/joint_4_r
      - group_2/joint_5_b
      - group_2/joint_6_t
      - group_3/joint_1
      - group_1/joint_1_s
      - group_1/joint_2_l
      - group_1/joint_3_u
      - group_1/joint_4_r
      - group_1/joint_5_b
      - group_1/joint_6_t
      - group_4/joint_1
      - group_4/joint_2
    action_ns: "" # <-- changed
    default: true
```

Appendix G.

The Launchfile for Planning

planning.launch.py

```
from moveit_configs_utils import MoveItConfigsBuilder
from moveit_configs_utils.launches import generate_move_group_launch
from moveit_configs_utils.launches import generate_rsp_launch
from moveit_configs_utils.launches import
    generate_moveit_rviz_launch
from launch import LaunchDescription

def generate_launch_description():
    moveit_config = MoveItConfigsBuilder("motoman_gp25sys",
    package_name="robco").to_moveit_configs()

    ld = [generate_move_group_launch(moveit_config),
    generate_rsp_launch(moveit_config), generate_moveit_rviz_launch(
    moveit_config)]
    return LaunchDescription(ld)
```

Appendix H.

utilities.h

```cpp

#include <cstdio>
#include <memory>

#include "iostream"

#include <rclcpp/rclcpp.hpp>
#include <moveit/move_group_interface/move_group_interface.h>

#include <moveit_msgs/msg/robot_state.hpp>
#include <moveit_msgs/msg/robot_trajectory.hpp>

#include <sensor_msgs/msg/joint_state.hpp>
#include <trajectory_msgs/msg/joint_trajectory.hpp>




int log(std::string data, const int verbose = 0);
std::vector<std::pair<std::string, std::pair<size_t, size_t>>>
    findNumberOfMatches(const std::vector<std::string>& keywords,
    const std::vector<std::string>& dataset);
int printTrajectory(moveit::planning_interface::MoveGroupInterface::
    Plan plan);
bool compareByIndex(const std::pair<std::string, std::pair<size_t,
    size_t>>& a, const std::pair<std::string, std::pair<size_t,
    size_t>>& b);
bool isGroupInPlan(const std::string& group, const moveit::
    planning_interface::MoveGroupInterface::Plan& plan);
int8_t isGroupInPlans(const std::string& group, const std::vector<
    moveit::planning_interface::MoveGroupInterface::Plan>& plans);
moveit::planning_interface::MoveGroupInterface::Plan
    expandTrajectory(moveit::planning_interface::MoveGroupInterface
    ::Plan plan, size_t lengthOfTrajectory);
moveit::planning_interface::MoveGroupInterface::Plan
```

```
    newPlanFromStartState(moveit::planning_interface::
    MoveGroupInterface::Plan templatePlan, std::string name, size_t
    numberOfJoints, size_t startIndex);
int8_t findIndex(std::vector<std::string> subset, std::vector<std::
    string> set);
sensor_msgs::msg::JointState concatenateStates(const std::vector<
    moveit::planning_interface::MoveGroupInterface::Plan>& plans);
std::vector<moveit::planning_interface::MoveGroupInterface::Plan>
    findPlans(const std::vector<geometry_msgs::msg::Pose>& points,
    const std::string& group, const std::shared_ptr<rclcpp::Node>
    node);
moveit_msgs::msg::JointConstraint createJointConstrain(std::string
    joint_name, double lower_limit, double upper_limit);
moveit_msgs::msg::Constraints createJointConstrains(std::vector<std
    ::string> joint_names, std::vector<double> lower_constrains, std
    ::vector<double> upper_constrains);
std::vector<geometry_msgs::msg::Pose> createStraightPathPoints(std::
    vector<double> xyz_start, std::vector<double> xyz_stop, std::
    vector<double> xyzw_orientation, int num_points);
void addToPlan(moveit::planning_interface::MoveGroupInterface::Plan&
     plan, const moveit::planning_interface::MoveGroupInterface::
    Plan planToAdd, int startidx=0);
geometry_msgs::msg::Pose createPose(double x, double y, double z,
    double ox, double oy, double oz, double ow);
moveit::planning_interface::MoveGroupInterface::Plan
    stupidPlanCreator(const std::string& group, const std::
    shared_ptr<rclcpp::Node> node);
builtin_interfaces::msg::Duration addDurations(builtin_interfaces::
    msg::Duration dur1, builtin_interfaces::msg::Duration dur2);
builtin_interfaces::msg::Duration divideDuration(const
    builtin_interfaces::msg::Duration& d, float value);
```

Appendix I.

utilities.cpp

```
#include "utilities.h"

bool DEBUG = true;

int log(std::string data, const int verbose){
  if (verbose == 0){

    std::cout << data << std::endl;
  }
  else if (verbose > 0 and DEBUG){
    std::cout << data << std::endl;
  }
  return 1;
}

std::vector<std::pair<std::string, std::pair<size_t, size_t>>> findNumberOfMatches(const std::vector<std::string>& keywords, const std::vector<std::string>& dataset) {
  std::vector<std::pair<std::string, std::pair<size_t, size_t>>> results;

  for (const auto& sequence : keywords){
    size_t counter  = 0;
    size_t idx_counter = 0;
    //for alle gruppene
    for (const auto& group : dataset){
      //se om sequence finnes i gruppe, evt hvor mange
      if (group.find(sequence) != std::string::npos){
        counter ++;
      }
      if (counter == 0){
        idx_counter ++; //vi vil bare finne posisjonen til den første matches.
      }
    }
```

```
    results.push_back(std::make_pair(sequence, std::make_pair(counter, idx_counter)));

  }
  return results;
}

int printTrajectory(moveit::planning_interface::MoveGroupInterface::Plan plan){
  for (auto it : plan.trajectory_.joint_trajectory.points){
    for (auto point : it.positions){
      std::cout << point << " ";
    }
    std::cout << std::endl;
  }
  return 1;
}

bool compareByIndex(const std::pair<std::string, std::pair<size_t, size_t>>& a, const std::pair<std::string, std::pair<size_t, size_t>>& b) {
    return a.second.second < b.second.second;
}

bool isGroupInPlan(const std::string& group, const moveit::planning_interface::MoveGroupInterface::Plan& plan){
    if (plan.trajectory_.joint_trajectory.joint_names[0].find(group) != std::string::npos){
      return true;
    }
  return false;
}


int8_t isGroupInPlans(const std::string& group, const std::vector<moveit::planning_interface::MoveGroupInterface::Plan>& plans){
  //returnerer indeksen gruppen finnes i planvektoren, -1 hvis den ikke finnes
  int8_t index = 0;
  for (const auto& plan : plans){
    if (isGroupInPlan(group, plan)){
      return index;
    }
    index ++;
  }
  return -1;
}


moveit::planning_interface::MoveGroupInterface::Plan expandTrajectory(moveit::planning_interface::MoveGroupInterface
```

```
::Plan plan, size_t lengthOfTrajectory){ //exapnds the length of the plan to a given length with the last value in the plan
  //hver plan skal inneholde start_state_

  moveit::planning_interface::MoveGroupInterface::Plan newPlan = plan;

  for(size_t i = plan.trajectory_.joint_trajectory.points.size(); i < lengthOfTrajectory; i++){
    newPlan.trajectory_.joint_trajectory.points.push_back(plan.trajectory_.joint_trajectory.points.back());
  }
  return newPlan;
}
moveit::planning_interface::MoveGroupInterface::Plan newPlanFromStartState(moveit::planning_interface::MoveGroupInterface::Plan templatePlan, std::string name, size_t numberOfJoints, size_t startIndex){ //returns a plan which starts at startstate and has the same "width"

  moveit::planning_interface::MoveGroupInterface::Plan newPlan(templatePlan);

  newPlan.trajectory_.joint_trajectory.joint_names = std::vector<std::string>(templatePlan.start_state_.joint_state.name.begin() + startIndex, templatePlan.start_state_.joint_state.name.begin() + startIndex + numberOfJoints);
  newPlan.trajectory_.joint_trajectory.points[0].positions = std::vector<double>(templatePlan.start_state_.joint_state.position.begin() + startIndex, templatePlan.start_state_.joint_state.position.begin() + startIndex + numberOfJoints);
  //effort er tom slik at dette leder til segfault
  //newPlan.trajectory_.joint_trajectory.points[0].effort = std::vector<double>(templatePlan.start_state_.joint_state.effort.begin() + startIndex, templatePlan.start_state_.joint_state.effort.begin() + startIndex + numberOfJoints);
  newPlan.trajectory_.joint_trajectory.points[0].velocities = std::vector<double>(templatePlan.start_state_.joint_state.velocity.begin() + startIndex, templatePlan.start_state_.joint_state.velocity.begin() + startIndex + numberOfJoints);
  newPlan.trajectory_.joint_trajectory.points[0].time_from_start = templatePlan.trajectory_.joint_trajectory.points[0].time_from_start; //dette vil finnes første punkt [0]
  newPlan.trajectory_.joint_trajectory.points[0].accelerations = std::vector<double>(numberOfJoints, 0.0);

  //Er bare interessert i første punkt
  std::vector<trajectory_msgs::msg::JointTrajectoryPoint> firstPoint;
  firstPoint.push_back(newPlan.trajectory_.joint_trajectory.points[0]);
```

```
  newPlan.trajectory_.joint_trajectory.points = firstPoint;

  return newPlan;
}

builtin_interfaces::msg::Duration addDurations(builtin_interfaces::msg::Duration dur1, builtin_interfaces::msg::Duration dur2){
  builtin_interfaces::msg::Duration result;
  result.nanosec = (dur1.nanosec + dur2.nanosec)%1000000000;
  result.sec = dur1.sec + dur2.sec + std::floor((dur1.nanosec + dur2.nanosec)/1000000000);
  return result;
}

builtin_interfaces::msg::Duration divideDuration(const builtin_interfaces::msg::Duration& d, float value) {
    builtin_interfaces::msg::Duration result;
    long long totalNanosec = d.sec * 1000000000 + d.nanosec; // Convert to nanoseconds
    totalNanosec = totalNanosec + totalNanosec/value; // Divide by input
    result.sec = totalNanosec / 1000000000; // Convert back to seconds and nanoseconds
    result.nanosec = totalNanosec % 1000000000;
    return result;
}


void addToPlan(moveit::planning_interface::MoveGroupInterface::Plan& plan,const moveit::planning_interface::MoveGroupInterface::Plan planToAdd, int startidx){
  //plan has a length i
  //and a with j
  //moveit::planning_interface::MoveGroupInterface::Plan results(plan);
  log("trying to add vector with length: " + std::to_string(planToAdd.trajectory_.joint_trajectory.points.size()) + " at index " + std::to_string(startidx) + " to a vector with size " + std::to_string(plan.trajectory_.joint_trajectory.points.size()), 1);

  for(int i = startidx; i < startidx + planToAdd.trajectory_.joint_trajectory.points.size(); i ++){
    int idx = findIndex(planToAdd.trajectory_.joint_trajectory.joint_names, plan.start_state_.joint_state.name); //denne sikrer at gruppen, dersom den ikke gjør det vil ikke det finnes fysiske joints å bevege
    if (idx >= 0){ //if the group exist insert, startidx is >= 0
      plan.trajectory_.joint_trajectory.points.at(i).positions.erase(plan.trajectory_.joint_trajectory.points.at(i).positions.begin() + idx, plan.trajectory_.joint_trajectory.points.at(i).
```

```
positions.begin() + idx + planToAdd.trajectory_.joint_trajectory.joint_names.size());
      plan.trajectory_.joint_trajectory.points.at(i).positions.insert(plan.trajectory_.joint_trajectory.points.at(i).positions.begin() + idx, planToAdd.trajectory_.joint_trajectory.points.at(i-startidx).positions.begin(), planToAdd.trajectory_.joint_trajectory.points.at(i-startidx).positions.end());
      plan.trajectory_.joint_trajectory.points.at(i).velocities.erase(plan.trajectory_.joint_trajectory.points.at(i).velocities.begin() + idx, plan.trajectory_.joint_trajectory.points.at(i).velocities.begin() + idx + planToAdd.trajectory_.joint_trajectory.joint_names.size());
      plan.trajectory_.joint_trajectory.points.at(i).velocities.insert(plan.trajectory_.joint_trajectory.points.at(i).velocities.begin() + idx, planToAdd.trajectory_.joint_trajectory.points.at(i-startidx).velocities.begin(), planToAdd.trajectory_.joint_trajectory.points.at(i-startidx).velocities.end());

      //planToAdd starter på t = 0, må starte ved t = t_prev_bane
      plan.trajectory_.joint_trajectory.points.at(i).time_from_start = addDurations(planToAdd.trajectory_.joint_trajectory.points.at(i - startidx).time_from_start, plan.trajectory_.joint_trajectory.points.back().time_from_start);
    }
  }
  //bruker plans siste tid til å lagre den akkumulerte tiden banen tar
  if(startidx + planToAdd.trajectory_.joint_trajectory.points.size() != plan.trajectory_.joint_trajectory.points.size()){
    plan.trajectory_.joint_trajectory.points.back().time_from_start = addDurations(plan.trajectory_.joint_trajectory.points.back().time_from_start, planToAdd.trajectory_.joint_trajectory.points.back().time_from_start);
  }
  //return results;
}



int8_t findIndex(std::vector<std::string> subset, std::vector<std::string> set){
  //rekkefølgen er lik slik at vi kan bare iterere opp
  //samtidig vil set.size() > subset.size() slik at vi kan iterere gjennom null problem
  for (uint8_t i = 0; i < set.size(); i ++){
    //finner den første matchen
    if(set.at(i) == subset.front()){
      return i;
    }
  }
  return -1;
```

```cpp
}



sensor_msgs::msg::JointState concatenateStates(const std::vector<moveit::planning_interface::MoveGroupInterface::Plan>& plans){ //Slår sammen states fra forskjellige grupper slik at du får en vector med statene
  sensor_msgs::msg::JointState state(plans[0].start_state_.joint_state);
  /*
  må finne ut hvilken posisjon i joint_state de forskjellige verdiene skal
  group_names = vector<string>
  */
  for (auto const& plan : plans){
    int8_t idx = findIndex(plan.trajectory_.joint_trajectory.joint_names, plan.start_state_.joint_state.name);
    int8_t sizeOfGroup = plan.trajectory_.joint_trajectory.joint_names.size();
    if (idx >= 0){
      state.position.erase(state.position.begin() + idx, state.position.begin() + idx + sizeOfGroup);
      state.position.insert(state.position.begin() + idx, plan.trajectory_.joint_trajectory.points.back().positions.begin(), plan.trajectory_.joint_trajectory.points.back().positions.end());
    }
  }
  return state;
}

moveit::planning_interface::MoveGroupInterface::Plan stupidPlanCreator(const std::string& group, const std::shared_ptr<rclcpp::Node> node){
  moveit::planning_interface::MoveGroupInterface::Plan newPlan;
  using moveit::planning_interface::MoveGroupInterface;
  auto move_group_interface = MoveGroupInterface(node, group);

  sensor_msgs::msg::JointState tempState;
  tempState.name = move_group_interface.getJointNames();
  for(auto const& joint : tempState.name){
    tempState.position.push_back(0.0);
  }
  move_group_interface.setJointValueTarget(tempState);
  move_group_interface.plan(newPlan);
  std::cout << newPlan.start_state_.joint_state.header.frame_id << std::endl;

  return newPlan;
}
```

```cpp



std::vector<moveit::planning_interface::MoveGroupInterface::Plan> findPlans(const std::vector<geometry_msgs::msg::Pose>& points, const std::string& group, const std::shared_ptr<rclcpp::Node> node){
  std::vector<moveit::planning_interface::MoveGroupInterface::Plan> plans;
  //Hentet fra moveit cpp tutorial
  using moveit::planning_interface::MoveGroupInterface;
  auto move_group_interface = MoveGroupInterface(node, group);
  move_group_interface.setPlanningTime(20); //dette er dumt

  //log(move_group_interface.getEndEffectorLink());
  move_group_interface.setMaxVelocityScalingFactor(0.02); //setter skalering -> nærmere 0 == tregere

  //the first point has the robot current startvalues
  for (auto const& point : points){
    if (point == points.front()){ //if first plans is empty
      log("setting target pose", 1);
      move_group_interface.setPoseTarget(point);
      log("creating plan", 1);
      auto const [success, plan] = [&move_group_interface]{
        moveit::planning_interface::MoveGroupInterface::Plan msg;
        auto const ok = static_cast<bool>(move_group_interface.plan(msg));
        return std::make_pair(ok, msg);
      }();
      if (success){
        plans.push_back(plan);
      }
      else{
        log("PLANNING FAILED! EXITING");
        exit(-1);
      }
    }
    else{//else calculate from previous position
      moveit_msgs::msg::RobotState startState;
      sensor_msgs::msg::JointState temp = concatenateStates(std::vector<moveit::planning_interface::MoveGroupInterface::Plan>{plans.back()});
      startState.joint_state.name = temp.name;
      startState.joint_state.position = temp.position;
      startState.joint_state.velocity = temp.velocity;
      startState.joint_state.effort = temp.effort;

      log("setting start state to previous end state", 1);
      move_group_interface.setStartState(startState);
      log("setting target pose", 1);
```

```
      move_group_interface.setPoseTarget(point);
      log("creating plan", 1);
      auto const [success, plan] = [&move_group_interface]{
        moveit::planning_interface::MoveGroupInterface::Plan msg;
        auto const ok = static_cast<bool>(move_group_interface.plan(msg));
        return std::make_pair(ok, msg);
      }();
      if (success){
        plans.push_back(plan);
      }
      else{
        log("PLANNING FAILED! EXITING");
        exit(-1);
      }
    }
  }
  return plans;
}




std::vector<geometry_msgs::msg::Pose> createStraightPathPoints(std::vector<double> xyz_start, std::vector<double> xyz_stop, std::vector<double> xyzw_orientation, int num_points){
  std::vector<geometry_msgs::msg::Pose> points;

    //std::cout << xyz_start.size();
  assert(xyzw_orientation.size() == 4); //må være verdier for alle
  //assert((xyz_start.size() == xyz_stop.size()) == 3); //må være verdier for xyz

  auto const target_pose = [](double x, double y, double z, std::vector<double> xyzw_orientation){
    geometry_msgs::msg::Pose msg;
    msg.position.x = x;
    msg.position.y = y;
    msg.position.z = z;
    msg.orientation.x = xyzw_orientation.at(0);
    msg.orientation.y = xyzw_orientation.at(1);
    msg.orientation.z = xyzw_orientation.at(2);
    msg.orientation.w = xyzw_orientation.at(3);
    return msg;
  };

  double dx = (xyz_stop.at(0) - xyz_start.at(0))/num_points;
  double dy = (xyz_stop.at(1) - xyz_start.at(1))/num_points;
  double dz = (xyz_stop.at(2) - xyz_start.at(2))/num_points;

  for(int i = 0; i < num_points - 1; i++){
```

```
    points.push_back(target_pose(xyz_start.at(0) + dx * i, xyz_start.at(1) + dy * i, xyz_start.at(2) + dz * i, xyzw_orientation));
  }
  points.push_back(target_pose(xyz_stop.at(0), xyz_stop.at(1), xyz_stop.at(2), xyzw_orientation));
  return points;
}


moveit_msgs::msg::JointConstraint createJointConstrain(std::string joint_name, double lower_limit, double upper_limit){
    moveit_msgs::msg::JointConstraint result;
    result.joint_name = joint_name;
    result.tolerance_below = lower_limit;
    result.tolerance_above = upper_limit;
    result.weight = 0.3;
    //må potensielt ha posisjon også
    return result;

}


moveit_msgs::msg::Constraints createJointConstrains(std::vector<std::string> joint_names, std::vector<double> lower_constrains, std::vector<double> upper_constrains){
    moveit_msgs::msg::Constraints results;
    for(int i = 0; i < joint_names.size(); i ++){
        results.joint_constraints.push_back(createJointConstrain(joint_names.at(i), lower_constrains.at(i), upper_constrains.at(i)));
    }
    return results;
}



geometry_msgs::msg::Pose createPose(double x, double y, double z, double ox, double oy, double oz, double ow){
  geometry_msgs::msg::Pose result;
  result.orientation.set__w(ow);
  result.orientation.set__x(ox);
  result.orientation.set__y(oy);
  result.orientation.set__z(oz);
  result.position.set__x(x);
  result.position.set__y(y);
  result.position.set__z(z);

  return result;
}
```

Appendix J.

planner_node.cpp

The main ROS2 node planner_node

```
#include "utilities.h"

#include <moveit_group_planner_interfaces/msg/waypointsets.hpp>
#include <moveit_group_planner_interfaces/msg/waypoints.hpp>
#include <moveit_group_planner_interfaces/srv/execute.hpp>
#include <moveit_group_planner_interfaces/srv/plan.hpp>

#include "fstream"

//#include <fstream> //used for storing and plotting the trajectory/
    velocity etc in python


using std::placeholders::_1; //used by service / subscription
using std::placeholders::_2; //used by service / subscription
using std::placeholders::_3; //used by service / subscription
using std::placeholders::_4; //used by service / subscription

//Class in main and not in .h because legacy from development.
class WaypointListener : public rclcpp::Node
{
  public:
    WaypointListener()
    : Node("waypoint_listener"), logger_(rclcpp::get_logger("
    waypoint_listener"))
    {
      joint_state_callback_group_     = this->create_callback_group(
    rclcpp::CallbackGroupType::MutuallyExclusive);
      ready_publisher_callback_group_ = this->create_callback_group(
    rclcpp::CallbackGroupType::MutuallyExclusive);

      auto joint_state_callback_group_options_ = rclcpp::
    SubscriptionOptions();
```

```
    auto execute_callback_group_options_     = rclcpp::SubscriptionOptions();
    joint_state_callback_group_options_.callback_group = joint_state_callback_group_;
    joint_state_subscription_ = this->create_subscription<sensor_msgs::msg::JointState>(
      "joint_states", 10, std::bind(&WaypointListener::joint_state_callback, this, _1), joint_state_callback_group_options_);

    plan_service_ = this->create_service<moveit_group_planner_interfaces::srv::Plan>(
      "plan_group", std::bind(&WaypointListener::plan_callback, this, std::placeholders::_1, std::placeholders::_2));

    execute_service_= this->create_service<moveit_group_planner_interfaces::srv::Execute>(
      "execute_plan",std::bind(&WaypointListener::execute_callback, this,
                                    std::placeholders::_2));

    ready_publisher_ = this->create_publisher<std_msgs::msg::Bool>("ready", 10);

    //alternative for inline like in execute_service and plan_service, because that did not work.
    std::function<void()> callback = std::bind(&WaypointListener::ready_publisher_callback, this, std::make_shared<std_msgs::msg::Bool>());
    ready_publisher_timer_ = this->create_wall_timer(std::chrono::milliseconds(50), callback, ready_publisher_callback_group_);
    RCLCPP_INFO_STREAM(logger_, "Node is spinning, ready to take waypoints");
  }

private:
  moveit::planning_interface::MoveGroupInterface::Plan plan; //For storing the trajectory for execution
  moveit::planning_interface::MoveGroupInterface* current_group; //For storing the relevant group for FK purposes
  std::string end_effector_link; //storing the relevant end effector, used to find FK such that in_position can be set
  bool in_position = false; //For notifying if the arm is in position to execute task - i.e in position to weld
  bool is_executing = false; //If robot is executing a plan - used in update_thread_function()

  //Because this is developed around motoros2 the main group is:
```

```
    //   follow_joint_trajectory
    //that is that motoros2 listens to actions for the group follow_joint_trajectory
    //however, if the controller is able to listen to multiple action topics, this may not be a good way to define this
    const std::shared_ptr<rclcpp::Node> followJointTrajectoryNode = std::make_shared<rclcpp::Node>("follow_joint_trajectory", rclcpp::NodeOptions().automatically_declare_parameters_from_overrides(true));
    moveit::planning_interface::MoveGroupInterface move_group_interface{followJointTrajectoryNode, "follow_joint_trajectory"}; //=MoveGroupInterface(followJointTrajectoryNode, "follow_joint_trajectory");
    moveit::core::RobotState robot_state = getRobotStateFromMoveGroupInterface(move_group_interface); //for storing the current robotstate and calculating FK for the system

    //these variables are used to store information about the current robot state - listening on /joint_states
    //the order of joints from controller may not be the same as from moveit
    std::vector<std::string> joint_names;           //reading the joint_names from joint_state topic
    std::vector<double> joint_positions;            //reading the joint_positions from joint_state topic
    std::vector<std::pair<bool, geometry_msgs::msg::Pose>> pairWaypoints; //used to store waypoint and the corresponding isJob - see update_thread_function()

    //multi-threading, need to lock
    std::mutex joint_position_mutex;
    std::mutex joint_names_mutex;
    std::mutex in_position_mutex;

    double end_effector_skip = 0.01;
    double jump_threshold    = 0.0;
    double end_effector_pose_tolerance = 0.05; // in meter. The tolerance of which end effector and waypoints is compared against in update_thread_function.
    int planner_time_limit   = 20; //sec, default 5


    bool allow_replanning = true;



    //_________________________ Class Functions _________________________//

    //joint_states_callback - listen to joint_states and updates the
```

```
private variables joint_position and joint_names
  void joint_state_callback(const sensor_msgs::msg::JointState::SharedPtr msg) {
    this->joint_position_mutex.lock();
    this->joint_names_mutex.lock();

    this->joint_positions = msg->position;
    this->joint_names = msg->name;

    this->joint_names_mutex.unlock();
    this->joint_position_mutex.unlock();



  }

    //plan_callback - callback for listening to waypoints - calls createPlan
    void plan_callback(const std::shared_ptr<moveit_group_planner_interfaces::srv::Plan::Request>  req,
                                std::shared_ptr<moveit_group_planner_interfaces::srv::Plan::Response> res){
      RCLCPP_INFO_STREAM(this->logger_, "Waypoints recived");
      std::string group_name = req->waypoints.groupname;
      std::vector<geometry_msgs::msg::Pose> waypoints = req->waypoints.waypoints;
      float speed = (req->waypoints.speed <= 0.0) ? 100000.0 : req->waypoints.speed; //if speed is set to <= 0, set the speed to a high value, else set to given value
      RCLCPP_INFO_STREAM(this->logger_, (req->waypoints.speed < 0.0) ? "Speed not valid. No limit set" :
                   ((req->waypoints.speed == 0.0) ? "No limit set" : ("Speed limit set to: " + std::to_string(speed) + "m/s")));


      //Asserts that the length of isJob and waypoints are the same
      //This is used to publish if job can be done
      //for example if waypoint is a part of a welding-path or not
      std::vector<bool> isJob = req->waypoints.is_job;
      if (isJob.size() not_eq waypoints.size()){
        RCLCPP_WARN_STREAM(this->logger_, "Length of isJob list is not equals length of waypoint list, sets all job to false");
        isJob = std::vector<bool>(waypoints.size(), false);
      }
      //stores in privat vector
      for (size_t i = 0; i < isJob.size(); i ++){
        pairWaypoints.push_back(std::make_pair(isJob.at(i), waypoints.at(i)));
      }

      //pass to createPlan
```

```
        double trajectory_fraction = createPlan(group_name, waypoints, speed);
        RCLCPP_INFO_STREAM(this->logger_, "Plan created, call /execute_plan service to execute");
        res->set__trajectory_fraction(static_cast<float>(trajectory_fraction));
    }

    //ready_publisher_callback - callback for publishing if robot is in position to start job or not
    void ready_publisher_callback(std::shared_ptr<std_msgs::msg::Bool> msg){
        this->in_position_mutex.lock();
        msg->data = this->in_position;
        this->ready_publisher_->publish(*msg);
        this->in_position_mutex.unlock();

    }

    //execute_callback - callback for executing planned trajectory if service is called - passes to moveit planning_interfaces
    void execute_callback(const std::shared_ptr<moveit_group_planner_interfaces::srv::Execute::Response> res){
        //using moveit::planning_interface::MoveGroupInterface;
        //auto const followJointTrajectoryNode = std::make_shared<rclcpp::Node>("follow_joint_trajectory", rclcpp::NodeOptions().automatically_declare_parameters_from_overrides(true));
        //auto move_group_interface = MoveGroupInterface(followJointTrajectoryNode, "follow_joint_trajectory");

        //execute the plan
        //while this should optionally return true if execute is finished, we can not get current state from this framework
        //if last point of end effector is end of waypoint list == return true
        //potentional workaround = listen to followjointtrajectory/result

        //if task is successfully sent down the pipeline, plan is executing, else something went wrong
        //this could be error in motoros or setup, for example wrong namespace or bad connection


        //FK thread because .execute wait for execution and need to update fk while this is happening
        this->is_executing = true;
        std::thread update_thread = std::thread(&WaypointListener::update_thread_function, this);

        if (this->move_group_interface.execute(this->plan).SUCCESS){
```

```
      res->set__success(true);
    }
    else{
      res->set__success(false);
    }

    this->is_executing = false;
    update_thread.join();
  }



  void update_thread_function(){


    int timeout = this->plan.trajectory_.joint_trajectory.points.back().time_from_start.sec;
    int start_time_ms = std::chrono::time_point_cast<std::chrono::milliseconds>(std::chrono::system_clock::now()).time_since_epoch().count(); // convert to milliseconds

    float tolerance = this->end_effector_pose_tolerance; //temp +- 5 cm may be too loose
    if (this->joint_positions.empty()){return;} //if joint position is empty, no FK is availible, segfault : return
    for(auto pair:this->pairWaypoints){//for each waypoint
      while(true){//wait untill waypoint is reached
        std::this_thread::yield();//update positions
        if(this->is_executing == false){
          //robot not executing - either finished or something went wrong
          this->in_position = false;
          return;
        }
        int current_time_ms = std::chrono::time_point_cast<std::chrono::milliseconds>(std::chrono::system_clock::now()).time_since_epoch().count();
        if((current_time_ms - start_time_ms) > ((1 + timeout) * 1000)){//if this has been going on longer than the path is expected, break
          RCLCPP_INFO_STREAM(this->logger_, "Execution timeout in thread, exiting thread");
          this->in_position = false;
          return;
        }

        if (this->joint_names_mutex.try_lock() || this->joint_position_mutex.try_lock()){
        //because the order of values from motoros2/robotcontroller may not be the same as values from moveit - need to reorder
```

```
        //restructure_vectors is a long process such that joint_state_listener may try access variables while the function is running leading to segfault
        //if these are locked we can not read values
          restructure_vectors(this->plan.trajectory_.joint_trajectory.joint_names, this->joint_names, this->joint_positions);
          this->joint_names_mutex.unlock();    //allow for joint_state_listener to update positions
          this->joint_position_mutex.unlock(); //allow for joint_state_listener to update positions
        }
        else{
          //this is updated quickly so we can skip one iteration in the "while loop"
          continue; //wait untill mutex is availible
        }
        //calculates FK
        this->robot_state.setVariablePositions(this->joint_positions);
        auto const fk = this->robot_state.getGlobalLinkTransform(this->end_effector_link);
        //fk is now a 4x4 Transformation matrix, while waypoints is a pose - need to convert one of them such that we can compare
        geometry_msgs::msg::Pose fk_pose = tf2::toMsg(fk);


        if(posesEqual(pair.second, fk_pose, tolerance)){
          if(this->in_position_mutex.try_lock()){ //this loop is way quicker than the publisher such that skipping an iteration in this is less bad than a publish
            this->in_position = pair.first;
            this->in_position_mutex.unlock();
            break; //waypoint reached
          }
        }
      }
    }
    this->in_position = false; //Asserts that the tool is off when finished.
    return;
  }

  //createPlan - takes group_name, a vector with waypoints and end effector speed
  //NOTE: THIS CAN NOT BE USED FOR <6 DOF
  //-Will create a cartesian path for the given group [group_name]
  //-Use iterative time parametrization to manipulate the trajectory of the end effector such that the desired speed is reached
  //-Expand the trajectory such that the trajectory contains the
```

```
values for each group in the system as system = [group_a, group_b, ...]
 //-Stores the plan class variable: plan
 double createPlan(std::string name, std::vector<geometry_msgs::msg::Pose> waypoints, float speed){
   using moveit::planning_interface::MoveGroupInterface;
   //makes a planning node and movegroup interface for calculating path for given group
   auto const group_node = std::make_shared<rclcpp::Node>(name, rclcpp::NodeOptions().automatically_declare_parameters_from_overrides(true));
   auto group_move_interface = MoveGroupInterface(group_node, name);
   group_move_interface.setPlanningTime(this->planner_time_limit); //Standard = 5 sec
   group_move_interface.allowReplanning(this->allow_replanning);

   //Since FK for the end effector only will work if the system can recognize the kinematics (i.e the link/joint pair-set from base to tip of the robot)
   //We need to store the relevant group for getting relevant FK later.
   this->current_group = &group_move_interface; //seg fault
   this->end_effector_link = group_move_interface.getEndEffectorLink();


   //contains plan for only a given group
   //stupidPlanCreator creates a plan with only some values filled, enough to work
   moveit::planning_interface::MoveGroupInterface::Plan tempPlan = stupidPlanCreator(name, group_node);

   //calculating cartesian path
   //If fraction < 1, the planner has exited before visiting all waypoints
   //this can be because of collision, orientation or out-of-reach
   moveit_msgs::msg::RobotTrajectory traj;
   double pathFraction = group_move_interface.computeCartesianPath(waypoints, this->end_effector_skip, this->jump_threshold,traj);
   RCLCPP_INFO_STREAM(this->logger_, "Fraction of trajectory found: " << pathFraction);
   if (pathFraction < 1){
       //todo find a way to detect what kind of error
       RCLCPP_WARN_STREAM(this->logger_, "Could not compute path for the whole set of waypoints, this is likely because of point out of reach, collision or orientation not reachable");
       RCLCPP_WARN_STREAM(this->logger_, "Does orientation-values have enough decimals?");
```

```
    }

    //limits the end effector velocity with method by : Benjamin Scholz, Thies Oelerich
    //---------------------------------------------------------------------------------------

    //The method used robot_Trajectory as input, while this code has used moveit_msgs::msg::RobotTrajectory
    //convert moveit_msgs::msg::RobotTrajectory to robot_trajectory::RobotTrajectory
    auto robot_state = getRobotStateFromMoveGroupInterface(group_move_interface);
    robot_trajectory::RobotTrajectory limitedTraj(robot_state.getRobotModel(), name);


    //sets robot_state.traj = traj
    //traj is the calculated path for the given group
    limitedTraj.setRobotTrajectoryMsg(robot_state, traj);

    //sets the speed with iterative time parametrization
    this->end_effector_link = group_move_interface.getEndEffectorLink();
    if(this->end_effector_link.empty()){RCLCPP_WARN_STREAM(this->logger_, "No end effector found for group: " << name);}
    trajectory_processing::limitMaxCartesianLinkSpeed(limitedTraj, speed, group_move_interface.getEndEffectorLink());
    // Because we need to know if the end effector
    // is in desired position (for example if we are ready to weld)
    // we need to store the poses for each position in the trajectory


    //inserts back into a moveit_msgs::msg::RobotTrajectory
    limitedTraj.getRobotTrajectoryMsg(traj);

    //makes a new plan for the group. The above step only creates the trajectory and we need start_state etc
    moveit::planning_interface::MoveGroupInterface::Plan newPlan = newPlanFromStartState(tempPlan, "this is not used", traj.joint_trajectory.joint_names.size(), findIndex(traj.joint_trajectory.joint_names, tempPlan.start_state_.joint_state.name));
    newPlan.trajectory_ = traj;

    // From motoros2, because of limited memory in the controller a trajectory can not be too long
    // There are no check for if the trajectory is too long, and depending of how many points and joints in the system,
```

```
    // this limit is not well-defined.
    // for a system consisting of 15 joint, this occured at 166 points, while the developer of motoros noticed 200 points
    // Warns a warning
    int size_of_trajectory = newPlan.trajectory_.joint_trajectory.points.size();
    RCLCPP_INFO_STREAM(this->logger_, "Trajectory has " << size_of_trajectory << " points");
    if(size_of_trajectory > 150){
      RCLCPP_WARN_STREAM(this->logger_, "WARNING: Path long, may cause crash in motoros2 as controller may not have enough memory");
    }

    //The above steps only for single group and not the whole system
    //need this path into a wider path containg all the joint in system
    moveit::planning_interface::MoveGroupInterface::Plan mergedPlan = expandTrajectory(newPlanFromStartState(newPlan, "this is not used", newPlan.start_state_.joint_state.name.size(), 0), newPlan.trajectory_.joint_trajectory.points.size());

    //adds the plan from groupplan into systemplan
    addToPlan(mergedPlan, newPlan);
    //stores the plan for later execution
    this->plan = mergedPlan;


    //________Stores pos and vel data for experimental purposes__________//
    //this will not work if "johan" not user, couts a warning
    std::cout << "Storing velocity and position data for experimental purposes - saves to user 'johan' and may cause error if other user" << std::endl;
    std::ofstream posfile("/home/johan/debugs/positions.txt");
    std::ofstream velfile("/home/johan/debugs/velocities.txt");
    for(auto const& it : this->plan.trajectory_.joint_trajectory.points){
      //for each point in plan
      posfile << it.time_from_start.sec <<"."<<it.time_from_start.nanosec<<" ";
      velfile << it.time_from_start.sec <<"."<<it.time_from_start.nanosec<<" ";
      for(auto const& jointpos : it.positions){
        //for each joint
        posfile << jointpos << " ";
      }
      posfile << std::endl;
      for(auto const& jointvel : it.velocities){
        //for each joint
```

```
          velfile << jointvel << " ";
        }
        velfile << std::endl;
      }
      posfile.close();
      velfile.close();

      //_______________________ END ________________________________//
      return pathFraction; //returns the fraction of the planned trajectory vs desired trajectory

    }


    //initializing ros-defined classes
    rclcpp::TimerBase::SharedPtr ready_publisher_timer_;
    rclcpp::Subscription<sensor_msgs::msg::JointState>::SharedPtr joint_state_subscription_;//??
    rclcpp::Publisher<std_msgs::msg::Bool>::SharedPtr ready_publisher_;
    rclcpp::Service<moveit_group_planner_interfaces::srv::Plan>::SharedPtr plan_service_;
    rclcpp::Service<moveit_group_planner_interfaces::srv::Execute>::SharedPtr execute_service_;
    rclcpp::Logger logger_;

    //used for multi-threading callbacks
    rclcpp::CallbackGroup::SharedPtr joint_state_callback_group_;
    rclcpp::CallbackGroup::SharedPtr execute_callback_group_;
    rclcpp::CallbackGroup::SharedPtr ready_publisher_callback_group_;



};





int main(int argc, char * argv[])
{
  //std::cout << argv[1]; could use this as global group name (/follow_joint_trajectory)
  rclcpp::init(argc, argv);
  //need executor as we require multiple callbacks at the same time
  //rclcpp::spin(std::make_shared<WaypointListener>()); //single thread

  rclcpp::executors::MultiThreadedExecutor executor;
```

```
    //creates executor
  auto waypoint_listener_node = std::make_shared<WaypointListener>(); //create node
  executor.add_node(waypoint_listener_node);    //add node to executor
  executor.spin();    //spins the executor - runs the program

  rclcpp::shutdown();
  return 0;
}
```

Appendix K.

jacobian_generator.py

This is an experimental program which calculates forward kinematics, jacobian and velocity from a given URDF, velocity- and position lists and plots the result. Furthermore, it has an experimental function which attempts to reduce the joint velocities by inv(jacobian) * desired_joint_velocities.

```
#import roslib
#roslib.load_manifest("urdfdom_py")
#import rospy
import modern_robotics as mr
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from urdf_parser_py.urdf import URDF


def skew(axis): #returns skew representation of the 3x1 vector
    assert(len(axis) == 3)
    return np.array([[0, -axis[2], axis[1]],
                     [axis[2], 0, -axis[0]],
                     [-axis[1], axis[0],0]])

def exp3(axis, theta = 0):
    return np.eye(3) + np.sin(theta) * skew(axis) + (1 - np.cos(
    theta)) * skew(axis) @ skew(axis)

def Tmat(R, r):
    T = np.eye(4)
    T[:3,:3] = R
    T[:3, 3] = r
    return T

def getPrefixes(jointlist) -> list:
    prefixes = []
    for joint in jointlist:
```

```
        prefix = joint.name.split("/")[0]
        if prefix not in prefixes:
            prefixes.append(prefix)
    return prefixes

def getTransformationInChain(chain): #returns a list with sudo-transformations from joint_i to joint_i+1
    Ts = []
    for i in range(len(chain)):
        dx = chain[i].origin.xyz[0]
        dy = chain[i].origin.xyz[1]
        dz = chain[i].origin.xyz[2]
        r = np.array([dx, dy, dz])

        axis      = np.array([chain[i].origin.rpy[0], chain[i].origin.rpy[1], chain[i].origin.rpy[2]])
        #because of the way the urdf is set up, if there are no rotation between the frame, the norm of "axis" will be 0
        #if there are any rotations, the norm will be non-zero
        #Furthermore, a joint rotate about its "z" axis, so that this doesn|t really makes sense
        #However, if the slt model is defined such that each joint/link follows the same coordinate system
        #the axis will be correct with the respect of global frame, and it can be used with PoE conventions
        theta = np.linalg.norm(axis)
        if theta != 0:
            axis /= theta

        rot = exp3(axis, theta)
        diff_rot = rot
        Ts.append(Tmat(diff_rot, r))
    return Ts



def getGroups(jointlist) -> dict: #returns a dict {group, [joints]} #if group_x exist as child of group_y, then group_y includes joints of group_x. but not other way
    prefixes = getPrefixes(jointlist)
    groups = {}
    for prefix in prefixes:
        groups[prefix] = []

    for joint in jointlist:
        for prefix in prefixes:
            if prefix in joint.name:
                groups[prefix].append(joint)
    return groups


```

```python
def getChain(jointlist) -> list: #A group may contain additional "virtual" fixed joint/links which will not be a part of the kinematic

    """Assuming each group has a "joint_1",
    this function returns the chain defined from the first joint and out.
    This will not take links before the first dynamic joint into account

    for example [fixed_joint_1, fixed_joint_2, joint_1, joint_2 ...] -> [joint_1, joint_2 ...]
    where fixed_jont_1 etc is joints defining the fixed transformation between two frames/joints
    """
    chains = {}

    groups = getGroups(jointlist)
    for key in groups:
        chain = []
        group_joints = groups[key]

        for i, joint in enumerate(group_joints):
            if "joint_1" in joint.name: ##if "fixed" not in joint.type
                chain.append(joint)
                child = joint.child
                for j, obj in enumerate(jointlist):
                    if obj.parent == child:
                        chain.append(obj)
                        child = obj.child

                chains[key] = chain
            chain = []
    return chains


def Slist(chain):
    #takes a chain and return the spatial twist in home position
    #e^(s*theta) = T
    #The M position (home position)
    Ts =  getTransformationInChain(chain)
    qs = []
    Rs = []

    M = np.eye(4) #Tsb
    for i in Ts:
        M = M @ i
        qs.append(M[:3,3])
        Rs.append(M[:3, :3])

```

```
    #S_i = rotation_axis, -(w x q)

    S = []
    for i, joint in enumerate(chain):
        #only supported for revolute and prismatic joints
        if joint.type == "revolute":
            #from urdf, the joint.axis is in the respect of the joint in question
            #it is however defined as rotation the geometries origin (.slt file)

            w_s = np.array([joint.axis[0], joint.axis[1], joint.axis[2]])
            v = -skew(w_s) @ qs[i] #qs[i]
            S.append(np.hstack((w_s, v)))

        if joint.type == "prismatic":
            w_s = np.array([0,0,0])
            v = np.array([joint.axis[0], joint.axis[1], joint.axis[2]])
            S.append(np.hstack((w_s, v)))
    #[s_1, s_2, s_3, ...]
    return np.array(S).T




def calculateSpatialJacobian(chain, thetalist):
    S = Slist(chain)
    return mr.JacobianSpace(S, thetalist)

def sToBtwist(chain, s):
    M = getM(chain)
    #To change the reference
    #Vb = Ad(Tbs) Vs
    return Adjoint(np.linalg.inv(M)) @ s

def sToBjac(chain, J_s):
    M = getM(chain)
    return mr.Adjoint(np.linalg.inv(M)) @ J_s

def getM(chain):
    Ts = getTransformationInChain(chain)
    M = np.eye(4) #Tsb
    for T in Ts:
        M = M @ T
    return M


def fk(chain, theta):
```

```
    #T0-n = Prod(exp6(s_i, theta_i))@M

    M = getM(chain)
    S = Slist(chain)

    return mr.FKinSpace(M, S, theta)



def capSpeed(chain, velocitylist, positionlist, timelist, max_speed = None, timestep = 0.1):

    """takes a trajectory, max cartesian speed and timestep
       returns a trajectory following the same paths but with max_speed velocities

       from moveit, timestep seems to be 0.1 but this is not nessisarily the case
    """

    #len position = len time = len velocity


    newTimelist     = []
    newVelocitylist = []
    newPositionlist = []


    #plan:
    #   for pos, vel:
    #       is linearvel > max_speed?
    #           new_vel = max_vel
    #           new_pos = pos + vel * timestep
    i = 0

    prev_time = 0.0
    delta = 0.0
    newTimelist.append(0) #first step at time 0

    for theta, dtheta, t in zip(positionlist, velocitylist, timelist):
        jac = sToBjac(chain, calculateSpatialJacobian(chain, theta))
        end_effector_twist = jac @ dtheta
        end_effector_speed = (np.linalg.norm(end_effector_twist[3:]))

        if end_effector_speed > max_speed:
            scalefactor = max_speed / end_effector_speed
            scaled_twist = end_effector_twist * scalefactor
            new_velocity = np.linalg.inv(jac) @ scaled_twist

```

```
        else:
            new_velocity = dtheta
        newVelocitylist.append(new_velocity)

        #updated version of setting speed

        #fk from this point and to next
        if (i != len(positionlist)-1): #if there exist a next point
            fk_current = fk(chain, theta)
            fk_next    = fk(chain, positionlist[i+1])
            euclidian_diff = np.linalg.norm(fk_next[:3,3] - fk_current[:3,3]) #length of the x,y,z components
            timestep = euclidian_diff / max_speed #m / m/s = s
            newTimelist.append(timestep + newTimelist[-1]) #time to reach the next point



        i += 1

    return newVelocitylist, positionlist, newTimelist



def main():
    robot = URDF.from_xml_file("/home/johan/ws_test2/src/motoman/motoman_gp25sys_support/urdf/gp25sys.urdf")
    #robot = URDF.from_parameter_server()
    joints = robot.joints
    robotchains = getChain(joints)



    timelist = []
    velocitylist = []
    positionlist = []

    endeffectorvelocity_space = []
    endeffectorvelocity_body = []
    endeffectorabsvelocity_space = []
    endeffectorabsvelocity_body = []

    endeffectorabsangular_body = []

    cartesianPosx = []
    cartesianPosy = []
    cartesianPosz = []

    startidx, endidx = 1, 7 #idx 0 = time, 1-15 = joints

    with open("/home/johan/debugs/positions.txt", "r") as file:
```

```
        lines = file.readlines()
        for line in lines:
            if line.strip():
                vals = [float(i) for i in line.split(" ") if i.strip()]  # skip empty strings
                #timelist.append(vals[0])
                positionlist.append(np.array(vals[startidx:endidx]))

    with open("/home/johan/debugs/positions.txt", "r") as file:
        lines = file.readlines()
        for line in lines:
            if line.strip():
                time = line.split(" ")[0]
                secs, nanos = time.split(".")
                if len(nanos) < 9: #asserts that x sec and 999 nanosec = x sec and 0.000000999 nanosec instead of 1 sec and 999000000 nanosec
                    nanos = "0"*(9-len(nanos)) + nanos
                timelist.append(float(secs + "." + nanos))


    with open("/home/johan/debugs/velocities.txt", "r") as file:
        lines = file.readlines()
        for line in lines:
            if line.strip():
                vals = [float(i) for i in line.split(" ") if i.strip()]  # skip empty strings
                velocitylist.append(np.array(vals[startidx:endidx]))

    chain = robotchains["group_2"] #the desired group to calculate the kinematics for

    for theta, dtheta in zip(positionlist, velocitylist):
        #twist = [w, v] = angular velocity, linear velocity
        jac = calculateSpatialJacobian(chain, theta)
        endeffectorvelocity_space.append(jac @ dtheta)
        endeffectorabsvelocity_space.append(np.linalg.norm(endeffectorvelocity_space[-1][3:]))
        endeffectorvelocity_body.append(sToBjac(chain, jac) @ dtheta)
        endeffectorabsvelocity_body.append(np.linalg.norm(endeffectorvelocity_body[-1][3:]))
        endeffectorabsangular_body.append(np.linalg.norm(endeffectorvelocity_body[-1][:3]))



        T = fk(chain, theta)
        cartesianPosx.append(T[0,-1])
        cartesianPosy.append(T[1,-1])
        cartesianPosz.append(T[2,-1])
```

```
    plt.figure()
    plt.plot(timelist, endeffectorabsvelocity_body, label="End effector speed")
    plt.xlabel("time [s]")
    plt.ylabel("Velocity [m/s]")
    plt.legend()

    ### 3D plot for plotting end effector trajectory ###
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')


    ax.scatter(cartesianPosx, cartesianPosy, cartesianPosz)
    for i in range(len(timelist)):
        ax.text(cartesianPosx[i], cartesianPosy[i], cartesianPosz[i], str(timelist[i]), color='red')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.show()



    newVelocitylist, newpositionlist, newtimelist = capSpeed(chain, velocitylist, positionlist, timelist, max_speed = 0.02)
    endeffectorvelocity_space = []
    endeffectorvelocity_body = []
    endeffectorabsvelocity_space = []
    endeffectorabsvelocity_body = []
    cartesianPosx = []
    cartesianPosy = []
    cartesianPosz = []


    for theta, dtheta in zip(newpositionlist, newVelocitylist):
        #twist = [w, v] = angular velocity, linear velocity
        jac = calculateSpatialJacobian(chain, theta)
        endeffectorvelocity_space.append(jac @ dtheta)
        endeffectorabsvelocity_space.append(np.linalg.norm(endeffectorvelocity_space[-1][3:]))
        endeffectorvelocity_body.append(sToBjac(chain, jac) @ dtheta)
        endeffectorabsvelocity_body.append(np.linalg.norm(endeffectorvelocity_body[-1][3:]))
        endeffectorabsangular_body.append(np.linalg.norm(endeffectorvelocity_body[-1][:3]))
        T = fk(chain, theta)
        cartesianPosx.append(T[0,-1])
        cartesianPosy.append(T[1,-1])
```

```
        cartesianPosz.append(T[2,-1])


    ### 3D plot for plotting end effector trajectory ####
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')


    ax.scatter(cartesianPosx, cartesianPosy, cartesianPosz)
    for i in range(len(timelist)):
        ax.text(cartesianPosx[i], cartesianPosy[i], cartesianPosz[i
], str(newtimelist[i]), color='red')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.figure()
    plt.plot(newtimelist, endeffectorabsvelocity_body, label="End
effector speed")
    plt.xlabel("time [s]")
    plt.ylabel("Velocity [m/s]")
    plt.legend()
    plt.show()


if __name__ == '__main__':
    main()
```

Appendix L.

waypoint_publisher.py

The script for testing the robots capability.

```python
import rclpy
import math
from rclpy.node import Node
import sys

from std_msgs.msg import String
from moveit_group_planner_interfaces.msg import Waypoints
from moveit_group_planner_interfaces.srv import Plan
from geometry_msgs.msg import Pose

DEBUG = True

#the translation from worldframe to workpiece center.
cy = 1.532
cx = 0.0
cz = 0.575

MAX_SPEED = 0.1 #zero is limitless
GROUP_NAME = "group_1"
JOB_NR     = 1

offset = 25.e-3/2 + 10.e-3 #half of the diameter of the end effector + 10mm # -approach


def quaternion_mult(q,r):
    # Extract individual components of the quaternions
    x1, y1, z1, w1 = q
    x2, y2, z2, w2 = r

    # Perform quaternion multiplication
    x = w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2
    y = w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2
```

```
    z = w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2
    w = w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2
    return [x,y,z,w]

def quaternion_qunj(q):
    x, y, z, w = q

    # Compute the conjugate
    conjugate = [-x, -y, -z, w]

    return conjugate


def point_rotation_by_quaternion(point,q):
    #q = xyzw
    r = point + [0] #adds w = 0 to xyz point for such that r represent a quaternion
    q_conj = quaternion_qunj(q)
    return quaternion_mult(quaternion_mult(q,r),q_conj)[:3]#returns xyz rotated

def create_pose(x, y, z, r = None , p = None , q = None, w= None):
    pose = Pose()
    pose.position.x = float(x)
    pose.position.y = float(y)
    pose.position.z = float(z)
    if (r != None):
        pose.orientation.x = float(r)
    if (p != None):
        pose.orientation.y = float(p)
    if(q != None):
        pose.orientation.z = float(q)
    if(w != None):
        pose.orientation.w = float(w)
    return pose


def circle(r, x, y, z, n_points = 100):
    points = []
    for i in range(n_points+1):  #resolution, + 1 for full circle

        #step = 2*3.1415/n_points

        xx = x + r*math.cos(i*2*3.1415/n_points)
        yy = y + r*math.sin(i*2*3.1415/n_points)
        points.append(create_pose(xx, yy, z ,1.0, 0.0, 0.0, 0.0)) # xyz + orientering
    return points
```

```
def getWaypoints(job_nr):
    msg = Waypoints()


    if job_nr == 0:

        diff = point_rotation_by_quaternion([0,0,  -offset], [-0.8535533905932737, 0.3535533905932736, -0.14644660940672624, 0.353553390593274]) # convert z in body frame to world
        #approach = point_rotation_by_quaternion([0,0,1], [-0.8535533905932737, 0.3535533905932736, -0.14644660940672624, 0.353553390593274]) #the end effector apporach dir

        msg.waypoints.append(create_pose(cx+0.5, cy, 1.4, -0.5720614, 0.5720614, 0, 0.5877852522924731))
        msg.waypoints.append(create_pose(cx+0.5, cy, 1.4, -0.8535533905932737, 0.3535533905932736, -0.14644660940672624, 0.353553390593274))
        msg.waypoints.append(create_pose(cx+0.5 + diff[0], cy + diff[1], 1.4 + diff[2], -0.8535533905932737, 0.3535533905932736, -0.14644660940672624, 0.353553390593274))

    if job_nr == 1:
        diff = point_rotation_by_quaternion([0,0,  -offset], [0.8535533905932738, -0.35355339059327373, -0.14644660940672619, 0.3535533905932738]) # convert z in body frame to world
        approach = point_rotation_by_quaternion([0,0,1], [0.8535533905932738, -0.35355339059327373, -0.14644660940672619, 0.3535533905932738])
        #gruppe_1 sveise langs kant
        msg.waypoints.append(create_pose(cx+0.5, cy, 1.4, -0.5720614, 0.5720614, 0, 0.5877852522924731)) #<- needed if robot is at home +-w if error
        msg.is_job.append(False) #død-bevegelse
        msg.waypoints.append(create_pose(cx + 0.065 + diff[0],
                                         cy + 0.065 + diff[1],
                                              0.73  + diff[2],
                                          1,0,0,0))#denne orienteringen er ca straight down group_1
        msg.is_job.append(False) #bevege ee til ca start

        msg.waypoints.append(create_pose(cx + 0.065 + diff[0],
                                         cy + 0.065 + diff[1],
                                               0.73 + diff[2],
                                         0.8535533905932738, -0.35355339059327373, -0.14644660940672619, 0.3535533905932738)) #q
        msg.is_job.append(False)

        #weld job
        msg.waypoints.append(create_pose(cx + 0.065 + diff[0],
```

```
                                                cy + 0.065 + diff[1],
                                                cz + 0.01  + diff[2],
                                                0.8535533905932738,
-0.35355339059327373, -0.14644660940672619, 0.353553390593273 8))
 #q
        msg.is_job.append(True)  #begynne sveis langs kortside av
profil
        msg.waypoints.append(create_pose(cx + 0.065   + diff[0],
                                                cy +  0.400  + diff[1],
                                                cz + 0.01    + diff[2],
                                                0.8535533905932738,
-0.35355339059327373, -0.14644660940672619, 0.3535533905932738))
 #q
        msg.is_job.append(False) #stopp sveis
        msg.waypoints.append(create_pose(cx + 0.065   + diff[0],
                                                cy +  0.400  + diff[1],
                                                cz + 0.13    + diff[2],
                                                0.8535533905932738,
-0.35355339059327373, -0.14644660940672619, 0.3535533905932738))
 #q
        msg.is_job.append(False) #løfte ee

    elif job_nr == 2:
        diff = point_rotation_by_quaternion([0,0,  -offset],
[-0.8535533905932737, 0.3535533905932736, -0.14644660940672624,
0.353553390593274]) # convert z in body frame to
        #gruppe_1 sveise langs kant
        msg.waypoints.append(create_pose(cx+0.5, cy, 1.4,
-0.5720614, 0.5720614, 0, 0.5877852522924731)) #<- needed if
robot is at home +-w if error
        msg.is_job.append(False) #død-bevegelse
        msg.waypoints.append(create_pose(cx-0.065, cy - 0.4 , 0.73,
1,0,0,0))#denne orienteringen er ca rett ned for gruppe 1
        msg.is_job.append(False) #bevege ee til ca start

        #weld job
        msg.waypoints.append(create_pose(cx - 0.065 + diff[0],
                                                cy - 0.4   + diff[1],
                                                cz + 0.13  + diff[2],
                                                -0.8535533905932737,
0.3535533905932736, -0.14644660940672624, 0.353553390593274)) #q
        msg.is_job.append(True)  #begynne sveis langs kortside av
profil

        msg.waypoints.append(create_pose(cx - 0.065 + diff[0],
                                                cy - 0.4   + diff[1],
                                                cz + 0.01  + diff[2],
                                                -0.8535533905932737,
0.3535533905932736, -0.14644660940672624, 0.353553390593274)) #q
        msg.is_job.append(False)
        msg.waypoints.append(create_pose(cx - 0.065 + diff[0],
```

```
                                                    cy -  0.065 + diff[1],
                                                    cz + 0.01  + diff[2],
                                                    -0.8535533905932737, 0.3535533905932736, -0.1464466094067262 4, 0.353553390593274)) #q
        msg.is_job.append(False) #stopp sveis
        msg.waypoints.append(create_pose(cx - 0.065 + diff[0],
                                                    cy -  0.065 + diff[1],
                                                    cz +  0.13 + diff[2],
                                                    -0.8535533905932737, 0.3535533905932736, -0.1464466094067262 4, 0.353553390593274)) #q
        msg.is_job.append(False) #løfte ee

    elif job_nr == 3:
        #gruppe_1 sveise invendig
        msg.is_job.append(False) #movement
        msg.waypoints.append(create_pose(cx+0.5, cy, 1.4 + 0.5, -0.5720614, 0.5720614, 0, -0.5877852522924731)) #<needed if robot at home +-w if error

        msg.waypoints.append(create_pose(cx+0.6, cy , 0.75, 0.856925, -0.514625, 0.0216612, -0.0192533))#denne orienteringen er ca rett ned for gruppe 1
        msg.is_job.append(False)#to position
        msg.waypoints.append(create_pose(cx + 0.50+0.05 , cy, 0.51 + 0.04 + 0.01 + 0.05, 0, -0.7071067811865476, 0.0, 0.7071067811865476))
        msg.is_job.append(False)#to position
        msg.waypoints.append(create_pose(cx + 0.50 , cy, 0.51 + 0.04 + 0.01 + 0.05, 0, -0.7071067811865476, 0.0, 0.7071067811865476))
        msg.is_job.append(True)
        msg.waypoints.append(create_pose(cx + 0.50 - 0.15 , cy, 0.51 + 0.04 + 0.01 + 0.05, 0, -0.7071067811865476, 0.0, 0.7071067811865476))
        msg.is_job.append(False) #"weld" along edge
        msg.waypoints.append(create_pose(cx + 0.50+0.05 , cy, 0.51 + 0.04 + 0.01 + 0.05, 0, -0.7071067811865476, 0.0, 0.7071067811865476))
        msg.is_job.append(False)#move ee out
        msg.waypoints.append(create_pose(cx+0.6, cy , 0.73, 0.856925, -0.514625, 0.0216612, -0.0192533))#denne orienteringen er ca rett ned for gruppe 1
        msg.is_job.append(False) #move up

    elif job_nr == 4:
        msg.waypoints.append(create_pose(cx, cy, 0.76, 1, 0, 0, 0))
        msg.waypoints.append(create_pose(cx+0.5, cy, 0.76, 1, 0, 0, 0))
        msg.waypoints.append(create_pose(cx-0.5, cy, 0.76, 1, 0, 0, 0))

```

```
    elif job_nr == 5:
        msg.waypoints = circle(0.5, cx, cy, 0.73, 100)
    if (len(msg.is_job) != len(msg.waypoints)):
            for w in msg.waypoints:
                msg.is_job.append(True)

    return msg.waypoints, msg.is_job

class MinimalPublisher(Node):
    #dette burde vært service
    def __init__(self):
        super().__init__('minimal_publisher')
        self.client_    = self.create_client(Plan, "plan_group")



    def call_service(self):
        #build the request
        req = Plan.Request()
        req.waypoints.groupname = GROUP_NAME
        req.waypoints.speed      = float(MAX_SPEED)
        print("creating plan...")
        req.waypoints.waypoints, req.waypoints.is_job = getWaypoints
    (JOB_NR)
        print("calling service...")
        #request request
        future = self.client_.call_async(req)
        rclpy.spin_until_future_complete(self, future)

        self.get_logger().info(str(future.result().
    trajectory_fraction))








def main(args=None):
    args=sys.argv
    try:
        JOB_NR = int(args)
    except Exception:
        pass #could not convert args to int

    rclpy.init()

    minimal_publisher = MinimalPublisher()
    minimal_publisher.call_service()

```

```
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Appendix M.

Joint Velocities for Test Cases 1, 2, 3, and 4

The joint velocities of the robot performing the test.

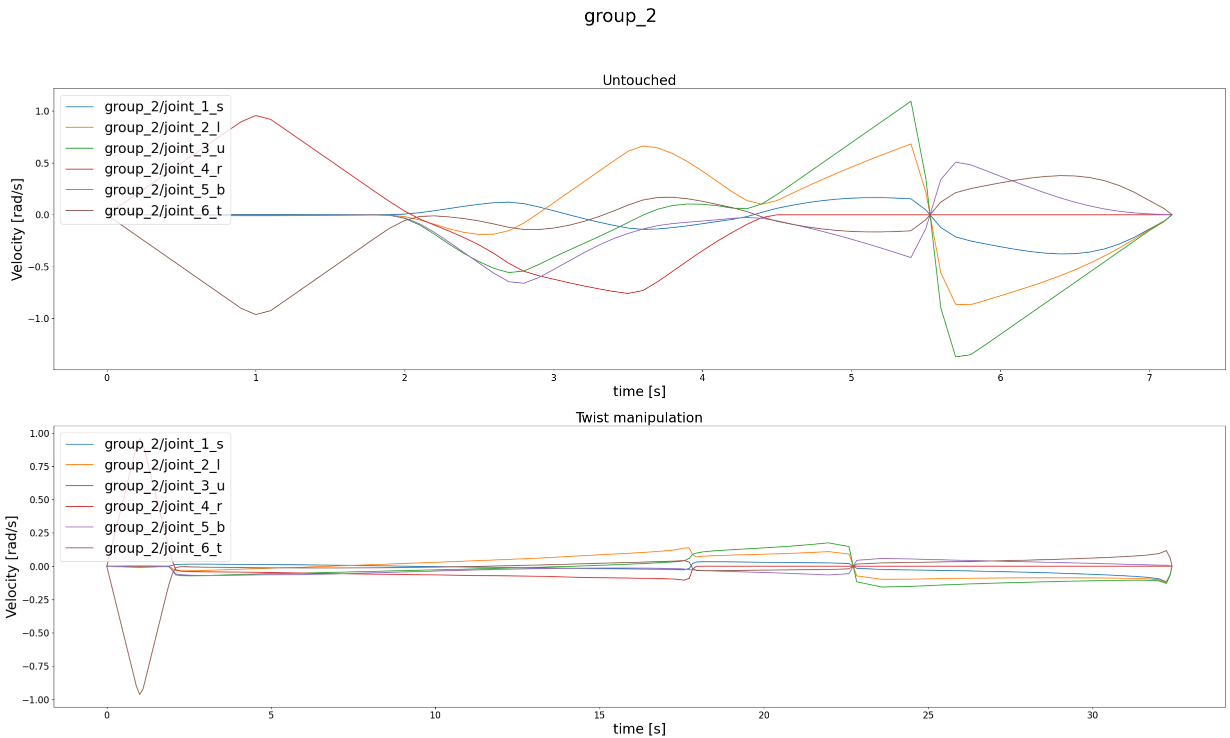


Figure M.1.: Joint velocities for straight line motion. Untouched (top) and twist method (bottom)

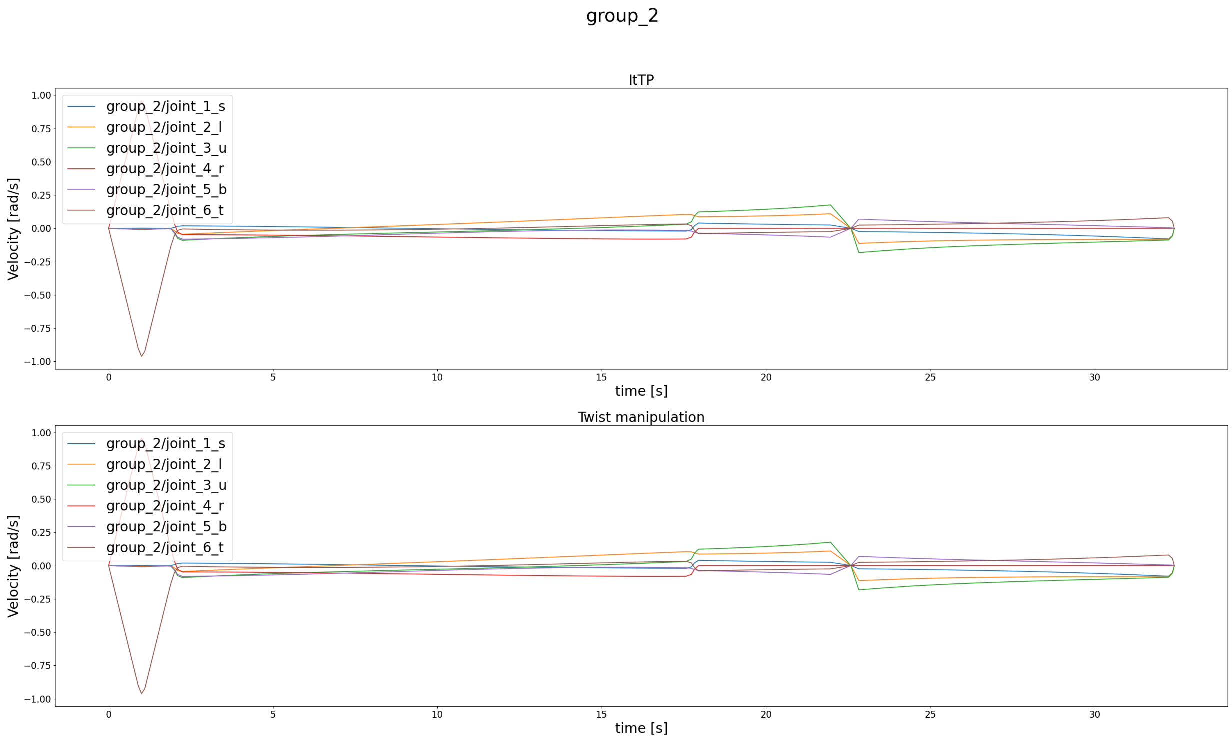


Figure M.2.: Joint velocities for straight line motion. Iterative time parameterization (top) and twist method (bottom)

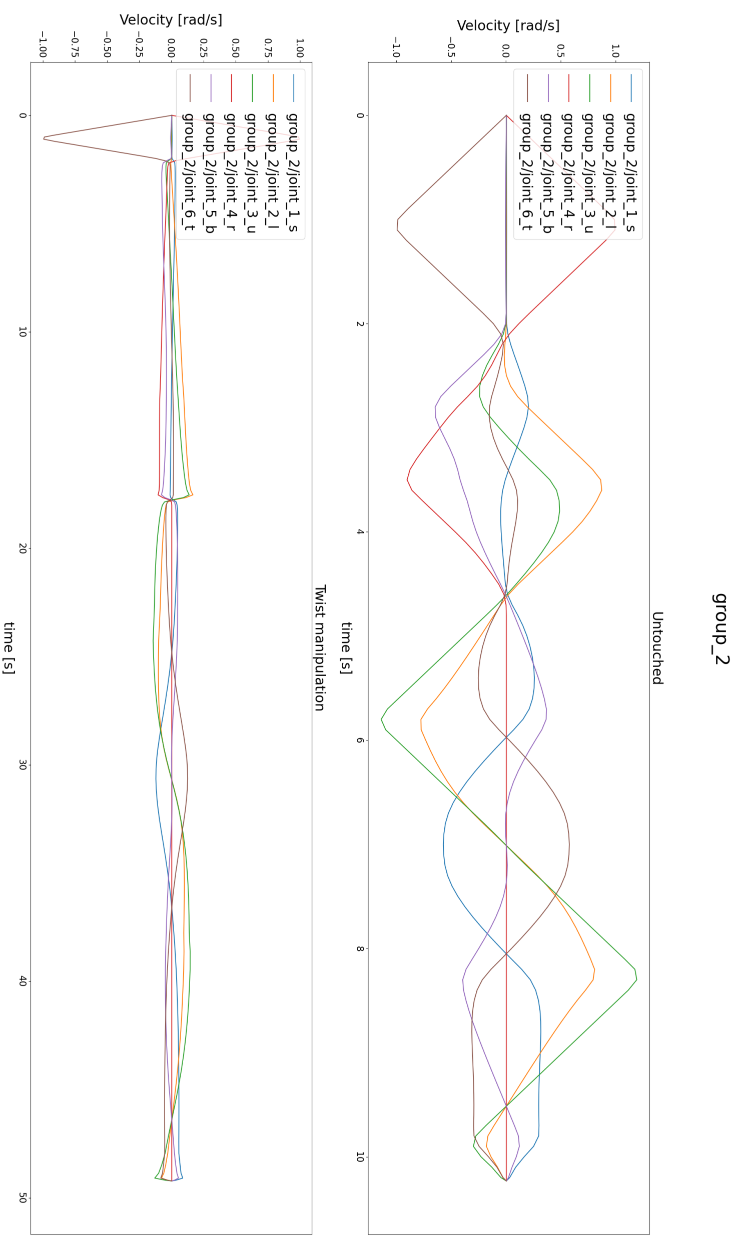


Figure M.3.: Joint velocities for circular path. Untouched (top) and twist (bottom).

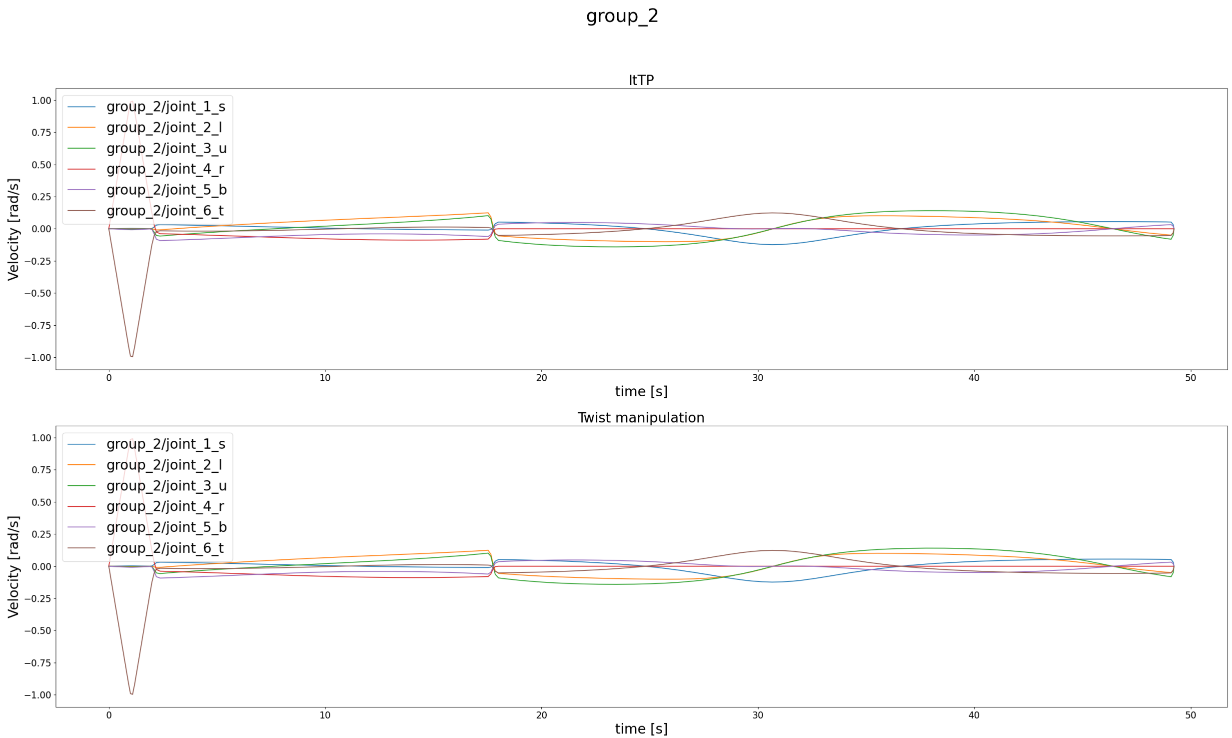


Figure M.4.: Joint velocities for ciruclar path. Iterative time parameterization (top) and twist method (bottom).

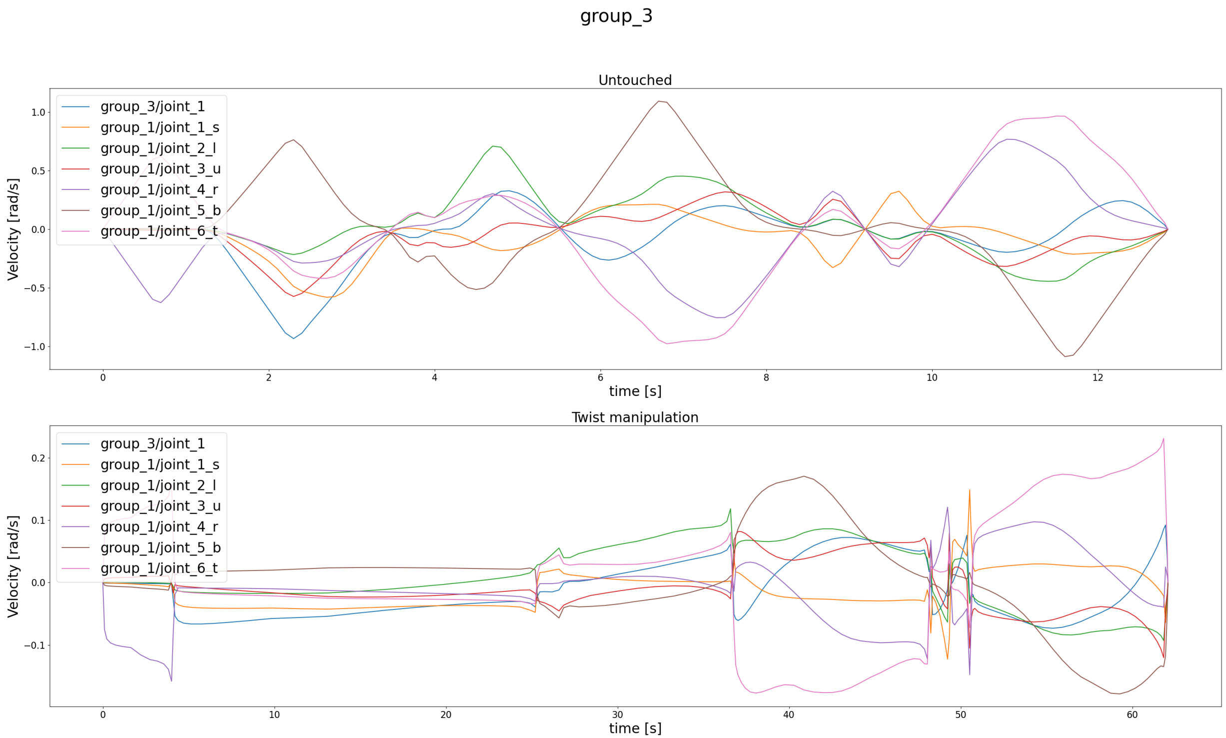


Figure M.5.: Joint velocities for inside motion. Untouched (top) vs twist method (bottom).

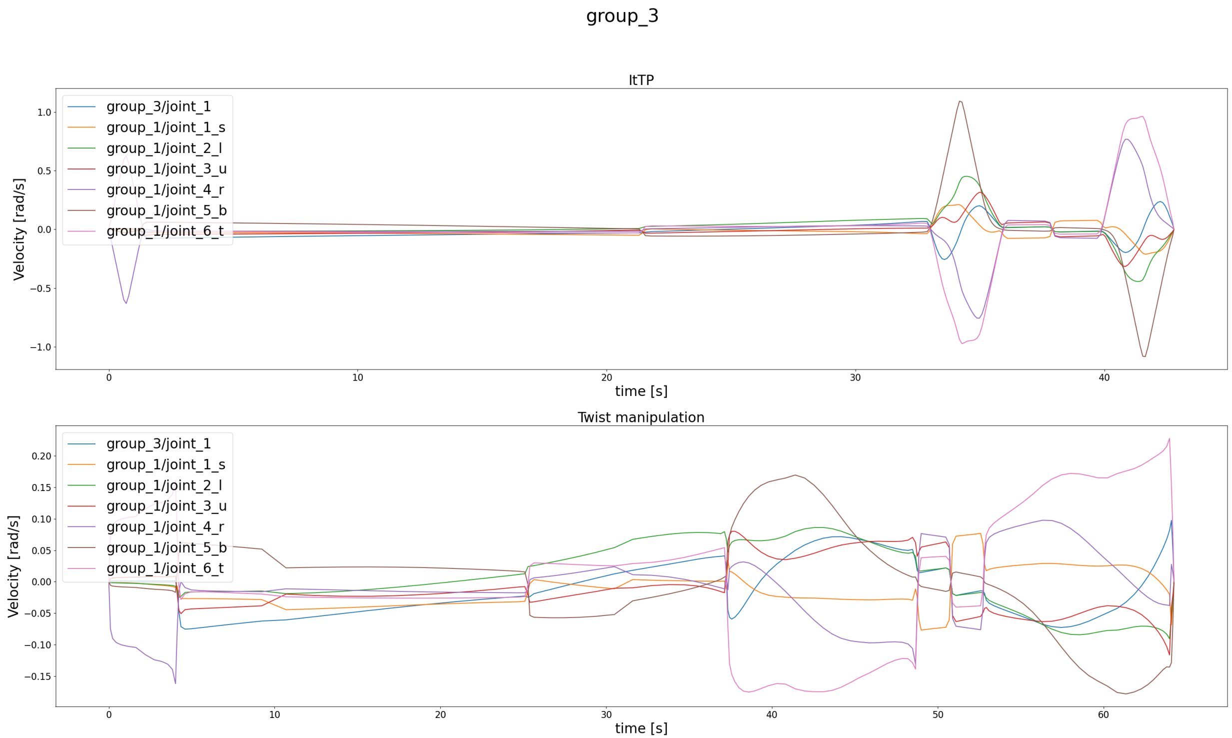


Figure M.6.: Joint velocities for inside motion. Iterative time parameterization (top) vs twist (bottom).

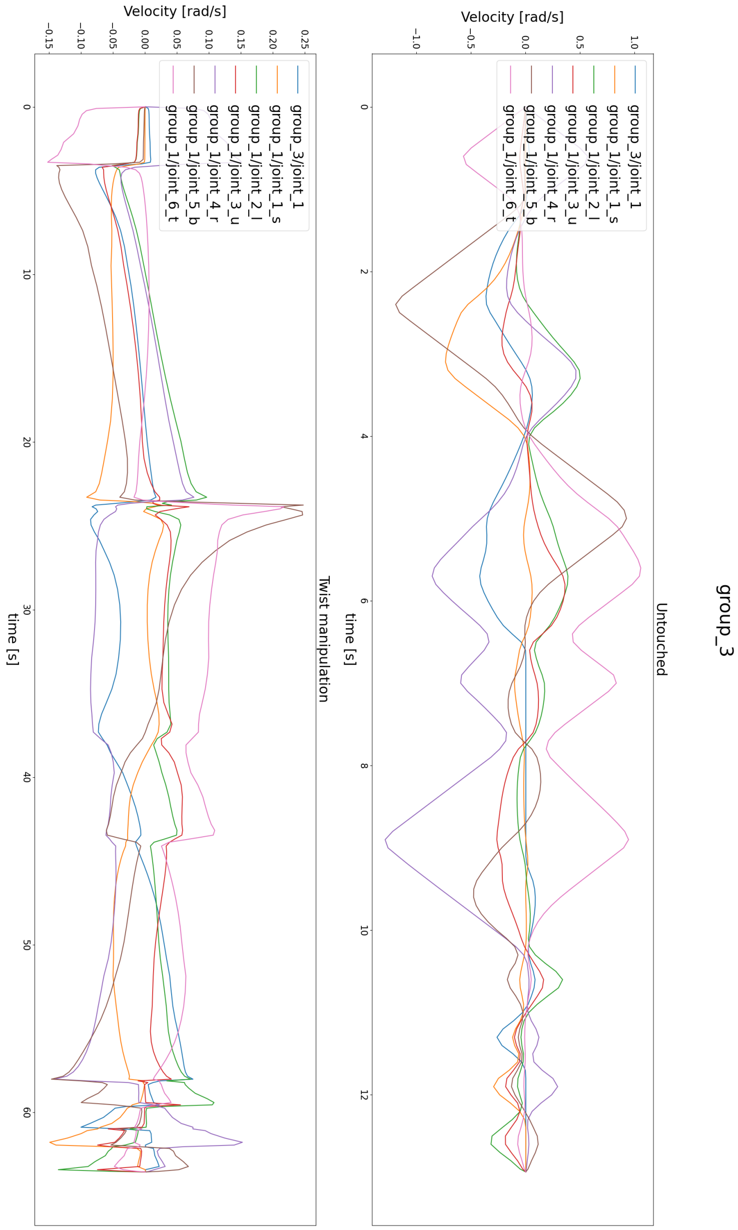


Figure M.7.: Joint velocities for test 4. Untouched (top) vs twist method (bottom).

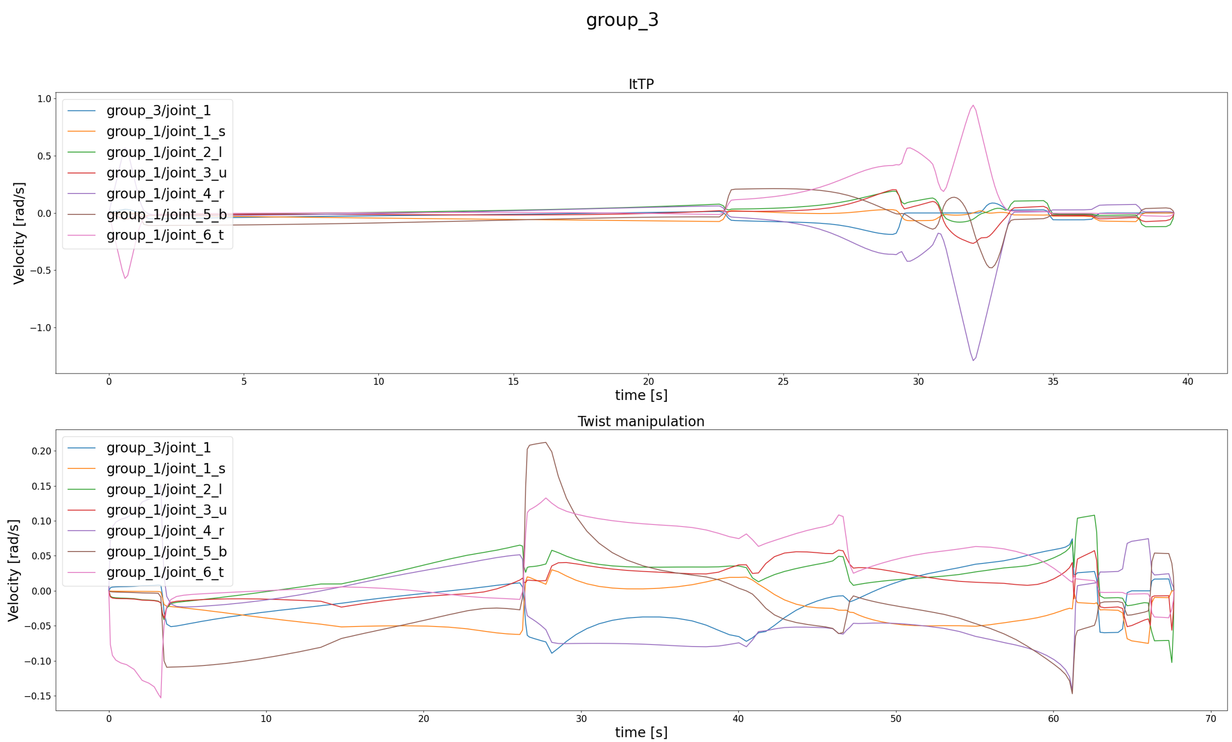


Figure M.8.: Joint velocities for test 4. Iterative time parametrization (top) vs twist method (bottom).

NTNU

Norwegian University of
Science and Technology