Politecnico di Torino

Politecnico di Torino

Mechatronic Engineering
A.a. 2024/2025
luglio 2025

Development of a ROS 2 HW interface for Comau Manipulators

Relatore:
Alessandro RIzzo
Co-relatori:
Andrea Perica from Comau Robotics
Alfio Minissale from Comau Robotics

Candidato:
Edoardo Reina

Table of Contents

List of Figures

List of Tables

Introduction

In recent years, the Robot Operating System (ROS) has established itself as a foundational framework for the development, simulation, and integration of robotic applications. Its flexible and modular architecture enables rapid prototyping, seamless communication between distributed components, and broad compatibility with a variety of hardware platforms. However, as robotic systems grow in complexity and are increasingly deployed in real-world, production-level environments, the limitations of ROS 1, particularly in areas such as real-time performance, security, and support for multi-robot systems, have become more apparent.

To address these challenges, ROS 2 was introduced as a complete redesign of the original framework, incorporating modern middleware, real-time capabilities, improved reliability, and enhanced support for embedded and distributed systems. As a result, migrating legacy ROS 1-based systems to ROS 2 has become an important step for developers and organizations aiming to maintain long-term maintainability, scalability, and performance in their robotics solutions.

This thesis focuses on the development of a ROS 2 wrapper to use Moveit 2 with Comau industrial robots, based on the already present ROS 1 example. The approach involved integrating the MoveIt motion planning framework to enable trajectory generation and execution on Comau robotic systems. The motion planning pipeline was initially validated using RViz, a visualization and simulation tool widely used in ROS development, followed by further testing in Comau's proprietary simulation environment, RoboShop. Final validation was carried out on actual hardware using a Comau e.DO educational robot.

Throughout the testing phase, a range of trajectories with varying levels of complexity were planned and executed, both in simulation and on the physical robot. The results demonstrated reliable performance and accurate execution,

indicating that the developed ROS 2 interface is capable of supporting real-world applications. This work contributes to the broader effort of bringing ROS 2 support to industrial robotics platforms and offers a replicable methodology for similar porting efforts in the future.

Chapter 1: State of the art, robotic manipulators and ROS.

1.1 State of the Art of Industrial Manipulators

[1] [2] Industrial manipulators have undergone a transformative evolution in recent years, becoming more intelligent, flexible, and integrated into modern manufacturing ecosystems. Traditionally used for repetitive and high-precision tasks such as welding, painting, and material handling, today's manipulators are increasingly characterized by their adaptability and ability to operate in dynamic, human-centric environments. A significant trend shaping the current landscape is the rise of collaborative robots, or cobots, manipulators specifically designed to work safely alongside human operators without the need for physical barriers. According to the International Federation of Robotics (2024), cobots now constitute over 11% of all industrial robot installations, reflecting their growing adoption by small and medium-sized enterprises (SMEs) due to their lower cost, ease of programming, and enhanced safety features.

Figure 1.1: A comau racer5-cobot working alongside a human

Major industrial players are increasingly deploying robotic manipulators at scale to achieve higher levels of automation and productivity. For instance, Hyundai's new electric vehicle plant in Georgia employs more than 475 robotic arms and over three hundred autonomous guided vehicles, illustrating how modern manipulators are integral to highly automated and synchronized manufacturing systems [3]. These systems are not only capable of performing physical tasks but are also interconnected with digital infrastructure, forming part of cyber-physical systems that align with the principles of Industry 4.0 and 5.0. The synergy between physical robots and digital twins allows real-time simulation, monitoring, and optimization, significantly improving the efficiency and reliability of production lines.

Advances in artificial intelligence, machine learning, and edge computing have further empowered manipulators to perform complex operations that were previously limited to human dexterity. Modern manipulators are equipped with rich sensor arrays, including 3D vision, tactile sensors, and force feedback, enabling them to perceive and respond to their environments in more sophisticated ways. Research shows that manipulators are increasingly being integrated into intelligent robotic work cells, where AI-driven decision-making enhances flexibility, especially in high-mix, low-volume production scenario.

[4] A critical enabler of these developments is the growing adoption of the Robot Operating System (ROS) and its industrial extension, ROS-Industrial. ROS provides a standardized and modular software framework for building complex robotic systems, facilitating greater interoperability, rapid prototyping, and reuse of software components. ROS-Industrial extends these capabilities to industrial hardware, offering drivers, libraries, and tools tailored to real-world manufacturing use cases. This has significantly lowered the barrier to entry for deploying sophisticated manipulation systems in industry and has also enabled greater collaboration between academia and industrial stakeholders. Case studies have shown how legacy robotic systems have been successfully upgraded to ROS-

based architectures, resulting in improved modularity, diagnostics, and integration with machine vision and planning systems (ScienceDirect, 2024).
In summary, the state of the art in industrial manipulators reflects a convergence of mechanical sophistication, intelligent control, and software-driven customization. The transition from isolated, task-specific robots to flexible, interconnected, and semi-autonomous manipulators is a defining characteristic of current developments. These systems are not only increasing productivity and precision in manufacturing but are also paving the way for a new generation of intelligent automation that is safer, more adaptive, and aligned with the goals of sustainable and human-centric industrial innovation.

As of 2025, the Robot Operating System has solidified its position as a foundational framework in both academic research and the robotics industry. With the advent and maturation of ROS 2, the platform has addressed many of the architectural and functional limitations of ROS 1, evolving into a robust, real-time-capable, and security-conscious middleware for developing scalable and distributed robotic systems. ROS 2 leverages the DDS (Data Distribution Service) protocol to support real-time communication, quality of service (QoS) policies, and multi-platform interoperability, including support for Windows, macOS, and embedded systems. This has opened the door for ROS adoption in industrial and mission-critical applications, ranging from autonomous vehicles and industrial automation to healthcare and service robots.

The ecosystem surrounding ROS 2 continues to expand rapidly, with actively maintained projects such as Navigation 2 (Nav2) for autonomous navigation, MoveIt 2 for robotic manipulation, and micro-ROS for resource-constrained embedded platforms. Integration with modern development tools, containerization (e.g., Docker), and simulation environments like Gazebo and Ignition further support advanced testing, deployment, and continuous integration

workflows. Backed by major stakeholders such as Open Robotics, NVIDIA, Intel, and Tier IV, ROS 2 is now at the centre of many robotics R&D initiatives, benefiting from strong community contributions and commercial investment. As the framework continues to evolve through long-term supported releases and modular architecture, ROS 2 stands as the current state of the art for building adaptable, intelligent, and interoperable robotic systems.

1.2 ROS Communication Infrastructure

[5] To better understand the working of the proposed solution it is vital to understand how the ROS (robotic operating system) framework works; ROS is an open-source middleware for robots, it is not an operating system (it runs on unix systems), but it provides the services you would expect from one, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server. ROS consists of a distributed framework of processes (or nodes), allowing executables to be individually designed and coupled at runtime. These processes can be grouped into Packages and Stacks, which can be easily shared and distributed, as one of the core tenets of ROS is the reuse of the code; for this reason, ROS supports many different programming languages and the code written for it can work with and on other robotics frameworks. The modularity and flexibility provided by this architecture enable developers to incrementally build complex robotic systems by integrating off-the-shelf packages or custom modules, often significantly reducing development time. Additionally, ROS offers

a rich set of tools for debugging, visualization, and simulation, such as rviz for 3D visualization and rosbag for data recording and playback, which facilitate the design, testing, and evaluation of robotic algorithms. Its widespread adoption across academia and industry has led to a vibrant ecosystem and extensive community support, making it an invaluable platform for both research and application development in robotics.

Nodes

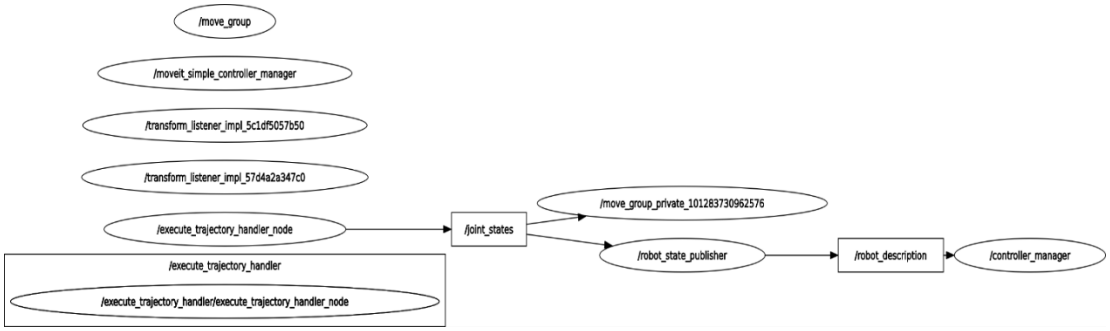


Figure 1.2: a graph showing nodes connected by topics, created with ROS tools

A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service. Each node can have a name and a namespace it belongs to and the name must be unique under the node's namespace; ROS manages a graph of each active node and the topics by which these communicate. The design philosophy behind nodes promotes modularity, as each node can encapsulate a specific functionality, which enhances the reusability and scalability of robotic applications. Nodes can be run individually or in groups, and developers often design systems where nodes are loosely coupled and interact solely through message passing, reducing dependencies and increasing system robustness. ROS provides tools such as rosnode to inspect, list, and manage nodes during runtime, making the debugging and orchestration of systems more accessible.

Topics

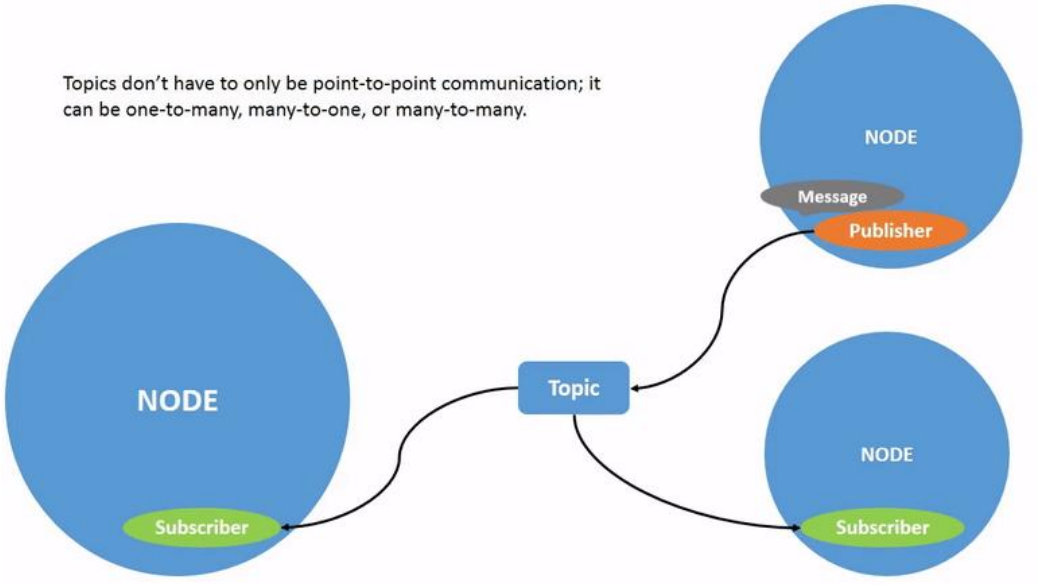


Figure 1.3: A diagram showing topic communication [5]

Topics are named buses over which the nodes exchange messages, they decouple the information production and consumption as nodes are not aware of which other nodes they are communicating with, instead a node can be either a subscriber (which consumes the data) or a publisher (which creates the data), multiple nodes can be subscribers/publishers for the same topic. Each topic is strongly typed, meaning that only one type of message can be published on it and the subscribers will receive the message only if they match the type. This publish-subscribe mechanism allows for real-time, scalable, and flexible communication between components. For example, a sensor node might continuously publish laser scan data on a topic, while several subscriber nodes may simultaneously process it for tasks such as mapping, obstacle detection, or localization. Topics can also be visualized using tools like rqt_graph or monitored using rostopic, which helps in understanding the flow of data and debugging communication issues in complex systems.

Services

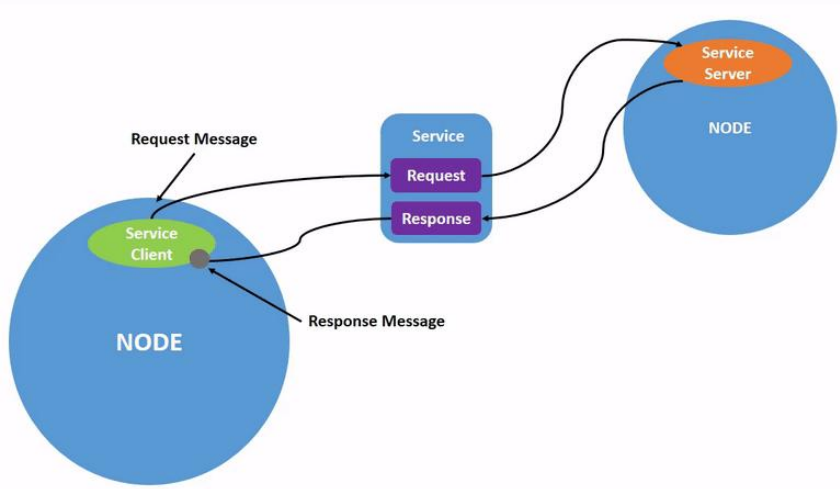


Figure 1.4: A diagram showing service communication [5]

Whereas topics work based on a publisher-subscriber communication model, services are based on a one-to-one transport paradigm, making them suitable for a request/reply communication paradigm. A service is defined by a pair of ROS messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote system call. Services are useful for actions that require a definite response, such as querying a parameter, commanding a robot to move to a specific position, or initializing a process. Although less flexible than topics for continuous data streams, services provide a synchronous and deterministic way of executing specific commands. Tools like rosservice enable users to inspect available services, their types, and to call them manually for testing purposes, enhancing the development workflow.

Parameter Server

The Parameter Server is a shared, multi-variable dictionary that is accessible via the ROS API and is used to store static, non-binary data such as configuration parameters or settings needed at runtime. Parameters are typically loaded at startup and remain unchanged throughout the execution of the system, although

they can be updated during runtime if needed. Nodes can read from, write to, and delete parameters on the server, enabling a centralized and flexible mechanism for configuration management. For instance, tuning parameters for a PID controller or defining robot-specific properties like dimensions or sensor offsets can be managed through the Parameter Server. It is especially useful in dynamic systems where parameters need to be shared across different nodes without hardcoding them. The command-line tool rosparam allows users to set, get, and list parameters on the server, making it easier to adjust system behavior without modifying source code.

1.2 ROS 2 Communication Extensions

First released in 2017, ROS 2 was developed to address some of the architectural and technical limitations of the original ROS framework, particularly in areas such as real-time programming, scalability, and cross-platform support. One of the key differences is that ROS 2 is built on top of the Data Distribution Service (DDS), a middleware protocol that allows for more reliable, secure, and scalable communication, particularly in distributed systems. This shift enables ROS 2 to support a broader range of use cases, including industrial and embedded systems, where deterministic behaviour and strict timing requirements are essential. Furthermore, ROS 2 introduces multi-threaded execution models, enabling improved performance and better utilization of multi-core processors. It also enhances support for multiple programming languages and operating systems, expanding its applicability in diverse development environments. In alignment with its support for real-time systems, ROS 2 introduces a new communication paradigm known as actions, designed to handle complex, long-running operations that require periodic feedback and the ability to cancel execution.

Actions

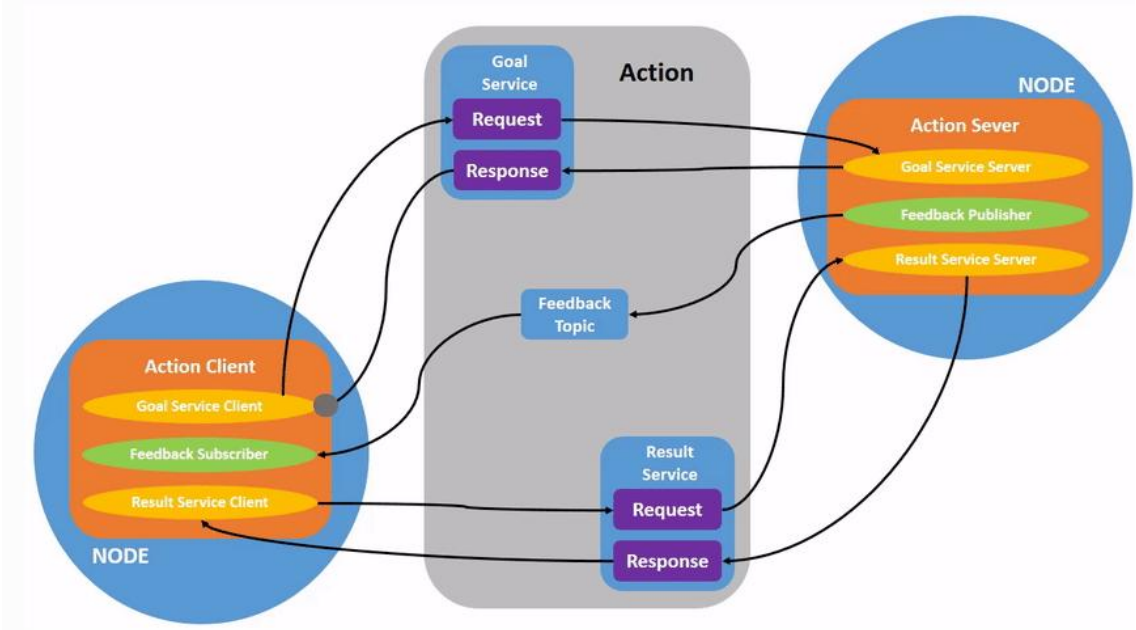


Figure 1.5: A diagram showing Action communication [5]

An action is a communication type introduced in ROS 2, designed for operations that take a significant amount of time to complete and require intermediate updates or the possibility of pre-emption. Unlike services, which follow a simple request-response model, actions consist of three components: a goal, feedback, and a result. When a client sends a goal to an action server, the server processes the goal while continuously publishing feedback messages, allowing the client to monitor progress. Upon completion, the server returns a result message. Actions are implemented using both topics and services, services are used to initiate and conclude the interaction, while topics are used to provide asynchronous feedback throughout the task. This makes actions ideal for tasks such as motion planning, navigation, or any process where continuous monitoring and the option to cancel or replace the goal are critical. The action interface aligns well with the design philosophy of ROS 2, enhancing modularity and responsiveness in real-time applications.

1.4 Comparison Between ROS and ROS 2

While ROS and ROS 2 share the same foundational goal of facilitating the development of robot software through modularity and reuse, they differ significantly in architecture and capabilities. The original ROS (often referred to as ROS 1) was designed primarily for research and academic use, focusing on simplicity and rapid prototyping. It relies on a centralized architecture, with a master node that manages the registration and coordination of other nodes. This model, while effective for small-scale systems, presents limitations in terms of scalability, fault tolerance, and real-time capabilities.

ROS 2, on the other hand, was built from the ground up to overcome these limitations. It adopts a decentralized, peer-to-peer communication model using the Data Distribution Service as its middleware layer, eliminating the need for a master node and enabling native support for multi-host, distributed systems. This architecture greatly enhances scalability and robustness, making ROS 2 more suitable for industrial and mission-critical applications.

Furthermore, ROS 2 introduces support for real-time execution, leveraging features like deterministic message delivery and memory-safe designs, which are essential in environments with strict timing constraints. It also provides enhanced security features, cross-platform compatibility (including real-time operating systems), and improved support for multi-threaded applications. While ROS 1 supports multiple programming languages and operating systems, ROS 2 extends this flexibility by enabling better integration with modern development tools and standards.
In terms of communication paradigms, ROS 2 expands on the original by introducing actions, a construct for handling long-duration tasks that require periodic feedback and pre-emption. Additionally, many of the ROS tools have

been updated or replaced in ROS 2 to align with the new architecture and capabilities.

Despite their differences, ROS 1 and ROS 2 are not entirely incompatible—ROS 1 bridges exist to facilitate migration, and many concepts and APIs remain familiar to developers transitioning between the two versions. However, the structural and functional improvements in ROS 2 mark a significant evolution aimed at meeting the growing demands of robotics in both academic and industrial contexts.

Another important feature introduced by ROS 2 are the lifecycle nodes, a structured approach to controlling the state and behaviour of nodes throughout their execution. Unlike traditional nodes that are either running or stopped, lifecycle nodes pass through well-defined states such as unconfigured, inactive, active, and finalized. This enables better control over system startup, shutdown, and error handling, which is particularly useful in complex or safety-critical robotic applications. By explicitly managing transitions between states, developers can implement predictable and deterministic behaviour, improving system reliability and maintainability.

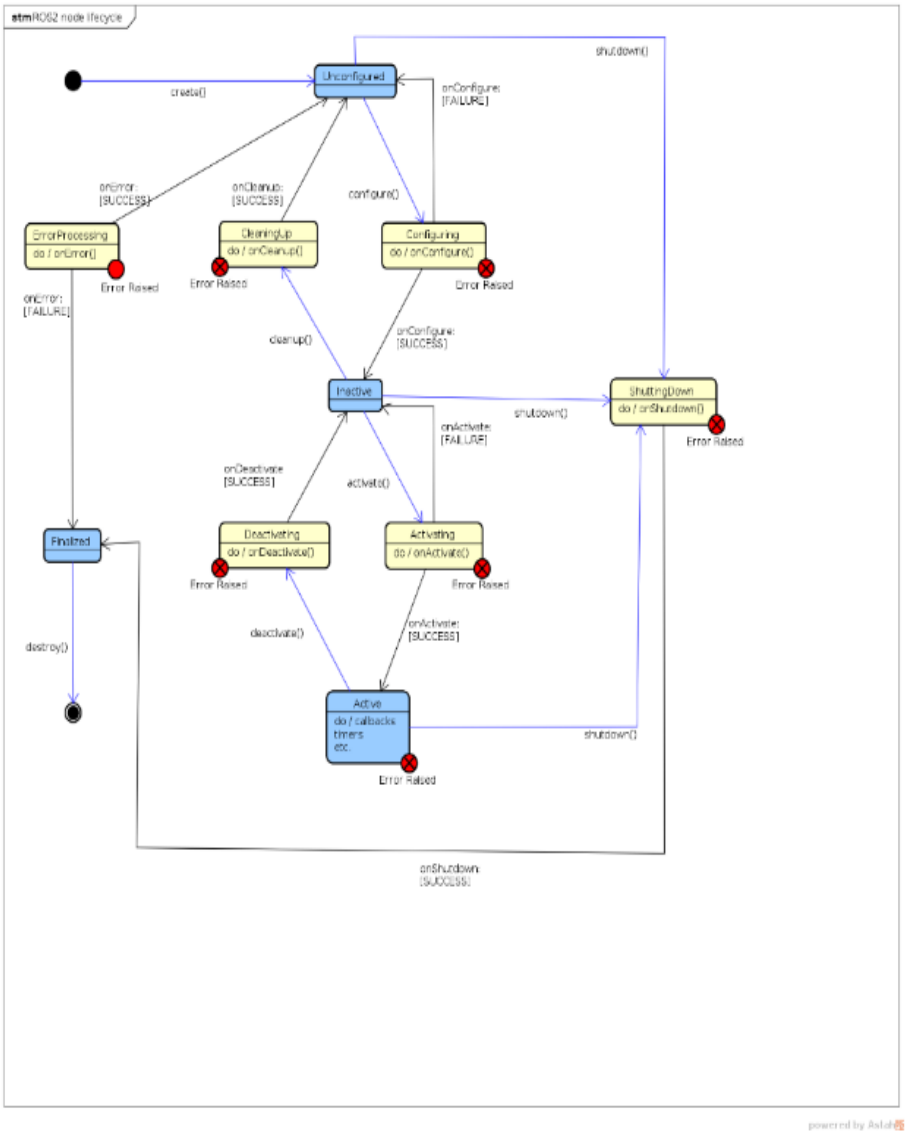


Figure 1.6: The lifecycle of a managed node [4]

1.5 Comau Industrial Manipulators

Figure 1.7: The Comau industrial manipulators family [7]

[7]Comau is a worldwide technological leader in industrial automation and robotics, designing and manufacturing high-performance robotic manipulators for a wide range of industries including automotive, aerospace, and general industrial manufacturing. Renowned for their precision, flexibility, and seamless integration, Comau manipulators allow for extensive customization and adaptability. This enables them to support a variety of tasks, such as welding, material handling, assembly, painting, and pressing, through the use of interchangeable tools. These robots are equipped with advanced control systems and are fully compatible with Industry 4.0 technologies, making them ideal for deployment in smart manufacturing environments.

Comau offers a broad portfolio of robots, divided into different series according to their purpose. For example, the NJ series supports high payloads and large workspaces, making it well-suited for heavy-duty industrial applications. In contrast, the Racer series, with its high-speed performance, is optimized for precision tasks in confined workspaces. All Comau robots are compatible with the company's ROS (Robot Operating System) driver, enabling flexible development across the full range of platforms.

For the purposes of this thesis, two robots were selected for integration and testing: the Racer5 COBOT and the e.DO educational robot. The wrapper was initially developed for the Racer5 COBOT and was later extended to ensure compatibility with additional Comau robots. The e.DO platform was chosen as second due to its suitability for testing on real hardware.

Comau Racer-5 COBOT

*F*igure 1.8: A Comau Racer-5 Cobot [7]

The first robot considered for testing with the new wrapper was the Comau Racer5 COBOT, a collaborative robot that represents a significant advancement in the field of industrial automation. It combines the high-speed capabilities of traditional industrial robots with the safety features required for human-robot interaction. This six-axis articulated robot features a 5 kg payload, an 809 mm reach, and a repeatability of ±0.03 mm.

A key feature of the Racer5 COBOT is its ability to automatically switch between industrial and collaborative modes. When no human operator is present, the robot functions at full industrial Cartesian speeds of 6 m/s. However, when a person is detected nearby, it seamlessly transitions to a safer collaborative speed of 500 mm/s, in compliance with ISO/TS 15066 safety standards. This adaptive

behaviour is made possible by integrated safety systems, including TÜV Süd-certified Safe Collision Detection and environmental monitoring via LiDAR sensors.

The Racer5 COBOT is suitable for a range of applications, including assembly, material handling, machine tending, dispensing, and pick-and-place operations. Its compact design, along with electrical and air connectors located near the wrist, minimizes external cabling and simplifies integration into various workspaces. An additional safety feature includes an LED light strip on the robot, which glows green during collaborative operation and turns off when operating at full speed—providing visual feedback to enhance operator awareness and safety. Furthermore, its ability to operate without protective barriers not only improves space efficiency but also reduces installation and maintenance costs.

Comau e.DO Robot

*F*igure 1.9: A Comau e.Do robot [7]

The second robot adapted for testing was the Comau e.DO, a versatile, open-source educational robot designed to support learning and experimentation in robotics, automation, and STEM-related disciplines. The e.DO is a 6-axis articulated robot with a 500 g payload, a reach of approximately 559 mm, and repeatability of ±0.5 mm, making it particularly well-suited for academic, training, and demonstrative use cases.

A key advantage of the e.DO platform is its rich educational ecosystem, which includes a variety of software applications, modular learning programs, and compatibility with e.DO Experience kits. These tools facilitate hands-on learning in subjects like robot programming, mechanical design, problem-solving, and collaborative teamwork. The e.DO is widely used in schools, universities, makerspaces, and corporate training programs, offering a user-friendly introduction to robotics and automation technologies.

In addition to its educational value, the e.DO can also be applied in basic automation and prototyping tasks, thanks to its flexible I/O ports, vision system options, and compact footprint. Its modular architecture allows it to be easily integrated with other hardware and control systems, making it a practical tool not only for education but also for light industrial applications and research.

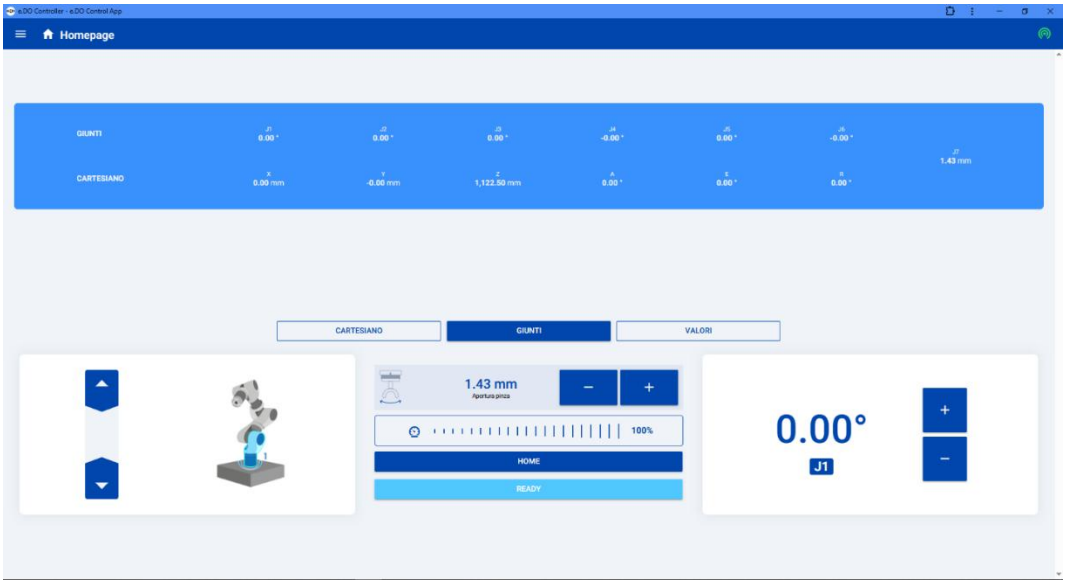


Figure 1.10: The Edo control App

Comau provides a series of apps for the e.Do that is a collection of interactive software packages that are intended to further develop the educational and experiential learning experience offered by the e.DO robot platform. The apps provide intuitive, easy-to-grasp interfaces allowing users to program and operate the robot with no programming know-how, making them ideal for training and classroom applications. The applications cover a wide range of functionalities, from simple movement commands to task scheduling, logic design, and sensor integration and supporting both visual and text-based programming. Through these apps, students can learn basic concepts in STEM, automation, and robotics while teachers can create personalized lessons and challenges for different skill levels. This flexible and modular approach helps fill the gap between practice and theory, confirming the e.DO robot as a valuable tool for learning technology.

Chapter 2: The Comau ROS driver:

[8] To better understand the work carried out in this thesis, it is essential to first analyse the structure of the existing ROS and ROS 2 drivers developed by Comau for their robotic manipulators. In both implementations, the Comau ROS library is divided into two main components.

The first component, the server side, runs on a cabinet IPC and consists of a collection of PDL (Programming Description Language) programs. These programs are responsible for direct communication with the robot, sending motion instructions and reading the feedback data returned by the robot's control system. It is also possible to use the library in virtual mode. In this case, the user only needs a laptop where both Windows and Linux run. The former is necessary to support RoboShop Comau's simulation tool; the latter is used for ROS.

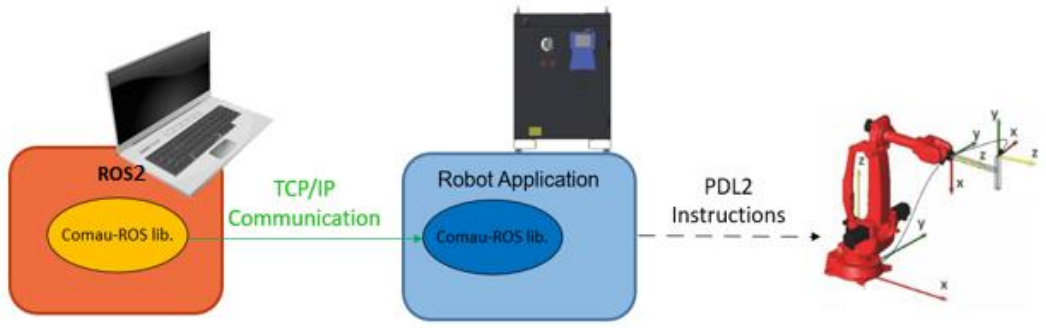


Figure 2.1: A diagram showing the functioning of the Comau ROS driver with a real robot [8]

The second component of the Comau library, and the primary focus of this thesis, is the ROS client module, which operates on an external PC running ROS Melodic, typically on Ubuntu 18.04 or in the case of ROS 2, Humble, typically on Ubuntu 22.04. This side comprises a set of ROS packages that handle various tasks such as motion planning, robot state monitoring, and communication with the server via a TCP/IP channel. Through this channel, the client can also access and control the robot's I/O interface, as well as configure and monitor external

sensors. This modular and distributed architecture enables flexible integration into broader robotic systems and simplifies the development and testing of advanced functionalities.

To start the Comau ROS driver the first step is to launch this command in the terminal:

```
roslaunch comau_bringup <robot_model>_bringup.launch \
 robot_net_config_file:=<robot_net_config_file>
```

where <robot_model> specifies the robot in use and <robot_net_config_file> refers to a file containing the network configuration. Afterwards a service need to be called to connect to the TCP/IP communication channel, like this:

```
rosservice call /tcpip_conn_manager “openConnection: true”
```

Finally, the driver can now be used to send trajectory commands to the robot.

The ROS 2 driver is similarly started by:

2.1 ROS Client Architecture

The following section provides a detailed explanation of the ROS client architecture. Serving as the central interface between the user and the robotic system, this module integrates the essential logic for planning, communication, and control, and forms the core focus of the work carried out in this thesis. As previously mentioned, the client is composed of multiple ROS packages, which are then instantiated as nodes, each with its own dedicated functionality.

Two of these packages are not directly associated with any ROS nodes but serve a foundational role by holding data used by the rest of the system. In particular, the comau_description package contains the URDF models and launch files for each robot supported by the driver, while the comau_msgs package defines the custom ROS messages, services, and actions employed throughout the system.

The first step in utilizing the client is to launch the bring-up package. This package uploads the selected robot's description to the ROS parameter server and initiates the main nodes of the driver, each with a specific role. These nodes include:

- **ComauRobot Node**: This node sets all necessary parameters defined in the configuration files and launches three key nodes required for TCP/IP communication:
    - robot_client, used for setting I/O and configuring sensors.
    - arm1_client, responsible for sending trajectories to the robot.
    - state_client, which monitors the robot's state, position, and feedback messages.

These communication nodes are defined in the comau_tcp_interface package, while the ComauRobot node itself is implemented within the comau_driver package.

- **HardwareInterface Node**: The hardware interface is a central abstraction layer in ROS, designed to decouple the robot's physical hardware from the software controllers that operate it. Rather than having controllers directly interact with low-level hardware such as motors or sensors, the hardware interface provides a standardized method for reading sensor data and sending actuator commands. This abstraction makes controllers hardware-agnostic and highly reusable. By supporting consistent interfaces for position, velocity, or effort control, the same control logic can be applied across different robotic platforms as long as they implement the corresponding interface.
- **controller_manager Node**: This node advertises the ControllerWrapper service, which allows users to switch between available controllers at runtime. By changing the active controller, the robot's operational mode

can be adjusted, defining how it executes motions. This dynamic control selection is not yet implemented in the ROS 2 version of the client. In ROS, controllers are modular software components that implement specific strategies to control robot actuators. They run on top of the hardware interface, processing sensor inputs, such as joint positions or velocities, and generating output commands, like desired joint targets. Thanks to their modularity and independence from hardware specifics, controllers can be reused across a wide range of robots that comply with the standard interfaces.

The ControllerWrapper service is defined as follows:

```
# The constants
uint8 LIST = 0                          # Returns the current mode
uint8 ASYNC_TRAJECTORY = 1              # Used for Pass through trajectories (real only)
uint8 SYNC_TRAJECTORY = 2               # Robot controller - Movefly
uint8 SENSOR_TRACKING_RELATIVE = 3      # Sensor Tracking relative (real only)
uint8 SENSOR_TRACKING_ABSOLUTE = 4      # Sensor Tracking absolute (real only)
uint8 MOVEIT_JOG = 5                    # MoveIt Jog (sim only as of now)
uint8 UNKNOWN = 6                       # Unknown set of controllers active

# The service call mode to switch to
uint8 Mode
---
bool SwitchOk       # Switch successfull
uint8 CurrentMode   # The current mode
bool Ok             # Service call succesfull
```

- **Trajectory Handler Nodes**: Two additional nodes handle trajectory execution: one for Cartesian trajectories and another for joint-space trajectories. These nodes expose action servers that accept asynchronous trajectory commands from ROS action clients. However, they only function when the joint_state_controller is the only active controller, which is the default state of the hardware interface. If the robot is in a ready state and the async_enable topic is set to true, the action servers will send the goal

to the real robot for execution; otherwise, they will abort. Once the arm1_handler PDL program is started, a trajectory can be sent to the ROS action server for execution. The action definitions can be found in the comau_msgs package. Although actions are native to ROS 2, the ROS 1 version of the driver uses a supporting library to implement them.

The execute trajectories used by the actions are defined as arrays of custom messages:

```
#goal definition
comau_msgs/CartesianPoseStamped[] trajectory # desired cartesian positions to
move to
---
#result definition
comau_msgs/ActionResult action_result
---
#feedback
comau_msgs/ActionFeedback action_feedback
```

```
#goal definition
comau_msgs/JointPose[] trajectory # desired cartesian positions to move to
---
#result definition
comau_msgs/ActionResult action_result
---
#feedback
comau_msgs/ActionFeedback action_feedback
```

The cartesian one is called CartesianPoseStamped and is thus defined as:

```
## Definition: A euler pose with a tf frame to transform the pose relative
from.
## If the frame is "" will not transform the pose
Header header
float64 x
float64 y
float64 z
float64 roll
float64 pitch
```

```
float64 yaw

float64 lin_vel
uint64  seg_ovr
string  move_type
```

where:

- header is the reference frame
- {x, y, z, roll, pitch, yaw} is the goal pose of the node in [m, rad]
- lin_vel is the maximum linear velocity that the robot can reach during the execution of the node in [m/s] (default default_linear_velocity m/s in config.yaml file. Always check the limit value $LIN_SPD_LIM)
- seg_ovr is an integer value and it represents the override of the node (default 100)
- move_type is a case-insensitive string which defines the type of robot movement to reach the node.

The join one is instead called JointPose and is defined as:

```
float64[] positions

uint64  seg_ovr
string  move_type
```

where:

- positions is an array of float which contains the joints position in radiants.
- seg_ovr is an integer value and it represents the override of the node (default 100)
- move_type is a case-insensitive string which defines the type of robot movement to reach the node.

Another operating mode supported by the Comau ROS driver is sensor tracking. In robotics, sensor tracking refers to the use of real-time feedback from internal or external sensors to dynamically adjust a robot's motion. This capability is typically used to follow moving targets, compensate for drift, or adapt to environmental changes that occur during task execution. Within the Comau ROS framework, sensor tracking enables the robot to adjust the position of its end-effector either relatively or absolutely, based on Cartesian feedback.

The sensor tracking mode is implemented through two dedicated controllers:

Relative Tracking Controller: This controller accepts velocity commands in the form of geometry_msgs/TwistStamped messages via the /arm_cmd_vel topic. This controller will transform the velocity message into cartesian correction based on the control loop frequency of the hardware interface and send the command to the robot controller.

Absolute Tracking Controller: This controller receives Cartesian position commands via the /arm_cmd_pos topic. Instead of applying incremental adjustments, it sends explicit target positions to the robot, guiding the end-effector to reach specified poses in space.

Key parameters for configuring sensor tracking—such as sensor type, gain values, unit conversion factors, and translational or rotational limits—can be defined statically in the controllers.yaml file. Additionally, these parameters can be updated dynamically at runtime using a ROS service call to /set_sensor_tracking_params, allowing for adaptive tuning of the controller during operation.

The driver also supports teleoperation during sensor tracking. Two dedicated scripts, arm_vel_teleop and arm_pos_teleop, enable manual control of the robot through keyboard inputs, sending either velocity or position commands depending on the tracking mode.

Lastly, the system provides real-time plotting capabilities through the /sensor_tracking_controller/plot topic. This feature logs the target values, actual

positions, and tracking errors across all six degrees of freedom: x, y, z, roll, pitch, yaw.

2.2 Examples of applications

Pick and Place

Pick and place operations are essential robotic tasks widely used in industrial automation, where robotic manipulators are programmed to identify, grasp, and transfer objects from one location to another with high precision and speed. These operations typically involve several coordinated steps, including object detection, positioning of the robotic arm, secure gripping using end-effectors such as mechanical grippers or vacuum suction devices, and accurate placement at a designated target area. Industrial manipulators performing pick and place tasks are often integrated with vision systems and advanced sensors, enabling them to handle objects of varying shapes, sizes, and orientations, even in dynamic or cluttered environments. This adaptability is crucial in sectors such as electronics assembly, food packaging, pharmaceuticals, and automotive manufacturing, where consistency, speed, and reliability are vital.

Figure 2.2: A Racer5-Cobot performing a pick and place operation [7]

The Comau ROS driver supports pick and place operations through multiple control strategies. For dynamic or sensor-driven tasks, the robot can use sensor tracking, either in relative or absolute mode, to adjust its motion in real-time based on feedback from external sensors or vision systems. This allows the robot to follow or respond to moving targets, such as items on a conveyor belt. Alternatively, for more structured environments, asynchronous or synchronous controllers can be employed. These allow the robot to move to the item's location, secure it with the end-effector (e.g., gripper or suction tool), and then follow a planned trajectory to the designated placement location

Manufacturing

Manufacturing tasks such as welding, painting, are some of the most common in industrial settings. These tasks require extremely precise movement, consistent

speed, and careful coordination with tools and sensors. A welding robot, for example, must follow a seam with accuracy while maintaining a steady torch angle and travel speed. Painting requires smooth, continuous motion to ensure

Figure 2.3: Comau robots of the NJ series working on a welding task. [7]

even coating without drips or overspray.

The Comau driver can handle such tasks thanks to its many operational modes and controllers, once again, sensor tracking can be used to handle the feedback from the system while the other trajectory handlers can be used to plan set paths while keeping the end effector in the correct orientation.

Assembly

Robotic assembly is one of the most complex and demanding tasks in industrial automation. Unlike welding or painting, which often follow predefined paths, assembly typically involves interacting with parts of varying shapes, sizes, and positions, requiring the robot to be both precise and adaptive. A typical assembly operation might involve multiple sequential steps such as picking a component from a bin, orienting it correctly, inserting it into a tight-fitting space, and fastening it with a tool. Each of these steps demands accurate motion, but also coordination with external tools like grippers, screwdrivers, or pneumatic actuators. In many cases, vision systems are used to detect the exact position and orientation of

parts, allowing the robot to adjust its motion in real time. Force and torque sensors may also be employed to guide insertion processes or detect misalignment. Thanks to the ROS driver, robots can integrate sensor feedback into motion planning and control loops, enabling them to make decisions on the fly, such as retrying a failed insertion, re-aligning a part, or reporting errors for human intervention. As a result, robotic assembly systems are becoming increasingly capable of handling not just repetitive tasks, but also variant-rich, precision-critical operations that were once considered too unpredictable for automation.

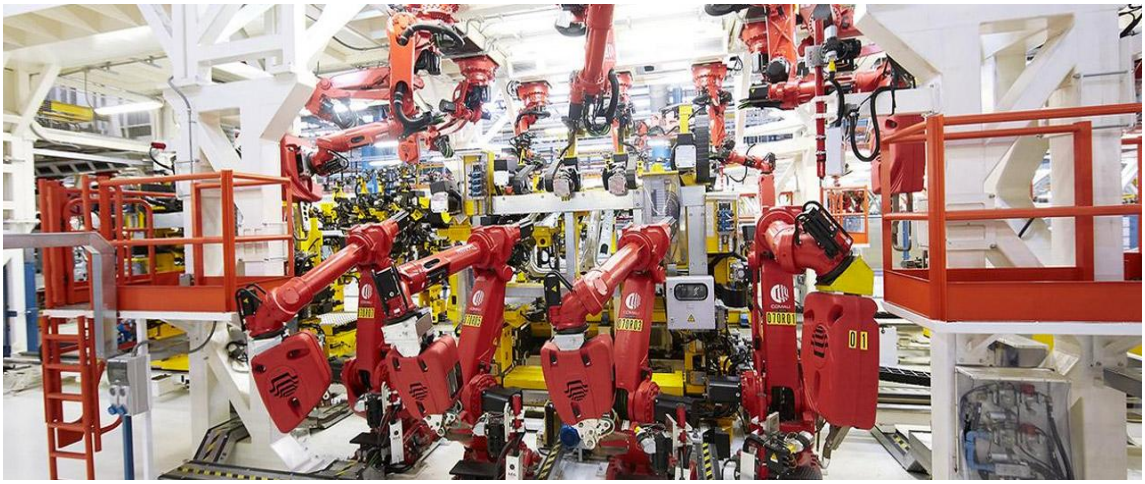


Figure 2.4: An assembly line made of Comau NJ Robots [7]

Chapter 3: The implementation

Prior to the work carried out in this thesis, the ROS 2 driver for Comau robotic manipulators was already under development. However, the driver was still in its early stages and provided limited functionality compared to the more mature ROS 1 implementation. In particular, key features such as motion planning, trajectory simulation, and seamless integration with the broader ROS 2 ecosystem were lacking or incomplete. To address this gap, the primary goal of this thesis was to implement a wrapper that enables the use of MoveIt 2, a motion planning framework for ROS 2, as both a simulation tool and a trajectory planner, integrated into the existing Comau driver.

3.1 Moveit

[6] MoveIt 2 is the ROS 2 version of the widely used MoveIt motion planning framework. It represents a substantial redesign of the original MoveIt system developed for ROS 1, aiming to fully exploit the features introduced in ROS 2. These include improved middleware based on DDS, enhanced support for real-time and deterministic behaviour, lifecycle-aware node architecture, and a modular, scalable structure more suited to modern robotic applications.

MoveIt 2 provides a comprehensive set of tools and libraries for robotic arm manipulation, including components for motion planning, inverse kinematics, collision detection, trajectory generation and execution, as well as planning scene management. These tools make MoveIt 2 one of the most complete and flexible solutions available for robotic manipulation tasks in the ROS 2 environment. Importantly, MoveIt 2 has been designed to integrate smoothly with ros2_control, the ROS 2 control framework, enabling trajectory execution on both simulated and real hardware.

A central component of any MoveIt 2 setup is the move_group node. This node acts as the primary interface between the planning system and external clients, whether they are user interfaces such as RViz2, or custom applications written in Python or C++. In ROS 2, the move_group node is implemented as a lifecycle-aware node, which allows for more robust and controlled startup, shutdown, and runtime behaviour. It exposes a unified set of ROS 2 services and actions used for executing core functions, including motion planning, inverse kinematics, and trajectory control.

Specifically, the move_group node provides functionality for:

- **Motion Planning**: It handles planning requests such as moving the robot's end-effector from one pose to another, utilizing motion planners like OMPL, STOMP, or CHOMP.
- **Inverse Kinematics**: It computes the necessary joint angles for achieving a given pose in Cartesian space.
- **Collision Checking**: It ensures that the planned motions are free from self-collisions and collisions with objects in the environment.
- **Trajectory Execution**: It sends generated trajectories to the robot controller, typically using ros2_control as the middleware interface.
- **Planning Scene Management**: It monitors and updates the robot's understanding of its environment, incorporating dynamic obstacles or changes in the workspace.

To begin using MoveIt 2 with a robot, it is first necessary to create a dedicated MoveIt configuration package. This is efficiently accomplished using the MoveIt Setup Assistant, a setup wizard that allows users to load the robot's URDF (Unified Robot Description Format) file and interactively generate the configuration needed for motion planning. The output includes all necessary files and parameter definitions, such as the SRDF (Semantic Robot Description Format), joint limits, kinematic solvers, planning groups, and controller configurations.

After launching the move_group node, the user can interact with MoveIt 2 through various interfaces. One of the most common is RViz2, which provides an interactive 3D visualization of the robot and the planning scene. Using RViz2, users can manipulate end-effector targets, preview planned trajectories, and initiate execution. Alternatively, developers can use the C++ interface (moveit_cpp) or the Python interface (moveit2_commander) to integrate planning and control directly into their applications.

Although not all plugins from MoveIt 1 have been fully ported to ROS 2, the majority of core functionality is stable and actively maintained by the MoveIt community. MoveIt 2's compatibility with ROS 2 standards and its support for modern robotics requirements make it a powerful tool for both research and industrial applications. The integration of MoveIt 2 into the ROS 2 Comau driver—as implemented in this thesis—adds critical capabilities such as motion planning, trajectory simulation, and real-time execution, substantially enhancing the driver's usability and functionality.

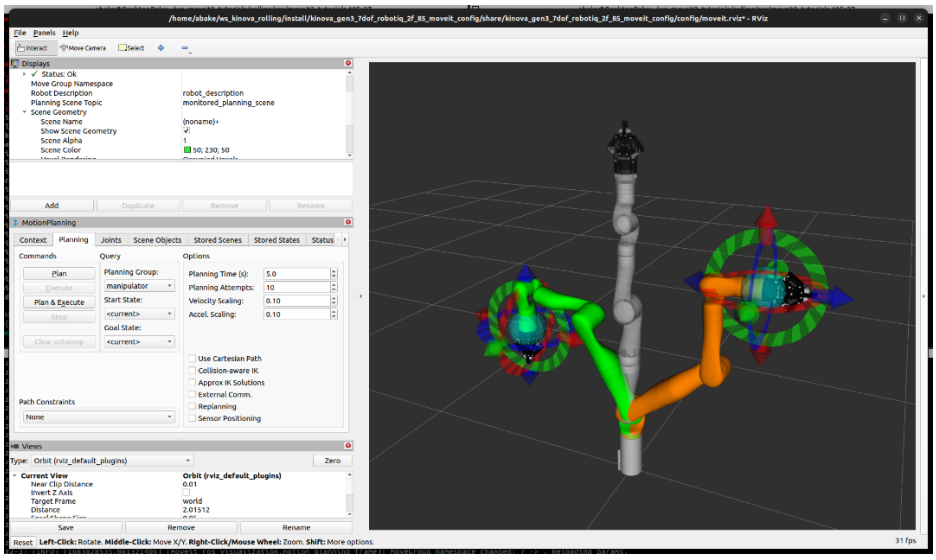


Figure 3.1: Rviz being used to generate a trajectory with Moveit [6]

3.2 First tests with Moveit

The initial step in creating a wrapper for Comau robots that would integrate with the existing ROS 2 driver involved making MoveIt 2 compatible with these robots. To accomplish this, development began using the ros2_control_demo_example_7 package, an official example designed to demonstrate how ros2_control works with a six-degree-of-freedom (6-DoF) manipulator. The robot model included in the example was replaced with the Comau Racer5 Cobot, by substituting the existing URDF with the one provided in the comau_description package. This approach allowed testing and experimentation with the robot's hardware interface without needing to launch the entire driver stack, thereby simplifying development and reducing complexity.

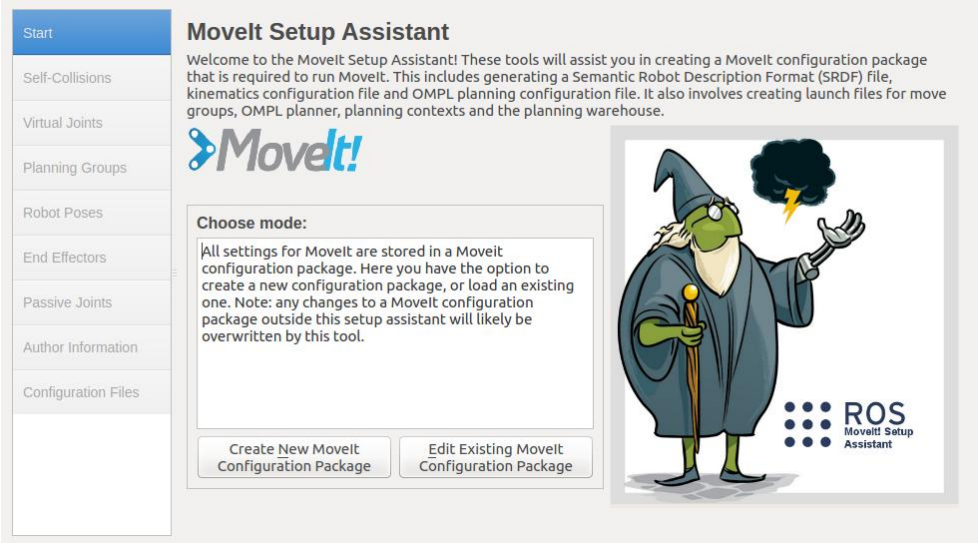


Figure 3.2: The Moveit setup assistant [6]

Once a basic test environment was established, the next phase involved generating a MoveIt 2 configuration package tailored to the Racer5 Cobot. Using the MoveIt Setup Assistant, the URDF of the cobot, manually edited to remove xacro macros for compatibility, was loaded to generate the necessary planning configuration. Several key parameters were specified during this setup process:

a collision matrix, which defines pairs of robot links that are known never to collide and thus can be excluded from collision checking; a planning group, which groups joints together for coordinated trajectory planning; and two controllers, which are responsible for executing planned motions. The OMPL planner was selected as the default motion planner, and an inverse kinematics solver appropriate for the robot's structure was chosen. Upon completion, the setup assistant generated a full configuration package named racer5_cobot_moveit_config. This package included the SRDF (Semantic Robot Description Format) file, kinematic limit definitions, controller configuration files (YAML), and the necessary launch files to start the move_group node and associated MoveIt components, including a standalone demo launch.

This setup enabled initial testing of MoveIt 2 with the Comau Cobot. Using RViz2, it became possible to plan and visualize motion trajectories interactively, as well as to execute them in simulation. Additionally, the configuration included auto-generated C++ demo applications that could be used to test basic planning and execution pipelines programmatically.

The final and most critical stage of this integration involved combining the previous steps to enable MoveIt planners to work in conjunction with the ros2_control_demo_example_7 hardware interface, replacing the simple KDL-based planner previously used. To achieve this, an in-depth understanding of the MoveIt Motion Planning C++ API was necessary. The API offers a modular and extensible interface for defining and solving motion planning problems within custom applications. The typical workflow starts by loading the robot's URDF and SRDF using the RobotModelLoader, then creating a RobotState object and a PlanningScene that reflects the robot's state and environment. Through MoveIt’s plugin-based architecture, planners such as OMPL and STOMP can be dynamically loaded and configured at runtime.

A crucial part of the Motion Planning API is the MoveGroupInterface, which provides a high-level abstraction for setting start and goal states, invoking

planning algorithms, and executing trajectories. This interface connects directly to the move_group node and allows developers to generate motion plans either in joint space or in Cartesian space with minimal overhead. After generating a trajectory using MoveIt, the final step was to extract the resulting trajectory message and redirect it through the appropriate ROS 2 topic (e.g., /joint_trajectory) to the existing controller defined in the example_7 hardware interface.

By successfully executing trajectories via this pipeline, originating from MoveIt and terminating in the existing ros2_control infrastructure, it became possible to confirm that MoveIt could be fully integrated into the Comau ROS 2 driver. This laid the foundation for future development involving real-time planning, sensor feedback integration, and higher-level robotic applications using Comau manipulators within the ROS 2 ecosystem.

3.3 Virtual Tests on Roboshop

The next phase of the development process involved integrating the MoveIt-based wrapper into the Comau driver. To achieve this, a new trajectory handler package was created, designed specifically to employ the MoveIt planner for generating trajectories to be executed by the robot.

To facilitate this integration, new launch files were created, and the comau_bringup package was updated. These modifications allowed the launcher to load all necessary MoveIt resources, including the URDF and SRDF files, kinematic limits, planner configuration files, and a parameter specifying which planner to use.

The first handler to be adapted was the one managing joint trajectories. The existing logic responsible for validating joint positions and interpolating between them was removed and replaced with the MoveIt trajectory planner. Concurrently,

a new action interface was defined: ExecuteJointTrajectoryMoveIt, within a newly created package called wrapper_msgs. The server for this action was implemented in a revised joint trajectory handler node, functioning similarly to the original version. It accepts an array of poses as a goal, leverages the MoveIt API to generate a valid trajectory through those poses, and transmits the resulting trajectory to the robot via a TCP/IP channel.

This new implementation was initially tested with single-point joint trajectories. To launch the updated wrapper, the same command used for starting the original system could be used (start_comau_client.launch.py from the comau_bringup package) by selecting the MoveIt-enabled robot type instead of the default. The new handler remains compatible with the same set of services, including TCP communication. Additionally, to enable trajectory planning, the move_group node must be launched in a separate terminal. Once the virtual robot server in Roboshop is running and a TCP/IP connection is established, the robot is ready to execute incoming trajectory commands.

After verifying that the new handler functioned correctly with single-point trajectories, the next step was to test multi-pose trajectories. This extension required minimal code changes. With the joint trajectory handler completed, development turned to the Cartesian trajectory handler. Thanks to MoveIt's native support for both joint and Cartesian goal types, along with built-in support for path constraints, this transition was relatively straightforward. Notably, all trajectories generated by MoveIt are in joint space, eliminating the need for manual inverse kinematics calculations in the code. Once completed, the Cartesian handler was also subjected to more demanding trajectory scenarios.

The first complex trajectory tested on the Racer5 Cobot was an hourglass-shaped path, illustrated in the accompanying figure. This trajectory was defined in joint space. The necessary joint values were determined by manually positioning the robot in Roboshop and writing down the corresponding joint states.

These were (other joints were positioned at 0, throughout the whole trajectory):

Table 3.1: The written down positions for the hourglass trajectory

	Joint 1	Joint 3
Point A	120°	-45°
Point B	120°	45°
Point C	60°	45°
Point D	60°	-45°

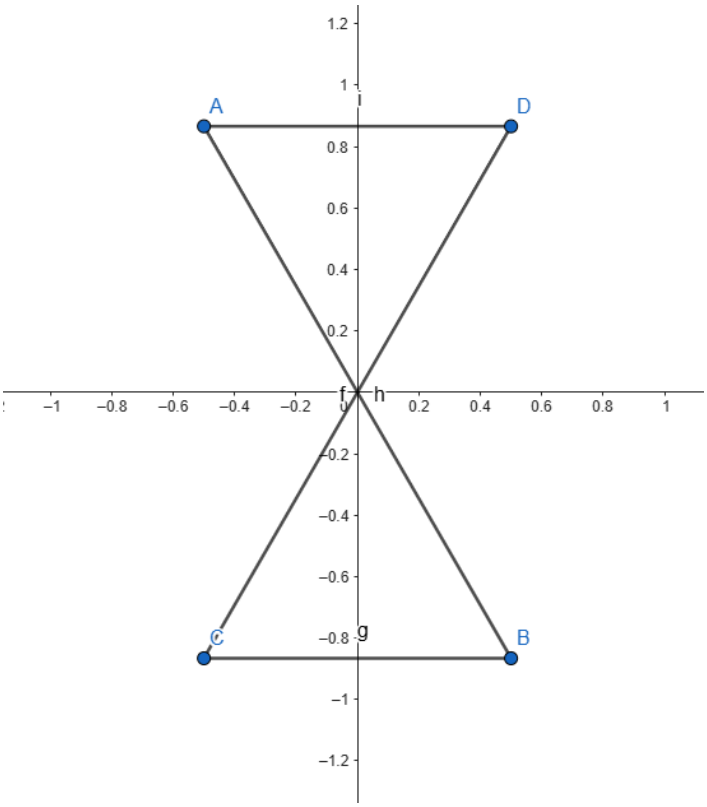


Figure 3.3: The shape of the desired trajectory

.

Using ROS 2's ros2 bag feature, the joint values and their time evolution were recorded. These logs were then processed using MATLAB's dedicated ROS bag analysis package. The generated plots were compared against the trajectory produced by MoveIt, which was saved in a JSON format. Roboshop's

visualization tools were also employed to compare the executed trajectory with the intended path.

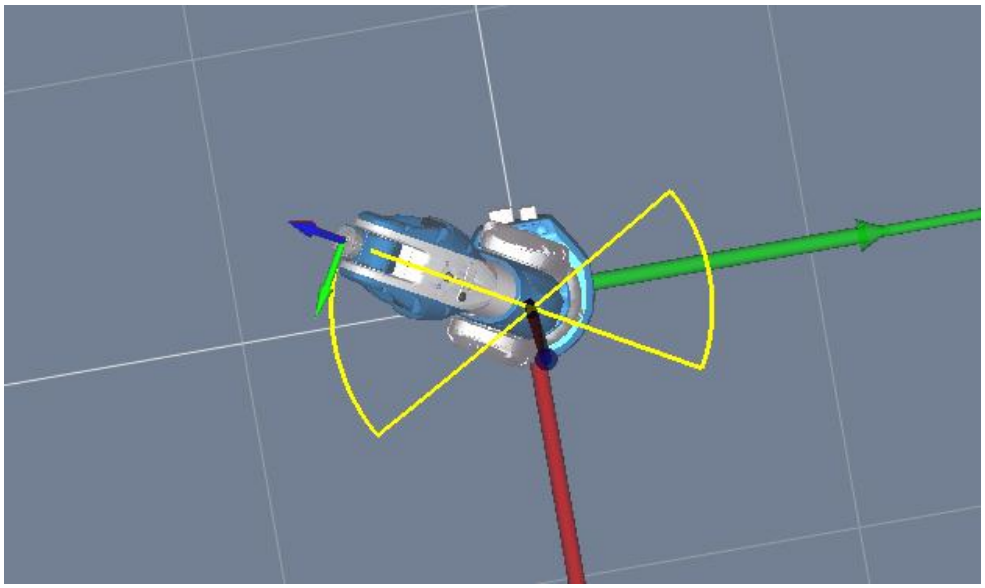


Figure 3.4: The hourglass trajectory simulated on Roboshop

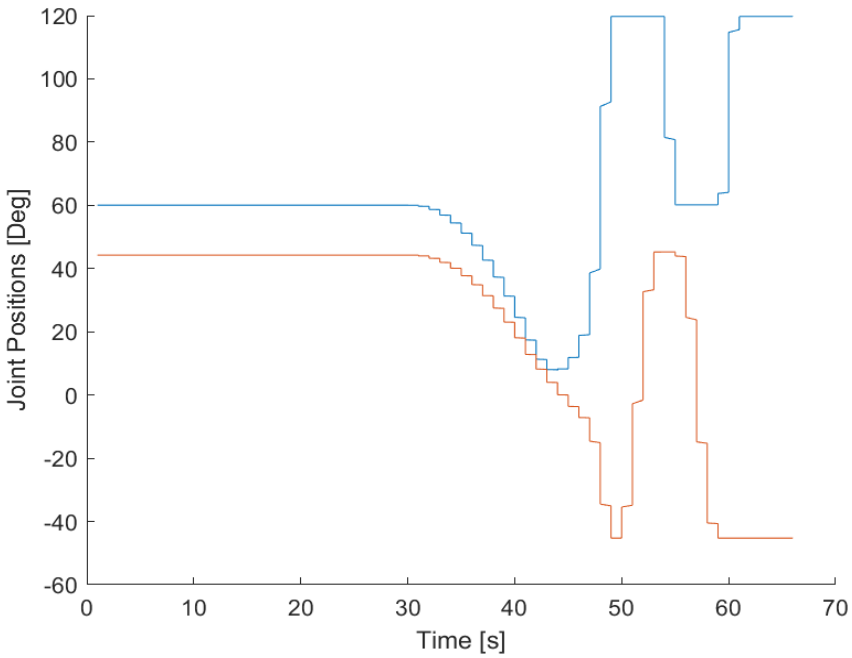


Figure 3.5: The positions of the joints (blue for joint 1 and orange for joint 3) recorded by rosbag

The recorded trajectory was then compared to the one generated by Moveit:

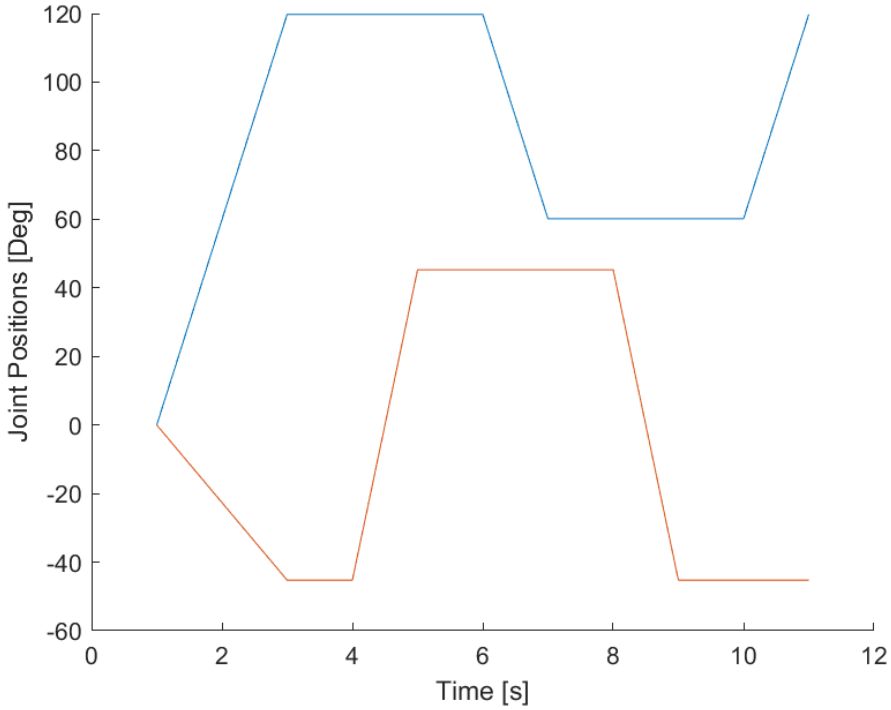


Figure 3.6: The positions of the joints generated by moveit

The results showed a high degree of alignment between the planned and executed trajectories. Joint values from the simulation closely matched those from the MoveIt-generated plan, and the system's response was deemed both fast and reliable.

The next trajectory tested on the Cobot followed a Cartesian triangular path. As before, the tool center point positions were calculated using Roboshop. Initially, the performance of Cartesian trajectories was less satisfactory compared to joint-space planning, an expected result, as MoveIt is generally more optimized for

joint space. However, the situation improved significantly with the appropriate application of path constraints.

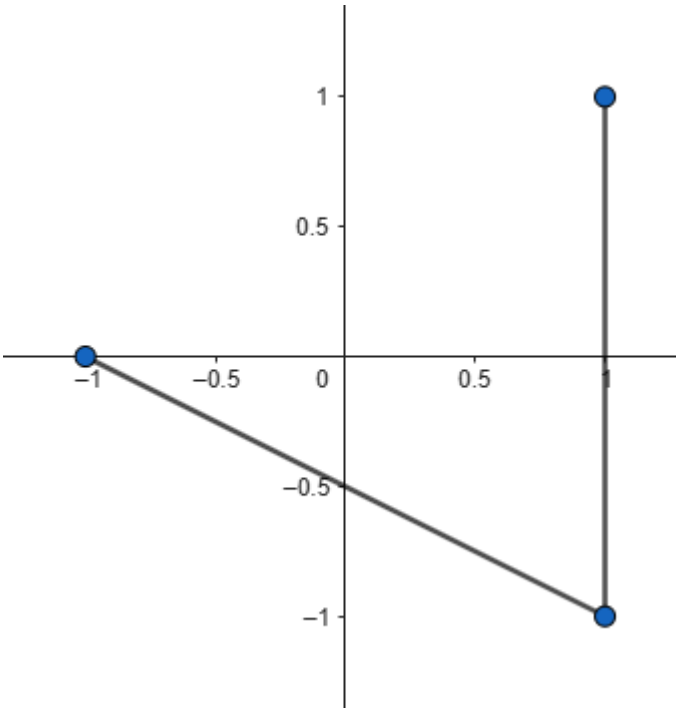


Figure 3.7: The shape of the desired triangular shape

Roboshop visualizations confirmed that the robot followed the desired trajectory with acceptable precision. These results validated the use of the new wrapper for Cartesian trajectories as well.

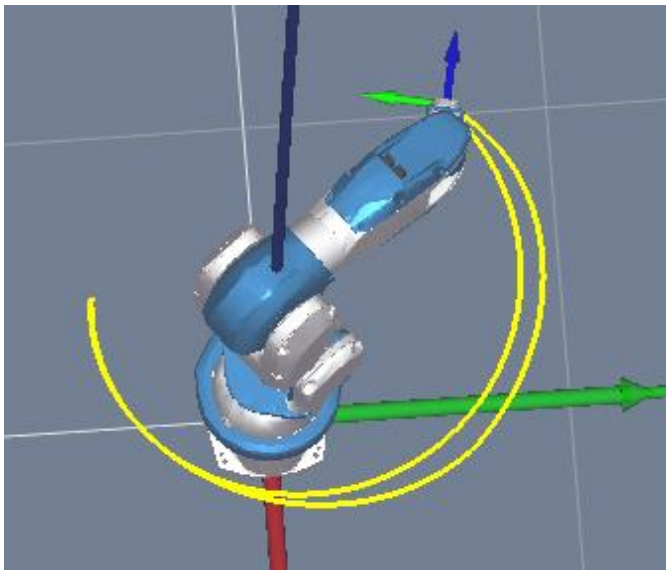


Figure 3.8: The triangular trajectory simulated on Roboshop

The subsequent test focused on deploying the wrapper with another Comau robot: the educational e.DO. This robot was chosen for its accessibility and the ability to conduct tests on actual hardware. A MoveIt configuration package was generated for e.DO by adapting its URDF file and importing it into the MoveIt Setup Assistant. The resulting files were added to the comau_description package, and new options were integrated into the bringup launcher to support switching between robots. With only minor code adjustments, support for multiple robot types was successfully implemented, demonstrating the scalability and modularity of the wrapper.

The first trajectory tested on the e.DO robot mirrored the hourglass-shaped joint trajectory previously executed on the Cobot. The results, visualized using MATLAB and Roboshop, were consistent with prior tests: the trajectory was well-formed, the joint values were accurate, and the execution response was prompt. These were the recorded positions (other joints were positioned at 0, throughout the whole trajectory):

Table 3.2: The written down positions for the e.Do hourglass trajectory

	Joint 1	Joint 5
Point A	-30°	-20°
Point B	-30°	20°
Point C	30°	20°
Point D	30°	-20°

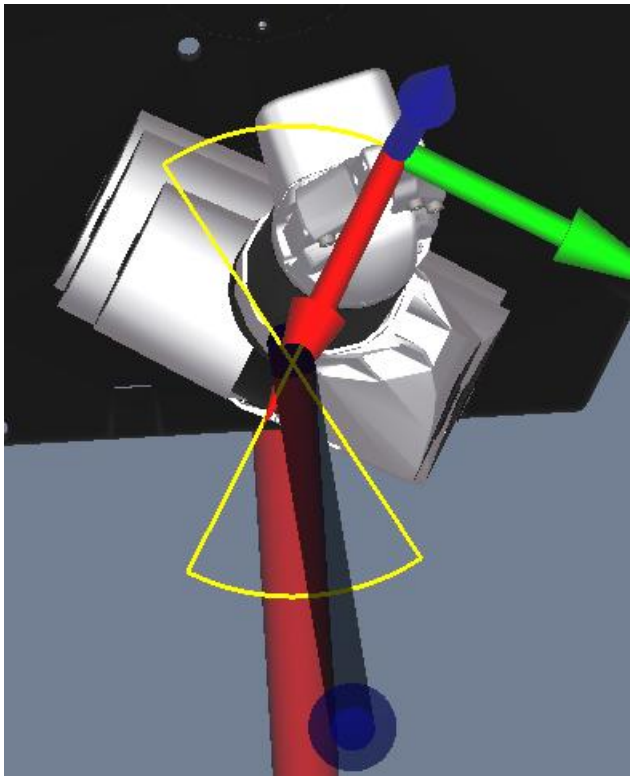


Figure 3.9: The e.DO hourglass trajectory simulated on Roboshop

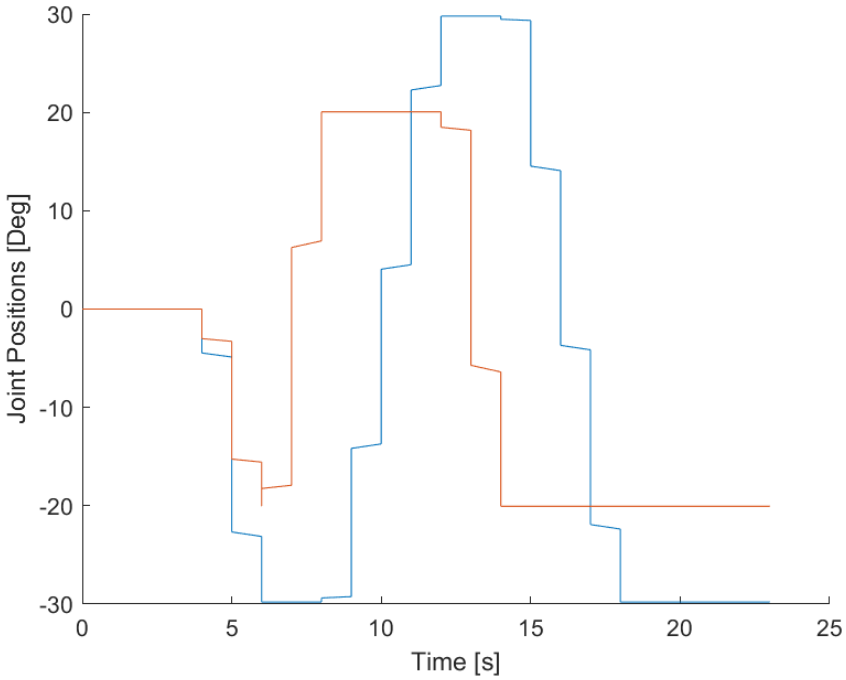


Figure 3.10: The positions of the joints (blue for joint 1 and orange for joint 5) recorded by rosbag

Again, the recorded trajectory was compared to the generated one:

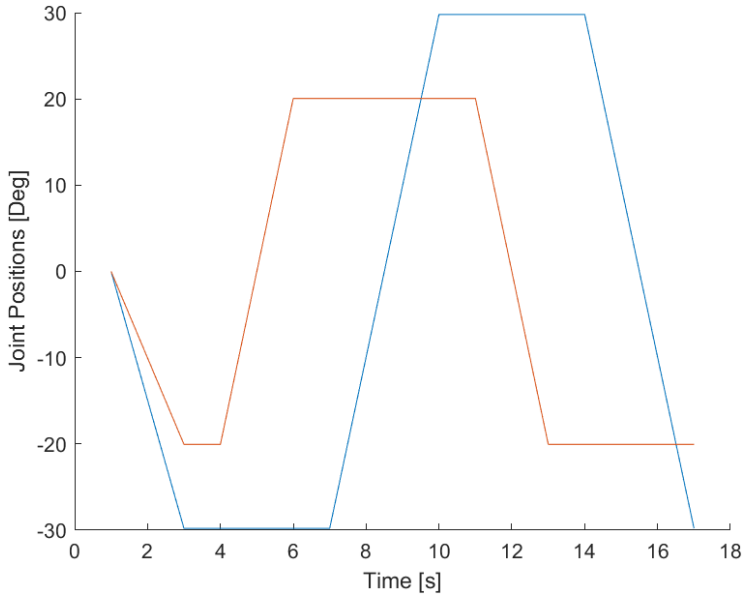


Figure 3.11: The positions of the joints generated by Moveit

A more challenging test followed, where the robot was programmed to "write" its own name using its end-effector. Roboshop was used to record the joint positions for each letter. These poses were then interpolated and validated by MoveIt, which produced a feasible trajectory. As shown in the accompanying figures, the robot successfully followed the desired path. The system maintained a fast response time, and the difference between planned and executed positions remained minimal. The shape of the trajectory was the following:

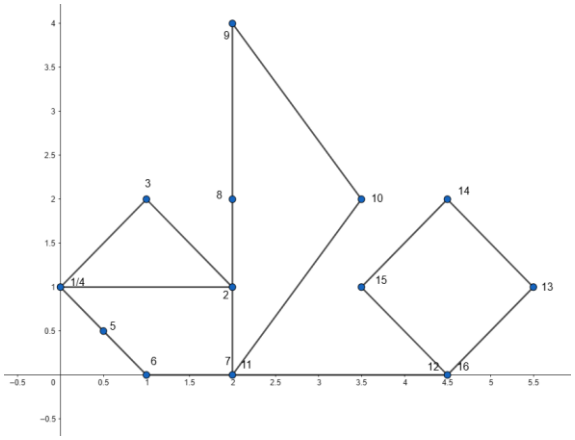


Figure 3.12: The desired shape of the sign trajectory.

These were the recorded joint positions:

Table 3.3: The written down positions for the e.DO sign trajectory

	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
Point 1	2°	55°	-94°	0°	40°	-2°
Point 2	-90°	45°	-93°	0°	48°	90°
Point 3	-26°	44°	-93°	0°	48°	27°
Point 4	2°	55°	-94°	0°	40°	-2°
Point 5	-153°	50°	-93°	0°	45°	56°
Point 6	-153°	44°	-93º	0°	49°	54°
Point 7	-135°	41°	-92°	0°	51°	35°

Point 8	-47°	41°	-93°	0°	51°	-52°
Point 9	-18°	22°	-83°	0°	60°	81°
Point 10	-64°	33°	-89°	0°	56°	-35°
Point 11	-135°	41°	-92°	0°	51°	35°
Point 12	-109°	22°	-83°	0°	61°	11°
Point 13	-91°	11°	-75°	0°	63°	-9°
Point 14	-72°	22°	-83°	0°	61°	-27°
Point 15	-92°	35°	-90°	0°	55°	-8°
Point 16	-109°	22°	-83°	0°	61°	11°

Roboshop was used to visualize the trajectory:

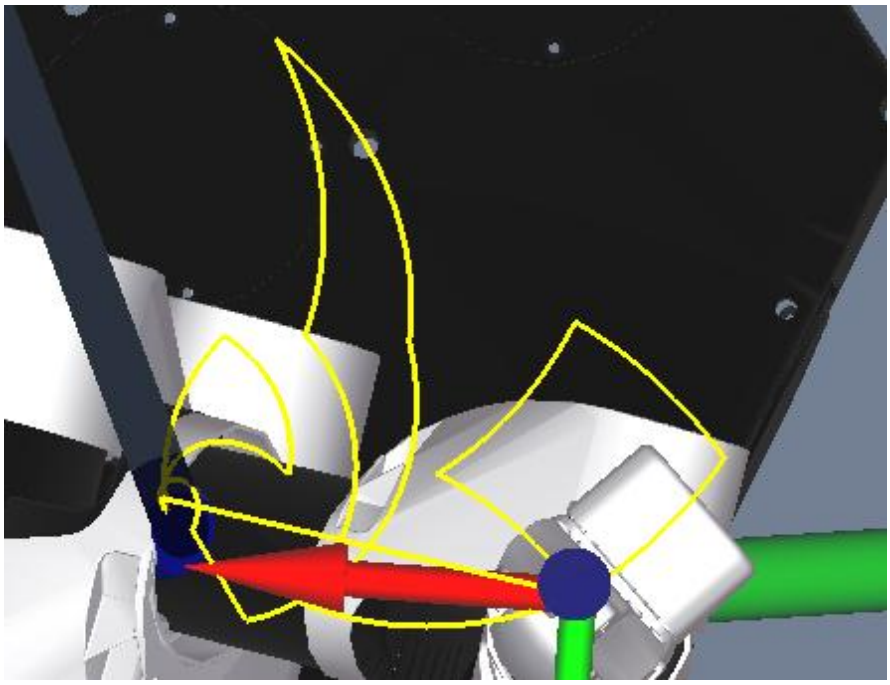


Figure 3.13: The shape of the sign trajectory simulated on Roboshop

Matlab was again used to visualize the joint positions recorded by rosbag and the ones generated by Moveit's trajectory planner.

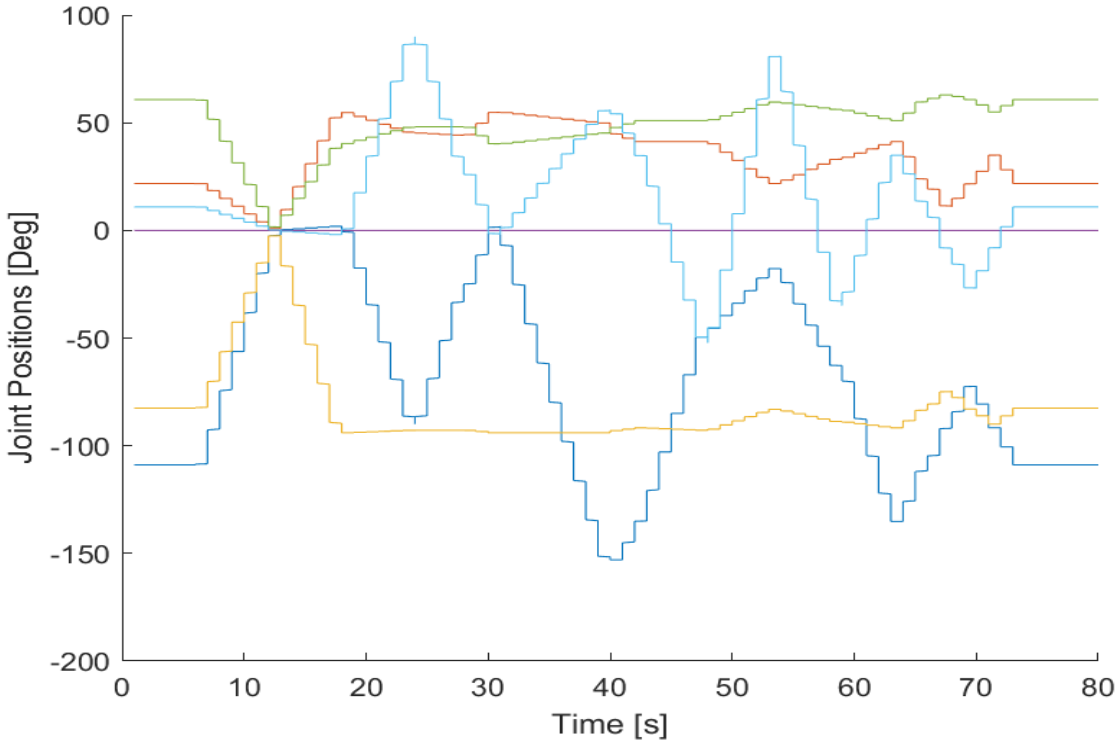


Figure 3.14: The positions of the joints (blue for joint 1, orange for joint 2, yellow for joint 3, purple for joint 4, green for joint 5 and cyan for joint 6) recorded by rosbag

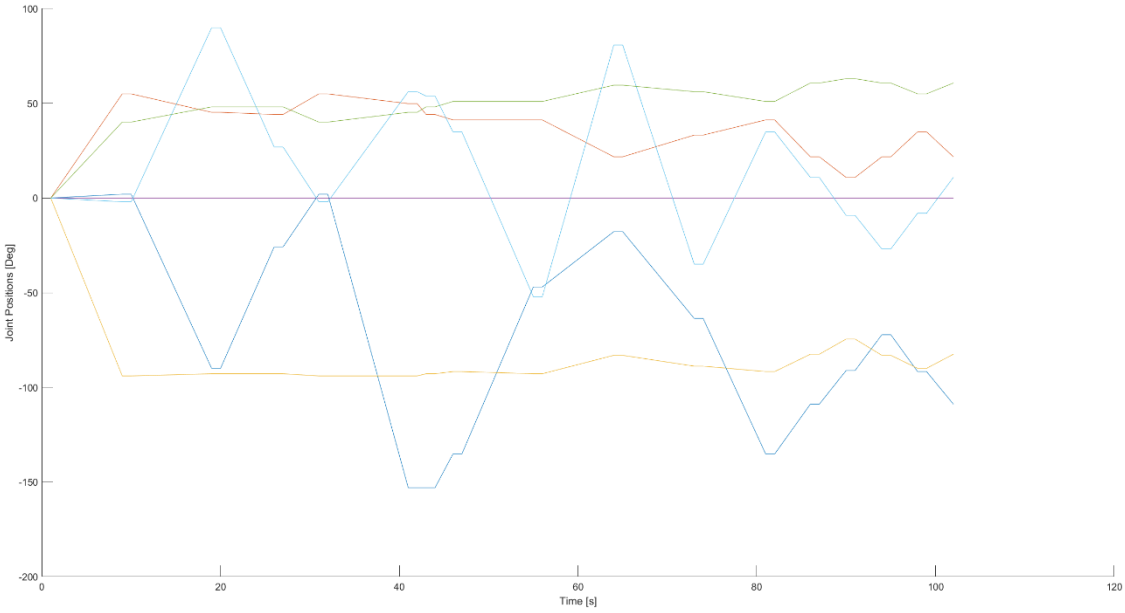


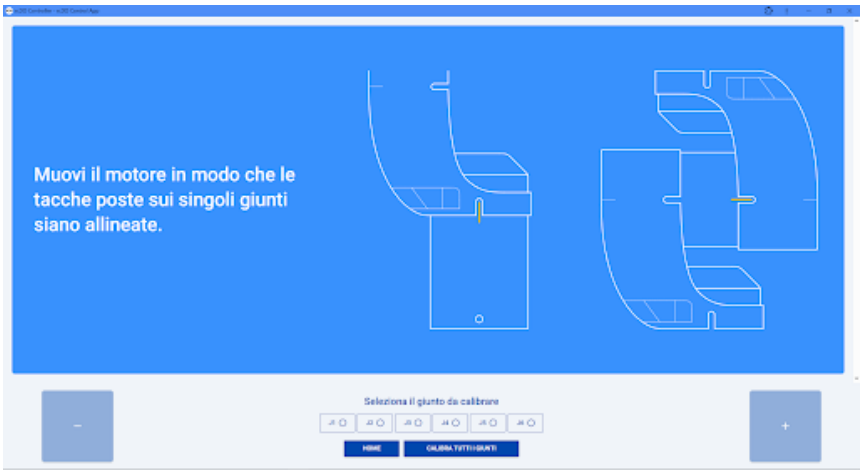
Figure 3.15: The positions of the joints generated by Moveit

Encouraged by the consistency and quality of the results in simulation, the next logical step was to validate the wrapper using real hardware.

3.4 Real Hardware tests

As previously mentioned, the robot selected for real-world testing was the e.DO educational robot. This choice was motivated by its ease of access and the encouraging results obtained during simulated experiments on Roboshop. Both trajectories developed in simulation, the hourglass-shaped path and the name-writing sequence, were replicated on the physical robot.

To carry out these tests, a specific setup was required. The e.DO robot was connected via Ethernet to a network comprising a Windows PC, an Ubuntu 22.04 machine, and a TP switch. The Windows computer was used to interface with the robot through the e.DO Controller application, a Comau Progressive Web Application accessible by navigating to the robot's IP address and accepting its web certificate.



*F*igure 3:16 The Comau e.Do control APP

This app provides a visual interface for commanding the robot, although in this context it was used solely for joint calibration and to initialize the robot with the onboard ROS 2 server.

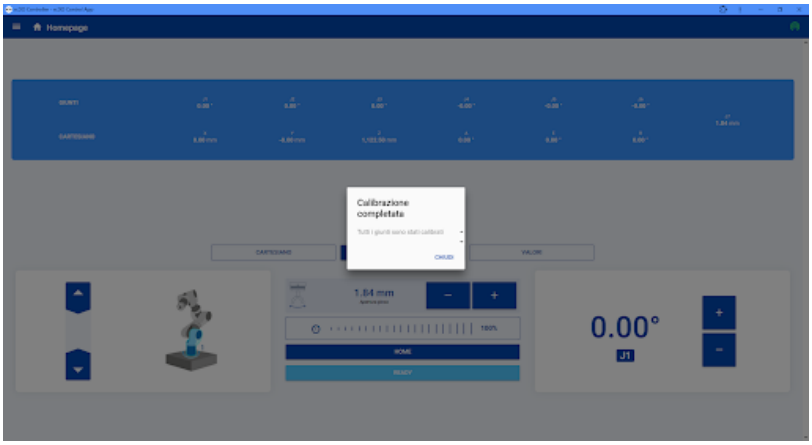


Figure 3.17: Calibrating joints on the control app

Once the robot was initialized, it became ready to receive commands from the Ubuntu computer, which was running the ROS 2 environment. The same procedure used to communicate with Roboshop was employed here, with the only difference being the need to update the IP address in the driver configuration files to match that of the physical robot.

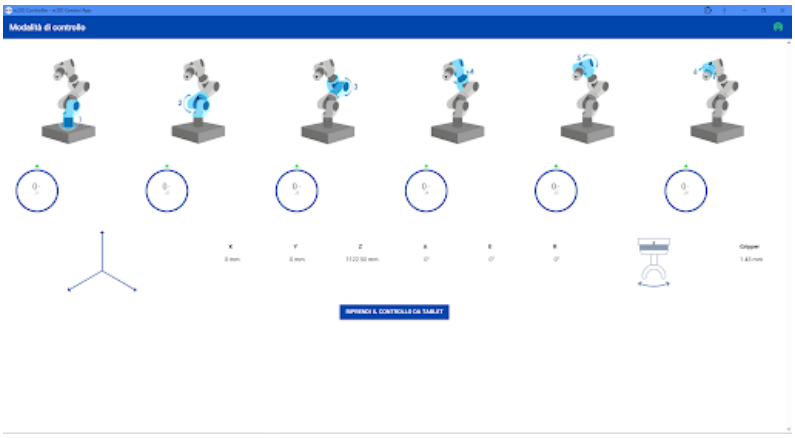


Figure 3.18: The control app showing the positions of the calibrated joints

With the setup complete, testing on the real robot began. The first trajectory to be executed was the hourglass shape, mirroring the earlier simulation. During execution, ROS 2 bags were used to record the real-time joint states from the robot's feedback, while the planned joint values from MoveIt were stored in a corresponding JSON file. MATLAB was then used to generate comparative plots from both datasets.

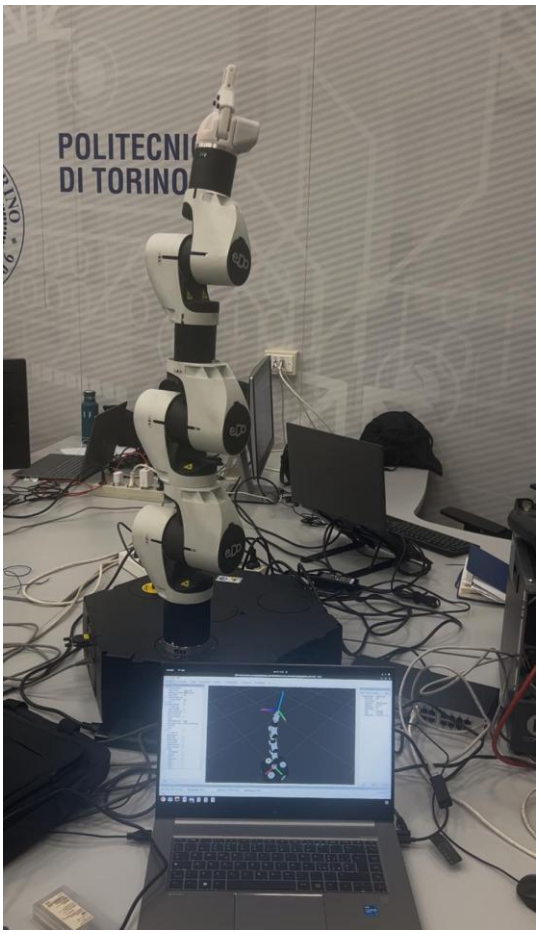


Figure 3.19: The e.Do robot setup

As illustrated below, despite a minor initial offset due to a different starting pose, the real robot's behavior closely matched the simulation. The difference between the planned and actual joint positions was minimal, and the system exhibited a

comparably short response time. This level of consistency was anticipated, given Roboshop's accuracy as a simulation environment.

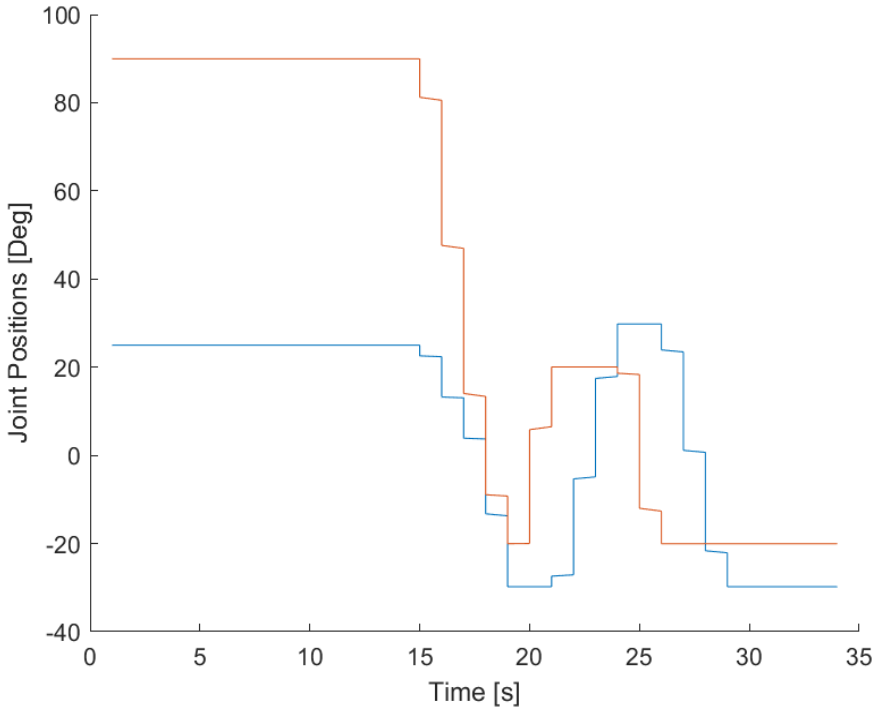


Figure 3.20: The positions of the joints (blue for joint 1 and orange for joint 5) recorded by rosbag during the test on the real robot

The recorded positions were compared to the ones created by moveit:

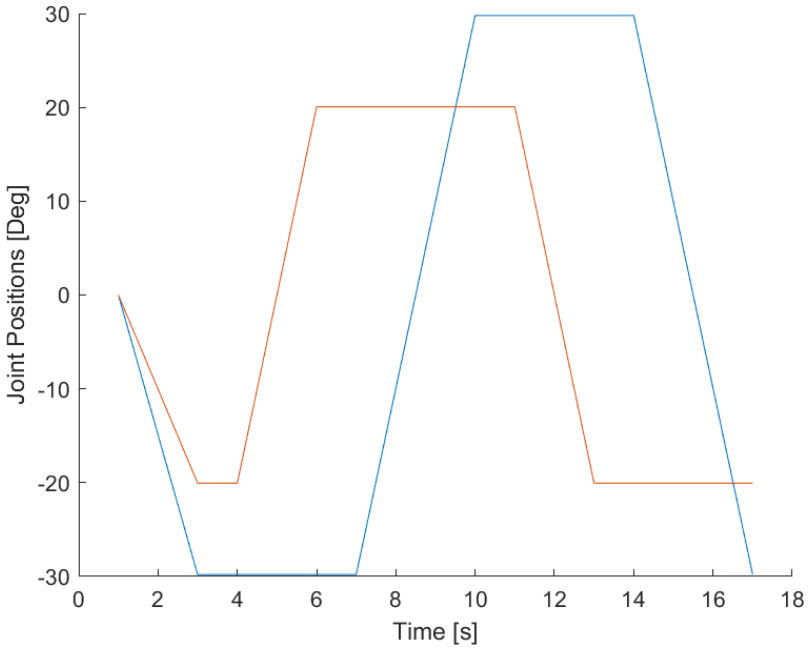


Figure 3.21: The positions of the joints generated by Moveit during the real robot test

Following the successful execution of the hourglass trajectory, the second path, spelling out the robot's name, was tested on the real hardware. The results were equally satisfying, with the robot precisely tracing the intended path. This final test reinforced the validity of the developed MoveIt-based wrapper as a reliable and flexible trajectory planner, capable of handling both joint-space and Cartesian trajectories on real robotic platforms.

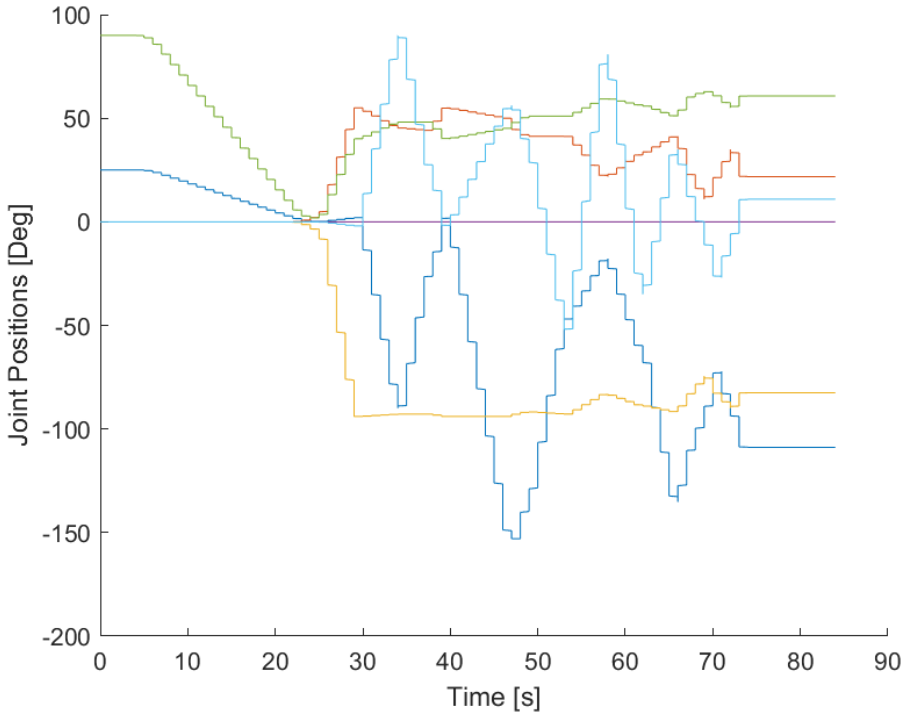


Figure 3.22: The positions of the joints (blue for joint 1, orange for joint 2, yellow for joint 3, purple for joint 4, green for joint 5 and cyan for joint 6) recorded by rosbag during the real robot test

Again, the recorded trajectory was compared to the generated one:

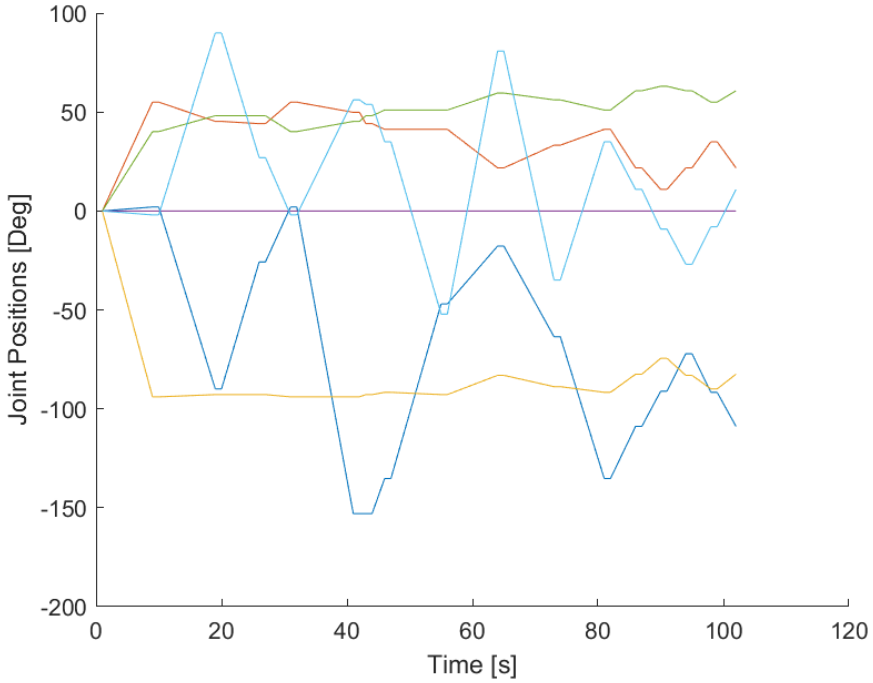


Figure 3.23: The positions of the joints generated by Moveit during the real hardware test

Chapter4: Conclusions and future developments

4.1 Conclusions

This thesis set out to address the growing need for transitioning robotic software frameworks from ROS 1 to ROS 2 by focusing on a critical module within Comau's ROS ecosystem: the integration of MoveIt for motion planning and trajectory generation. After a careful assessment of the existing ROS 1 Moveit package, the objective was to replicate a similar behaviour in the ROS 2 one.

Through this work, a fully functional wrapper was developed that connects MoveIt 2 with the Comau ROS 2 driver, enabling both joint-space and Cartesian-space trajectory planning. This wrapper was designed to be as modular and reusable as possible, replicating the structure and logic of the original ROS 1 implementation while embracing the benefits of ROS 2, including its node lifecycle management, enhanced communication middleware and improved real-time capabilities.

The new wrapper was rigorously tested in both simulation and on physical hardware. The simulation tests, conducted using Roboshop and MATLAB analysis, demonstrated a high degree of accuracy between the trajectories planned by MoveIt and those executed in the simulation. These results were further validated in real-world tests using the Comau e.DO robot, where performance in terms of trajectory precision and system responsiveness mirrored that observed in simulation.

Importantly, the development effort emphasized not only functionality but also extensibility. The wrapper was designed with generality in mind, capable of supporting additional robot models with minimal code changes. This was confirmed through successful adaptation and execution of trajectories on both the Racer5 Cobot and the e.DO educational robot.

Overall, this thesis demonstrates that MoveIt 2 can be successfully integrated into a ROS 2-based industrial robot driver to enable robust and flexible motion planning. The solution not only meets current requirements but also opens the door to a wide range of future enhancements. By achieving seamless execution of both joint and Cartesian trajectories, with reliable performance on simulated and real robots, this work significantly advances the ROS 2 driver's utility for both research and industry applications.

Furthermore, the alignment of this development with the modular and scalable principles of ROS 2 ensures that the wrapper can serve as a foundational component in broader robot control architectures, including applications involving perception, task planning, and adaptive control. Given its success and stability, the MoveIt wrapper developed in this thesis is well-positioned to be integrated into the mainline ROS 2 driver for Comau robots, contributing directly to ongoing open-source efforts and future industrial deployments.

4.2 Future Developments

Although this thesis successfully demonstrates the core integration of MoveIt within the Comau ROS 2 driver, there remain numerous opportunities to build upon this work and expand its functionality. These avenues of future development span improvements in planning flexibility, support for more complex environments, integration of sensing, and real-time performance optimization.

A logical first step is to extend the wrapper to support Comau's full range of industrial robots. The architecture designed during this thesis has already shown its scalability through tests on two different robot platforms. Extending support further would primarily involve updating URDF descriptions and reconfiguring the MoveIt Setup Assistant for each model. Given that the controller and planning logic remain consistent, this task is both feasible and efficient. Such an extension

would significantly increase the utility of the wrapper for Comau's clients and collaborators.

Another promising direction is the enhancement of planning capabilities. While this thesis used OMPL as the default planner, MoveIt supports multiple planning backends, such as STOMP and CHOMP, each with different strengths. Adding the ability to switch dynamically between planners, depending on the application or trajectory constraints, could greatly improve the system's adaptability. For instance, CHOMP may be better suited for smooth path generation in cluttered environments, while OMPL may excel in open, high-speed scenarios.

A deeper understanding of MoveIt's APIs and configuration options could also lead to improved use of constraints, enabling finer control over trajectory execution. This could include orientation constraints, velocity or acceleration bounds, and end-effector path constraints. Supporting complex constraint sets would allow the robot to perform tasks that require precision and adherence to strict physical limitations, such as welding or surgical assistance.

An important area for future work lies in the integration of sensor feedback. By incorporating real-time data from cameras, LiDAR, or force-torque sensors, the robot could dynamically update its planning scene in MoveIt. This would allow it to react to changes in its environment and perform tasks such as obstacle avoidance, object manipulation, or human-robot interaction. MoveIt already supports a planning scene interface that can be updated during execution; leveraging this capability would enable the development of reactive and adaptive systems.

Beyond the motion layer, combining the MoveIt wrapper with higher-level task planners (such as behavior trees or task planning frameworks like PDDL or FlexBE) could allow the robot to perform sequences of actions intelligently, based on task goals rather than raw trajectory inputs. This would move the system closer to autonomous operation in semi-structured or dynamic environments.

Finally, a significant long-term goal is achieving real-time deterministic behavior, which is increasingly essential for industrial robotics. While ROS 2 offers improved real-time support compared to ROS 1, achieving full real-time capabilities requires careful design, including the use of appropriate real-time operating systems, real-time safe middleware configurations, and control loops with bounded latency. MoveIt 2 and ros2_control provide the foundation for this, but additional optimization and testing would be necessary to meet strict industrial standards.

In conclusion, the wrapper developed in this thesis provides a robust, modular, and extensible foundation for motion planning with Comau robots in ROS 2 using Moveit. With continued development, it can evolve into a powerful motion layer for advanced robotic applications, capable of real-time, sensor-aware, and adaptive control in both research and industrial domains.

Bibliography

[1] International Federation of Robotics (IFR) – Top 5 Robot Trends 2024

[2] Financial Times – Cobots Gaining Momentum

[3] The Verge – Hyundai's Robotic EV Factory

[4] ROS-Industrial Official Site

[5] ROS 2 Tutorials: https://docs.ros.org/en/humble/Tutorials.html

[6] MoveIt Tutorials: https://moveit.picknik.ai/main/doc/tutorials/tutorials.html

[7] Comau Official Site: https://www.comau.com/it/

[8] Comau ROS/ROS2 library

[9] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. 2008. Robotics: Modelling, Planning and Control (1st. ed.). Springer Publishing Company, Incorporated.