



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

The Grainger College of Engineering
University of Illinois at Urbana-Champaign
Universidad Politécnica de Valencia

Set up and control of a UR3 Robot using ROS2 Humble

MASTER'S THESIS

17th May 2023, Urbana, Illinois

Author: Diego Jarque Pérez

Tutor: Daniel Block

Master's Degree in Mechatronics Engineering

ABSTRACT

This Master's Thesis aims to control a UR3 robotic arm using Robotics Operative System 2 (ROS2) software, specifically the Humble version. The project will involve computer setup, ROS2 methods learning, UR driver packages analysis, and utilization of simulation, visualization (RViz), and data analysis (PlotJuggler) software. The project's primary objective is to achieve joint position control and movement of the robot, with further exploration of other control types and the MoveIt tool. By the project's conclusion, the goal is to develop a comprehensive understanding of ROS2 Humble and its applications in robotics control, specifically for the UR3.

Key words: UR3 robotic arm, ROS2 Humble, Control framework, Simulation, MoveIt.

RESUMEN

Este Trabajo de Fin de Máster busca realizar el control de un brazo robótico UR3 empleando Robotics Operative System 2 (ROS2), concretamente la versión Humble. El proyecto conllevará la configuración del ordenador, el aprendizaje de los procedimientos de ROS2, el análisis del controlador de UR, y la utilización de programas de simulación, visualización (RViz) y análisis de datos (PlotJuggler). El objetivo principal del proyecto es lograr el control de posición y movimiento del robot, así como investigar otros tipos de control y la herramienta MoveIt. Al finalizar el proyecto, se pretende haber alcanzado un entendimiento teórico y práctico de ROS2 Humble y sus aplicaciones en el control de robots, concretamente del UR3.

Palabras clave: Brazo robóticoUR3, ROS2 Humble, Control framework, Simulación, MoveIt.

Index

| | | |
|--------|---|----|
| 1. | INTRODUCCTION | 11 |
| 1.1. | CONTEXT | 11 |
| 1.2. | APPLICATION | 11 |
| 2. | OBJECTIVE..... | 12 |
| 3. | SOFTWARE..... | 12 |
| 3.1. | ROS2 | 12 |
| 3.1.1. | Organization..... | 13 |
| 3.1.2. | ROS2 Graph..... | 13 |
| 3.2. | Gazebo | 16 |
| 3.3. | Universal Robots Simulator..... | 17 |
| 3.4. | RViz 2..... | 18 |
| 3.5. | PlotJuggler..... | 19 |
| 4. | HARDWARE..... | 21 |
| 4.1. | Universal Robots UR3..... | 21 |
| 5. | ROS2 SETUP AND INSTALLATION | 22 |
| 6. | ROS2 CONTROL FRAMEWORK..... | 27 |
| 6.1. | ros2_control package | 30 |
| 6.2. | ros2_controllers package | 30 |
| 7. | UR ROS2 DRIVER..... | 31 |
| 7.1. | Universal_Robots_ROS2_Driver..... | 34 |
| 7.2. | Universal_Robots_ROS2_Description | 40 |
| 7.3. | Universal_Robots_Client_Library..... | 42 |
| 8. | STARTING UNIVERSAL ROBOTS SIMULATIONS | 43 |
| 8.1. | URSim Image Docker..... | 43 |
| 8.2. | Gazebo | 51 |
| 9. | VISUALIZATION TOOL, RVIZ 2 | 55 |
| 10. | UR3 ROBOT SETUP | 60 |
| 11. | CONTROLLING UR3 | 63 |
| 11.1. | forward_position_controller | 66 |
| 11.2. | forward_velocity_controller..... | 67 |
| 11.3. | joint_trajectory_controller..... | 68 |
| 11.4. | scaled_joint_trajectory_controller..... | 70 |

| | | |
|-------|--|-----|
| 11.5. | rqt_joint_trajectory_controller | 77 |
| 11.6. | Movelt 2 control..... | 79 |
| 12. | DATA AQUISITION AND ANALYSIS | 79 |
| 13. | MOVEIT 2 | 86 |
| 13.1. | Movelt Architecture | 87 |
| 13.2. | RViz Plugin | 89 |
| 13.3. | Movelt with URSim..... | 96 |
| 13.4. | Movelt with Gazebo | 97 |
| 13.5. | Movelt with real UR3 | 97 |
| 13.6. | Other Movelt user interfaces | 97 |
| 14. | CONCLUSION | 98 |
| 15. | FUTURE INVESTIGATION PATHS | 98 |
| 16. | BIBLIOGRAPHY | 100 |
| | APPENDIX 1. COMMENTED CODE | 104 |
| | APPENDIX 2. INSTALLATION AND SETUP MANUAL..... | 133 |
| | APPENDIX 3. LINUX AND ROS2 BASIC COMMANDS | 147 |
| | APPENDIX 4. SETUP AN UBUNTU SYSTEM WITH REAL-TIME CAPABILITIES | 153 |

Figure table

| | |
|---|----|
| Figure 1. Representation of nodes into an application. Source: Created by the author..... | 14 |
| Figure 2. Communication trough a topic between two nodes. Source: Created by the author. | 14 |
| Figure 3. Communication trough a service between two nodes. Source: Created by the author. | 15 |
| | 15 |
| Figure 4. Communication trough an action between two nodes. Source: Created by the author. | 15 |
| | 15 |
| Figure 5. Parameters as settings or characteristics of nodes. Source: Created by the author.... | 16 |
| Figure 6. Gazebo graphic interface with UR3 model. Source: Created by the author..... | 17 |
| Figure 7. URSim FreeDrive interface. Source: Created by the author..... | 18 |
| Figure 8. RViz interface with UR3 model. Source: Created by the author | 19 |
| Figure 9. PlotJuggler layout graphing real-time data stream from ROS2 topics. Source: Created by the author | 20 |
| Figure 10. PlotJuggler layout graphing stored data from ROS2 topics. Source: Created by the author..... | 21 |
| Figure 11. Universal Robots CB3 – Series UR3. Source: UR3 3D Model [72]. | 21 |
| Figure 12. Control panel with touch screen, and graphical interface provided by Universal Robots for their cobots. Source: Do more than just keep up. Try our virtual simulator [69]..... | 22 |
| Figure 13. ROS2 Humble underlay workspace. Source: Created by the author..... | 24 |
| Figure 14. Installed packages inside /include folder. Source: Created by the author. | 25 |
| Figure 15. Line to source ROS2 when a new shell is opened added to the bashrc file. Source: Created by the author. | 26 |
| Figure 16. Line to enable colcon autocomplete added to the bashrc file. Source: Created by the author. | 27 |
| Figure 17. Architecture of ROS2 Control Framework. Source: ros2_control Maintainers [64]... . | 28 |
| Figure 18. ROS2 control commands. Source: Created by the author..... | 29 |
| Figure 19. Packages included inside the ros2_control package. Source: Created by the author.30 | |
| Figure 20. Packages included inside the ros2_controllers package. Source: Created by the author. | 31 |
| Figure 21. Packages added to the new workspace /src folder. Source: Created by the author. . | 33 |
| Figure 22. Packages installed inside the Universal_Robots_ROS2_Driver package. Source: Created by the author. | 34 |
| Figure 23. Trajectory generated by joint_trajectory_controller and explanation of the results. Source: GitHub ur_controllers [32]. | 35 |
| Figure 24. Trajectory generated by scaled_joint_trajectory_controller and explanation of the results. Source: GitHub ur_controllers [32]..... | 36 |
| Figure 25. ur_robot_driver package. Source: Created by the author. | 37 |
| Figure 26. Launch folder and launch files of the ur_robot_driver package, ur_control.launch.py marked in red. Source: Created by the author. | 37 |
| Figure 27. Src folder of the ur_robot_driver package, with the executables that start the nodes required by the driver..... | 39 |
| Figure 28. Config folder of the ur_robot_driver package, with the YAML files to configure the driver and the update rates. Source: Created by the author. | 39 |
| Figure 29. Scripts folder of the ur_robot_driver package, start_urtsim.sh will initiate URSim docker simulation and tool_communication.py is only available for e-series. Source: Created by the author..... | 39 |

| | |
|---|----|
| Figure 30. Folders from the UniversalRobots_ROS2_Description package. Source: Created by the author..... | 40 |
| Figure 31. Main files from the urdf folder of the UR description package. Source: Created by the author..... | 41 |
| Figure 32. STL files from the meshes/ur3/collision folder of the UR description package. Source: Created by the author. | 41 |
| Figure 33. DAE files from the meshes/ur3/visual folder of the UR description package. Source: Created by the author. | 41 |
| Figure 34. YAML files from the config/ur3 folder of the UR description package. Source: Created by the author. | 42 |
| Figure 35. Architecture overview of the robot interface created by URDriver class of the package. Source: GitHub Universal_Robots_Client_Library [29]. | 42 |
| Figure 36. Creating ursim_net and printing the link to access the URSim. Source: Created by the author. | 44 |
| Figure 37. Browser tab to connect with the Docker URSim, field to insert your password located in the top right corner (red circled). Source: Created by the author..... | 45 |
| Figure 38. “Power off” robot mode. Source: Created by the author. | 45 |
| Figure 39. “Idle” robot mode. Source: Created by the author. | 46 |
| Figure 40. “Normal” robot mode. Source: created by the author..... | 46 |
| Figure 41. Main PolyScope URSim screen. Source: Created by the author..... | 47 |
| Figure 42. URSim FreeDrive interface. Source: Created by the author. | 47 |
| Figure 43. Program Robot screen. Source: Created by the author..... | 48 |
| Figure 44. Empty Program screen. Source: Created by the author. | 49 |
| Figure 45. Structure Tab inside Empty Program screen. Source: Created by the author. | 49 |
| Figure 46. Load and start the simulation by pressing the Play button. Source: Created by the author. | 50 |
| Figure 47. Graphics tab from the URSim Program section. Source: Created by the author..... | 50 |
| Figure 48. Exit to main screen and “Initialize Robot” screen. Source: Created by the author.... | 51 |
| Figure 49. UR3 Gazebo simulation showing joints breaking. Source: Created by the author.... | 53 |
| Figure 50. UR3 Gazebo simulation after solving the dynamic discrepancy. Source: Created by the author..... | 54 |
| Figure 51. RViz layout when Gazebo simulation is launched, with added TF display. Source: Created by the author. | 56 |
| Figure 52. Global Options, Global Status and Grid displays in RViz 2. Source: Created by the author. | 57 |
| Figure 53. Robot Model element from RViz2 display. Source: Created by the author..... | 57 |
| Figure 54. TF element from RViz2 display. Source: Created by the author. | 58 |
| Figure 55. Display types in RViz and Description chart. Source: Created by the author. | 59 |
| Figure 56. Views RViz menu. Source: Created by the author..... | 60 |
| Figure 57. External Control in the Installation tab from Program Robot menu. Source: Created by the author. | 61 |
| Figure 58. Set Network detailed setting of the UR3. Source: Created by the author. | 62 |
| Figure 59. Setting the IP address of the control PC. Source: Created by the author..... | 62 |
| Figure 60. Configure Host IP with control PC address. Source: Created by the author..... | 63 |
| Figure 61. Gazebo simulation and RViz visualization of the UR3 after reaching a goal sent by the joint_trajectory_controller. Source: Created by the author. | 70 |
| Figure 62. Move to the Move tab to move the robot to the home position manually. Source: Created by the author. | 72 |

| | |
|--|-----|
| Figure 63. Select Move Joints angles to set the intial position as goal. Source: Created by the author..... | 73 |
| Figure 64. Introduce the initial position and click OK. Source: Created by the author..... | 73 |
| Figure 65. Click Auto until the Cancel button changes to Ok. Source: Created by the author.... | 74 |
| Figure 66. Go back to Program tab and Play the external control. Source: Created by the author. | 74 |
| Figure 67. Laboratory real UR3 joint position range restrictions. Source: Created by the author. | 75 |
| Figure 68. URCaps program to control the real UR3. Source: Created by the author..... | 76 |
| Figure 69. GUI displayed after running the rqt_joint_trajectory_controller. Source: Created by the author..... | 77 |
| Figure 70. GUI after turning on the controller. Source: Created by the author. | 78 |
| Figure 71. URSim, RViz and GUI after performing some joint position changes. Source: Created by the author..... | 78 |
| Figure 72. Data being recorded with rosbag. Source: Created by the author..... | 81 |
| Figure 73. PlotJuggler initial GUI. Source: Created by the author..... | 82 |
| Figure 74. Topic selector menu. Source: Created by the author. | 82 |
| Figure 75. PlotJuggler layout designed for real-time data stream. Source: Created by the author. | 83 |
| Figure 76. Layout saving screen. Source: Created by the author. | 84 |
| Figure 77. Message outputted after PlotJuggler crush for trying to load a saved layout. Source: Created by the author. | 85 |
| Figure 78. Recorded data representation with the same layout as in real-time. Source: Created by the author. | 85 |
| Figure 79. MoveIt2 architecture. Source: PickNik Robotics, https://moveit.picknik.ai/humble/doc/concepts/concepts.html | 87 |
| Figure 80. move_group architecture and relation to MoveIt components. Source: PickNik Robotics [55]. | 88 |
| Figure 81. RViz GUI with UR3 Motion Planning added. Source: Created by the author. | 89 |
| Figure 82. Sections from the display menu where users can select the robot description, planning scene topic and planning group. Source: Created by the author. | 90 |
| Figure 83. Scene Geometry and Scene Robot dropdowns from display menu. Source: Created by the author. | 90 |
| Figure 84. Planning Request and Planning Metrics dropdowns from display menu, and impacts in the RViz robot scene. Source: Created by the author. | 91 |
| Figure 85. Planned Path dropdown from display menu with planning path silhouette moving from start to goal position in RViz scene. Source: Created by the author..... | 91 |
| Figure 86. Context tab from specific menu. Source: Created by the author..... | 92 |
| Figure 87. Planning tab from specific menu. Source: Created by the author. | 92 |
| Figure 88. Joint tab from specific menu. Source: Created by the author..... | 93 |
| Figure 89. Scene Objects tab from specific menu. Source: Created by the author..... | 94 |
| Figure 90. Stored Scene tab from specific menu. Source: Created by the author. | 94 |
| Figure 91. Stored States tab from specific menu. Source: Created by the author. | 95 |
| Figure 92. Status tab from specific menu. Source: Created by the author. | 95 |
| Figure 93. Manipulation tab from specific menu. Source: Created by the author..... | 96 |
| Figure 94. Front page of the ROS2 Manipulation Basics course from The Construct. Source: Created by the author. | 98 |
| Figure 95. “Power off” robot mode. Source: Created by the author. | 137 |

| | |
|---|-----|
| Figure 96. “Idle” robot mode. Source: Created by the author..... | 138 |
| Figure 97. “Normal” robot mode. Source: created by the author..... | 138 |
| Figure 98. Main PolyScope URSim screen. Source: Created by the author..... | 139 |
| Figure 99. URSim FreeDrive interface. Source: Created by the author..... | 139 |
| Figure 100. Program Robot screen. Source: Created by the author..... | 140 |
| Figure 101. Empty Program screen. Source: Created by the author..... | 141 |
| Figure 102. Structure Tab inside Empty Program screen. Source: Created by the author..... | 141 |
| Figure 103. Load and start the simulation by pressing the Play button. Source: Created by the author..... | 142 |
| Figure 104. Graphics tab from the URSim Program section. Source: Created by the author .. | 142 |
| Figure 105. External Control in the Installation tab from Program Robot menu. Source: Created by the author .. | 144 |
| Figure 106. Set Network detailed setting of the UR3. Source: Created by the author .. | 145 |
| Figure 107. Setting the IP address of the control PC. Source: Created by the author..... | 146 |
| Figure 108. Configure Host IP with control PC address. Source: Created by the author. | 146 |

1. INTRODUCTION

1.1. CONTEXT

The field of robotics has been rapidly evolving in recent years and has had a significant impact on various industries, ranging from manufacturing to healthcare. Robots have been increasingly used to perform tasks that are repetitive, dangerous, or require a high level of precision. This has led to an increase in productivity, improved safety for workers, and a reduction in costs associated with manufacturing processes.

One of the key challenges in robotics has been developing a standard framework that can be used to design, develop, and deploy robotic systems. The Robot Operating System (ROS) has emerged as the dominant standard for robotics software development. ROS provides a framework for developing and controlling robotic systems, as well as tools for simulation, visualization, and data analysis.

ROS2 is the latest version of ROS, and it has introduced several improvements over its predecessor, including better performance, security, and scalability. ROS2 has also added support for real-time and embedded systems, making it suitable for a wider range of robotics applications.

The use of ROS and ROS2 has opened up opportunities for researchers, developers, and industry professionals to collaborate and share code, making it easier to develop complex robotic systems. Additionally, the use of open-source software has enabled the development of affordable robotic systems, which has made robotics more accessible to a wider audience.

The main objective of this project is to set up and control a Universal Robots UR3 robotic arm using ROS2 Humble. The UR3 is a lightweight robotic arm that is commonly used in manufacturing and research environments. This robot is a prime example of a robot that can benefit from the use of ROS2 for efficient and effective control. By successfully completing this project, the skills and knowledge gained can be applied to other robotics projects, contributing to the continued development and growth of the robotics industry.

1.2. APPLICATION

The outcomes of this project will have broad applications in future robotics development and education. Specifically, the documentation and manuals created as part of this project will enable quick and efficient acquisition of knowledge related to ROS2, ROS Control framework, and hardware setup for the UR3 robot. This will facilitate the integration of ROS2 into wider robotics systems, and provide a valuable resource for students, Teacher Assistants, and professors at the University of Illinois at Urbana-Champaign (UIUC) who are interested in exploring ROS2.

In addition, this project will be useful for independent users, as well as students and professionals from the Universidad Politécnica de Valencia (UPV), who can benefit from the knowledge and experience gained through this project. By automating the setup process and providing clear documentation, this project will enable a more streamlined and efficient approach to controlling the UR3 robot with ROS2, allowing users to focus on more complex control topics.

Overall, the results of this project will contribute to the ongoing development and expansion of ROS2 as the de facto standard for robotics software development. By providing a comprehensive guide to setting up and controlling the UR3 robot with ROS2, this project will help ensure that future robotics development and education continues to benefit from the powerful tools and frameworks provided by ROS2.

2. OBJECTIVE

The objective of this project is to set up the necessary software and hardware components, learn the ROS2 methods and control frameworks, and analyze the UR driver packages and MoveIt2, ROS2's robotic manipulation platform. Additionally, the project aims to validate the control framework using simulation and visualization tools before applying it to the UR3 robot. Finally, the project will test the control framework on a real UR3 robot and evaluate its performance. This report will discuss the procedures followed to achieve these objectives, including the software, hardware, and techniques used. The conclusion section will summarize the results and evaluate whether the objectives were met.

3. SOFTWARE

During the project, different applications and software tools were used to achieve the stated objectives. All these programs are closely related and work together under the main system which specifies all the methods and provides libraries to make the robot application work, ROS2. In this section a concise and basic description of ROS2 will be provided, as well as some general information related to the simulation, visualization and data analysis software used, to give some context before start explaining the procedures of the project.

3.1. ROS2

ROS, or Robot Operative System, is a collection of open-source tools and techniques designed to facilitate the development of robot applications. Its core function is to provide developers with various software libraries and tools necessary for building these applications.

Initially established in 2007, ROS has since become the premier system for building robot applications worldwide. Its open-source nature and its skilled community have contributed significantly to its progress over the years. As a result, ROS has undergone major development in recent times.

The first iteration of the Robot Operating System was ROS1. It had a vast number of characteristics that made it gain popularity in robotics industry, however it also shown some limitations. One major issue was its lack of real-time capabilities, which made it difficult to use in safety-critical applications. In addition, ROS1's communication system was based on a centralized master node, which made it susceptible to single points of failure.

In response to these challenges, ROS2 was released in 2015, which addressed many of these issues while retaining the strengths of ROS1. ROS2 introduced a new communication protocol, DDS (Data Distribution Service), which is more robust and reliable than ROS1's centralized system. This new protocol also allows for real-time communication and supports distributed systems, making ROS2 suitable for larger and more complex robot applications.

ROS2 also added several new features, such as better support for multi-robot systems, improved debugging tools, and increased modularity. Additionally, ROS2 maintained the core features that made ROS1 so popular, such as its open-source nature and its vast community of developers.

Right now, there are 8 different ROS distributions, but only two of them are currently being supported, ROS2 Foxy and ROS2 Humble. This project is developed using Humble distribution, which means that it will have slightly different features or compatibilities with other software components or hardware than Foxy, but the main procedures and ideas will be similar. The reason of choosing Humble over the rest of distributions is that it is the latest long-term support ROS2 release, meaning that it will ideally be the next most used ROS distribution.

ROS2 is a very complex system with lots of different applications, methods and tools. All the different features, concepts and ideas are widely discussed in its own website <https://www.ros.org/> and the basic ideas explained in this section can be found there or in different websites that will be cited in the Bibliography section at the end of this document with a more in-depth treatment.

3.1.1. Organization

To start understanding how ROS2 works, you need to understand how it is organized. ROS 2 is developed in your system in what are called “Workspaces”. The core workspace or underlay is where the main sets of packages or libraries are stored. But you can also create other subsequent local workspaces in which you can develop your own applications with ROS2. These are called overlays.

The way workspaces work is by sourcing setup files whenever you open a new shell, allowing the different workspaces to combine and use their utilities together, for example using ROS2 commands in the terminal.

Inside a workspace you can usually find the “build”, the “install”, the “log” and the “src” directories. Each of these directories have different proposes. For example, in the install directory, is where your workspace’s setup files are, which you can use to source your overlay.

The next level in ROS2 organization are the packages. Packages are the basic unit of organization for code files. They contain a collection of related files and directories that provide a specific set of functionalities. Packages are often used to share work among the ROS2 community or to implement other members' work into your own application.

Each package contains its own set of files and directories, including configuration files, source code, launch files, and other resources. These files are organized according to a standard ROS2 package structure which allows an easier shearing and reuse between developers.

In your system, all the packages are stored inside the src folder of the workspaces.

3.1.2. ROS2 Graph

The next step to understand ROS2 is to analyze how its different components work together to process data. These components are represented in the ROS2 Graph and represents the connections between the various executables that make up your application and shows how they communicate with one another.

The first component to consider are “Nodes”. Nodes are responsible for carrying a single purpose in the application, such as controlling the joint motors of a robot arm, managing the data obtained from a sensor, etc. In ROS2, every executable file can contain one or more nodes.

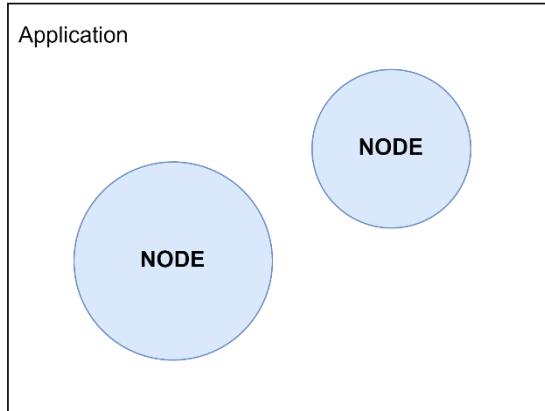


Figure 1. Representation of nodes into an application. Source: Created by the author.

Nodes can communicate to each other using different methods: topics, services, actions or parameters. These are the next components to analyze.

Topics act like communication buses between nodes. They are the most basic communication method and are used to exchange messages. Every node can publish (send) or subscribe (receive) to any number of topics, to do so they must have the same number of publishers and subscribers.

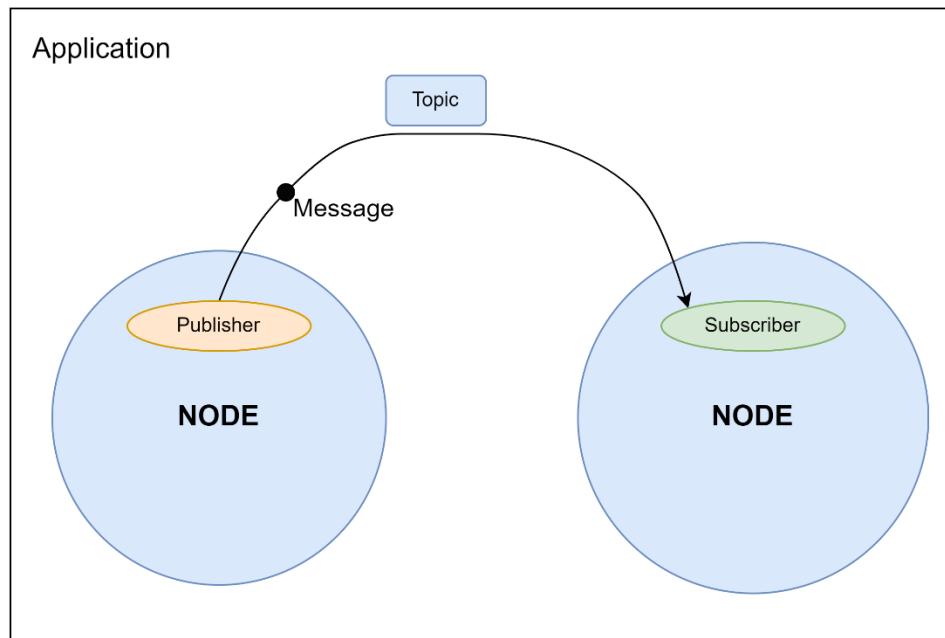


Figure 2. Communication through a topic between two nodes. Source: Created by the author.

Services, on the other hand, are a communication method based on a call-response model, this means that they only provide data when are called. Services have clients, on the nodes that request data, and servers, on the nodes that send data. Unlike with topics, where messages are continually updating, services only update when a request and a response are called.

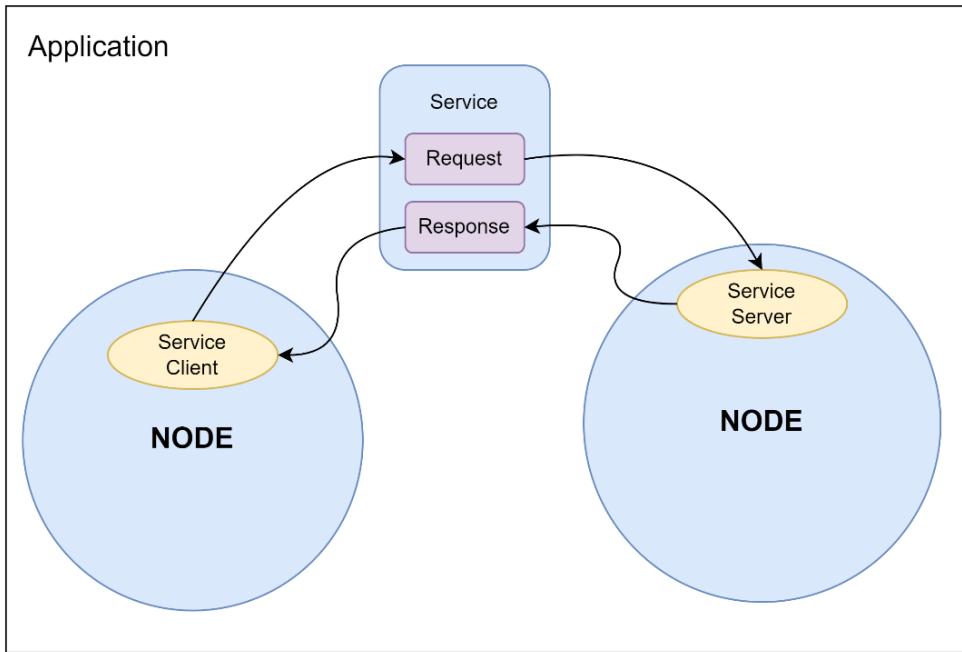


Figure 3. Communication through a service between two nodes. Source: Created by the author.

The last communication method are actions. An action works like a combination of topics and services. They use a client-server configuration, like services, but actions can be cancelled and also provide a feedback to the client, apart from the response. Basically an action client sends a goal to the action server, through a goal service, which acknowledges the goal and returns a stream of feedback, through a topic, and a result, through a result service.

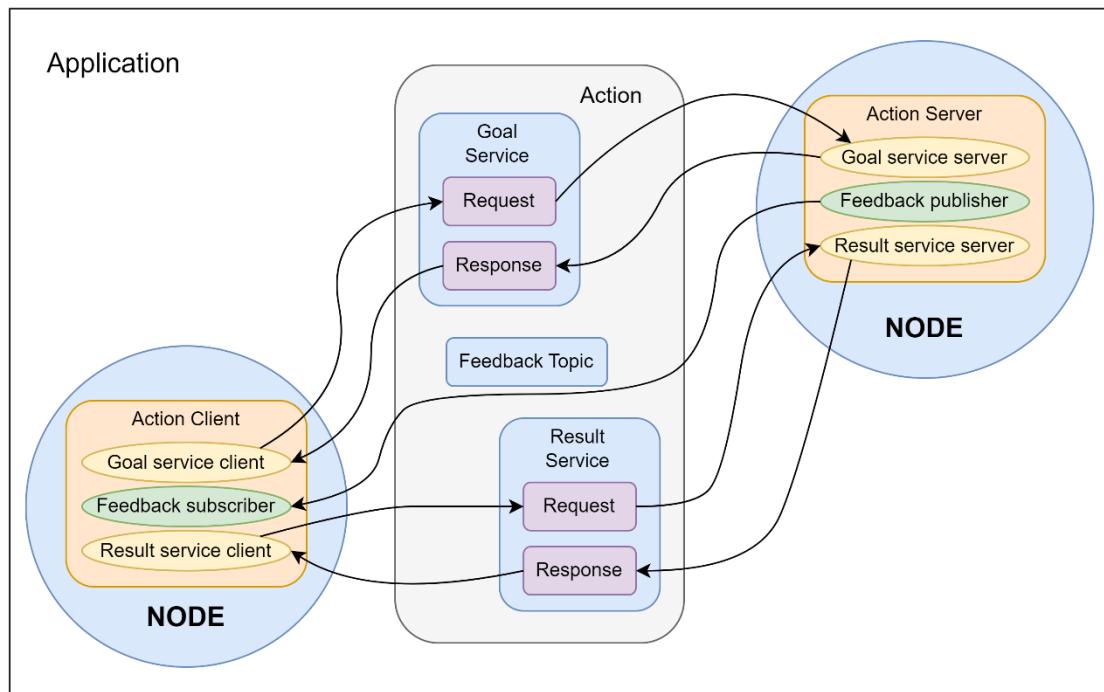


Figure 4. Communication through an action between two nodes. Source: Created by the author.

Another vital component that serves as a mean of communication between a node and the application developer, rather than between nodes themselves, are parameters. They are the

settings that define the characteristics of a node and enable developers to configure a node's behavior. Parameters can be used to specify properties that affect a node's performance.

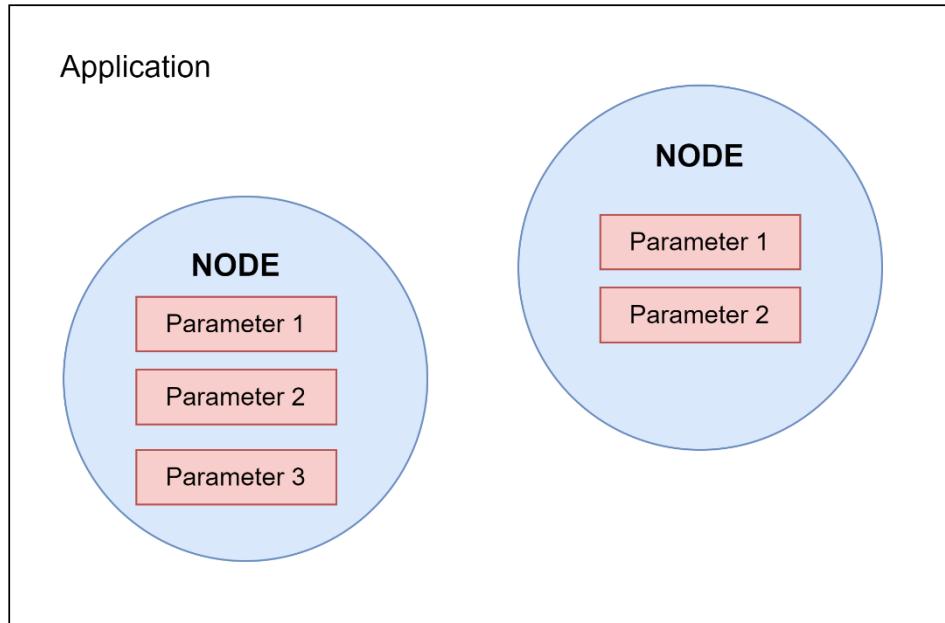


Figure 5. Parameters as settings or characteristics of nodes. Source: Created by the author.

How ROS2 is organize and how its main components relate between them are the basic ideas to understand how it works and gives the elementary knowledge to better understand the rest of the project.

3.2. Gazebo

Moving on from the main system of the project, ROS2, let's now explore one of the simulation software used for the UR3: Gazebo.

Gazebo is a popular open-source simulation software tool used in robotics for developing and testing robot applications. It allows users to create virtual environments that simulate real-world scenarios, enabling developers to test robot designs, algorithms, and controllers in a safe and cost-effective manner. Gazebo is a key tool in the development of autonomous robots and robot systems, as it allows researchers and developers to test their ideas and prototypes in a virtual environment before deploying them in the real world.

One of the main features of Gazebo is its physics engine, which accurately simulates the behavior of objects and robots in a virtual environment. This includes modeling of gravity, friction, collisions, and other physical properties, enabling developers to test and evaluate their robot's performance under various conditions. Gazebo also provides a wide range of sensors that can be added to robots in the virtual environment, including cameras, lidars, and sonars. These sensors can be used to gather data about the robot's surroundings, allowing for the testing and refinement of perception and navigation algorithms.

Another key feature of Gazebo is its modular architecture, which allows users to easily customize and extend the software for their specific needs. Gazebo is built using a plugin system, which enables developers to add new features and capabilities to the software without needing to modify the core codebase. This flexibility has made Gazebo a popular tool in research and industry, as it can be adapted to suit a wide range of applications.

In addition to its simulation capabilities, Gazebo also provides a user-friendly interface for designing and configuring robot models and environments. Users can import 3D models of robots and objects, and then place them in a virtual environment using Gazebo's graphical interface. Gazebo also provides tools for defining the robot's kinematics and dynamics, as well as for setting up controllers and sensors. This allows users to create complex robot systems and test them in a virtual environment.

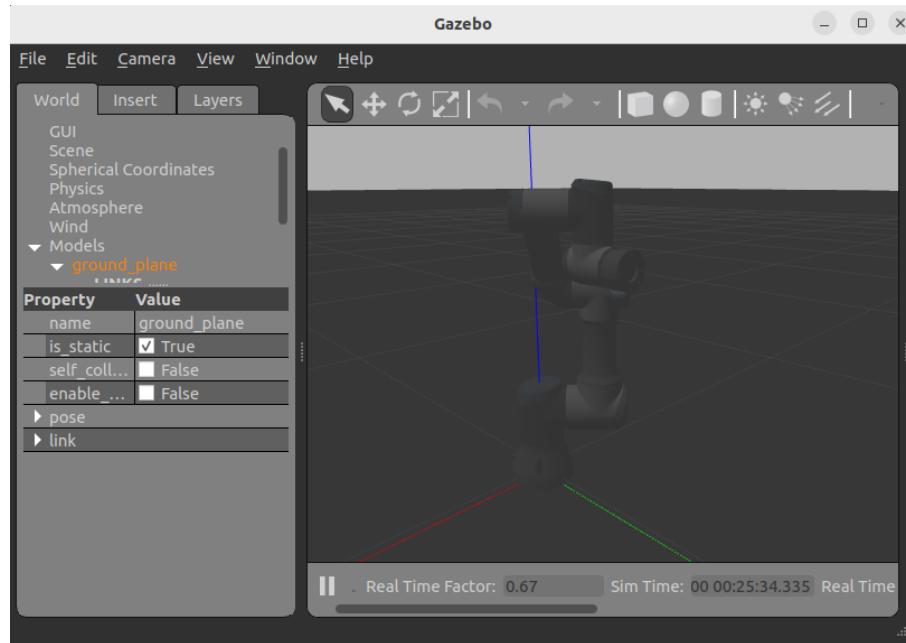


Figure 6. Gazebo graphic interface with UR3 model. Source: Created by the author.

3.3. Universal Robots Simulator

The second simulator used for the UR3 is a docker image provided by the Universal Robots company. As they state in their website [5], it is a simulation tool design to test the drivers for their robots in ROS (URSim).

In appearance, it has the same interface and configuration methods as PolyScope does, in the real Fieldbus Monitor attached to the UR robots. The interesting part of this simulator is that it is already configured and ready to use with ROS controllers. With this feature, the developer can visualize an image of the robot, that moves and follows the commands given either by manual control ("FreeDrive") or by an external control driver. The simulator can interact with the same interfaces as the real robot in PolyScope.

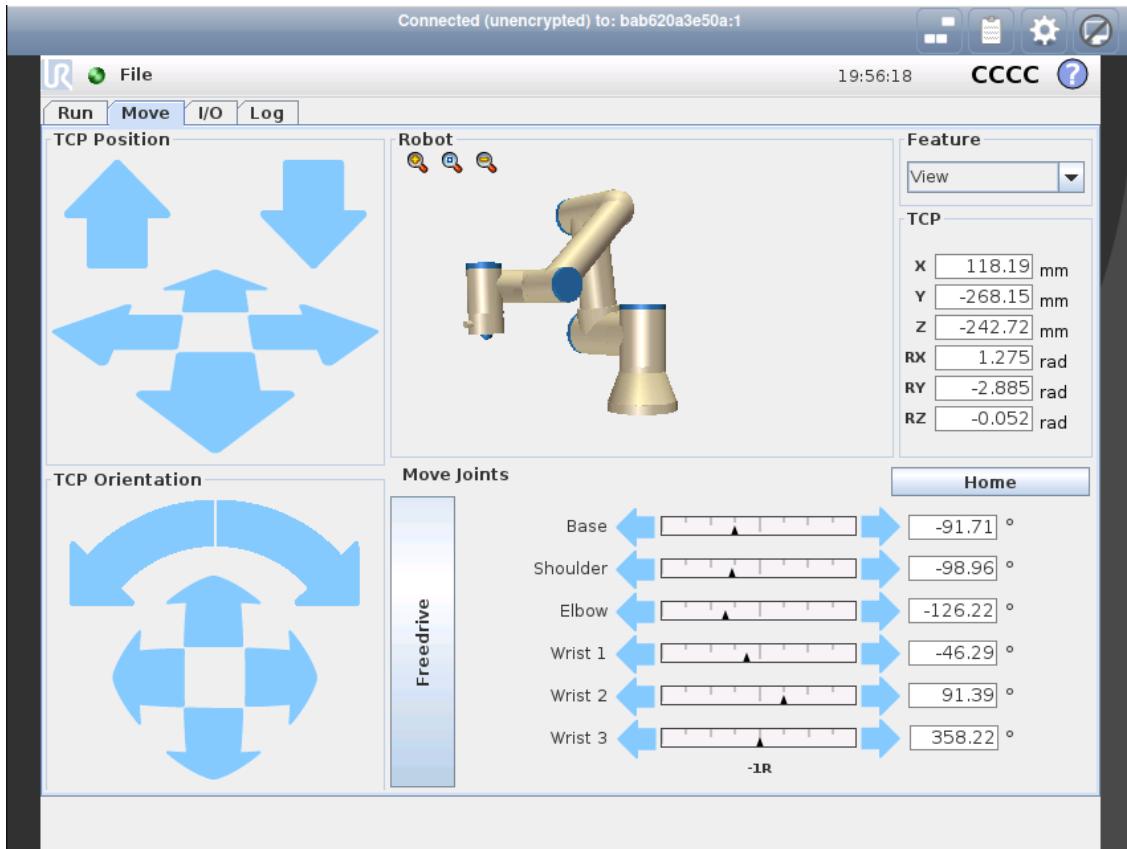


Figure 7. URSim FreeDrive interface. Source: Created by the author.

The UR Simulator, included in the UR ROS2 Driver package, provides a convenient and easy-to-use simulation solution for developers working with UR robots. Its pre-configured setup allows for quick setup and deployment. However, compared to Gazebo, the UR Simulator has limitations in terms of environment modification and task simulation. Despite these limitations, the UR Simulator is still a useful tool for testing and observing the robot's control behavior, providing valuable insights before deployment on the real robot.

3.4. RViz 2

The next item to cover inside the Software section of the document is the visualization tool of ROS2: RViz 2.

RViz 2 is ROS2 3D visualization tool for robotics applications that provides an interactive display of sensor data and robot models. Compared to its predecessor, RViz, it presents enhanced performance, flexibility, and extensibility. RViz 2 enables developers to create custom plugins, modify the interface, and integrate it with other ROS2 tools. The tool allows developers to visualize the robot's movements and interactions in real-time and quickly identify potential issues or errors.

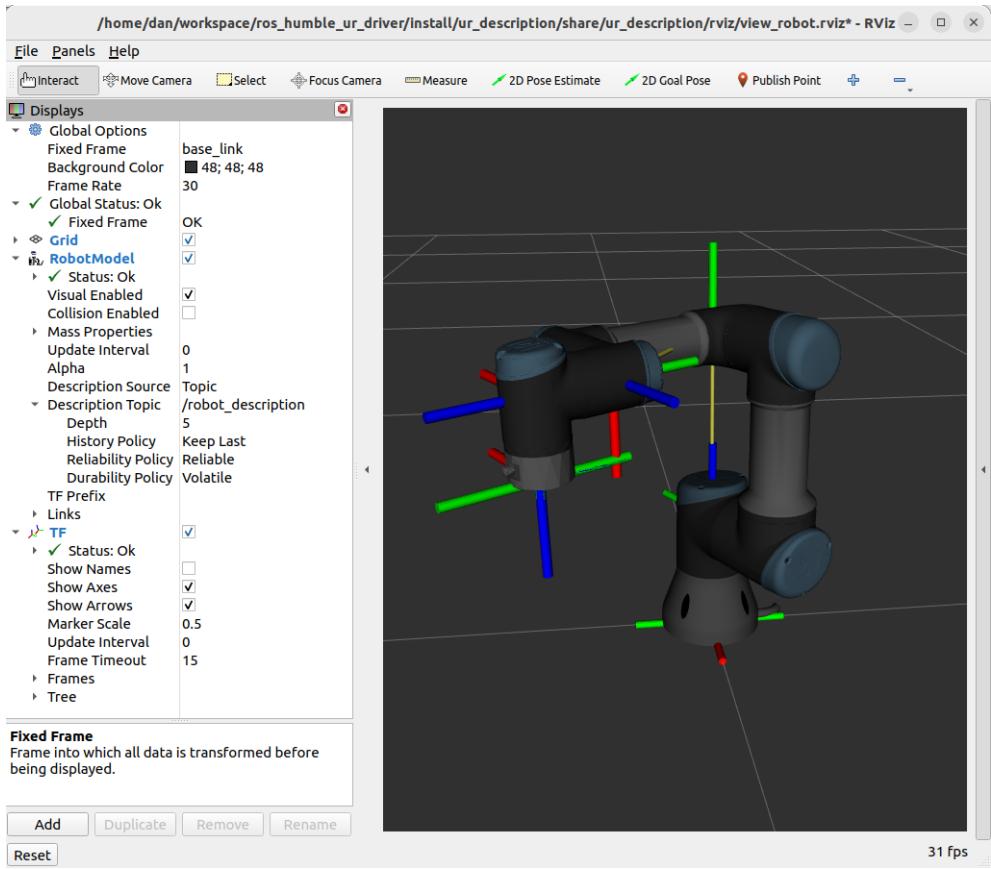


Figure 8. RViz interface with UR3 model. Source: Created by the author.

RViz 2 offers a suite of powerful visualization tools to help developers better understand and debug their robots. These tools include displays for sensor data (e.g. lidar scans, camera images), trajectory visualization, and robot state monitoring. To access this information, RViz 2 uses data published in the topics of the application. It can also send data to the robot or other components of the system by publishing messages into those topics. MoveIt 2 is an example of this, and it will be covered in section 13. MoveIt 2 of the document.

Moreover, RViz 2 allows developers to create and save custom layouts that organize and arrange the various displays and tools in a way that suits their workflow. Developers can create new layouts from scratch or modify existing ones, and then save them for future use.

This tool is fully integrated with ROS2, which means that developers can easily visualize and debug their ROS2-based robotic systems, in this case UR3 control.

3.5. PlotJuggler

Plot Juggler is a powerful data analysis tool that has gained popularity in the ROS 2 community due to its ease of use and flexibility visualization capabilities. Whether you need to process, analyze, or visualize large amounts of data generated by your robotic systems, Plot Juggler provides a user-friendly and efficient way to do so, making it an essential tool for anyone working with ROS 2.

One of the main features of Plot Juggler is its ability to handle multiple data sources and formats, including ROS messages, CSV files, and custom data logs. This means that you can easily import

and merge data from different sources into a single workspace, allowing you to perform complex analyses and comparisons across your data sets.

Another key feature of Plot Juggler is its real-time plotting capability, which enables you to stream your data as it is being generated by your robotic systems. This feature is particularly useful for debugging and monitoring purposes, as it allows you to quickly identify anomalies, trends, and patterns in your data and take appropriate actions in real-time.

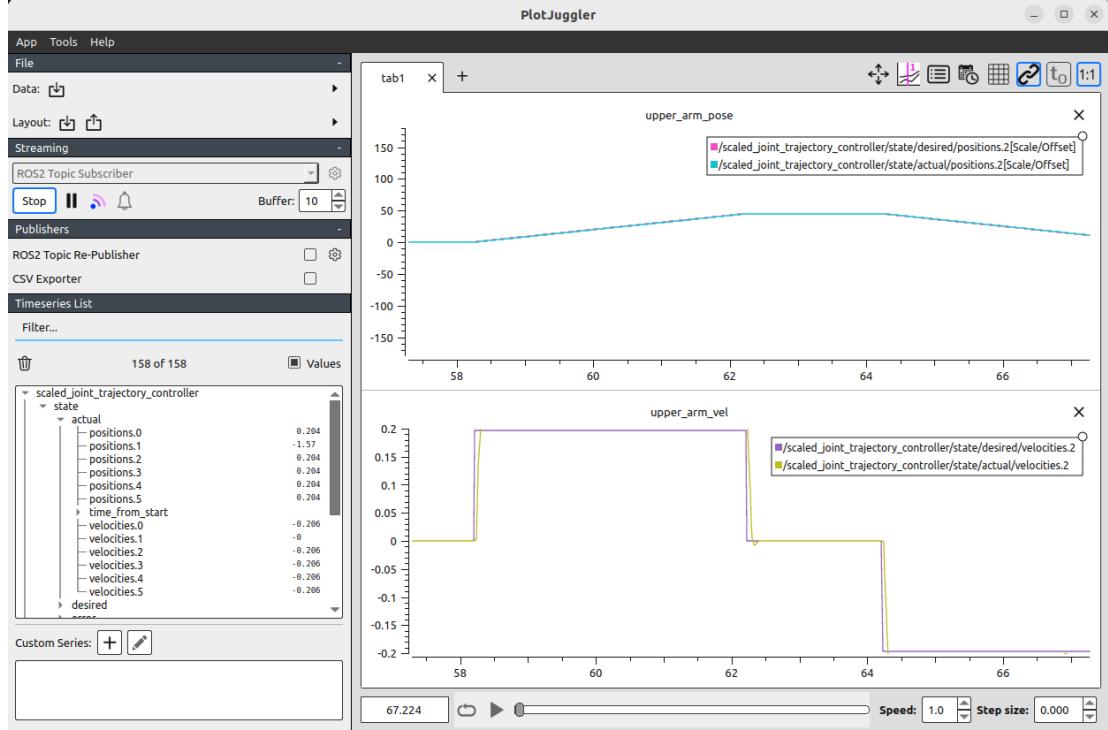


Figure 9. PlotJuggler layout graphing real-time data stream from ROS2 topics. Source: Created by the author.

In addition to real-time plotting, Plot Juggler offers a wide range of visualization options, including scatter plots, histograms, time series plots, and more. These visualizations can be customized and combined in various ways to provide a comprehensive and insightful view of your data.

Finally, Plot Juggler comes with a set of advanced data analysis and processing tools, such as filtering, built-in transformations (integral, derivative, moving average, etc.), interpolation, and regression, that can help you extract insights from your data. It also counts with a function editor that allows you to design and implement your own equations. These tools can be applied to individual data streams or across multiple data sets, and the results can be exported or saved for further analysis.

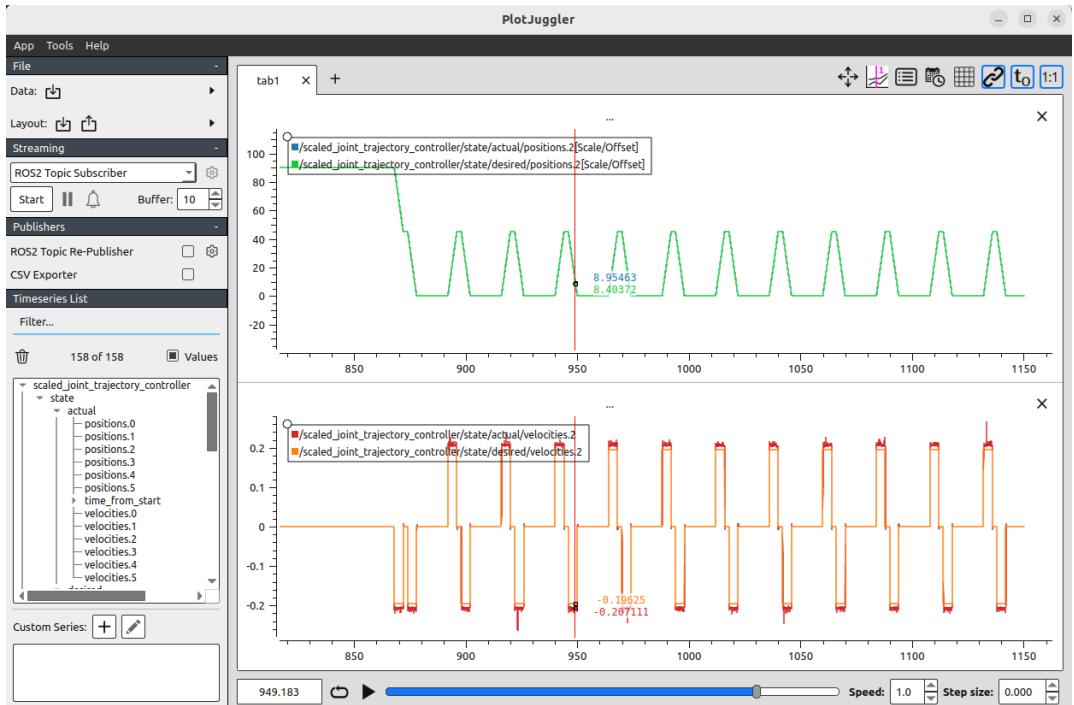


Figure 10. PlotJuggler layout graphing stored data from ROS2 topics. Source: Created by the author.

4. HARDWARE

4.1. Universal Robots UR3

The Universal Robots UR3 robot arm is an ideal solution for a wide range of industrial collaborative tasks. With its compact size and 3kg payload capacity, it is suitable for assistant workbench tasks like assembly, gluing or screwing. This cobot is the smallest member of the CB3 – Series and weighs only 11kg. Its precision is up to 0.1mm, and each of its six separate joints present a 360-degree rotation.



Figure 11. Universal Robots CB3 – Series UR3. Source: UR3 3D Model [72].

One of the key advantages of the UR3 is its lightweight design, which allows for easy movement and relocation without requiring a robust surface for mounting. The robot comes with its own control panel and software, featuring a user-friendly interface that simplifies the programming and manipulation of the robot.



Figure 12. Control panel with touch screen, and graphical interface provided by Universal Robots for their cobots. Source: Do more than just keep up. Try our virtual simulator [69].

The UR3 is an excellent choice for small-scale industrial tasks or research projects, owing to its structure and the tools provided by Universal Robots. The company has a strong connection to the open-source robot software community, which means that all the cobots in the CB3 – Series have different packages and drivers that are compatible with ROS or ROS2. This feature makes it an ideal fit for the current project, which relies heavily on these ROS2 packages and was the main reason why the UR3 robot was selected.

5. ROS2 SETUP AND INSTALLATION

This section will explain how to install and setup ROS2 Humble in a computer.

Ubuntu version: Ubuntu Jammy 22.04

All the installing instructions were based on the ROS2 Humble Documentation website: ROS2 Documentation: Humble. Installation/Ubuntu (Debian) [46]. Take into consideration that due to the frequent updates of the documentation, the steps explained in this document could have changed. If so, navigate the previous website and follow the instructions given.

Steps to follow:

1. Make sure your locale supports UTF-8.

In Shell #1:

```
locale # check for UTF-8
```

```
# If you do not have UTF-8, run the next lines
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale # verify settings
```

Expected output:

```
dan@dan-HP-Z240-Tower-Workstation:~$ locale
LANG=en_US.UTF-8
LANGUAGE=
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

2. Setup the sources to add the ROS2 apt repository.

Ensure Ubuntu Universe repository is enabled.

In Shell #1:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Add ROS2 GPG key with apt.

In Shell #1

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
/usr/share/keyrings/ros-archive-keyring.gpg
```

Add the repository to the sources list.

In Shell #1:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo
$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

3. Update and Upgrade the apt repositories

In Shell #1:

```
sudo apt update
sudo apt upgrade
```

This need to be done because ROS2 packages are usually built on updated Ubuntu systems. In fact, you will always want to execute this step previously to install any new package.

4. Install ROS2 packages.

There are different options to install ROS2 Humble, but the one used in this project is the desktop-full one. This command will install all ROS2 Humble basic packages as well as some GUI, such as RViz and Gazebo, and some demo and tutorial packages.

In Shell #1:

```
sudo apt install ros-humble-desktop-full
```

The installation will take some time, as there are few hundreds of packages to be added.

After that, run the next command to install Development tools for ROS2, such us colcon, that will allow you to compile and build packages.

In Shell #1:

```
sudo apt install ros-dev-tools
```

All the packages will be located at the /opt/ros/humble folder of the system.

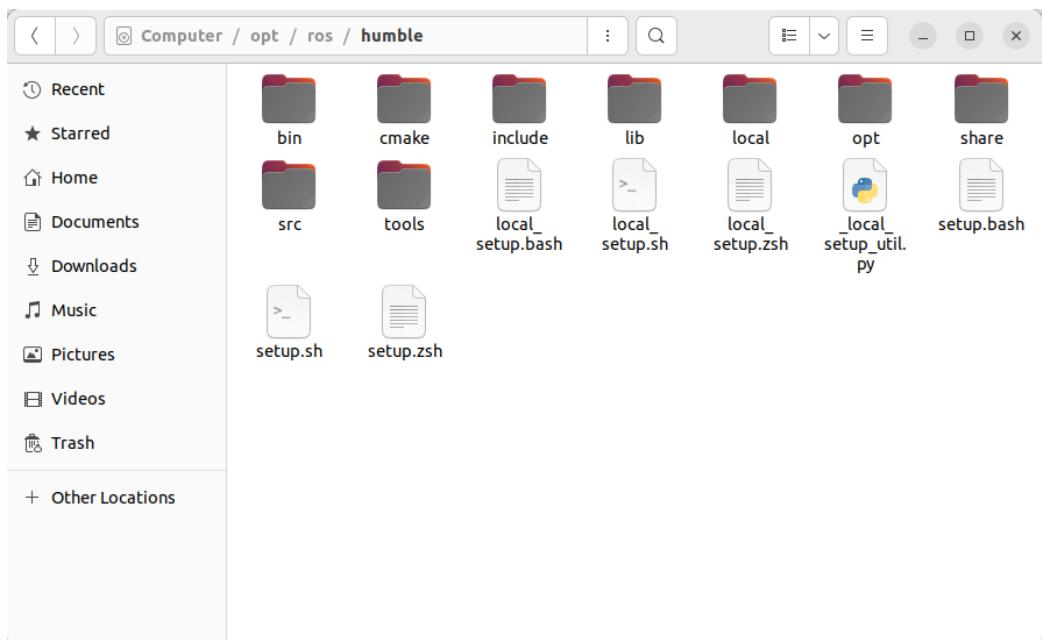


Figure 13. ROS2 Humble underlay workspace. Source: Created by the author.

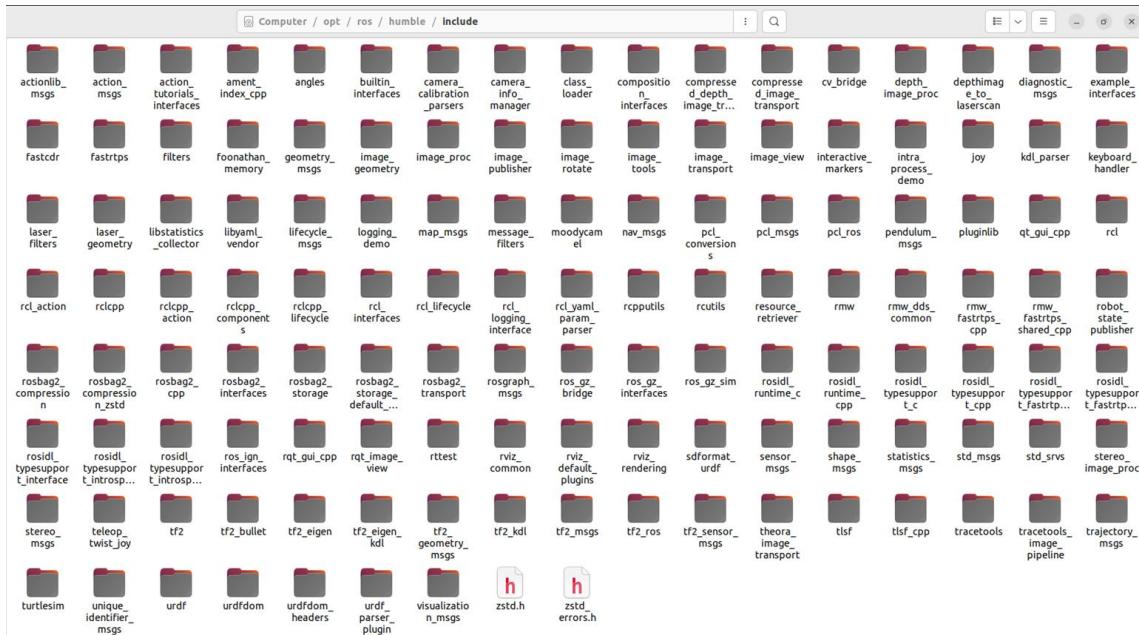


Figure 14. Installed packages inside /include folder. Source: Created by the author.

5. Environment setup to be able to use ROS2.

In order to use ROS2 packages and commands, you will have to run the next command in every new shell that you open.

In Shell #1:

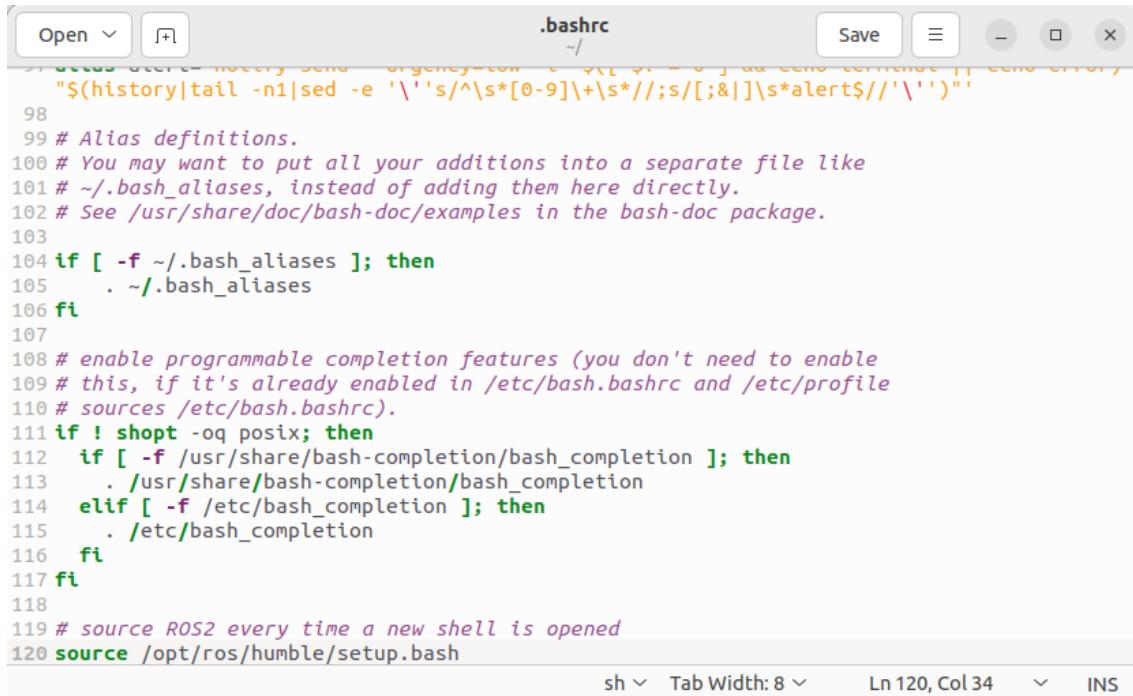
```
source /opt/ros/humble/setup.bash
```

Another option is to add the previous line into the `~/.bashrc` file of your user, so that it is executed every time you open a new shell.

In Shell #1:

```
gedit ~/.bashrc
```

Go to the end of the file and copy the source `/opt/ros/humble/setup.bash` line.



```

.bashrc
~/
98
99 # Alias definitions.
100 # You may want to put all your additions into a separate file like
101 # ~/.bash_aliases, instead of adding them here directly.
102 # See /usr/share/doc/bash-doc/examples in the bash-doc package.
103
104 if [ -f ~/.bash_aliases ]; then
105     . ~/.bash_aliases
106 fi
107
108 # enable programmable completion features (you don't need to enable
109 # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
110 # sources /etc/bash.bashrc).
111 if ! shopt -oq posix; then
112     if [ -f /usr/share/bash-completion/bash_completion ]; then
113         . /usr/share/bash-completion/bash_completion
114     elif [ -f /etc/bash_completion ]; then
115         . /etc/bash_completion
116     fi
117 fi
118
119 # source ROS2 every time a new shell is opened
120 source /opt/ros/humble/setup.bash

```

Figure 15. Line to source ROS2 when a new shell is opened added to the bashrc file. Source: Created by the author.

Save the file.

Now you will not need to execute source /opt/ros/humble/setup.bash when you open a new terminal.

One optional step could be to implement the colcon autocomplete command. Colcon is the tool used to build packages and, as with the rest of the commands, it is convenient to have the tab autocomplete option while coding in any shell. To do so implement the following line at the end of the ~/.bashrc file.

```
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```

In Shell #1:

```
gedit ~/.bashrc
```

Copy the line at the end of the file.

```

98
99 # Alias definitions.
100 # You may want to put all your additions into a separate file like
101 # ~/.bash_aliases, instead of adding them here directly.
102 # See /usr/share/doc/bash-doc/examples in the bash-doc package.
103
104 if [ -f ~/.bash_aliases ]; then
105     . ~/.bash_aliases
106 fi
107
108 # enable programmable completion features (you don't need to enable
109 # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
110 # sources /etc/bash.bashrc).
111 if ! shopt -oq posix; then
112     if [ -f /usr/share/bash-completion/bash_completion ]; then
113         . /usr/share/bash-completion/bash_completion
114     elif [ -f /etc/bash_completion ]; then
115         . /etc/bash_completion
116     fi
117 fi
118
119 # source ROS2 every time a new shell is opened
120 source /opt/ros/humble/setup.bash
121 # enable colcon autocomplete
122 source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash

```

Figure 16. Line to enable colcon autocomplete added to the bashrc file. Source: Created by the author.

Your system is now ready to start developing any ROS2 application.

6. ROS2 CONTROL FRAMEWORK

Once ROS2 Humble is installed in the system, there are several tutorials and websites that explain in detail the steps to follow to understand ROS2 methods. Node definition, create communication streams (topics, services or actions) and complex robot theory, are some examples covered in these tutorials. Some recommended websites to start learning ROS2 are The Construct [66] (paid option, more versatile) and ROS Org Tutorials [48] (free option, simpler).

Nevertheless, this document will assume some basic ROS2 understanding and, therefore, it will move straight up to explain the fundamentals of ROS2 Control Framework.

ROS2 Control Framework is an implemented tool in ROS2 focused on real-time control of any kind of robot using ROS2. This will be the framework used alongside the rest of the project to move the UR3, focusing on joint trajectory control, but diving into several different controllers, too. ROS2 Control Framework is based on two ROS packages:

- ros2_control
- ros2_controllers

Together, these two packages shape the architecture of ROS2 Control Framework shown in **Figure 17**.

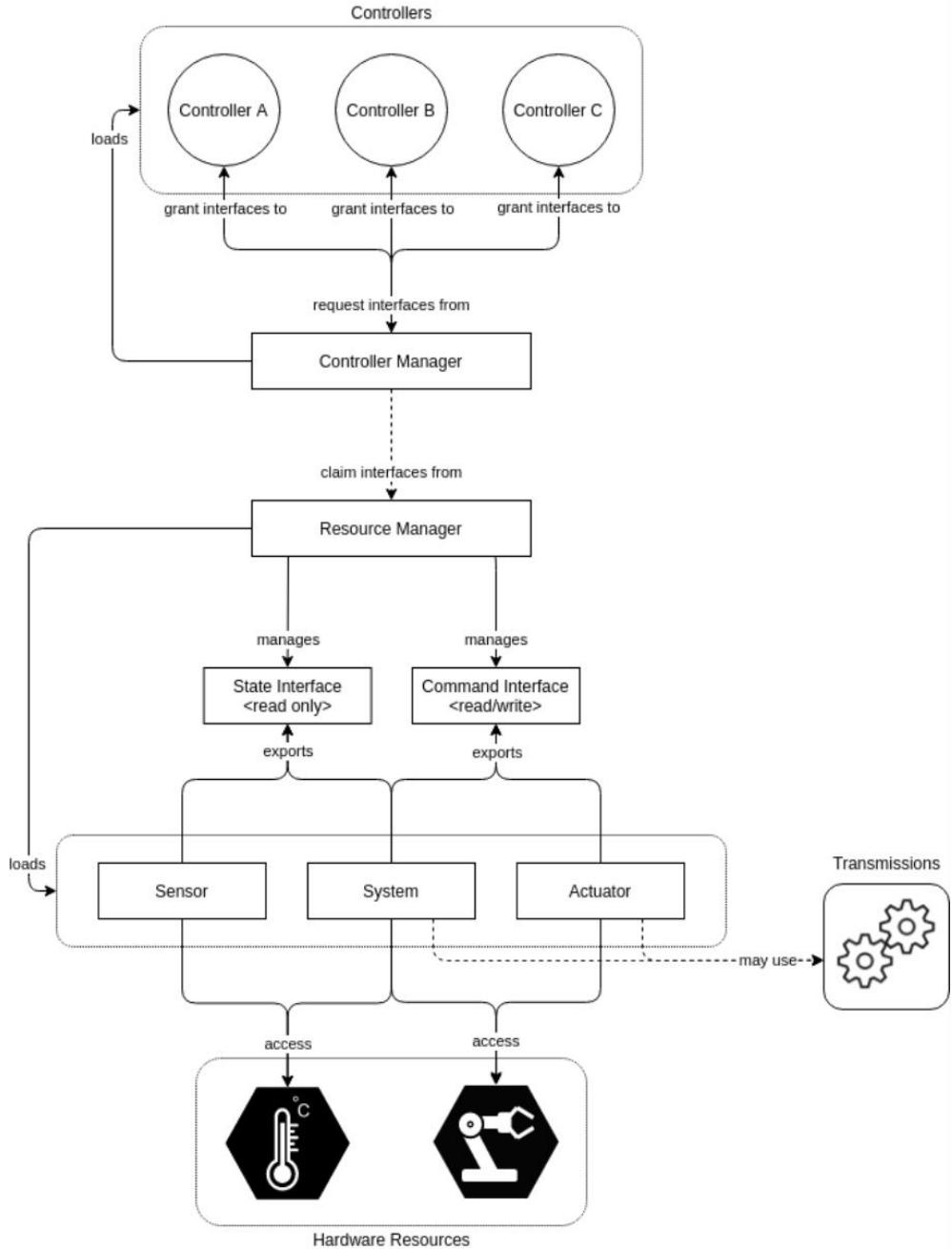


Figure 17. Architecture of ROS2 Control Framework. Source: [ros2_control Maintainers \[64\]](#).

The architecture is centralized in the Controller Manager (CM) node. Its functionality is to connect the controllers and the hardware (or simulation) components of the application (not real hardware), but it is also the node that lets users enter the application through ROS2 services, allowing them to manage the controllers and observe the states of the hardware components.

The CM works with controllers and hardware interfaces to give controllers access to the hardware when activated, and using the update() method, it reads data from hardware elements, updates the outputs of the controllers and writes back the results to the components.

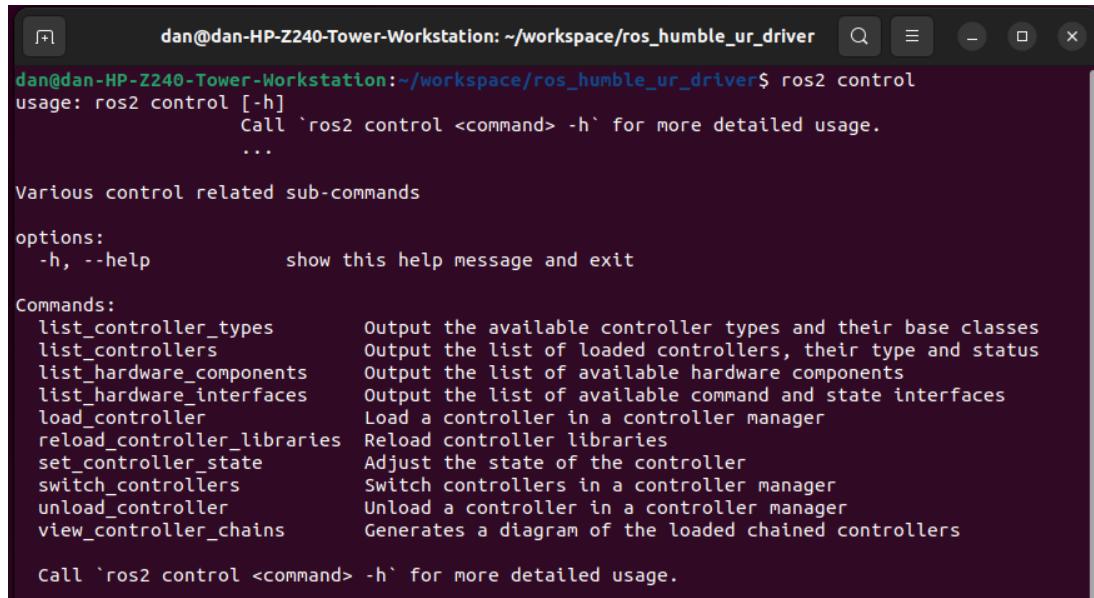
Another key component of the framework architecture is the Resource Manager (RM). It works together with the CM, and its main application is to abstract hardware components using plugin libraries. This allows to implement separated state and command interfaces for the components.

The way it helps the CM to communicate with hardware components is using the read() and write() methods.

On the opposite side of the hardware components are the controllers, which take the current states of the components provided by the RM and CM and compute the error to reach the desired state. Once they obtain the error, they send it back to the components using its own interface.

These controllers are derived from the controller_interface package inside the ros2_control package and exported as plugins, stored in the ros2_controllers package.

The way users access this framework is through the CM, using its services. Users can call these services through shells, using the command line, or through nodes, using files. There is also a user-friendly Command Line Interface that provides a range of common commands and supports auto-complete. The base command for this Command Line Interface is ros2 control.



```
dan@dan-HP-Z240-Tower-Workstation:~/workspace/ros_humble_ur_driver$ ros2 control
usage: ros2 control [-h]
                    Call `ros2 control <command> -h` for more detailed usage.
...
Various control related sub-commands
options:
-h, --help           show this help message and exit
Commands:
list_controller_types      Output the available controller types and their base classes
list_controllers            Output the list of loaded controllers, their type and status
list.hardware_components    Output the list of available hardware components
list.hardware_interfaces    Output the list of available command and state interfaces
load_controller             Load a controller in a controller manager
reload_controller_libraries Reload controller libraries
set_controller_state        Adjust the state of the controller
switch_controllers          Switch controllers in a controller manager
unload_controller           Unload a controller in a controller manager
view_controller_chains      Generates a diagram of the loaded chained controllers
call `ros2 control <command> -h` for more detailed usage.
```

Figure 18. ROS2 control commands. Source: Created by the author.

In order for the application to work with real hardware, the hardware components are the ones who communicate the orders and receive the feedback from them, representing their abstraction in the control framework. These components are exported as plugins, same as controllers, and there are three main types: sensors, actuators and systems (combination of sensors and actuators).

Hardware components are defined in the description packages of the application, using URDF or xacro files. In these files the ROS2 Control Framework uses the <ros2_control> tag to describe its components. Inside the description package you can also find configuration files for the components, setting values to certain parameters like initial position for the joints.

6.1. ros2_control package

Inside this package, there are all the packages needed to create the core of the framework architecture described previously, meaning the Controller Manager, the Resource Manager and the different interfaces that they need to communicate with both ends of the structure (hardware components and controllers) and the user.

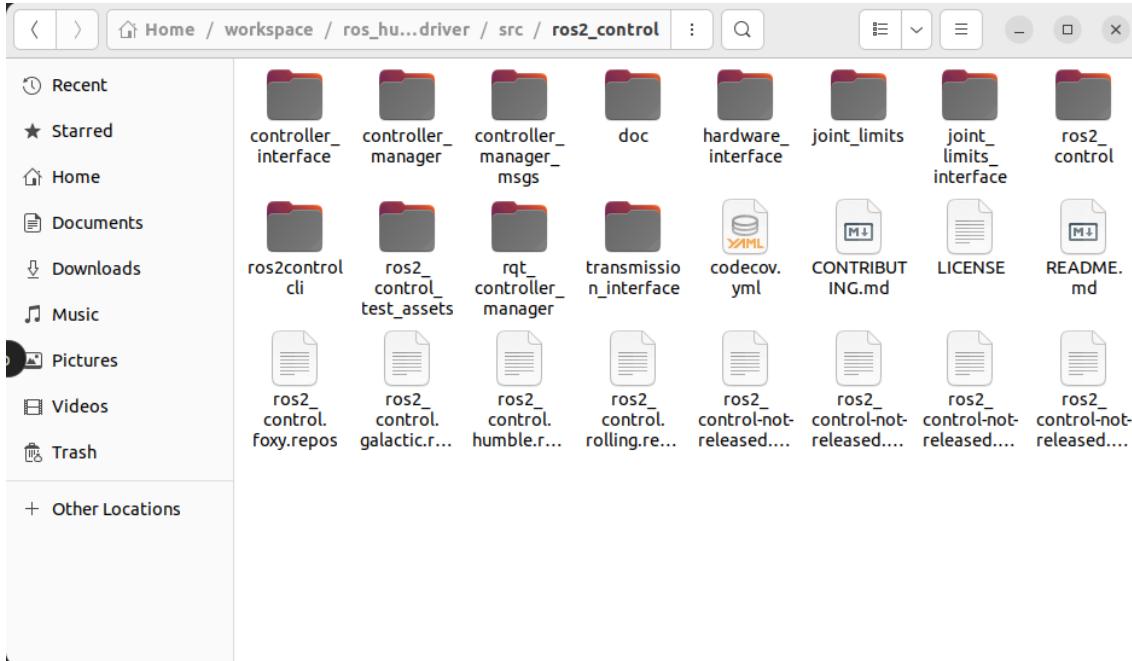


Figure 19. Packages included inside the ros2_control package. Source: Created by the author.

6.2. ros2_controllers package

Inside this package, there are widely used controllers and broadcasters (used to broadcast actuators or sensors data) that are ready-to-use in any application developed with ROS2. The controllers are the following (extracted directly from [45]):

- **position_controllers:** Command a desired position to the HardwareInterface.
 - joint_position_controller - Receives a position input and sends a position output, just transferring the input with the forward_command_controller.
 - joint_group_position_controller - Set multiple joint positions at once.
- **velocity_controllers:** Command a desired velocity to the HardwareInterface.
 - joint_position_controller - Receives a position input and sends a velocity output, using a PID controller.
 - joint_velocity_controller - Receives a velocity input and sends a velocity output, just transferring the input with the forward_command_controller.
 - joint_group_velocity_controller - Set multiple joint velocities at once.
- **effort_controllers:** Command a desired effort(force/torque) to the Hardware Interface.
 - joint_position_controller - Receives a position input and sends an effort output, using a PID controller.
 - joint_group_position_controller - Set multiple joint positions at once.
 - joint_velocity_controller - Receives a velocity input and sends an effort output, using a PID controller.

- joint_effort_controller - Receives an effort input and sends an effort output, just transferring the input with the forward_command_controller.
- joint_group_effort_controller - Set multiple joint efforts at once.
- **joint_trajectory_controllers:** Extra functionality for splining an entire trajectory. It implements the previous controllers in order to compute a trajectory.

Note: the forward_command_controller is a controller that passes the same type of data that receives (if it receives a position command, it sends a position command).

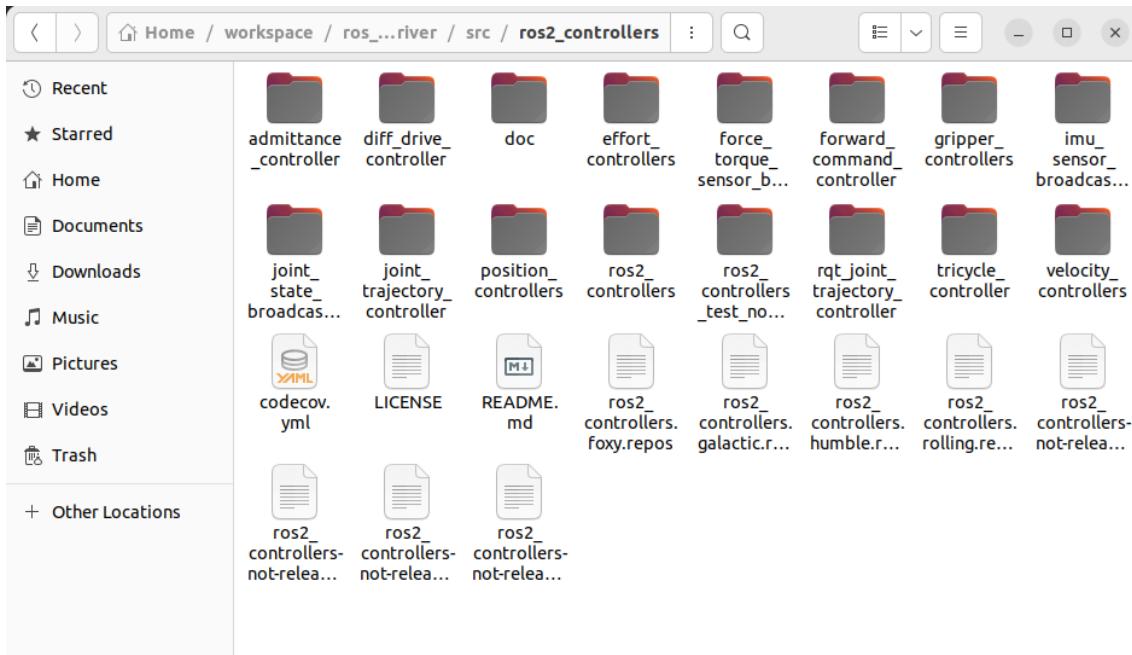


Figure 20. Packages included inside the ros2_controllers package. Source: Created by the author.

7. UR ROS2 DRIVER

To control the UR3 robot, Universal Robots supplies developers with different packages that will let them build the control framework around their UR robot models in an easier way. These packages can be found and download from GitHub [28], and they are grouped in a package called Universal_Robots_ROS2_Driver.

There are different ways to install this package, from binary packages, or building from source. Binary packages is the easiest way, as it only requires the user to execute the following line:

In Shell #1:

```
sudo apt-get install ros-humble-ur-robot-driver
```

This will install the packages from the Universal_Robots_ROS2_Driver in the underlay workspace, the one created when we installed ROS2 Humble. That is convenient if you already know how to set up the robot, meaning you have installed the description package of the robot, the ros2_control and ros2_controllers packages, and the rest packages that the environment would need to start the simulation or the real hardware trial. Moreover, with binary packages,

you will not be able to access or edit most of the files in the packages. This option works better if you already understand how it works or if you are just following a tutorial.

The second option, build from source, is the one used for this project. It is more complicated to handle in terms of dependencies and it will require the creation of an overlay workspace, but it provides all the packages needed to set up the framework for the application, as well as access to all the files.

It is essential to acknowledge the dynamic nature of the packages to be installed, as they undergo frequent updates. Consequently, there may be occasional errors that lead to workspace building failures. Throughout the project, three errors were encountered while building from source- two at the beginning and a new one towards the end, following a system crash that necessitated a hard drive reset. Fortunately, developers typically work to address such issues. In this case, the first two errors were resolved before the second installation.

During this installation guide, the solution to the current building error will be provided. Take into consideration that in the near future this error will be solved by the developers and new errors might be encountered.

Steps to follow to build from source:

1. Create a new ROS2 workspace.

In Shell #1:

```
export COLCON_WS=~/workspace/ros_humble_ur_driver  
mkdir -p $COLCON_WS/src
```

2. Clone relevant packages and install dependencies.

In this step we will not only install the UR ROS2 Driver package, but also all the packages needed to create the control framework for the UR3. These packages are Universal_Robots_ROS2_Driver, Universal_Robots_ROS2_Description, Universal_Robots_Client_Library, ur_msgs, control_msgs, kinematics_interface, and the previously analyzed ros2_control and ros2_controllers.

In Shell #1:

```
cd $COLCON_WS  
git clone https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver.git  
src/Universal_Robots_ROS2_Driver  
vcs import src --skip-existing -input  
src/Universal_Robots_ROS2_Driver/Universal_Robots_ROS2_Driver.humble.repos
```

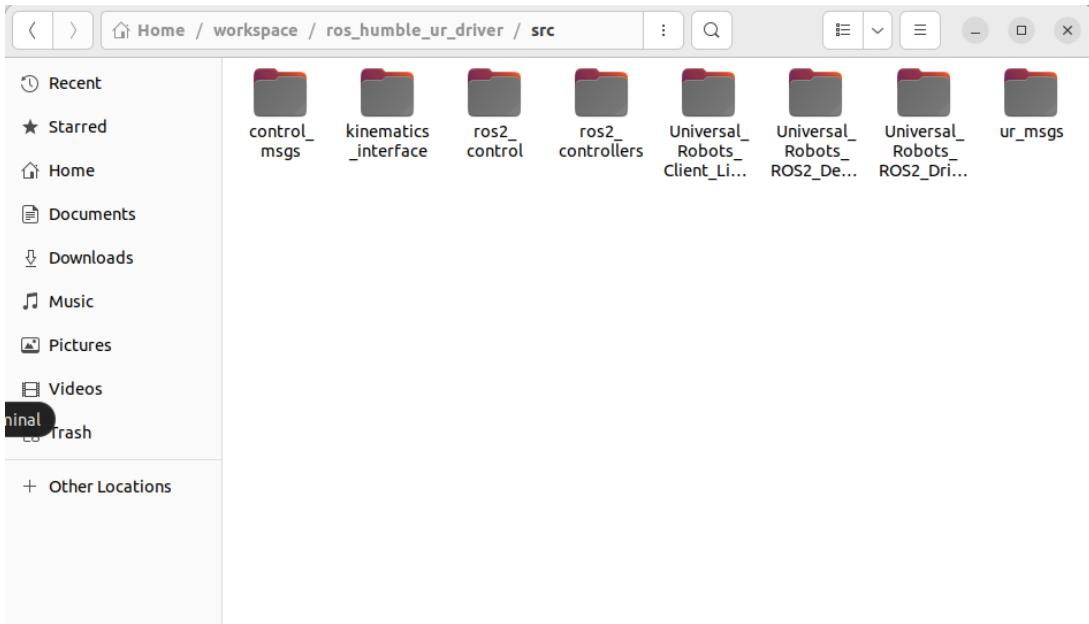


Figure 21. Packages added to the new workspace /src folder. Source: Created by the author.

3. Initialize rosdep, run it and build the workspace.

In Shell #1:

```
sudo rosdep init
rosdep update
rosdep install --ignore-src --from-paths src -y
colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release
```

With the current version of the files, once the colcon build is executed, an error will show up in the terminal output.

This error will indicate that the function `publish_state` in line 262 from the `~/workspace/ros_humble_ur_driver/src/Universal_Robots_ROS2_Driver/ur_controllers/src/scaled_joint_trajectory_controller.cpp` has extra arguments. The original function in line 249 from the `~/workspace/ros_humble_ur_driver/install/joint_trajectory_controller/include/joint_trajectory_controller/joint_trajectory_controller.hpp` has no argument called time.

In order to solve the problem, open the `scaled_joint_trajectory_controller.cpp`, go to line 262, eliminate the time argument from the function call, and save the file (Ctrl+S).

4. Rebuild the workspace and source it.

In Shell #1:

```
colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release
source install/setup.bash
```

Your new overlay workspace should be built and operational to start testing control applications with a simulated or real UR3 robot.

In the following sections, a general explanation about the Universal Robots packages added will be given, focusing on Universal_Robots_ROS2_driver, Universal_Robots_ROS2_Description and Universal_Robots_Client_Library. They will also contemplate how these packages relate.

7.1. Universal_Robots_ROS2_Driver

Inside the Universal_Robots_ROS2_Driver package, there are 7 more packages.

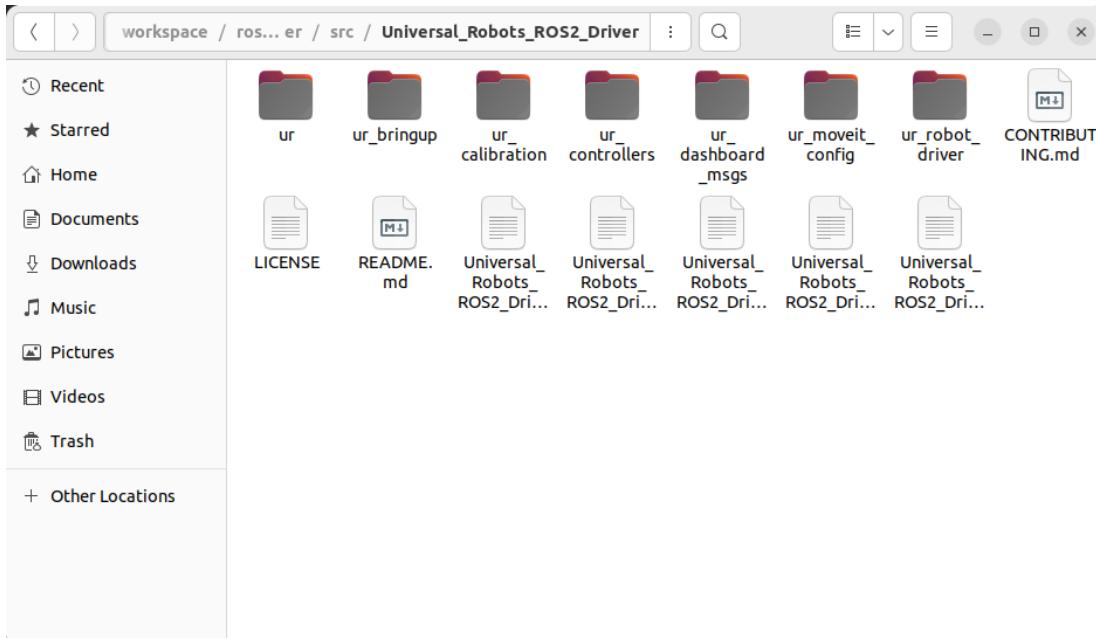


Figure 22. Packages installed inside the Universal_Robots_ROS2_Driver package. Source: Created by the author.

The ur package, is the metapackage for Universal Robots, it also stores a changelog file where the updates are noted.

The ur_bringup package is the deprecated version of the ur_robot_driver. Its files were used in previous versions, but on June 20th 2022 all the files were transferred to the ur_robot_driver package and since then all the updates and modifications were developed in that package.

The ur_calibration package function is to extract the factory calibration data from a UR robot and modify it so it can be used by ur_description, a package that contains the URDF model of the robot and will be explained in more detail in section 7.2 Universal_Robots_ROS2_Description.

The calibration of the robot is not strictly necessary to control the robot with the driver, but if a higher resolution application is required, such as pick&place tasks, calibrating the robot is highly recommended as the end effector positions could be off by several centimeters.

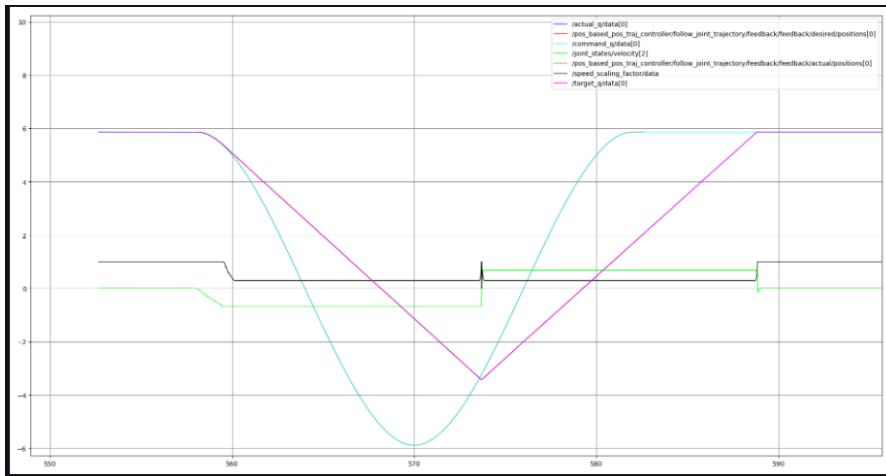
The ur_controllers package contains controllers and hardware interfaces that are not available in the ros2_control package and that support features offered by the UR robots.

On the one hand, it provides two interfaces. A speed_scaling_interface that reads the value of the current speed scaling from controllers (floating value between 0 and 1, 0 being min speed and 1 max speed). A scaled_joint_command_interface that gives access to joint values and commands using the speed scaling values.

On the other hand, it provides two controllers. A speed_scaling_state_controller, working similar to a broadcaster, publishing current execution speeds of the joints into a topic interface (floating values between 0 and 1). A scaled_joint_trajectory_controller, comparable to the joint_trajectory_controller provided by the ros2_control package, but using the speed scaling reported to adjust the trajectory.

With the joint_trajectory_controller, the robot will follow a trajectory assuming always maximum velocity and acceleration supported by the joints. That could lead to path deviation due to reasons such as scaled down motions due to configured safety limits (slower motion than expected) or no movement caused by an emergency stop, between others.

An example of the problems presented by the joint_trajectory_controller is perfectly presented at the ur_controllers package GitHub documentation.

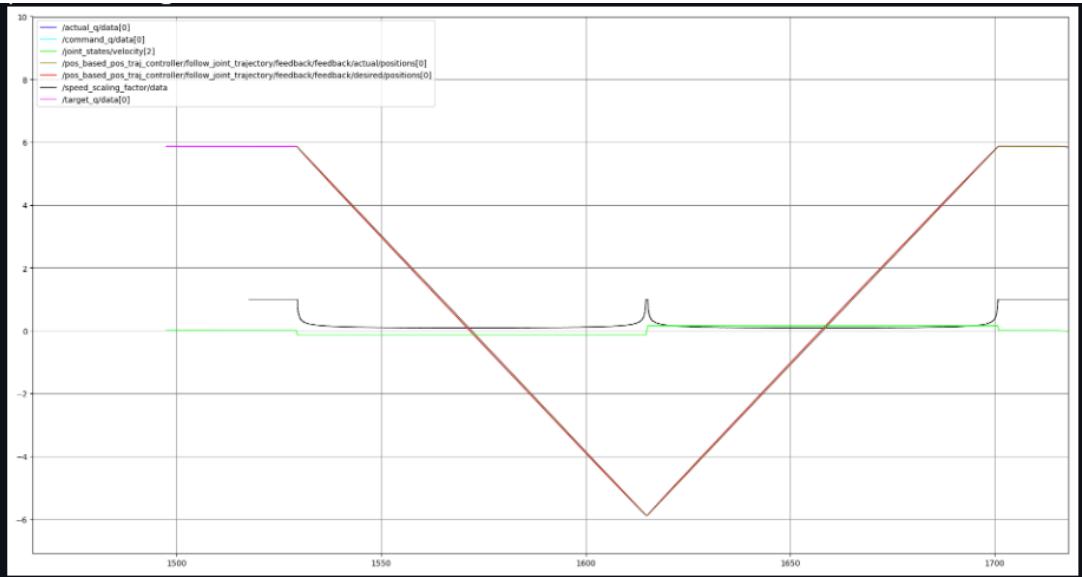


The graph shows a trajectory with one joint being moved to a target point and back to its starting point. As the joint's speed is limited to a very low setting on the teach pendant, speed scaling (black line) activates and limits the joint speed (green line). As a result, the target trajectory (light blue) doesn't get executed by the robot, but instead the pink trajectory is executed. The vertical distance between the light blue line and the pink line is the path error in each control cycle. We can see that the path deviation gets above 300 degrees at some point and the target point at -6 radians never gets reached.

Figure 23. Trajectory generated by joint_trajectory_controller and explanation of the results.

Source: GitHub ur_controllers [32].

Using the scaled_joint_trajectory_controller, because it implements the scaled speed data from the robot to generate the trajectory, the path will not experience any deviations due to configuration limitations or scaled down trajectories defined by the user (for example using the speed slider of the UR). In fact, if a protective stop occurred, the speed scaling would be 0 and the trajectory would not continue until the program is resumed.



The deviation between trajectory interpolation on the ROS side and actual robot execution stays minimal and the robot reaches the intermediate setpoint instead of returning "too early" as in the example above.

Figure 24. Trajectory generated by scaled_joint_trajectory_controller and explanation of the results. Source: GitHub ur_controllers [32].

The `ur_dashboard_msgs` package defines different types of messages, services and actions that can be used in the UR dashboard server, a server that permits remote control of UR robots. Together with the `control_msgs` and `ur_msgs` packages, conform the list of communication for controlling the UR3. These last two packages contain messages, services and actions useful for controlling robots. The first one could be used for any kind of robot while the second one is specific for UR robots.

The `ur_moveit_config` package is an example package with MoveIt 2 configuration for UR robots. The UR3 control using MoveIt 2 will be studied in section 13. MoveIt 2 of this document. In that section a detailed explanation about MoveIt 2 will be given, as well as `ur_moveit_config` usage in simulation and real UR3.

The `ur_robot_driver` package contains the real driver for the UR robots. This package is similar in functionalities to the `ros2_control` package, it contains all the files needed to set up the control framework for the UR robots. In addition, a series of launch files are included in order to ease the initialization of the control.

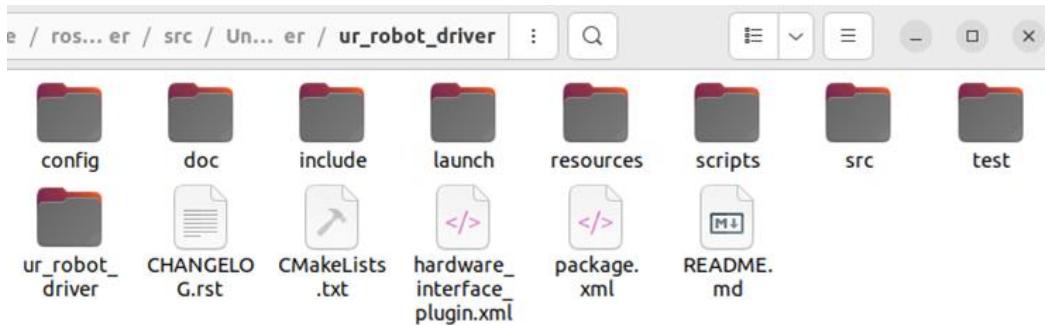


Figure 25. *ur_robot_driver* package. Source: Created by the author.

The main launch file in this package is the `ur_control.launch.py`. Understanding and going through this file will help understanding the rest of the package, as it launches all the nodes and starts the communications needed to control, in this case, the UR3.

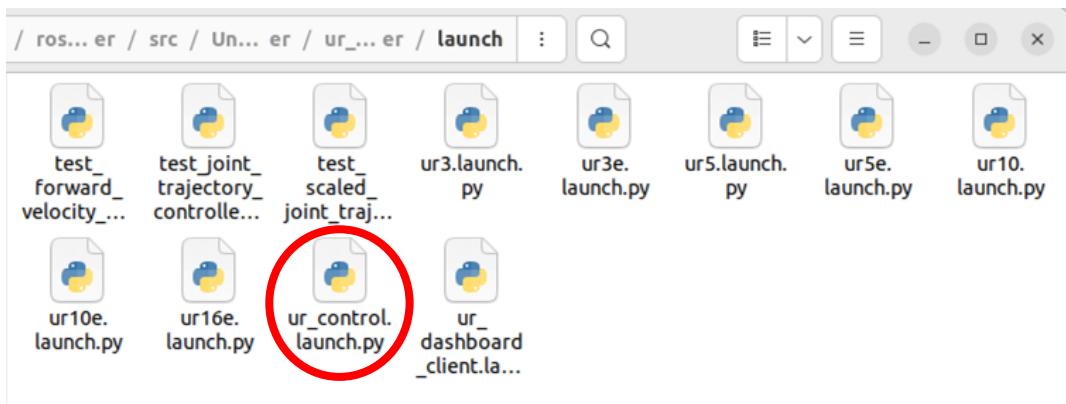


Figure 26. Launch folder and launch files of the *ur_robot_driver* package, `ur_control.launch.py` marked in red. Source: Created by the author.

Note: `ur_control.launch.py` code, together with other important files, will be added and commented in Appendix 1 Commented Code.

When launched, `ur_control.launch.py` will start a ros2 control node that will perform as Controller Manager together with:

- The robot description set up, obtained from the `Universal_Robots_ROS2_Description` package, used to generate the CM.
- The update rate of the communications, used to generate the CM.
- The configuration for the initial joint controllers, used to generate the CM.
- The robot state publisher to monitor the current state of the robotic system.
- The control spawner, a node that initiates the `joint_state_broadcaster`, the `speed_scaling_state_broadcaster`, the `force_torque_sensor_broadcaster` and the `forward_position_controller`. These controllers and broadcasters will be used by the `scaled_joint_trajectory_controller` to elaborate the trajectory of the robot. The spawner will also start the `io_and_status_controller`, to manage the I/O of the UR robot.
- Other nodes that will only activate depending on the arguments' values of the launch file when launched. These nodes will be named when explaining the arguments.

The arguments for the `ur_control.launch.py` file can be listed using:

In Shell #1:

```
ros2 launch ur_robot_driver ur_control.launch.py --show-args
```

The following will elaborate on the file's arguments, detailing their impact on the nodes launched by the file.

- *ur_type*: mandatory argument to select the UR robot type you are going to use (ur3, ur3e, ur5, ur5e, ur10, ur10e, or ur16e).
- *use_fake_hardware*: a boolean type argument, default to false, that switches between the two possible modes of operation, fake hardware (offline testing) or real hardware (real robot or simulation). This argument will determine what type of CM node the application uses, if using fake hardware, the CM will be loaded from the ros2_control package, if not, it will be loaded from the ur_robot_driver package. This is because, when using fake hardware, the system does not activate some of the interfaces or controllers as they are not needed or supported.

That can lead to deduce that the *ur_ros2_control_node* from the *ur_robot_driver* package is a similar version to the *ros2_control_node* from the *ros2_control* package, but it is designed to be a CM for the UR robots specifically.

Other nodes that will be initiated if *use_fake_hardware* is active are the *dashboard_client_node* (allows remote control of the UR robot), and *controller_stopper_node* (node to stop and restart controllers).

- *robot_ip*: mandatory argument to set the IP address by which the root can be reached. If using fake hardware, it must be set to *robot_ip*:=yyy.yyy.yyy.yyy.
- *safety_limits*: boolean argument that enables safety limits controller if true.
- *safety_psd_margin*: floating argument that sets the upper and lower limits for the safety limits controller. Default value set to 0.15.
- *safety_k_position*: integer argument that sets the k-position factor in the safety controller. Default value set to 20.
- *runtime_config_package*: package with the controller's configuration in its config folder. Set to *ur_robot_driver*.
- *controllers_file*: YAML file with the controller's configuration. Set to *ur_controllers.yaml*.
- *description_package*: description package with robot URDF/XACRO files. Set to *ur_description*.
- *description_file*: URDF/XACRO description file with the robot. Set to *ur.urdf.xacro*.
- *prefix*: prefix of the joint names, useful for multi-robot setup. Set to "" (empty).
- *fake_sensor_commands*: enable fake command interfaces for sensors used with fake hardware. Useful to set sensor values for offline testing. Set to false.
- *headless_mode*: enables headless mode for robot control. Headless mode does not require the "External Control" URCap to be installed in the robot, as the program is sent directly to the robot. This is not recommended. Set to false.
- *controller_spawner_timeout*: timeout when spawning controllers. Set to 10.
- *initial_joint_controller*: initially loaded robot controller. Set to *scaled_joint_trajectory_controller*.

- *activate_joint_controller*: activates the loaded joint controller and starts the *initial_joint_controller_spawner_stopped* and the *initial_joint_controller_spawner_started*. Set to true.
- *launch_rviz*: starts *rviz_node* and runs RViz2 visualization tool.
- *launch_dashboard_client*: starts *dashboard_client_node* if not using fake hardware.
- *use_tool_communication* and the rest of the tool's arguments: not used, only available for UR e-series. Set to false.
- *reverse_ip*: IP used by the robot controller to communicate back to the driver. Set to 0.0.0.0.
- *script_command_port*: port opened to forward script commands from the driver to the robot. Set to 50004.

The following figures show the main folders of the *ur_robot_driver* package and their files, most of them named and commented while explaining *ur_control.launch.py* performance.

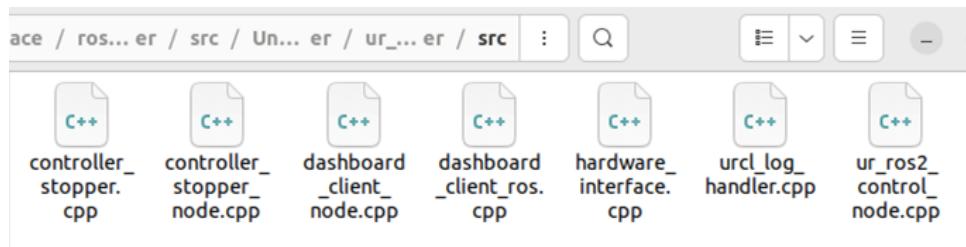


Figure 27. Src folder of the *ur_robot_driver* package, with the executables that start the nodes required by the driver.

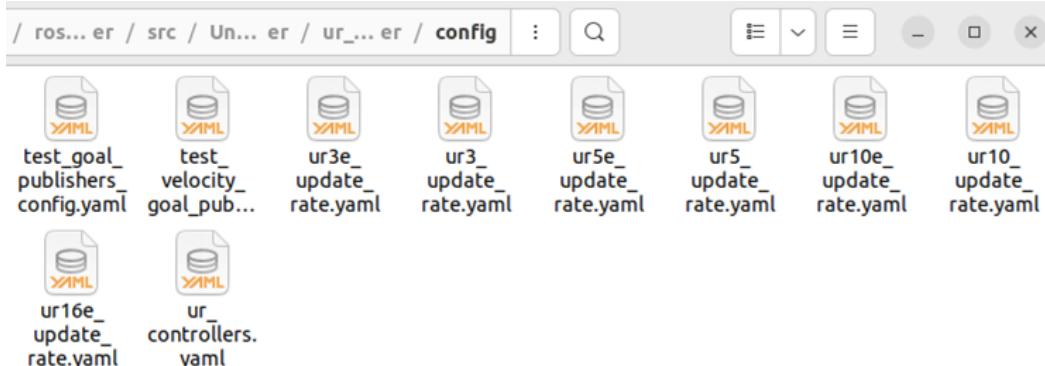


Figure 28. Config folder of the *ur_robot_driver* package, with the YAML files to configure the driver and the update rates. Source: Created by the author.

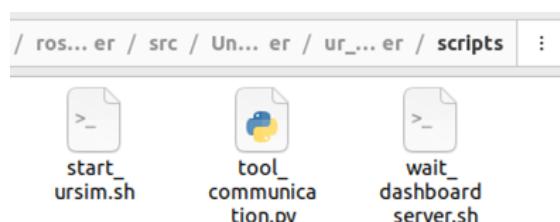


Figure 29. Scripts folder of the *ur_robot_driver* package, *start_ursim.sh* will initiate URSim docker simulation and *tool_communication.py* is only available for e-series. Source: Created by the author.

7.2. Universal_Robots_ROS2_Description

This package provides descriptions for the UR robot model, this means it defines not only its visuals, but also its physical properties and relation between joints. The description aims to behave as accurate as possible to the real robot physics.

In the case of the UR description package, there are three main folders that build the description of any cobot from the Universal Robots series. These folders are urdf, config and meshes.

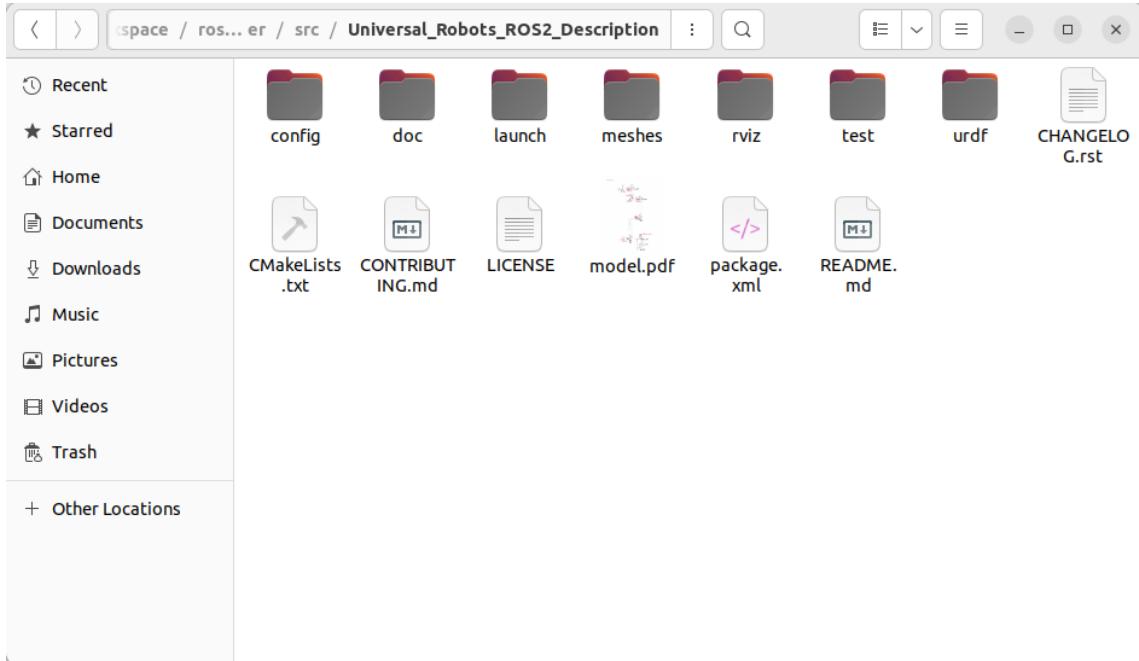


Figure 30. Folders from the *UniversalRobots_ROS2_Description* package. Source: Created by the author.

The way the package works is with only one description for all manipulators. This description is located inside the urdf folder, and depending on the model you want to describe, you can use the configuration files (from the config folder) and the collision and visual files (from the meshes folder) of the desired robot.

The ur_macro.xacro file contains the description of the UR manipulator. The description will depend on arguments that define the characteristics of the desired cobot. It also includes the ros2 control macro defined in ur.ros2_control.xacro.

This file is then included in the ur.urdf.xacro file, which implements the arguments. It makes the description use the correct configuration and mesh files to match the desired UR model, in this case the UR3.

The last file, ur.ros2_control.xacro, defines the robot joints and interfaces to use in the control framework created by ROS2 when running the driver. This file is crucial for the control to make the robot move, as it defines what type of commands can each joint send and receive (command and state interfaces) and establish limitations for safety or application purposes. As mentioned before, this file is included in the ur_macro.xacro file, as every single manipulator from UR use the same control configuration.

In fewer words, ur.ros2_control.xacro and ur_macro.xacro are general files that describe the control configuration and the hardware components configuration (respectively) for any UR

robot. While `ur.urdf.xacro` is the file where specific arguments receive certain values so the description of the final URDF generated for the robot matches the model the developer wants to work with.

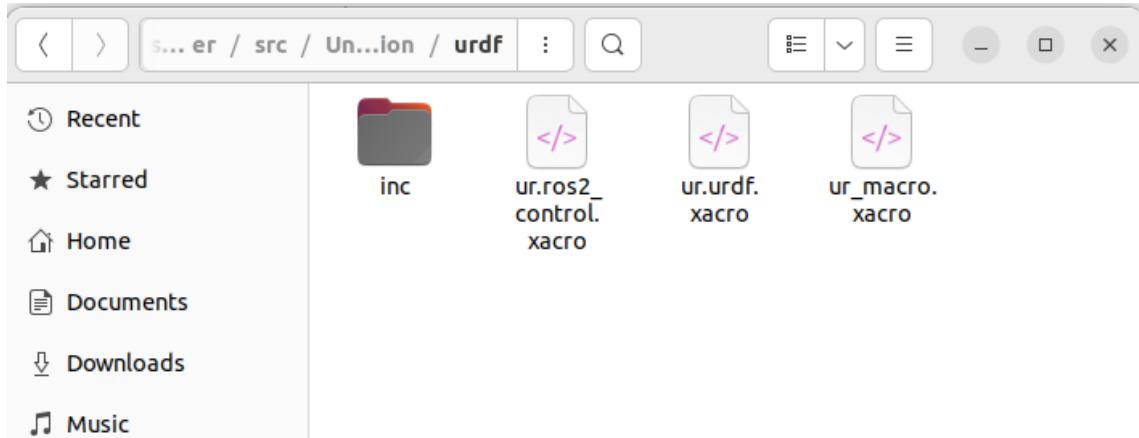


Figure 31. Main files from the `urdf` folder of the UR description package. Source: Created by the author.

As mentioned before there are two types of mesh files, the visual files, which consist of DAE files, and collision files, consisting of STL files. In the case of the configuration files, there are four types: default_kinematics, joint_limits, physical_parameters and visual_parameters. All of them defined in YAML files.

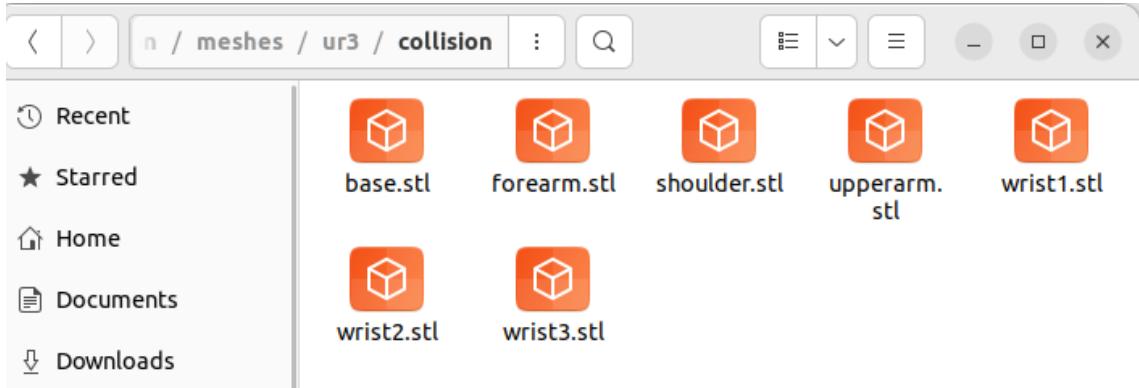


Figure 32. STL files from the `meshes/ur3/collision` folder of the UR description package. Source: Created by the author.

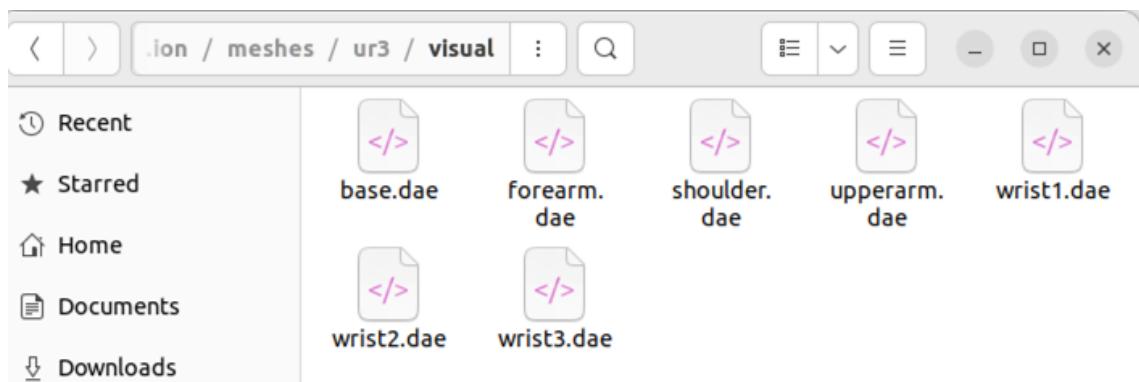


Figure 33. DAE files from the `meshes/ur3/visual` folder of the UR description package. Source: Created by the author.

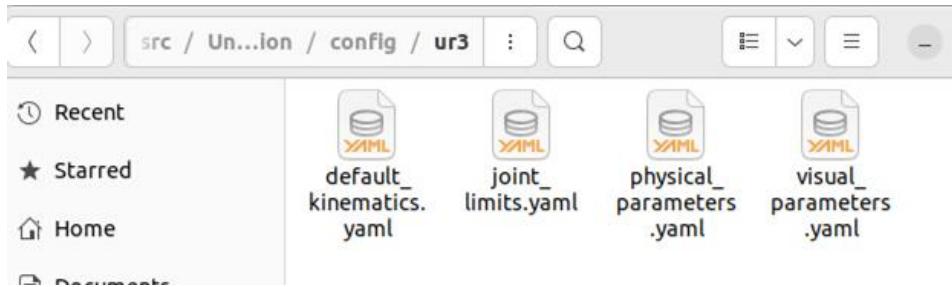


Figure 34. YAML files from the config/ur3 folder of the UR description package. Source: Created by the author.

All the description files for the UR3 were tested during simulation and real hardware control. No changes should be needed in these files, but in case any modification is needed they have enough comments to understand their sections. The one file which might be interesting to adjust depending on the application designed, is the initial position configuration file.

For this project, the config/initial_positions.yaml file was modified to set a desired initial position, as the UR3 robot from the lab where it was tested had some motion range limitations.

Note: initial_positions.yaml code, together with other important files, will be added and commented in Appendix 1 Commented Code.

7.3. Universal_Robots_Client_Library

This package consists of a C++ library to access Universal Robots interfaces. It is handed by Universal Robots to developers to provide a base to build and implement drivers for their cobots. The ROS2 UR Driver used during this project is built on top of this library.

Taking into consideration that the aim of the project is not building a driver, this document will not focus on details about this package as it is not directly used. However, in order to control any robot from the industry using ROS, the developer needs the interfaces of that precise robot to be able to communicate with it (send, receive and interpret its messages). This package exemplifies that for the UR manipulators.

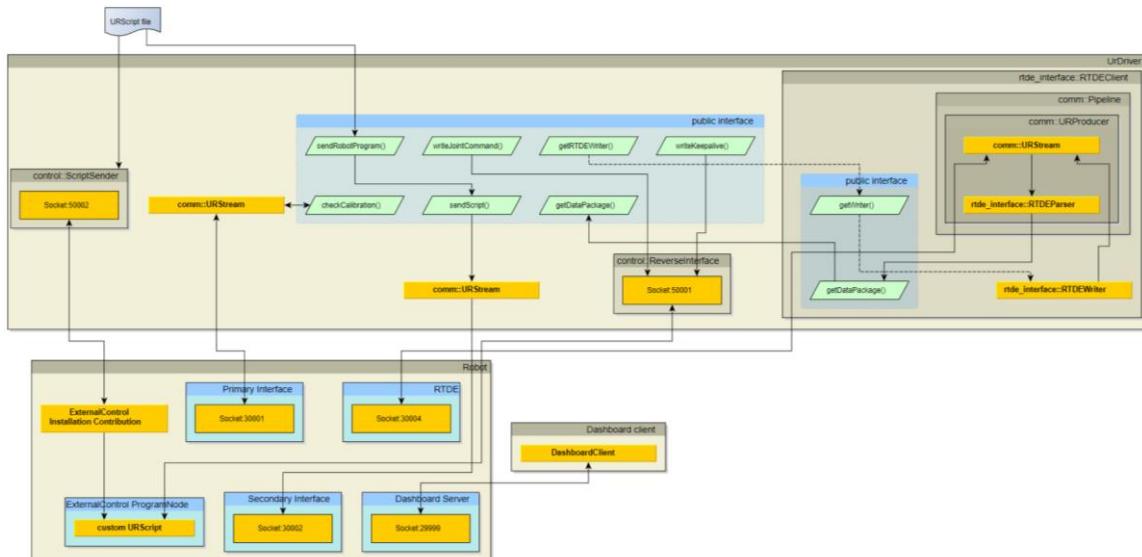


Figure 35. Architecture overview of the robot interface created by URDriver class of the package. Source: GitHub Universal_Robots_Client_Library [29].

As mentioned in the GitHub repository documentation of the Universal_Robots_Client_Library, the core of the library is the URDriver class which creates a fully functioning robot interface. For more information about the modules of the class check the repository documentation [\[29\]](#).

8. STARTING UNIVERSAL ROBOTS SIMULATIONS

During any robot application testing, simulation tools give the developer a useful and safe way of debugging and error-solving. Trying the code directly in real hardware can lead to major profit loss and could be dangerous for the environment if anything goes wrong.

Nowadays, simulation tools could be extremely accurate, not only in representing the robot and its physics but also its environment. Without having to worry about consequences, simulations apps can be used even to test unfinished code, making it easier for developers to identify the errors or missing parts.

During this project two simulation tools were used to test the controls provided by ROS2 and the UR Driver package. One of these simulations, provided by Universal Robots company (URSim), and the second one, the main open-source simulation software, Gazebo. Both tools were defined with their advantages and disadvantages in sections 3.2 Gazebo and 3.3 Universal Robots Simulator. This part will focus on setting them up in a system with the characteristics and software indicated in previous sections, so they can be used to start a control simulation of the UR3 robot.

8.1. URSim Image Docker

As the simulation executable is in the UR Driver package, you need to first source the workspace.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash
```

Then run the executable in the same shell. Remember to select the correct type of robot to simulate, in this case ur3. The -m represents the model parameter selection.

In Shell #1

```
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

The first time you run this, your system will acknowledge you that the ursim_net network is not created or that it does not exist, so it will try to create it by pulling it from universalrobots/ur3_cb3.

Your system might then give you a “permission denied” error. To solve it, you will have to follow the next steps.

1. Create docker group (if you haven't yet).

In Shell #1

```
sudo groupadd Docker
```

- Add your user to docker group.

In Shell #1

```
sudo usermod -aG docker ${USER}
```

- Log out and back in from the group so the changes apply.

In Shell #1

```
su -s ${USER}
```

- Change docker directory permissions so the “permission denied” error does not appear.

In Shell #1

```
sudo chmod 666 /var/run/docker.sock
```

Once this is done, rerun the ursim command.

In Shell #1

```
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

The output in the terminal should look like **Figure ???**.

The screenshot shows a terminal window titled "dan@dan-HP-Z240-Tower-Workstation: ~/workspace/ros_ur_driver". The terminal output is as follows:

```
See 'docker run --help'.
dan@dan-HP-Z240-Tower-Workstation:~/workspace/ros_ur_driver$ sudo chmod 666 /var/run/docker.sock
dan@dan-HP-Z240-Tower-Workstation:~/workspace/ros_ur_driver$ ros2 run ur_robot_driver start_ursim.sh -m ur3
Error: No such network: ursim_net
Creating ursim_net
4df1dfcf96a928db4088aaff139ba70a120a7dd8ef51b4979ee1e617b648ff37
Unable to find image 'universalrobots/ursim_cb3:latest' locally
latest: Pulling from universalrobots/ursim_cb3
fb668870d8a7: Pull complete
53ce29d13bc8: Pull complete
70ac01804590: Pull complete
24c55be12904: Pull complete
48e0d3c7bcfd: Pull complete
2380ef68af72: Pull complete
6c095f7cdebe: Pull complete
7c60bdac58cc: Pull complete
071131581830: Pull complete
Digest: sha256:abe012b8b2437b63841ce57affabacd7d640dedf557867f6df4c19844e4726eb
Status: Downloaded newer image for universalrobots/ursim_cb3:latest
142442450e60c0abb8b60b3265e341e7c43eb3050d81b9faa74593334f7458ab
Docker URSim is running

To access Polyscope, open the following URL in a web browser.
http://192.168.56.101:6080/vnc.html

To exit, press CTRL+C
```

Figure 36. Creating ursim_net and printing the link to access the URSim. Source: Created by the author.

Once the ursim_net is created, the link to the URSim website will be displayed. The next time the simulation is started, only the link will be printed in the shell.

Note: if in future runs the “permission denied” appears again, you will just need to change the permissions of the docker directory as you did in the 4th step, and then run the URSim.

In Shell #1

```
sudo chmod 666 /var/run/docker.sock  
ros2 run ur_robot_driver start_ur3.sh -m ur3
```

To access the Docker URSim, you will have to Ctrl+LeftClick the link printed in the terminal or copy and paste the link in your browser. That will open a browser tab where the robot IP and the assigned port will appear in the top right corner. Introduce your system password to connect.

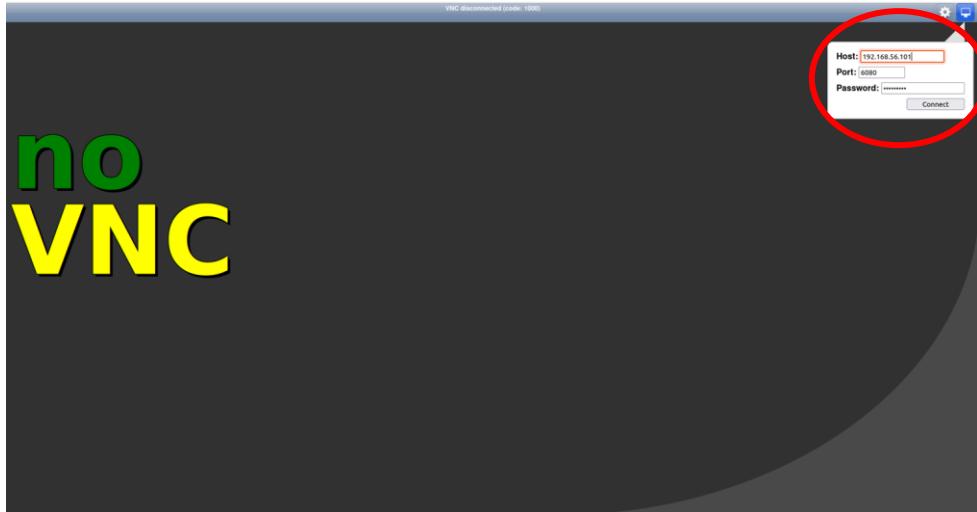


Figure 37. Browser tab to connect with the Docker URSim, field to insert your password located in the top right corner (red circled). Source: Created by the author.

Once you introduce your password, a charging screen will appear. Once it is done loading, the docker will ask you to initialize the robot. To do so press the “ON” button as in **Figure 38** and the initialization screen will change to the standby “Idle” mode as in **Figure 39**. Then press “ON” again and the simulated robot will be ready to use, **Figure 40**.

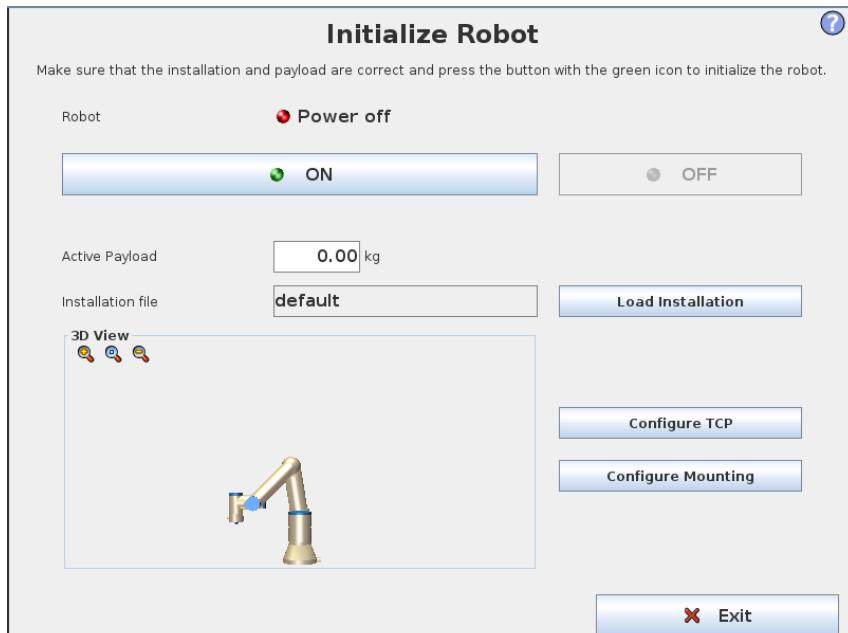


Figure 38. “Power off” robot mode. Source: Created by the author.

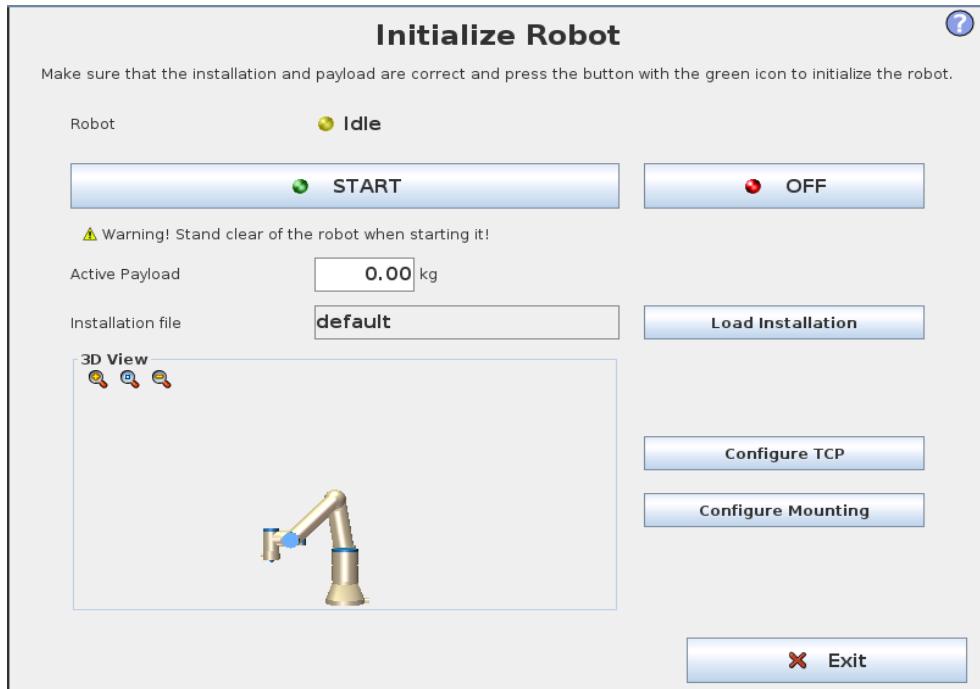


Figure 39. “Idle” robot mode. Source: Created by the author.

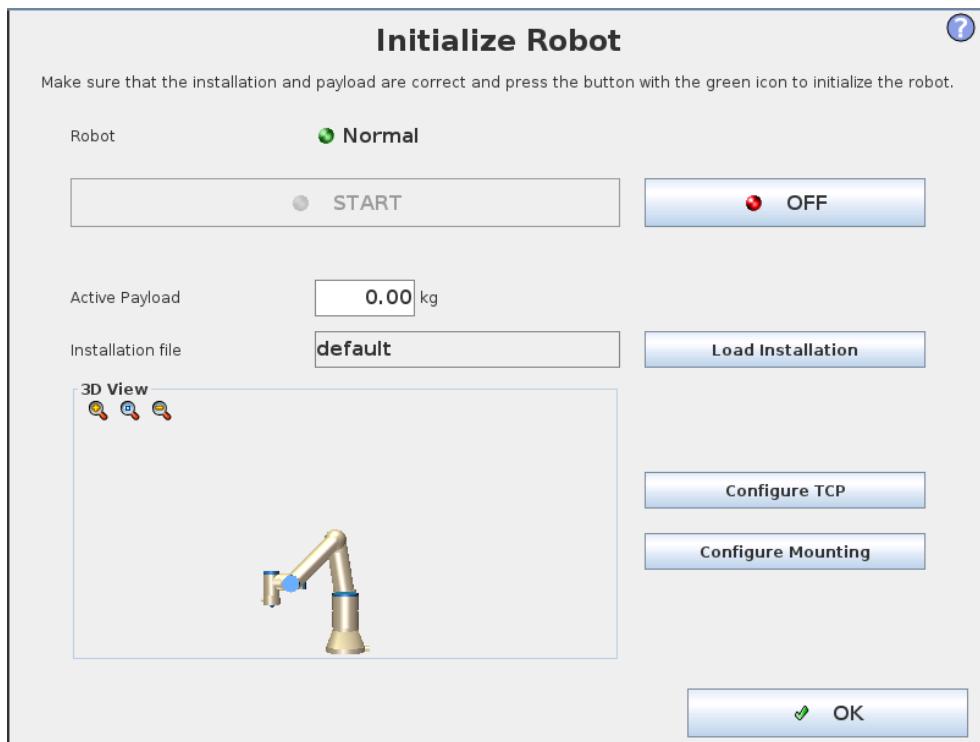


Figure 40. “Normal” robot mode. Source: created by the author.

Now that the robot is activated in its “Normal” mode, the main PolyScope screen will show in the tab. That PolyScope will simulate a real UR PlyScope panel.

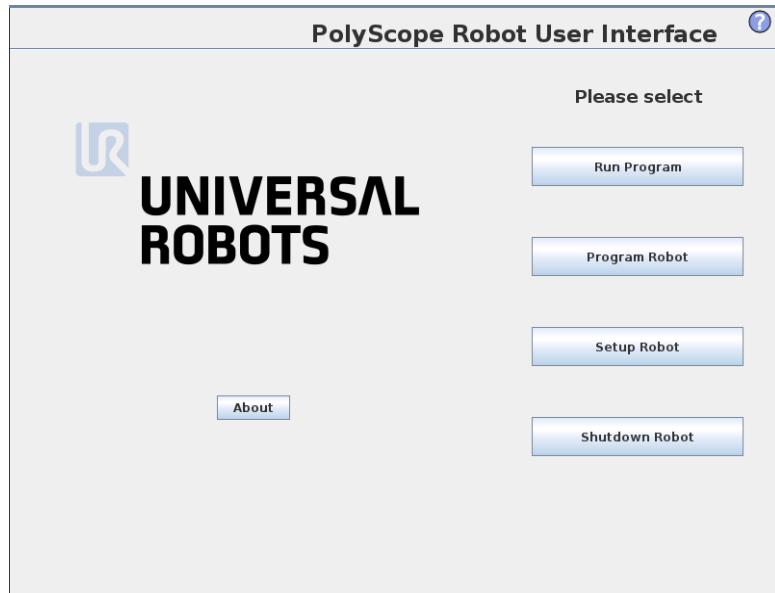


Figure 41. Main PolyScope URSim screen. Source: Created by the author.

The first thing you would need to do if this was a real UR manipulator PolyScope would be to set it up for the simulation, connecting the IPs and installing the URCap, which will be explained in section 10. Real UR3 Setup, but because this is a simulation, the URCap and the connection have already been set.

From this point two options can be considered, controlling the movement of the robot manually with the “FreeDrive” mode or with an extern program.

To use the FreeDrive, you will need to select the Run Program option from the main screen. That will lead you to the interface shown in **Figure 42**.

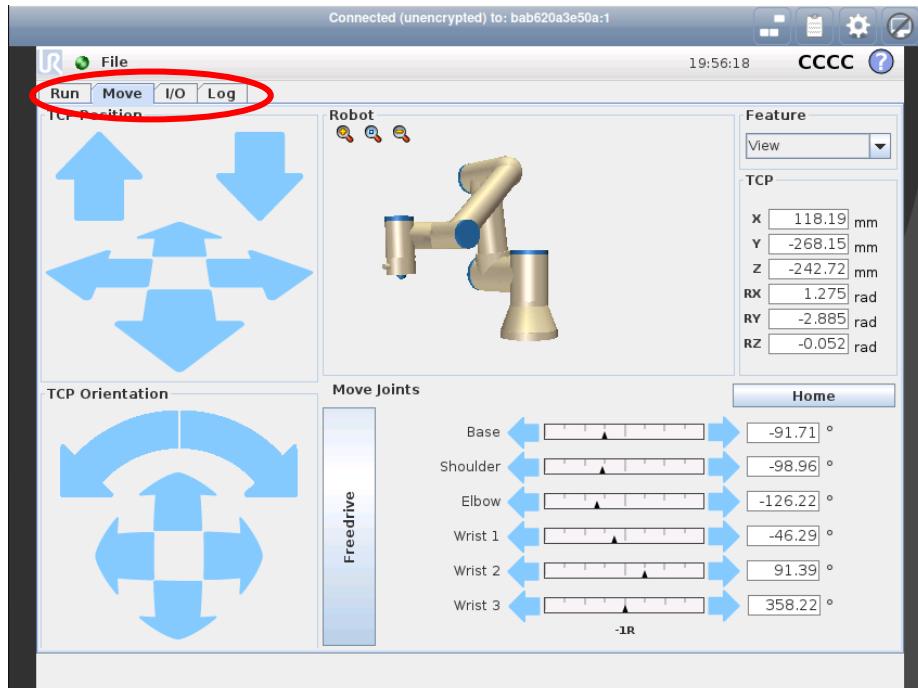


Figure 42. URSim FreeDrive interface. Source: Created by the author.

In the FreeDrive mode, you can use the same tools as in PolyScope to move the robot around, using the TCP or Joint movements to set goals for the robot.

But for this project, the section to use will be the external program, which will allow you to use ROS2 control framework to load controllers and files to move the robot. To do so you will have to select the Program Robot option in the main screen.

Then you will have to open a new shell and source the workspace and launch the ur_control file to create the control framework for the UR3.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.56.101
```

Once the controller is running, move to the URSim browser tab and follow the next steps:

1. Select Empty Program

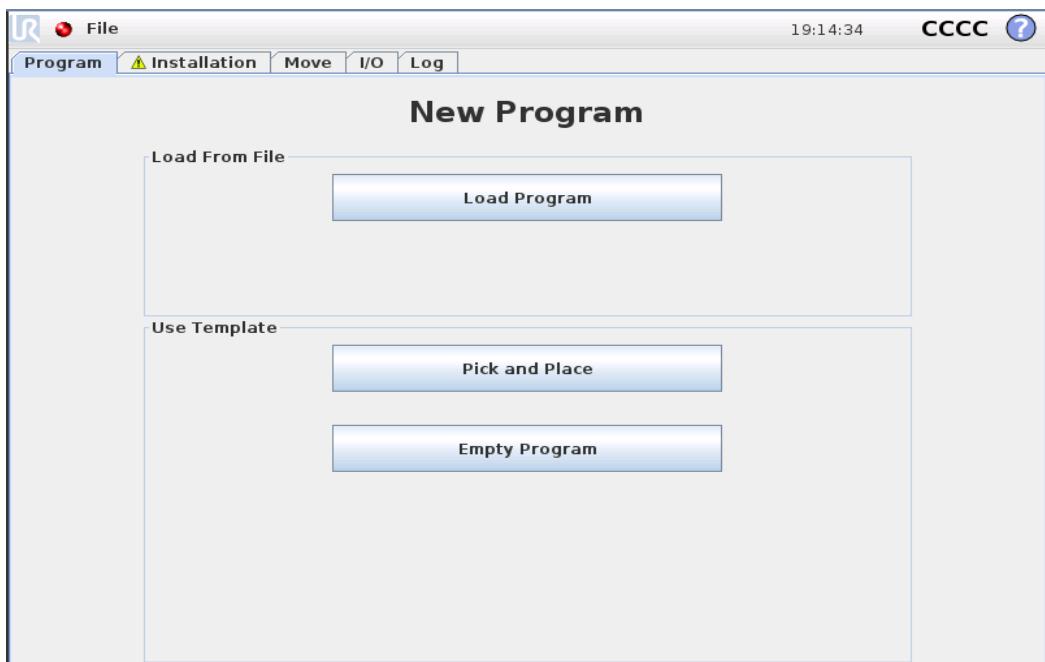


Figure 43. Program Robot screen. Source: Created by the author.

2. Inside Empty Program, go to the Program Tab (up left corner of the window) and select the Structure Tab.

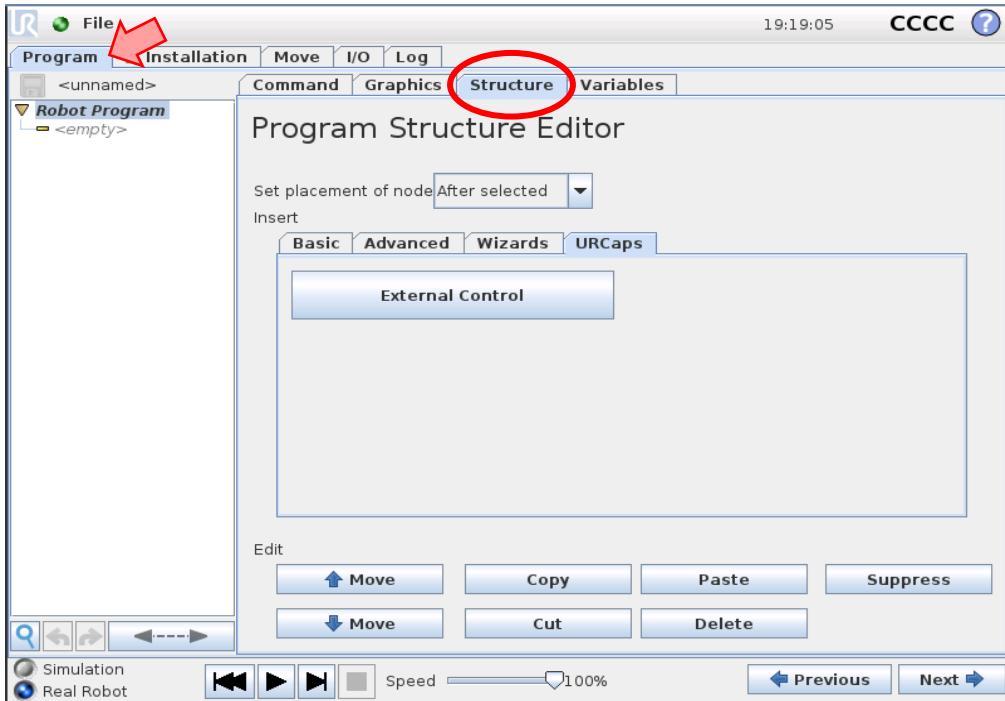


Figure 44. Empty Program screen. Source: Created by the author.

3. Inside the Structure Tab move to the URCaps Tab and select External Control.

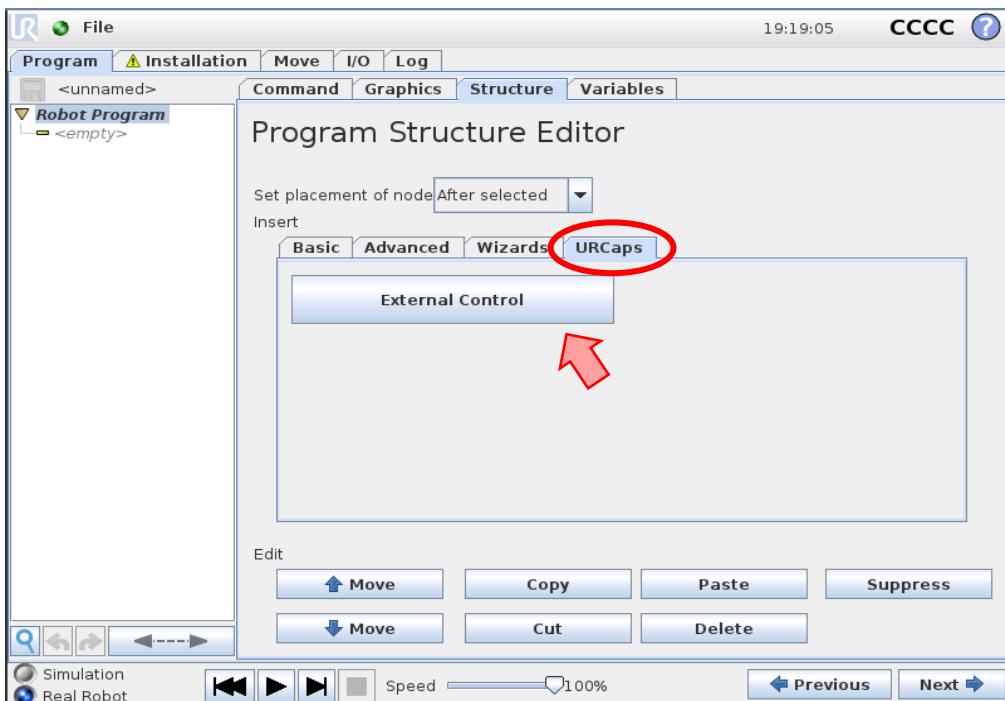


Figure 45. Structure Tab inside Empty Program screen. Source: Created by the author.

4. After that, the driver will appear in a list on the left side of the screen (with your robot IP). Select that control and press the “Play” black button at the bottom of the screen. That will connect the simulation to your control framework.

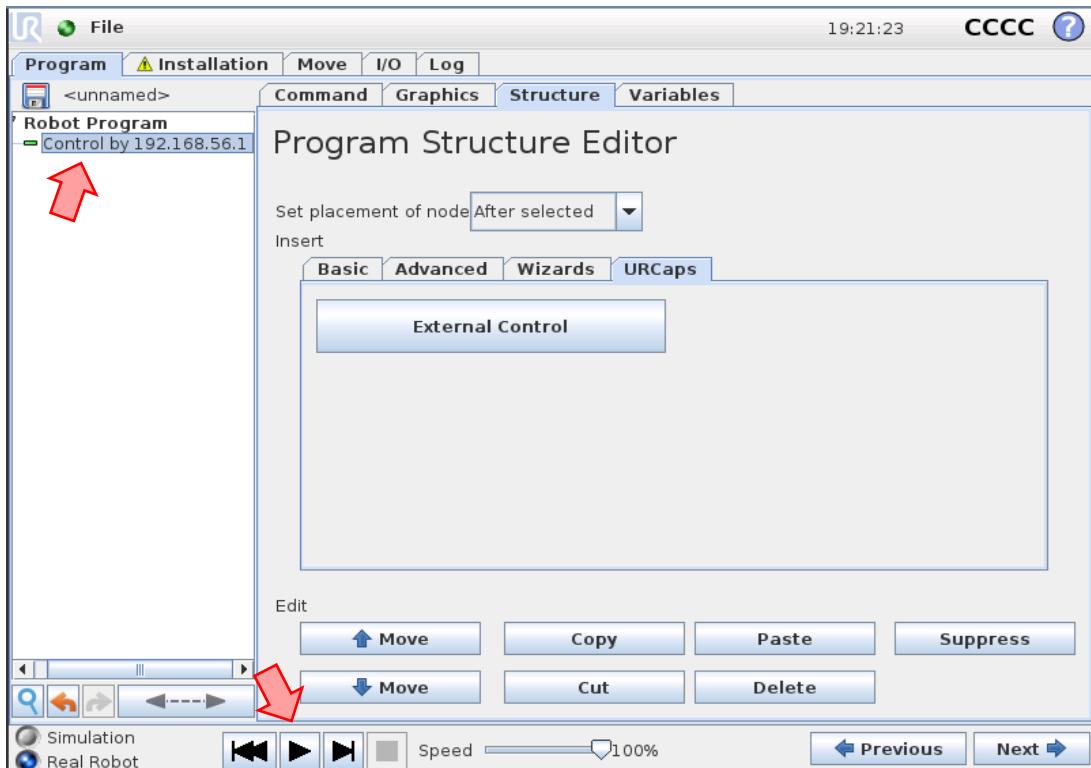


Figure 46. Load and start the simulation by pressing the Play button. Source: Created by the author.

5. To see the robot, go to the Graphics Tab.

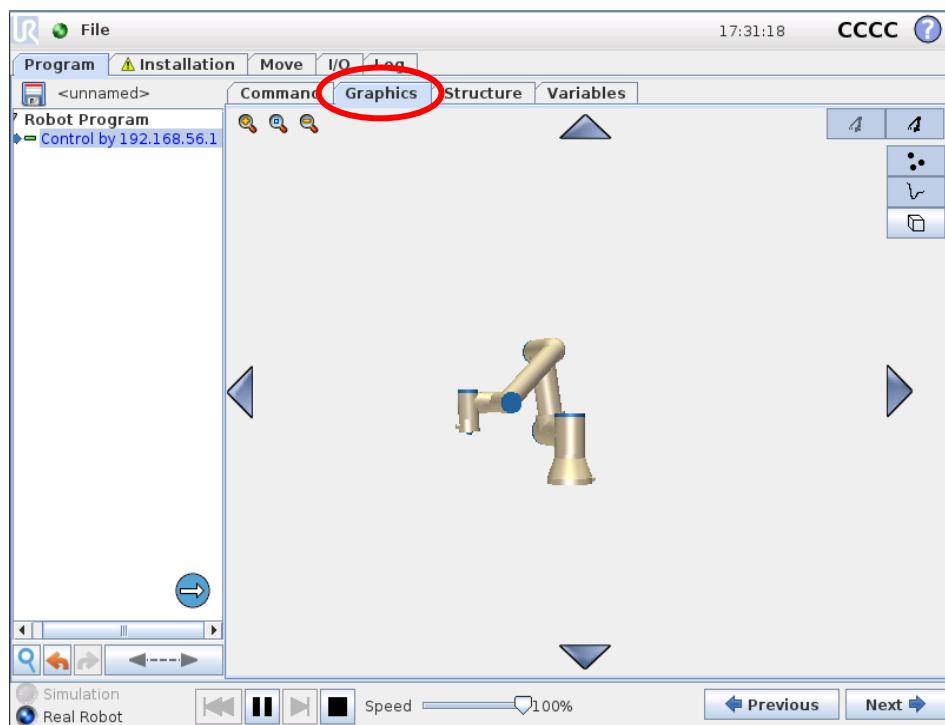


Figure 47. Graphics tab from the URSim Program section. Source: Created by the author.

Your simulation is now ready and connected to the control framework, so if you want to control the robot you could start loading controllers and sending messages commanding goals through the terminal or using scripts. This will be explained in section 11. Controlling UR3.

If you want to exit to the main screen, you can press “File” at the top left corner of the screen and select “Exit”. If you want to move to the “Initialize Robot” screen, press the green circle next to “File” at the top left corner of the screen.

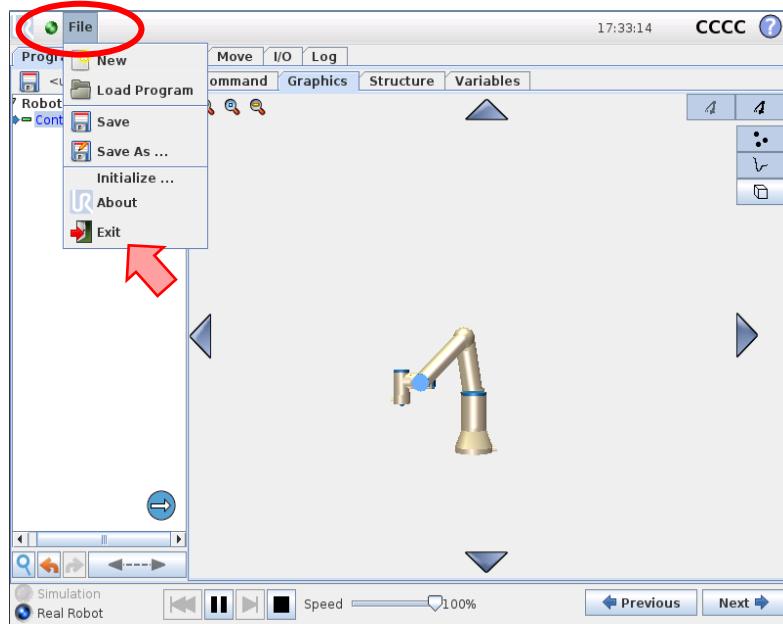


Figure 48. Exit to main screen and “Initialize Robot” screen. Source: Created by the author.

8.2. Gazebo

In the case of Gazebo, Universal_Robots_ROS2_Driver package does not provide a direct way to simulate the UR manipulators as is the case for the URSim. Nevertheless, the description package includes a sim_gazebo parameter to use the robot description package in a Gazebo simulation. Apart from that, there is an early-stage package called Universal_Robots_ROS2_Gazebo_Simulation that aims to start a simulation using Gazebo plugin libraries, the ur_description package and the joint_trajectory_controller.

The fact that the package is still being developed causes frequent errors and a limited use in applications for the UR3 robot in the Gazebo environment. Moreover, the complexity developing large applications in Gazebo using ROS2 Humble and the limited time to develop this project made it impossible to go any further on this topic. Taking that into consideration, a basic simulation with Gazebo was achieved. During this section step-by-step indications will be given about how to setup a UR3 Gazebo simulation using ROS2 Humble.

First step consists of adding the gazebo_ros2_control package to the system. This package integrates the ros2_control architecture with the Gazebo simulator. It provides a plugin that connects the controller manager with a Gazebo model. This package is not specific for UR manipulators, it needs to be implemented to your system to use ros2_control in any Gazebo simulation.

In Shell #1

```
sudo apt install ros-humble-gazebo-ros2-control
```

This line will add the package to the underlay workspace, where ROS2 Humble is installed.

Next step, include the Universal_Robots_ROS2_Gazebo_Simulation in the overlay workspace. You want to install it from source, as the description or the driver packages, as you will be doing some modifications to its files, and because that will allow you to study its files and understand them better.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
git clone https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation.git
src/Universal_Robots_ROS2_Gazebo_Simulation
```

That will add all the packages from the GitHub repository [\[19\]](#). Now you will have to build and source the workspace in order to use it.

In Shell #1

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

This package includes two directories:

- A config folder that contains a ur_controllers.yaml file, similar to the one from the ur_robot_driver/config folder, used to add the configuration of the controllers used with the UR Gazebo simulation.
- A launch folder that contains a ur_sim_control.launch.py file, similar to the ur_control.launch.py file from the ur_robot_driver/launch folder, that initiates all the nodes needed to set up the control architecture, as well as Gazebo and RViz. And a ur_sim_moveit.launch.py, which implements the previous launch file and adds to it the initialization of the MoveIt 2 tool to control the UR manipulator.

Note: ur_sim_control.launch.py code, together with other important files, will be added and commented in Appendix 1 Commented Code.

To start the simulation, run the following command in a shell after sourcing the workspace.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur3
```

This will start the control framework, spawn the UR3 in a Gazebo simulation, start the joint state broadcaster and the joint trajectory controller, and RViz visualization tool.

For the moment, the scaled_joint_trajectory_controller is not supported by the Gazebo simulation, that is why the joint_trajectory_controller is used.

Once the simulation starts, you will notice that the robot links start shaking and the joints break. This is the main problem of the package and what needs to be solved in order to execute a proper control of the cobot.

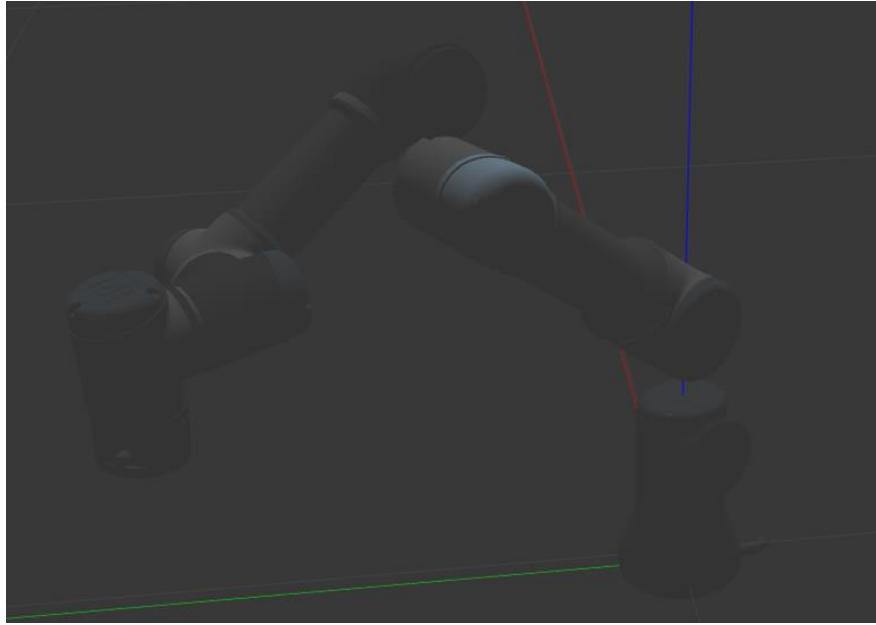


Figure 49. UR3 Gazebo simulation showing joints breaking. Source: Created by the author.

A discussion about the problem was found in GitHub Issues. There, a solution for the problem was stated by the user danzimmerman. The problem with Gazebo simulation seemed to be related to a dynamic conflict in the robot description. To stabilize UR manipulators' behavior in Gazebo simulations, fiction needs to be added to joint description dynamics' tag if sim_gazebo parameter is true.

This was deduced studying the Gazebo simulation in ROS1. Using ROS1, the breaking joints problem does not happen. That is because the parameter causing the issue, fmax parameter (or dParamFMax for Gazebo source code), is set explicitly by a function in gazebo_ros_control (same as gazebo_ros2_control but for ROS1) if the joints are using a position or velocity command interface in Open Dynamics Engine (the default rigid body simulator solver in ROS [43]). However, this does not happen in ROS2's gazebo_ros2_control package.

Running some tests, danzimmerman found that the friction parameter in the UR description package could also be used to set the dParamFMax. He suggests setting friction to 100 times the maximum joint effort, if the simulation is running with Gazebo. With this modification in the ur_macro.xacro file of the UR description package, the friction parameter would be set to zero for any simulation or real test, and 100x maximum joint effort for Gazebo simulation.

More information and details about this solution is stated in GitHub site Add conditional joint friction to stabilize Gazebo Classic simulation #55 [13]. There, some limitations to this solution were also stated, as in case of ROS2 the answer was modifying a general file from the UR description package, instead of changing the gazebo_ros2_control package as was done in ROS1. That could lead to several problems in the future and contradicts the working method followed by ROS community (solving a specific problem in a general file). That is why this is more of a temporary arrangement until Universal Robots develops a more sophisticated result.

Some other GitHub sites used to understand and develop the issue or similar topics are:

- Robot's joints break on Gazebo #19 [\[24\]](#).
- [humble] can not control robot with MoveIt #21 [\[10\]](#).
- Draft concept for injecting ros2_control hardware system #1 [\[18\]](#).
- Generically expose URDF joint dynamics. #56 [\[20\]](#).
- Unstable behaviour of Position Controller in UR10 #73 [\[31\]](#).
- Joints unable to maintain static position under PositionJointInterface #54 [\[21\]](#).

Note: ur_macro.xacro code, together with other important files, will be added and commented in Appendix 1 Commented Code.

The changes to implement in the ur_macro.xacro files can be found in [\[12\]](#).

Once the changes are added to the file, build and source the workspace, and run the launch file again.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
colcon build
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur3
```

Now the joints will not break, and the simulation will be ready to send control commands.

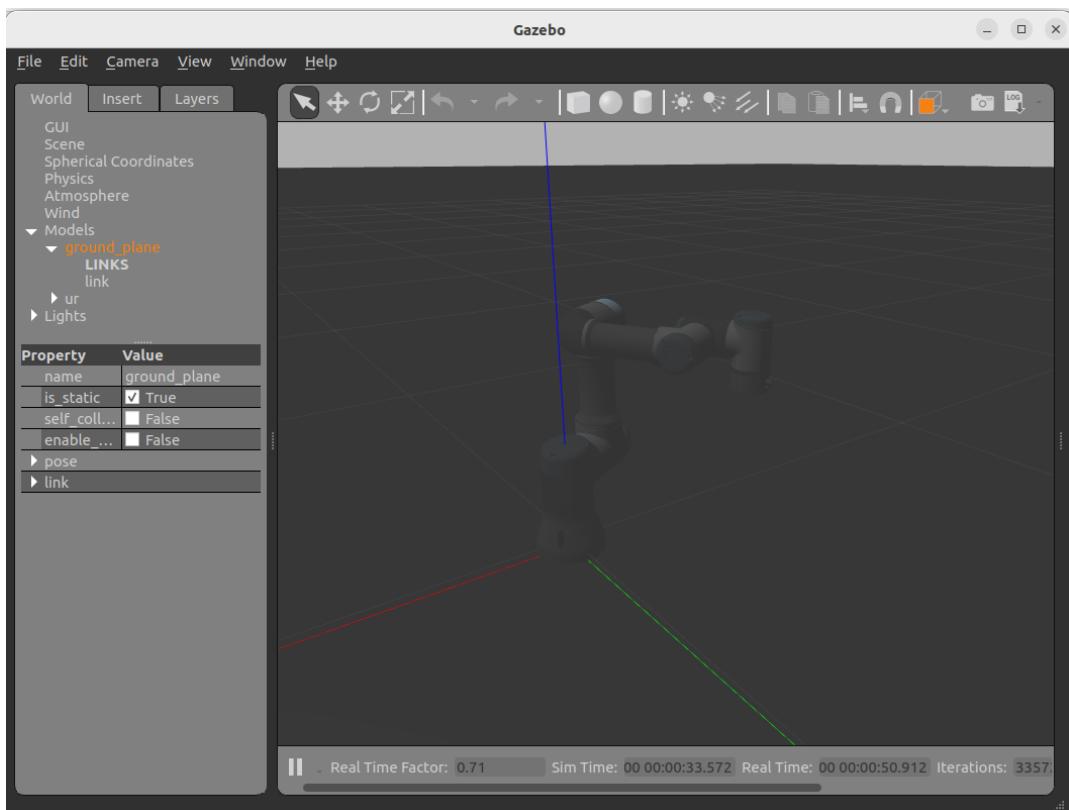


Figure 50. UR3 Gazebo simulation after solving the dynamic discrepancy. Source: Created by the author.

To start the simulation using the MoveIt 2, the file to be launched would be `ur_sim_moveit.launch.py`. But this file will be better explained in section 13. MoveIt 2, where the functionalities of the tool will be presented.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_moveit.launch.py ur_type:=ur3
```

9. VISUALIZATION TOOL, RVIZ 2

Once the simulation or the real robot are setup to start controlling the robot, there is one more feature to consider before moving the robot: RViz 2. This visualization tool, fully integrated in ROS2, provides a GUI aimed to observe the behavior of the robot in real time and easily detect bugs on the application.

In ROS2 there are several ways to monitor the states of a system, including sensors and actuators. Most of these consist of reading topic information, which can be displayed in a terminal echoing them. For example, if you run:

In Shell #1

```
ros2 topic echo /joint_states
```

In a started simulation, a message will display in the shell stating the position, velocity and effort of the robot joints.

However, these methods are not as intuitive as observing the actual robot and sensors of your system emulating the behavior of the real application. This is the main objective of RViz 2 and why is highly recommended to implement it in your ROS2 applications.

RViz 2 can be started in different ways. One of them is to run the following command in a shell:

In Shell #1

```
rviz2
```

This will execute the processes needed to start an empty RViz 2 environment, which can be adjusted to match your application by adding different components. This creates a new RViz layout, which can be saved and loaded after restarting RViz.

Other option is to initiate RViz using launch files, that way the developer does not need to load or add extra components from RViz interface. During this project launch files are used to start RViz, as was mentioned while explaining the files from `ur_robot_driver` package and from `ur_description` package.

RViz 2 uses information from the robot description and different topics to display graphically the environment of the robot. In fact, it can show data as laser scan or camera images (impossible to analyze in real life) that sensors from the environment capture, allowing the user to understand what the robot sees or detects while the application is running.

During this project, due to the focus on understanding the control framework of the UR3 and considering that the application was not oriented to perform any specific tasks, no external sensors were needed (only the joint sensors of the robot). Therefore, the RViz layout used is not too complicated and does not demonstrate the complete potential of the software. However, it serves as an example of some RViz display tools.

Once any of the simulations covered in the previous section are launched, a RViz tab opens with a layout defined in the launch files (if the launch_rviz parameter is not set to false). For this example, RViz from the Gazebo simulation will be used to explain the display section.

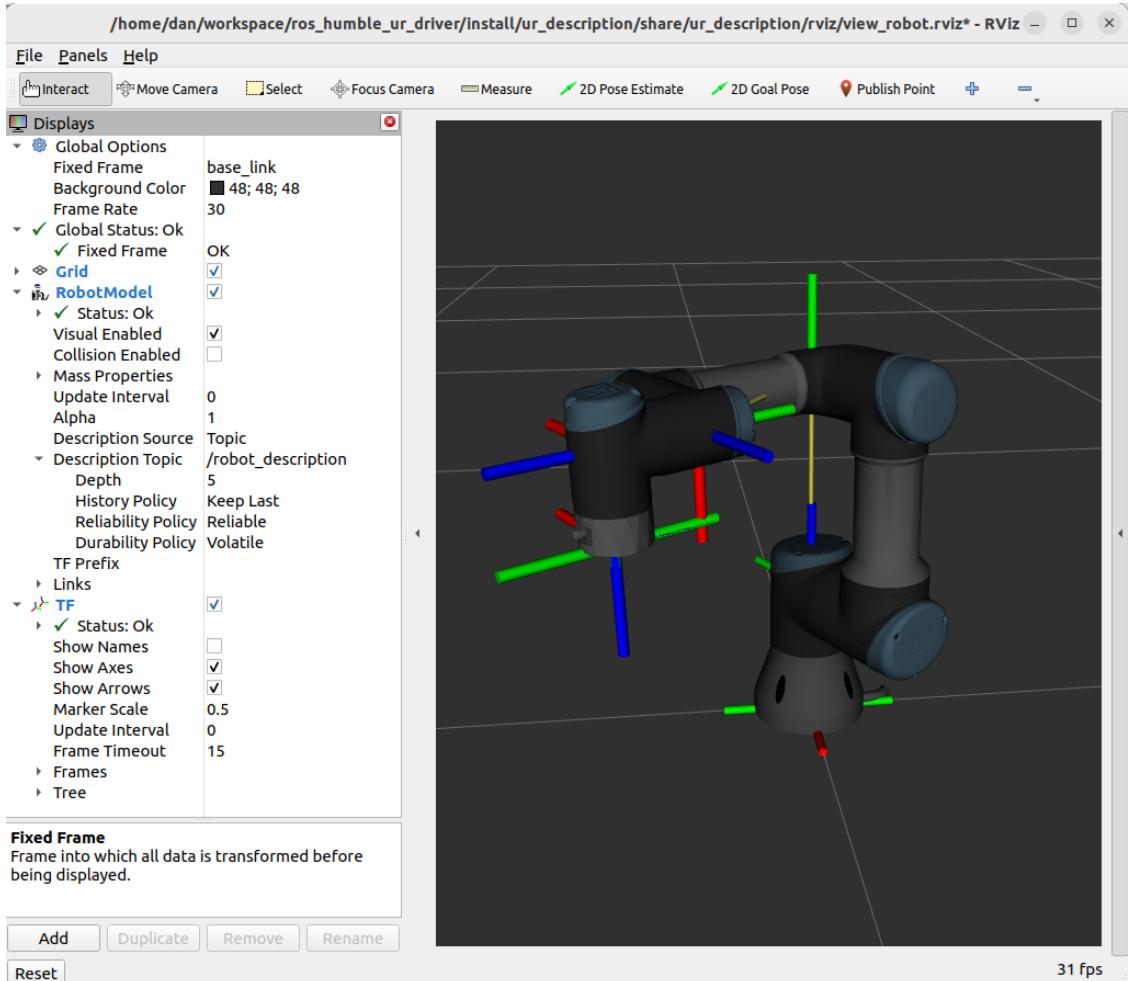


Figure 51. RViz layout when Gazebo simulation is launched, with added TF display. Source: Created by the author.

As can be appreciated five displays are set, together with the robot model spawned in the same position as the one in Gazebo.

The first three displays are common to every layout and represent general setting for the environment. Global Options allows you to set the coordinate frame that RViz uses as the fixed frame of reference. This is the frame that all other frames are transformed to and is typically the frame that represents the world or environment in which the robot operates. In this case, because only the UR3 is controlled, the Fixed Frame is set to the base_link, link that attaches the base of the robot to the world frame. It also provides a way to change the background color and the Frame Rate at which the visualization is updated and displayed on the screen. Global Status

acknowledges possible warnings or errors in the layout setup, and Grid can be used to modify the base grid of the environment.

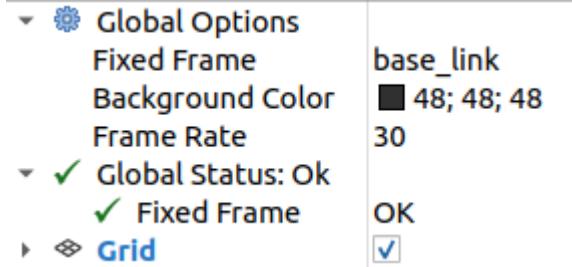


Figure 52. Global Options, Global Status and Grid displays in RViz 2. Source: Created by the author.

The Robot Model element uses the description files to display a 3D model of the robot. It can be configured to display different types of visualizations and contains information from the links, the joints and the end effector.

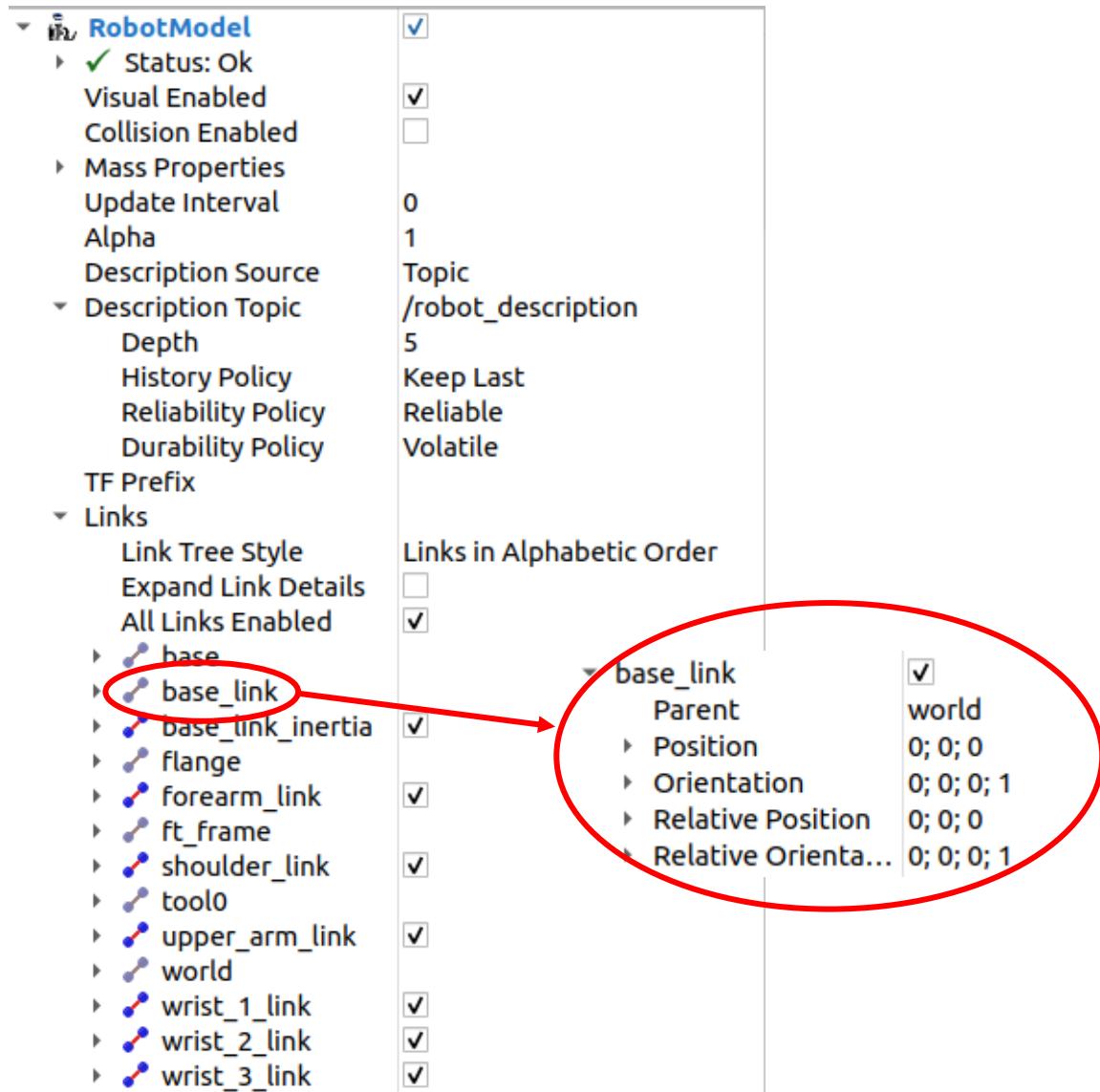


Figure 53. Robot Model element from RViz2 display. Source: Created by the author.

TF display is not started with RViz when Gazebo simulation is launched, but it is included in the document because it provides useful information and it is easy to set up. To do it you will have to press the Add button bellow the Display section and navigate the menu that will appear to find TF option. Then add it and it will automatically show the transform frames defined in the robot description, using the information from /tf and /tf_static topics.

This element allows you to modify TF visualization parameters and implements a Tree with the parent and child relation between frames.

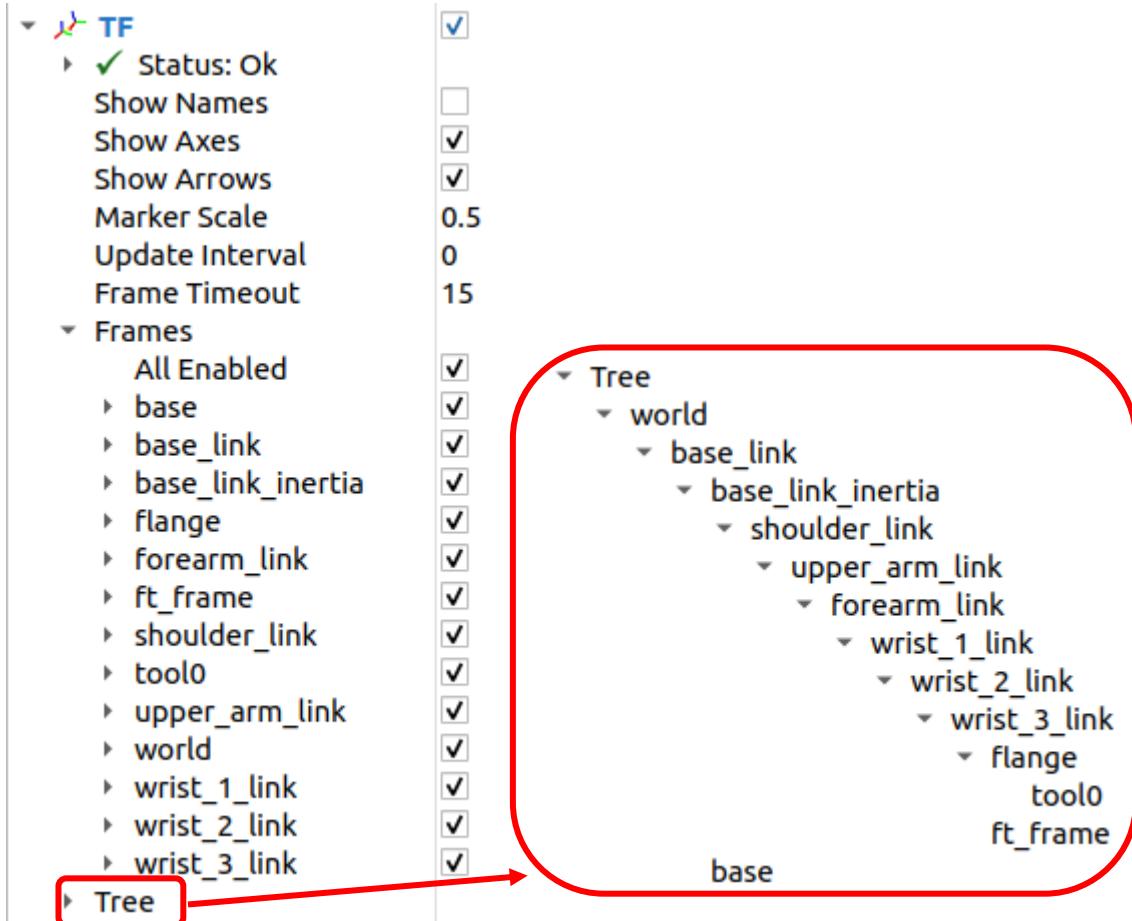


Figure 54. TF element from RViz2 display. Source: Created by the author.

One last display used during the project is the Motion Planning element. This display is added when MoveIt 2 is used to control the robot. To better understand how it works, it will be explained during section 13. MoveIt 2 of the document.

In order to add any other display, they can be selected from the Add menu, situated below the Display chart. A brief definition about the selected element is given at the bottom chart from the Add menu.

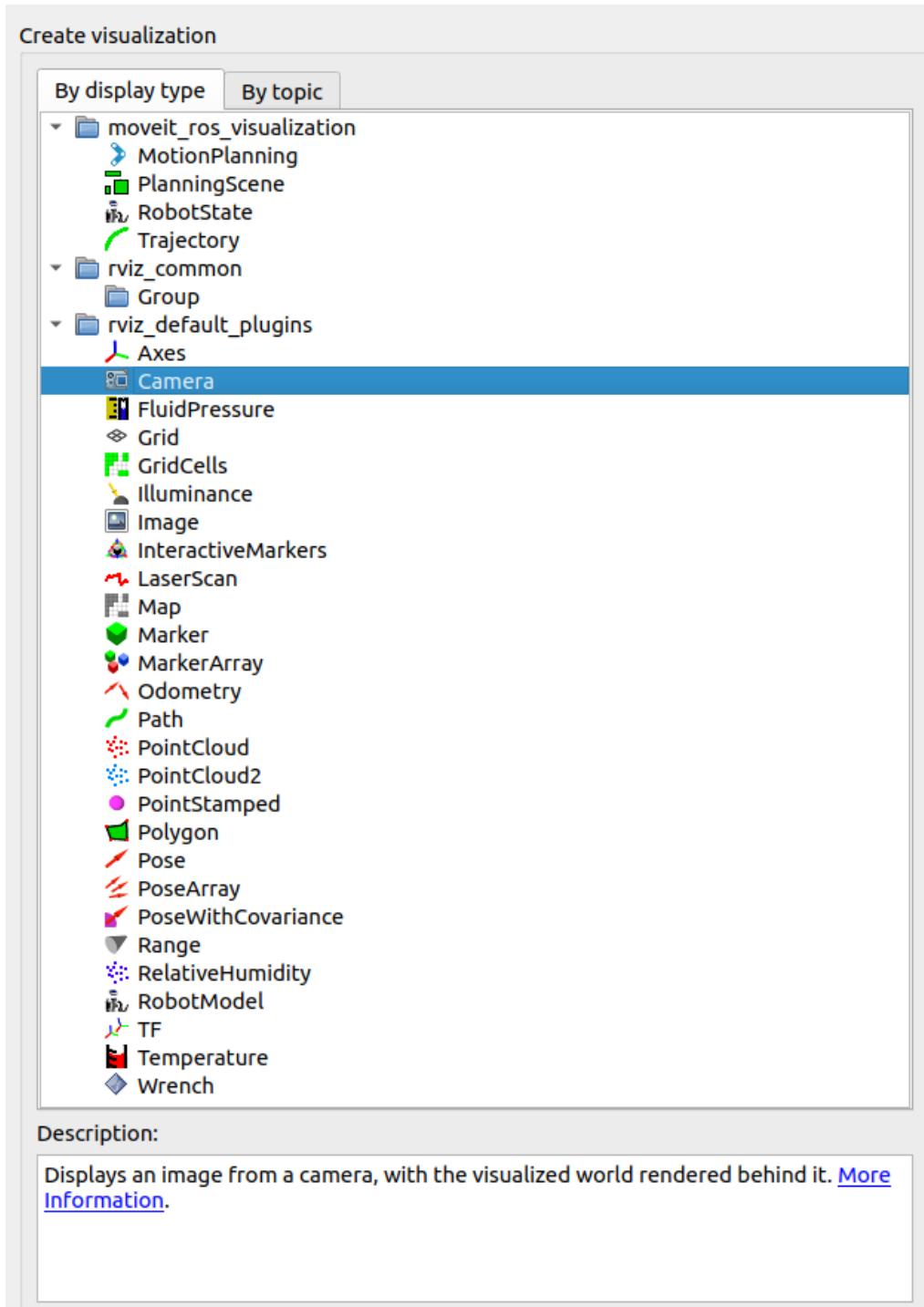


Figure 55. Display types in RViz and Description chart. Source: Created by the author.

Apart from the display elements, there is another menu that allows the user to select different views of the environment and offers a description of the current view. This menu is usually located at the left side of the window.

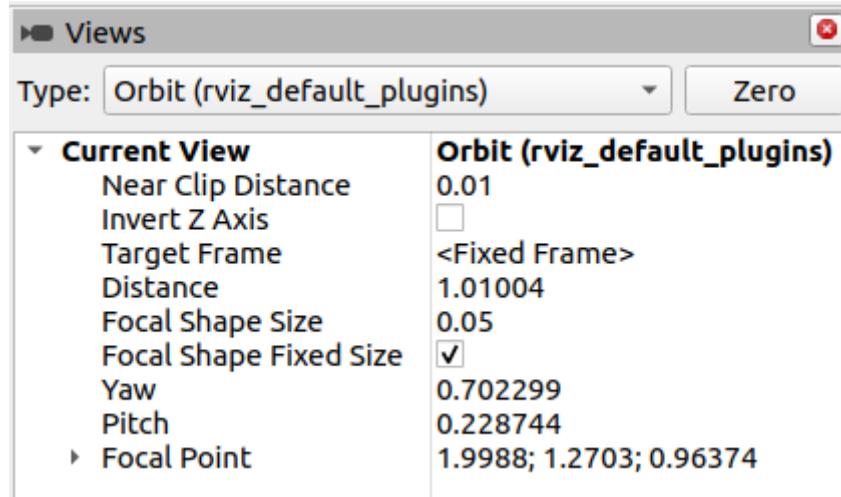


Figure 56. Views RViz menu. Source: Created by the author.

There are multiple websites and tutorials about RViz, explaining how to use its different tools and adapt it to any type of application.

10. UR3 ROBOT SETUP

The last component where control framework will be tested is a real UR3 robot. To be able to send commands to it from ROS2 and to connect it to the control framework, the external control software URCap must be installed in the robot. A deeper explanation about URCap software is given in the Universal Robots URCAP – BASICS website [\[73\]](#). The steps to install it will be explained below and are extracted from [\[57\]](#).

The software to be installed is `extrnalcontrol-1.0.5.urcap`, which can be found inside `~/workspace/ros_ws_humble_ur_driver/Universal_Robots_ROS2_Driver/ur_robod_driver/resources` folder from the overlay workspace in your system (if you installed UR ROS2 Driver as explained in section 7. UR ROS2 Driver) or downloaded from URCap External Control GitHub repository [\[34\]](#). A minimal PolyScope 5.1 version is required.

The first step to install it is to copy it to the robot's program folder using a USB. Then, follow the next steps:

1. On the welcome screen, click on the Setup Robot settings. Inside the Setup Robot select URCaps to enter the installation screen.
2. Click the plus sign at the bottom to open the file selector. All the URCap files stored inside the robot's program folder will show up. Open the `extrnalcontrol-1.0.5.urcap` file. Your URCaps view will now show the External Control in the URCap active list, as well as a notification to restart the robot.
3. Reboot the robot and check that the External Control appears in the Installation tab, inside Program Robot menu.

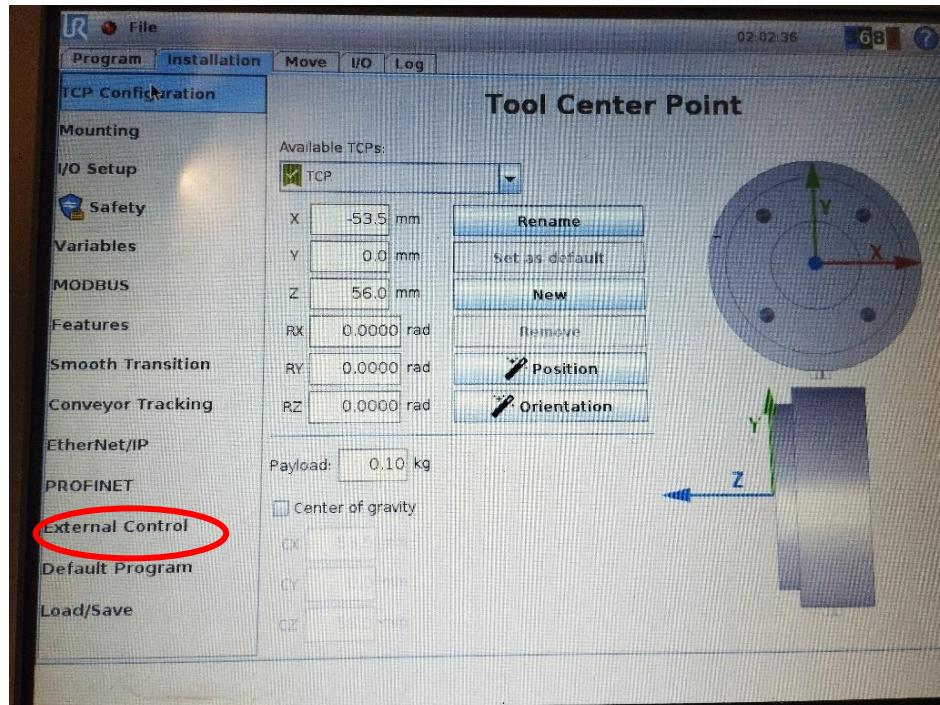


Figure 57. External Control in the Installation tab from Program Robot menu. Source: Created by the author.

4. Set up the IP address of the external PC that will be running the ROS2 driver. Both robot and external PC must be in the same network (if directly connected, network disturbances will minimize). The custom port should be left untouched for the moment.
5. To use URCap, create a new program and insert External Control program node into the program tree.
6. Click on the Command tab, the entered settings should appear inside Installation. Check that they are correct and save the program.

Once the URCap External Control is installed in the UR3, and the program has been created, the connection between robot and PC has to be established.

First, connect the robot and the PC using an Ethernet cable.

Second, set the IP address of the robot in PolyScope. To do it, access the Setup Robot menu and select Network. Then choose Static Address and fill the detailed settings.

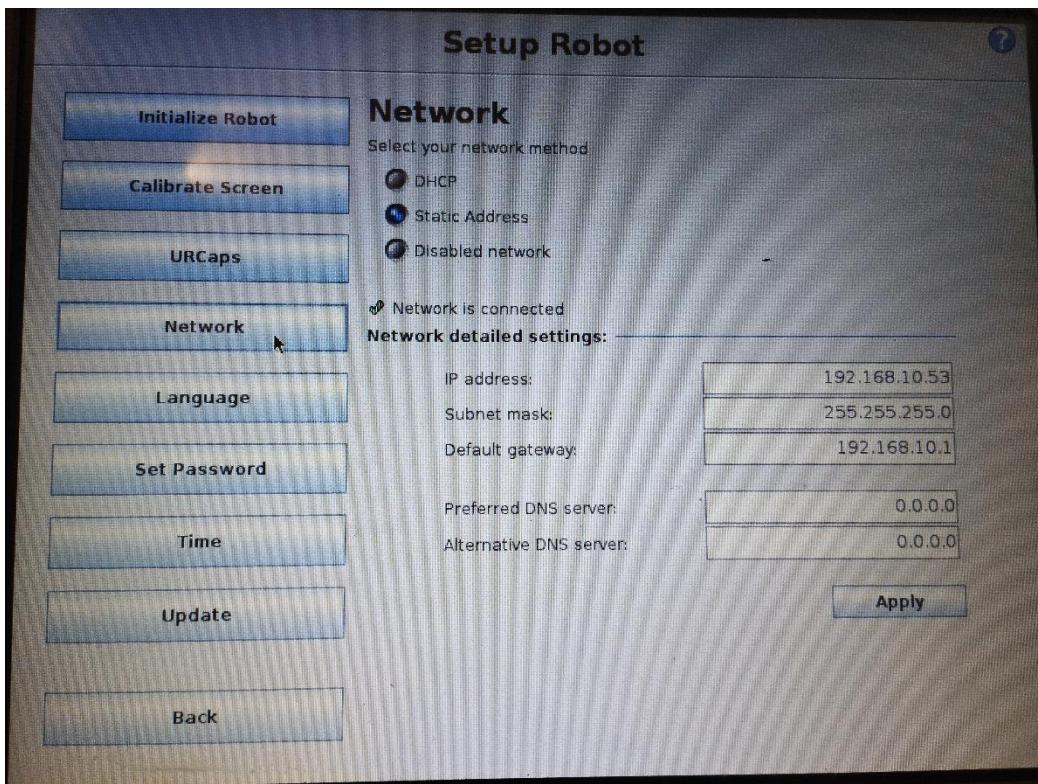


Figure 58. Set Network detailed setting of the UR3. Source: Created by the author.

Third, configure the IP address of the control PC.

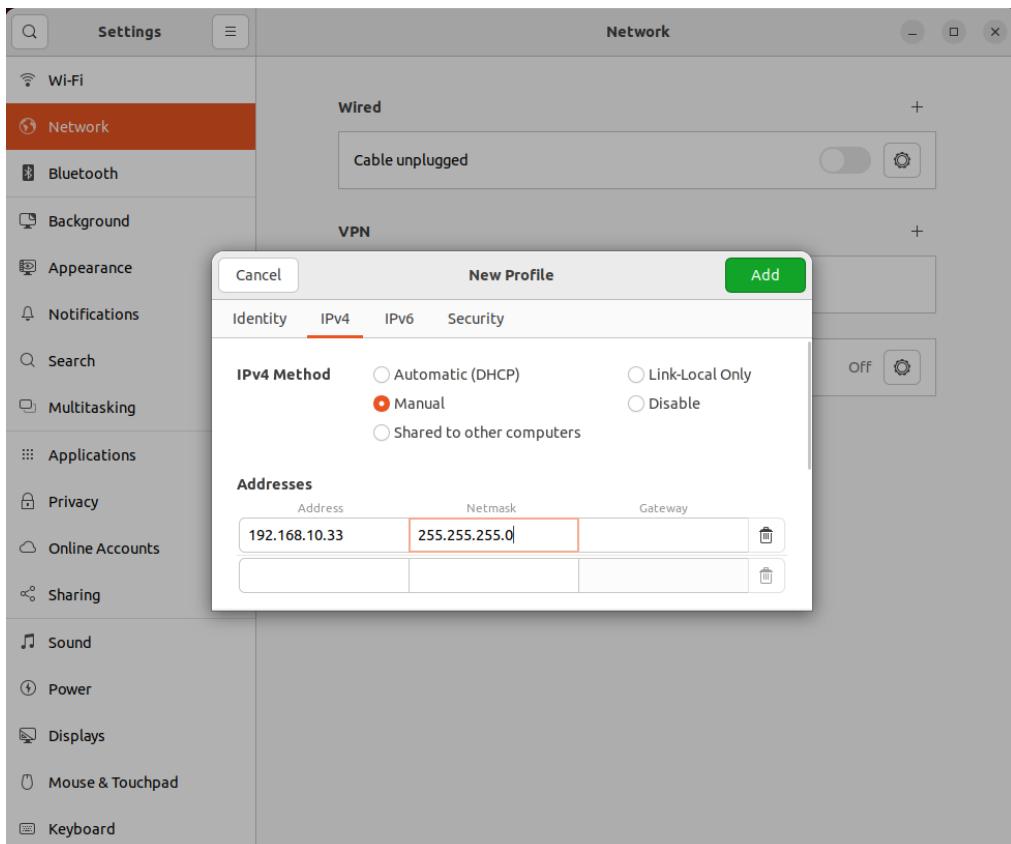


Figure 59. Setting the IP address of the control PC. Source: Created by the author.

Fourth, configure the Host IP and Host name information of URCaps External Control in Installation.

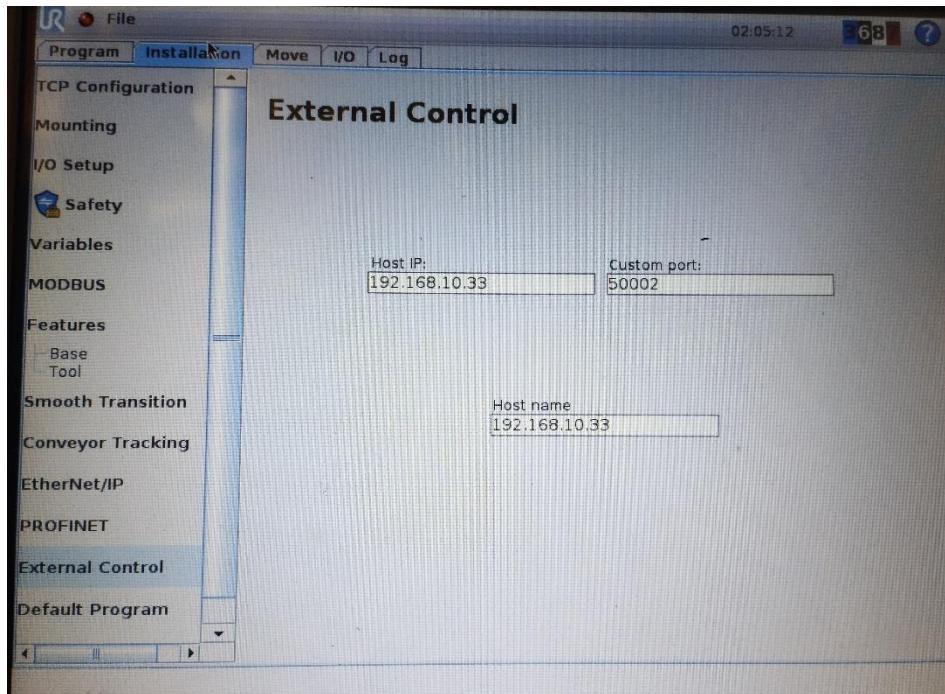


Figure 60. Configure Host IP with control PC address. Source: Created by the author.

The UR3 is now ready to receive ROS2 UR Driver commands from your control PC.

11. CONTROLLING UR3

This section will explain how to use different controllers with the UR3. It will be assumed that the system used to control the UR3, either simulated or real robot, is setup as detailed in previous sections of the document. Moreover, it will consider that the user knows how to initiate URSim or Gazebo simulations, configure the UR3 robot, and understands the topics and software previously explained.

The controllers to be tested will be:

- forward_position_controller – ros2_controllers package
- forward_velocity_controller – ros2_controllers package
- rqt_joint_trajectory_controller – ros2_controllers package
- joint_trajectory_controller – ros2_controllers package
- scaled_joint_trajectory_controller – ur_controllers package

In order to use these controllers, the user must first activate them, to be able to send commands to the robot thought them, as well as activate broadcasters, to be able to acknowledge the states of the robot and control it.

The procedure to activate controls and broadcasters is the same and will be described below.

First the controller has to be loaded into the control framework.

In Shell #1

```
ros2 control load_controller <controller_name>
```

To know what controllers are available to load, you can use:

In Shell #1

```
ros2 control list_controller_types
```

This command will list the available controller types and their base classes.

To know if the controller has been successfully loaded, you can use:

In Shell #1

```
ros2 control list_controllers
```

This command will output the list of loaded controllers, their type and status.

The status of a controller can be loaded, unloaded, configured, unconfigured, active, inactive. Once a controller is loaded it will appear in the list_controllers command. If the controller is unconfigured, it will appear as so. If the controller is configured, it will appear as active or inactive.

Once the controller is loaded, you will have to configure it. To do it, run:

In Shell #1

```
ros2 control set_controller_state <controller_name> configure
```

That will configure the controller and change its status to inactive.

To start a controller use:

In Shell #1

```
ros2 control set_controller_state <controller_name> active
```

When you have more than one controller loaded and configured, it is recommended having only one controller active at the same time to control a specific robot. To set the state of a controller to inactive on order to use another one, use:

In Shell #1

```
ros2 control set_controller_state <controller_name> inactive
```

If you stop needing one controller, you can unload it running:

In Shell #1

```
ros2 control unload_controller <controller_name>
```

This process can be tedious and the configuration status command might fail. There is an executable inside control_manager package that allows users to load and start controllers:

In Shell #1

```
ros2 run controller_manager spawner <controller_name> --controller-type
<controller_type/ControllerType>
```

Controllers and broadcasters can be also loaded and configured using launch files. Most of the controllers and broadcasters that will be used in this project will be initiated with the launch file that will start the control framework for the UR3 and tools like RViz. These launch files are ur_control.launch.py, for URSim and real UR3 control, and ur_sim_control.launch.py, for Gazebo control.

Section of the ur_control.launch.py code where the controllers and broadcasters are spawned:

```
# Spawn controllers
def controller_spawner(name, active=True):
    inactive_flags = ["--inactive"] if not active else []
    return Node(
        package="controller_manager",
        executable="spawner",
        arguments=[
            name,
            "--controller-manager",
            "/controller_manager",
            "--controller-manager-timeout",
            controller_spawner_timeout,
        ] + inactive_flags,
    )

controller_spawner_names = [
    "joint_state_broadcaster",
    "io_and_status_controller",
    "speed_scaling_state_broadcaster",
    "force_torque_sensor_broadcaster",
]
controller_spawner_inactive_names = ["forward_position_controller"]

controller_spawners = [controller_spawner(name) for name in controller_spawner_names]
+ [
    controller_spawner(name, active=False) for name in
controller_spawner_inactive_names
]

# There may be other controllers of the joints, but this is the initially-started one
initial_joint_controller_spawner_started = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[
        initial_joint_controller,
        "-c",
        "/controller_manager",
        "--controller-manager-timeout",
        controller_spawner_timeout,
    ],
    condition=IfCondition(activate_joint_controller),
)
```

Section of the ur_sim_control.launch.py code where the controllers and broadcasters are spawned:

```

joint_state_broadcaster_spawner = Node(
    package="controller_manager",
    executable="spawner",
    arguments=["joint_state_broadcaster", "--controller-manager",
    "/controller_manager"],
)
# (...)

# There may be other controllers of the joints, but this is the initially-started one
initial_joint_controller_spawner_started = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[initial_joint_controller, "-c", "/controller_manager"],
    condition=IfCondition(start_joint_controller),
)

```

Not all the controllers will be used in every simulation or the real robot due to incompatibilities, configuration issues and time management of the project.

11.1. forward_position_controller

Forward position controller will only be tested in URSim docker simulation, as it is one of the simpler controllers used in many ROS2 control tutorials as introduction to joint trajectory controller. This controller receives an array of floats that corresponds to the joint position values and forwards that same array as output.

To test this controller, execute in one shell the process of the URSim. In a second shell launch the ur_control.launch.py file. Remember to build the workspace and source every single shell.

The ur_control.launch.py will load and configure the necessary broadcaster and the controller. Check it by listing the load controllers in a third shell.

In Shell #3

```

cd ~/workspace/ros_humble_ur_driver
colcon build
source install/setup.bash
ros2 control list_controllers

```

In that same shell, activate the controller.

In Shell #3

```

ros2 control set_controller_state forward_position_controller active

```

Now that the controller is active, the simulation running and RViz configured it is time to test the controller.

The control commands can be sent publishing messages in the controller topic directly from the terminal or using scripts. As this is an introductory controller, only the terminal will be used. The message to send a position command with the forward_position_controller is the next one:

In Shell #3

```

ros2 topic pub /forward_position_controller/commands std_msgs/msg/Float64MultiArray "data:
- 2.55
- -1.7

```

```
- 1.45
- -1.57
- 0.0
- 0.0" -1
```

Note: To check the different topics and their information, to know in which you will have to publish for each controller, refer to Appendix 3. Linux and ROS2 basic commands.

The six floats represent the position values for the corresponding joints, starting from the shoulder_pan_joint to the wrist_3_joint.

Once the message is published, the robot should move to the forwarded position.

11.2. forward_velocity_controller

This controller is similar to the previous one, it cannot be used directly on real robots, due to security problems, and does not have a direct application. Forward_velocity_controller is used to change the speed of the joint motors of a robot. It will just be tested with the URSim.

In this case, the controller is not loaded. So, first thing will be to load and configure the controller. There is no need to stop the Shell #1 URSim process or the Shell #2 ur_control process.

```
ros2 run controller_manager spawner forward_velocity_controller --controller-type
forward_velocity_controller/ForwardVelocityController
```

Then start the controller:

```
ros2 control set_controller_state forward_velocity_controller active
```

Remember to stop the forward_position_controller if you have not.

```
ros2 control set_controller_state forward_position_controller inactive
```

You can always make sure every controller has the desired state using the list_controllers command.

Then publish the next message to send the velocity commands to the joints:

```
ros2 topic pub /forward_velocity_controller/commands std_msgs/msg/Float64MultiArray "data:
- 0.1
- 0.0
- 0.0
- 0.1
- 0.0
- 0.0" -1
```

For this controller, the values must be between 0 and 1, as they refer to the speed scaling of the motor. If the value is set to 0.1, it will mean that the motor goes to 0.1 times the motor maximum speed.

To make the joints stop publish the next message:

```
ros2 topic pub /forward_velocity_controller/commands std_msgs/msg/Float64MultiArray "data:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0" -1
```

If you do not set the joint velocity to 0, the simulation will automatically stop once the maximum position is reached by any of the motors due to safety configurations.

As you can see, this controller is not useful by itself for any specify robotic application, similar to the previous one, as they provide little control configuration or possibilities. However, they can be used together with other controllers or as a base to develop a more complex controller.

11.3.joint_trajectory_controller

A more complex controller, integrated with the ros2_controllers package, is the joint_trajectory controller. It is designed to execute trajectories on a group of joints given a set of waypoints and a specific time to execute the trajectory.

A spline interpolator is provided to calculate the trajectory depending on the waypoint specification. If only position is specified, a linear trajectory is interpolated. If position and velocity are, cubic interpolation is used. And if position, velocity and acceleration are provided, a quintic interpolation gives the trajectory.

This controller supports actuators with position, velocity and effort hardware interfaces. For the UR3 robot, with position-controlled joints, no PID adjustment will be needed as the desire position for the motors will be directly forwarded to the joints.

Only Gazebo simulation will be used to test the joint_trajectory_controller. This is due to simplicity, joint_trajectory_controller is already setup in the ur_simulation_gazebo package, and because scaled_joint_trajectory_controller will be used in URSim and the real UR3, a similar but improved version of the joint_trajectory_controller designed for the UR manipulators and their actuators' characteristics.

In order to send commands to the controller and set goals for the UR3, the action interface is used (control_msgs/action/FollowJointTrajectory). Action goals allows to specify the trajectory to execute, and path and goal tolerances. If no tolerances are specified, the default values are used. Tolerances could be useful in specific applications for precise robot movements. If the tolerances are not met, the action goal is aborted (when a goal is aborted, the controller will attempt to execute the trajectory as good as possible).

To send goals, users can use terminal commands or publisher_joint_trajectory_controller executable, setting the goal configuration in a YAML file. The advantage of using files instead of the terminal, is that the user can set several goals to execute in a row.

It is important to note that with the joint_trajectory_controller, if a goal is sent before the actual trajectory is completed, the controller will combine them instead of discarding the current one. This is called trajectory replacement.

More information about the joint_trajectory_controller can be found in [\[63\]](#).

To start the test with the Gazebo simulation, you will have to run a simulation a new shell.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash  
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur3
```

Then, open a second shell to send the goal to the robot.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 action send_goal /joint_trajectory_controller/follow_joint_trajectory  
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:  
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo  
int], points: [{positions: [1.09,-1.57,1.57,-1.57,-1.57,0.0], velocities: [],  
accelerations: [], time_from_start: {sec: 6, nanosec: 0}}]} }"
```

The values for the positions can be modified. The velocities and accelerations are left empty as the actuators of the UR3 can be commanded using just position. To make a velocity control using the joint_trajectory_controller some modifications should be implemented in the control files, but that is not covered in this project.

In order to send goals to the robot using scripts, the following file has to be launched:

In Shell #2

```
ros2 launch ur_robot_driver test_joint_trajectory_controller.launch.py ur_type:=ur3
```

This launch file starts an executable called publisher_joint_trajectory_controller.py from the ros2_controllers_test_nodes package, inside ros2_controllers package. This executable checks the goals set in the test_goal_publishers_config.yaml file and publishes them in the action server if they are correctly configured.

Remember that the robot must be in its initial position in order for the first goal to be accepted. If it is not, send the next goal before launching the file:

In Shell #2

```
ros2 action send_goal /joint_trajectory_controller/follow_joint_trajectory  
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:  
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo  
int], points: [{positions: [2.09,-1.57,1.57,-1.57,-1.57,0.0], velocities: [],  
accelerations: [], time_from_start: {sec: 6, nanosec: 0}}]} }"
```

Another option is to ignore the initial position condition by modifying the test_goal_publishers_config.yaml file. This file is the configuration code that sets the goals to be send when test_joint_trajectory_controller.launch.py is launched. To do so, go to line 50 of the code, and set the check_starting_point parameter to false.

```
50 check_starting_point: false
```

If you want to set different goals, change the values in lines 37, 38, 39 and 40 from this same file.

```
37 pos1: [0.785, -1.57, 0.785, 0.785, 0.785, 0.785]
38 pos2: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]
39 pos3: [0.0, -1.57, 0.0, 0.0, -0.785, 0.0]
40 pos4: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]
```

Remember that, if you modify any file, you will have to stop the simulation, build and source the workspace and restart the simulation, in order for the changes to apply.

Note: test_goal_publishers_config.yaml code, together with other important files, will be added and commented in Appendix 1. Commented Code.

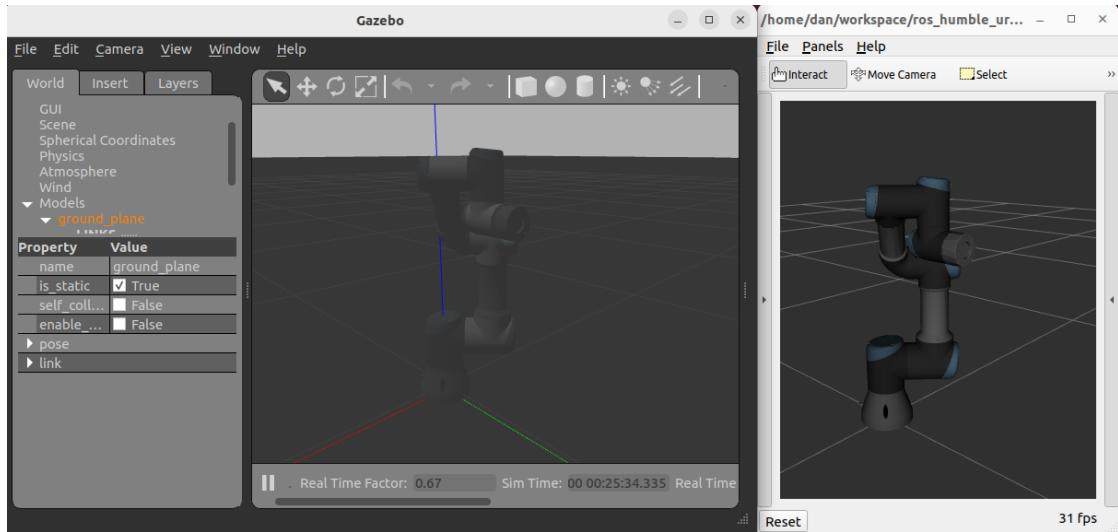


Figure 61. Gazebo simulation and RViz visualization of the UR3 after reaching a goal sent by the joint_trajectory_controller. Source: Created by the author.

11.4.scaled_joint_trajectory_controller

As explained in section 7. UR ROS2 Driver of the document, the scaled_joint_trajectory_controller (SJTC) is a configured joint_trajectory_controller (JTC) to match the characteristics of the UR motors. It uses the data provided by speed_scaling_state_controller, to control the speed scaling of the robot joints while performing the trajectory. This gives the controller several advantages over the JTC. The most significant one is the fact that the path followed by the cobot will not experience deviations due to configuration limitations or scaled down trajectories.

This controller is not supported by Gazebo, and that is why it was not tested in that simulation tool. Instead, the SJTC will be simulated in URSim docker and then tested in the real UR3 robot.

In order to send commands to the controller and set goals for the UR3, the action interface used is the same as for the JTC (control_msgs/action/FollowJointTrajectory). Action goals allows to specify the trajectory to execute, and path and goal tolerances. If no tolerances are specified, the default values are used. Tolerances could be useful in specific applications for precise robot movements. If the tolerances are not met, the action goal is aborted (when a goal is aborted, the controller will attempt to execute the trajectory as good as possible).

To send goals, users can use terminal commands or publisher_joint_trajectory_controller executable, setting the goal configuration in a YAML file. The advantage of using files instead of the terminal, is that the user can set several goals to execute in a row.

To start the URSim docker simulation, run the following commands in a new shell and follow the steps described in section 8. Starting Universal Robots Simulations.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash  
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

Then, open a new shell and launch the UR driver.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.56.101
```

That will start the controllers and broadcasters needed, and RViz tool.

To use the terminal to send goals through the action server, open a third shell and send the following command:

In Shell #3

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 action send_goal /scaled_joint_trajectory_controller/follow_joint_trajectory  
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:  
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo  
int], points: [{positions: [0.0,-1.57,0.0,-1.57,0.0,0.0]}, velocities: [], accelerations:  
[], time_from_start: {sec: 6, nanosec: 0}}]}"
```

The second option is to publish goals using files. To do it, launch the test_scaled_joint_trajectory_controller.launch.py. This file starts an executable called publisher_joint_trajectory_controller.py from the ros2_controllers_test_nodes package, inside ros2_controllers package. This executable checks the goals set in the test_goal_publishers_config.yaml file and publishes them in the action server if they are correctly configured.

In Shell #3

```
ros2 launch ur_robot_driver test_scaled_joint_trajectory_controller.launch.py ur_type:=ur3
```

The test_scaled_joint_trajectory_controller.launch.py requires the robot to be in the same initial position as the test_joint_trajectory_controller to accept the goals. However, I have experience some cases when the limits set in the starting_point_limits parameter from test_goal_publishers_config.yaml are ignored, and it will only work if the default starting position is set ([0.0,-1.57,0.0,-1.57,0.0,0.0]). Unfortunately, it was not possible changing the default starting position for the URSim or UR3 robot during this project.

Therefore, the robot will have to be in the initial position established in the test_goal_publishers_config.yaml or, if those limits are ignored, in the default home position [0.0,-1.57,0.0,-1.57,0.0,0.0].

One option to set the robot to the starter position is to run the action server command through the terminal with the desired position. A second option is to move the robot in the URSim using the “Free Drive” mode. To do it go to the Move tab of the simulation window.

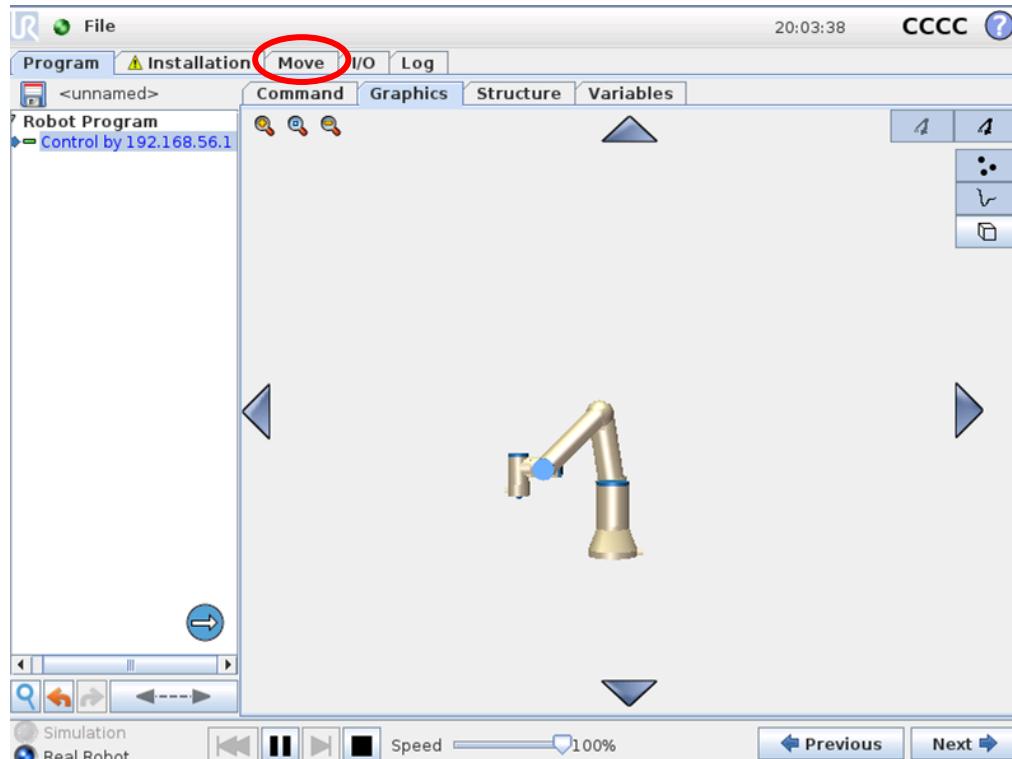


Figure 62. Move to the Move tab to move the robot to the home position manually. Source: Created by the author.

Then select one of the Move Joints angles to set the initial position as goal.

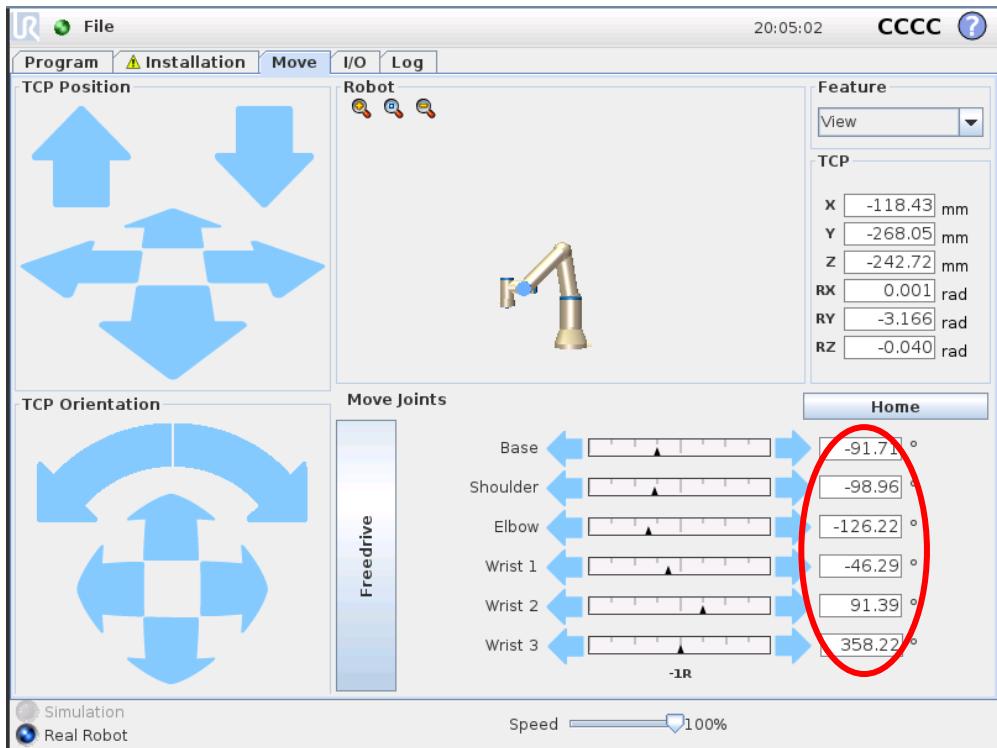


Figure 63. Select Move Joints angles to set the intial position as goal. Source: Created by the author.

Then set the desired angles for all the joints and press OK.

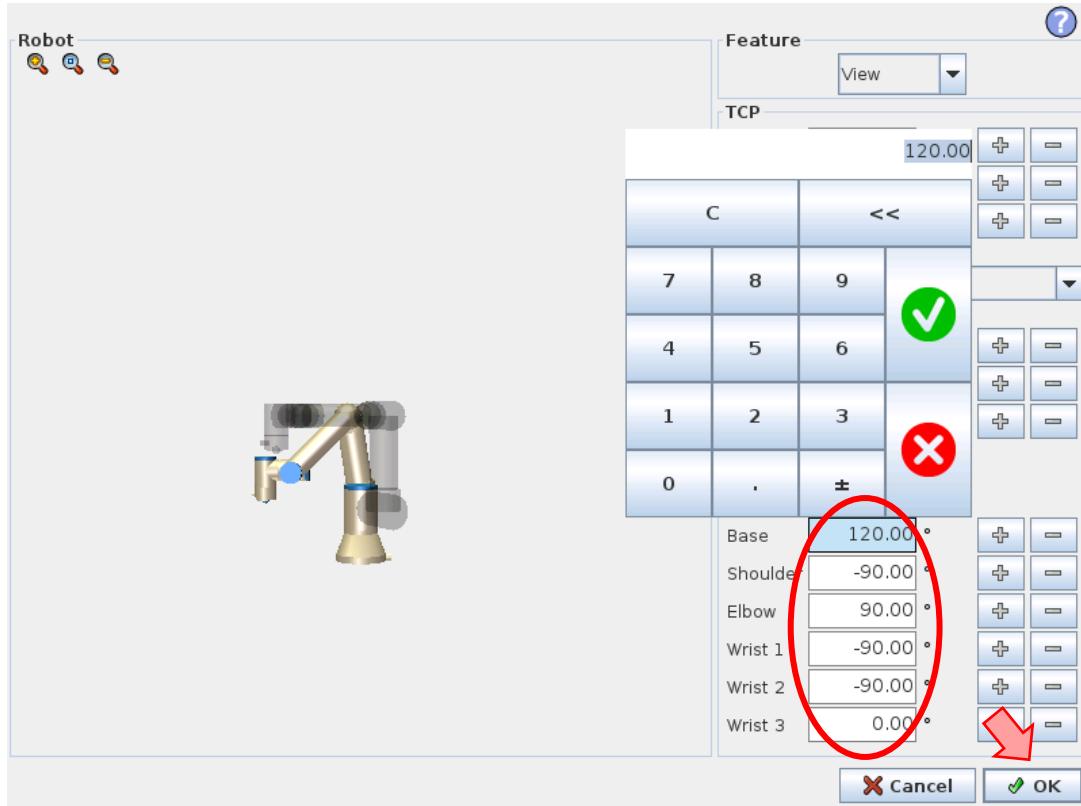


Figure 64. Introduce the initial position and click OK. Source: Created by the author.

Then, click on Auto and maintain it until the button on the bottom right corner is set to OK.

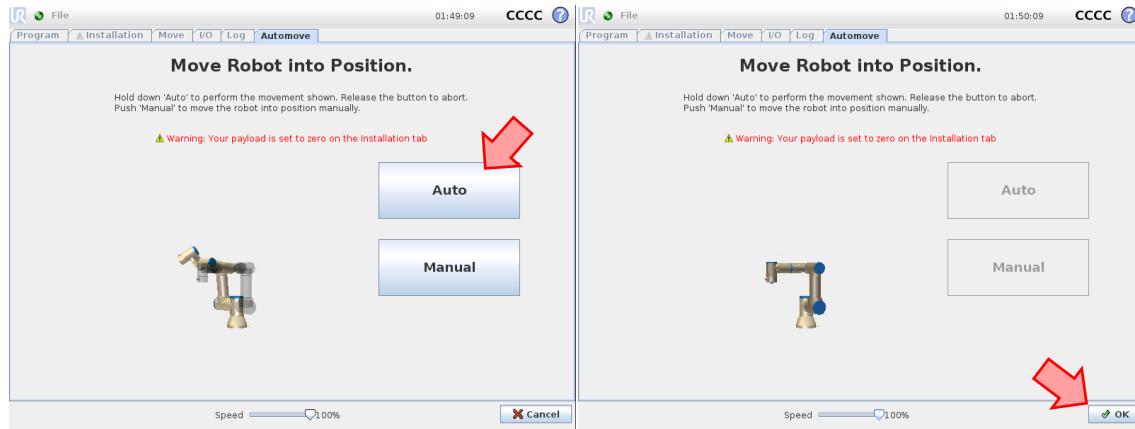


Figure 65. Click Auto until the Cancel button changes to Ok. Source: Created by the author.

Then click Ok, go back to the Program tab (you will see the robot in the home configuration), and Play the external control.

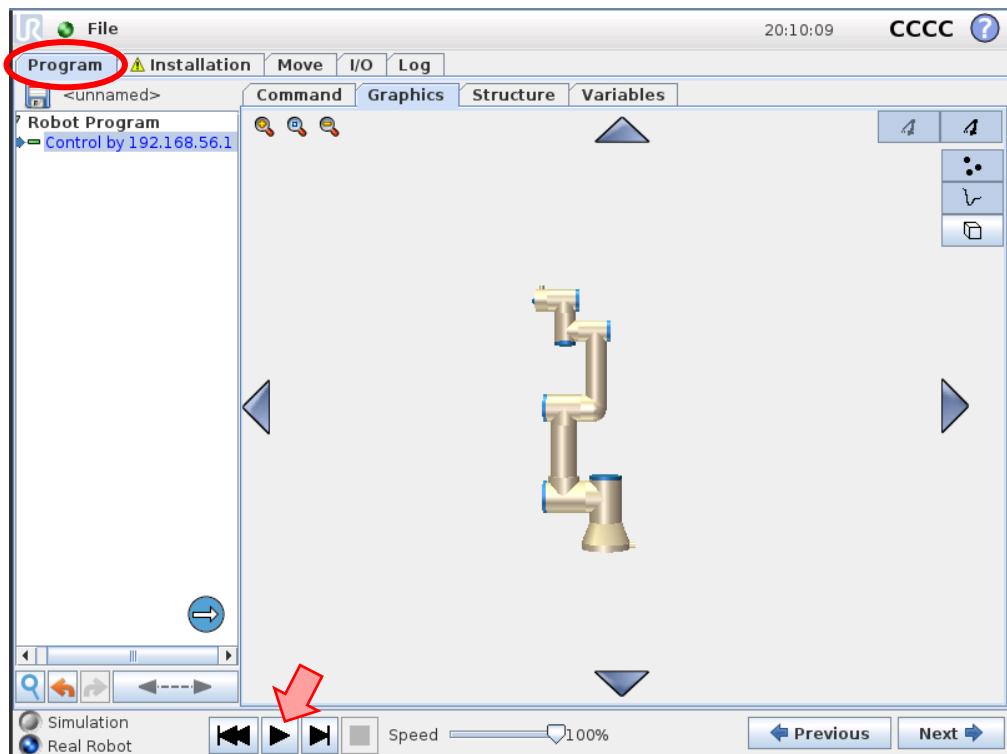


Figure 66. Go back to Program tab and Play the external control. Source: Created by the author.

The third option is to modify the `test_goal_publishers_config.yaml` file and change the `check_starting_point` parameter to false in line 21. By doing that, it will not matter the initial position of the robot, the goal will be published and executed by the simulation if the joints limits are exceeded.

```
21 check_starting_point: false
```

Once any of those three procedures is followed, proceed to launch the test_scaled_joint_trajectory_controller.launch.py.

If you want to set different goals, change the values in lines 8, 9, 10 and 11 from this same file.

```
8 pos1: [0.785, -1.57, 0.785, 0.785, 0.785, 0.785]
9 pos2: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]
10 pos3: [0.0, -1.57, 0.0, 0.0, -0.785, 0.0]
11 pos4: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]
```

Remember that, if you modify any file, you will have to stop the simulation, build and source the workspace and restart the simulation, in order for the changes to apply.

Note: test_goal_publishers_config.yaml code, together with other important files, will be added and commented in Appendix 1. Commented Code.

The same process should be followed to control the real UR3, but in this case, instead of running the URSim, the real UR3 must be configured and connected to the control PC, as explained in section 10. UR3 Robot Setup.

In the case of the real UR3, due to laboratory restrictions and the provided environment for the robot, the default start position for the UR manipulator cannot be reached. That is why the only solution to send goals through test_scaled_joint_trajectory_controller.launch.py is by setting check_starting_point parameter to false.

Another point to consider is that, because of environment limitations, the robot has joint motion restrictions. That causes that the default goals set in the test_goal_publishers_config.yaml cannot be reached. Here are some values that could be modified in the file to meet the restrictions.

```
8 pos1: [2.09, -1.57, 1.57, -1.57, -1.57, 0.0]
9 pos2: [3.5, -2.57, 1.1, -2.3, 1.1, 0.0]
10 pos3: [3.5, -1.57, 0.5, -2.3, -0.785, 0.0]
11 pos4: [2.09, -1.57, 1.57, -1.57, -1.57, 0.0]
```

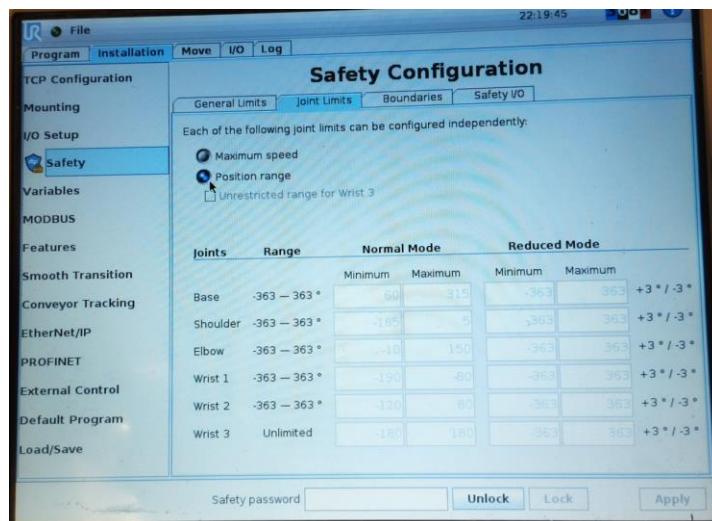


Figure 67. Laboratory real UR3 joint position range restrictions. Source: Created by the author.

Once the robot is properly setup and connected to the control PC, and the proper changes have been made in the mentioned files, in PolyScope main screen select Program Robot. Inside Program Robot menu, access Load Program and start the URCaps program that you created while following section 10. UR3 Robot Setup. In this case, the program created was named dan.urp.

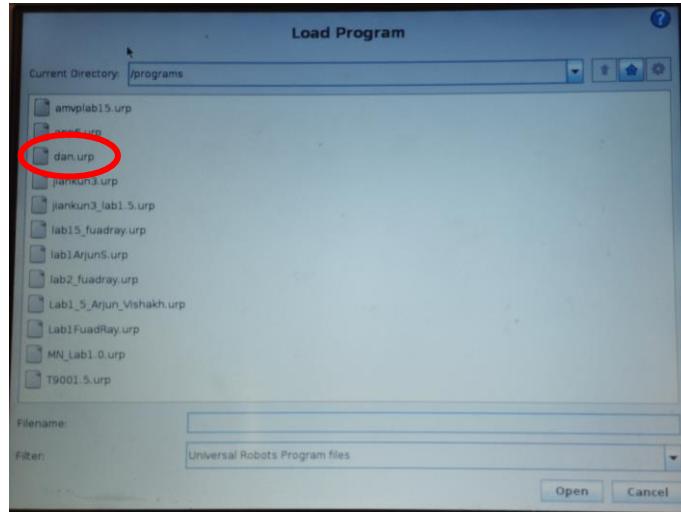


Figure 68. URCaps program to control the real UR3. Source: Created by the author.

Then, open a shell and command:

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver
colcon build
source install/setup.bash
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.10.53
```

Note that the robot IP is set to the address of the real UR3.

Next, open a second shell, source the workspace and send goal messages to the action server through commands or launching the test_scaled_joint_trajectory.launch.py file to watch the robot move to the four goals repetitively. Remember to Ctrl+C to stop sending goals.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver
source install/setup.bash
ros2 action send_goal /scaled_joint_trajectory_controller/follow_joint_trajectory
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo
int], points: [{positions: [0.0,-1.57,0.0,-1.57,0.0,0.0], velocities: [], accelerations:
[], time_from_start: {sec: 6, nanosec: 0}}]}}}
```

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver
source install/setup.bash
ros2 launch ur_robot_driver test_joint_trajectory_controller.launch.py ur_type:=ur3
```

11.5.rqt_joint_trajectory_controller

Another controller was tested in URSim simulation and the real UR3 robot. In this case, this controller implements a GUI with sliders that allows the user to move every joint independently and adjust the speed scaling of the movement. This controller is based on scaled_joint_trajectory_controller and provides a useful tool to execute precise moves with the robot similar to the ones that can be commanded with PolyScope “FreeDrive” mode, but from the control PC.

The process to follow for the URSim simulation is the same as for the SJTC.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash  
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

Then, open a new shell and launch the UR driver.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.56.101
```

Now, in the third shell you will execute the following command:

In Shell #3

```
ros2 run rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```

The GUI will appear in the screen.

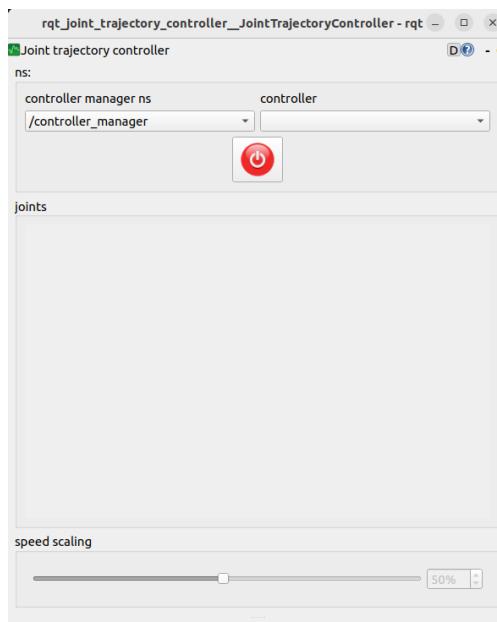


Figure 69. GUI displayed after running the rqt_joint_trajectory_controller. Source: Created by the author.

Select the SJTC in the controller dropdown and turn on the controller. Make sure that the controller manager ns is set to /controller_manager. The sliders will change from gray to blue and you will be able to change the position of the motors with it and adjust the speed scaling with the bottom slider.



Figure 70. GUI after turning on the controller. Source: Created by the author.

Note that the response of the robot is not immediate, and you might notice that the motor keeps moving after you release the slider. That is due to the update rates of the simulation and the controller.

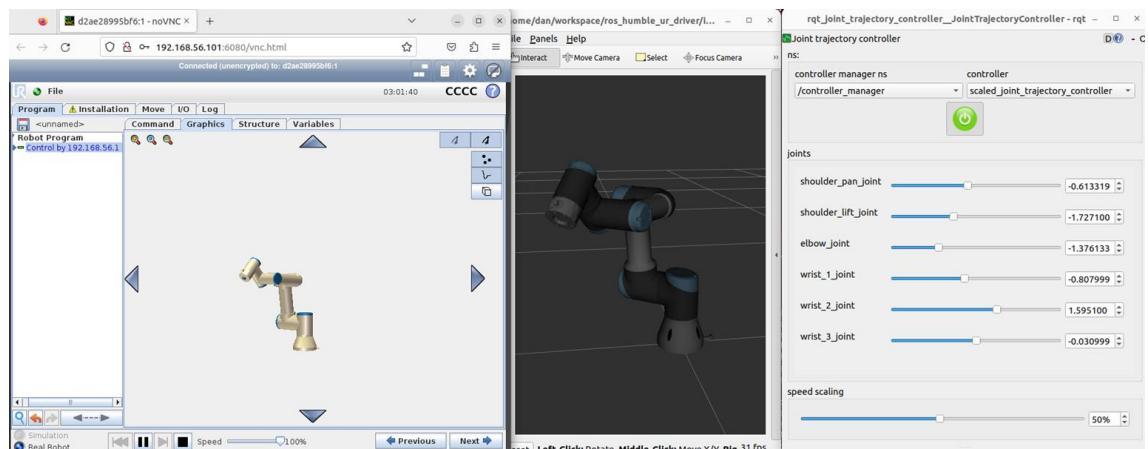


Figure 71. URSim, RViz and GUI after performing some joint position changes. Source: Created by the author.

The same procedure needs to be followed to test it with the real UR3 but performing the robot initialization described in the SJTC section.

Then in two new shells execute:

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash  
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.10.53
```

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 run rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```

Consider that because of the joint movement limitations of the real UR3, the robot might reach its upper or lower limits while moving the sliders. By editing some files from the `rqt_joint_trajectory_controller` package in the `ros2_controllers` package, there could be ways to limit the joint motion range in the GUI displayed when running the controller. But that was out of the scope of the project.

11.6. MoveIt 2 control

There is one more tool that was used during this project to control the UR3 robot in simulations and the laboratory, MoveIt 2. But this tool is more complex than a controller and requires some further explanation. Therefore, MoveIt 2 will be described in section 13. MoveIt 2 of the document and the process to control the UR3 using it will be introduced and explained step by step.

12. DATA AQUISITION AND ANALYSIS

Data acquisition and analysis play a crucial role in robot control as they provide valuable insights into the behavior of the robot and enable the development of effective control strategies. Through data acquisition, sensors can gather information about the robot's environment and internal state. This data can be used to optimize the control algorithms, improve the robot's performance, and ensure safety. By analyzing the acquired data developers can identify potential issues and debug problematic robot's behavior. Data analysis can reveal patterns in the robot's performance, such as unexpected movements or deviations from the desired trajectory. Additionally, data analysis can be used to optimize the control algorithms, enabling the robot to perform more efficiently and effectively.

There are different ways to analyze data from a ROS2 application like the control of a UR3. RViz is an example of this, by transforming sensor and robot data into a graphic interface, users can visualize the environment as the robot does, and keep track of robot movements and behaviors.

However, there is another powerful tool, developed by Davide Faconti, called PlotJuggler that implements a real-time data stream for ROS2 procedures. With this tool, users can analyze published data from the robot or sensors at the moment and organize it in different layouts, which can be then saved and relogged anytime for future examination.

Some of the most interesting features mentioned in its GitHub repository [23] are:

- Simple Drag & Drop user interface.

- Load data from file.
- Connect to live streaming of data.
- Save the visualization layout and configurations to re-use them later.
- Fast OpenGL visualization.
- Can handle thousands of timeseries and millions of data points.
- Transform your data using a simple editor: derivative, moving average, integral, etc...
- PlotJuggler can be easily extended using plugins.

As mentioned in the last point, PlotJuggler can be integrated with ROS or ROS2 through plugins, and they can both be installed as Debian packages for any ROS version. For this project, the Humble version will be chosen.

In Shell #1

```
sudo apt install ros-humble-plotjuggler-ros
```

Once PlotJuggler is installed in your underlay workspace, there are three tutorials that are highly recommended to take before starting using PlotJuggler. These tutorials will not be covered in this document, as they are already condensed and simple enough to get basic notions of the tool [8] [7] [6].

In this section, an example of basic PlotJuggler layout will be designed for the UR3 control and will be tested for simulation.

Before designing a layout using real-time data with PlotJuggler, it's important to ensure that a control framework has already been established, since PlotJuggler relies on data that is published in topics. The designed layout for real-time data analysis can also be used to study recorded data from the topics. Rosbag 2 will be used to record data and demonstrate the PlotJuggler capability of post-test evaluation.

Rosbag 2 should be included in ROS2 Humble if you followed the steps from section 5. Setup and Installation. To start using it, first create a directory to store the recorded data files.

In Shell #1

```
cd  
mkdir bag_files
```

The rosbag files will be saved in the directory where you run it, so make sure that you are in the bag_files directory once you start recording a topic.

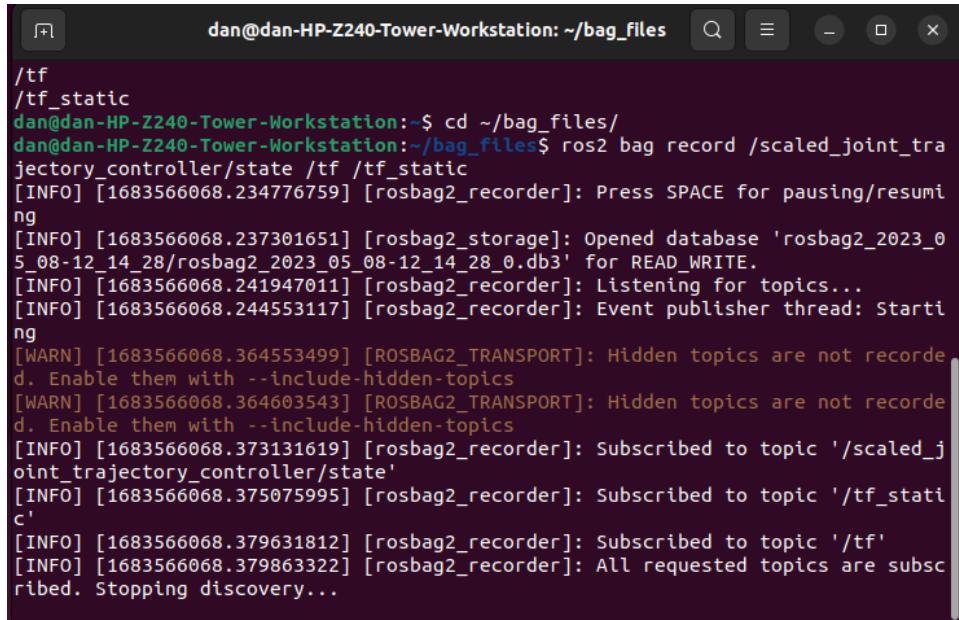
The tested controller is the scaled_joint_trajectory_controller. Therefore, a URSim simulation must be started before running PlotJuggler. After following the steps from sections 8. Starting Universal Robots Simulation and 11. Controlling the UR3, you will end up with a shell running the URSim docker simulation, a shell running the driver and a sourced shell ready to run control commands.

Next, open a fourth shell to start data recording.

In Shell #4

```
cd ~/bag_files
```

```
ros2 bag record scaledjointtrajectorycontroller/state /tf /tf_static
```



The screenshot shows a terminal window titled "dan@dan-HP-Z240-Tower-Workstation: ~/bag_files". The window displays the command "ros2 bag record scaledjointtrajectorycontroller/state /tf /tf_static". The terminal output includes several INFO messages from the rosbag2_recorder node, such as "Press SPACE for pausing/resuming", "Opened database 'rosbag2_2023_05_08-12_14_28/rosbag2_2023_05_08-12_14_28.0.db3' for READ_WRITE.", and "Listening for topics...". It also shows WARN messages about hidden topics and subscription to "/scaled_joint_trajectory_controller/state" and "/tf". The message "[INFO] [1683566068.375075995] [rosbag2_recorder]: Subscribed to topic '/tf_static'" is highlighted.

Figure 72. Data being recorded with rosbag. Source: Created by the author.

The data studied in this project will be extracted from the SJTC to analyze how motors behave when a goal is sent to the UR3. The data sheared in the /tf and /tf_static topics could also be interesting, as it comprises the cartesian position and orientation of the link frames of the robot. By creating a frame at the end effector of the robot with the base frame as parent frame, the user could be able to know the position and orientation of the tool while the robot moves. This will not be studied in this project, but it is worth mentioning to understand different usages of robot data.

Once the setup is done, initiate PlotJuggler by:

In Shell #5

```
ros2 run plotjuggler plotjuggler
```

A window with PlotJuggler GUI will open. There, select ROS2 Topic Subscriber from the left dropdown and press Start.

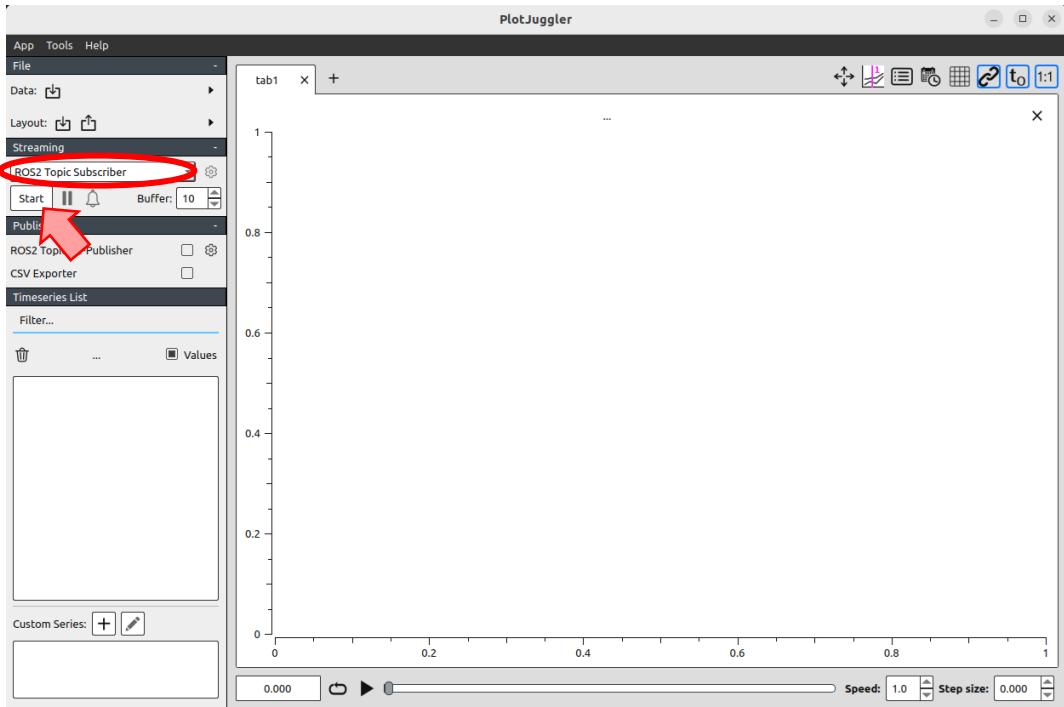


Figure 73. PlotJuggler initial GUI. Source: Created by the author.

Select the desired topics from the displayed menu. For this project, the topics to analyze in real-time will be the same as the recorded ones.

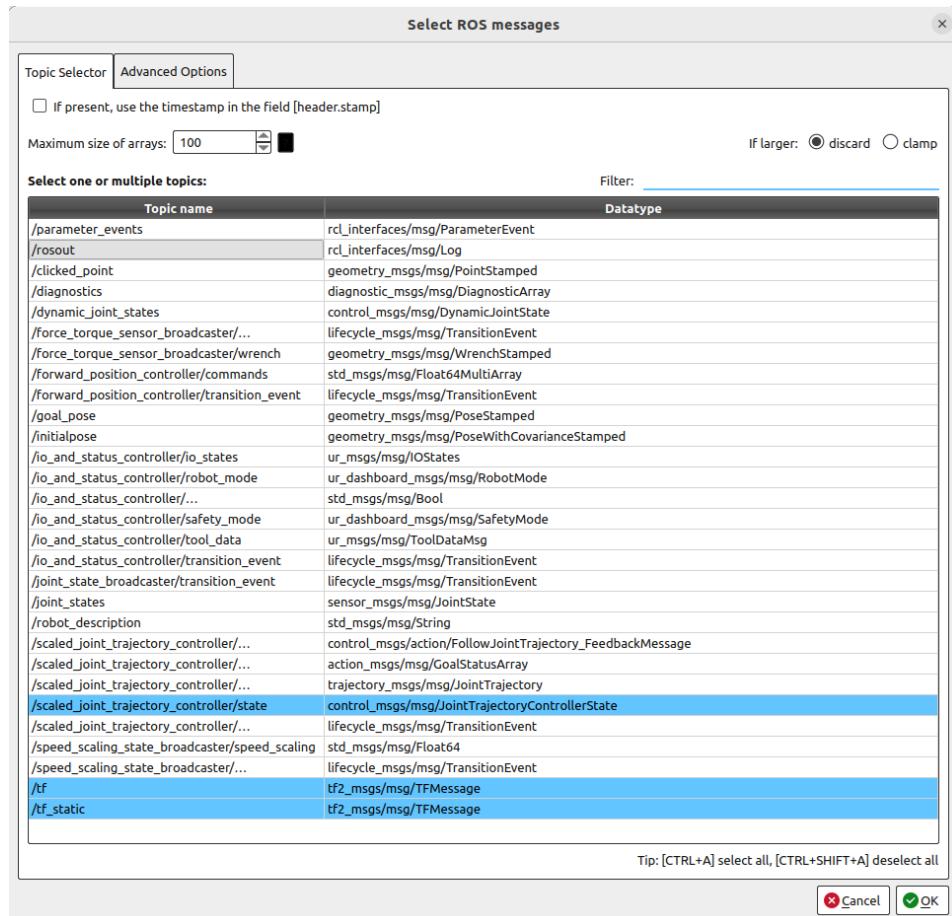


Figure 74. Topic selector menu. Source: Created by the author.

Once the topics are selected and the data stream initiates, begin configuring your layout.

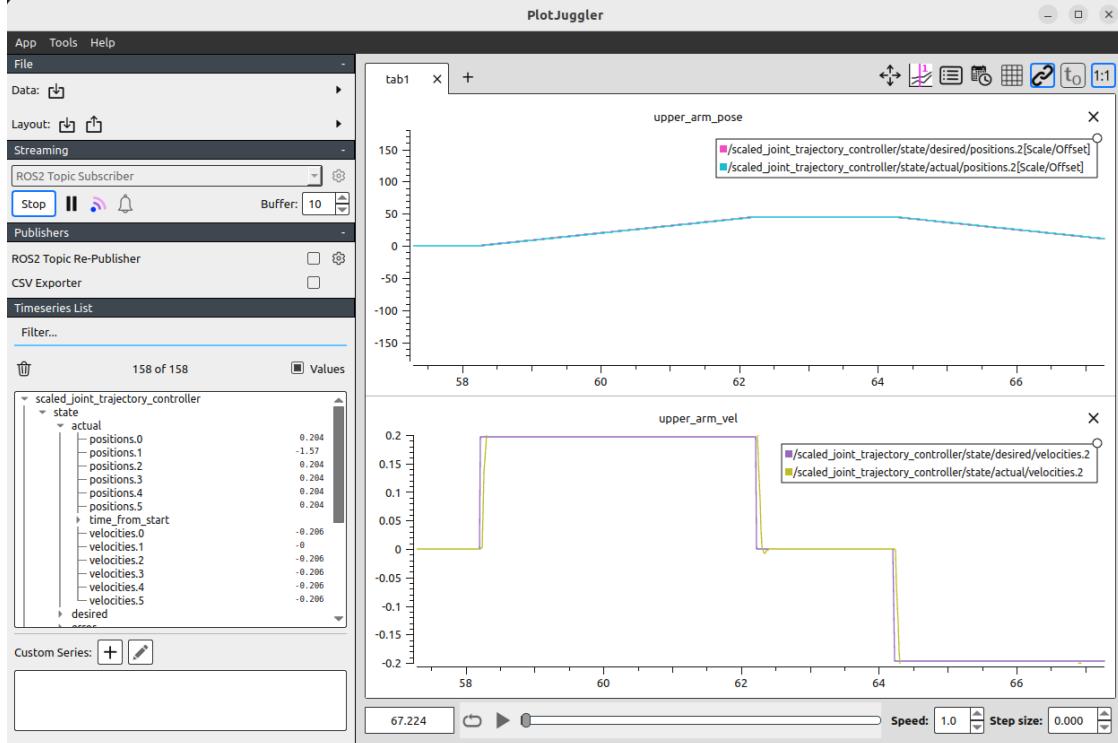


Figure 75. PlotJuggler layout designed for real-time data stream. Source: Created by the author.

In this project the chosen layout was quite simple, since the UR3 was not required to perform any specific trajectory to be monitored. The idea was to observe the desired position and velocity of one of the motors compared to the actual position and velocity. In this case the selected joint was the elbow_joint. From the figure it can be seen that the motor executed a linear trajectory, moving from position 0° to 45° in 4 seconds and it was moving towards its next goal after 6 seconds passed from the previous published goal, as stated in the test_goal_publishers_config.yaml file. The velocity of the motor varied by steps, with a maximum absolute value of around 0.2 rad/s and experiencing an insignificant overshoot with a settling time of 0.2 seconds.

Once the robot completes its task, in this case the UR3 finishes the desired trajectories, the layout can be saved.

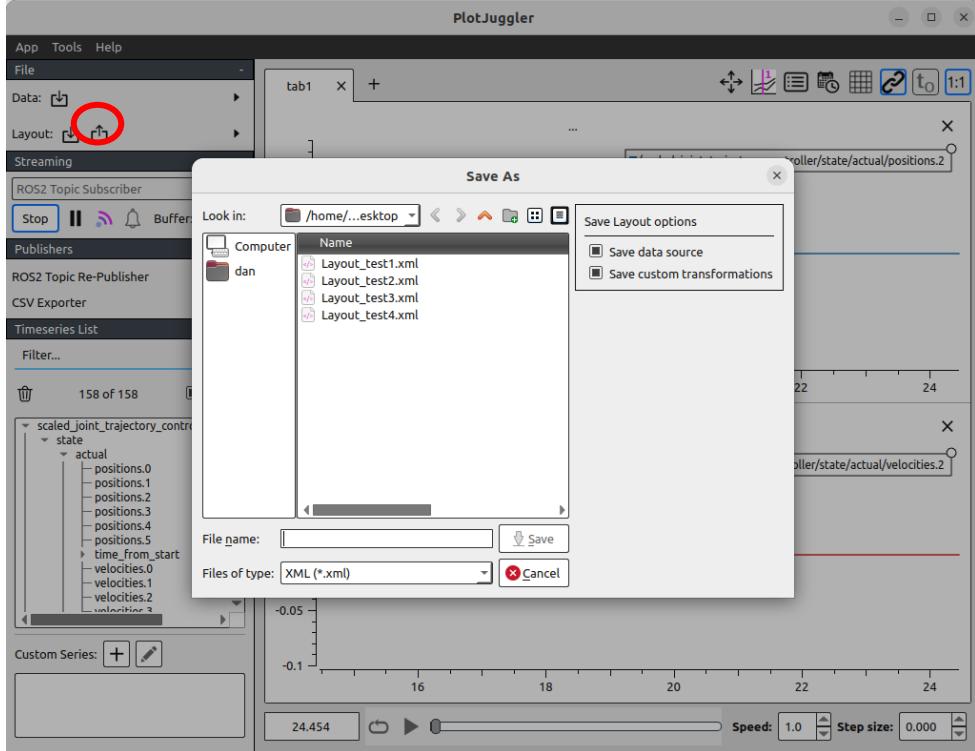


Figure 76. Layout saving screen. Source: Created by the author.

Once the layout is saved, all the running processes can be stopped (simulation, driver, data recording and PlotJuggler). If the user wants to check the recorded data from the previous test, PlotJuggler can be initiated again. Then the data can be loaded from the saved rosbag file.

Technically, the recorded data could be analyzed in the same layout as the real-time test by just loading the saved layout after the recorded date. Unfortunately, during this project every single layout saved from a real-time test were causing a PlotJuggler crush. The terminal outputted the message shown in **Figure 77**.

Figure 77. Message outputted after PlotJuggler crush for trying to load a saved layout. Source: Created by the author.

From the message, it can be deduced that the problem is caused by some internal coding error. In the future this error might be debugged.

In order to check the recorded data, the same layout was designed in the new PlotJuggler window.

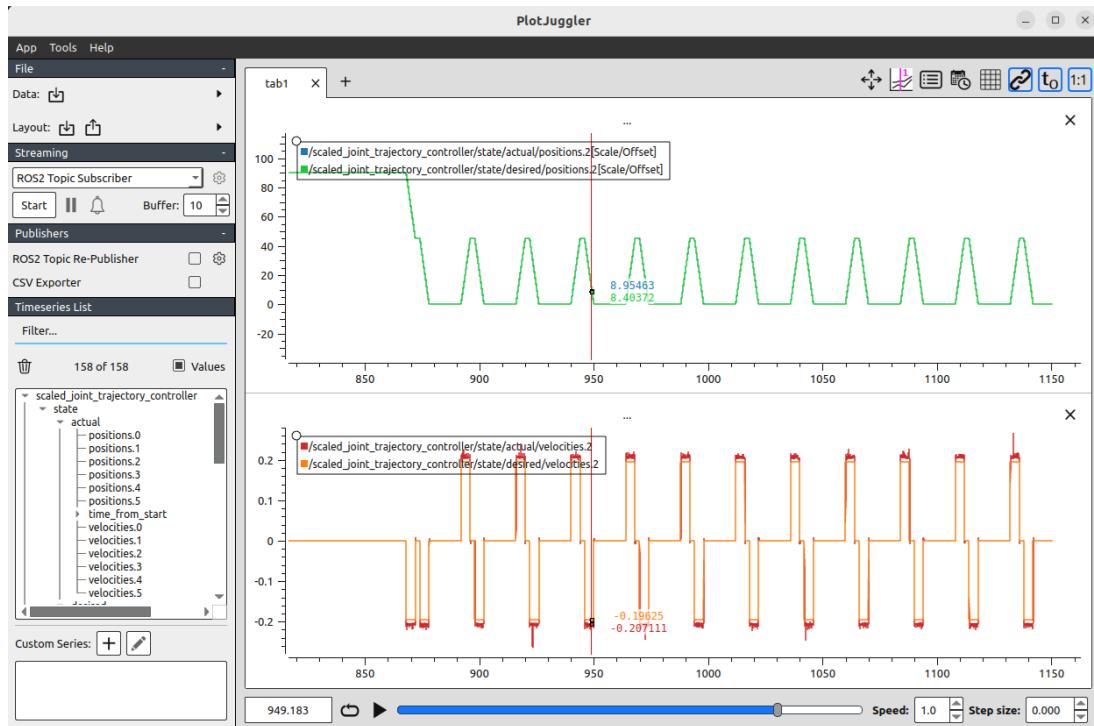


Figure 78. Recorded data representation with the same layout as in real-time. Source: Created by the author.

These are the movements executed by the elbow_joint while running the test scaled joint trajectory controller.launch.py. With this method, users can review the

recoded data from the ROS2 application and check errors or bugs spotted while testing in real-time or acknowledge new issues while inspecting the robot performance.

13. MOVEIT 2

In this section, an introduction to MoveIt 2 will be given, allowing to understand its architecture and utilities in a generic manner. A step-by-step explanation about how to control the UR3 robot using MoveIt will be also given, using the packages and tools already installed and setup explained in previous sections.

MoveIt2 is a software framework, built on top of ROS2, that enables motion planning, control, and manipulation of robots in complex environments. It is designed to provide an intuitive, user-friendly interface for controlling robots, enabling users to quickly and easily create and execute robot motion plans. It has its own website with useful information that was the base for the theoretical concepts described in this document. To find more information about MoveIt check out [\[51\]](#) [\[52\]](#) [\[55\]](#).

MoveIt utilizes a plugin-based structure, designed to enable users to create their own inverse kinematics algorithms. Forward kinematics and the process of finding Jacobians are integrated within the RobotState class. MoveIt's default inverse kinematics plugin is set up using the KDL numerical Jacobian-based solver.

That means MoveIt can use the robot description package to obtain its inverse kinematics using this solver. The solver computes the inverse kinematics solution for the robot's end effector given a desired pose and considers any joint limits, collision avoidance, and other constraints specified by the user. This allows MoveIt to plan and execute complex motion sequences that involve precise positioning of the robot's end effector while avoiding obstacles in the environment.

13.1. MoveIt Architecture

The MoveIt2 architecture looks like the following diagram.

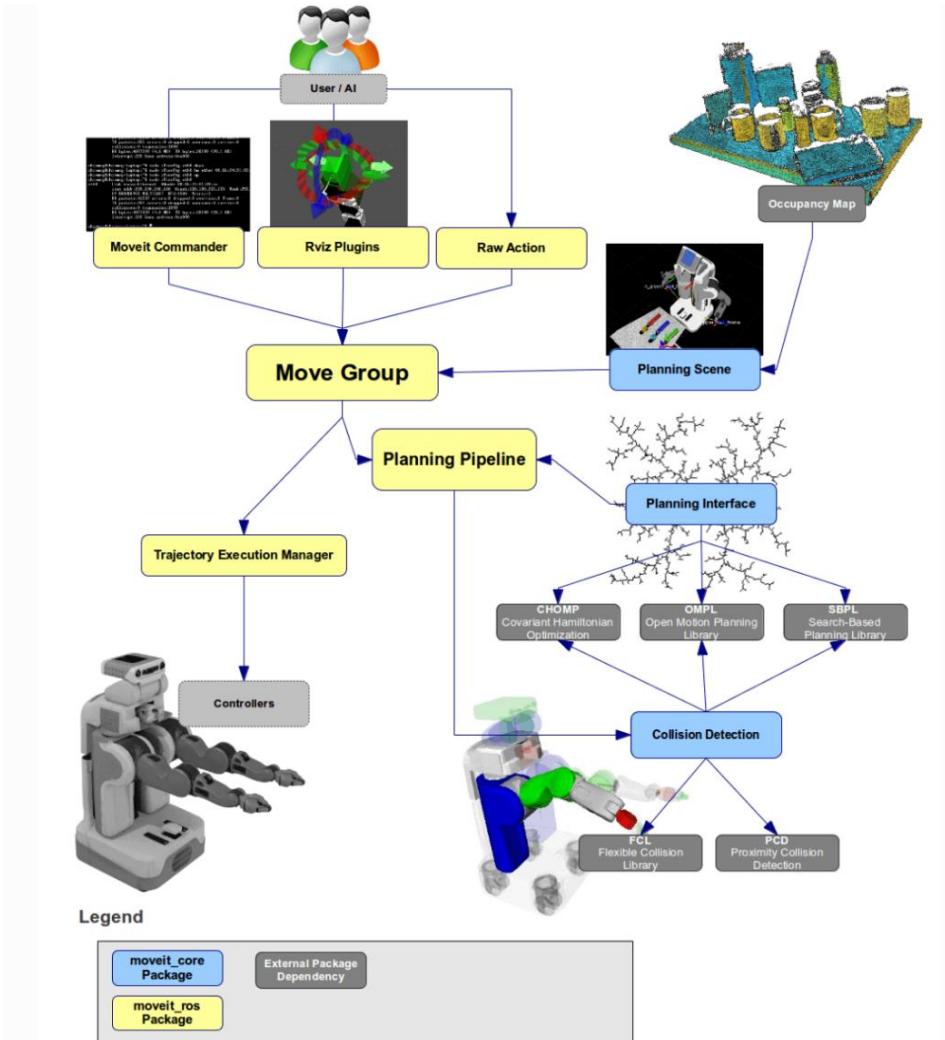


Figure 79. MoveIt2 architecture. Source: PickNik Robotics, <https://moveit.picknik.ai/humble/doc/concepts/concepts.html>.

The architecture of MoveIt2 is divided into three main components: the motion planner, the planning scene, and the robot controller.

The motion planner component generates a sequence of robot motions that satisfy a given task specification while avoiding collisions with the environment. The plugin interface that MoveIt is based on, allows users to use different motion planners. To communicate with them through ROS2 actions or services, the `move_group` node is used, which has its own default motion planners configured using Open Motion Planning Library (OMPL).

Users can define a desired motion plan request, including different kinematic constraints, and the `move_group` node will generate the trajectory in response to the motion plan request following the restrictions. It is important to note that the `move_group` node generates a trajectory, but to get to the final trajectory, a previous motion plan must be established. This motion plan is generated through the motion planning pipeline, which includes the motion planner and planning request adapters.

These planning request adapters are responsible for pre-processing motion plan, for example if the robot is slightly outside the specified joint limit, and for post-processing motion plan, for example to convert the path generated into time-parametrized trajectories. More information about motion planning can be found here [\[53\]](#).

The planning scene component provides a representation of the robot's environment, including the robot itself, the objects it needs to manipulate, and the obstacles it needs to avoid. The planning scene is updated based on sensor input from the robot's sensors and from user input, and it is maintained by a planning scene monitor using all data input. You can find more information about the planning scene here [\[54\]](#).

All the collision checking is also configured inside the planning scene using CollisionWorld object. It is carried out automatically by the FCL package, so users do not need to worry about how it is happening. More information about how collision checking internally works in [\[52\]](#).

Finally, the robot controller component sends commands to the robot's actuators to execute the generated motion plan. MoveIt2 provides a variety of control interfaces, including joint position, joint velocity, and Cartesian control, that can be used to control the robot.

All these main components of the MoveIt architecture are coordinated by the key MoveIt node, move_group. This node provides a set of ROS2 actions and services that the user can use by pulling all the individual components.

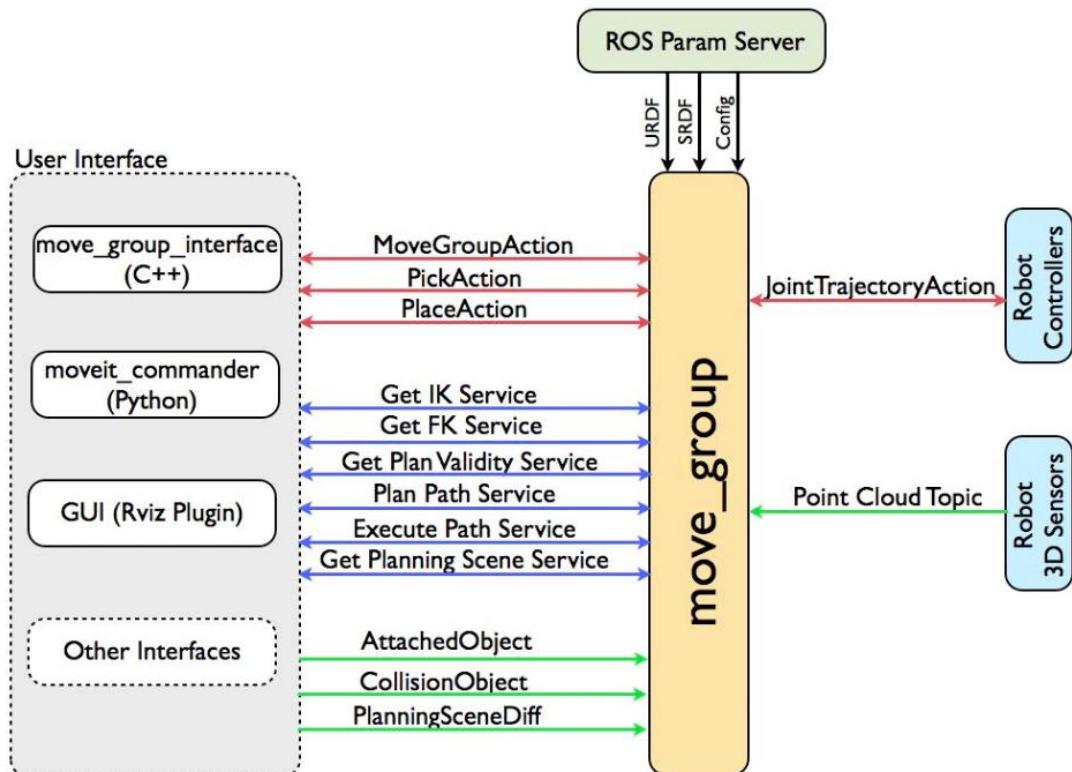


Figure 80. move_group architecture and relation to MoveIt components. Source: PickNik Robotics [\[55\]](#).

It can be seen in the figure how move_group connects component from MoveIt framework. Some of them were explained in previous sections, such as the ROS Param Server, that refers to the robot description files, the robot controllers or RViz plugin as visualization tool.

In fact, user interface comprises RViz plugin as one of the ways users can communicate with MoveIt to control a robot. Later in this section, RViz interface for MoveIt will be explained and use it to control the UR3, as well as other user interface like executables using move_group_interface.

More information about the move_group node can be found here [\[55\]](#).

As MoveIt packages are complex to design and create, there is a graphical interface tool called MoveIt Setup Assistant that allows users to create their own MoveIt package in an easy way and configure it for any robot. MoveIt Setup Assistant generates SRDF, configuration files, launch files, and scripts from the robot URDF model, which is required to configure the move_group node.

Unfortunately, this tool is not yet provided for ROS2, so in order to use MoveIt with it, developers usually edit and transform the files generated with the assistant for ROS1 into ROS2. In the case of the UR3, everything needed to control it with MoveIt was provided by Universal Robots and the rest of the packages installed in previous sections.

13.2. RViz Plugin

As mentioned, a simple way to interact with MoveIt is through its RViz plugin. This plugin is called Motion Planning and can be added as a display component. It has two menus to interact with, the display dropdown menu and its own Motion Planning menu, added below the description chart.

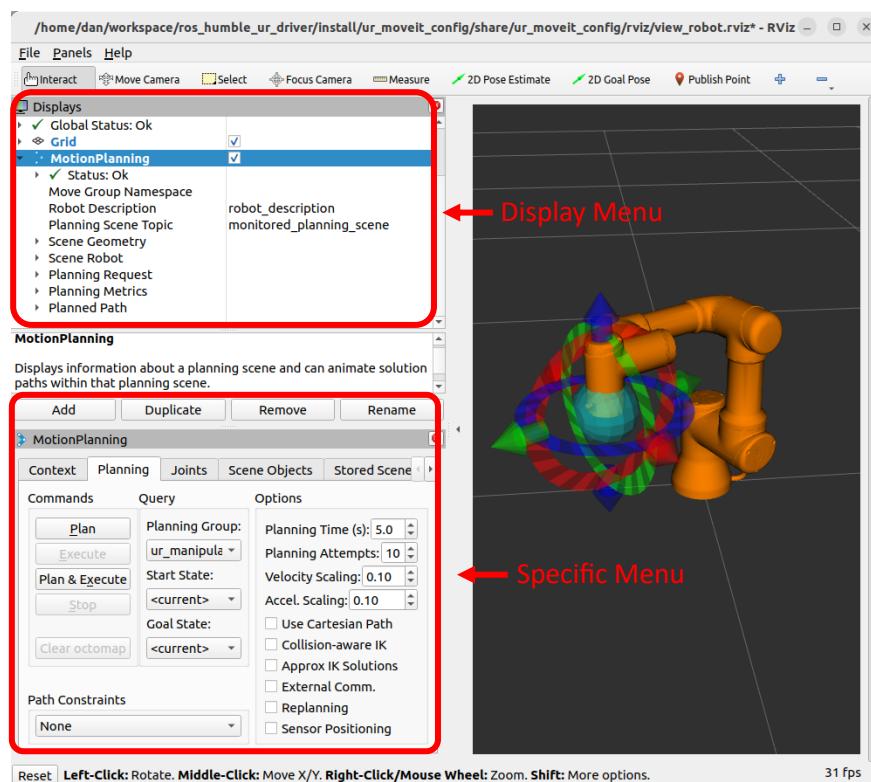


Figure 81. RViz GUI with UR3 Motion Planning added. Source: Created by the author.

The Motion Planning display menu allows users to modify the visual components of the scene, as well as select the robot description parameter where the URDF of the robot is loaded, the topic on which the MoveIt messages are received (Planning Scene Topic), and the name of the group of links to plan for (Planning Group), defined in the SRDF generated with the MoveIt Setup Assistant.

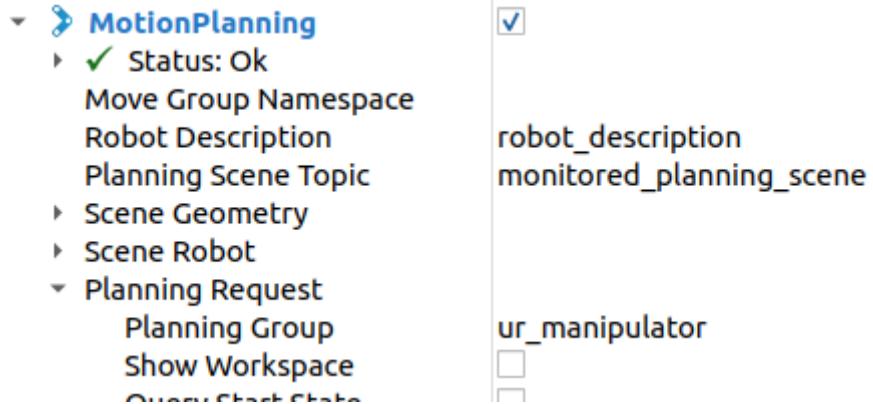


Figure 82. Sections from the display menu where users can select the robot description, planning scene topic and planning group. Source: Created by the author.

The Scene Geometry is used to set the scene visual properties, and the Scene Robot to set the robot visual properties, including collisions and objects attached to the robot.

| Scene Geometry | |
|----------------------|-------------------------------------|
| Scene Name | (noname)+ |
| Show Scene Geometry | <input checked="" type="checkbox"/> |
| Scene Alpha | 0.9 |
| Scene Color | [Green] 50; 230; 50 |
| Voxel Rendering | Occupied Voxels |
| Voxel Coloring | Z-Axis |
| Scene Display Time | 0.01 |
| Scene Robot | |
| Show Robot Visual | <input checked="" type="checkbox"/> |
| Show Robot Collision | <input type="checkbox"/> |
| Robot Alpha | 1 |
| Attached Body Color | [Purple] 150; 50; 150 |
| Links | |

Figure 83. Scene Geometry and Scene Robot dropdowns from display menu. Source: Created by the author.

The Planning Request, apart from letting the user select the planning group, it is used to change the color of the robot silhouettes that appear in the scene to follow the planning process (goal state, start state, colliding links, or joint violation silhouettes). Meanwhile, the Planning Metrics display information in the robot scene as text about the robot weight limit for a particular pose for an end effector, the manipulability and manipulability index for an end effector, and the joint torques for a certain configuration and payload.

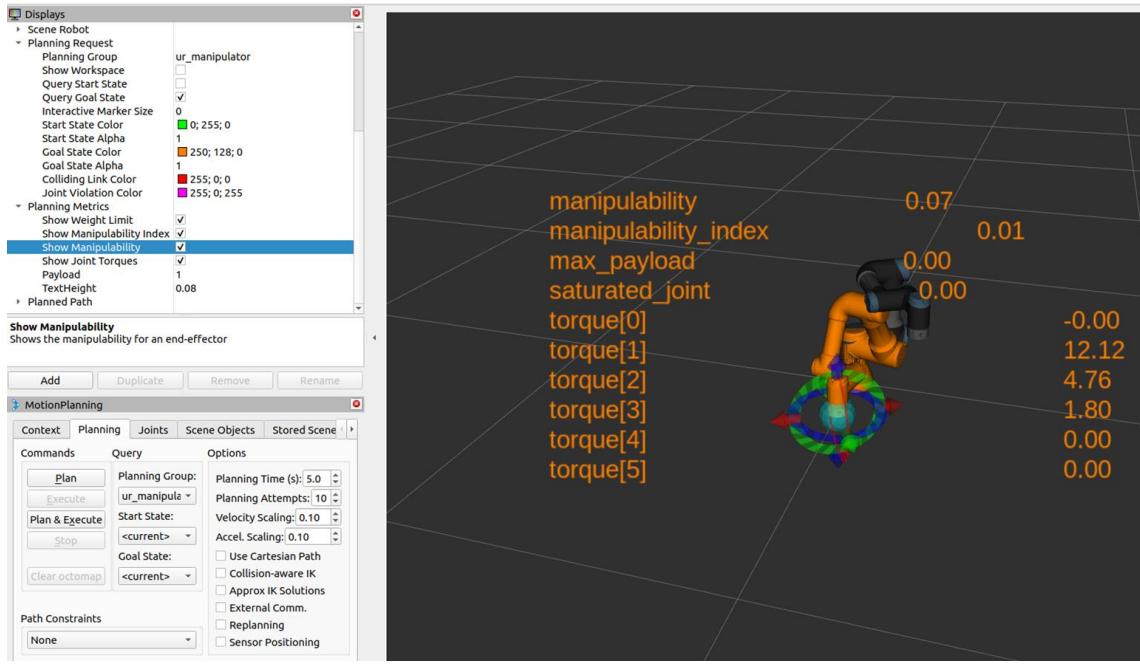


Figure 84. Planning Request and Planning Metrics dropdowns from display menu, and impacts in the RViz robot scene. Source: Created by the author.

Lastly, the Planned Path is used to display the silhouette of the robot while performing a planned path, as well as configuration for this animation. Once a path is planned, the robot will not move unless the user executes the trajectory, so to check the motion that the robot will do before executing, a planning animation will display a silhouette of the robot following the path. The user can modify parameters as the trajectory topic, the collisions during the animation, looping the animation, using simulation time to display the animation, or the color of the silhouette.

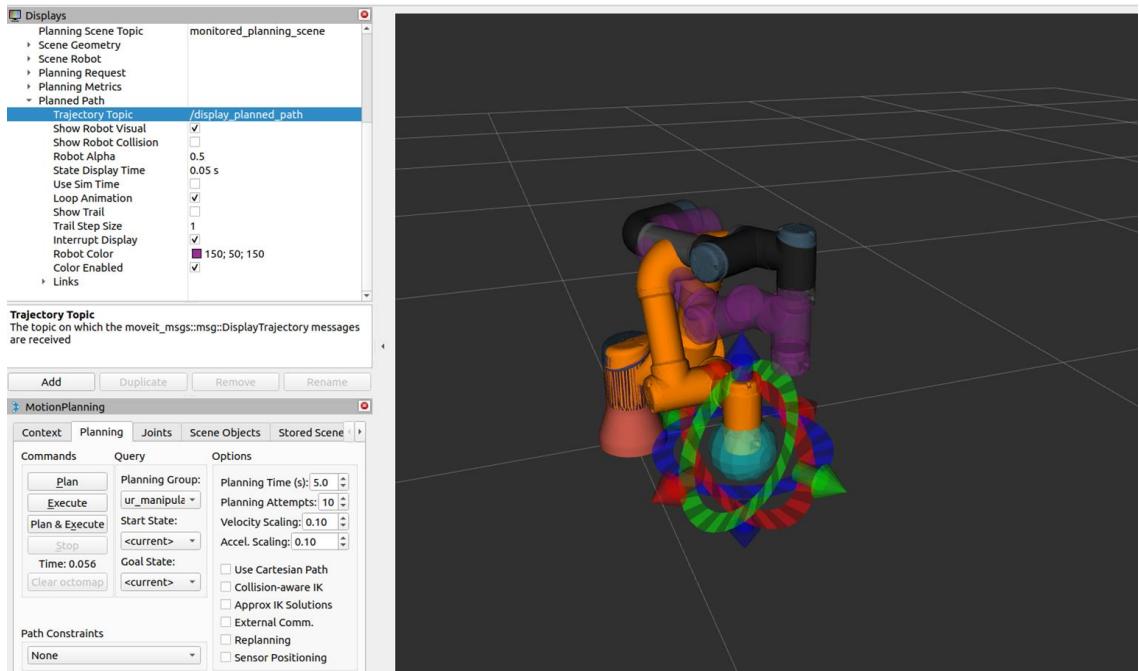


Figure 85. Planned Path dropdown from display menu with planning path silhouette moving from start to goal position in RViz scene. Source: Created by the author.

On the other hand, the specific Motion Planning menu has different tabs to setup, control, and configure the robot and the robot scene using MoveIt.

The first tab from the specific menu is the Context tab.

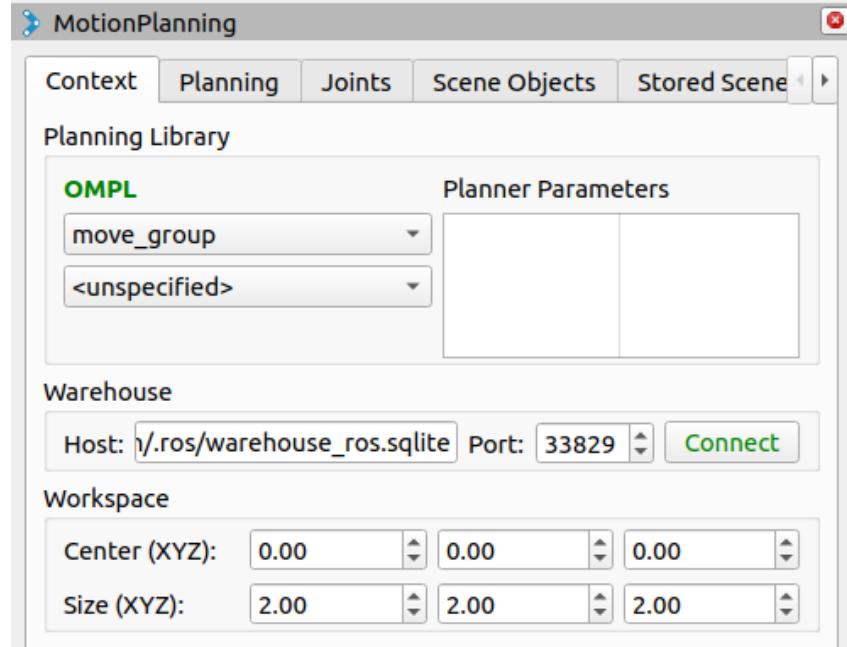


Figure 86. Context tab from specific menu. Source: Created by the author.

This tab allows you to select from different planning algorithms and change the size and center of the workspace where the robot is placed. The default planning algorithm is the OMPL.

The second one is the Planning tab.

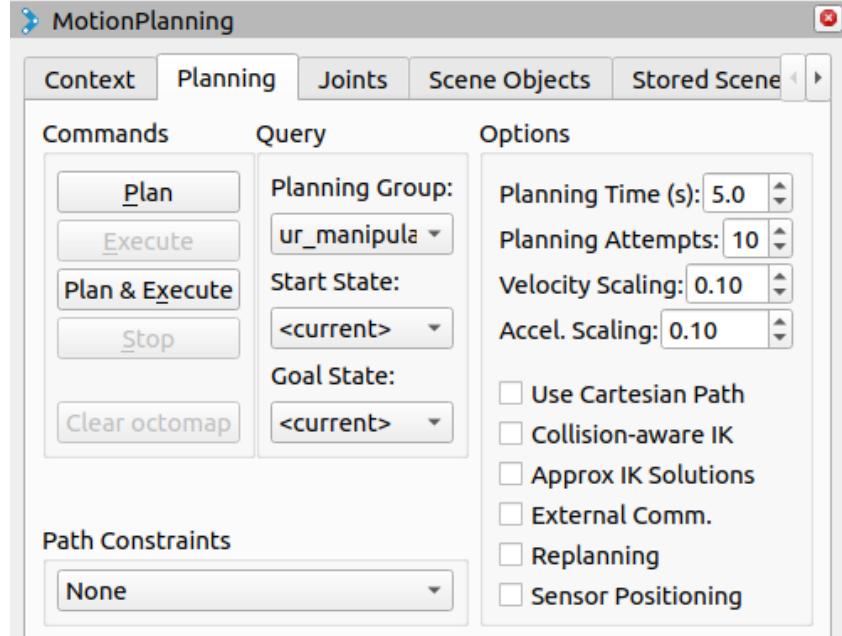


Figure 87. Planning tab from specific menu. Source: Created by the author.

The Planning tab allows the user to plan and execute trajectories with the robot. By moving the orange silhouette to a certain position, a trajectory can be planned and then executed using this tab. It also allows to move to random positions or default positions defined in the SRDF file of the robot (like the home position), by selecting them in the Goal State dropdown. With the Options, users can configure the trajectory by selecting the planning time and attempts, select a

velocity and acceleration scaling value to generate a faster or slower trajectory, and select the following parameters:

- Use cartesian path: the robot will try to move the end effector linearly in cartesian space.
- Collision-aware IK: the solver will attempt to find a collision-free solution for the desired pose. When it is not checked, the solver will allow collisions to happen in the solution. The links in collision will be shown in red even if the option is not selected.
- Approx IK Solutions: it allows to plan and execute trajectories with an approximated solution to the goal.
- External Comm.: it allows to use external commands to control the robot with MoveIt!, for example if you have a robot car, you could use a joystick to move the robot in RViz.
- Replanning: allows to replan if a plan is found to have a problem while the plan is being executed, for example if a new obstacle appears in the path.
- Sensor positioning:

The third tab is Joints.

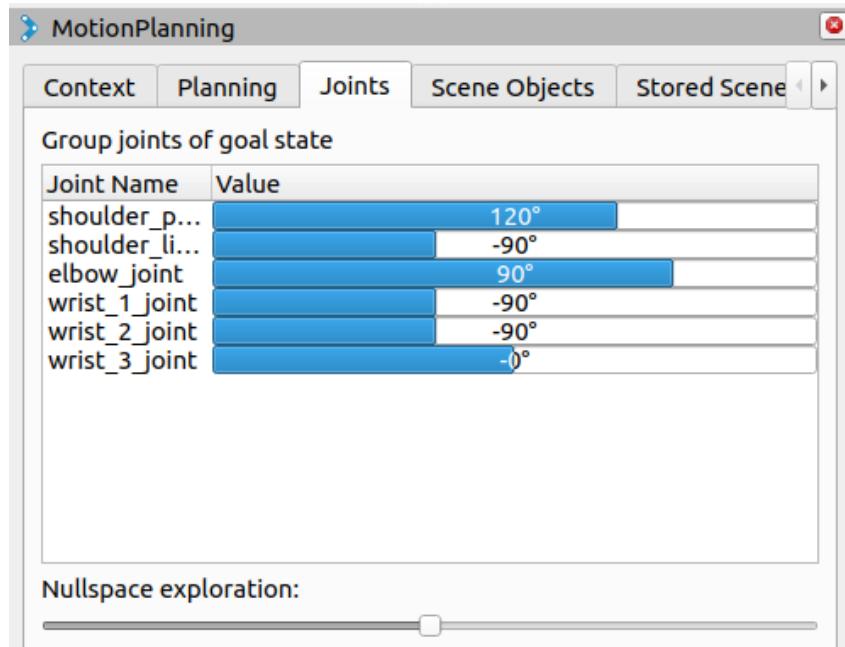


Figure 88. Joint tab from specific menu. Source: Created by the author.

Joints tab is similar to the rqt_joint_trajectory_controller explained in section 11. Controlling UR3. It allows the user to move independently the joints of the goal state (orange silhouette), this is useful if you need the robot to go to an exact position and moving the end effector from the visualization screen is not precise enough.

The fourth tab is Scene Objects.

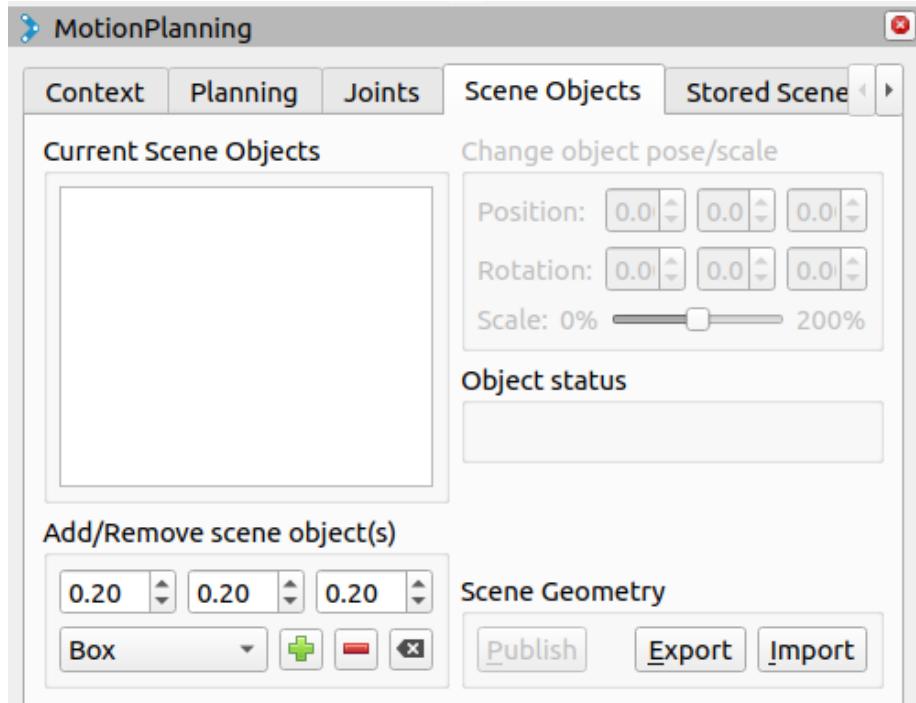


Figure 89. Scene Objects tab from specific menu. Source: Created by the author.

This tab allows you to insert basic objects (boxes, cylinders, or spheres) or CAD file objects to the scene. This is useful to replicate the real robot environment or the simulation environment in RViz to plan trajectories taking them into consideration.

The fifth tab is Stored Scenes.

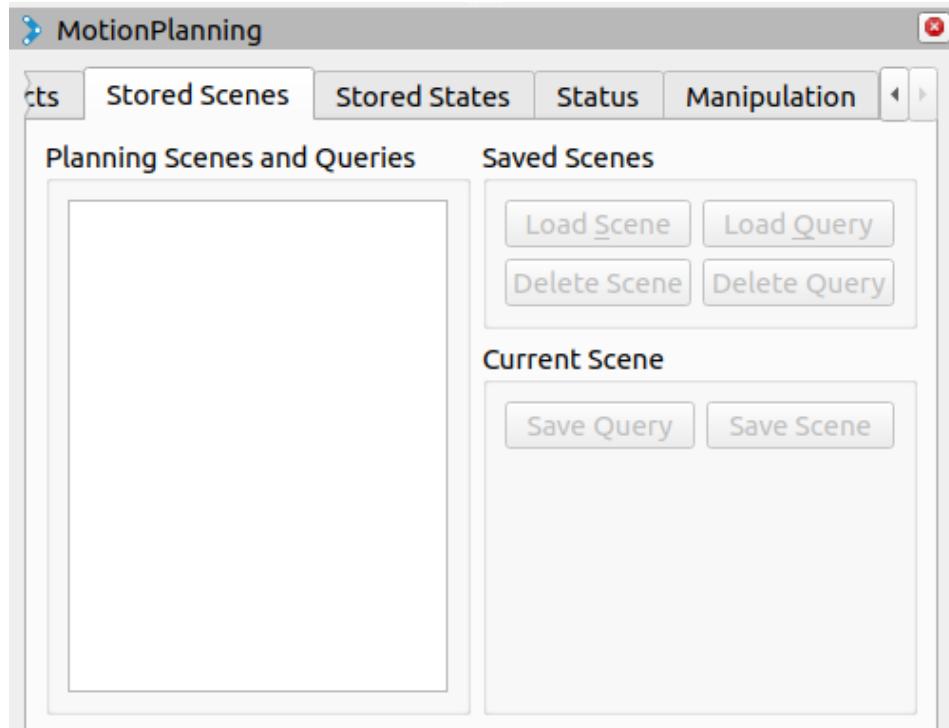


Figure 90. Stored Scene tab from specific menu. Source: Created by the author.

This allows users to save or load scenes to RViz for Movelt control purposes. Sometimes it is faster so save and load a scene than import or create objects each time you run Movelt.

The sixth tab is Stored States.

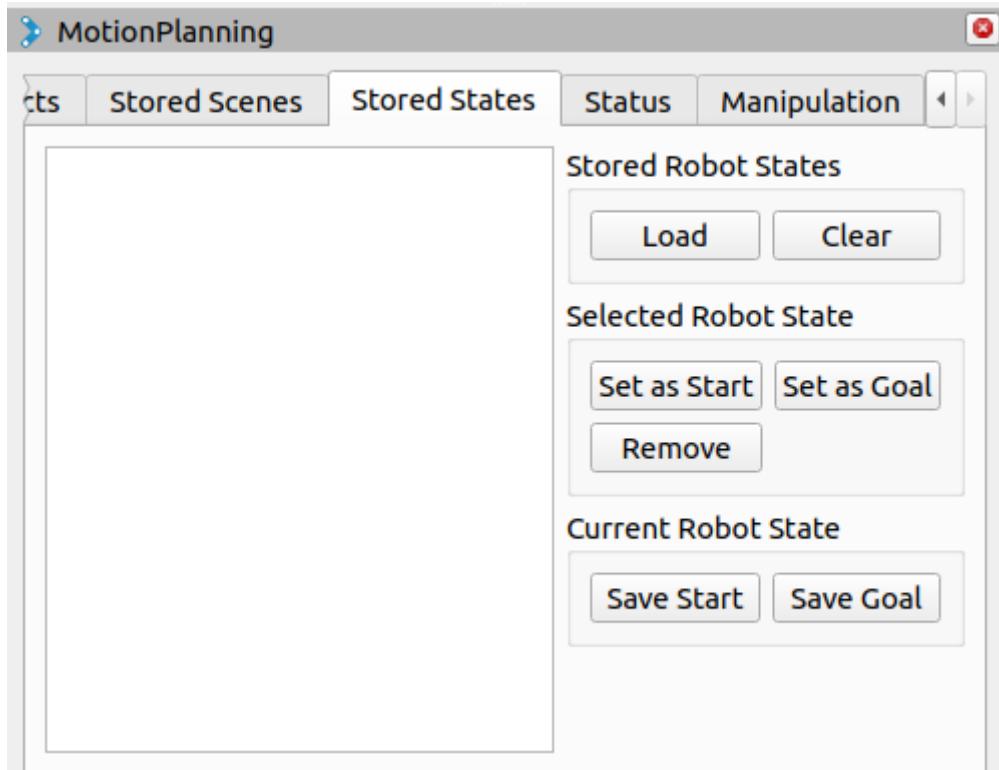


Figure 91. Stored States tab from specific menu. Source: Created by the author.

This allows you to save robot states and load them. It is a useful tool if the application has different repetitive positions that the robot must move to in order to complete a task.

The seventh tab is Status.

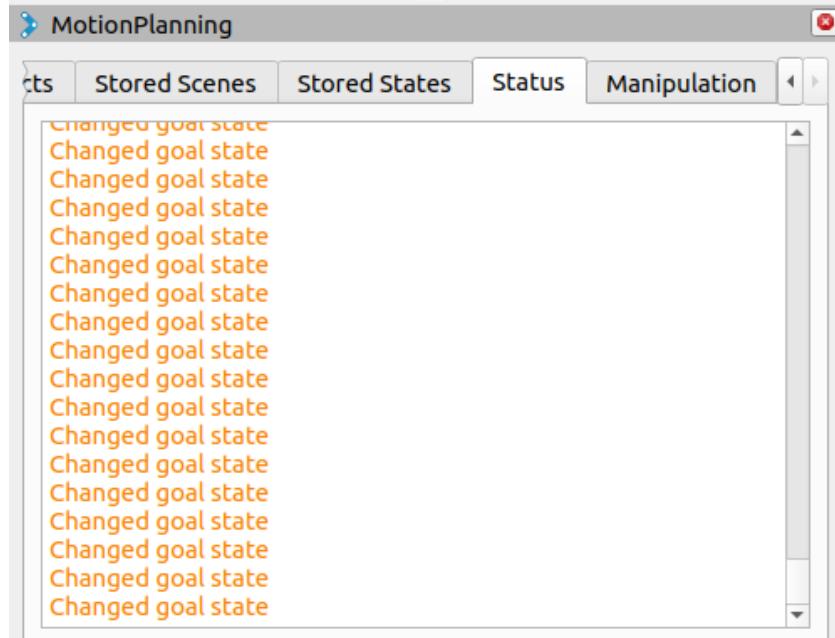


Figure 92. Status tab from specific menu. Source: Created by the author.

This tab shows the constant status of the robot.

The last tab is Manipulation.

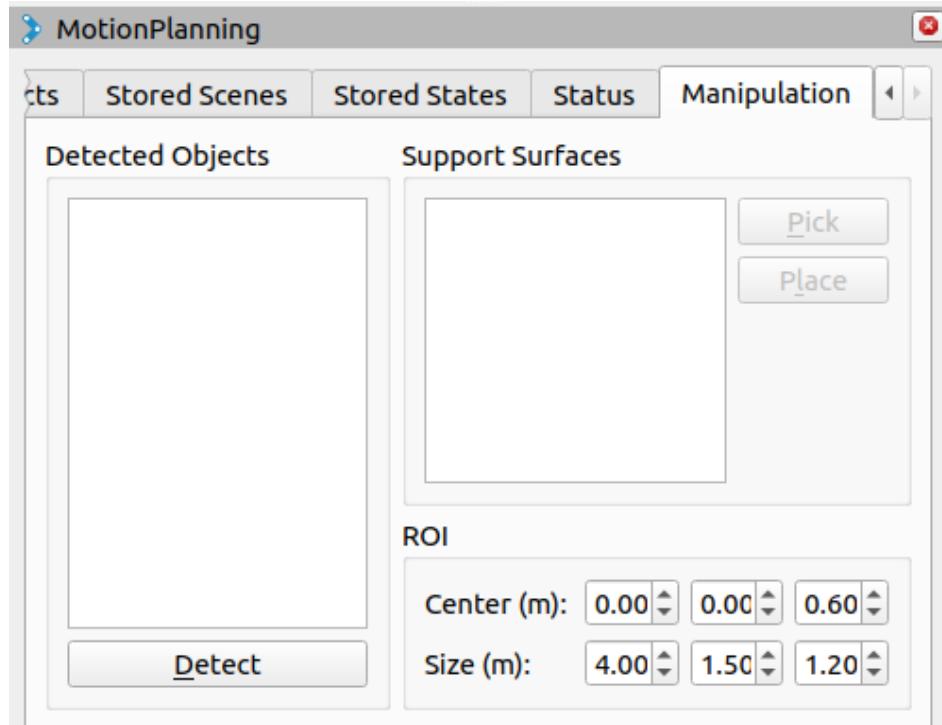


Figure 93. Manipulation tab from specific menu. Source: Created by the author.

This is a tab that can be used to perform manipulation tasks, like a pick&place. It uses the objects detected in the environment either from sensor data or by generating them in the RViz environment.

During this project, the Planning tab was predominantly utilized, as the goal was to control the robot using MoveIt and understand how the trajectory planning and execution worked. It was tested with both simulators (URSim and Gazebo) and the real UR3. The following will demonstrate how to set up and initiate the testing with MoveIt for the three scenarios.

13.3. MoveIt with URSim

To run MoveIt with the URSim, everything is provided in the packages installed in previous sections of the document. To initialize it you will need to start a shell for starting the URSim with the docker, as explained in section 8. Starting Universal Robots Simulation. Remember to build and source the workspace.

Then, in a second shell launch the ur_control.launch.py file, but this time without launching RViz.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver
source install/setup.bash
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.56.101
launch_rviz:=false
```

Lastly, in a third shell launch the following commands:

In Shell #3

```
cd ~/workspace/ros_humble_ur_driver
source install/setup.bash
```

```
ros2 launch ur_moveit_config ur_moveit.launch.py ur_type:=ur3
```

That will start a configured RViz with the MoveIt plugin, allowing you to plan and execute trajectories using the Planning tab from the specific menu.

13.4. MoveIt with Gazebo

UR Gazebo package also has a file to initiate the control framework and RViz with MoveIt plugin. In this case you will only need to do the following:

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver
colcon build
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_moveit.launch.py ur_type:=ur3
```

13.5. MoveIt with real UR3

The process to use MoveIt to control the real UR3 is similar to the one required for the URSim. The only difference is that instead of using a shell to start the simulation, you will need to configure the robot and connect it to your control PC. Then in the first shell start the ur_control and in a second shell launch the same the MoveIt launch file ur_moveit.launch.py, the same way as you did for the URSim.

Take into account that the restrictions from your real UR3 will not apply to RViz. For example, the UR3 from this project has joint restrictions due to the laboratory environment. However, the RViz model can be moved freely through the entire robot workspace. That could cause the real UR3 to reach its limits while moving it. The planning tool is useful in these cases, as you can verify the goal settings before commanding the execution of the trajectory.

13.6. Other MoveIt user interfaces

There are other ways to control a robot with MoveIt that does not require the RViz plugin. Users can create Python and C++ files to configure goals and plan and execute trajectories. By using the moveit_commander and move_group_interface, for Python and C++ respectively, they can program all kinds of task to perform with any robot running with MoveIt. Unfortunately, during this project, due to lack of time and the complexity of the topic, this control method with MoveIt was not achieved. However, there is a course from TheConstruct that gives a notion on how to create and control a UR3e robot using MoveIt, including control through scripts. The name of the course is *ROS2 Manipulation Basics*, and it could be the recommended starting point to understand MoveIt scripts.

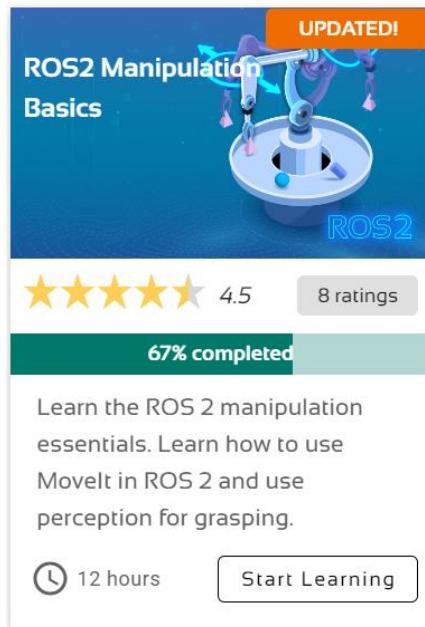


Figure 94. Front page of the ROS2 Manipulation Basics course from The Construct. Source: Created by the author.

14. CONCLUSION

In conclusion, the project successfully achieved its primary objective of controlling a UR3 robotic arm using ROS2 Humble software. The project involved setting up the necessary software and hardware components, learning ROS2 methods and control framework, analyzing UR driver packages and MoveIt2, and validating the control framework using simulation and visualization tools before applying it to the UR3 robot. The project also tested the control framework on a real UR3 and evaluated its performance.

Throughout the project, there were several problems and issues that arose. However, these complications were efficiently solved or adapted to still achieve the main objectives of the project. This was accomplished by conducting in-depth research and experimentation to identify and treat the problems.

The results of the project demonstrated a comprehensive understanding of ROS2 Humble and its applications in robotics control, specifically for the UR3. The control framework was successfully implemented, and joint position control and movement of the robot were achieved. Moreover, the exploration of other control types and the MoveIt tool showed the potential for further development and improvement in robotic manipulation. Overall, this project serves as a valuable contribution to the field of robotics and highlights the potential of ROS2 Humble as a powerful and versatile tool for controlling robotic arms.

15. FUTURE INVESTIGATION PATHS

The project primarily focuses on basic ROS2 control tools and software, but there are numerous additional topics related to control frameworks, simulation tools, and ROS2 robotic applications that can lead to further investigation.

Starting from this project, several paths can be studied. Building upon the analyzed controllers for the UR3, more specific applications can be developed. Incorporating a vision sensor would enable programming the UR3 to perform manipulation tasks. By adding different types of end effectors, the robot could be programmed to automatically move between multiple positions. Additionally, the `rqt_joint_trajectory_controller` could be modified to account for joint limitations of real robots, such as the UR3 in this project.

In terms of MoveIt, a more comprehensive study on using scripts could result in the generation of multiple files for planning and executing trajectories, similar to what was done for the UR3 using the RViz plugin. Furthermore, utilizing Motion Planning in RViz, it becomes possible to design more complex robot scenes, requiring the robot to perform trajectories with obstacles. This could include varying trajectory parameters like, scaling speed and acceleration, to obtain more safe and effective applications.

One last potential future investigation, deriving from the primary motivation behind selecting this project, involves migrating the laboratory sessions of the ECE470 course, "Introduction to Robotics," from ROS1 to ROS2. This transition will be needed as ROS1 will cease to receive updates starting in 2026.

16. BIBLIOGRAPHY

- [1] Clearpath Robotics. *Rviz2*. <https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html>. Last update: unknown.
- [2] Davide Faconti. *Plot Juggler. Fast, intuitive and extensible time series visualization tool*. <https://plotjuggler.io/#one>. Last update: unknown.
- [3] Digital Ocean. *How to fix docker: Got permission denied while trying to connect to the Docker daemon socket*. <https://www.digitalocean.com/community/questions/how-to-fix-docker-got-permission-denied-while-trying-to-connect-to-the-docker-daemon-socket>. Last update: Jan 21, 2020.
- [4] Docker Inc. *Linux post-installation steps for Docker Engine*. <https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-use>. Last update: unknown.
- [5] Docker Inc. *Universal Robots simulator for CB3 series*. https://hub.docker.com/r/universalrobots/ursim_cb3. Last update: Jan 10, 2023
- [6] Faconti, Davide. *PlotJuggler: custom plots, filters and transformations*. <https://slides.com/davidefaconti/plotjuggler-transforms>. Last update: 2021.
- [7] Faconti, Davide. *PlotJuggler: data source*. <https://slides.com/davidefaconti/plotjuggler-data>. Last update: 2021.
- [8] Faconti, Davide. *PlotJuggler: learn the basics*. <https://slides.com/davidefaconti/introduction-to-plotjuggler>. Last update: 2020.
- [9] Fetch Robotics Inc. *Tutorial: Manipulation*. <https://docs.fetchrobotics.com/manipulation.html>. Last update: Apr 22, 2016.
- [10] GitHub. *[humble] can not control robot with MoveIt #21*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation/issues/21. Last update: Mar 24, 2023.
- [11] GitHub. *Absolute file URIs for Gazebo and Ignition simulations break visualization on remote machine #53*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Description/issues/53. Last update: Mar 22, 2023
- [12] GitHub. *Add conditional joint friction to stabilize Gazebo Classic simulation. #55, commits*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Description/pull/55/commits/985ff827b0fe46f279421eef43b2adffb567e29. Last update: Mar 23, 2023.
- [13] GitHub. *Add conditional joint friction to stabilize Gazebo Classic simulation. #55*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Description/pull/55. Last update: Apr 7, 2023.
- [14] GitHub. *Add JTC rrbot demo node #90*. https://github.com/ros-controls/ros2_control_demos/pull/90. Last update: Jul 21, 2021.
- [15] GitHub. *Can't lower the controller manager update rate #66*. https://github.com/ros-controls/gazebo_ros2_control/issues/66. Last update: May 28, 2021.
- [16] GitHub. *Check clock synchronization problem with getCurrentStat() #1399*. <https://github.com/ros-planning/moveit2/issues/1399>. Last update: Jun 28, 2022.
- [17] GitHub. *control_msgs*. https://github.com/ros-controls/control_msgs. Last update: Apr 2, 2023.
- [18] GitHub. *Draft concept for injecting ros2_control hardware system #1*. https://github.com/ipa-cmh/Universal_Robots_ROS2_Description/pull/1. Last update: Apr 17, 2023.

- [19] GitHub. *gazebo_ros2_control*. https://github.com/ros-controls/gazebo_ros2_control. Last update: Feb 7, 2023.
- [20] GitHub. *Generically expose URDF joint dynamics*. #56. https://github.com/UniversalRobots/Universal_Robots_ROS2_Description/pull/56. Last update: Apr 7, 2023.
- [21] GitHub. *Joints unable to maintain static position under PositionJointInterface* #54. https://github.com/ros-controls/gazebo_ros2_control/issues/54. Last update: Mar 21, 2023.
- [22] GitHub. *kinematics_interface*. https://github.com/ros-controls/kinematics_interface. Last update: Feb 14, 2023.
- [23] GitHub. *PlotJuggler*. <https://github.com/facontidavide/PlotJuggler>. Last update: Apr 21, 2023.
- [24] GitHub. *Robot's joints brake on Gazebo* #19. https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation/issues/19. Last update: Apr 7, 2023.
- [25] GitHub. *ROS plugins for PlotJuggler*. https://github.com/PlotJuggler/plotjuggler-ros_plugins. Last update: Jan 14, 2023.
- [26] GitHub. *ros2_control*. https://github.com/ros-controls/ros2_control. Last update: Apr 7, 2023.
- [27] GitHub. *RViz*. <https://github.com/ros2/rviz>. Last update: Apr 28, 2023.
- [28] GitHub. *Universal_Robots_ROS2_Driver*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver. Last update: Apr 12, 2023.
- [29] GitHub. *Universal_Robots_Client_Library*. https://github.com/UniversalRobots/Universal_Robots_Client_Library. Last update: Jan 5, 2023.
- [30] GitHub. *Universal_Robots_ROS2_Description*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Description. Last update: Apr 20, 2023.
- [31] GitHub. *Unstable behaviour of Position Controller in UR10* #73. https://github.com/ros-controls/gazebo_ros2_control/issues/73. Last update: Mar 21, 2023.
- [32] GitHub. *ur_controllers*. https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver/tree/main/ur_controllers. Last update: Mar 16, 2023.
- [33] GitHub. *ur_msgs*. https://github.com/ros-industrial/ur_msgs. Last update: Jan 27, 2023.
- [34] GitHub. *URCaps External Control*. https://github.com/UniversalRobots/Universal_Robots_ExternalControl_UCap. Last update: Dec 13, 2022.
- [35] Josh Newans. *Making a Mobile Robot #13 – Using ros2_control on a real robot*. <https://articulatedrobotics.xyz/mobile-robot-13-ros2-control-real/>. Last update: Oct 14, 2022.
- [36] Lucas Schulze. *PlotJuggler: The Best Time Series Visualization Tool for ROS*. https://www.youtube.com/watch?v=9kFRecDU1bg&ab_channel=LucasSchulze. Last update: Mar 25, 2022.
- [37] Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022.
- [38] Open Robotics. *control_msgs*. http://wiki.ros.org/control_msgs. Last update: Jan 22, 2014.

- [39] Open Robotics. *Gazebo Answers*. *How to install Gazebo for Ubuntu 22.04 with ROS2 Humble.* <https://answers.gazebosim.org/question/28361/how-to-install-gazebo-for-ubuntu-2204-with-ros2-humble/?answer=28427>. Last update: Aug 6, 2022
- [40] Open Robotics. *Gazebo Answers*. *Why does my robot shake / wobble?* <https://answers.gazebosim.org/question/16700/why-does-my-robot-shake-wobble/>. Last update: Jul 12, 2017.
- [41] Open Robotics. *Gazebo*. *Simulate before you build*. <https://gazebosim.org/home>. Last update: unknown.
- [42] Open Robotics. *joint_trajectory_controller*. http://wiki.ros.org/action/fullsearch/joint_trajectory_controller?action=fullsearch&context=180&value=linkto%3A%22joint_trajectory_controller%22. Last update: unknown.
- [43] Open Robotics. *ODE Solvers*. http://wiki.ros.org/physics_ode/ODE. Last update: Feb 25, 2015.
- [44] Open Robotics. *Recording and playing back data*. <https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html>. Last update: May 4, 2023.
- [45] Open Robotics. *ros_control*. http://wiki.ros.org/ros_control. Last update: Jan 9, 2022.
- [46] Open Robotics. *ROS2 Documentation: Humble. Installation/Ubuntu (Debian)*. <https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html>. Last update: Nov 10, 2022.
- [47] Open Robotics. *ROS2 Documentation: Humble. Monitoring for parameter changes (C++)*. <http://docs.ros.org/en/humble/Tutorials/Intermediate/Monitoring-For-Parameter-Changes-CPP.html>. Last update: Oct 17, 2022.
- [48] Open Robotics. *ROS2 Documentation: Humble. Tutorials*. <http://docs.ros.org/en/humble/Tutorials.html>. Last update: Jun 20, 2022.
- [49] Open Robotics. *rviz*. <http://wiki.ros.org/rviz>. Last update: May 16, 2018.
- [50] OpenAI. *Chat GPT*. <https://chat.openai.com/>. Last update: Mar 23, 2023.
- [51] PickNik Robotics. *MoveIt Documentation: Humble. MoveIt 2 Documentation*. <https://moveit.picknik.ai/humble/index.html>. Last update: Jan 20, 2022.
- [52] PickNik Robotics. *MoveIt Humble. Kinematics*. <https://moveit.picknik.ai/humble/doc/concepts/kinematics.html>. Last update: Jan 18, 2022.
- [53] PickNik Robotics. *MoveIt Humble. Motion planning*. https://moveit.picknik.ai/humble/doc/concepts/motion_planning.html. Last update: Jan 18, 2022.
- [54] PickNik Robotics. *MoveIt Humble. Planning scene monitor*. https://moveit.picknik.ai/humble/doc/concepts/planning_scene_monitor.html. Last update: Jan 18, 2022.
- [55] PickNik Robotics. *MoveIt Humble. The move_group node*. https://moveit.picknik.ai/humble/doc/concepts/move_group.html. Last update: Feb 17, 2022.
- [56] PickNik Robotics. *MoveIt Quickstart in RViz*. https://ros-planning.github.io/moveit_tutorials/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html. Last update: Sep 4, 2021.
- [57] Qiu zhe. *How to control Universal Robot by using ROS2*. https://www.ritsumei.ac.jp/~kawamura/doc/ros2_ur.pdf. Last update: Jun 6, 2021.

- [58] Reddit. *Robot Shaking with Gazebo + ros-control*. https://www.reddit.com/r/ROS/comments/uxtdi/robot_shaking_with_gazebo_roscontrol/. Last update: May 30, 2022.
- [59] Robotics Back-End. *Create and Set Up a ROS2 Workspace – ROS2 Tutorial 3*. <https://www.youtube.com/watch?v=3GbrKQ7G2P0>. Last update: Aug 17, 2022.
- [60] Robotics Back-End. *Intro: Install and Setup ROS2 Humble – ROS2 Tutorial 1*. <https://www.youtube.com/watch?v=0aPbWsyENA8>. Last update: Aug 15, 2022.
- [61] ROS Industrial. *Motion Planning using RViz*. https://industrial-training-master.readthedocs.io/en/melodic/_source/session3/Motion-Planning-RVIZ.html. Last update: Jun 5, 2019.
- [62] ros2_control Development Team. *ros2_controllers. Command Line Interface*. https://control.ros.org/master/doc/ros2_control/ros2controlcli/doc/userdoc.html. Last update: unknown.
- [63] ros2_control Development Team. *ros2_controllers. joint_trajectory_controller*. https://control.ros.org/foxy/doc/ros2_controllers/joint_trajectory_controller/doc/userdoc.html. Last update: unknown.
- [64] ROS2_control Maintainers. *ROS2_Control: Rolling. Getting Started*. https://control.ros.org/master/doc/getting_started/getting_started.html. Last update: Apr 10, 2023.
- [65] ROS2_control Maintainers. *Welcome to the ros2_control documentation!*. <https://control.ros.org/master/index.html>. Last update: Apr 12, 2023.
- [66] The Construct. *Develop the robots of the future*. <https://www.theconstructsim.com/>. Last update: unknown.
- [67] Universal Robots A/S. *ur_robot_driver. Setting up Ubuntu with a PREEMPT_RT kernel*. https://docs.ros.org/en/ros2_packages/rolling/api/ur_robot_driver/installation/real_time.html. Last update: unknown.
- [68] Universal Robots. *Dashboard server e-series, port 29999*. <https://www.universal-robots.com/articles/ur/dashboard-server-e-series-port-29999/>. Last update: Dec 15, 2022.
- [69] Universal Robots. *Do more than just keep up. Try our virtual simulator*. <https://www.universal-robots.com/products/polyscope/#:~:text=PolyScope%20connects%20operators%20to%20robots,experience%20to%20automate%20your%20processes>. Last update: unknown.
- [70] Universal Robots. *Move UR with a MoveIt2 Script*. <https://forum.universal-robots.com/t/move-ur-with-a-moveit2-script/24497>. Last update: Oct 20, 2022.
- [71] Universal Robots. *The UR3e*. <https://www.universal-robots.com/products/ur3-robot/>. Last update: unknown.
- [72] Universal Robots. *UR3 3D Model*. <https://www.universal-robots.com/3d/ur3.html>. Last update: unknown.
- [73] Universal Robots. *URCap – Basics*. <https://www.universal-robots.com/articles/ur/urplus-resources/urcap-basics/>. Last update: unknown.
- [74] WiredWorkers. *Meet Universal Robots UR3*. <https://www.wiredworkers.io/cobot/brands/universal-robots/ur3/>. Last update: unknown.

APPENDIX 1. COMMENTED CODE

ur_control.launch.py

Import the needed functions for the launch file.

```
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, OpaqueFunction
from launch.conditions import IfCondition, UnlessCondition
from launch.substitutions import Command, FindExecutable, LaunchConfiguration,
PathJoinSubstitution
```

Assign the launch configuration of the arguments to variables to simplify the code.

```
def launch_setup(context, *args, **kwargs):

    # Initialize Arguments
    ur_type = LaunchConfiguration("ur_type")
    robot_ip = LaunchConfiguration("robot_ip")
    safety_limits = LaunchConfiguration("safety_limits")
    safety_pos_margin = LaunchConfiguration("safety_pos_margin")
    safety_k_position = LaunchConfiguration("safety_k_position")
    # General arguments
    runtime_config_package = LaunchConfiguration("runtime_config_package")
    controllers_file = LaunchConfiguration("controllers_file")
    description_package = LaunchConfiguration("description_package")
    description_file = LaunchConfiguration("description_file")
    tf_prefix = LaunchConfiguration("tf_prefix")
    use_fake_hardware = LaunchConfiguration("use_fake_hardware")
    fake_sensor_commands = LaunchConfiguration("fake_sensor_commands")
    controller_spawner_timeout = LaunchConfiguration("controller_spawner_timeout")
    initial_joint_controller = LaunchConfiguration("initial_joint_controller")
    activate_joint_controller = LaunchConfiguration("activate_joint_controller")
    launch_rviz = LaunchConfiguration("launch_rviz")
    headless_mode = LaunchConfiguration("headless_mode")
    launch_dashboard_client = LaunchConfiguration("launch_dashboard_client")
    use_tool_communication = LaunchConfiguration("use_tool_communication")
    tool_parity = LaunchConfiguration("tool_parity")
    tool_baud_rate = LaunchConfiguration("tool_baud_rate")
    tool_stop_bits = LaunchConfiguration("tool_stop_bits")
    tool_rx_idle_chars = LaunchConfiguration("tool_rx_idle_chars")
    tool_tx_idle_chars = LaunchConfiguration("tool_tx_idle_chars")
    tool_device_name = LaunchConfiguration("tool_device_name")
    tool_tcp_port = LaunchConfiguration("tool_tcp_port")
    tool_voltage = LaunchConfiguration("tool_voltage")
    reverse_ip = LaunchConfiguration("reverse_ip")
    script_command_port = LaunchConfiguration("script_command_port")
```

Obtain the file paths for the robot description command. Depending on the UR manipulator selected, the configuration files that will be passed as arguments to the XACRO file will be different.

```
    joint_limit_params = PathJoinSubstitution(
        [FindPackageShare(description_package), "config", ur_type, "joint_limits.yaml"]
    )
    kinematics_params = PathJoinSubstitution(
        [FindPackageShare(description_package), "config", ur_type,
"default_kinematics.yaml"]
    )
    physical_params = PathJoinSubstitution(
```

```

        [FindPackageShare(description_package), "config", ur_type,
"physical_parameters.yaml"]
    )
    visual_params = PathJoinSubstitution(
        [FindPackageShare(description_package), "config", ur_type,
"visual_parameters.yaml"]
    )
    script_filename = PathJoinSubstitution(
        [FindPackageShare("ur_client_library"), "resources", "external_control.urscript"]
    )
    input_recipe_filename = PathJoinSubstitution(
        [FindPackageShare("ur_robot_driver"), "resources", "rtde_input_recipe.txt"]
    )
    output_recipe_filename = PathJoinSubstitution(
        [FindPackageShare("ur_robot_driver"), "resources", "rtde_output_recipe.txt"]
    )
)

```

Create the robot description command with the needed arguments to be used as input to the XACRO file. All the required arguments can be found listed at the Universal_Robots_ROS2_Description/urdf/ur.urdf.xacro file. This command will generate the URDF file to describe the robot.

```

robot_description_content = Command(
[
    PathJoinSubstitution([FindExecutable(name="xacro")]),
    " ",
    PathJoinSubstitution([FindPackageShare(description_package), "urdf",
description_file]),
    " ",
    "robot_ip:=",
    robot_ip,
    " ",
    "joint_limit_params:=",
    joint_limit_params,
    " ",
    "kinematics_params:=",
    kinematics_params,
    " ",
    "physical_params:=",
    physical_params,
    " ",
    "visual_params:=",
    visual_params,
    " ",
    "safety_limits:=",
    safety_limits,
    " ",
    "safety_pos_margin:=",
    safety_pos_margin,
    " ",
    "safety_k_position:=",
    safety_k_position,
    " ",
    "name:=",
    ur_type,
    " ",
    "script_filename:=",
    script_filename,
    " ",
    "input_recipe_filename:=",
    input_recipe_filename,
    " ",
    "output_recipe_filename:=",
    output_recipe_filename,
    " ",
    "tf_prefix:=",
    tf_prefix,
)

```

```

    " ",
    "use_fake_hardware:=",
    use_fake_hardware,
    " ",
    "fake_sensor_commands:=",
    fake_sensor_commands,
    " ",
    "headless_mode:=",
    headless_mode,
    " ",
    "use_tool_communication:=",
    use_tool_communication,
    " ",
    "tool_parity:=",
    tool_parity,
    " ",
    "tool_baud_rate:=",
    tool_baud_rate,
    " ",
    "tool_stop_bits:=",
    tool_stop_bits,
    " ",
    "tool_rx_idle_chars:=",
    tool_rx_idle_chars,
    " ",
    "tool_tx_idle_chars:=",
    tool_tx_idle_chars,
    " ",
    "tool_device_name:=",
    tool_device_name,
    " ",
    "tool_tcp_port:=",
    tool_tcp_port,
    " ",
    "tool_voltage:=",
    tool_voltage,
    " ",
    "reverse_ip:=",
    reverse_ip,
    " ",
    "script_command_port:=",
    script_command_port,
    " ",
)
]
)

```

Create, from the description command, the robot description that will be used to define the launched nodes in the file.

```
robot_description = {"robot_description": robot_description_content}
```

Define as variables the path of the files that contain a list of initial joint controllers for the robot, configuration for the RViz tool to display the robot and the desired update rate for the robot's control loop.

```

initial_joint_controllers = PathJoinSubstitution(
    [FindPackageShare(runtime_config_package), "config", controllers_file]
)

rviz_config_file = PathJoinSubstitution(
    [FindPackageShare(description_package), "rviz", "view_robot.rviz"]
)

# define update rate
update_rate_config_file = PathJoinSubstitution(
    [

```

```

        FindPackageShare(runtime_config_package),
        "config",
        ur_type.perform(context) + "_update_rate.yaml",
    ]
)

```

Define the first nodes that will be launched. These use the parameters previously implemented with the robot description, the update rate, the initial controllers, etc. The defined nodes are control_node, ur_control_node, dashboard_client_node, tool_communication_node, controller_stopper_node, robot_state_publisher_node and rviz_node. The general functionality of these nodes is explained in section 7.1 Universal_Robots_ROS2_Driver.

```

control_node = Node(
    package="controller_manager",
    executable="ros2_control_node",
    parameters=[robot_description, update_rate_config_file,
initial_joint_controllers],
    output="screen",
    condition=IfCondition(use_fake_hardware),
)

ur_control_node = Node(
    package="ur_robot_driver",
    executable="ur_ros2_control_node",
    parameters=[robot_description, update_rate_config_file,
initial_joint_controllers],
    output="screen",
    condition=UnlessCondition(use_fake_hardware),
)

dashboard_client_node = Node(
    package="ur_robot_driver",
    condition=IfCondition(launch_dashboard_client) and
UnlessCondition(use_fake_hardware),
    executable="dashboard_client",
    name="dashboard_client",
    output="screen",
    emulate_tty=True,
    parameters=[{"robot_ip": robot_ip}],
)

tool_communication_node = Node(
    package="ur_robot_driver",
    condition=IfCondition(use_tool_communication),
    executable="tool_communication.py",
    name="ur_tool_comm",
    output="screen",
    parameters=[
        {
            "robot_ip": robot_ip,
            "tcp_port": tool_tcp_port,
            "device_name": tool_device_name,
        }
    ],
)

controller_stopper_node = Node(
    package="ur_robot_driver",
    executable="controller_stopper_node",
    name="controller_stopper",
    output="screen",
    emulate_tty=True,
    condition=UnlessCondition(use_fake_hardware),
    parameters=[

]
)

```

```

        {"headless_mode": headless_mode},
        {"joint_controller_active": activate_joint_controller},
        {
            "consistent_controllers": [
                "io_and_status_controller",
                "force_torque_sensor_broadcaster",
                "joint_state_broadcaster",
                "speed_scaling_state_broadcaster",
            ],
        },
    ],
)
robot_state_publisher_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_description],
)
rviz_node = Node(
    package="rviz2",
    condition=IfCondition(launch_rviz),
    executable="rviz2",
    name="rviz2",
    output="log",
    arguments=["-d", rviz_config_file],
)

```

Spawn the controllers that will be used to control the robot and provide feedback of its state. These controllers will be started or stopped using the nodes initial_joint_controller_started and initial_joint_controller_stopped.

```

# Spawn controllers
def controller_spawner(name, active=True):
    inactive_flags = ["--inactive"] if not active else []
    return Node(
        package="controller_manager",
        executable="spawner",
        arguments=[
            name,
            "--controller-manager",
            "/controller_manager",
            "--controller-manager-timeout",
            controller_spawner_timeout,
        ]
        + inactive_flags,
    )

controller_spawner_names = [
    "joint_state_broadcaster",
    "io_and_status_controller",
    "speed_scaling_state_broadcaster",
    "force_torque_sensor_broadcaster",
]
controller_spawner_inactive_names = ["forward_position_controller"]

controller_spawners = [controller_spawner(name) for name in controller_spawner_names]
+ [
    controller_spawner(name, active=False) for name in
controller_spawner_inactive_names
]

# There may be other controllers of the joints, but this is the initially-started one
initial_joint_controller_spawner_started = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[


```

```

        initial_joint_controller,
        "-c",
        "/controller_manager",
        "--controller-manager-timeout",
        controller_spawner_timeout,
    ],
    condition=IfCondition(activate_joint_controller),
)
initial_joint_controller_spawner_stopped = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[
        initial_joint_controller,
        "-c",
        "/controller_manager",
        "--controller-manager-timeout",
        controller_spawner_timeout,
        "--inactive",
    ],
    condition=UnlessCondition(activate_joint_controller),
)

```

Return all the previously defined nodes to initiate them in the Launch Description.

```

nodes_to_start = [
    control_node,
    ur_control_node,
    dashboard_client_node,
    tool_communication_node,
    controller_stopper_node,
    robot_state_publisher_node,
    rviz_node,
    initial_joint_controller_spawner_stopped,
    initial_joint_controller_spawner_started,
] + controller_spawners

return nodes_to_start

```

Declare all the arguments that will be used to define the nodes and robot description previously commented. All the arguments have name, description and default_value (except for the type and IP that must be provided by the user when launching) listed.

```

def generate_launch_description():
    declared_arguments = []
    # UR specific arguments
    declared_arguments.append(
        DeclareLaunchArgument(
            "ur_type",
            description="Type/series of used UR robot.",
            choices=["ur3", "ur3e", "ur5", "ur5e", "ur10", "ur10e", "ur16e"],
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "robot_ip", description="IP address by which the robot can be reached."
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_limits",
            default_value="true",
            description="Enables the safety limits controller if true."
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_pos_margin",

```

```

        default_value="0.15",
        description="The margin to lower and upper limits in the safety controller.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "safety_k_position",
        default_value="20",
        description="k-position factor in the safety controller.",
    )
)
# General arguments
declared_arguments.append(
    DeclareLaunchArgument(
        "runtime_config_package",
        default_value="ur_robot_driver",
        description='Package with the controller\'s configuration in "config" folder.
\\
        Usually the argument is not set, it enables use of a custom setup.',
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "controllers_file",
        default_value="ur_controllers.yaml",
        description="YAML file with the controllers configuration.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "description_package",
        default_value="ur_description",
        description="Description package with robot URDF/XACRO files. Usually the
argument \
        is not set, it enables use of a custom description.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "description_file",
        default_value="ur.urdf.xacro",
        description="URDF/XACRO description file with the robot.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tf_prefix",
        default_value="",
        description="tf_prefix of the joint names, useful for \
multi-robot setup. If changed, also joint names in the controllers' configuration
\\
        have to be updated.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "use_fake_hardware",
        default_value="false",
        description="Start robot with fake hardware mirroring command to its states.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "fake_sensor_commands",
        default_value="false",
        description="Enable fake command interfaces for sensors used for simple
simulations. \
        Used only if 'use_fake_hardware' parameter is true.",
    )
)

```

```

declared_arguments.append(
    DeclareLaunchArgument(
        "headless_mode",
        default_value="false",
        description="Enable headless mode for robot control",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "controller_spawner_timeout",
        default_value="10",
        description="Timeout used when spawning controllers.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "initial_joint_controller",
        default_value="scaled_joint_trajectory_controller",
        description="Initially loaded robot controller.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "activate_joint_controller",
        default_value="true",
        description="Activate loaded joint controller.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument("launch_rviz", default_value="true", description="Launch
RViz?")
)
declared_arguments.append(
    DeclareLaunchArgument(
        "launch_dashboard_client", default_value="true", description="Launch Dashboard
Client?"
)
)
declared_arguments.append(
    DeclareLaunchArgument(
        "use_tool_communication",
        default_value="false",
        description="Only available for e series!",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_parity",
        default_value="0",
        description="Parity configuration for serial communication. Only effective, if
\
        use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_baud_rate",
        default_value="115200",
        description="Baud rate configuration for serial communication. Only effective,
if \
        use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_stop_bits",
        default_value="1",
        description="Stop bits configuration for serial communication. Only effective,
if \
        use_tool_communication is set to True.",
    )
)

```

```

        )
    )
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_rx_idle_chars",
        default_value="1.5",
        description="RX idle chars configuration for serial communication. Only
effective, \
            if use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_tx_idle_chars",
        default_value="3.5",
        description="TX idle chars configuration for serial communication. Only
effective, \
            if use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_device_name",
        default_value="/tmp/ttyUR",
        description="File descriptor that will be generated for the tool communication
device. \
            The user has be be allowed to write to this location. \
            Only effective, if use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_tcp_port",
        default_value="54321",
        description="Remote port that will be used for bridging the tool's serial
device. \
            Only effective, if use_tool_communication is set to True.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "tool_voltage",
        default_value="0", # 0 being a conservative value that won't destroy anything
        description="Tool voltage that will be setup.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "reverse_ip",
        default_value="0.0.0.0",
        description="IP that will be used for the robot controller to communicate back
to the driver.",
    )
)
declared_arguments.append(
    DeclareLaunchArgument(
        "script_command_port",
        default_value="50004",
        description="Port that will be opened to forward script commands from the
driver to the robot",
    )
)
)

```

Return the Launch Description of the arguments and the launch setup that returns all the nodes to be executed.

```
return LaunchDescription(declared_arguments + [OpaqueFunction(function=launch_setup)])
```

ur_controllers.yaml

Define the controllers that the control manager will use. The name could be anyone, as long as the type is defined correctly. In the case of the UR manipulators, the controllers needed belong to the ros2_controllers package, except for the scaled_joint_trajectory_controller which is part of the ur_controllers package. Some of these controllers broadcast information from the robot, while the rest can be used as independent controllers.

```
controller_manager:  
  ros_parameters:  
    joint_state_broadcaster:  
      type: joint_state_broadcaster/JointStateBroadcaster  
  
    io_and_status_controller:  
      type: ur_controllers/GPIOController  
  
    speed_scaling_state_broadcaster:  
      type: ur_controllers/SpeedScalingStateBroadcaster  
  
    force_torque_sensor_broadcaster:  
      type: force_torque_sensor_broadcaster/ForceTorqueSensorBroadcaster  
  
    joint_trajectory_controller:  
      type: joint_trajectory_controller/JointTrajectoryController  
  
    scaled_joint_trajectory_controller:  
      type: ur_controllers/ScaledJointTrajectoryController  
  
    forward_velocity_controller:  
      type: velocity_controllers/JointGroupVelocityController  
  
    forward_position_controller:  
      type: position_controllers/JointGroupPositionController
```

Define the parameters of the speed_scaling_state_broadcaster. This broadcaster provides the speed scaling from the joints, and it is only used with the scaled_joint_trajectory_controller.

```
speed_scaling_state_broadcaster:  
  ros_parameters:  
    state_publish_rate: 100.0
```

Define the parameters of the force_torque_sensor_broadcaster. This broadcaster sends data from the torque sensor at the tool frame (end effector) of the UR manipulators. Only used for specific tool applications where controlling the force is needed.

```
force_torque_sensor_broadcaster:  
  ros_parameters:  
    sensor_name: tcp_fts_sensor  
    state_interface_names:  
      - force.x  
      - force.y  
      - force.z  
      - torque.x  
      - torque.y  
      - torque.z
```

```

frame_id: tool0
topic_name: ft_data

```

Defining the parameters of the joint_trajectory_controller. Controller that creates a trajectory given a goal for the robot joints or a goal position for the end effector. The name of the joints needs to be the same as the ones defined in the description file of the robot. The command interfaces are the type of data that the control can send to the robot, and the state interface the data that the controller need to receive. The constraints added could be implemented either for safety or to improve the performance of the robot during the trajectory. In this case the stopped_velocity_tolerance marks the speed that indicates that controlled system is stopped at the end of the trajectory. The goal_time is the time tolerance for achieving trajectory goal before or after commanded time. And the trajectory and goal constraints mark the offset tolerance during and at the end of the trajectory.

```

joint_trajectory_controller:
  ros_parameters:
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint
    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity
  state_publish_rate: 100.0
  action_monitor_rate: 20.0
  allow_partial_joints_goal: false
  constraints:
    stopped_velocity_tolerance: 0.2
    goal_time: 0.0
    shoulder_pan_joint: { trajectory: 0.2, goal: 0.1 }
    shoulder_lift_joint: { trajectory: 0.2, goal: 0.1 }
    elbow_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_1_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_2_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_3_joint: { trajectory: 0.2, goal: 0.1 }

```

Defining the parameters of the scaled_joint_trajectory_controller. This is the UR improved version of the previous controller. It present the same parameters, but this controller uses the data from the speed scaling values at the joints to generate better trajectories that prevents some external factors that might cause a path deviation, like emergency stops or speed limitations at the joints for safety reasons.

```

scaled_joint_trajectory_controller:
  ros_parameters:
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint
    command_interfaces:
      - position
    state_interfaces:
      - position

```

```

    - velocity
    state_publish_rate: 100.0
action_monitor_rate: 20.0
allow_partial_joints_goal: false
constraints:
    stopped_velocity_tolerance: 0.2
    goal_time: 0.0
    shoulder_pan_joint: { trajectory: 0.2, goal: 0.1 }
    shoulder_lift_joint: { trajectory: 0.2, goal: 0.1 }
    elbow_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_1_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_2_joint: { trajectory: 0.2, goal: 0.1 }
    wrist_3_joint: { trajectory: 0.2, goal: 0.1 }

```

Defining the parameters of the forward_velocity_controller and forward_position_controller. These two controllers do not perform any calculations, they just send velocity and position commands to the respective joints.

```

forward_velocity_controller:
ros_parameters:
joints:
- shoulder_pan_joint
- shoulder_lift_joint
- elbow_joint
- wrist_1_joint
- wrist_2_joint
- wrist_3_joint
interface_name: velocity

forward_position_controller:
ros_parameters:
joints:
- shoulder_pan_joint
- shoulder_lift_joint
- elbow_joint
- wrist_1_joint
- wrist_2_joint
- wrist_3_joint

```

initial_positions.yaml

This file allows the user to set a certain initial position for the robot during simulations with programs like Gazebo. The file is implemented as a property in the urdf/ur.urdf.xacro file of the description package, which will be also commented in this Appendix 1. This initial position needs to be considered in other files too, specifically in files where the initial position is set as a security or needed condition, for example the test_scaled_joint_trajectory_controller.launch.py. This file will be commented in this Appendix 1.

```

shoulder_pan_joint: 2.09
shoulder_lift_joint: -1.57
elbow_joint: 1.57
wrist_1_joint: -1.57
wrist_2_joint: -1.57
wrist_3_joint: 0.0

```

ur_macro.xacro

First the code starts the XML file, defines a robot object, and sets a commented explanation about what the file does and who developed it.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro">
  <!--
    Base UR robot series xacro macro.
    NOTE this is NOT a URDF. It cannot directly be loaded by consumers
    expecting a flattened '.urdf' file. See the top-level '.xacro' for that
    (but note that .xacro must still be processed by the xacro command).
    This file models the base kinematic chain of a UR robot, which then gets
    parameterised by various configuration files to convert it into a UR3(e),
    UR5(e), UR10(e) or UR16e.
    NOTE the default kinematic parameters (i.e., link lengths, frame locations,
    offsets, etc) do not correspond to any particular robot. They are defaults
    only. There WILL be non-zero offsets between the Forward Kinematics results
    in TF (i.e., robot_state_publisher) and the values reported by the Teach
    Pendant.
    For accurate (and robot-specific) transforms, the 'kinematics_parameters_file'
    parameter MUST point to a .yaml file containing the appropriate values for
    the targeted robot.
    If using the UniversalRobots/Universal_Robots_ROS_Driver, follow the steps
    described in the readme of that repository to extract the kinematic
    calibration from the controller and generate the required .yaml file.
    Main author of the migration to yaml configs Ludovic Delval.
    Contributors to previous versions (in no particular order)
    - Denis Stogl
    - Lovro Ivanov
    - Felix Messmer
    - Kelsey Hawkins
    - Wim Meeussen
    - Shaun Edwards
    - Nadia Hammoudeh Garcia
    - Dave Hershberger
    - G. vd. Hoorn
    - Philip Long
    - Dave Coleman
    - Miguel Prada
    - Mathias Luedtke
    - Marcel Schnirring
    - Felix von Drigalski
    - Felix Exner
    - Jimmy Da Silva
    - Ajit Krissna N L
    - Muhammad Asif Rana
  -->
```

Then, it includes the XACRO files needed to define the robot properties and parameters, which are defined afterwards in the macro object called ur_robot. Some of the generic parameters have values assigned, and others are obtained from files assigned in the `<!-- Load configuration data from the provided .yaml files -->` section. The ur.ros2_control.xacro is also included to define all the parameters related to the control framework.

```
<xacro:include filename="$(find ur_description)/urdf/inc/ur_transmissions.xacro" />
<xacro:include filename="$(find ur_description)/urdf/inc/ur_common.xacro" />
<xacro:macro name="ur_robot" params="
  name
  prefix
  parent
  *origin
  joint_limits_parameters_file
  kinematics_parameters_file
```

```

physical_parameters_file
visual_parameters_file
transmission_hw_interface:=hardware_interface/PositionJointInterface
safety_limits:=false
safety_pos_margin:=0.15
safety_k_position:=20
use_fake.hardware:=false
fake_sensor_commands:=false
sim_gazebo:=false
sim_ignition:=false
headless_mode:=false
initial_positions:=${dict(shoulder_pan_joint=0.0,shoulder_lift_joint=-1.57,elbow_joint=0.0,wrist_1_joint=-1.57,wrist_2_joint=0.0,wrist_3_joint=0.0)}
use_tool_communication:=false
tool_voltage:=0
tool_parity:=0
tool_baud_rate:=115200
tool_stop_bits:=1
tool_rx_idle_chars:=1.5
tool_tx_idle_chars:=3.5
tool_device_name:=/tmp/ttyUR
tool_tcp_port:=54321
robot_ip:=0.0.0.0
script_filename:=to_be_filled_by_ur_robot_driver
output_recipe_filename:=to_be_filled_by_ur_robot_driver
input_recipe_filename:=to_be_filled_by_ur_robot_driver
reverse_port:=50001
script_sender_port:=50002
reverse_ip:=0.0.0.0
script_command_port:=50004">
<!-- Load configuration data from the provided .yaml files -->
<xacro:read_model_data
    joint_limits_parameters_file="${joint_limits_parameters_file}"
    kinematics_parameters_file="${kinematics_parameters_file}"
    physical_parameters_file="${physical_parameters_file}"
    visual_parameters_file="${visual_parameters_file}"
    force_abs_paths="${sim_gazebo or sim_ignition}"/>
<!-- ros2 control include -->
<xacro:include filename="$(find ur_description)/urdf/ur.ros2_control.xacro" />
<!-- ros2 control instance -->
<xacro:ur_ros2_control
    name="${name}" prefix="${prefix}"
    use_fake.hardware="${use_fake.hardware}"
    initial_positions="${initial_positions}"
    fake_sensor_commands="${fake_sensor_commands}"
    headless_mode="${headless_mode}"
    sim_gazebo="${sim_gazebo}"
    sim_ignition="${sim_ignition}"
    script_filename="${script_filename}"
    output_recipe_filename="${output_recipe_filename}"
    input_recipe_filename="${input_recipe_filename}"
    tf_prefix=""
    hash_kinematics="${kinematics_hash}"
    robot_ip="${robot_ip}"
    use_tool_communication="${use_tool_communication}"
    tool_voltage="${tool_voltage}"
    tool_parity="${tool_parity}"
    tool_baud_rate="${tool_baud_rate}"
    tool_stop_bits="${tool_stop_bits}"
    tool_rx_idle_chars="${tool_rx_idle_chars}"
    tool_tx_idle_chars="${tool_tx_idle_chars}"
    tool_device_name="${tool_device_name}"
    tool_tcp_port="${tool_tcp_port}"
    reverse_port="${reverse_port}"
    script_sender_port="${script_sender_port}"
    reverse_ip="${reverse_ip}"
    script_command_port="${script_command_port}"
/>
```

Then, add the URDF links of the general UR manipulator, dependent of parameters that will be assigned to the specific values in the ur.urdf.xacro file. This code includes the visual properties and physical properties of the links.

```
<!-- Add URDF transmission elements (for ros_control) -->
<!--<xacro:ur_arm_transmission prefix="${prefix}" hw_interface="${transmission_hw_interface}" />-->
<!-- Placeholder for ros2_control transmission which don't yet exist -->
<!-- links - main serial chain -->
<link name="${prefix}base_link"/>
<link name="${prefix}base_link_inertia">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 ${pi}" />
        <geometry>
            <xacro:get_mesh name="base" type="visual"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 ${pi}" />
        <geometry>
            <xacro:get_mesh name="base" type="collision"/>
        </geometry>
    </collision>
    <xacro:cylinder_inertial radius="${base_inertia_radius}" length="${base_inertia_length}" mass="${base_mass}">
        <origin xyz="0 0 0" rpy="0 0 0" />
    </xacro:cylinder_inertial>
</link>
<link name="${prefix}shoulder_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 ${pi}" />
        <geometry>
            <xacro:get_mesh name="shoulder" type="visual"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 ${pi}" />
        <geometry>
            <xacro:get_mesh name="shoulder" type="collision"/>
        </geometry>
    </collision>
    <xacro:cylinder_inertial radius="${shoulder_inertia_radius}" length="${shoulder_inertia_length}" mass="${shoulder_mass}">
        <origin xyz="0 0 0" rpy="0 0 0" />
    </xacro:cylinder_inertial>
</link>
<link name="${prefix}upper_arm_link">
    <visual>
        <origin xyz="0 0 ${shoulder_offset}" rpy="${pi/2} 0 ${-pi/2}" />
        <geometry>
            <xacro:get_mesh name="upper_arm" type="visual"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 ${shoulder_offset}" rpy="${pi/2} 0 ${-pi/2}" />
        <geometry>
            <xacro:get_mesh name="upper_arm" type="collision"/>
        </geometry>
    </collision>
    <xacro:cylinder_inertial radius="${upperarm_inertia_radius}" length="${upperarm_inertia_length}" mass="${upper_arm_mass}">
        <origin xyz="${-0.5 * upperarm_inertia_length} 0.0 ${upper_arm_inertia_offset}" rpy="0 ${pi/2} 0" />
    </xacro:cylinder_inertial>
</link>
<link name="${prefix}forearm_link">
    <visual>
        <origin xyz="0 0 ${elbow_offset}" rpy="${pi/2} 0 ${-pi/2}" />
```

```

<geometry>
    <xacro:get_mesh name="forearm" type="visual"/>
</geometry>
</visual>
<collision>
    <origin xyz="0 0 ${elbow_offset}" rpy="${pi/2} 0 ${-pi/2}" />
    <geometry>
        <xacro:get_mesh name="forearm" type="collision"/>
    </geometry>
</collision>
<xacro:cylinder_inertial radius="${forearm_inertia_radius}">
    length="${forearm_inertia_length}" mass="${forearm_mass}">
        <origin xyz="${-0.5 * forearm_inertia_length} 0.0 ${elbow_offset}" rpy="0 ${pi/2} 0" />
    </xacro:cylinder_inertial>
</link>
<link name="${prefix}wrist_1_link">
    <xacro:get_visual_params name="wrist_1" type="visual_offset"/>
    <visual>
        <origin xyz="0 0 ${visual_params}" rpy="${pi/2} 0 0" />
        <geometry>
            <xacro:get_mesh name="wrist_1" type="visual"/>
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 ${visual_params}" rpy="${pi/2} 0 0" />
        <geometry>
            <xacro:get_mesh name="wrist_1" type="collision"/>
        </geometry>
    </collision>
    <xacro:cylinder_inertial radius="${wrist_1_inertia_radius}">
        length="${wrist_1_inertia_length}" mass="${wrist_1_mass}">
            <origin xyz="0 0 0" rpy="0 0 0" />
        </xacro:cylinder_inertial>
    </link>
    <link name="${prefix}wrist_2_link">
        <xacro:get_visual_params name="wrist_2" type="visual_offset"/>
        <visual>
            <origin xyz="0 0 ${visual_params}" rpy="0 0 0" />
            <geometry>
                <xacro:get_mesh name="wrist_2" type="visual"/>
            </geometry>
        </visual>
        <collision>
            <origin xyz="0 0 ${visual_params}" rpy="0 0 0" />
            <geometry>
                <xacro:get_mesh name="wrist_2" type="collision"/>
            </geometry>
        </collision>
        <xacro:cylinder_inertial radius="${wrist_2_inertia_radius}">
            length="${wrist_2_inertia_length}" mass="${wrist_2_mass}">
                <origin xyz="0 0 0" rpy="0 0 0" />
            </xacro:cylinder_inertial>
        </link>
        <link name="${prefix}wrist_3_link">
            <xacro:get_visual_params name="wrist_3" type="visual_offset"/>
            <visual>
                <origin xyz="0 0 ${visual_params}" rpy="${pi/2} 0 0" />
                <geometry>
                    <xacro:get_mesh name="wrist_3" type="visual"/>
                </geometry>
            </visual>
            <collision>
                <origin xyz="0 0 ${visual_params}" rpy="${pi/2} 0 0" />
                <geometry>
                    <xacro:get_mesh name="wrist_3" type="collision"/>
                </geometry>
            </collision>
            <xacro:cylinder_inertial radius="${wrist_3_inertia_radius}">
                length="${wrist_3_inertia_length}" mass="${wrist_3_mass}">

```

```

        <origin xyz="0.0 0.0 ${-0.5 * wrist_3_inertia_length}" rpy="0 0 0" />
    </xacro:cylinder_inertial>
</link>

```

After that, the joints to attach the model to the environment are defined (previous to the robot joints definitions).

```

<!-- base_joint fixes base_link to the environment -->
<joint name="${prefix}base_joint" type="fixed">
    <xacro:insert_block name="origin" />
    <parent link="${parent}" />
    <child link="${prefix}base_link" />
</joint>
<!-- joints - main serial chain -->
<joint name="${prefix}base_link-base_link_inertia" type="fixed">
    <parent link="${prefix}base_link" />
    <child link="${prefix}base_link_inertia" />
    <!-- 'base_link' is REP-103 aligned (so X+ forward), while the internal
        frames of the robot/controller have X+ pointing backwards.
        Use the joint between 'base_link' and 'base_link_inertia' (a dummy
        link/frame) to introduce the necessary rotation over Z (of pi rad).
    -->
    <origin xyz="0 0 0" rpy="0 0 ${pi}" />
</joint>

```

Then, the robot joints are defined. This section was modified to include the needed dynamic property value to the friction of the joint, which should be 100x the joint effort limit for Gazebo simulations, and zero for the rest of the cases. In this section, the joints are linked to its parent joint, placed in a specific origin depending on the model to be used (which is set in the ur.urdf.xacro file), its limits and safety parameters, and its dynamic properties.

```

<!-- Conditional dynamics tags set the joint friction to 100x the joint's effort limit
per discussion at Universal_Robots_ROS2_Gazebo_Simulation/issues/19 -->
<joint name="${prefix}shoulder_pan_joint" type="revolute">
    <parent link="${prefix}base_link_inertia" />
    <child link="${prefix}shoulder_link" />
    <origin xyz="${shoulder_x} ${shoulder_y} ${shoulder_z}" rpy="${shoulder_roll}
${shoulder_pitch} ${shoulder_yaw}" />
    <axis xyz="0 0 1" />
    <limit lower="${shoulder_pan_lower_limit}" upper="${shoulder_pan_upper_limit}"
        effort="${shoulder_pan_effort_limit}" velocity="${shoulder_pan_velocity_limit}"/>
    <xacro:if value="${safety_limits}">
        <safety_controller soft_lower_limit="${shoulder_pan_lower_limit +
safety_pos_margin}" soft_upper_limit="${shoulder_pan_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
    </xacro:if>
    <dynamics damping="0" friction="${shoulder_pan_effort_limit*100.0 if sim_gazebo else
0}"/>
</joint>
<joint name="${prefix}shoulder_lift_joint" type="revolute">
    <parent link="${prefix}shoulder_link" />
    <child link="${prefix}upper_arm_link" />
    <origin xyz="${upper_arm_x} ${upper_arm_y} ${upper_arm_z}" rpy="${upper_arm_roll}
${upper_arm_pitch} ${upper_arm_yaw}" />
    <axis xyz="0 0 1" />
    <limit lower="${shoulder_lift_lower_limit}" upper="${shoulder_lift_upper_limit}"
        effort="${shoulder_lift_effort_limit}"
velocity="${shoulder_lift_velocity_limit}"/>
    <xacro:if value="${safety_limits}">
        <safety_controller soft_lower_limit="${shoulder_lift_lower_limit +
safety_pos_margin}" soft_upper_limit="${shoulder_lift_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
    </xacro:if>
    <dynamics damping="0" friction="${shoulder_lift_effort_limit*100.0 if sim_gazebo
else 0}"/>
</joint>

```

```

<joint name="${prefix}elbow_joint" type="revolute">
  <parent link="${prefix}upper_arm_link" />
  <child link="${prefix}forearm_link" />
  <origin xyz="${forearm_x} ${forearm_y} ${forearm_z}" rpy="${forearm_roll}
${forearm_pitch} ${forearm_yaw}" />
  <axis xyz="0 0 1" />
  <limit lower="${elbow_joint_lower_limit}" upper="${elbow_joint_upper_limit}"
    effort="${elbow_joint_effort_limit}" velocity="${elbow_joint_velocity_limit}"/>
  <xacro:if value="${safety_limits}">
    <safety_controller soft_lower_limit="${elbow_joint_lower_limit +
safety_pos_margin}" soft_upper_limit="${elbow_joint_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
  </xacro:if>
  <dynamics damping="0" friction="${elbow_joint_effort_limit*100.0 if sim_gazebo else
0}"/>
</joint>
<joint name="${prefix}wrist_1_joint" type="revolute">
  <parent link="${prefix}forearm_link" />
  <child link="${prefix}wrist_1_link" />
  <origin xyz="${wrist_1_x} ${wrist_1_y} ${wrist_1_z}" rpy="${wrist_1_roll}
${wrist_1_pitch} ${wrist_1_yaw}" />
  <axis xyz="0 0 1" />
  <limit lower="${wrist_1_lower_limit}" upper="${wrist_1_upper_limit}"
    effort="${wrist_1_effort_limit}" velocity="${wrist_1_velocity_limit}"/>
  <xacro:if value="${safety_limits}">
    <safety_controller soft_lower_limit="${wrist_1_lower_limit + safety_pos_margin}"
soft_upper_limit="${wrist_1_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
  </xacro:if>
  <dynamics damping="0" friction="${wrist_1_effort_limit*100.0 if sim_gazebo else
0}"/>
</joint>
<joint name="${prefix}wrist_2_joint" type="revolute">
  <parent link="${prefix}wrist_1_link" />
  <child link="${prefix}wrist_2_link" />
  <origin xyz="${wrist_2_x} ${wrist_2_y} ${wrist_2_z}" rpy="${wrist_2_roll}
${wrist_2_pitch} ${wrist_2_yaw}" />
  <axis xyz="0 0 1" />
  <limit lower="${wrist_2_lower_limit}" upper="${wrist_2_upper_limit}"
    effort="${wrist_2_effort_limit}" velocity="${wrist_2_velocity_limit}"/>
  <xacro:if value="${safety_limits}">
    <safety_controller soft_lower_limit="${wrist_2_lower_limit + safety_pos_margin}"
soft_upper_limit="${wrist_2_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
  </xacro:if>
  <dynamics damping="0" friction="${wrist_2_effort_limit*100.0 if sim_gazebo else
0}"/>
</joint>
<joint name="${prefix}wrist_3_joint" type="revolute">
  <parent link="${prefix}wrist_2_link" />
  <child link="${prefix}wrist_3_link" />
  <origin xyz="${wrist_3_x} ${wrist_3_y} ${wrist_3_z}" rpy="${wrist_3_roll}
${wrist_3_pitch} ${wrist_3_yaw}" />
  <axis xyz="0 0 1" />
  <limit lower="${wrist_3_lower_limit}" upper="${wrist_3_upper_limit}"
    effort="${wrist_3_effort_limit}" velocity="${wrist_3_velocity_limit}"/>
  <xacro:if value="${safety_limits}">
    <safety_controller soft_lower_limit="${wrist_3_lower_limit + safety_pos_margin}"
soft_upper_limit="${wrist_3_upper_limit - safety_pos_margin}"
k_position="${safety_k_position}" k_velocity="0.0"/>
  </xacro:if>
  <dynamics damping="0" friction="${wrist_3_effort_limit*100.0 if sim_gazebo else
0}"/>
</joint>
```

In the last section of the code, the force-torque sensor frame is defined, at the end of the wrist, where a tool would be placed, as well as the `<!-- ROS-Industrial 'base' frame - base_link to UR`

'Base' Coordinates transform -->, the flange frame as attachment point for End Effectors, and the tool0 frame.

```
<link name="${prefix}ft_frame"/>
<joint name="${prefix}wrist_3_link-ft_frame" type="fixed">
  <parent link="${prefix}wrist_3_link"/>
  <child link="${prefix}ft_frame"/>
  <origin xyz="0 0 0" rpy="${pi} 0 0"/>
</joint>
<!-- ROS-Industrial 'base' frame - base_link to UR 'Base' Coordinates transform -->
<link name="${prefix}base"/>
<joint name="${prefix}base_link-base_fixed_joint" type="fixed">
  <!-- Note the rotation over Z of pi radians - as base_link is REP-103
      aligned (i.e., has X+ forward, Y+ left and Z+ up), this is needed
      to correctly align 'base' with the 'Base' coordinate system of
      the UR controller.
  -->
  <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
  <parent link="${prefix}base_link"/>
  <child link="${prefix}base"/>
</joint>
<!-- ROS-Industrial 'flange' frame - attachment point for EEF models -->
<link name="${prefix}flange" />
<joint name="${prefix}wrist_3-flange" type="fixed">
  <parent link="${prefix}wrist_3_link" />
  <child link="${prefix}flange" />
  <origin xyz="0 0 0" rpy="0 ${-pi/2.0} ${-pi/2.0}" />
</joint>
<!-- ROS-Industrial 'tool0' frame - all-zeros tool frame -->
<link name="${prefix}tool0"/>
<joint name="${prefix}flange-tool0" type="fixed">
  <!-- default toolframe - X+ left, Y+ up, Z+ front -->
  <origin xyz="0 0 0" rpy="${pi/2.0} 0 ${pi/2.0}"/>
  <parent link="${prefix}flange"/>
  <child link="${prefix}tool0"/>
</joint>
</xacro:macro>
</robot>
```

ur.urdf.xacro

First the code starts the XML file and defines a robot object, giving it a name using the argument "name".

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="$(arg name)">
  <!-- robot name parameter -->
  <xacro:arg name="name" default="ur"/>
```

After that, the ur_macro.xacro is included, which contains the main macro use to describe the UR manipulators and their control interfaces.

```
<!-- import main macro -->
<xacro:include filename="$(find ur_description)/urdf/ur_macro.xacro"/>
```

Next section defines the ur_type argument, which specifies the type of the UR robot being used. This argument will be used to implement the correct values for the specific arguments of the selected model.

```
<!-- possible 'ur_type' values: ur3, ur3e, ur5, ur5e, ur10, ur10e, ur16e -->
<!-- the default value should raise an error in case this was called without defining
the type -->
<xacro:arg name="ur_type" default="ur5x"/>
```

Next section assigns values to parameters related with the robot, the control and the tool communication. Those parameters specific to the selected model in the ur_type will be dependent on the ur_type parameter, the rest will have generic values useful for any model.

```
<!-- parameters -->
<xacro:arg name="tf_prefix" default="" />
<xacro:arg name="joint_limit_params" default="$(find ur_description)/config/$(arg
ur_type)/joint_limits.yaml"/>
<xacro:arg name="kinematics_params" default="$(find ur_description)/config/$(arg
ur_type)/default_kinematics.yaml"/>
<xacro:arg name="physical_params" default="$(find ur_description)/config/$(arg
ur_type)/physical_parameters.yaml"/>
<xacro:arg name="visual_params" default="$(find ur_description)/config/$(arg
ur_type)/visual_parameters.yaml"/>
<xacro:arg name="transmission_hw_interface" default="" />
<xacro:arg name="safety_limits" default="false" />
<xacro:arg name="safety_pos_margin" default="0.15" />
<xacro:arg name="safety_k_position" default="20" />
<!-- ros2_control related parameters -->
<xacro:arg name="headless_mode" default="false" />
<xacro:arg name="robot_ip" default="0.0.0.0" />
<xacro:arg name="script_filename" default="" />
<xacro:arg name="output_recipe_filename" default="" />
<xacro:arg name="input_recipe_filename" default="" />
<xacro:arg name="reverse_ip" default="0.0.0.0" />
<xacro:arg name="script_command_port" default="50004" />
<!-- tool communication related parameters-->
<xacro:arg name="use_tool_communication" default="false" />
<xacro:arg name="tool_voltage" default="0" />
<xacro:arg name="tool_parity" default="0" />
<xacro:arg name="tool_baud_rate" default="115200" />
<xacro:arg name="tool_stop_bits" default="1" />
<xacro:arg name="tool_rx_idle_chars" default="1.5" />
<xacro:arg name="tool_tx_idle_chars" default="3.5" />
<xacro:arg name="tool_device_name" default="/tmp/ttyUR" />
```

```
<xacro:arg name="tool_tcp_port" default="54321" />
```

The next section defines arguments to configure the simulation of the robot. The fake hardware and sensor commands are to indicate if the simulation should use fake hardware and sensor data for testing and simulation purposes. The sim_gazebo and sim_ignition are used to indicate whether the simulation will be performed using Gazebo or Ignition software. And the simulation_controllers to specify which controller should be loaded for simulation.

```
<!-- Simulation parameters -->
<xacro:arg name="use_fake_hardware" default="false" />
<xacro:arg name="fake_sensor_commands" default="false" />
<xacro:arg name="sim_gazebo" default="false" />
<xacro:arg name="sim_ignition" default="false" />
<xacro:arg name="simulation_controllers" default="" />
```

Next section defines an argument initial_positions_file, which is used to specify the path to a YAML file containing the initial positions of the robot for simulation. This argument is then converted to a property to load the YAML file later on, when defining the arm object.

```
<!-- initial position for simulations (Fake Hardware, Gazebo, Ignition) -->
<xacro:arg name="initial_positions_file" default="$(find
ur_description)/config/initial_positions.yaml"/>

<!-- convert to property to use substitution in function -->
<xacro:property name="initial_positions_file" default="$(arg initial_positions_file)"/>
```

Next section defines the root link of the robot.

```
<!-- create link fixed to the "world" -->
<link name="world" />
```

Next section uses the ur_robot macro included previously, which defines the robot arm. It uses the parameters defined previously to configure the robot's name, joint limits, kinematics parameters, physical parameters, visual parameters, transmission hardware interface, safety limits, and tool communication parameters. It also includes arguments for simulating the robot in Gazebo or Ignition, for controlling the robot with ROS2 controllers, and for specifying the initial positions of the robot for simulation. Moreover, it defines the initial position of the robot respect the world frame.

```
<!-- arm -->
<xacro:ur_robot
  name="$(arg name)"
  tf_prefix="$(arg tf_prefix)"
  parent="world"
  joint_limits_parameters_file="$(arg joint_limit_params)"
  kinematics_parameters_file="$(arg kinematics_params)"
  physical_parameters_file="$(arg physical_params)"
  visual_parameters_file="$(arg visual_params)"
  transmission_hw_interface="$(arg transmission_hw_interface)"
  safety_limits="$(arg safety_limits)"
  safety_pos_margin="$(arg safety_pos_margin)"
  safety_k_position="$(arg safety_k_position)"
  use_fake_hardware="$(arg use_fake_hardware)"
  fake_sensor_commands="$(arg fake_sensor_commands)"
  sim_gazebo="$(arg sim_gazebo)"
  sim_ignition="$(arg sim_ignition)"
```

```

headless_mode="$(arg headless_mode)"
initial_positions="${xacro.load_yaml(initial_positions_file)}"
use_tool_communication="$(arg use_tool_communication)"
tool_voltage="$(arg tool_voltage)"
tool_parity="$(arg tool_parity)"
tool_baud_rate="$(arg tool_baud_rate)"
tool_stop_bits="$(arg tool_stop_bits)"
tool_rx_idle_chars="$(arg tool_rx_idle_chars)"
tool_tx_idle_chars="$(arg tool_tx_idle_chars)"
tool_device_name="$(arg tool_device_name)"
tool_tcp_port="$(arg tool_tcp_port)"
robot_ip="$(arg robot_ip)"
script_filename="$(arg script_filename)"
output_recipe_filename="$(arg output_recipe_filename)"
input_recipe_filename="$(arg input_recipe_filename)"
reverse_ip="$(arg reverse_ip)"
script_command_port="$(arg script_command_port)"
>
<origin xyz="0 0 0" rpy="0 0 0" />           <!-- position robot in the world -->
</xacro:ur_robot>

```

Next section will check if the simulation will be runed in Gazebo, if so, a Gazebo plugin will be included. The first element, sets the plugin to be applied to the entire Gazebo world with the reference="world". Then the plugin is included specifying the name of the library and its package. The parameters set the controllers of the simulations to the ones defined previously.

```

<xacro:if value="$(arg sim_gazebo)">
  <!-- Gazebo plugins -->
  <gazebo reference="world">
    </gazebo>
    <gazebo>
      <plugin filename="libgazebo_ros2_control.so" name="gazebo_ros2_control">
        <parameters>$(arg simulation_controllers)</parameters>
      </plugin>
    </gazebo>
  </xacro:if>

```

The last section does the same but for Ignition simulation. In this case the name of the node that controls the robot needs to be included (controller_manager_node_name), too.

```

<xacro:if value="$(arg sim_ignition)">
  <!-- Gazebo plugins -->
  <gazebo reference="world">
    </gazebo>
    <gazebo>
      <plugin filename="libign_ros2_control-system.so"
name="ign_ros2_control::IgnitionROS2ControlPlugin">
        <parameters>$(arg simulation_controllers)</parameters>
        <controller_manager_node_name>$(arg
tf_prefix)controller_manager</controller_manager_node_name>
      </plugin>
    </gazebo>
  </xacro:if>
</robot>

```

ur_sim_control.launch.py

Import the needed functions for the launch file.

```
from launch import LaunchDescription
from launch.actions import (
    DeclareLaunchArgument,
    IncludeLaunchDescription,
    OpaqueFunction,
    RegisterEventHandler,
)
from launch.conditions import IfCondition, UnlessCondition
from launch.event_handlers import OnProcessExit
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import Command, FindExecutable, LaunchConfiguration,
PathJoinSubstitution
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare
```

Assign the launch configuration of the arguments to variables to simplify the code.

```
def launch_setup(context, *args, **kwargs):
    # Initialize Arguments
    ur_type = LaunchConfiguration("ur_type")
    safety_limits = LaunchConfiguration("safety_limits")
    safety_pos_margin = LaunchConfiguration("safety_pos_margin")
    safety_k_position = LaunchConfiguration("safety_k_position")
    # General arguments
    runtime_config_package = LaunchConfiguration("runtime_config_package")
    controllers_file = LaunchConfiguration("controllers_file")
    description_package = LaunchConfiguration("description_package")
    description_file = LaunchConfiguration("description_file")
    prefix = LaunchConfiguration("prefix")
    start_joint_controller = LaunchConfiguration("start_joint_controller")
    initial_joint_controller = LaunchConfiguration("initial_joint_controller")
    launch_rviz = LaunchConfiguration("launch_rviz")
```

Define as variables the path of the files that contain a list of initial joint controllers for the robot, configuration for the RViz tool to display the robot.

```
initial_joint_controllers = PathJoinSubstitution(
    [FindPackageShare(runtime_config_package), "config", controllers_file]
)

rviz_config_file = PathJoinSubstitution(
    [FindPackageShare(description_package), "rviz", "view_robot.rviz"]
)
```

Create the robot description command with the needed arguments to be used as input to the XACRO file. All the required arguments can be found listed at the Universal_Robots_ROS2_Description/urdf/ur.urdf.xacro file. This command will generate the URDF file to describe the robot.

```
robot_description_content = Command(
    [
        PathJoinSubstitution([FindExecutable(name="xacro")]),
        " ",
        PathJoinSubstitution(
            [FindPackageShare(description_package), "urdf", description_file]
        ),
        " ",
    ]
)
```

```

    "safety_limits:=",
    safety_limits,
    " ",
    "safety_pos_margin:=",
    safety_pos_margin,
    " ",
    "safety_k_position:=",
    safety_k_position,
    " ",
    "name:=",
    "ur",
    " ",
    "ur_type:=",
    ur_type,
    " ",
    "prefix:=",
    prefix,
    " ",
    "sim_gazebo:=true",
    " ",
    "simulation_controllers:=",
    initial_joint_controllers,
]
)

```

Create, from the description command, the robot description that will be used to define the launched nodes in the file.

```
robot_description = {"robot_description": robot_description_content}
```

Define the first nodes that will be launched. The first two use the parameters previously implemented with the robot description and the RViz configuration file. The third one is in charge of spawning the broadcaster that will provide the UR3 joint states. The defined nodes are `robot_state_publisher_node`, `rviz_node` and `joint_state_broadcaster_spawner`. The general functionality of these nodes is the same as the nodes from `ur_control.launch.py` file.

```

robot_state_publisher_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    output="both",
    parameters=[{"use_sim_time": True}, robot_description],
)

rviz_node = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="log",
    arguments=["-d", rviz_config_file],
    condition=IfCondition(launch_rviz),
)

joint_state_broadcaster_spawner = Node(
    package="controller_manager",
    executable="spawner",
    arguments=["joint_state_broadcaster", "--controller-manager",
    "/controller_manager"],
)

```

Create a delay between node initialization to reduce computing cost and minimize possible starting failures.

```

# Delay rviz start after `joint_state_broadcaster`
delay_rviz_after_joint_state_broadcaster_spawner = RegisterEventHandler(
    event_handler=OnProcessExit(
        target_action=joint_state_broadcaster_spawner,
        on_exit=[rviz_node],
    )
)

```

Spawn the controllers that will be used to control the robot and provide feedback of its state. These controllers will be started or stopped using the nodes initial_joint_controller_started and initial_joint_controller_stopped.

```

# There may be other controllers of the joints, but this is the initially-started one
initial_joint_controller_spawner_started = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[initial_joint_controller, "-c", "/controller_manager"],
    condition=IfCondition(start_joint_controller),
)
initial_joint_controller_spawner_stopped = Node(
    package="controller_manager",
    executable="spawner",
    arguments=[initial_joint_controller, "-c", "/controller_manager", "--stopped"],
    condition=UnlessCondition(start_joint_controller),
)

```

Launch a Gazebo simulation, executing all needed nodes with gazebo.launch.py file, and spawn a robot using the previously defined robot_description inside the simulation. This is the main difference with the ur_control.launch.py file used for URSim and real UR3 robot.

```

# Gazebo nodes
gazebo = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        [FindPackageShare("gazebo_ros"), "/launch", "/gazebo.launch.py"]
    ),
)

# Spawn robot
gazebo_spawn_robot = Node(
    package="gazebo_ros",
    executable="spawn_entity.py",
    name="spawn_ur",
    arguments=["-entity", "ur", "-topic", "robot_description"],
    output="screen",
)

```

Return all the previously defined nodes to initiate them in the Launch Description.

```

nodes_to_start = [
    robot_state_publisher_node,
    joint_state_broadcaster_spawner,
    delay_rviz_after_joint_state_broadcaster_spawner,
    initial_joint_controller_spawner_stopped,
    initial_joint_controller_spawner_started,
    gazebo,
    gazebo_spawn_robot,
]
return nodes_to_start

```

Declare all the arguments that will be used to define the nodes and robot description previously commented. All the arguments have name, description and default_value (except for the type that must be provided by the user when launching) listed.

```

def generate_launch_description():
    declared_arguments = []
    # UR specific arguments
    declared_arguments.append(
        DeclareLaunchArgument(
            "ur_type",
            description="Type/series of used UR robot.",
            choices=["ur3", "ur3e", "ur5", "ur5e", "ur10", "ur10e", "ur16e"],
            default_value="ur5e",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_limits",
            default_value="true",
            description="Enables the safety limits controller if true.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_pos_margin",
            default_value="0.15",
            description="The margin to lower and upper limits in the safety controller.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_k_position",
            default_value="20",
            description="k-position factor in the safety controller.",
        )
    )
    # General arguments
    declared_arguments.append(
        DeclareLaunchArgument(
            "runtime_config_package",
            default_value="ur_simulation_gazebo",
            description='Package with the controller\'s configuration in "config" folder.
        \
            Usually the argument is not set, it enables use of a custom setup.',
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "controllers_file",
            default_value="ur_controllers.yaml",
            description="YAML file with the controllers configuration.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "description_package",
            default_value="ur_description",
            description="Description package with robot URDF/XACRO files. Usually the
argument \
            is not set, it enables use of a custom description.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "description_file",
            default_value="ur.urdf.xacro",
            description="URDF/XACRO description file with the robot.",
        )
    )

```

```

        )
    declared_arguments.append(
        DeclareLaunchArgument(
            "prefix",
            default_value="",
            description="Prefix of the joint names, useful for \
multi-robot setup. If changed than also joint names in the controllers'\
configuration \
have to be updated.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "start_joint_controller",
            default_value="true",
            description="Enable headless mode for robot control",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "initial_joint_controller",
            default_value="joint_trajectory_controller",
            description="Robot controller to start.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument("launch_rviz", default_value="true", description="Launch\
RViz?")
    )
)

```

Return the Launch Description of the arguments and the launch setup that returns all the nodes to be executed.

```
return LaunchDescription(declared_arguments + [OpaqueFunction(function=launch_setup)])
```

test_goal_publishers_config.yaml

This code was modified to match the new initial position of the robot set in the initial_positions.yaml file. If the initial position is modified again, the starting_point_limits should be modified too, or the check_starting_point should be set to false. It is used in the test_scaled_joint_trajectory_controller.launch.py to send 4 goal positions to the publisher_joint_trajectory_controller.cpp file. When launched, this file will make the robot in the simulation, or the real robot, go to the set goals (if accepted) one after another.

The first section sets the goals when using the scaled_joint_trajectory_controller. It assigns the values for the 4 goals, the time to wait between goal publication, the name of the joints in order to correspond the position values in the goal, and the starting point limits. The last ones were modified to match the new initial position.

```
publisher_scaled_joint_trajectory_controller:  
  ros__parameters:  
  
    controller_name: "scaled_joint_trajectory_controller"  
    wait_sec_between_publish: 6  
  
    goal_names: ["pos1", "pos2", "pos3", "pos4"]  
    pos1: [0.785, -1.57, 0.785, 0.785, 0.785, 0.785]  
    pos2: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]  
    pos3: [0.0, -1.57, 0.0, 0.0, -0.785, 0.0]  
    pos4: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]  
  
    joints:  
      - shoulder_pan_joint  
      - shoulder_lift_joint  
      - elbow_joint  
      - wrist_1_joint  
      - wrist_2_joint  
      - wrist_3_joint  
  
    check_starting_point: true  
    starting_point_limits:  
      shoulder_pan_joint: [-0.1,0.1]  
      shoulder_lift_joint: [-1.6,-1.5]  
      elbow_joint: [-0.1,0.1]  
      wrist_1_joint: [-1.6,-1.5]  
      wrist_2_joint: [-1.6,-1.5]  
      wrist_3_joint: [-0.1,0.1]
```

Last section does the same but for the joint_trajectory_controller.

```
publisher_joint_trajectory_controller:  
  ros__parameters:  
  
    controller_name: "joint_trajectory_controller"  
    wait_sec_between_publish: 6  
  
    goal_names: ["pos1", "pos2", "pos3", "pos4"]  
    pos1: [0.785, -1.57, 0.785, 0.785, 0.785, 0.785]  
    pos2: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]  
    pos3: [0.0, -1.57, 0.0, 0.0, -0.785, 0.0]  
    pos4: [0.0, -1.57, 0.0, 0.0, 0.0, 0.0]  
  
    joints:  
      - shoulder_pan_joint  
      - shoulder_lift_joint  
      - elbow_joint  
      - wrist_1_joint  
      - wrist_2_joint
```

```
- wrist_3_joint  
check_starting_point: true  
starting_point_limits:  
    shoulder_pan_joint: [2.0,2.2]  
    shoulder_lift_joint: [-1.6,-1.5]  
    elbow_joint: [1.5,1.6]  
    wrist_1_joint: [-1.6,-1.5]  
    wrist_2_joint: [-1.6,-1.5]  
    wrist_3_joint: [-0.1,0.1]
```

APPENDIX 2. INSTALLATION AND SETUP MANUAL

It is important to notice that some of these indications could change in the future, some of the bugs founded will be solved and new ones might appear.

ROS2 SETUP AND INSTALLATION

Ubuntu versión: Ubuntu Jammy 22.04

Steps to follow:

6. Make sure your locale supports UTF-8.

In Shell #1:

```
locale # check for UTF-8

# If you do not have UTF-8, run the next lines
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale # verify settings
```

Expected output:

```
dan@dan-HP-Z240-Tower-Workstation:~$ locale
LANG=en_US.UTF-8
LANGUAGE=
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

7. Setup the sources to add the ROS2 apt repository

Ensure Ubuntu Universe repository is enable.

In Shell #1:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Add ROS2 GPG key with apt.

In Shell #1:

```
sudo apt update && sudo apt install curl -y  
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o  
/usr/share/keyrings/ros-archive-keyring.gpg
```

Add the repository to the sources list.

In Shell #1:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-  
keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo  
$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

8. Update and Upgrade the apt repositories

In Shell #1:

```
sudo apt update  
sudo apt upgrade
```

9. Install ROS2 packages

In Shell #1:

```
sudo apt install ros-humble-desktop-full
```

After that, run the next command to install Development tools for ROS2, such us colcon, that will allow you to compile and build packages.

In Shell #1:

```
sudo apt install ros-dev-tools
```

All the packages will be located at the /opt/ros/humble folder of the system.

10. Environment setup to be able to use ROS2

In order to use ROS2 packages and commands, you will have to run the next command in every new shell that you open.

In Shell #1:

```
source /opt/ros/humble/setup.bash
```

Another option is to add the previous line into the ~/.bashrc file of your user, so that it is executed every time you open a new shell.

In Shell #1:

```
gedit ~/.bashrc
```

Go to the end of the file and copy the source /opt/ros/humble/setup.bash line.

As optional step implement the colcon autocomplete command.

In Shell #1:

```
gedit ~/.bashrc
```

Copy the line at the end of the file.

```
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```

IMPLEMENT UR ROS2 DRIVER TO YOUR SYSTEM

Steps to follow to build from source:

5. Create a new ROS2 workspace.

In Shell #1:

```
export COLCON_WS=~/workspace/ros_humble_ur_driver  
mkdir -p $COLCON_WS/src
```

6. Clone relevant packages and install dependencies.

In this step we will not only install the UR ROS2 Driver package, but also all the packages needed to create the control framework for the UR3. These packages are Universal_Robots_ROS2_Driver, Universal_Robots_ROS2_Description, Universal_Robots_Client_Library, ur_msgs, control_msgs, kinematics_interface, and the previously analyzed ros2_control and ros2_controllers.

In Shell #1:

```
cd $COLCON_WS  
git clone https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver.git  
src/Universal_Robots_ROS2_Driver  
vcs import src --skip-existing -input  
src/Universal_Robots_ROS2_Driver/Universal_Robots_ROS2_Driver.humble.repos
```

7. Initialize rosdep, run it and build the workspace.

In Shell #1:

```
sudo rosdep init  
rosdep update  
rosdep install --ignore-src --from-paths src -y  
colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release
```

With the current version of the files, once the colcon build is executed, an error will show up in the terminal output.

This error will indicate that the function *publish_state* in line 262 from the *~/workspace/ros_humble_ur_driver/src/Universal_Robots_ROS2_Driver/ur_controllers/src/scaled_joint_trajectory_controller.cpp* has extra arguments. The original function in line 249 from the file *~/workspace/ros_humble_ur_driver/install/joint_trajectory_controller/include/joint_trajectory_controller/joint_trajectory_controller.hpp* has no argument called time.

In order to solve the problem, open the scaled_joint_trajectory_controller.cpp, go to line 262, eliminate the time argument from the function call, and save the file (Ctrl+S).

8. Rebuild the workspace and source it.

In Shell #1:

```
colcon build -cmake-args -DCMAKE_BUILD_TYPE=Release  
source install/setup.bash
```

Your new overlay workspace should be built and operational to start testing control applications with a simulated or real UR3 robot.

STARTING URSIM DOCKER SIMULATION

As the simulation executable is in the UR Driver package, you need to first source the workspace.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver  
colcon build  
source install/setup.bash
```

Then run the executable in the same shell. Remember to select the correct type of robot to simulate, in this case ur3. The -m represents the model parameter selection.

In Shell #1

```
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

The first time you run this, your system will acknowledge you that the ursim_net network is not created or that it does not exist, so it will try to create it by pulling it from universalrobots/ur3_cb3.

Your system might then give you a “permission denied” error. To solve it, you will have to follow the next steps.

5. Create docker group (if you haven't yet).

In Shell #1

```
sudo groupadd Docker
```

6. Add your user to docker group.

In Shell #1

```
sudo usermod -aG docker ${USER}
```

7. Log out and back in from the group so the changes apply.

In Shell #1

```
su -s ${USER}
```

8. Change docker directory permissions so the “permission denied” error does not appear.

In Shell #1

```
sudo chmod 666 /var/run/docker.sock
```

Once this is done, rerun the ursim command.

In Shell #1

```
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

Once the ursim_net is created, the link to the URSim website will be displayed. The next time the simulation is started, only the link will be printed in the shell.

Note: if in future runs the “permission denied” appears again, you will just need to change the permissions of the docker directory as you did in the 4th step, and then run the URSim.

In Shell #1

```
sudo chmod 666 /var/run/docker.sock  
ros2 run ur_robot_driver start_ursim.sh -m ur3
```

Once you introduce your password, a charging screen will appear. Once it is done loading, the docker will ask you to initialize the robot. To do so press the “ON” button as in **Figure 95** and the initialization screen will change to the standby “Idle” mode as in **Figure 96**. Then press “ON” again and the simulated robot will be ready to use, **Figure 97**.

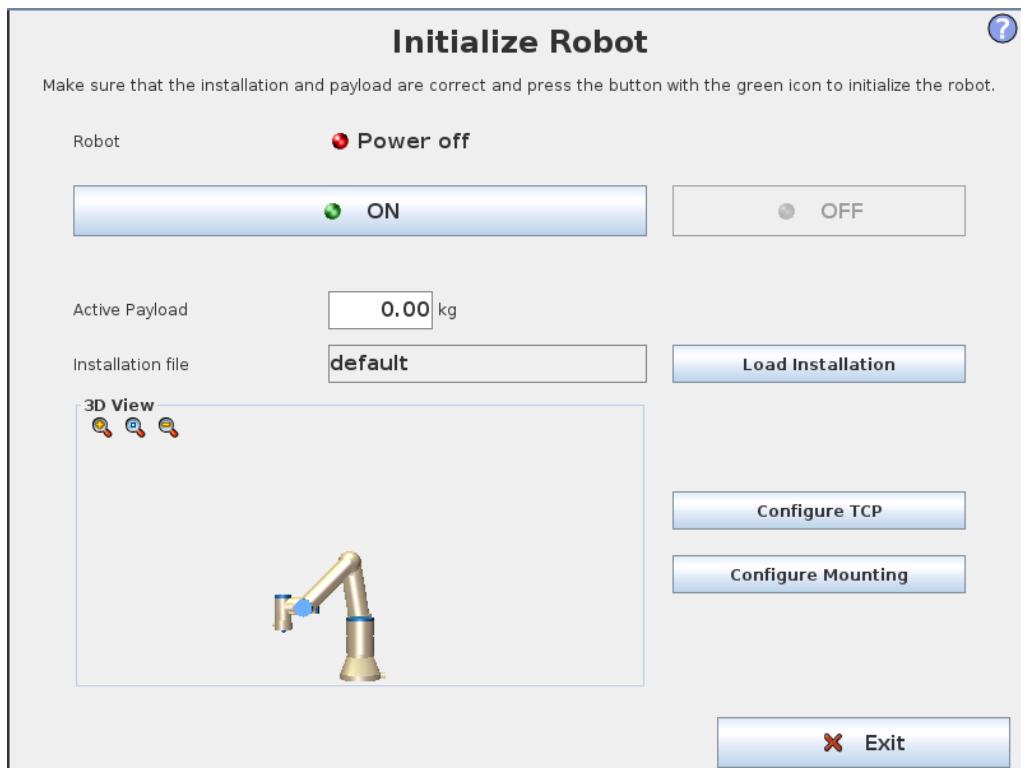


Figure 95. “Power off” robot mode. Source: Created by the author.

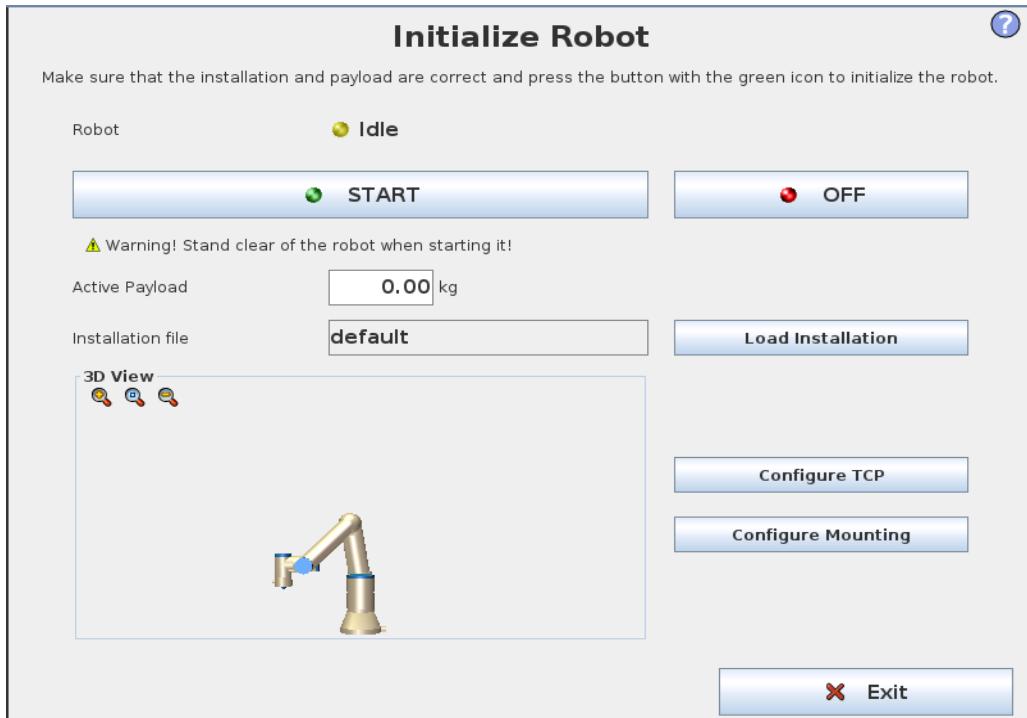


Figure 96. “Idle” robot mode. Source: Created by the author.

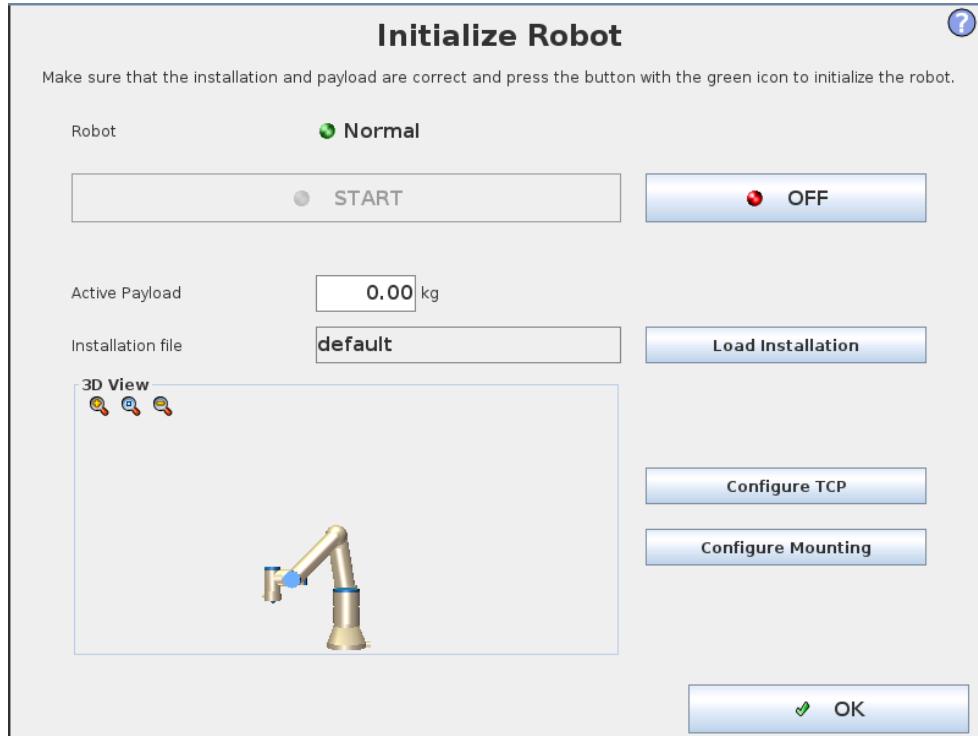


Figure 97. “Normal” robot mode. Source: created by the author.

Now that the robot is activated in its “Normal” mode, the main PolyScope screen will show in the tab. That PolyScope will simulate a real UR PlyScope panel.

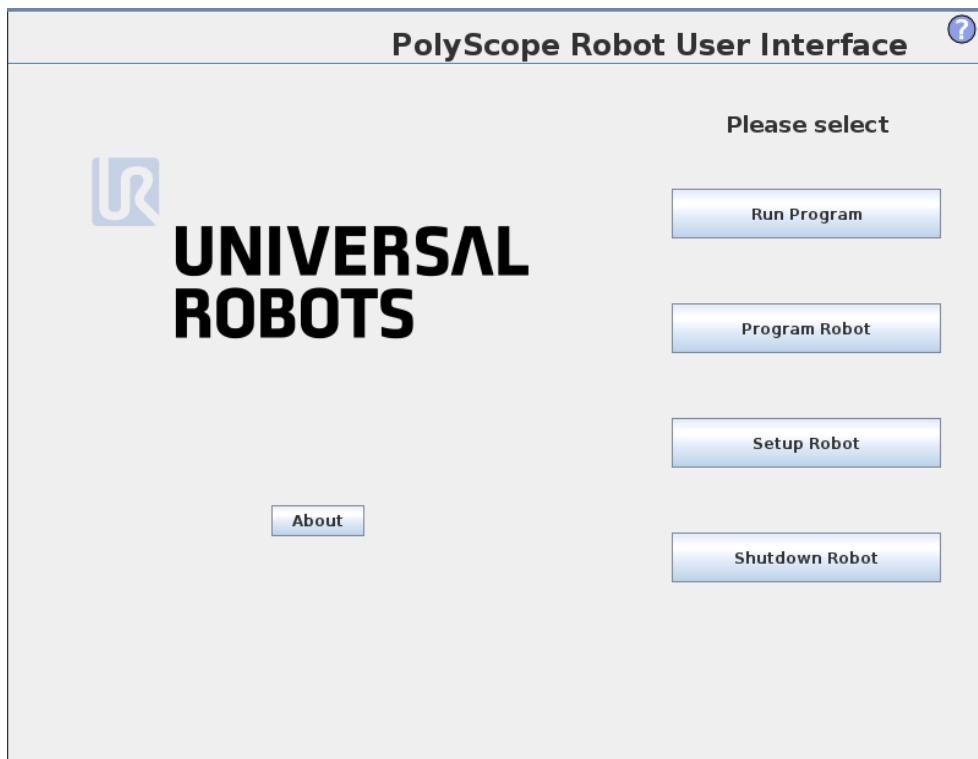


Figure 98. Main PolyScope URSim screen. Source: Created by the author.

From this point two options can be considered, controlling the movement of the robot manually with the “FreeDrive” mode or with an extern program.

To use the FreeDrive, you will need to select the Run Program option from the main screen. That will lead you to the interface shown in **Figure 99**.

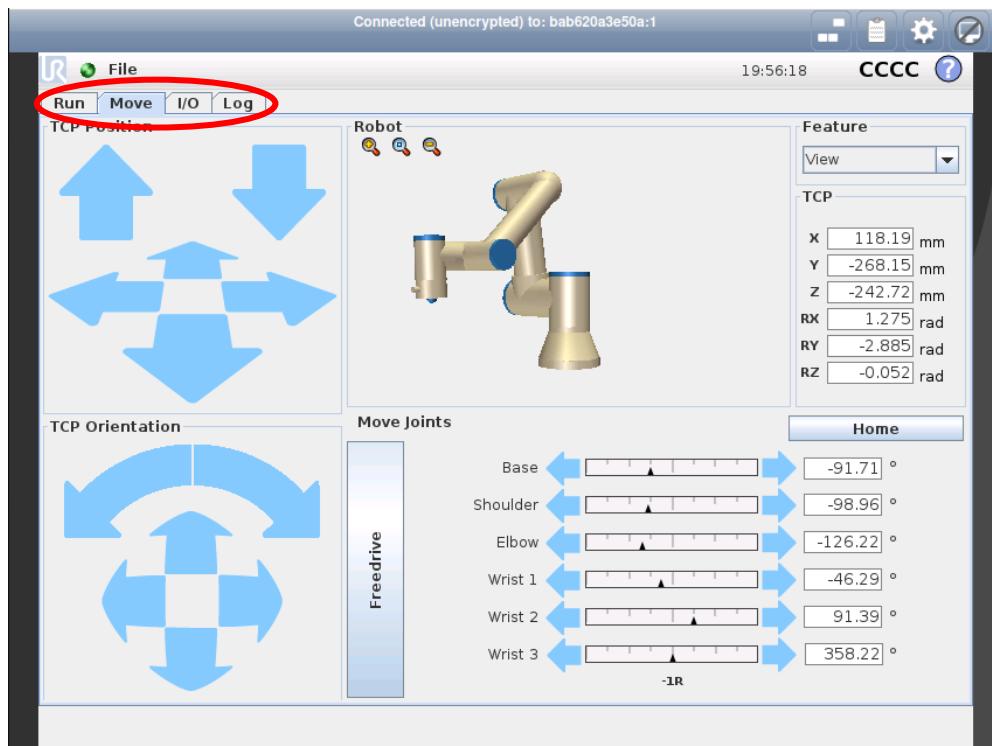


Figure 99. URSim FreeDrive interface. Source: Created by the author.

In the FreeDrive mode, you can use the same tools as in PolyScope to move the robot around, using the TCP or Joint movements to set goals for the robot.

The other option is use an external program to control the simulation. To do so you will have to select the Program Robot option in the main screen.

Then you will have to open a new shell and source the workspace and launch the ur_control file to create the control framework for the UR3.

In Shell #2

```
cd ~/workspace/ros_humble_ur_driver  
source install/setup.bash  
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur3 robot_ip:=192.168.56.101  
launch_rviz:=false
```

Once the controller is running, move to the URSim browser tab and follow the next steps:

6. Select Empty Program

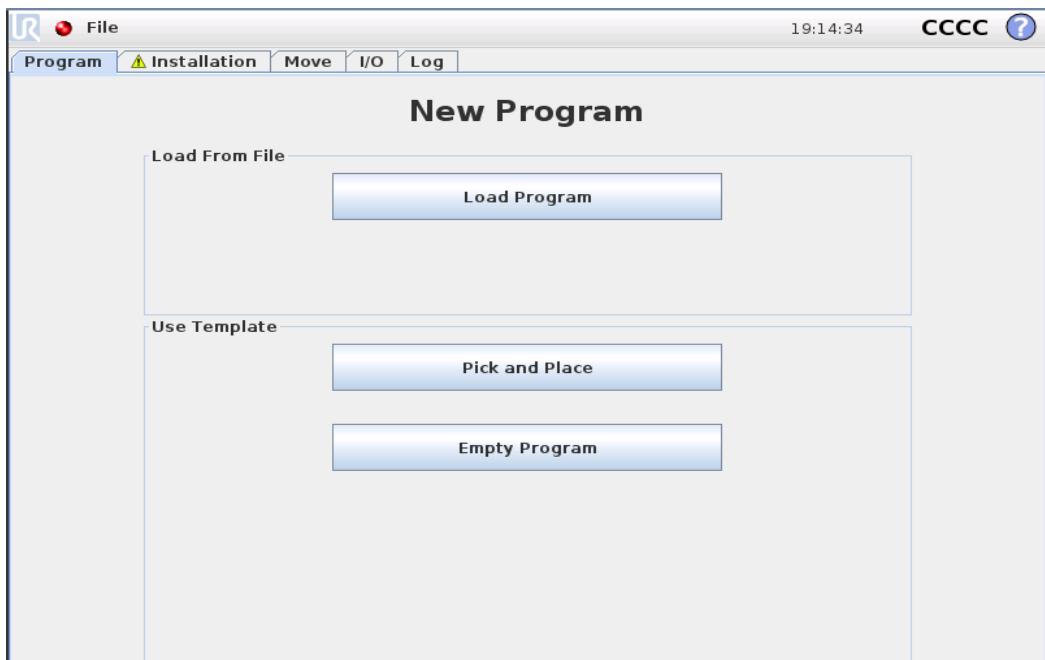


Figure 100. Program Robot screen. Source: Created by the author.

7. Inside Empty Program, go to the Program Tab (up left corner of the window) and select the Structure Tab.

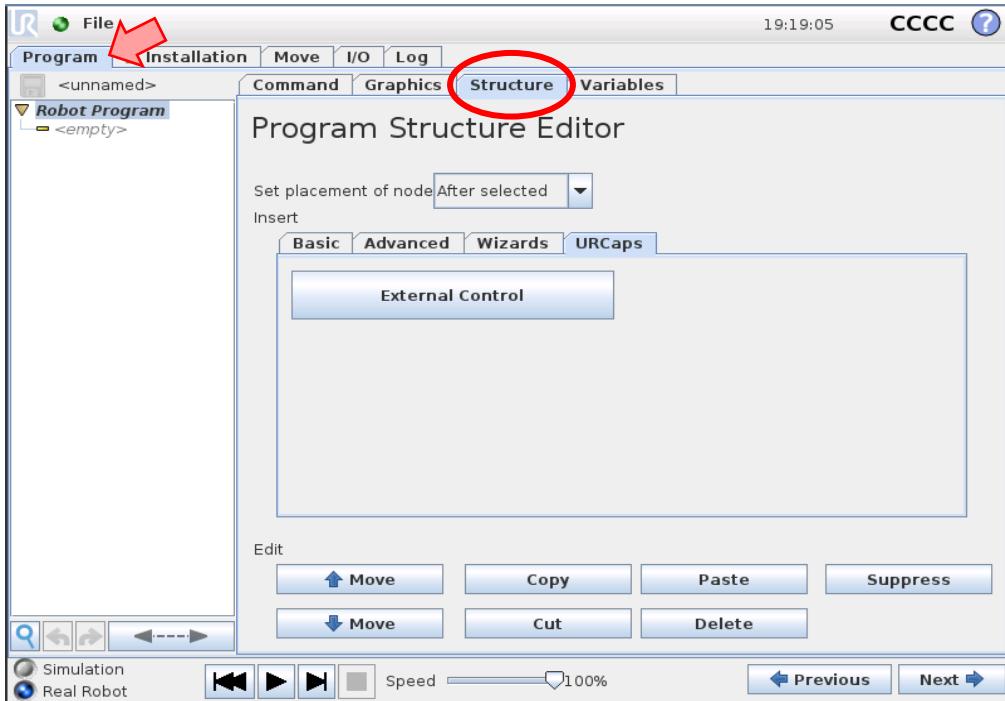


Figure 101. Empty Program screen. Source: Created by the author.

8. Inside the Structure Tab move to the URCaps Tab and select External Control.

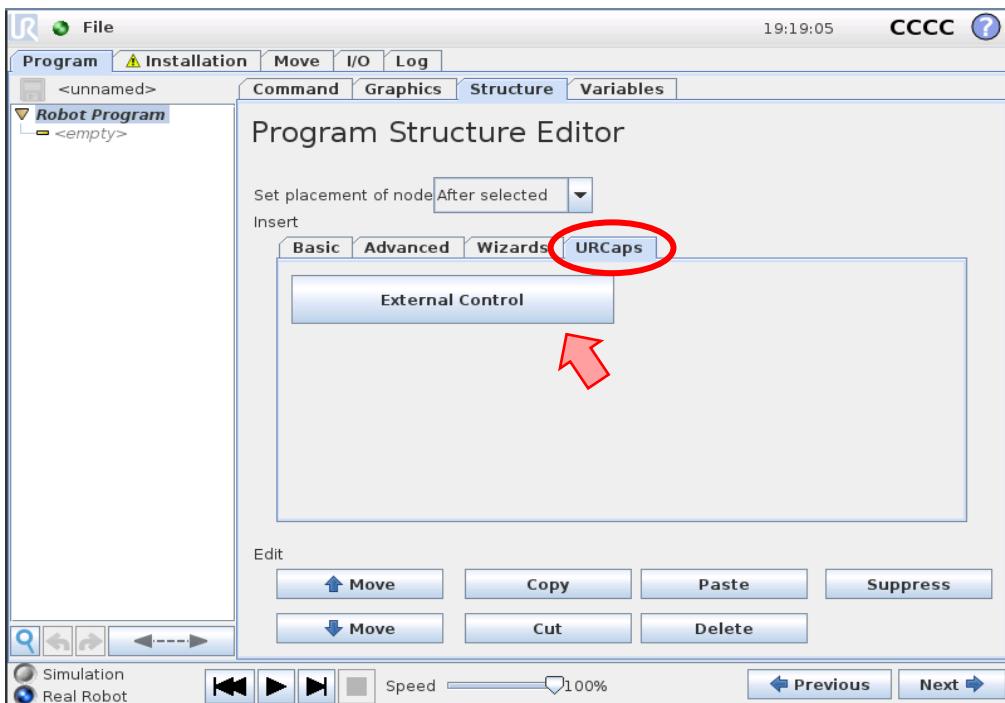


Figure 102. Structure Tab inside Empty Program screen. Source: Created by the author.

9. After that, the driver will appear in a list on the left side of the screen (with your robot IP). Select that control and press the “Play” black button at the bottom of the screen. That will connect the simulation to your control framework.

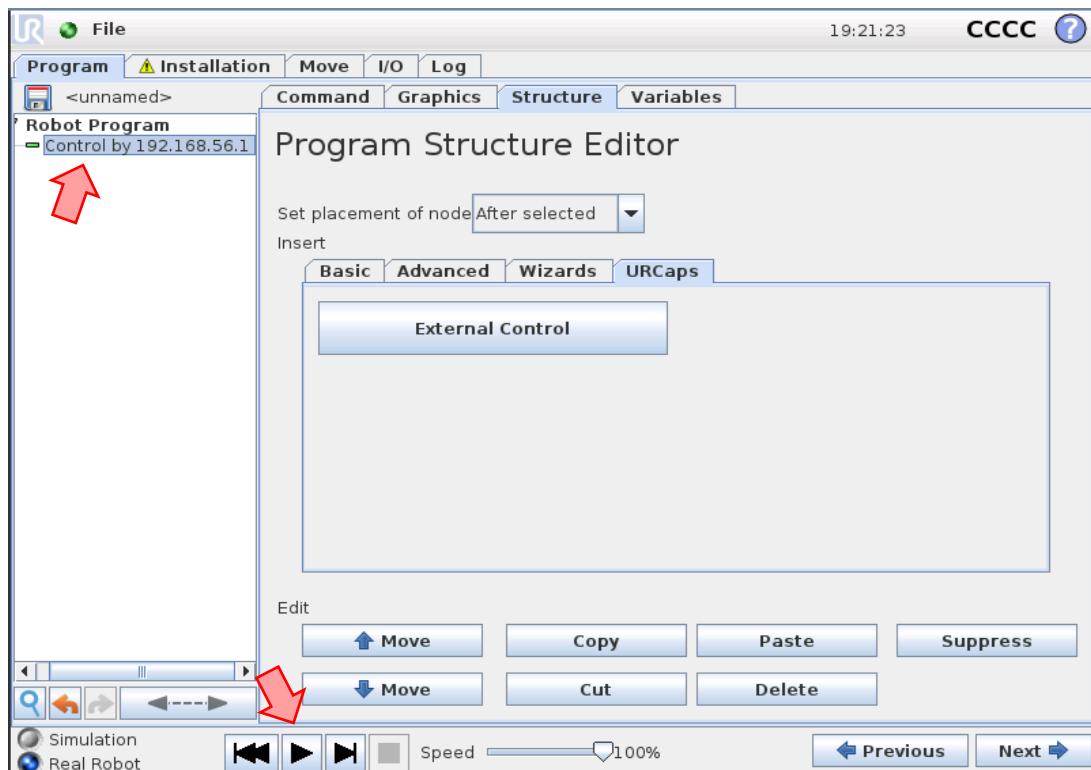


Figure 103. Load and start the simulation by pressing the Play button. Source: Created by the author

10. To see the robot, go to the Graphics Tab.

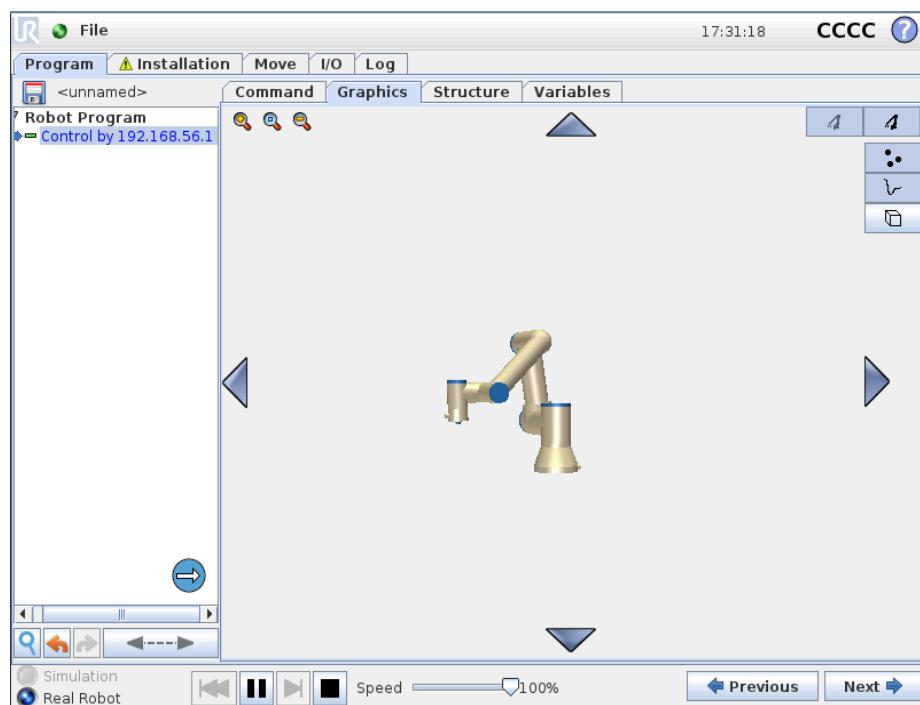


Figure 104. Graphics tab from the URSim Program section. Source: Created by the author.

Your simulation is now ready and connected to the control framework.

STARTING A GAZEBO SIMULATION

First step consists of adding the gazebo_ros2_control package to the system.

In Shell #1

```
sudo apt install ros-humble-gazebo-ros2-control
```

This line will add the package to the underlay workspace, where ROS2 Humble is installed.

Next step, include the Universal_Robots_ROS2_Gazebo_Simulation in the overlay workspace. You want to install it from source, as the description or the driver packages, as you will be doing some modifications to its files, and because that will allow you to study its files and understand them better.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
git clone https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation.git
src/Universal_Robots_ROS2_Gazebo_Simulation
```

Build and source the workspace in order to use it.

In Shell #1

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

To start the simulation, run the following command in a shell after sourcing the workspace.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur3
```

Once the simulation starts, you will notice that the robot links start shaking and the joints break. To solve it modify the ur_macro.xacro file of the UR description package, changing **friction parameter** of all the robot joints to be zero for any simulation or real test, and 100x maximum joint effort for Gazebo simulation. The modified file is commented in Appendix 1 Commented code.

Parameter to modify in ur_macro.xacro

```
friction="${<joint_name>_effort_limit*100.0 if sim_gazebo else 0}"
```

Once the changes are added to the file, build and source the workspace, and run the launch file again.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
colcon build
source install/setup.bash
```

```
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur3
```

To start the simulation using the Motoman 2, the file to be launched would be `ur_sim_moveit.launch.py`.

In Shell #1

```
cd ~/workspace/ros_humble_ur_driver/
source install/setup.bash
ros2 launch ur_simulation_gazebo ur_sim_moveit.launch.py ur_type:=ur3
```

UR3 ROBOT SETUP

The software to be installed is `externalcontrol-1.0.5.urcap`, which can be found inside `~/workspace/ros_ws_humble_ur_driver/Universal_Robots_ROS2_Driver/ur_robod_driver/resources` folder from the overlay workspace in your system (if you installed UR ROS2 Driver as explained in section 7. UR ROS2 Driver) or downloaded from URCap External Control GitHub repository [34]. A minimal PolyScope 5.1 version is required.

The first step to install it is to copy it to the robot's program folder using a USB. Then, follow the next steps:

7. On the welcome screen, click on the Setup Robot settings. Inside the Setup Robot select URCaps to enter the installation screen.
8. Click the plus sign at the bottom to open the file selector. All the URCap files stored inside the robot's program folder will show up. Open the `externalcontrol-1.0.5.urcap` file. Your URCaps view will now show the External Control in the URCap active list, as well as a notification to restart the robot.
9. Reboot the robot and check that the External Control appears in the Installation tab, inside Program Robot menu.

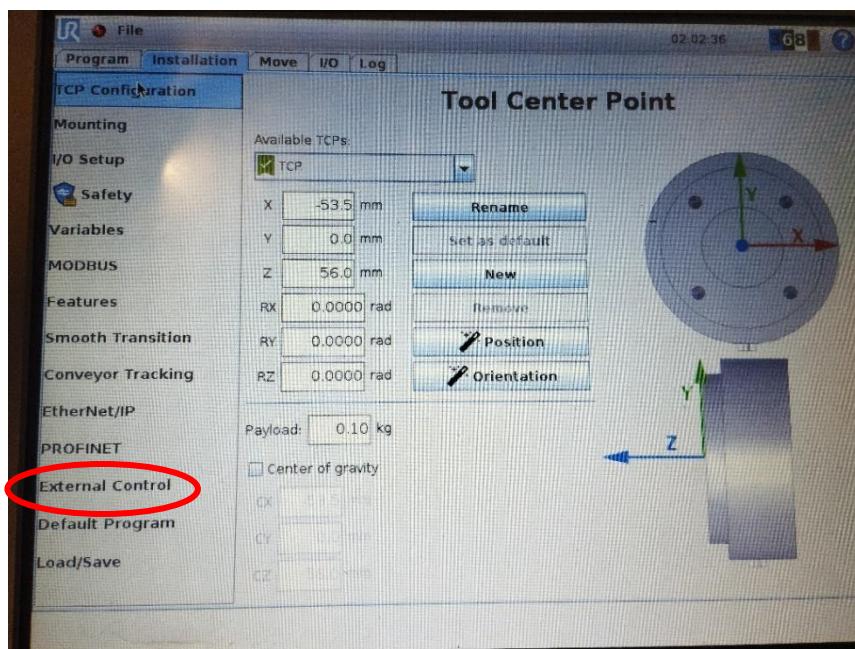


Figure 105. External Control in the Installation tab from Program Robot menu. Source: Created by the author.

10. Set up the IP address of the external PC that will be running the ROS2 driver. Both robot and external PC must be in the same network (if directly connected, network disturbances will minimize). The custom port should be left untouched for the moment.
11. To use URCap, create a new program and insert External Control program node into the program tree.
12. Click on the Command tab, the entered settings should appear inside Installation. Check that they are correct and save the program.

Once the URCap External Control is installed in the UR3, and the program has been created, the connection between robot and PC has to be established.

First, connect the robot and the PC using an Ethernet cable.

Second, set the IP address of the robot in PolyScope. To do it, access the Setup Robot menu and select Network. Then choose Static Address and fill the detailed settings.

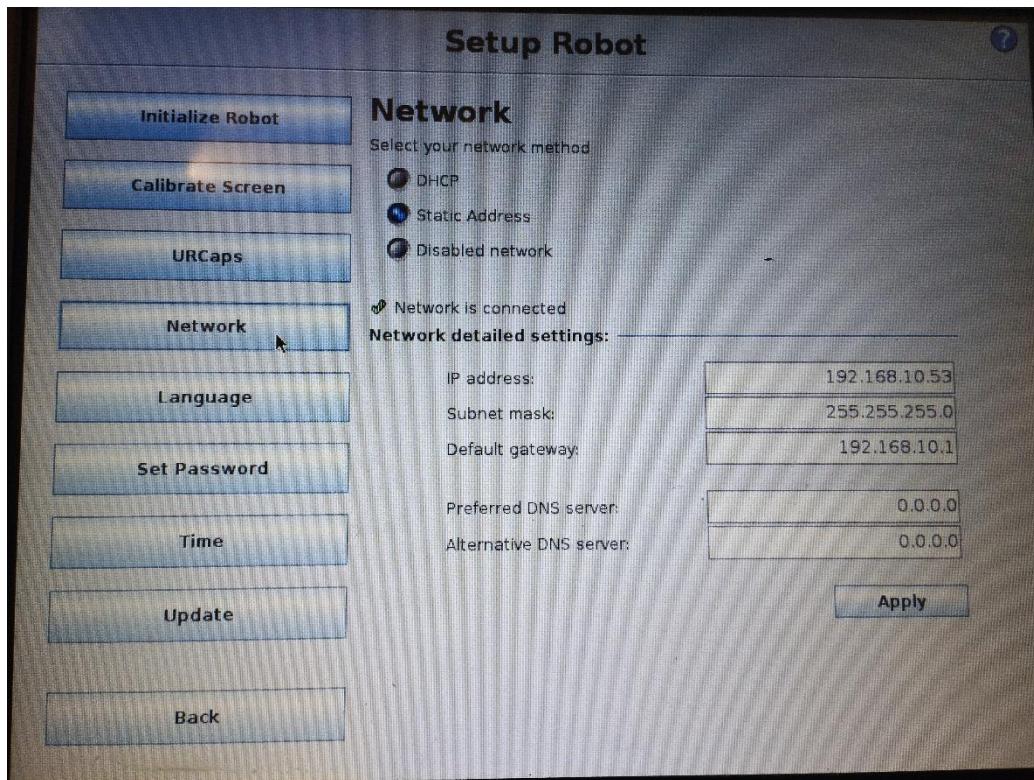


Figure 106. Set Network detailed setting of the UR3. Source: Created by the author.

Third, configure the IP address of the control PC.

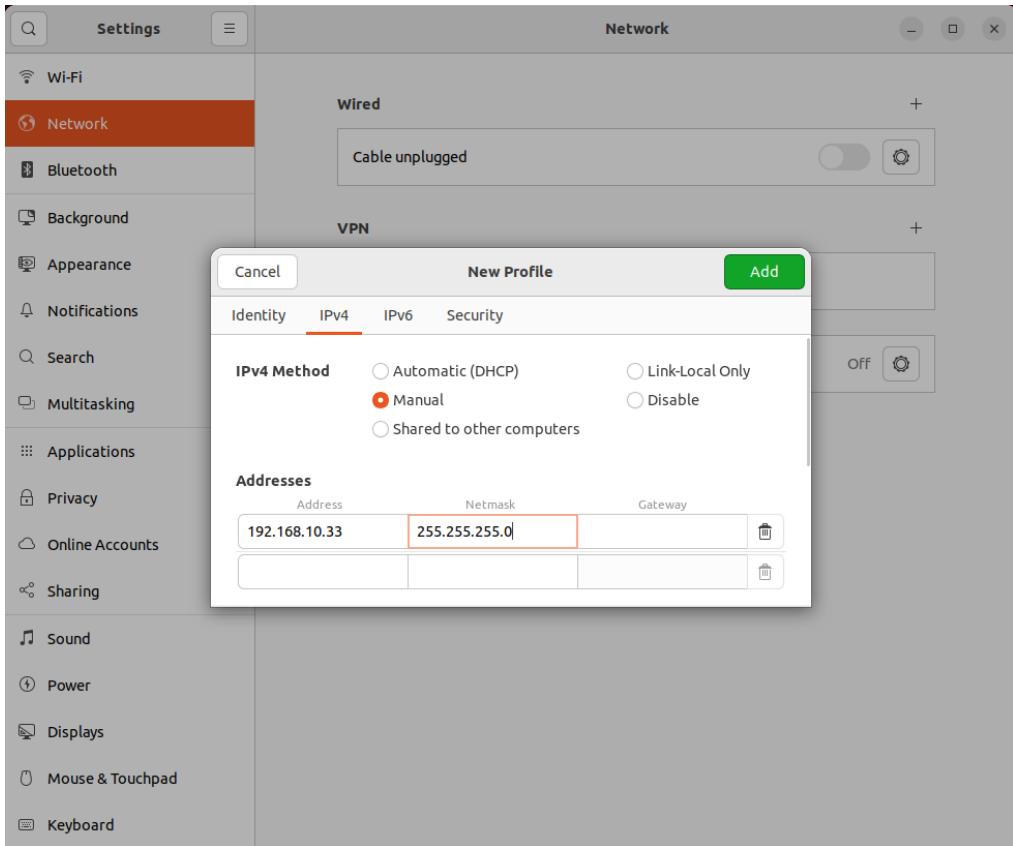


Figure 107. Setting the IP address of the control PC. Source: Created by the author.

Fourth, configure the Host IP and Host name information of URCaps External Control in Installation.

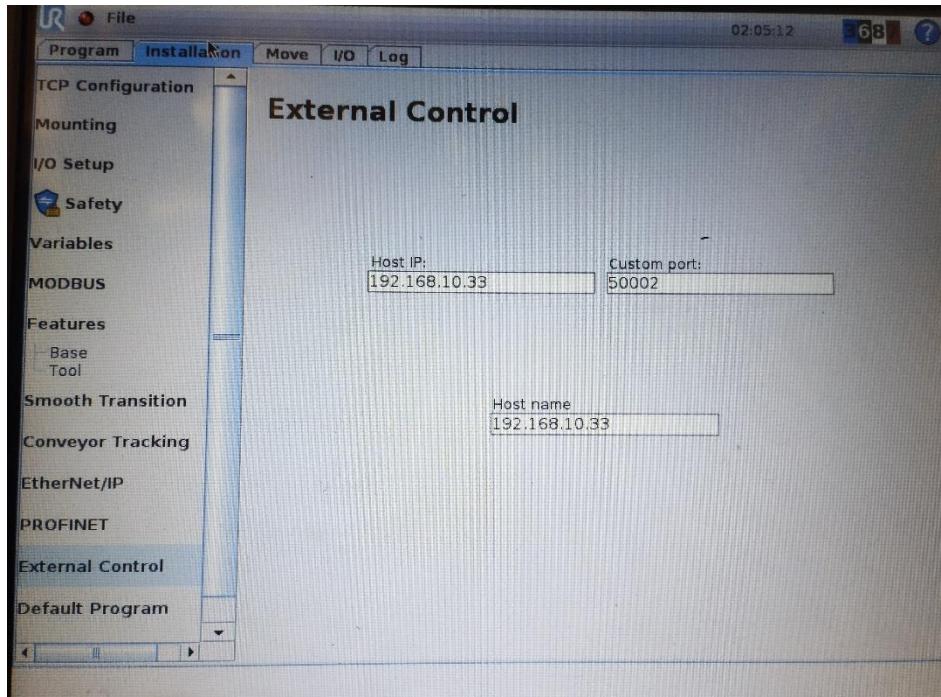


Figure 108. Configure Host IP with control PC address. Source: Created by the author.

The UR3 is now ready to receive ROS2 UR Driver commands from your control PC.

APPENDIX 3. LINUX AND ROS2 BASIC COMMANDS

LINUX BASIC COMMANDS:

1. Move around the directories

cd – go to a specific directory

```
cd /home/user/catkin_ws/src/linux_course_files/move_bb8_pkg/src/
```

cd .. – move to parent folder

```
cd ..
```

cd <folder name> - move to son folder (must be inside the folder you currently are)

```
cd src
```

pwd – check current path

```
pwd
```

ls – see the content of a folder or directory (folder = directory)

```
ls
```

ls -la – see all the content of a folder or directory (included hidden ones)

```
ls -la
```

grep – filter elements

```
ls | grep bb8
```

2. Operate with files and folders

mkdir <name of directory> - make a new folder

```
mkdir my_folder
```

touch <file name> - create a new file (remember to ad extension)

```
touch my_file.txt
```

mv <file/folder we want to move> <destination> – move files and folders (you can move files and folders from one directory to another by adding the whole directory path)

```
mv my_scripts move_bb8_pkg
```

cp <file we want to copy> <name of the new file > - copy files

```
cp my_file.txt my_new_file.txt
```

cp -r <folder we want to copy> <name of the new folder > - copy folders

```
cp -r my_scripts my_new_scripts
```

rm <file to remove> - remove files

```
rm mt_file.txt
```

rm -r <folder to remove> - remove folders

```
rm -r my_scripts
```

3. Edit directories

chmod <groups to assign the permissions><permissions to assign/remove> <file/folder names> - modify permissions of a directory (instead of using group-permission system use numeric system)

read = 4, write = 2, execute = 1.

We will have 3 numbers, first one assigns permissions to owner, second one permission to group, and last one permissions to all other users.

```
chmod 740 move_bb8_square.py
```

4. Launch and edit files

vi <file name> - open the file (press I to edit, press esc to exit edit mode, press :wq to save and exit)

```
vi my_file.txt
```

python <file name> - launch a python program

```
python bb8_keyboard.py
```

./bash_script.sh – execute a bash script (Linux file) (Anything you can run normally on the command line can be put into a script and it will do exactly the same thing)

```
./bash_script.sh
```

ros2 run <package_name> <executable_file> - run executables that are located inside a package

```
ros2 run move_bb8_pkg move_bb8_square.py
```

ros2 run move_bb8_pkg test_process.py & - add the & at the end of a launch to run it in the background

5. Processes

ps faux – check what processes are running (use with grep to check any specific process)

```
ps faux | grep test_process
```

Command Ctrl+C – stop a foreground process

Command Ctrl+Z – suspend a process sending it to the background (when suspended signals sent to the process will be ignored)

bg – resume the execution of a suspended process in the background

```
bg
```

kill <Process ID> - stop a process running in the background

6. Connect to another server (Secure Shell)

ssh <user>@<host> -p <port number> - connect to a host server from your user computer using the indicated port (to see the port where the other server is connected, run `ps faux | grep ssh` and look for the `sudo` line and the 4-digit number at the end of it)

exit – close a ssh session

7. Update or add packages

sudo apt-get update – update all current packages (sudo gives you root permission, so you can do whatever you want)

```
sudo apt-get update
```

sudo apt-get install <name of the package> - install a specific package

```
sudo apt-get install ltris
```

ROS 2 BASIC COMMANDS AND PROCESSES:

1. Package creation

`ros2 pkg create <package_name> --build-type ament_cmake my_package --dependencies <package_dependencies>`

- `ament_cmake` as the `build type` indicates that you are creating a CMake package.
- Package dependencies: indicate other ROS2 packages that your package depends on

After creating a package:

`cd ~/<workspace_name>/` → go to workspace

`colcon build` → build the package to check errors

`source install/setup.bash` → to make ROS find the packages in the workspace

**To compile only one package:

`colcon build --packages-select <package_name>`

2. Topics

Communication method. Receive or send particular information with continuous updates.

`ros2 topic -h` → check options and commands for ros2 topic

`ros2 topic list` → list of available topics

`ros2 topic list | grep '/name'` → list of topics with the word “name”

`ros2 topic echo <topic_name>` → read the information that is being published on a Topic

`ros2 topic info <topic_name>` → get information about a particular Topic

`ros2 topic pub <topic_name> <message_type> <value>` → publish in a particular Topic (to know the message_type, check the topic info)

3. Messages (interfaces)

`ros2 interface show <message>` → show the interface of a certain message (

`ros2 interface proto <message>` → see the full structure of the message (can be used to know how to publish certain value into a particular topic using *ros2 topic pub*)

4. Services

Communication method. Service only provides data when called explicitly by a Client.

`ros2 service list` → list of services

`ros2 service call <service_name> <service_type> <value>` → call a service (send a request for data) (to know the service type and value, use the interfaces commands).

5. Executors

The purpose of an Executor is to coordinate different Nodes and Callbacks by running them in multiple threads to execute in parallel so that Callbacks do not block the execution of the remaining part of the program, which can significantly improve the execution flow and performance.

6. Control Framework

`ros2 control list_controllers` → list of all controllers and their state

```
ros2 control list_controllers
```

`ros2 control set_controller_state <controller_name> inactive` → set controller state to inactive

```
ros2 control set_controller_state forward_position_controller inactive
```

`ros2 control set_controller_state <controller_name> active` → set controller state to active

```
ros2 control set_controller_state forward_position_controller active
```

`ros2 control unload_controller <controller_name>` → shut down the controller (won't appear in list)

```
ros2 control unload_controller forward_position_controller
```

`ros2 control load_controller <controller_name>` → initiates a controller or broadcaster (appears in list) (The controller will be unconfigured, which means that it will not publish to any topic)

```
ros2 control load_controller forward_position_controller
```

`ros2 control set_controller_state <controller_name> configure` → Configures the controller (sets the controller to inactive, after being loaded)

```
ros2 control set_controller_state forward_position_controller configure
```

`ros2 service call /controller_manager/list_controllers`
`controller_manager_msgs/srv/ListControllers` → verify which controllers are loaded

```
ros2 service call /controller_manager/list_controllers
controller_manager_msgs/srv/ListControllers
```

`ros2 service list | grep controller_manager` → to check all services with the controllers using the controller_manager

```
ros2 service list | grep controller_manager
```

`ros2 run controller_manager spawner <my_controller_name> --controller-type <my_controller_type/MyControllerType>` → loads and starts a controller using only one command

```
ros2 run controller_manager spawner my_joint_state_broadcaster --controller-type
joint_state_broadcaster/JointStateBroadcaster
```

Command message to use `forward_position_controller` (angular positions as data).

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/Float64MultiArray "data:
- 2.55
- -1.7
- 1.45
- -1.57
- 0.0
- 0.0" -1
```

Command message to use `forward_velocity_controller` (angular velocities as data).

```
ros2 topic pub /forward_velocity_controller/commands std_msgs/msg/Float64MultiArray "data:
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0" -1
```

Command message to use `joint_trajectory_controller` (angular velocities as data).

```
ros2 action send_goal /joint_trajectory_controller/follow_joint_trajectory
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo
int], points: [{positions: [2.09,-1.57,1.57,-1.57,-1.57,0.0], velocities: [], accelerations: [], time_from_start: {sec: 6, nanosec: 0}}]} }"
```

Command message to use scaled_joint_trajectory_controller (angular velocities as data).

```
ros2 action send_goal /scaled_joint_trajectory_controller/follow_joint_trajectory
control_msgs/action/FollowJointTrajectory "{trajectory: {joint_names:
[shoulder_pan_joint,shoulder_lift_joint,elbow_joint,wrist_1_joint,wrist_2_joint,wrist_3_jo
int], points: [{positions: [2.09,-1.57,1.57,-1.57,-1.57,0.0], velocities: [], accelerations: [], time_from_start: {sec: 6, nanosec: 0}}]}{}"
```

7. Building problems

Remove the build folders and recompile the workspace:

```
cd ~/ros2_ws/
rm -rf build/ install/ log/
colcon build
```

APPENDIX 4. SETUP AN UBUNTU SYSTEM WITH REAL-TIME CAPABILITIES

All this appendix was literally copied from [\[67\]](#).

In order to run the `universal_robot_driver`, we highly recommend to setup a ubuntu system with real-time capabilities. Especially with a robot from the e-Series the higher control frequency might lead to non-smooth trajectory execution if not run using a real-time-enabled system.

You might still be able to control the robot using a non-real-time system. This is, however, not recommended.

To get real-time support into a ubuntu system, the following steps have to be performed:

1. Get the sources of a real-time kernel
2. Compile the real-time kernel
3. Setup user privileges to execute real-time tasks

This guide will help you setup your system with a real-time kernel.

In order to run the `universal_robot_driver`, we highly recommend to setup a ubuntu system with real-time capabilities. Especially with a robot from the e-Series the higher control frequency might lead to non-smooth trajectory execution if not run using a real-time-enabled system.

You might still be able to control the robot using a non-real-time system. This is, however, not recommended.

To get real-time support into a ubuntu system, the following steps have to be performed:

1. Get the sources of a real-time kernel
2. Compile the real-time kernel
3. Setup user privileges to execute real-time tasks

This guide will help you setup your system with a real-time kernel.

Preparing

To build the kernel, you will need a couple of tools available on your system. You can install them using:

```
$ sudo apt-get install build-essential bc ca-certificates gnupg2 libssl-dev wget gawk flex  
bison
```

Before you download the sources of a real-time-enabled kernel, check the kernel version that is currently installed:

```
$ uname -r  
4.15.0-62-generic
```

To continue with this tutorial, please create a temporary folder and navigate into it. You should have sufficient space (around 25GB) there, as the extracted kernel sources take much space. After the new kernel is installed, you can delete this folder again.

In this example we will use a temporary folder inside our home folder:

```
$ mkdir -p ${HOME}/rt_kernel_build  
$ cd ${HOME}/rt_kernel_build
```

All future commands are expected to be run inside this folder. If the folder is different, the **\$** sign will be prefixed with a path relative to the above folder.

Getting the sources for a real-time kernel

To build a real-time kernel, we first need to get the kernel sources and the real-time patch.

First, we must decide on the kernel version that we want to use. Above, we determined that our system has a 4.15 kernel installed. However, real-time patches exist only for selected kernel versions. Those can be found on the [linuxfoundation wiki](#).

In this example, we will select a 4.14 kernel. Select a kernel version close to the one installed on your system.

Go ahead and download the kernel sources, patch sources and their signature files:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/projects/rt/4.14/patch-4.14.139-  
rt66.patch.xz  
$ wget https://cdn.kernel.org/pub/linux/kernel/projects/rt/4.14/patch-4.14.139-  
rt66.patch.sign  
$ wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.14.139.tar.xz  
$ wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.14.139.tar.sign
```

To unzip the downloaded files do:

```
$ xz -dk patch-4.14.139-rt66.patch.xz  
$ xz -d linux-4.14.139.tar.xz
```

Verification

Technically, you can skip this section, it is however highly recommended to verify the file integrity of such a core component of your system!

To verify file integrity, you must first import public keys by the kernel developers and the patch author. For the kernel sources use (as suggested on [kernel.org](#))

```
$ gpg2 --locate-keys torvalds@kernel.org gregkh@kernel.org
```

And for the patch search for a key of the author listed on [linuxfoundation wiki](#).

```
$ gpg2 --keyserver hkp://keys.gnupg.net --search-keys zanussi  
gpg: data source: http://51.38.91.189:11371  
(1) German Daniel Zanussi <german.zanussi@globant.com>  
     4096 bit RSA key 0x537F98A9D92CEAC8, created: 2019-07-24, expires: 2023-07-24  
(2) Michael Zanussi <mzanussi@gmail.com>  
     4096 bit RSA key 0x7C7F76A2C1E3D9EB, created: 2019-05-08  
(3) Tom Zanussi <tzanussi@gmail.com>
```

```

Tom Zanussi <zanussi@kernel.org>
Tom Zanussi <tom.zanussi@linux.intel.com>
    4096 bit RSA key 0xDE09826778A38521, created: 2017-12-15
(4) Riccardo Zanussi <riccardo.zanussi@gmail.com>
    2048 bit RSA key 0xD299A06261D919C3, created: 2014-08-27, expires: 2018-08-27
(expired)
(5) Zanussi Gianni <g.zanussi@virgilio.it>
    1024 bit DSA key 0x78B89CB020D1836C, created: 2004-04-06
(6) Michael Zanussi <zanussi@unm.edu>
Michael Zanussi <mzanussi@gmail.com>
Michael Zanussi <michael_zanussi@yahoo.com>
Michael Zanussi <michael@michaelzanussi.com>
    1024 bit DSA key 0xB3E952DCAC653064, created: 2000-09-05
(7) Michael Zanussi <surfnpk@yahoo.com>
    1024 bit DSA key 0xEB10BBD9BA749318, created: 1999-05-31
(8) Michael B. Zanussi <surfnpk@yahoo.com>
    1024 bit DSA key 0x39EE4EAD7BBB1E43, created: 1998-07-16
Keys 1-8 of 8 for "zanussi". Enter number(s), N(ext), or Q(uit) > 3

```

Now we can verify the downloaded sources:

```

$ gpg2 --verify linux-4.14.139.tar.sign
gpg: assuming signed data in 'linux-4.14.139.tar'
gpg: Signature made Fr 16 Aug 2019 10:15:17 CEST
gpg:           using RSA key 647F28654894E3BD457199BE38DBBDC86092693E
gpg: Good signature from "Greg Kroah-Hartman <gregkh@kernel.org>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:           There is no indication that the signature belongs to the owner.
Primary key fingerprint: 647F 2865 4894 E3BD 4571 99BE 38DB BDC8 6092 693E

$ gpg2 --verify patch-4.14.139-rt66.patch.sign
gpg: assuming signed data in 'patch-4.14.139-rt66.patch'
gpg: Signature made Fr 23 Aug 2019 21:09:20 CEST
gpg:           using RSA key 0x0129F38552C38DF1
gpg: Good signature from "Tom Zanussi <tom.zanussi@linux.intel.com>" [unknown]
gpg:           aka "Tom Zanussi <zanussi@kernel.org>" [unknown]
gpg:           aka "Tom Zanussi <tzanussi@gmail.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:           There is no indication that the signature belongs to the owner.
Primary key fingerprint: 5BDF C45C 2ECC 5387 D50C E5EF DE09 8267 78A3 8521
Subkey fingerprint: ACF8 5F98 16A8 D5F0 96AE 1FD2 0129 F385 52C3 8DF1

```

Compilation

Before we can compile the sources, we have to extract the tar archive and apply the patch:

```

$ tar xf linux-4.14.139.tar
$ cd linux-4.14.139
linux-4.14.139$ xzcat ../patch-4.14.139-rt66.patch.xz | patch -p1

```

Now to configure your kernel, just type:

```
linux-4.14.139$ make oldconfig
```

This will ask for kernel options. For everything else then the **Preemption Model** use the default value (just press Enter) or adapt to your preferences. For the preemption model select **Fully Preemptible Kernel**:

Preemption Model

```

1. No Forced Preemption (Server) (PREEMPT_NONE)
> 2. Voluntary Kernel Preemption (Desktop) (PREEMPT_VOLUNTARY)
3. Preemptible Kernel (Low-Latency Desktop) (PREEMPT_LL) (NEW)
4. Preemptible Kernel (Basic RT) (PREEMPT_RT) (NEW)
5. Fully Preemptible Kernel (RT) (PREEMPT_RT_FULL) (NEW)
choice[1-5]: 5

```

Now you can build the kernel. This will take some time...

```
linux-4.14.139$ make -j `getconf _NPROCESSORS_ONLN` deb-pkg
```

After building, install the `linux-headers` and `linux-image` packages in the parent folder (only the ones without the `-dbg` in the name)

```
linux-4.14.139$ sudo apt install ../linux-headers-4.14.139-rt66_*.deb ../linux-image-4.14.139-rt66_*.deb
```

Setup user privileges to use real-time scheduling

To be able to schedule threads with user privileges (what the driver will do) you'll have to change the user's limits by changing `/etc/security/limits.conf` (See [the manpage](#) for details)

We recommend to setup a group for real-time users instead of writing a fixed username into the config file:

```
$ sudo groupadd realtime
$ sudo usermod -aG realtime $(whoami)
```

Then, make sure `/etc/security/limits.conf` contains:

```
@realtime soft rtprio 99
@realtime soft priority 99
@realtime soft memlock 102400
@realtime hard rtprio 99
@realtime hard priority 99
@realtime hard memlock 102400
```

Note: You will have to log out and log back in (Not only close your terminal window) for these changes to take effect. No need to do this now, as we will reboot later on, anyway.

Setup GRUB to always boot the real-time kernel

To make the new kernel the default kernel that the system will boot into every time, you'll have to change the grub config file inside `/etc/default/grub`.

Note: This works for ubuntu, but might not be working for other linux systems. It might be necessary to use another menuentry name there.

But first, let's find out the name of the entry that we will want to make the default. You can list all available kernels using:

```
$ awk -F\` '/menuentry |submenu / {print $1 $2}' /boot/grub/grub.cfg
menuentry Ubuntu
submenu Advanced options for Ubuntu
```

```

menuentry Ubuntu, with Linux 4.15.0-62-generic
menuentry Ubuntu, with Linux 4.15.0-62-generic (recovery mode)
menuentry Ubuntu, with Linux 4.15.0-60-generic
menuentry Ubuntu, with Linux 4.15.0-60-generic (recovery mode)
menuentry Ubuntu, with Linux 4.15.0-58-generic
menuentry Ubuntu, with Linux 4.15.0-58-generic (recovery mode)
menuentry Ubuntu, with Linux 4.14.139-rt66
menuentry Ubuntu, with Linux 4.14.139-rt66 (recovery mode)
menuentry Memory test (memtest86+)
menuentry Memory test (memtest86+, serial console 115200)
menuentry Windows 7 (on /dev/sdc2)
menuentry Windows 7 (on /dev/sdc3)

```

From the output above, we'll need to generate a string with the pattern "submenu_name>entry_name". In our case this would be

```
"Advanced options for Ubuntu>Ubuntu, with Linux 4.14.139-rt66"
```

The double quotes and no spaces around the > are important!

With this, we can setup the default grub entry and then update the grub menu entries. Don't forget this last step!

```
$ sudo sed -i 's/^GRUB_DEFAULT=.*/GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 4.14.139-rt66"/' /etc/default/grub
$ sudo update-grub
```

Reboot the PC

After having performed the above-mentioned steps, reboot the PC. It should boot into the correct kernel automatically.

Check for preemption capabilities

Make sure that the kernel does indeed support real-time scheduling:

```
$ uname -v | cut -d" " -f1-4
#1 SMP PREEMPT RT
```

Optional: Disable CPU speed scaling

Many modern CPUs support changing their clock frequency dynamically depending on the currently requested computation resources. In some cases this can lead to small interruptions in execution. While the real-time scheduled controller thread should be unaffected by this, any external components such as a visual servoing system might be interrupted for a short period on scaling changes.

To check and modify the power saving mode, install cpufrequtils:

```
$ sudo apt install cpufrequtils
```

Run `cpufreq-info` to check available "governors" and the current CPU Frequency (`current CPU frequency is XXX MHZ`). In the following we will set the governor to "performance".

```
$ sudo systemctl disable ondemand
```

```
$ sudo systemctl enable cpufrequtils
$ sudo sh -c 'echo "GOVERNOR=performance" > /etc/default/cpufrequtils'
$ sudo systemctl daemon-reload && sudo systemctl restart cpufrequtils
```

This disables the `ondemand` CPU scaling daemon, creates a `cpufrequtils` config file and restarts the `cpufrequtils` service. Check with `cpufreq-info`.