DEVELOPMENT OF THE CONTROL OF A ROBOTIC ARM USING ROS2

IBRAHIM MALLI
MSc Mechanical Engineering

Szent Istvan Campus, Gödöllő

2023

Hungarian University of Agriculture and Life Science
Szent István Campus
MSc Mechanical Engineering Course

DEVELOPMENT OF THE CONTROL OF A ROBOTIC ARM USING ROS2

Primary Supervisor: **Tóth János**
Assistant Professor

Author: **Ibrahim Malli**
BLD9OB

Institute/Department: **Institute of Technology**
MSc Mechanical Engineering

Szent Istvan Campus, Gödöllő

2023

HUNGARIAN UNIVERSITY OF AGRICULTURE AND LIFE SCIENCES

INSTITUTE OF TECHNOLOGY

INSTITUTE OF TECIINOLOGY MECIIANICAL ENGINEERING (MSc)

THESIS

worksheet for

IBRAHIM MALLI (BLD9OB)

(MSc) student

Entitled:

Development of the Control of a Robotic Arm Using ROS2

Task description: The robotic arm must be capable to move to the predefined positions in a virtual environment.

Department: MSc Mechanical Engineering

Supervisor: Dr. Tóth János, Assistant Professor, MATE, Institute of Technology

Submission deadline: 3 May 2023.

Gödöllő, 20 January 2023

Approved		**Received**
(head of department)	(host course leader)	(student)

ABSTRACT

Development of the Control of a Robotic Arm Using ROS2

Ibrahim Malli

Course, level of education: MSc Mechanical Enginering

Host Department/Institute: Institute of Technology

Primary thesis advisor: Dr. Tóth János, Assistant Professor, Institute of Technology, MSc Mechanical Enginering

In this study, it is aimed to develop the control of a robotic arm using ROS2. The collaborative robot arm, developed using the ROS 2 MoveIt 2 repository and the RViz visualization tool and ROS 2 Development Studio, is intended to be capable to move in the Gazebo simulation toolbox.

ROS is the most widely used middleware in robotics. It is the interface that allows the robot to process the data it receives from the outside world through sensors and send it back to the robot as a command. It allows to use different languages (C++, Python) on the same robot. The primary goal of ROS is to support code reuse in robotics research and development which enables the robotic arm to be developed more easily thanks to common repository and documentations, facilitates robot coding in C++ language.

Key Words: Robotic Arm, Robot Operating System-ROS, Programming

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

Some abbreviations and symbols used in this study are presented below with their explanations.

Abbreviations	Explanations
DOF	Degrees of freedom
BSD	Berkeley Software Distribution
ROS	Robot Operating System
LTS	Long Term Support
GPS	Global Positioning System
RIA	The Robotics Industries Association
SLAM	Simultaneous Localization and Mapping
API	Application Programming Interface.
IEEE	Institute of Electrical and Electronics Engineers
LTS	Long Term Support
EOL	End-of-life
SDK	Software development kit
UTF-8	Unicode Transformation Format 8-bit
RViz	ROS Visualization
SCARA	Selective Compliance Articulated Robot Arm
AIST	Advanced Industrial Science and Technology
GUI	Graphical User Interface
PMSM	Permanent Magnet Synchronous Machine

1 INTRODUCTION

The term "Robot" refers to an electromechanical device with numerous degrees of freedom (DOF) that humans can program to perform various tasks. [1] The term “Industrial Robot” The Robotics Industries Association (RIA) defines robot in the following way: “An industrial robot is a programmable, multi-functional manipulator designed to move materials, parts, tools, or special devices through variable programmed motions for the performance of a variety of tasks.” To fulfil their purposes, many robots are required to interact with their environment, and the world around them. Sometimes they are required to move or reorient objects from their environments without direct contact by human operators. Unlike the Body/frame and the Control System, manipulators are not integral to a robot, for instance, a robot can exist without a manipulator.

A robot arm in a robotic system represents the end-effector that simulates the human arm to carry out tasks such as picking and placing. For years, robotics has been used in the industry. A robotic system can also fulfil its duty in agriculture to complete some acceptable jobs by researching the parallels between agriculture and industries. The study of robotics is to combine and simulate certain aspects of human body function by implementing mechanisms, sensors, actuators, and computers.

Industrial robot and manipulator systems are machines that can be reprogrammed, move objects, workpieces, tools according to the programmed software, and perform the operation. Industrial robot and manipulator systems can be used for various purposes and in different application areas. People wanted to design robot and manipulator systems with factors such as reducing workload, increasing product quality, and speeding up the production process. For this reason, a wide variety of tasks and applications can be made in industry for robots.

In recent years, robot studies have gained importance industrially, especially with the initiatives of Asian and American countries. Robots have market value not only as a manufacturing tool, but also as a stand-alone product. This means opening a new market based on qualified workforce and high technology.

In today's industry conditions and competitive market, the perfection, quality, and efficiency of the work is the biggest factor. Under these working conditions, the use of robots and manipulators is undeniable. Thus, factories that have manipulators gradually increase the difference between them and their competitors. Robot and manipulator systems have saved people from a great deal of work by working in places such as paint, welding, and spot works, which are unsuitable for human health.

1.1 Robotic Arm

A robotic arm is a type of mechanical arm, usually programmable, with similar functions to a human arm; the arm may be the sum total of the mechanism or may be part of a more complex robot. The links of such a manipulator are connected by joints allowing either rotational motion (such as in an articulated robot) or translational (linear) displacement. The links of the manipulator can be considered to form a kinematic chain. The terminus of the kinematic chain of the manipulator is called the end effector and it is analogous to the human hand. However, the term "robotic hand" as a synonym of the robotic arm is often proscribed. [2]

Typical industrial robot arm includes a series of joints, articulations and manipulators that work together to closely resemble the motion and functionality of a human arm (at least from a purely mechanical perspective). A programmable robotic arm can be a complete machine in and of itself, or it can function as an individual robot part of a larger and more complex piece of equipment.

A great many smaller robotic arms used in countless industries and workplace applications today are benchtop-mounted and controlled electronically. Larger versions might be floor-mounted, but either way they tend to be constructed from sturdy and durable metal (often steel or cast iron), and most will feature between 4-6 articulating joints. Again, from a mechanical perspective, the key joints on a robotic arm are designed to closely resemble the main parts of its human equivalent - including the shoulder, elbow, forearm and wrist.

Such is the speed and power that industrial robot arms can work at, there's a pressing need to be extremely safety-conscious when programming and using them. However, when deployed appropriately, they can vastly increase production rates and accuracy of placement and picking tasks, as well as performing heavy-duty lifting and repositioning functions that would be impossible even for groups of multiple human workers to carry out at any sort of pace.

As technology has advanced and the manufacturing costs of robotic components has fallen over the years, the past decade or so has seen a very rapid expansion in the availability and affordability of robots and robotic arms across a very wide range of industries. This means that they're far more commonly encountered in smaller-scale operations than they once were, because they're no longer only an economically viable option for large-scale production lines outputting very high volumes of product.

1.2 Robotic Arm Types

1.2.1 Cartesian Robot / Gantry Robot

Used for pick and place work, application of sealant, assembly operations, handling machine tools and arc welding. It is a robot whose arm has three prismatic joints, whose axes are coincident with a Cartesian coordinator.

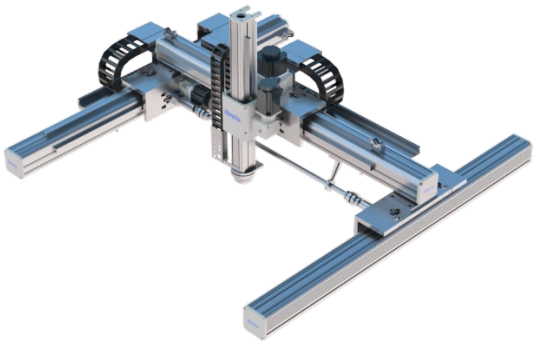


Figure 1. Gantry robot GR-1750 series [3]

1.2.2 Collaborative Robot / Cobot

Cobot applications contrast with traditional industrial robot applications in which robots are isolated from human contact. Cobot has a large variety of applications such as: Commercial Application, Robotic Research, Dispensing, Material Handling, Assembly, Finishing, Quality Inspection. Cobot safety may rely on lightweight construction materials, rounded edges, and the inherent limitation of speed and force, or on sensors and software that ensures safe behaviour.

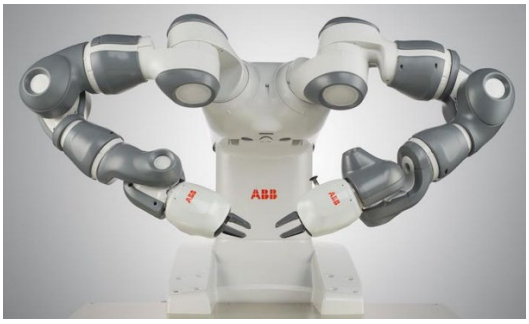


Figure 2. YuMi, a collaborative robot developed by ABB. [4]

1.2.3 Cylindrical Robot

Used for assembly operations, handling at machine tools, spot welding, and handling at die casting machines. It is a robot whose axes form a cylindrical coordinate system.

Figure 3. A cylindrical robot arm [5]

1.2.4 Spherical Robot / Polar Robot

Used for handling machine tools, spot welding, die casting, fettling machines, gas welding and arc welding. It is a robot whose axes form a polar coordinate system.

Figure 4. Spherical Robot [6]

1.2.5 SCARA Robot

Used for pick and place work, application of sealant, assembly operations and handling machine tools. This robot features two parallel rotary joints to provide compliance in a plane.

Figure 5. SCARA-type robot AR-F500HCs [3]

1.2.6 Articulated Robot

Used for assembly operations, diecasting, fettling machines, gas welding, arc welding and spray-painting. It is a robot whose arm has at least three rotary joints.

Figure 6. Kuka Articulated robot KR 700 PA [21]

1.2.7 Parallel Robot

One use is a mobile platform handling cockpit flight simulator. It is a robot whose arms have concurrent prismatic or rotary joints.

Figure 7. Omron iX3 Parallel Robot [7]

1.2.8 Anthropomorphic / Humanoid Robot

It is shaped in a way that resembles a human hand, i.e. with independent fingers and thumbs.

Figure 8. Valkyrie from NASA. [22]

1.3 Robot Software Platform

Recently, within the robotics industry, platforms have been gaining much attention. Platforms are divided into two: software and hardware.

Software platforms include tools used to develop robotic programs such as hardware abstraction, low-level device control, sensing, recognition, SLAM (Simultaneous Localization and Mapping), navigation, manipulation, package management, libraries, debugging, and development tools. Meanwhile, hardware platforms focus on tangible, commercial components and systems or products, such as mobile robots, drones, and humanoids. [8]

Notably, hardware abstraction occurs in tandem with software platforms, allowing application programs to be developed using a software platform even without prior knowledge of hardware. This is similar to how we can create mobile apps without knowing the hardware configuration or specifications of the most recent smartphone.

In other words, software platforms have allowed many people to contribute to robot development, and robot hardware is being designed according to the interface provided by software platforms.

Major software platforms include the Robot Operating System (ROS), the Japanese Open Robotics Technology Middleware (OpenRTM), the European real-time control centred OROCOS, and the Korean OPRoS. Although their names differ, the fundamental reason for introducing robot software platforms is that there are far too many different types of robot software, and their complexities are causing numerous issues.

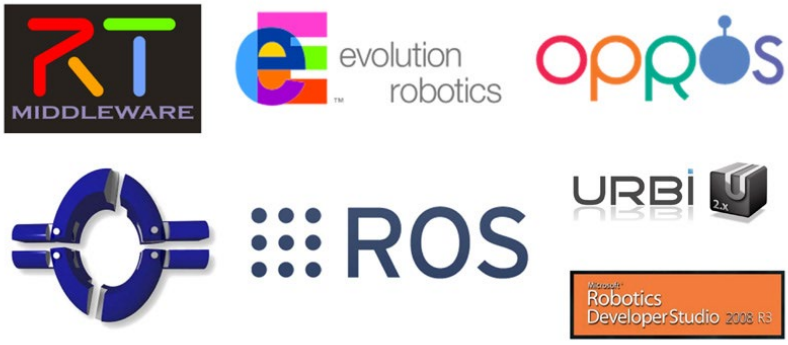


Figure 9. Various Robot Software Platforms [8]

- ERSP11 Evolution Robotics Software Platform, Evolution Robotics - Europe
- ROS Robot Operating System, Open Robotics12 - U.S.
- OpenRTM National Institute of Adv. Industrial Science and Technology (AIST) - Japan
- OROCOS Europe
- OPRoS ETRI, KIST, KITECH, Kangwon National University - South Korea
- NAOqi OS13 SoftBank and Aldebaran - Japan and France

Aside from these, there are also Player, YARP, MARIE, URBI, CARMEN, Orca and MOOS.

As a result, robot researchers from all over the world are working together to find a solution. Robot Operating System is the most widely used robot software platform (ROS). Many of the companies leading the industrial robot field that have joined this consortium are working to solve some of the industry's most difficult and newly emerging problems, such as automation, sensing, and collaborative robot. Using a common platform, particularly a software platform, promotes collaboration to solve difficult problems and increase efficiency.

For instance, when implementing a function that helps a robot to recognize its surrounding situation, the diversity of hardware and the fact that it is directly applied in real-life can be a burden. Some tasks may be considered easy for humans, but researchers in a college laboratory or company are too difficult to deal with robots to perform a lot of functions such as sensing, recognition, mapping, and motion planning.

However, it would be a different story if professionals from around the world shared their specialized software to be used by others. For example, the robotics company Robotbase, which drew attention in the social funding KickStarter and CES2015, recently developed the Robotbase Personal Robot and successfully launched it through a social funding. In the case of Robotbase, they focused on their core technology, which is face recognition and object recognition, and for their mobile robot they used the mobile robot base from Yujin Robot which supports ROS, for the actuator they used ROBOTIS Dynamixel, and for the obstacle recognition, navigation, motor drive, etc. they used the public package of ROS.

Another example can be found in the ROS Industrial Consortium (ROS-I). Many of the companies leading the industrial robot field participate in this consortium and are solving some of the newly emerging and difficult problems from the industrial robot field one by one, such as in automation, sensing, and collaborative robot. Using a common platform, especially a software platform, is proved to be promoting collaboration to solve problems that were previously difficult to tackle and increasing efficiency. [8]

ROS does not necessitate the repeated development of existing systems and programs. Nonetheless, it is simple to convert a non-ROS system to a ROS system by inserting a few standardized codes. Furthermore, ROS provides a variety of commonly used tools and software. It enables users to concentrate on the features they are interested in or want to contribute to, reducing development and maintenance time.

2 ROS: Robot Operating System

ROS: Robot Operating System is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

The ROS Wiki defines ROS as above. In other words, ROS includes hardware abstraction layer similar to operating systems. However, unlike conventional operating systems, it can be used for numerous combinations of hardware implementation. Furthermore, it is a robot software platform that provides various development environments specialized for developing robot application programs.

Robotic Operating System (ROS) was introduced to the world with an article presented at the IEEE International Conference on Robotics and Automation in Kobe, Japan, during May 12 - 17, 2009. ROS was the first standard operating system for robot development. Moreover, a free, open-source, inherently flexible system saved robot developers the daunting hassle of spending time developing an operating system from scratch. With open-source robotics now becoming the norm, development efforts have gained momentum with the support of the masses.

ROS stands for Robot Operating System and is a software that allows controlling robots. Even though the operating system is mentioned in its name, it can be called an open-source interface software that enables communication between human and robot.

ROS is the most widely used software in robots. It is the interface that allows the robot to process the data it receives from the outside world through sensors and send it back to the robot as a command. The working logic here works in the broadcast/subscriber logic, that is, in a simpler sense, the sender-receiver logic. This communication between the computer and root is provided by topics and messages. It is an open source (BSD) system. It can be described as a standalone programming language. It allows to use different languages (Java, Lisp, C++, Python) on the same

robot. The Indigo version released in 2014 is supported until 2019, and the Kinetic Kame (LTS) version released in 2016 is supported until 2021. [9]

Figure 10. Robot Operating System Logo [9]

The primary goal of ROS is to support code reuse in robotics research and development. It has been quickly adopted by many robotics research institutions and companies as their standard development framework. The list of robots using ROS is quite extent and it includes platforms from diverse domains, ranging from aerial and ground robots to humanoids and underwater vehicles. To name some, the humanoid robots PR-2 and REEM-C, ground robots such as the ClearPath platforms and aerial platforms such as the ones from Ascending Technologies are fully developed in ROS. Furthermore, a quite extent list of popular sensors for robots is already supported in ROS. To name some, sensors such as Inertial Measurement Units, GPS receivers, cameras and lasers can already be accessed from the drivers available on the ROS system. [10]

However, ROS supports soft real-time applications. ROS has a repository in which developers can use code that has been already written for robot components like motion planners and vision systems. In addition, it can be integrated easily with other popular open-source software libraries such as OpenCV, Point Cloud Library, Gazebo simulator, etc. As ROS is Linux based software, it cannot guarantee the hard real-time properties of a system.

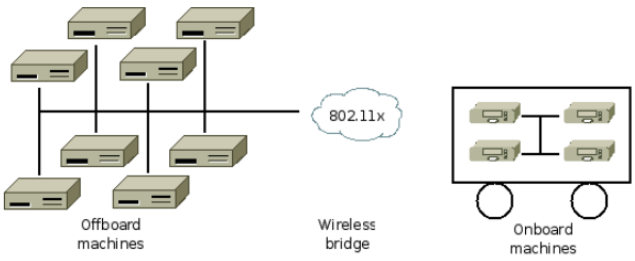


Figure 11. A typical ROS network configuration [13]

ROS implements several different styles of communication: asynchronous, synchronous and data storage. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure.

2.1 Why Should We Use the ROS

ROS (Robot Operating System) is an open-source software development kit for robotics applications. ROS offers a standard software platform to developers across industries that will carry them from research and prototyping all the way through to deployment and production. Don't reinvent the wheel. Create something new and do it faster and better by building on ROS! [9]

First is the reusability of the program: A user can focus on the feature that the user would like to develop, and download the corresponding package for the remaining functions. At the same time, they can share the program that they developed so that others can reuse it. As an example, it is said that for NASA to control their robot Robonaut2 used in the International Space Station, they not only used programs developed in-house but also used ROS, which provides various drivers for multi-platforms, and OROCOS, which supports real-time control, message communication restoration and reliability, in order to accomplish their mission in outer space. The Robotbase above is another example of thoroughly implemented reusable programs.

Second is that ROS is a communication-based program: Often, in order to provide a service, programs such as hardware drivers for sensors and actuators and features such as sensing, recognition and operating are developed in a single frame. However, in order to achieve the reusability of robot software, each program and feature is divided into smaller pieces based on its function. This is called componentization or modularization according to the platform. Data should be exchanged among nodes(a process that performs computation in ROS) that are divided into units of minimal functions, and platforms have all necessary information for exchanging of data among nodes. The network programming, which is greatly useful in remote control, becomes possible when communication among node is based on network so that nodes are not restricted by hardware. The concept of network connected minimal functions is also applied to Internet of

Things (IOT), so ROS can replace the IoT platforms. It is remarkably useful for finding errors because programs that are divided into minimal functions can be debugged separately.

Third is the support of development tools: ROS provides debugging tools, 2D visualization tool (rqt, rqt is a software framework of ROS that implements the various GUI tools in the form of plugins) and 3D visualization tool (RViz) that can be used without developing necessary tools for robot development. For example, there are many occasions where a robot model needs to be visualized while developing a robot. Simply matching the predefined message format allows users to not only check the robot's model directly, but also perform a simulation using the provided 3D simulator (Gazebo). The tool can also receive 3D distance information from recently spotlighted Intel RealSense or Microsoft Kinect and easily convert them into the form of point cloud, and display them on the visualization tool. Apart from this, it can also record data acquired during experiments and replay them whenever it is necessary to recreate the exact experiment environment. As shown above, one of the most important characteristics of ROS is that it provides software tools necessary for robot development, which maximizes the convenience of development.

Fourth is the active community: The robot academic world and industry that have been relatively closed until now are changing in the direction of emphasizing collaboration as a result of these previously mentioned functions. Regardless of the difference in individual objectives, collaboration through these software platforms is actually occurring. At the center of this change, there is a community for open source software platform. In case of ROS, there are over 5,000 packages that have been voluntarily developed and shared as of 2017, and the Wiki pages that explain their packages are exceeding 18,000 pages by the contribution of individual users. Moreover, another critical part of the community which is the Q&A has exceeded 36,000 posts, creating a collaboratively growing community. The community goes beyond discussing the instructions, and into finding necessary components of robotics software and creating regulations thereof. Furthermore, this is progressing to a state where users come together and think of what robot software should entail for the advancement of robotics and collaborate in order to fill the missing pieces in the puzzle.

Fifth is the formation of the ecosystem: The previously mentioned smartphone platform revolution is said to have occurred because there was an ecosystem that was created by software platforms such as Android or iOS. This type of progression is likewise underway for the robotic field. In the beginning, every kind of hardware technology was overflowing, but there was no operating system to integrate them. Various software platforms have developed and the most esteemed platform among them, ROS, is now shaping its ecosystem. It is creating an ecosystem that everyone — hardware developers from the robotic field such as robot and sensor companies, ROS development operational team, application software developers, and users — can be happy with it. Although the beginning may yet be marginal, when looking at the increasing number of users and robot-related companies and the surge of related tools and libraries, we can anticipate a lively ecosystem in the near future. [8]

2.2 Meta-Operating System

Operating Systems (OS) for general purpose computers include Windows (XP, 7, 8, 10), Linux (Linux Mint, Ubuntu, Fedora, Gentoo) and Mac (OS X Mavericks, Yosemite, El Capitan). For smartphones, there are Android, iOS, Symbian, RiMO, Tizen, etc.

Figure 12. ROS as a Meta-Operating System [8]

As shown in Figure 13. ROS data communication is supported not only by one operating system, but also by multiple operating systems, hardware, and programs, making it highly suitable for robot development where various hardware are combined.

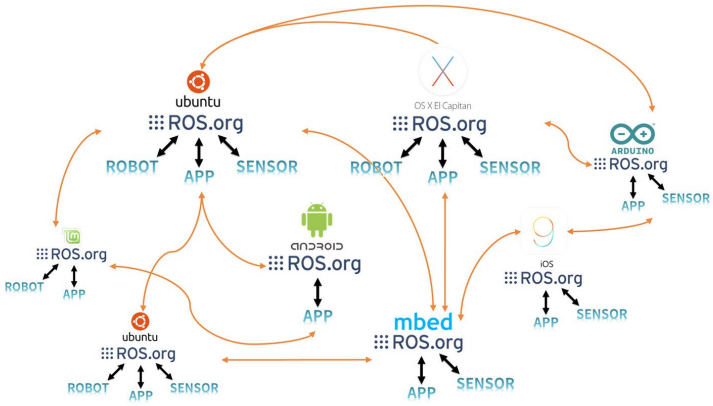


Figure 13. ROS Multi-Communication [8]

2.3 Objectives of ROS

One of the most frequently asked questions received over the years in ROS-related seminars is to compare ROS with other robot software platforms (OpenRTM, OPRoS, Player, YARP, Orocos, CARMEN, Orca, MOOS, Microsoft Robotics Studio). A simple comparison with these platforms may be possible, but the comparison is not meaningful because they each have different purposes. As a user of ROS, the goal of ROS is to build the development environment that allows robotic software development to collaborate on a global level. That is to say, ROS is focused on maximizing code reuse in the robotics research and development, rather than orienting towards the so-called robot software platform, middleware, and framework. To support this, ROS has the following characteristics.

- **Distributed process:** It is programmed in the form of the minimum units of executable processes (nodes), and each process runs independently and exchanges data systematically.
- **Package management:** Multiple processes having the same purpose are managed as a package so that it is easy to use and develop, as well as convenient to share, modify, and redistribute.

- **Public repository:** Each package is made public to the developer's preferred public repository (e.g., GitHub) and specifies their license.
- **API:** When developing a program that uses ROS, ROS is designed to simply call an API and insert it easily into the code being used. In the source code introduced in each chapter, you will see that ROS programming is not much different from C++ and Python.
- **Supporting various programming languages:** The ROS program provides a client library2 to support various programming languages. The library can be imported in programming languages that are popular in the robotics field such as Python, C++, and Lisp as well as languages such as JAVA, C#, Lua, and Ruby. In other words, you can develop a ROS program using a preferred programming language. [11]

These characteristics of ROS have allowed users to establish an environment where it is possible to collaborate on robotics software development on a global level. Reusing a code in robotics research and development is becoming more common, which is the ultimate goal of ROS.

2.4 Components of ROS

As shown in Figure 2-3, ROS consists of a client library to support various programming languages, a hardware interface for hardware control, communication for data transmission and reception, the Robotics Application Framework to help create various Robotics Applications, the Robotics Application which is a service application based on the Robotics Application Framework, Simulation tools which can control the robot in a virtual space, and Software Development Tools.

ROS supports several Client Libraries, though the main supported libraries are C++ roscpp and rospy. There are also numerous ROS stacks and packages that provide higher-level functionality. [12]

APIs:

ROS: functionality available via ROS topics and services

C++: functionality available in C++ libraries

Python: functionality available in Python modules/packages

Table 1. Support for higher-level functionality in various languages [12]

API	ROS	C++	Python
ROS	ROS	roscpp	rospy
Basic Datatypes	common_msgs	common_msgs	common_msgs
Manipulating message streams	topic_tools	message_filters	message_filters
Drivers	joystick_drivers, camera_drivers, laser_drivers, audio_common (aka sound_drivers), imu_drivers	joystick_drivers, camera_drivers, laser_drivers, audio_common, imu_drivers	
Driver Implementation	driver_common	driver_common	
Filtering data		filters	
3D processing	laser_pipeline, perception_pcl	laser_pipeline, perception_pcl	
Image processing		image_common, image_pipeline, vision_opencv	vision_opencv
Transforms/Coordinates		tf, tf_conversions, robot_state_publisher	tf, tf_conversions
Actions	actionlib	actionlib	actionlib
Executive/Task Manager	executive_smach		executive_smach
Navigation	navigation	via actionlib	via actionlib
Simulation (2D)	simulator_stage	simulator_stage	
Simulation (3D)	simulator_gazebo	simulator_gazebo	
Robot Model		robot_model	
Realtime Controllers	pr2_controller_manager	pr2_controller_interface, realtime_tools	
Motion planning (arms)	ompl, chomp_motion_planner, sbpl	actionlib through move_arm	actionlib through move_arm
Humanoid walk	walk_msgs	walk_interfaces	

2.5 The ROS Ecosystem

The term 'Ecosystem' is often mentioned in the smartphone market after the advent of various operating systems such as Android, iOS, Symbian, RiMO, and Bada. The ecosystem refers to the structure that connects hardware manufacturers, operating system developing companies, app developers, and end users. [13]

For example, the smartphone manufacturers will produce devices that support hardware interfaces of the operating system, and operating system companies create a generic library to operate devices from various manufacturers. Therefore, software developers can use numerous devices without understanding hardware to develop applications. The ecosystem includes the distribution of application to end users.

Despite the name, ROS is not, in fact, an operating system. Rather, it's an SDK (software development kit) that provides the building blocks you need to build your robot applications. Whether your application is a class project, a scientific experiment, a research prototype, or a final product, ROS will help you to achieve your goal faster. And it's all open source.

Figure 14. The ROS Ecosystem [23]

2.6 History of ROS

In May 2007, ROS was started by borrowing the early open- source robotic software frameworks including switchyard, which is developed by Dr. Morgan Quigley by the Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot STAIR (STanford AI Robot) project. Dr. Morgan Quigley is one of the founders and software development manager of Open Robotics (formerly the Open-Source Robotics Foundation, OSRF), which is responsible for the development and management of ROS. Switchyard is a program created for the development of artificial intelligence robots used in the AI lab's projects at the time and is the predecessor of ROS.

In addition, Dr. Brian Gerkey (http://brian.gerkey.org/), the developer of the Player/Stage Project (Player network server and 2D Stage simulator, later affects the development of 3D simulator Gazebo), which was developed since 2000 and has had a major impact on ROS's networking program, is the CEO and co-founder of Open Robotics. Thus, ROS was influenced by Player/Stage from 2000 and Switchyard from 2007 before Willow Garage changed the name to ROS in 2007.

Figure 15. Open Robotics and OSRF Logo [24]

In November 2007, U.S. robot company Willow Garage succeeded the development of ROS. Willow Garage is a well-known company in the field for personal robots and service robots. It is also famous for developing and supporting the Point Cloud Library (PCL), which is widely used for 3D devices such as Kinect and the image processing open-source library OpenCV.

Willow Garage started to develop ROS in November of 2007, and on January 22, 2010, ROS 1.0 came out into the world. The official version known to us was released on March 2, 2010 as ROS Box Turtle. Afterwards, C Turtle, Diamondback and many versions were released in alphabetical order like Ubuntu and Android.

ROS is based on the BSD 3-Clause License and Apache License 2.0, which allows anyone to modify, reuse, and redistribute. ROS also provided a large number of the latest software and participated actively in education and academics, becoming known first through the robotics academic society. There is now ROSDay and ROSCon conferences for developers and users, and also various community gatherings under the name of ROS Meetup. In addition, the development of robotic platforms that can apply ROS are also accelerating. Some examples are the PR214 which stands for Personal Robot and the TurtleBot15, and many applications have been introduced through these platforms, making ROS as the dominating robot software platform. [14]

2.7 The ROS Versions

ROS distribution is a versioned set of ROS packages. These are akin to Linux distributions (e.g. Ubuntu). The purpose of the ROS distributions is to let developers work against a relatively stable codebase until they are ready to roll everything forward. Therefore, once a distribution is released, we try to limit changes to bug fixes and non-breaking improvements for the core packages (everything under ros-desktop-full). And generally, that applies to the whole community, but for "higher" level packages, the rules are less strict, and so it falls to the maintainers of a given package to avoid breaking changes.

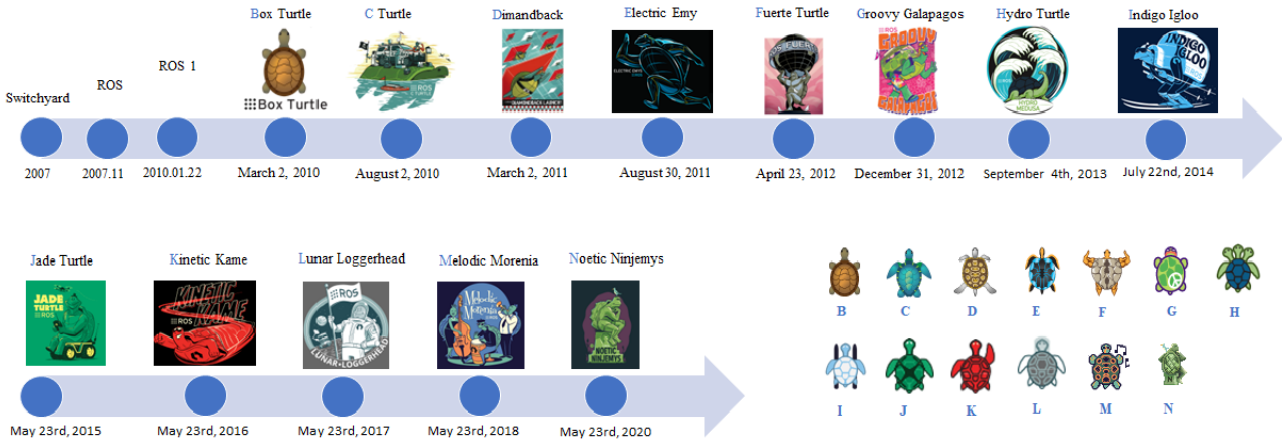


Figure 16. ROS versions timeline [9]

End-of-life ROS 1 distributions:

- C Turtle
- Diamondback
- Electric Emys
- Fuerte Turtle
- Groovy Galapagos
- Hydro Medusa
- Indigo Igloo
- Jade Turtle
- Kinetic Kame
- Lunar Loggerhead

End-of-life ROS 2 distributions:

- Ardent Apalone
- Bouncy Bolson
- Crystal Clemmys
- Dashing Diademata
- Eloquent Elusor
- Galactic Geochelone

Table 2. ROS2 distributions delivered until August 2021. [9]

Distro name	Release Data	EOL date	Ubuntu version
Galactic Geochelone	May 23rd, 2021	November 2022	Ubuntu 20.04
Foxy Fitzroy	June 5th, 2020	May 2023 (LTS)	
Eloquent Elusor	November 22nd, 2019	November 2020	Ubuntu 18.04
Dashing Diademata	May 31st, 2019	May 2021 (LTS)	
Crystal Clemmys	December 14th, 2018	December 2019	
Bouncy Bolson	July 2nd, 2018	July 2019	Ubuntu 16.04
Ardent Apalone	December 8th, 2017	December 2018	

Figure 17. Active ROS 1 distributions. [9]

Figure 18. Active ROS 2 distributions. [9]

ROS 2 is a software platform for developing robotics applications, also known as a robotics software development kit (SDK). Importantly, ROS 2 is open source. ROS 2 is distributed under the Apache 2.0 License, which grants users broad rights to modify, apply, and redistribute the software, with no obligation to contribute back. ROS 2 relies on a federated ecosystem, in which contributors are encouraged to create and release their own software. Most additional packages also use the Apache 2.0 License or similar. Making code free is fundamental to driving mass adoption - it allows users to leverage ROS 2 without constraining how they use or distribute their applications. [15]

Table 3. Summary of ROS 2 features compared to ROS 1. [15]

Category	ROS 1	ROS 2
Network Transport	Bespoke protocol built on TCP/UDP	Existing standard (DDS), with abstraction supporting addition of others
Network Architecture	Central name server(`roscore`)	Peer-to-peer discovery
Platform Support	Linux	Linux, Windows, macOS
Client Libraries	Written independently in each language	Sharing a common underlying C library (`rcl`)
Node vs. Process	Single node per process	Multiple nodes per process
Threading Model	Callback queues and handlers	Swappable executor
Node State Management	None	Lifecycle nodes
Embedded Systems	Minimal experimental support (`rosserial`)	Commercially supported implementation (micro-ROS)
Parameter Access	Auxiliary protocol built on XMLRPC	Implemented using service calls
Parameter Types	Type inferred when assigned	Type declared and enforced

3 CONFIGURING THE ROS 2 DEVELOPMENT ENVIRONMENT

Before starting to develop with ROS 2, there are a few preliminary preparations that you should make:

1. Operating System
2. Programming language
3. Installing ROS 2 and Packages
4. Installing visualization and robotic tools

Although ROS has been made to work on a wide variety of systems, in this article I will explain the details about Ubuntu Linux, a popular and relatively user-friendly Linux distribution. Ubuntu provides an easy-to-use installer that allows computers to dual-boot between the operating system they were shipped with (typically Windows or Mac OS X) and Ubuntu itself. It is important to back up your computer before installing Ubuntu, in case something unexpected happens and the drive is completely erased in the process.

Even though there are virtualization environments such as VirtualBox and VMware that allow Linux to run concurrently with a host operating system such as Windows or Mac OS X. The simulator can be rather compute- and graphics-intensive and might be overly sluggish in a virtualized environment. I recommend running Ubuntu Linux natively by following the instructions on the Ubuntu website. Ubuntu Linux can be downloaded freely from “https://ubuntu.com”.

ROS2 works on Linux Ubuntu, Linux Debian or Linux Gentoo as well as MacOS and Windows. But Linux Ubuntu installation is the simplest and the one more mature, so it is a more preferred option.

The ROS 2 application development environment can be used is as follows.

- Hardware: Desktop or laptop using Intel or AMD processor
- Operating System: Linux Ubuntu, Linux Debian, Linux Gentoo, MacOS and Windows

- ROS 2: Foxy Fitzroy, Humble Hawksbill
- Tools and packages: Rviz, Gazebo, MoveIt, Webots

It would be better to check their official websites of Operating Systems and ROS 2 for the future updates. They continuously updated and some of version might out of date.

3.1 Installing Operating Systems: Linux

You may be wondering if you could create ROS 2 programs using Windows or even Mac, instead of Linux. The Linux version of ROS 2 is the most mature of all of them, which means that you will have less trouble trying to make something to work in ROS 2. The problem here is that learning ROS 2 has a very stepped curve, so you don't need to add more confusion to your learning than the one of ROS 2 itself. You check a lot of recourse about Linux. To make this work easier, I used ROS 2 Development Studio, which is already a Linux system.

The Windows version of ROS 2 is available as well. You may find confusing to try to make basic ROS 2 things work properly. There is no detailed information about the use of ROS 2 in Windows. Under the next topic, I will explain detailed installation explanations and give more detailed information.

3.2 Programming ROS 2 in Python or C++

As you may know, you can create ROS 2 programs mainly in two programming languages: Python and C++. Mostly, whatever you can do in ROS 2 with C++, you can do it also with Python. Furthermore, you can have C++ ROS 2 programs talking to other Python ROS 2 programs in the same robot. Since the nodes communicate using standard ROS 2 messages, the actual language implementation of the nodes is not affecting their communication. That is the beauty of ROS 2.

Pros of programming ROS 2 in Python:

- Is faster to build a prototype. You can create a working demo of your node very fast if you use Python, because the language takes care of a lot of things by itself, so the programmer doesn't have to bother.
- You don't need to compile and spend endless hours trying to catch a hidden bug. Even if you can have bugs in Python, the nature of them is a lot easier and faster to catch.
- You can learn Python very fast. You can make short programs for complex things.
- The final code of your node is quite easy to read and understand what it does.
- It is easier to integrate it with web services based on Django. Since Django is based on Python, you can integrate ROS 2 functions easily in the server calls.
- It is easier to understand some ROS 2 concepts if you use the Python API, because some complex concepts are hidden for the developer in the ROS 2 Python API. For example, things like the *Callback Queue* are handled directly by the ROS 2 Python API.

Cons of programming ROS in Python:

- It runs slower. Python is an interpreted language, which means that it is compiled in run time, while the program is being executed. That makes the code slower.
- Higher chances of crashing in run time, that is, while the program is running on the robot. You can have a very silly mistake in the code that won't be cached until the program runs that part of the code (in run time).
- Unless you define very clear procedures, you can end with a messy code in a short time if the project grows. I mean, you need to impose some rules for developing, like indicating every class you import from which library is, force strong types on any definition of a variable, etc.
- Doesn't allow real time applications.

Pros of programming ROS in C++:

- The code runs fast. Maybe in your project, you need some fast code.

- By having to compile, you can catch a lot of errors during compilation time, instead of having them in run time.
- C++ has an infinite number of libraries that allow you to do whatever you want with C++.
- It is the language used in the robotics industry, so you need to master it if you want to work there.
- It is necessary for real time code.

Cons of programming ROS in C++:

- C++ is a lot more complex to learn and master.
- Just creating a small demo of something requires creating a lot of code.
- To understand what a C++ program does can take you a long time.
- Debugging errors in C++ is complex and takes time.

As can be understood from the details above C++ is a good choice for developing robotic applications in ROS 2 because of its performance, memory management, library support, and compatibility. Therefore, I used C++ language in my study.

3.3 Installing ROS 2

Robots must be programmed to be useful. It is useless for a robot to be a mechanical prodigy without providing it with software that processes the information from the sensors to send the correct commands to the actuators to fulfil the mission for which it was created. This chapter introduces the middleware for programming robots with ROS 2.

Pre-requisites that you must meet before trying to get into ROS. If you want to develop for ROS based robots, you need to know in advance how to program either in C++ or Python. Also, you must be comfortable using the Operating System Linux or other system.

Installing from binary packages or from source will both result in a fully functional and usable ROS 2 install. Differences between the options depend on what you plan to do with ROS 2.

Binary packages are for general use and provide an already-built install of ROS 2. This is great for people who want to dive in and start using ROS 2 as-is, right away.

Linux users have two options for installing binary packages:

- Debian packages
- “fat” archive

Installing from Debian packages is the recommended method. It’s more convenient because it installs its necessary dependencies automatically. It also updates alongside regular system updates. However, you need root access in order to install Debian packages. If you don’t have root access, the “fat” archive is the next best choice.

MacOS and Windows users who choose to install from binary packages only have the “fat” archive option (Debian packages are exclusive to Ubuntu/Debian).

Building from source is meant for developers looking to alter or explicitly omit parts of ROS 2’s base. It is also recommended for platforms that don’t support binaries. Building from source also gives you the option to install the absolute latest version of ROS 2.

Next thing I need to do is to set my system up so I can get ROS2 installed in a machine which to develop my programs and practice. Basically, two options to set up a machine for programming for robots with ROS2.

1. Install ROS2 in your local computer. As I mention before several options are available for different operating system like Linux Ubuntu, Linux Debian, Linux Gentoo, MacOS and Windows.

2. Use online ROS Development Studio (ROSDS) which already provides everything installed and setup for any type of computer and requires only a web browser.

By completing these preliminary preparations, you will have a solid foundation for developing with ROS 2. You will be ready to start building and testing your own robotic applications.

3.3.1 Installing ROS 2 Foxy Fitzroy to Linux Ubuntu

The section explains how to install ROS 2 Foxy Fitzroy in Linux. ROS 2 works on Linux Ubuntu, Linux Debian or Linux Gentoo as well as MacOS and Windows. As up today, April 2023 on Ubuntu website the latest Ubuntu 22.04.2 LTS version is available for desktop PCs and laptops.

For a guide on how to install Ubuntu on a windows computer. It covers two options: removing completely the Windows partition substituting it by Ubuntu or making a dual boot where you can have both systems working on the same machine (you decide which operating system to use on boot time).

Once you have an Ubuntu system working in your computer, you need to install ROS2 on it. I am going to cover how to install the latest release, ROS2 Foxy Fitzroy. Those instructions are a simplification from the original instructions published by Open Robotics.

Step 1. Set Locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like POSIX. We test with the following settings. However, it should be fine if you're using a different UTF-8 supported locale.

```
locale # check for UTF-8

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
```

Step 2. Configuration

The first step is setup your package environment. You will need to add the ROS 2 apt repository to your system. First ensure that the Ubuntu Universe repository is enabled. Type the following commands:

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

Now add the ROS 2 GPG key with apt.

```
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
```

Then add the repository to your sources list.

```
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
```

Step 3. Install Development and ROS Tools

Update your apt repository caches after setting up the repositories.

```
sudo apt update && sudo apt install -y \
  libbullet-dev \
  python3-pip \
  python3-pytest-cov \
  ros-dev-tools

# install some pip packages needed for testing
python3 -m pip install -U \
  argcomplete \
  flake8-blind-except \
  flake8-builtins \
  flake8-class-newline \
  flake8-comprehensions \
  flake8-deprecated \
  flake8-docstrings \
  flake8-import-order \
  flake8-quotes \
  pytest-repeat \
```

```
  pytest-rerunfailures \
  pytest
# install Fast-RTPS dependencies
sudo apt install --no-install-recommends -y \
  libasio-dev \
  libtinyxml2-dev
# install Cyclone DDS dependencies
sudo apt install --no-install-recommends -y \
  libcunit1-dev
```

Step 4. Get ROS 2 Code

Create a workspace and clone all repos:

```
mkdir -p ~/ros2_foxy/src
cd ~/ros2_foxy
vcs import --input https://raw.githubusercontent.com/ros2/ros2/foxy/ros2.repos src
```

Install dependencies using rosdep

ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt upgrade
```

```
sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src -y --skip-keys "fastcdr rti-connext-dds-5.3.1
urdfdom_headers"
```

If you're using a distribution that is based on Ubuntu (like Linux Mint) but does not identify itself as such, you'll get an error message like `Unsupported OS [mint]`. In this case append `--os=ubuntu:focal` to the above command.

Step 5. Environment Setup

Sourcing the setup script and set up your environment by sourcing the following file.

```
# Replace ".bash" with your shell if you're not using bash
```

```
# Possible values are: setup.bash, setup.sh, setup.zsh
. ~/ros2_foxy/install/local_setup.bash
```

Step 6. Try Some Examples

If you installed `ros-foxy-desktop` above you can try Talker-listener examples. In one terminal, source the setup file and then run a C++ `talker`:

```
. ~/ros2_foxy/install/local_setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python `listener`:

```
. ~/ros2_foxy/install/local_setup.bash
ros2 run demo_nodes_py listener
```

You should see the `talker` saying that it's `Publishing` messages and the `listener` saying `I heard` those messages. This verifies both the C++ and Python APIs are working properly.

3.3.2 Using the ROS 2 Development Studio (ROSDS)

This is by far the easiest way to start with ROS 2. The ROS Development Studio (ROSDS) is an online development platform provided by The Construct to allow people develop for ROS 2 fast and without caring about the details under the hood.

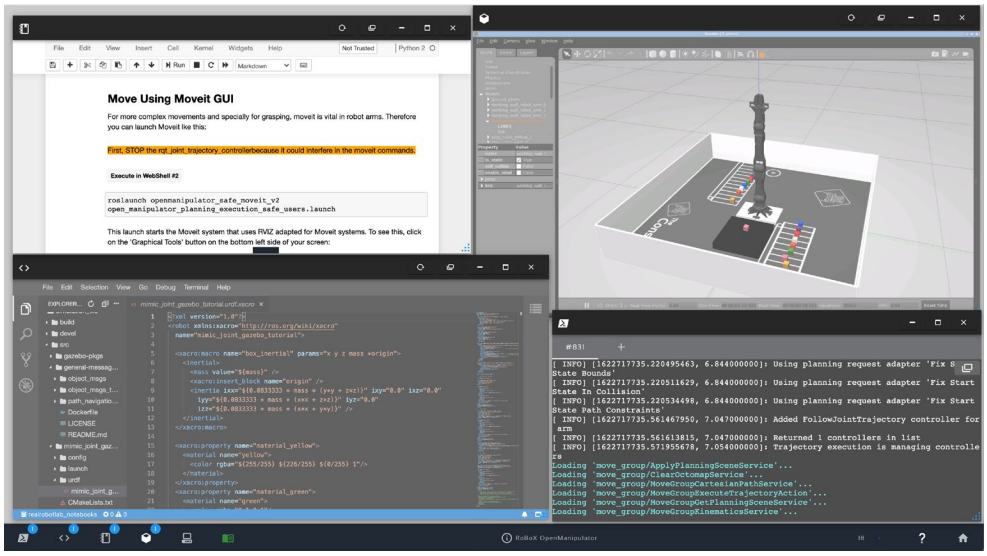


Figure 19. The ROS Development Studio (ROSDS) by The Construct. [16]

The Construct's ROSDS is widely used by universities and research centres worldwide because they don't require the students to install anything in their computers, especially when that requires to partition the hard disk with different operating systems. [16]

The ROSDS provides a complete development environment as if you had everything properly configured in your computer. The advantages are:

- You don't need to install anything in your computer.
- You can use Windows, Mac, or Linux based computers. So don't worry about the system of your computer.
- If something you write breaks the ROS system, you don't care because just by re-starting the ROSject, the ROS system gets re-started to its initial state.
- You can create projects for many different distributions: Foxy, Humble, etc…
- You can share your ROS code with a single link and the code will work the same way to anybody who receives the link (including simulations, datasets, and documentation).

3.4 Installing MoveIt 2 Packages

MoveIt can be defined as Motion Planning Framework. Easy-to-use open-source robotics manipulation platform for developing commercial applications, prototyping designs, and benchmarking algorithms.

MoveIt 2 is the robotic manipulation platform for ROS 2, and incorporates the latest advances in motion planning, manipulation, 3D perception, kinematics, control, and navigation. Alternatively, you can easily use any robot that has already been configured to work with MoveIt. There is list of robots running MoveIt on ROS on website to see whether MoveIt is already available for your robot. Otherwise, you can setup MoveIt to work with your custom robot.

- Install MoveIt 2 packages: You will need to install the MoveIt 2 packages. You can do this using the following command in your terminal:

```
sudo apt-get install ros-<ros2-distro>-moveit
```

Replace <ros2-distro> with the name of your ROS 2 distribution (e.g. foxy or humble).

- Build MoveIt 2 from source (optional): If you want to build MoveIt 2 from source, you can clone the MoveIt 2 repository from GitHub and build it using colcon. Here are the commands you can use:

```
mkdir -p ~/moveit2_ws/src
cd ~/moveit2_ws/src
git clone https://github.com/ros-planning/moveit2.git
cd ..
rosdep install --from-paths src --ignore-src -r -y
colcon build
```

- Verify the installation: To verify that MoveIt 2 is installed correctly, you can launch the MoveIt 2 demo. Use the following command in your terminal:

```
ros2 launch moveit2_tutorials move_group_interface_tutorial.launch.py
```

This will launch the MoveIt 2 demo, which will allow you to control a simulated robot arm using MoveIt 2. By following these steps, you can install MoveIt 2 and start using it to plan and control robot manipulation in ROS 2.

3.5 Installing Gazebo

Gazebo is an open-source 3D robotics simulator. It integrated the ODE physics engine, OpenGL rendering, and support code for sensor simulation and actuator control.

Gazebo brings a fresh approach to simulation with a complete toolbox of development libraries and cloud services to make simulation easy. Iterate fast on your new physical designs in realistic environments with high fidelity sensors streams. Test control strategies in safety and take advantage of simulation in continuous integration tests. [17]

Gazebo can use multiple high-performance physics engines, such as ODE, Bullet, etc. (the default is ODE). It provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle), Kinect style sensors, etc. [18]

Gazebo is a popular open-source simulation environment that is commonly used in robotics research and development. Here are the steps you can follow to use Gazebo with ROS 2:

- Install Gazebo: The first step is to install Gazebo on your system. You can download the latest version of Gazebo from the official Gazebo website. Make sure to choose the version that is compatible with your operating system.
- Install ROS 2: If you haven't already, you will need to install ROS 2 on your system. You can download the latest version of ROS 2 from the official ROS 2 website.
- Install Gazebo-ROS 2 packages: Next, you will need to install the Gazebo-ROS 2 packages that allow ROS 2 to communicate with Gazebo. You can install these packages using the following command in your terminal:

```
sudo apt-get install ros-<ros2-distro>-gazebo-ros-pkgs
```

You can Replace <ros2-distro> with the name of your ROS 2 distribution (e.g. foxy or humble).

- Launch Gazebo and ROS 2: To launch Gazebo and ROS 2 together, you can use the following command in your terminal:

```
gazebo --verbose worlds/empty.world
```

This will launch Gazebo with an empty world. You can then launch your ROS 2 nodes and start sending commands to the simulated robot.

3.6 Installing RViz

Rviz, abbreviation for ROS visualization, is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. RVIZ is a ROS graphical interface that allows you to visualize a lot of information, using plugins for many kinds of available topics.

Differences between RViz and Gazebo is basically RViz shows you what the robot thinks is happening, while Gazebo shows you what is really happening.

Install RViz 2 packages: You will need to install the RViz 2 packages. You can do this using the following command in your terminal:

```
sudo apt-get install ros-<ros2-distro>-rviz2
```

You can Replace <ros2-distro> with the name of your ROS 2 distribution (e.g. foxy or humble). By following this step, you can install RViz 2 and start using it for 3D visualization in ROS 2.

4 CONTROL A COLLABORATIVE ROBOT ARM IN ROS 2 PLATFORM

The UR3e collaborative robot is a small, lightweight, and flexible robotic arm designed for safe and efficient interaction with humans in various industrial and service applications. It is one of the products in the Universal Robots line of collaborative robots, which are known for their ease of use, versatility, and safety features.

The UR3e robot arm has six degrees of freedom, meaning it can move in six different directions. Its compact size and lightweight design make it easy to integrate into existing production lines or workspaces, and it can be easily moved between different tasks or locations as needed.

One of the key features of the UR3e robot arm is its collaborative capability. It is designed to work safely alongside humans, without the need for safety barriers or cages. This is made possible through a range of safety features, including built-in force sensing, speed and proximity monitoring, and the ability to stop or slow down if it encounters a person or object.

The UR3e robot arm is also highly adaptable and can be programmed to perform a wide range of tasks, from simple pick-and-place operations to more complex assembly and manufacturing processes. It can be easily programmed using the Universal Robots graphical user interface, which requires no prior programming experience.

Overall, the UR3e collaborative robot is a powerful and versatile tool that can help improve efficiency, productivity, and safety in a wide range of industrial and service applications.

Table 4. Specifications for the Universal Robots e-Series robots.

	UR3e	UR5e	UR10e	UR16e
Reach	500 mm	850 mm	1300 mm	900 mm
Payload	3 kg	5 kg	10 kg	16 kg
Footprint	Ø128 mm	Ø149 mm	Ø190 mm	Ø190 mm
Weight	11.2 kg	20.6 kg	33.5 kg	33.1 kg

4.1 Dynamic Model

The dynamic model of the robot manipulator relates the actuator torques to the motion of the robot manipulator. It contains a set of kinematic parameters such as the position and the orientation of the joints and a set of dynamic parameters such as masses, centre-of-mass positions, and inertia components. The robot arm is a 6 degree-of-freedom (DOF) serial-link articulated robot manipulator which means that the robot links are connected in series and the joints admit rotary motion of the links.

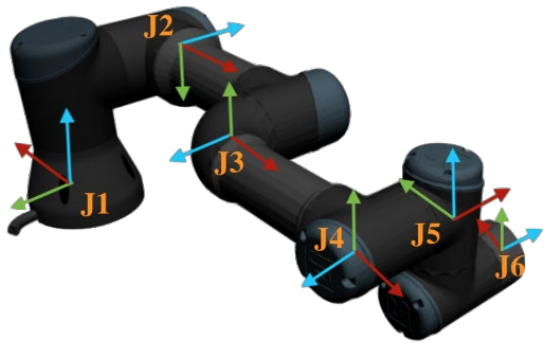


Figure 20. UR3e robot arm joints (6 DOF)

The dynamic model of the UR3e robot arm can be used to analyse the robot's performance, optimize its motion, and develop control algorithms for different tasks. It can also be used to design and test new components and systems for the robot arm, and to evaluate the impact of changes in the robot's configuration on its performance.

4.1.1 Robot Kinematics

The kinematic model describes the relation between joint angles **q** and the position and orientation **x** of the end-effector. The forward kinematics describes the transformation **Γ(·)** from joint angles to the end-effector position and orientation, and the inverse kinematics describes the transformation from the end-effector position and orientation to the joint angles. The forward kinematics can be represented as joint angles.

Table 5. The Denavit–Hartenberg parameters of UR3e robots are shown as below. [19]

UR3e							
Kinematics	Theta [rad]	a [m]	D [m]	Alpha [rad]	**Dynamics**	Mass [kg]	Center of Mass[m]
Joint 1	0	0	0.15185	π/2	Link 1	1.98	[0, -0.02, 0]
Joint 2	0	-0.24355	0	0	Link 2	3.4445	[0.13, 0, 0.1157]
Joint 3	0	-0.2132	0	0	Link 3	1.437	[0.05, 0, 0.0238]
Joint 4	0	0	0.13105	π/2	Link 4	0.871	[0, 0, 0.01]
Joint 5	0	0	0.08535	-π/2	Link 5	0.805	[0, 0, 0.01]
Joint 6	0	0	0.0921	0	Link 6	0.261	[0, 0, -0.02]

The forward kinematics can be represented as

$$\mathbf{x} = \boldsymbol{\Gamma}(\mathbf{q}) \tag{4.1}$$

and the inverse kinematic is given:

$$\mathbf{q} = \boldsymbol{\Gamma}^{-1}(\mathbf{x}) \tag{4.2}$$

The kinematic model is derived using Denavit–Hartenberg (DH) convention or its modified variant. The kinematic parameters of the UR3e robots are listed in Table 4 and illustrated on a UR3e robot in figure 21.

The system dynamics are represented by the equations of motion which can be expressed according to:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{u}_f = \mathbf{u}, \tag{4.3}$$

where **q** is the vector of joint coordinates, **M(q)** describes the inertia matrix, **C(q, q̇)q̇** considers coriolis and centripetal torques, while **G(q)** includes gravitational loads. The robot is driven by the torque vector **u**, while additional external forces or torques are not considered in this paper. Mechanical losses in the joints are considered by the friction torque $\mathbf{u}_f$. The equations of motion can be evaluated by the recursive Newton–Euler algorithm based on mechanical parameters.

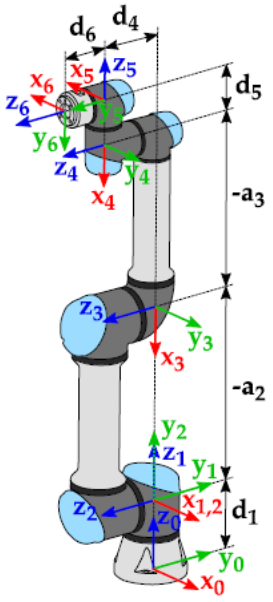


Figure 21. UR3e robot arm in its home position with the kinematic parameters

Corresponding kinematic and mass properties are provided by the universal robots and are summarized in Table 4. Different payloads are additionally implemented as a point mass ml at the position r1 with respect to the flange of the robot. Thus, the adapted mass $\hat{\mathbf{m}}_6$ and centre of gravity $\hat{\mathbf{r}}_6$ of link 6 are determined according to:

$$\hat{\mathbf{m}}_6 = \mathbf{m}_6 + \mathbf{m}_l \; ; \; \hat{\mathbf{r}}_6 = \frac{\mathbf{m}_6\mathbf{r}_6 + \mathbf{m}_l\mathbf{r}_l}{\mathbf{m}_6\mathbf{m}_l} \tag{4.4}$$

One relevant mechanical characteristic of the actuators related to the system dynamics is the reduced inertia, which is included in **M(q)**. It is assumed that the weight of the actuators is included

in the weights of the links provided by the manufacturer, thus, the reduced rotational inertia must be considered additionally. Furthermore, mechanical losses are considered in the drive train. A simple model describing the corresponding joint friction considers Coulomb and viscous friction with the coefficients $\mathbf{B_c}$ and $\mathbf{B_v}$, respectively, and is given by [20]

$$\mathbf{u_f = B_c\ sgn(\dot{q}) + B_v\dot{q}} \tag{4.4}$$

If an external wrench $f_{ext} \in R^N$ is applied at the end–effector, the resulting torques at the joints are

$$\mathbf{u = J^T(q).\ f_{ext}} \tag{4.5}$$

with **J(q)** the kinematic Jacobian of the manipulator. For free motion $\mathbf{u = f_{ext} = 0.}$

4.1.2 Kinematic Model

The robot to a kinematic skeleton of its axes assuming, without any loss of generality, that the base point is situated at the world origin: (0, 0, 0) with standard coordinate directions x-axis: (1, 0, 0), y: (0, 1, 0) and z: (0, 0, 1). The robot has no tool end effector attached; therefore, the target is assumed at the front face centre point of the flange.

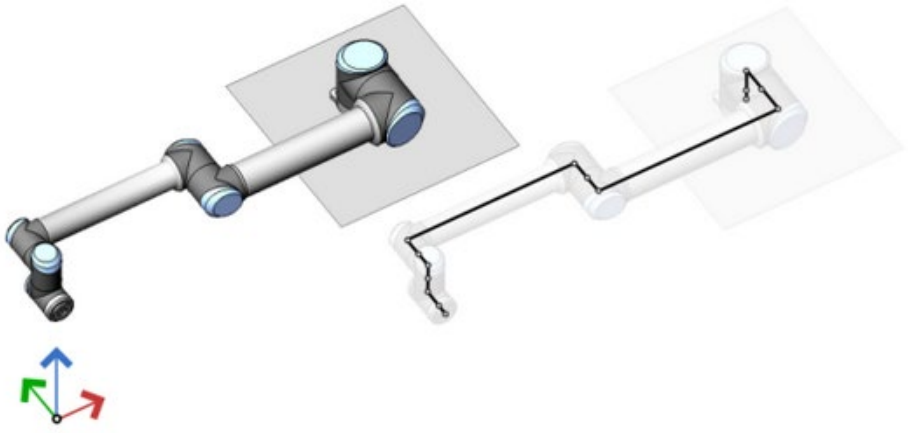


Figure 22. UR3e robot arm standard coordinate directions

The zero joint angle position $J_{1..6}$ = (0, 0, 0, 0, 0, 0) → x-axis: (1, 0, 0), y-axis: (0, 0, 1), z-axis: (0, -1, 0) for UR seen below as the horizontal pose is different than its home position which is $J_{1..6}$ = (0, -90, 0, -90, 0, 0) → x-axis: (-1, 0, 0), y-axis: (0, -1, 0), z-axis: (0, -1, 0)) seen below as the upright pose.

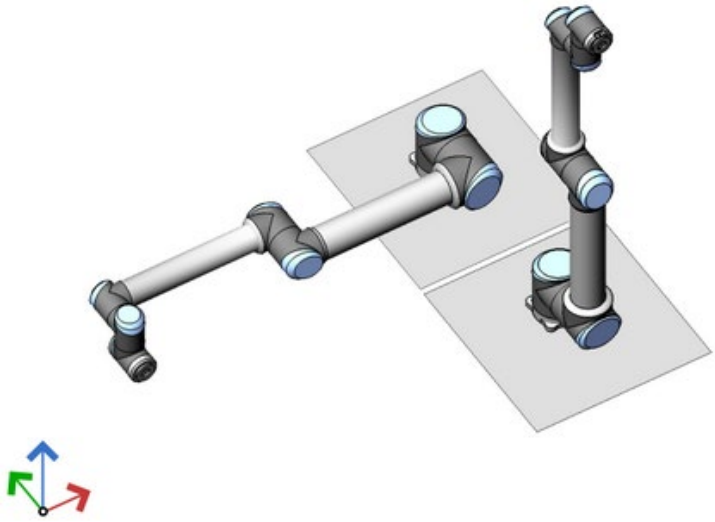


Figure 23. UR3e robot arm zero and home positions

4.2 Gazebo Model

I mentioned before to make process easy, I used online ROS Development Studio (ROSDS) which already provides everything installed and setup for any type of computer and requires.

To start the simulation, need to source workspace. Bellow codes will start to gazebo simulation:

```
source ~/simulation_ws/devel/setup.bash
roslaunch ur_e_gazebo ur3e.launch
```

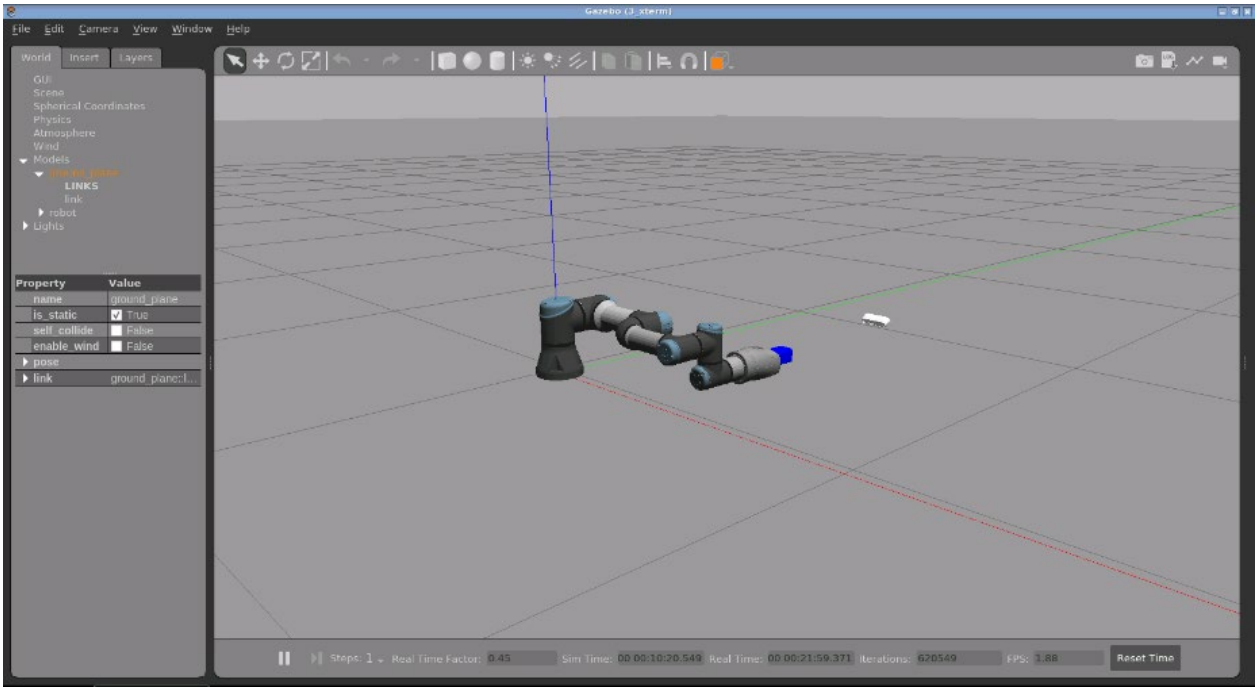


Figure 24. Gazebo UR3e collaborative Robot simulation interface.

It should be able to see the simulation and control everything as if it was the real robot.

4.3 Test Moveit2

Before starting with the robot movement, let's check that Moveit2 runs correctly. For this action, you will need 3 steps:

Step 1. Launch the Parameter Bridge

Setup launch the parameter bridge between ROS 1 and ROS 2, run below codes to build the bridge. The topics that bridged:

```
source ~/catkin_ws/devel/setup.bash
roslaunch load_params load_params.launch
source /home/simulations/ros2_sims_ws/install/setup.bash
ros2 run ros1_bridge parameter_bridge __name:=parameter_bridge
```

Also, we can check ros2 topic list to control is bridged or not.

```
ros2 topic list
```

```
/clock
/joint_states
/parameter_events
/rosout
/tf
/tf_static
```

Step 2. Launch the Action Bridge

I need to bridge between ROS 1 and ROS 2. Action servers that I need from ROS 1 in order to with the arm and gripper controller. Bellow commands for the action bridge.

```
source /opt/ros/noetic/setup.bash
source /home/simulations/ros2_sims_ws/install/setup.bash
ros2 launch start_bridge start_bridge.launch.py
```

To check for action list:

```
ros2 action list
```

/arm_controller/follow_joint_trajectory

/gripper_controller/gripper_cmd

We can check arm controller and gripper connected. In this case I want control and communicate with arm controller only. Plan and execute trajectories for your robot using the MoveIt2 RVIZ environment and for the script in C++ can be found in Appendix A.

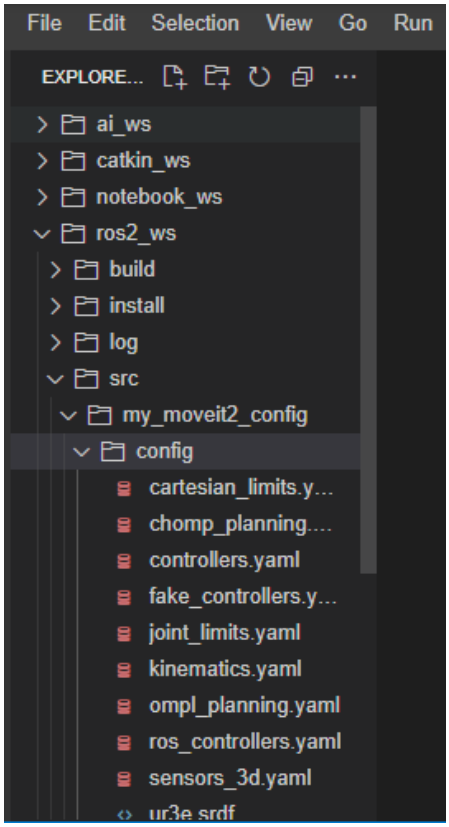


Figure 25. ROS 2 package at the code editor.

Step 3. Launch Moveit 2

To start Moveit 2 package and Rviz so that I have Moveit2 graphical interface.

```
source ~/ros2_ws/install/setup.bash
ros2 launch my_moveit2_config my_planning_execution.launch.py launch_rviz:=true
```

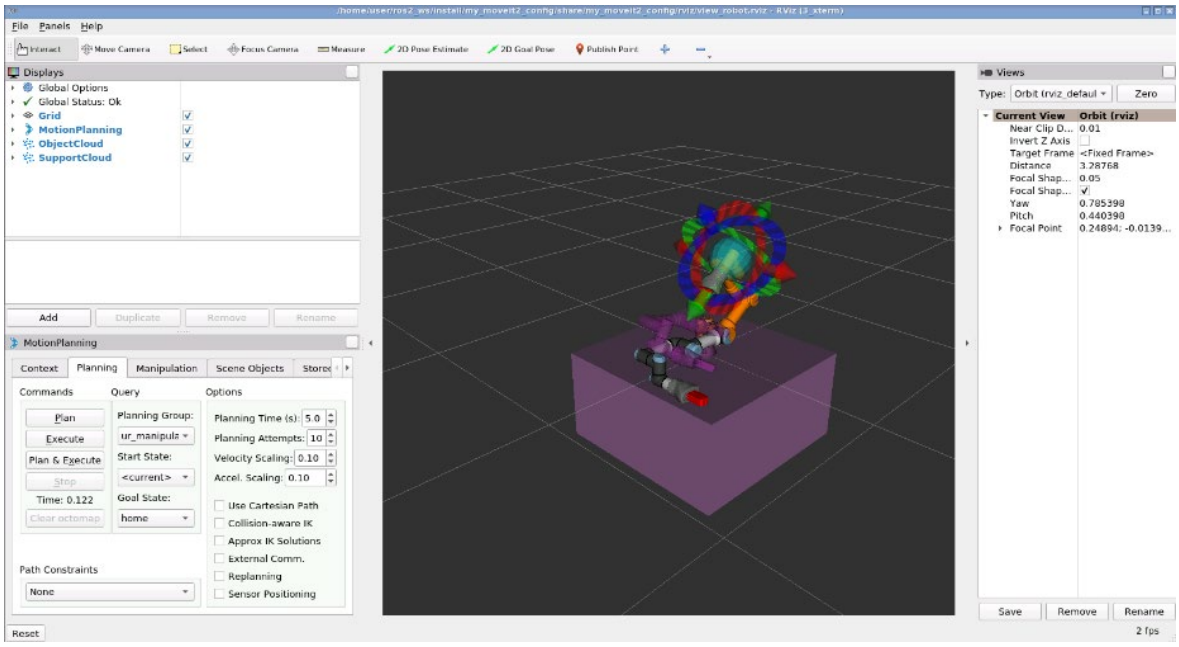


Figure 27. MoveIt 2 graphical interface package first positioning of the collaborative robot arm

Here, on the planning section I changed to Goal State to “Home position”. It is final position for robot arm to reach. Final position show as orange colour and moving direction show as dark purple colour image of arm. I planned the trajectory as show Figure 22. position after that when click execute button it will give command to RViz.

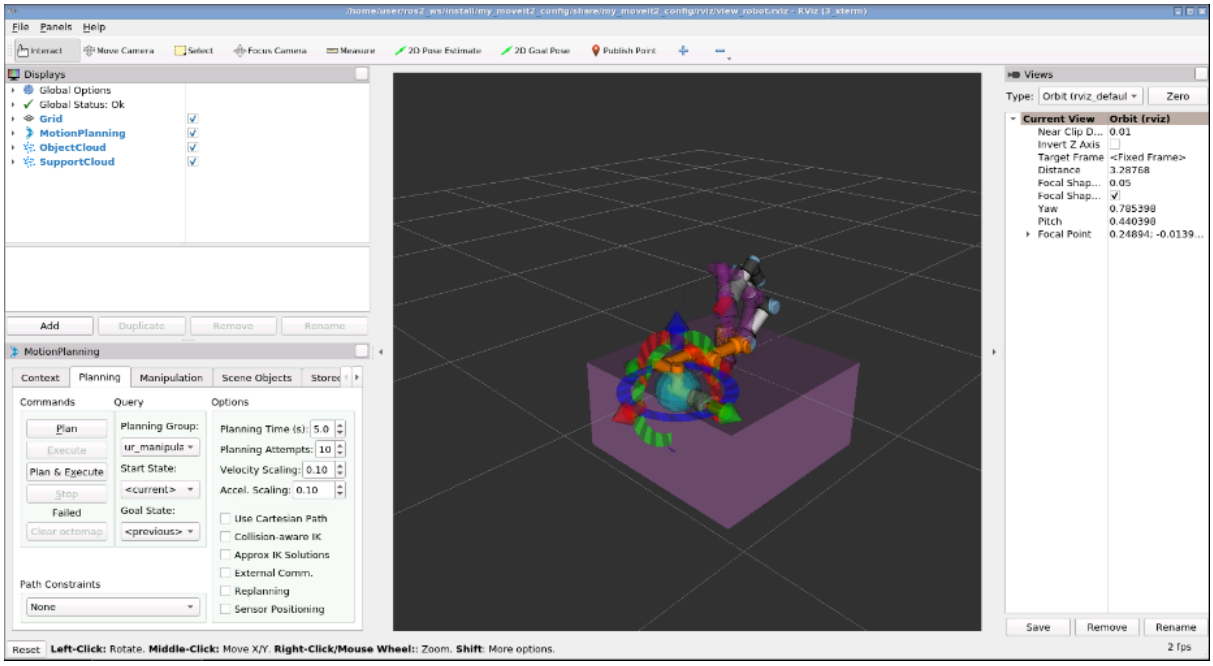


Figure 26. Returning the arm to the previous position in MoveIt2.

For now, robotic arm has not move at all on RViz. In order to move the arm, I need to execute the trajectory and it will be executed in the simulation as well.

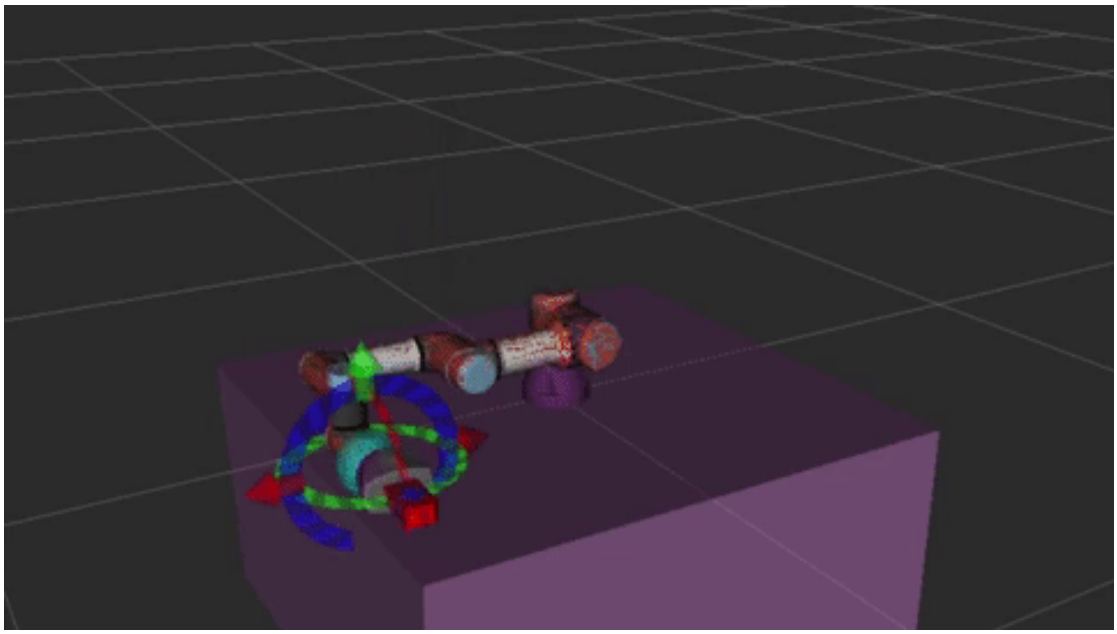


Figure 28. Visualization of the collaborative robot arm using with ROS MoveIt2 visualisation tool RViz

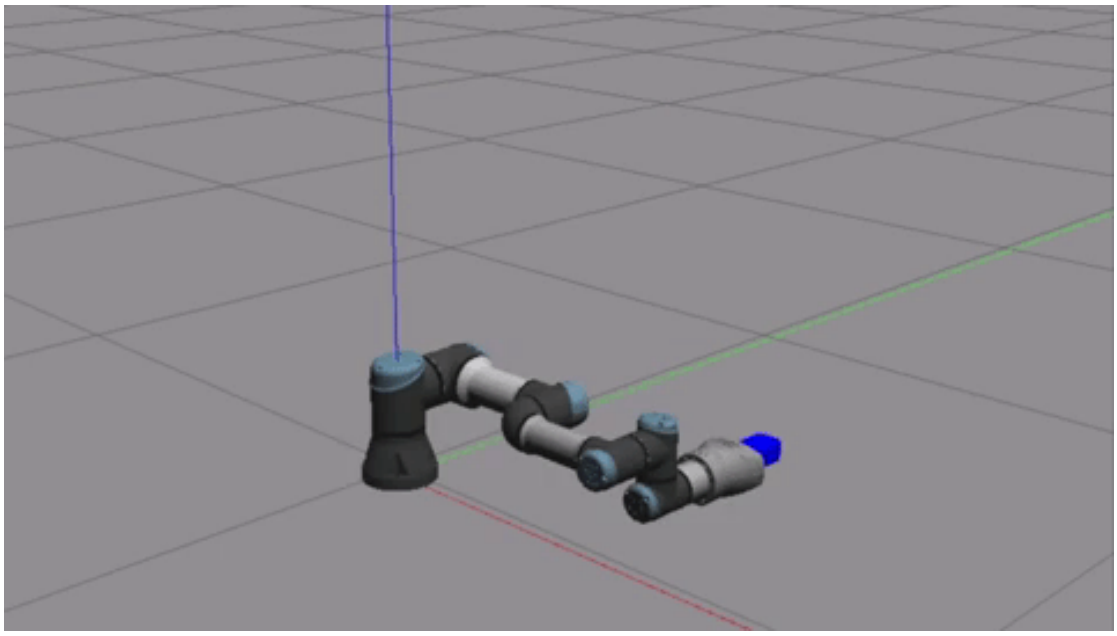


Figure 29. Visualization of the collaborative robot arm on Gazebo

5 CONCLUSION AND FUTURE WORK

In conclusion, the collaborative robot arm was successfully capable to move in the ROS 2 Foxy Fitzroy environment, simulated in the Gazebo software. First trajected at the ROS MoveIt2 visualisation tool RViz then the necessary adjustments are made for the collaborative robot arm to take its initial position at the virtual environment. When final planning is complete, it was executed on MoveIt2 graphical interface package and simulated in the Gazebo software. This study was programmed with C++ and the help of repository ROS 1, ROS 2 and MoveIt2 open-source documentation. And ROS 2 Development Studio was used to make this study easy and fast.

In this study, it could plan and execute trajectories for your robot using the MoveIt2 RVIZ environment. But this is not the typical case. For the future goal to improve this study, robot arm will control complex movement with the intended scripts.

DECLARATION

on authenticity and public assess of final master's thesis

Student's name : Ibrahim Malli

Student's Neptun ID : BLD9OB

Title of the document : Development of the Control of a Robotic Arm Using ROS2

Year of publication : 2023

Department : MSc Mechanical Engineering

I declare that the submitted final master's thesis is my own, original individual creation. Any parts taken from another author's work are clearly marked and listed in the table of contents.

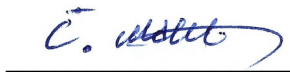
If the statements above are not true, I acknowledge that the Final examination board excludes me from participation in the final exam, and I am only allowed to take final exam if I submit another final essay/thesis/master's thesis/portfolio.

Viewing and printing my submitted work in a PDF format is permitted. However, the modification of my submitted work shall not be permitted.

I acknowledge that the rules on Intellectual Property Management of Hungarian University of Agriculture and Life Sciences shall apply to my work as an intellectual property.

I acknowledge that the electric version of my work is uploaded to the repository system of the Hungarian University of Agriculture and Life Sciences.

Gödöllő, 2023, May 2.



Student's signature

STATEMENT ON CONSULTATION PRACTICES

As a supervisor of **IBRAHIM MALLI, BLD9OB**, I here declare that the final essay/thesis/master's thesis/portfolio[1] has been reviewed by me, the student was informed about the requirements of literary sources management and its legal and ethical rules.

I recommend/don't recommend[2] the final essay/thesis/**master's thesis**/portfolio to be defended in a final exam.

The document contains state secrets or professional secrets: yes **no***[3]

Place and date: Gödöllő, 2023, May, 9.

Internal supervisor

[1] Please select applicable and delete non-applicable.
[2] Please underline applicable.
[3] Please underline applicable.

6 REFERENCES

[1] B. Williams, “An Introduction to Robotics,” Ohio University, Ohio, 2021.

[2] Wikipedia, “Robotic arm,” [Online]. Available: https://en.wikipedia.org/wiki/Robotic_arm. [Accessed April 2023].

[3] Hirata The Global Production Engineering Company, “Industrial Robot,” [Online]. Available: https://www.hirata.co.jp/en/products/items/archives/170. [Accessed April 2023].

[4] ABB, “YuMi® - IRB 14000 | Collaborative Robot,” ABB, [Online]. Available: https://new.abb.com/products/robotics/robots/collaborative-robots/yumi/irb-14000-yumi. [Accessed April 2023].

[5] Association for Advancing Automation (A3), “Cylindrical,” Association for Advancing Automation (A3), [Online]. Available: https://www.automate.org/products/cylindrical. [Accessed April 2023].

[6] www.robots.com, “RobotWorx,” [Online]. Available: https://www.robots.com/articles/starting-it-off-spherical-robots. [Accessed April 2023].

[7] MSI TEC, “What is a Parallel Robot?,” Omron, [Online]. Available: https://msitec.com/robotics/parallel-robots/. [Accessed April 2023].

[8] Y. Pyo, H. Cho, R. Jung and T. Lim, ROS Robot Programming, Seoul,: ROBOTIS Co.,Ltd., 2017.

[9] Robot Operating System, “ROS,” [Online]. Available: https://www.ros.org/. [Accessed April 2023].

[10] D. Serrano, “Introduction to ROS – Robot Operating System,” *NATO Science & Technology Organization,* 2015.

[11] github, “ROS 2 - Version 2 of the Robot Operating System (ROS) software stack,” Repository. [Online]. [Accessed April 2023].

[12] I. Saito, “APIs,” March 2016. [Online]. Available: http://wiki.ros.org/APIs. [Accessed April 2023].

[13] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, “ROS: an open-source Robot Operating System,” Stanford University, California, 2009.

[14] ROS, “ROSCon,” [Online]. Available: https://roscon.ros.org/2023/. [Accessed April 2023].

[15] S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, “Robot Operating System 2: Design, Architecture, and Uses In The Wild,” *Science Robotics,* 2022.

[16] R. Tellez, “ROS 2 Developers Guide,” The Construct Sim, Catalonia, 2021.

[17] Open Robotics, “Gazebo,” Open Robotics, [Online]. Available: https://staging.gazebosim.org/home. [Accessed April 2023].

[18] E. Ackerman, “Latest Version of Gazebo Simulator Makes It Easier Than Ever to Not Build a Robot,” *IEEE Spectrum,* 2016.

[19] J. Denavit and R. S. Hartenberg, “A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices,” *The Journal of Applied Mechanics,* vol. 22, no. 2, p. 7, 2021.

[20] K. Lynch and F. Park, Modern Robotics: Mechanics, Planning, and Control, Cambridge: Cambridge University Press, 2017.

[21] Robots.com, “RobotWorx, T.I.E Tennessee Industrial Electronics,” Kuka, [Online]. Available: https://www.robots.com/robots/kuka-kr-700-pa. [Accessed April 2023].

[22] L. Hall, “https://www.nasa.gov,” NASA , 7 August 2017. [Online]. Available: https://www.nasa.gov/feature/nasa-looks-to-university-robotics-groups-to-advance-latest-humanoid-robot. [Accessed April 2023].

[23] Open Robotics, “The ROS Ecosystem,” Open Robotics, [Online]. Available: https://www.ros.org/blog/ecosystem/. [Accessed April 2023].

[24] Open Source Robotics Foundation, “Open Robotics,” [Online]. Available: https://www.openrobotics.org/. [Accessed April 2023].

[25] The Construct, “The Construct,” [Online]. Available: https://app.theconstructsim.com. [Accessed April 2023].

7 APPENDIX

C++ Scripts

Plan and execute trajectories for your robot using the MoveIt2 RVIZ environment.

Planning a Trajectory

There is a difference between planning and executing a trajectory. So, in this first part, you will see how to plan a trajectory with C++.

test_trajectory.cpp

```
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>

#include <moveit_msgs/msg/display_robot_state.hpp>
#include <moveit_msgs/msg/display_trajectory.hpp>

#include <moveit_msgs/msg/attached_collision_object.hpp>
#include <moveit_msgs/msg/collision_object.hpp>

#include "rclcpp/rclcpp.hpp"
#include <string>

static const rclcpp::Logger LOGGER = rclcpp::get_logger("move_group_demo");
static const std::string PLANNING_GROUP_ARM = "ur_manipulator";

class TestTrajectory : public rclcpp::Node {
public:
  TestTrajectory(std::shared_ptr<rclcpp::Node> move_group_node)
      : Node("test_trajectory"),
        move_group_arm(move_group_node, PLANNING_GROUP_ARM),
        joint_model_group_arm(
            move_group_arm.getCurrentState()->getJointModelGroup(
                PLANNING_GROUP_ARM)) {

    this->timer_ =
        this->create_wall_timer(std::chrono::milliseconds(500),
                                std::bind(&TestTrajectory::timer_callback, this));

  } // end of constructor
```

```
  // Getting Basic Information
  void get_info() {

    RCLCPP_INFO(LOGGER, "Planning frame: %s", move_group_arm.getPlanningFrame().c_str
());
    RCLCPP_INFO(LOGGER, "End-effector link: %s", move_group_arm.getEndEffectorLink().
c_str());
    RCLCPP_INFO(LOGGER, "Available Planning Groups:");
    std::copy(move_group_arm.getJointModelGroupNames().begin(), move_group_arm.getJoi
ntModelGroupNames().end(),
              std::ostream_iterator<std::string>(std::cout, ", "));

  }

  void current_state() {
    RCLCPP_INFO(LOGGER, "Get Robot Current State");

    current_state_arm = move_group_arm.getCurrentState(10);

    current_state_arm->copyJointGroupPositions(this->joint_model_group_arm,
                                               this->joint_group_positions_arm);
  }

  void plan_arm_joint_space() {

    RCLCPP_INFO(LOGGER, "Planning to Joint Space");

    //joint_group_positions_arm[0] = 0.00;  // Shoulder Pan
    joint_group_positions_arm[1] = -2.50;  // Shoulder Lift
    joint_group_positions_arm[2] = 1.50;  // Elbow
    joint_group_positions_arm[3] = -1.50;  // Wrist 1
    joint_group_positions_arm[4] = -1.55;  // Wrist 2
    //joint_group_positions_arm[5] = 0.00;  // Wrist 3

    move_group_arm.setJointValueTarget(joint_group_positions_arm);

    bool success_arm = (move_group_arm.plan(my_plan_arm) ==
                        moveit::planning_interface::MoveItErrorCode::SUCCESS);

  }

  // Timer Callback function
  void timer_callback() {

    this->timer_->cancel();
    get_info();
    current_state();
    plan_arm_joint_space();
  }

private:
  moveit::planning_interface::PlanningSceneInterface planning_scene_interface;
  std::vector<double> joint_group_positions_arm;
```

```
  moveit::planning_interface::MoveGroupInterface::Plan my_plan_arm;
  rclcpp::TimerBase::SharedPtr timer_;

  moveit::planning_interface::MoveGroupInterface move_group_arm;

  const moveit::core::JointModelGroup *joint_model_group_arm;

  moveit::core::RobotStatePtr current_state_arm;

}; // End of Class

int main(int argc, char **argv) {
  rclcpp::init(argc, argv);
  rclcpp::NodeOptions node_options;
  node_options.automatically_declare_parameters_from_overrides(true);
  auto move_group_node =
      rclcpp::Node::make_shared("move_group_demo", node_options);

  rclcpp::executors::SingleThreadedExecutor planner_executor;
  std::shared_ptr<TestTrajectory> planner_node =
      std::make_shared<TestTrajectory>(move_group_node);
  planner_executor.add_node(planner_node);
  planner_executor.spin();

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

First of all, we define a couple of static variables for the **logger** system and for the **planning group of our arm**, which is named `ur_manipulator`:

```
static const rclcpp::Logger LOGGER = rclcpp::get_logger("move_group_demo");
static const std::string PLANNING_GROUP_ARM = "ur_manipulator";
```

We define a class named `TestTrajectory`, which inherits from `Node`:

```
class TestTrajectory : public rclcpp::Node
```

Next we find the constructor of the class:

```
TestTrajectory(std::shared_ptr<rclcpp::Node> move_group_node)
  : Node("pick_and_place"),
    move_group_arm(move_group_node, PLANNING_GROUP_ARM),
    joint_model_group_arm(
        move_group_arm.getCurrentState()->getJointModelGroup(
            PLANNING_GROUP_ARM)) {

this->timer_ =
    this->create_wall_timer(std::chrono::milliseconds(500),
```

```
                            std::bind(&TestTrajectory::timer_callback, this));
}
```

As you can see, we need to initialize in the constructor 3 things:

- A `Node` objects.
- A `MoveGroupInterface` object, in this case **move_group_arm**.
- A `JointModelGroup` object, in this case **joint_model_group_arm**.

Pay attention to the `MoveGroupInterface`, which allows you to communicate with MoveIt2. It can be set up using just the name of the planning group you would like to control and plan for.

Finally, also inside the constructor, we create a Timer object `timer_`.

Next we find the `get_info()` method:

```
void get_info() {

    RCLCPP_INFO(LOGGER, "Planning frame: %s", move_group_arm.getPlanningFrame().c_str
());
    RCLCPP_INFO(LOGGER, "End-effector link: %s", move_group_arm.getEndEffectorLink().
c_str());
    RCLCPP_INFO(LOGGER, "Available Planning Groups:");
    std::copy(move_group_arm.getJointModelGroupNames().begin(), move_group_arm.getJoi
ntModelGroupNames().end(),
                std::ostream_iterator<std::string>(std::cout, ", "));

}
```

Here we get some data about our MoveIt2 setup. This is not mandatory for the planning, but it can be very useful to make sure everything is configured as it should. Specifically, you are getting data about:

- The name of the frame in which the robot is planning.
- The current end-effector link
- The available Planning Groups

Next we find the `current_state()` method:

```
void current_state() {
    RCLCPP_INFO(LOGGER, "Get Robot Current State");

    current_state_arm = move_group_arm.getCurrentState(10);

    current_state_arm->copyJointGroupPositions(this->joint_model_group_arm,
                                               this->joint_group_positions_arm);
}
```

What we are doing here is to get the current state of the robot. Then we copy the values of the variable `joint_model_group_arm` (which contains the position value of each joint) into the variable `joint_group_positions_arm`.

Next we find the `plan_arm_joint_space()` method:

```
void plan_arm_joint_space() {

    RCLCPP_INFO(LOGGER, "Planning to Joint Space");

    //joint_group_positions_arm[0] = 0.00;  // Shoulder Pan
    joint_group_positions_arm[1] = -2.50;  // Shoulder Lift
    joint_group_positions_arm[2] = 1.50;  // Elbow
    joint_group_positions_arm[3] = -1.50;  // Wrist 1
    joint_group_positions_arm[4] = -1.55;  // Wrist 2
    //joint_group_positions_arm[5] = 0.00;  // Wrist 3

    move_group_arm.setJointValueTarget(joint_group_positions_arm);

    bool success_arm = (move_group_arm.plan(my_plan_arm) ==
                        moveit::planning_interface::MoveItErrorCode::SUCCESS);

}
```

Here we are doing a couple of important things:

- First, we set the desired position values of the joints. In this case, we are setting the joint values of the **home** position.

- Second, we set these new values as the new target using the `setJointValueTarget` method.
- Finally, we call the `plan()` method of the move group interface to plan the trajectory.

Finally we have the `timmer_callback()` method

```
void timer_callback() {

    this->timer_->cancel();
    get_info();
    current_state();
    plan_arm_joint_space();
  }
```

Here we just call the different methods previously defined to perform the planning pipeline. Also, we cancel the timer so that it's only executed 1 time.

At the end of the class, you can find the definition of all the different variables that are needed:

```
private:
  moveit::planning_interface::PlanningSceneInterface planning_scene_interface;
  std::vector<double> joint_group_positions_arm;
  moveit::planning_interface::MoveGroupInterface::Plan my_plan_arm;
  rclcpp::TimerBase::SharedPtr timer_;

  moveit::planning_interface::MoveGroupInterface move_group_arm;

  const moveit::core::JointModelGroup *joint_model_group_arm;

  moveit::core::RobotStatePtr current_state_arm;

};
```

Finally we have the `main()` function:

```
int main(int argc, char **argv) {
  rclcpp::init(argc, argv);
  rclcpp::NodeOptions node_options;
  node_options.automatically_declare_parameters_from_overrides(true);
  auto move_group_node =
      rclcpp::Node::make_shared("move_group_demo", node_options);
```

```
  rclcpp::executors::SingleThreadedExecutor planner_executor;
  std::shared_ptr<TestTrajectory> planner_node =
      std::make_shared<TestTrajectory>(move_group_node);
  planner_executor.add_node(planner_node);
  planner_executor.spin();

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

Here we are creating a ROS2 node and adding this node to a `SingleThreadedExecutor` executor. Then, we spin this executor until somebody terminates the program.

Create the Files and Launch.

a. Inside your **ros2_ws**, create a new package named **moveit2_scripts** with some dependencies.

```
source /opt/ros/foxy/setup.bash
cd ~/ros2_ws/src
ros2 pkg create moveit2_scripts --dependencies rclcpp rclcpp_action moveit_core movei
t_ros_planning moveit_ros_planning_interface interactive_markers geometric_shapes con
trol_msgs moveit_msgs
```

Inside this package, create a new file named **test_trajectory.cpp** and copy the code you've just seen above inside this file.

```
cd ~/ros2_ws/src/moveit2_scripts/src
touch test_trajectory.cpp
```

b. Inside the package, also create a launch file to launch the program, called **test_trajectory.launch.py**.

```
cd ~/ros2_ws/src/moveit2_scripts
mkdir launch
touch launch/test_trajectory.launch.py
```

Paste the code below into this launch file:

test_trajectory.launch.py

```
import os
import yaml
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import Command, FindExecutable, LaunchConfiguration, PathJo
inSubstitution
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare
from ament_index_python.packages import get_package_share_directory


def load_file(package_name, file_path):
    package_path = get_package_share_directory(package_name)
    absolute_file_path = os.path.join(package_path, file_path)

    try:
        with open(absolute_file_path, "r") as file:
            return file.read()
    except EnvironmentError:  # parent of IOError, OSError *and* WindowsError where a
vailable
        return None

def load_file2(file_path):
    """Load the contents of a file into a string"""
    try:
        with open(file_path, 'r') as file:
            return file.read()
    except EnvironmentError: # parent of IOError, OSError *and* WindowsError where av
ailable
        return None


def load_yaml(package_name, file_path):
    package_path = get_package_share_directory(package_name)
    absolute_file_path = os.path.join(package_path, file_path)

    try:
        with open(absolute_file_path, "r") as file:
            return yaml.safe_load(file)
    except EnvironmentError:  # parent of IOError, OSError *and* WindowsError where a
vailable
        return None

def get_package_file(package, file_path):
    """Get the location of a file installed in an ament package"""
    package_path = get_package_share_directory(package)
    absolute_file_path = os.path.join(package_path, file_path)
    return absolute_file_path


def generate_launch_description():
```

```
    declared_arguments = []
    # UR specific arguments
    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 controlle
r.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "safety_k_position",
            default_value="20",
            description="k-position factor in the safety controller.",
        )
    )
    # General arguments
    declared_arguments.append(
        DeclareLaunchArgument(
            "description_package",
            default_value="ur_e_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="ur3e_robot.urdf.xacro",
            description="URDF/XACRO description file with the robot.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
            "moveit_config_package",
            default_value="my_moveit2_config",
            description="MoveIt config package with robot SRDF/XACRO files. Usually t
he argument \
        is not set, it enables use of a custom moveit config.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument(
```

```
            "moveit_config_file",
            default_value="ur3e.srdf",
            description="MoveIt SRDF/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' confi
guration \
        have to be updated.",
        )
    )
    declared_arguments.append(
        DeclareLaunchArgument("launch_rviz", default_value="true", description="Launc
h RViz?")
    )

    # Initialize Arguments
    safety_limits = LaunchConfiguration("safety_limits")
    safety_pos_margin = LaunchConfiguration("safety_pos_margin")
    safety_k_position = LaunchConfiguration("safety_k_position")
    # General arguments
    description_package = LaunchConfiguration("description_package")
    description_file = LaunchConfiguration("description_file")
    moveit_config_package = LaunchConfiguration("moveit_config_package")
    moveit_config_file = LaunchConfiguration("moveit_config_file")
    prefix = LaunchConfiguration("prefix")
    launch_rviz = LaunchConfiguration("launch_rviz")

    robot_description_content = Command(
        [
            PathJoinSubstitution([FindExecutable(name="xacro")]),
            " ",
            PathJoinSubstitution([FindPackageShare(description_package), "urdf", desc
ription_file]),
            " ",
            "safety_limits:=",
            safety_limits,
            " ",
            "safety_pos_margin:=",
            safety_pos_margin,
            " ",
            "safety_k_position:=",
            safety_k_position,
            " ",
            "name:=",
            # Also, ur_type parameter could be used, but then the planning group name
s in YAML
            # configs have to be updated!
            "ur",
```

```
            " ",
            "prefix:=",
            prefix,
            " ",
        ]
    )
    robot_description = {"robot_description": robot_description_content}

    # MoveIt Configuration
    srdf_file = load_file('my_moveit2_config', 'config/ur3e.srdf')
    robot_description_semantic = {"robot_description_semantic": srdf_file}

    kinematics_yaml = load_yaml(
        "my_moveit2_config", "config/kinematics.yaml"
    )

    # MoveGroupInterface demo executable
    move_group_demo = Node(
        name="test_trajectory",
        package="moveit2_scripts",
        executable="test_trajectory",
        output="screen",
        parameters=[
            {'use_sim_time': True},
            robot_description,
            robot_description_semantic,
            kinematics_yaml,
            ],
    )

    nodes_to_start = [move_group_demo]

    return LaunchDescription(declared_arguments + nodes_to_start)
```

As you can see, the launch file is very similar to the one used for launching MoveIt2. You are loading the same parameters, and the only difference is that you are now launching our C++ script. Also, it's unnecessary to start RVIZ this time since you already have it running.

```
move_group_demo = Node(
    name="test_trajectory",
    package="moveit2_scripts",
    executable="test_trajectory",
    output="screen",
    parameters=[
        {'use_sim_time': True},
        robot_description,
        robot_description_semantic,
        kinematics_yaml,
        ],
)
```

c. Now, update the **CMakeLists.txt** file to compile everything. First, open the file and add the following lines to the code:

Add to CMakeLists.txt

```
# Generate the executable
add_executable(test_trajectory
    src/test_trajectory.cpp)
target_include_directories(test_trajectory
    PUBLIC include)
ament_target_dependencies(test_trajectory
    ament_cmake
    rclcpp
    rclcpp_action
    moveit_core
    moveit_ros_planning_interface
    interactive_markers
    moveit_ros_planning
    control_msgs)

# Install the executable
install(TARGETS
    test_trajectory
    DESTINATION lib/${PROJECT_NAME}
)

# Install the launch file
install(DIRECTORY
    launch
    DESTINATION share/${PROJECT_NAME}
)
```

You are generating an executable from the file **test_trajectory.cpp** and installing it alongside the contents of the launch folder.

d. Now compile our workspace.

```
cd ~/ros2_ws
colcon build
source install/setup.bash
```

e. Now test your new program! First, you'll need to launch the MoveIt2 RVIZ environment.

```
ros2 launch my_moveit2_config my_planning_execution.launch.py launch_rviz:=true
```

f. Now, run the code:

```
source ~/ros2_ws/install/setup.bash
ros2 launch moveit2_scripts test_trajectory.launch.py
```

You will see how the plan to the **home** position is computed.

Executing a trajectory

So, at this point, you've seen some methods that allow you to plan a trajectory with C++ code. But what about executing this trajectory?

You need to call the `execute (plan)` function from the planning group to execute a trajectory. For instance, given a plan `my_plan`, you would execute the planned motion like this:

```
move_group_arm.execute(my_plan_arm);
```

By executing this line of code, you will be telling your robot to **Execute** the trajectory stored in the `my_plan` variable.

Also, there's the option to **Plan & Execute** a motion, just as the GUI provides.

```
move_group_arm.move();
```