Final Master's Tesis

Double Master's degree in Industrial Engineering and in Automatic Control and Robotics

Robot Agnostic Interface for Industrial Aplications

MEMORY

Author:	Álvaro Ibáñez Moreno
Supervisor:	Cecilio Angulo Bahón
Academic Year:	2022-23

ETSEIB

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Summary

The quick evolution of robotic arms has generated many manufacturers of robotic arms, such as Universal Robots, ABB, or Fanuc. Each manufacturer offers a unique interface to program and control their robots. This can limit companies choices when selecting a suitable robot for their industrial operations, as they will choose an interface that doesn't require new training. For that reason, and based on the experience at UPC CIM, this project will focus on creating a common interface for robotic arms.

The main objectives are to produce an interface to simulate robots from different manufacturers, save and load data, and create a simple scripting language. By using ROS, an open-source software infrastructure to communicate between different robotic elements, and Python, the code will be created in five different modules: the launch application, obtaining information about the robot, editing files, moving the robot, and scripting actions.

To test the resulting interface, first a setup sequence is performed to see the limitations of the interaction. Then, three theoretical scenarios are proposed, and a scripting sequence is created for each one: Pick and Place, Sorting, and Bin Picking.

While limited in some aspects, the application performs as expected and offers the basic options to solve many robot implementations. New options for the future of robot interaction are open with this project, as people could also further develop this program if considered.

ETSEIB

Contents

ETSEIB

List of Figures

List of Tables

Listings

1 Introducction

A Bit of History

The Industrial Revolution of the 18th and 19th centuries brought significant advancements in mechanical engineering, leading to the creation of various machines and automated systems. During this period, robotic arms began to emerge as tools for automating repetitive tasks in manufacturing industries.

The 20th century witnessed remarkable progress in robotics and the birth of modern robotic arms. In the 1950s, George Devol and Joseph Engelberger introduced the first digitally operated robotic arm. This breakthrough marked the beginning of an era where robotic arms played an integral role in automation.

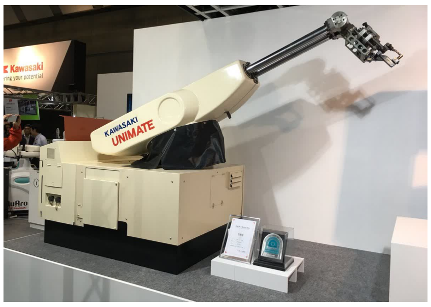


Figure 1: Unimate, the robot developed by George Devol, [16]

As technology advanced, robotic arms became more versatile and capable. In the 1960s, General Electric developed the first electrically powered robot, the GE 400 Series. This robot had six axes of motion, providing greater flexibility and dexterity. Around the same time, researchers at Stanford University developed the Stanford Arm, which utilized hydraulics to achieve precise movements.

The advent of microprocessors and computer control in the 1970s and 1980s brought significant advancements to robotic arms. These arms could be programmed and controlled with greater precision, making them ideal for applications in assembly lines, welding, and material handling. Companies like ABB, Fanuc, and KUKA emerged as leaders in the industrial robotics market, producing robotic arms capable of performing complex tasks with high accuracy and repeatability.

Figure 2: Fanuc Robots in an Industrial Environment, [17]

Today, robotic arms continue to evolve, driven by advancements in artificial intelligence, sensors, and materials. Collaborative robots, or cobots, are becoming increasingly popular, working alongside humans in a shared workspace. With improved sensing capabilities and advanced control algorithms, these robots can perform intricate tasks while ensuring the safety of human workers.

More Common, Easier to Programm

The recent developments in robotics have allowed them to stop being a luxury item reserved only for big companies with large production lines and become more affordable and manageable for medium and small companies as well.

The overall reduction in costs was caused by the improvement of manufacturing technologies and the scalability of the robots, as more models, sizes, and capabilities allowed for the selection of the robot that fit the best in the planned environment.

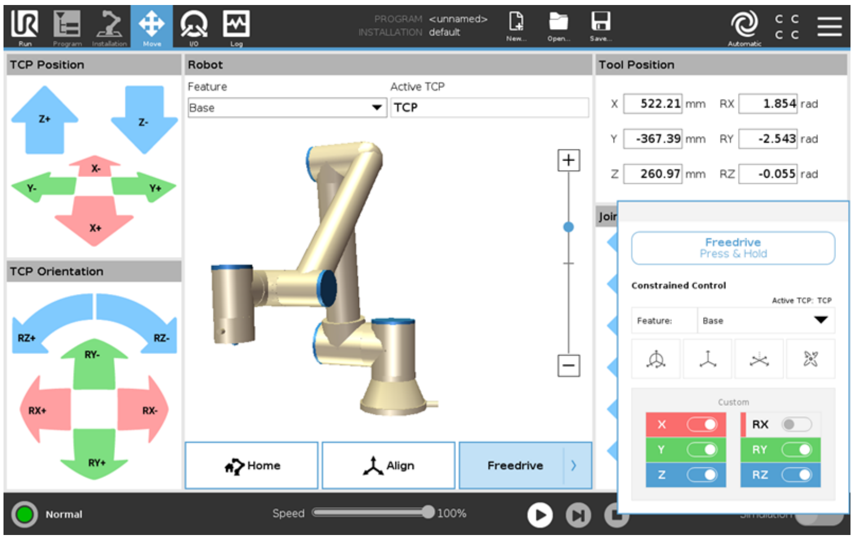
On the other hand, accessibility was improved with the simplification of the installation and configuration processes. User-friendly interfaces and standardized connections reduced the need for a highly qualified technician and allowed anyone, given a little training, to set up, program, and use the robot in the target environment.

1.1 State of the Art

Accessible Automation

Universal Robots (UR) is a leading manufacturer and pioneer in collaborative robotic technology. It specializes in the design and production of lightweight, flexible, and user-friendly industrial robotic arms. The company's mission is to democratize automation by making robotic solutions accessible to businesses of all sizes.

The collaborative robots, also known as cobots, offered by Universal Robots come in various models with different payload capacities, allowing businesses to select the robot that best matches their specific needs. The cobots can perform tasks such as pick-and-place operations, assembly, machine tending, quality inspection, packaging, and more. They can be easily integrated into existing workflows thanks to their plug-and-play functionality and compatibility with a wide range of accessories and tools. They are also designed to work alongside humans without the need for safety barriers, enabling safe and efficient human-robot collaboration.



Feature: Base
Active TCP: TCP

X	522.21	mm	RX	1.854	rad
Y	-367.39	mm	RY	-2.543	rad
Z	260.97	mm	RZ	-0.055	rad

Active TCP: TCP
Feature: Base

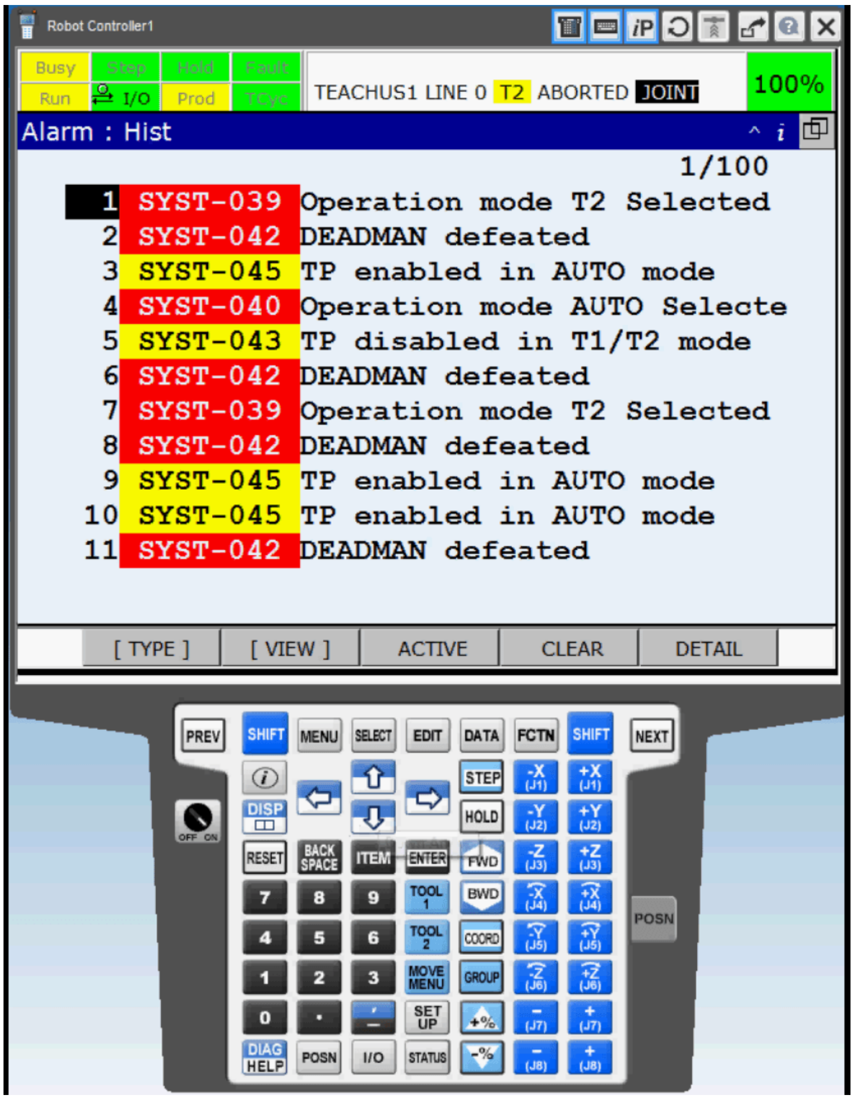
Figure 3: UR Polyscope interface, [18]

Universal Robots' cobots are known for their user-friendly programming. To program a Universal Robots (UR) robot, you can use the UR teach pendant, a handheld device with an intuitive graphical user interface (GUI). With the teach pendant, you can manually guide the robot through the desired movements and record waypoints to create a program. Additionally, UR robots support script programming using a simplified version of Python. This allows for more complex logic and customization. Universal Robots also provides powerful software, such as the UR+ platform, which offers a wide range of pre-built software and hardware components that can be easily integrated with their cobots.

With their innovative technology and emphasis on accessibility, Universal Robots have played a significant role in revolutionizing the automation landscape. Their cobots have helped businesses of all sizes increase productivity, improve efficiency, and enhance worker safety.

Other Robot Interfaces

FANUC Corporation is a global leader in the manufacturing of industrial robots, renowned for their high performance, reliability, and precision. FANUC robots are widely used in industries such as automotive, electronics, aerospace, and more. They offer a diverse range of robot models, including articulated robots, delta robots, and collaborative robots (cobots). FANUC robots excel in speed, accuracy, and repeatability, enabling them to handle complex tasks with efficiency. With user-friendly programming interfaces and advanced control systems, FANUC robots are easy to integrate into production lines and adapt to changing needs. FANUC's commitment to innovation and quality has made them a trusted choice for industrial automation worldwide.


Robot Controller1

TEACHUS1 LINE 0 T2 ABORTED JOINT 100%

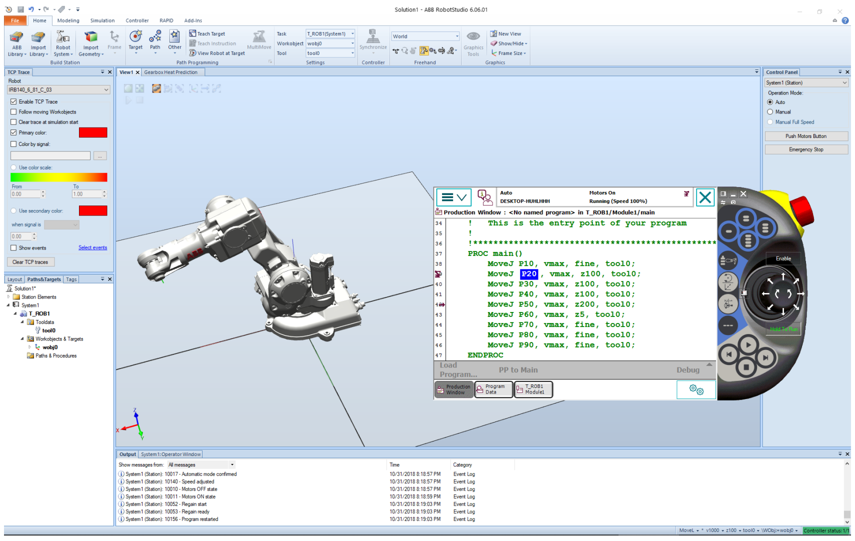
Alarm : Hist

```
                              1/100
 1 SYST-039 Operation mode T2 Selected
 2 SYST-042 DEADMAN defeated
 3 SYST-045 TP enabled in AUTO mode
 4 SYST-040 Operation mode AUTO Selecte
 5 SYST-043 TP disabled in T1/T2 mode
 6 SYST-042 DEADMAN defeated
 7 SYST-039 Operation mode T2 Selected
 8 SYST-042 DEADMAN defeated
 9 SYST-045 TP enabled in AUTO mode
10 SYST-045 TP enabled in AUTO mode
11 SYST-042 DEADMAN defeated
```

Figure 4: Fanuc Teach Pendant interface, [19]

To program a FANUC robot, you typically use the proprietary programming language called FANUC Robotics TP (Teach Pendant) or KAREL (Kawasaki Advanced Robot Language). Using the teach pendant, you can manually guide the robot through the desired motions and record these movements as part of the program. Alternatively, you can write TP or KAREL code directly on the teaching pendant or through offline programming software. The code instructs the robot on specific tasks, such as movements, interactions with peripherals, and logic operations. Once the program is completed, it can be executed, modified, or saved for future use.

ABB is a leading manufacturer of industrial robots that are widely recognized for their precision, reliability, and versatility. ABB robots are designed to perform a wide range of tasks in various industries, including automotive, electronics, pharmaceuticals, and more. They are known for their high-speed operation, advanced motion control, and superior accuracy. ABB offers a diverse portfolio of robot models, including articulated robots, SCARA robots, and collaborative robots (cobots), catering to different payload capacities and application requirements. With user-friendly programming interfaces, ABB robots can be easily integrated into existing workflows and controlled efficiently. ABB's robotic solutions empower businesses to optimize productivity, improve quality, and enhance overall manufacturing efficiency.



Production Window : <No named program> in T_ROB1/Module1/main

```
    !   This is the entry point of your program
    !
    !*******************************************
    PROC main()
        MoveJ P10, vmax, fine, tool0;
        MoveJ P20, vmax, z100, tool0;
        MoveJ P30, vmax, z100, tool0;
        MoveJ P40, vmax, z100, tool0;
        MoveJ P50, vmax, z200, tool0;
        MoveJ P60, vmax, z5, tool0;
        MoveJ P70, vmax, fine, tool0;
        MoveJ P80, vmax, fine, tool0;
        MoveJ P90, vmax, fine, tool0;
    ENDPROC
```

```
Show messages from: All messages                              Time                    Category
System1 (Station): 10017 - Automatic mode confirmed           10/31/2018 8:18:57 PM   Event Log
System1 (Station): 10140 - Speed adjusted                     10/31/2018 8:18:57 PM   Event Log
System1 (Station): 10010 - Motors OFF state                   10/31/2018 8:18:57 PM   Event Log
System1 (Station): 10011 - Motors ON state                    10/31/2018 8:18:59 PM   Event Log
System1 (Station): 10052 - Regain start                       10/31/2018 8:19:03 PM   Event Log
System1 (Station): 10053 - Regain ready                       10/31/2018 8:19:03 PM   Event Log
System1 (Station): 10156 - Program restarted                  10/31/2018 8:19:03 PM   Event Log
```

Figure 5: ABB RoboStudio software interface, [20]

To program an ABB robot, you can use the ABB RobotStudio software. It offers a user-friendly interface for offline programming and simulation. You can create robot programs by dragging and dropping commands or by writing ABB's proprietary programming language, RAPID, which is similar to C++. The software allows you to define robot movements, logic operations, and interactions with peripherals. After programming, you can simulate and validate the robot's behavior virtually. Once satisfied, the program can be transferred to the ABB robot controller for execution. ABB RobotStudio simplifies the programming process and enables efficient integration of ABB robots into various applications.

ROS and ROS Industrial

ROS, which stands for Robot Operating System, is an open-source framework widely used in the field of robotics for building and controlling robotic systems. ROS provides a collection of software libraries, tools, and conventions that aid in the development of robotic applications. It offers a flexible and modular architecture, enabling seamless communication between various components of a robotic system.

ROS provides a wide range of functionalities, including hardware abstraction, device drivers, messaging systems, visualization tools, and more. It allows developers to write robot control software in various programming languages, such as C++ or Python, for example. ROS also offers a rich ecosystem of packages and libraries contributed by a large community of developers, which accelerates the development process by providing ready-to-use components and functionalities.

ROS-Industrial (ROS-I) is an extension of ROS, specifically focused on industrial automation and robotics. It aims to bridge the gap between traditional industrial systems and the ROS framework. ROS-I provides a set of packages, tools, and best practices tailored for industrial applications, making it easier to integrate ROS into industrial environments.

ROS-I offers standardized interfaces and drivers for industrial robots, allowing seamless communication and control of various robot models and brands. It provides integration with industrial hardware, such as sensors, grippers, and programmable logic controllers (PLCs). ROS-I also facilitates interoperability with other industrial systems, such as manufacturing execution systems (MES) and enterprise resource planning (ERP) systems.

The adoption of ROS and ROS-I has enabled significant advancements in industrial automation. They have simplified the development process, reduced integration efforts, and promoted the reusability of software components.

1.2 Motivation

CIM Looming Factory

During the course of this project, I had the opportunity to briefly work on the Looming Factory project, carried out by the CIM foundation, which is part of the UPC group.

The CIM foundation is an entity that has as its mission to transmit technology and tools to companies to create and improve their products and manufacturing processes. By means of I+D+i projects, it generates new product and process technologies, which it then incorporates into its formative courses.

One of these projects is the Looming Factory project. Its objectives are focused on grouping, consolidating, and directing the current research in the 4.0 Industry at the different research centers in Catalonia. To do so, different experiments and exhibits are carried out in order to digitize environments, integrate big data technologies, implement cyber-physical hybridization, implement cloud technologies, or achieve energetic and economic optimization.

Problem Description

The exhibit on which I collaborated aimed to integrate a CNC machine, an AGV, a robotic arm, and a PLC. Different university departments worked on different parts, so a level of coordination was required. The main objective was also to make all parts as independent of each other as possible, so if a company would like to implement a similar system, they could do it with the appropriate resources for the application.

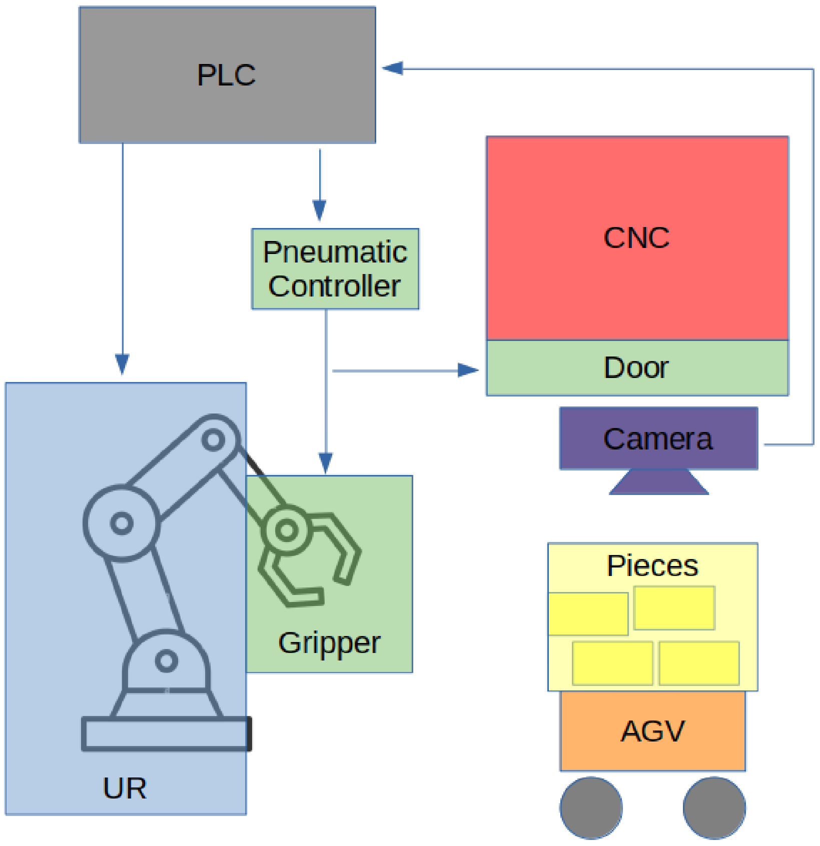


Figure 6: Looming Factory project description diagram

The AGV would bring an unsorted set of pieces in a box to machine in the CNC, and the robotic arm would have to pick one to load into the CNC. To do so, a camera attached to the robotic arm would capture the position of the pieces, and using a computer vision algorithm, the position of the closest piece would be given to the robot. Once the robot has picked the piece and loaded it into the CNC, the doors are shut, and the robot waits for the machining process to finish. When the doors open, the robot picks up the piece and places it in another box. The whole process is managed by a PLC to which all components are connected, as well as a pneumatic system to open and close the door and activate the suction gripper on the robotic arm.

My main function was to set up the robot in order to be able to move it on command from the PLC. It would also have to receive information about the piece's location and move to it.

Obstacles and Problems

With so many parts having to work in a synchronized manner, it comes as no surprise that many obstacles and problems arose during the development of the project. The robot chosen for this application was a UR10 robot, and while the simplicity of the Polyscope interface facilitated some things, it also made others more complicated.

The Polyscope interface, characteristic of the UR series, greatly helped in storing and changing the different positions the robot had to navigate. Also, the simple programming language allowed the robot to execute different sequences of movements depending on the values set by the PLC.

But, in order to integrate the robot with the camera, a more complex datatype containing the pose calculated by the camera had to be used. More over, a change of frame of reference and an inverse kinematics computation had to be carried out, and the Polyscope interface fell short to do that.

Finally, by using the UR interface, we were limiting the implementation to UR robots. In order to allow the use of other Robotic arms, a program independent of the robot manufacturer would need to be written. Even if this objective was more of an extra, I found it interesting to have a common interface for different Robots. This way, Robot operators would only require training in a single system, and not for every robot manufacturer.

1.3 Objectives

Robot Independent Interface

The first objective of this project is to create an interface to control any type of robotic arm. Currently, all major robot manufacturers provide a custom interface in order to interact with their robots. This causes operators to need to learn a new interface for each type of robot they encounter. This also limits the selection pool for new robots, as companies would always prefer a robot with which the operators are already familiar.

With a common interface, training costs would be reduced, and the new robot selection pool would expand.

Easy Storing and Loading of Data

As part of the interaction of the user with the robot, a system to store, load, and change data will be set up. In order to facilitate the data exchange and also allow for fine-tuning without much hassle, functions to import from a text file will be set in a simple language.

Simple Script Language

Finally, the main feature the interface has to have is a simple scripting language. Apart from the common functions to move and compute trajectories, the main feature that opens up the interface is the capacity to work cyclically and according to input parameters. A simple scripting language allows for more flexible automation and reduces the level of supervision robots need, as they can work independently of the operators if all major cases are taken into account.

1.4 Scope

Robot Simulation

Given the limited access to real robots, a simulation of those robots will be set up to mimic their behavior. ROS Industrial offers a variety of 3D robot models and the required parameters in order to simulate as close to reality as possible. Also, given that ROS Industrial also provides drivers to control the real robots and that the communication between a real robot and the simulated robot is compatible, changing from a simulation to the real robot shouldn't present much of a problem if, in future work, this project is expanded.

Input/Output Signals

There are numerous possible communication devices, each with its own communication protocol or system. It is unrealistic to try to implement communication protocols when there is no actual hardware available and knowledge about them is limited. Data type transformations, connections to servers or master-slave systems, and sending and receiving data also deviate from the approach to this project. A more solid case analysis would be needed to implement this function.

For these reasons, communication with external elements will also be simulated by using a Python script. The information that would be sent will be written into a file, and the communication simulation script will read the file and modify it accordingly with the data that would be sent to the program.

Simple UI

The user interface will be simplified, as the core objective is to create an interface, but not necessarily an eye-catching interface. The interaction with the program will be done using the terminal command line, to which the user will input the menu option that wants to be accessed. However, the functions and objects will be set in place to facilitate the integration of a more pleasant UI. By associating the functions with the interactive objects, the UI can be set without many complications.

2 Resources

2.1 Linux-Ubuntu

The choice of the OS is conditioned by the version of ROS that wishes to be used, as currently two versions of ROS are being maintained. ROS 1's latest release is targeted at the Ubuntu 20.04 (Focal) release, while ROS 2's latest version can be installed on Ubuntu 22.04 (Jammy) and Windows 10.

As it will be discussed later, the use of ROS 1 is preferred, so the Ubuntu Focal Release is chosen as the OS. Ubuntu Focal is the previous LTS (Long Time Service) version of the OS, released on April 23, 2020, with an end of support planned for April 2025. LTS versions of Ubuntu are stable releases that will be maintained for 5 years since the initial release. There is a LTS version every 2 years, the latest one being Ubuntu 22.04 (Jammy), released on April 21, 2022.

Ubuntu is a popular open-source operating system based on the Linux kernel. It is known for its user-friendly interface, stability, and security. Ubuntu provides a complete software ecosystem, including a wide range of applications and tools, and offers regular updates and long-term support. It promotes free and open-source software principles, which allow for flexibility and customization.

2.2 ROS

ROS, which stands for Robot Operating System, is an open-source framework widely used in the field of robotics for building and controlling robotic systems. ROS provides a collection of software libraries, tools, and conventions that aid in the development of robotic applications. It offers a flexible and modular architecture, enabling seamless communication between various components of a robotic system.

ROS provides a wide range of functionalities, including hardware abstraction, device drivers, messaging systems, visualization tools, and more. It allows developers to write robot control software in various programming languages such as C++, Python, and more. ROS also offers a rich ecosystem of packages and libraries contributed by a large community of developers, which accelerates the development process by providing ready-to-use components and functionalities.

ROS versions

The ROS project started back in 2007, and a lot has changed since then in the robotics community. The ROS-2 project started in 2015 to accommodate new trends in the robotics world. Instead of being a common update to the ROS environment, ROS-2 goes a step further and has been rewritten from the ground up. The main differences between the two systems are as follows:

	ROS-1	ROS-2
Communication Protocol	TCPROS protocol based on TCP/IP	Data Distribution Service (DDS) protocol
Scalability	Designed for single-host systems	Built with a more distributed architecture
Real-time Capabilities	Lacks native support	Support through the use of DDS
Security	Limited built-in security features	Security as a core design principle
Development and Ecosystem	Well-established, wide range of libraries, packages, and tools	Gradually growing, fewer packages and libraries

Table 1: Main ROS-1 and ROS-2 differences

Given that the improvements offered by ROS-2 don't add any direct value to this project and that the ROS-2 number of packages and libraries is considerably smaller than ROS-1, it has been decided to use ROS-1 for this project. Another reason for this choice is that I am already familiar with ROS-1, and ROS-2 would require a considerable update in my knowledge of the system.

ROS-1, however, will not be updated any further, but it will be maintained until May 2025. This last version that will be used is called ROS Noetic Ninjemys. The ROS development team recommends a gradual migration to ROS-2 and offers different tools to ease the process. If this project has to be updated to ROS-2, it shouldn't be complicated, as it is envisioned to work with the common ROS features.

Basic Features

Nodes: Represent the different processes that are being run in the ROS framework. Nodes are the main feature in ROS programming, as most of the code is written for a node that communicates with other nodes and takes actions based on the information provided.

Topics: They are data buses with names that nodes use to communicate with each other. Nodes may publish to a topic to send information, or subscribe to it to read the information it contains. The content handled by topics can range from simple bits to complex data structures.

Services: Refer to the different actions that a node offers to do. Nodes advertise their services and perform an action that yields a single result when called upon. They are meant to be one-time actions, like capturing an image frame or reading the current sensor, rather than continually processing commands.

Parameter Server: A shared database that all nodes can access. It contains static information that does not usually change over time (like the robot geometry or the controller parameters) and is loaded when launching the application.

Utilities

Rviz: 3D visualization tool that allows the display of robots, environments, and sensor data.

Rosbag: Useful tool for recording and playing back historical data published on ROS topics.

Catkin: The build system for ROS. Based on Cmake, it allows for the building, testing, and packaging of software.

Rosbash: Expansion of the bash shell's functionality. Adds functions and improves some of the default ones.

Roslaunch: tool used to launch multiple nodes with the same file and load parameters to the Parameter Server.

Additional Packages

Actionlib: A standardized interface for services that can be preempted.

MoveIt!: Motion planning package for robotic manipulators.

tf2: System to represent, tack, and transform coordinate frames.

gazebo_ros: Integration of ROS with the Gazebo Simulator.

ros_control: includes controller interfaces, controller messages, transmissions, and hardware interfaces.

trac_ik: a library for solving generic Inverse Kinematics.

RQT: GUI for development in ROS. Offers many visualization tools for different parameters.

2.3 Programming Languages

C++: The main body of ROS is written in C++. It is a high-level, multipurpose language designed for high performance, flexibility, and efficiency. A basic knowledge of this language is required if some functions or ROS need to be analyzed in detail.

Python: General-purpose language focused on readability. It avoids premature optimization, which allows it to be more flexible and forgiving with data operations. The main body of the project will be written in Python in order to simplify readability, error handling, and debugging. Python also provides multiple packages in the default installation that expand the possibilities offered. In recent years, it has grown into one of the most popular programming languages for its portability and flexibility.

XML: Markup language and file format for storing, transmitting, and reconstructing arbitrary data. It is used in a variety of places in the ROS architecture.

The ROSlaunch utility uses XML files to write roslaunch files. It can load parameters to the server, configure the startup of a node, or load other XML files, such as other roslaunch files or, for example, urdf files.

URDF (Unified Robot Description Format) is a file format based on XML used for the description of the geometry of robots. It can be used to obtain the kinematic parameters for calculations or to load a model into a simulator.

Xacro is an XML macro language used to ease the construction of XML files. In ROS is commonly used to transform a set of parameters into an URDF file.

2.4 Robots

The ROS Industrial Package offers a wide pool of robot models to work with. UR, Fanuc, and ABB, for example, are some of the possible robot models that can be used. Each manufacturer can also decide which information is made public, so not all robots have a complete set of parameters ready for simulation. The ROS-I package is mainly focused on real applications, and manufacturers offer their own simulation software as a service. The UR series is the only robot that has published the physical characteristics of its robots, such as their links' mass and inertia, as well as the controllers and files necessary for simulation. The main focus will be on using different UR models and also trying to complete some of the missing files for other models. Specific values will not be accurate, and therefore the simulation will not be precise, but a rough simulation and control can be achieved.

2.5 Other Resources

Atom: Integrated development environment (IDE) that supports multiple languages and a variety of plugins.

Terminator: Terminal Interface that simplifies working with multiple terminals by juxtaposing them in a single window.

Git: Version control system to keep track of code modifications as well as store data in the cloud. Simplifies the task of working on multiple computers.

3 Implementation

3.1 Structure

Given the objectives set, the environment, and the resources given, the code length can be roughly estimated. The size is planned to be too big for a single code file to hold it all and have a clear organization. For this reason, the code is decidedly segmented into modules, each containing a specific set of functions for a specific purpose.

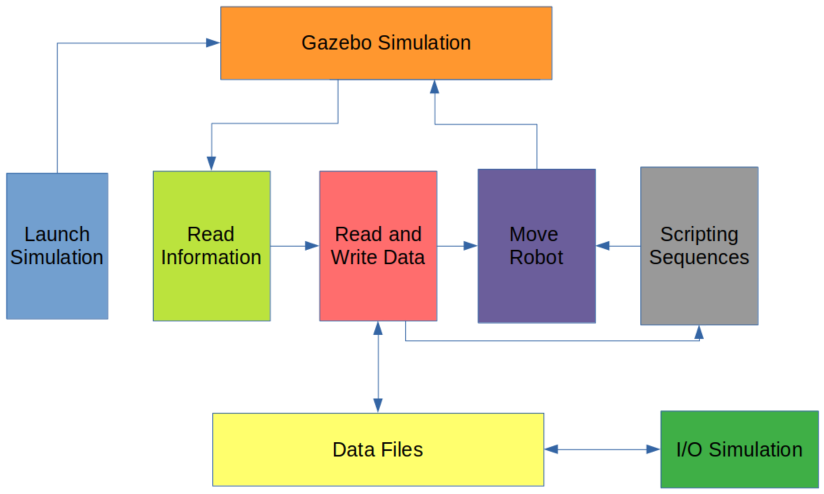


Figure 7: Program Structure diagram

The program will contain a main menu used to access the rest of the functionalities. In the first menu, the robot will be selected, and the simulation will start. From there, the next module will focus on accessing information about the robot, like the joint states, the pose of the Tool Center Point (TCP), the robot geometry to compute the Inverse Kinematics (IK) or the parameters to configure JointState and Pose ROS datatype objects given input data.

Once we are able to read information about the robot, the next step will be storing and reading that information. Data will be stored in a Python Dictionary, and functions to load, save, and modify this data will be set. To ease the setup of data, a tool is needed to read information from a text file and transform it into a Python Dictionary.

The next step is moving the robot to, for example, the positions stored previously. The module will offer the basic functions to compute trajectories and publish those to the execution server. Also, a set of utility functions will be set up to ease use.

Finally, with the robot being able to move, a Scripting Module will be set in place. The module will read the corresponding necessary files and execute the script. It will be able to read and modify the stored data in order to act according to external signals. To simulate the input and output signals, a script that simulates the behavior and makes sequential modifications to the data based on the existing data will be executed in parallel.

3.2 Launching the Simulation

Launch Files

ROS offers a tool named roslaunch, which uses XML files to start different programs, load variables, or load other XML files at the same time. With this tool, it becomes much easier to configure the multiple elements that must be started in order to start a ROS program.

The most commonly used functions in launch files are loading arguments into the parameter server, configuring and launching ROS nodes, and loading other launch files. With this combination of functions, one can have generic launch files and load specific configurations depending on the variables used. This way, every robot can have simple launch files that call generic files to start the application. This is also useful for debugging, as solving an error solves it for all the robots that use that file.

In the main menu, the robot will be selected, and when the simulation is started, the launch file will be set and executed. A series of parameters need to be set to properly interact with ROS from Python, and after that, the rest of the initial Python configuration is set. Finally, the main menu is shown.

Launch Configuration

One of the main objectives of this project is to be able to use different robots with the same interface. For that reason, it is necessary to load the same information for each robot.

The ROS Industrial package offers a variety of robot models, launch files, configurations, and drivers. The most complete one is the package belonging to the UR series, which offers ready-to-use launch files for the real robot and the simulation in Gazebo. In order to simulate other robots, it will be necessary to set the launch files necessary to start the simulation with them. To create these files, it is taken as a base the structure used in the UR series launch files.

```
robot_bringup.launch
 ├─ Include:  load_robot.launch.xml (Robot Description)
 │   └─ Param:  Execute robot.xacro
 │       ├─ Load:  robot_macro.xacro (From sim package)
 │       │   ├─ Load:  robot_macro.xacro (From description package)
 │       │   ├─ Load:  robot_transmissions.xacro (From sim package)
 │       │   ├─ Param:  Link Configurations
 │       │   └─ Param:  Gazebo plugin
 │       └─ Param:  world, tool0 (Links)
 ├─ Node:  Robot State Publisher
 └─ Include:  robot_control.launch
     ├─ Rosparam:  robot/type_controllers.yaml (Controller Config File)
     ├─ Include:  Launch empty Gazebo world
     ├─ Node:  Spawn Model
     ├─ Node:  Load and Start Ros Controllers
     └─ Node:  Load Stopped Ros Controllers
```

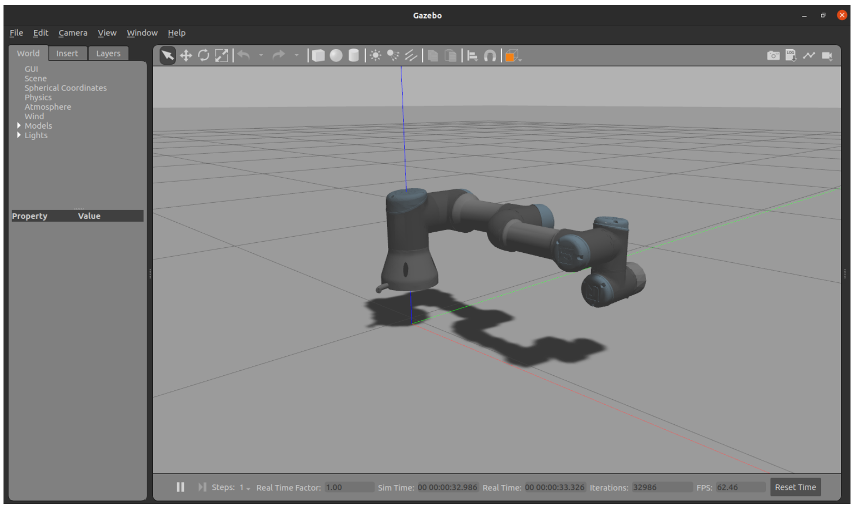
Figure 8: File Structure for Launch Files

The main launch file for each robot has three functions: load the file that loads the robot description into the parameter server, start the Robot State Publisher node that gives access to the current position of the actuators of the robot, and load the file that starts the gazebo simulation.

The robot description is loaded by executing a xacro command that returns the parameter to load.

The xacro file loads a macro to compute the robot description and adds links to join the robot to the world and the TCP to its controller. The robot description macro adds the geometry description given by the robot description macro, adds the description of the transmissions, configures the collision of the links for gazebo, and loads the gazebo plugin to use Ros_control.

The second file, first, loads into the parameter server the parameters described in the controller configuration file. It then loads the launch file provided by the Gazebo package, which starts a gazebo simulation with an empty environment. Finally, it starts nodes to spawn the robot model into the simulation, to load and start the controllers, and to load but not start the controllers that are not needed at the start.



```
Steps: 1   Real Time Factor: 1.00   Sim Time: 00 00:00:32.986   Real Time: 00 00:00:33.326   Iterations: 32986   FPS: 62.46
```

Figure 9: Gazebo whith a UR3 loaded when the launch files are correctly configured

Creating Launch Files

The ROS Industrial packages for the UR series contain the file structure shown previously and allow running simulations of the robots. Unfortunately, not all the robots provided offer a complete set of files. For example, the Fanuc package is not conceived for simulation but only to directly control the real robot. For this reason, specific files to launch a simulation of these robots need to be set up.

Fanuc provides the files to compute the macro that loads the robot's geometric description. Apart from that, the rest of the files must be manually created. Firstly, the generic files to load the simulations and the transmissions are set. Then, for each robot, a macro file that joins the geometry description and the rest of the gazebo parameters is set. Also, for each robot, a xacro file that loads the previous macro and joins it with the additional links is also set. Next, the file that executes the xacro command is set for each robot. Finally, controllers for each robot are written, and everything is wrapped in a single launch file for each robot.

When writing the transmission interface and the controllers, these must share the type of transmission, be it position, velocity, or effort. Also, Fanuc does not provide the inertial values for the links, which need to be added in order to be able to run in Gazebo. For that reason, the PID values set for the controllers can not be calibrated, but are set for the simulation to run. The inertial values are set to generic values, so the control of the Fanuc robots is expected to not be accurate.

3.3 Obtaining Information about the Robot

Once the simulation has started, information from the robot starts to be published into different topics by the Robot State Publisher node started by the launch file. The information published then needs to be accessed and treated in order to standardize its use by other functions, to store it, or to simply present it to the user.

Current Joint State and Pose

ROS offers two standard data types useful for representing the current state of the robot. The first one is the JointState (JS) type, which contains the name of the joints, the position value of the actuators, and can contain their velocity and effort but are not required parameters. The second main data type is the Pose type. It contains the information of a point in space in Cartesian coordinates x, y, and z and its orientation in x, y, z, and w. It will be useful to store the position of the TCP.

The procedure to obtain the current Joint State is very straightforward, as the data type published is the same as the type that will store the information. Simply by reading the published information, the values can be obtained. From this information, another useful function is set that simply returns the joint names.

Obtaining the Pose of the TCP needs a bit more work as the information is not directly published. What is published is a set of transformations between all the Cartesian frames set in the geometry description file of the robot. Every object has a reference frame from which its geometry is defined. Then these frames are placed relative to the previous frame in the kinematic chain. The frames of interest are the world frame, which is used as the global origin of coordinates, and the tool0 frame, which represents the position of the TCP. With the use of the tf2 library, the current frame transformations are read and stored in a Buffer, and from that buffer, the specific transformation from the world frame to the tool0 frame is obtained. These transformation values are then set to a pose object to obtain the TCP coordinates relative to the origin.

Joint State and Pose from List of Values

These utility functions will simplify translating a list of values into a JointState or Pose object. For the JointState, apart from the list of values, it will be necessary to provide a list of the joint names to which each value corresponds. Other methods may provide the values in a different order. In the case of the Pose, it will be required to provide the format in which the rotation is written. The first 3 values will always refer to the x, y, and z values, but the rotation values may be given in RPY or Quaternion format. RPY is a rotation format that is easier for humans to visualize and calculate, but for computers, it is much easier to do operations using Quaternions. Both formats are available, but the default value is the RPY format.

Joint State and Pose from User Input

It is also taken into account the possibility of manually entering each value for the Joint State or Pose desired. For the Joint State, the function will ask the user to write the value for each of the joints of the robot, and for the Pose, it will ask for the x, y, and z values, and then for the RPY or Quaternion values, depending on the format desired. These functions will then return the objects for use by other functions.

Inverse Kinematics

One of the main issues in robotics is computing the inverse kinematics of a kinematic chain. The inverse kinematics (IK) has as an objective to obtain the Joint State values that place the TCP in a given Pose. It is usually an under-determined type of system, so the solution may be multiple, unique, or nonexistent. Consistently and rapidly solving these kinds of systems is a big field of research and study, but for the purposes of this project, the trac_ik package will be used.

Trac_IK is based on the widely used KDL IK algorithm but solves some shortcomings by considering joint limits, limiting computing time instead of the number of iterations, and running different methods in parallel to give the fastest result. Trac_IK needs to be specified the reference frames for the IK desired to compute, in this case the world frame and the tool0 frame. With this information, an IK solver object is created that also feeds on the kinematic model of the robot to configure itself. Once it is set up, by providing the x, y, and z values and the Quaternion values, it returns the joint values that place the TCP in that Pose. The joint values then need to be rearranged, as they are returned in alphabetical order, not the order used as a standard by the robot. A seed state can also be given to the solver to accelerate the convergence.

3.4 Storing and Loading Data

Now that the simulation is running and data about the robot can be obtained, the next step is to be able to store and load this data. This section will focus on creating an interface in order to read and write data from files that can later be easily moved. It will also feature a menu with the basic features used in many programs, like load, save, save as, etc.

Variables Needed

First, a set of common variables must be set in order to keep track of the state of the program and the information flow.

File Path: path to the folder that contains the files to read or write.

List of files: a list of the names of the files in the File Path.

Working File: name of the file on which the program is currently working.

Data Dictionary: data object that contains the information read, stored, or intended for use.

File Saved: Boolean variable that indicates if the data has been saved to the file or if there are still changes to be saved.

Dictionaries

Dictionaries are a Python data type that allows for storing multiple key-value pairs and efficiently retrieving the data when needed. Each data unit is uniquely identified by a key, which serves to access the corresponding data. Each dictionary can contain multiple data types at the same time, be it text, numbers, or even other dictionaries. They are an unordered data type, meaning that data is not guaranteed to maintain a specific order.

Dictionaries are a very flexible data type as they can store multiple data types. For this application, it will be useful to store diverse information in the same file. One can create a dictionary by using curly braces ({ }) and separating the key-value pairs with colons (:). They also offer multiple built-in functions to add, remove, overwrite, or check the existing keys.

Loading and Saving Complex Data

Dictionaries, as they can contain any data type, cannot be easily stored in a simple text file. It is necessary to store them in a binary file for it to remember the data structures present. Python provides the Pickle library, which allows reading and writing binary files in the Pickle format.

The functions to load and save data are written as the point of interaction between the program and the file system. They are conceived as simply as possible in order to avoid bugs and problems when reading or writing data.

Loading and Saving Data From a File

Once the interaction with the file system has been set, the next step is to configure how the user interacts with it. For that, the functions to load and save the files display a series of messages, await the input of the user, and act accordingly.

To load a file, it displays the list of available files to open, asks the user to input the name of the desired file, and opens it. If the file is not found, it will ask if the user wants to create it. Additional input parameters to the function can skip the UI to use the function faster.

To save a file, it will ask for the name of the file to save it as, and if it already exists, it will ask if the user wants to overwrite it. It will then be saved or aborted. As before, additional function parameters can skip the UI.

Removing Files

To remove files from within the program, it will ask the user for the name of the file to delete. If found in the file list, it will ask for confirmation from the user and delete the file requested, and it will reset the necessary variables. Additional parameters can skip the UI.

Storing Data

As dictionaries can store multiple data types, the data parsed into the function to store the data can be of any type. This function will add an entry to the dictionary that will contain the data provided, and the key will be asked for. If the key already exists, it will ask the user if they want to overwrite the data, and if they choose not to, it will ask if they want to choose another name for the key and start again. If a name for the key is already provided in the parameters, the UI asking for the name will be skipped.

Importing Data From a Text File

Currently, adding data to the dictionary can be slow if a lot of data needs to be stored. To add a small amount of data, it is manageable, but if a collection of points wants to be stored, it can become tedious moving between the different menus for each single point. For this reason, a utility function is set up in order to transform data described in a text file into entries in the dictionary. This will allow for easy input of data into the system in a more readable and manageable way.

The syntax expected is as follows: Each line of the text file will contain first the type of data to store, a colon, and then the value to store. The possible expected data types available for reading are the following:

JS: If it is read, the following information will refer to the ordered names of the joints of the robot. It will also indicate that if a ‘point’ data type is read, it will follow this format. The string is transformed into a list and then given to the function that transforms a list of values into the correct data type.

POSE: indicates the format of the rotation and that the next ‘point’ data inputs will be a pose of that format. The string is transformed into a list and then given to the function that transforms a list of values into the correct data type.

Name: indicates the key of the next value to be stored in the dictionary.

Point: the following set of values will represent a JointState or a Pose in RPY or Quaternion. Values are separated by commas.

Bool: indicates a Boolean type, and next is the value, True or False, to be stored.

Num: a number is to be stored.

Text: The following string is to be saved.

```
Name: ScriptFile
Text: cycle

Name: Repetitions
Num: 10

Name: GripperActive
Bool: False
```

```
Name: GripperInactive
Bool: True

JS: elbow_joint shoulder_lift_joint shoulder_pan_joint wrist_1_joint
    wrist_2_joint wrist_3_joint
Name: Home
Point: 0.0, 0.0, 0.0, 0.0, 0.0, 0.0

Name: DropPoint
Point: 0.5, 0.2, 2.0, 0.0, 0.0, 0.0

Pose: RPY
Name: PickPoint
Point: 5.0, 3.0, 2.0, 0.0, 0.0, 0.0
```

Listing 1: Example of a Text File to Import

For example, if the points are JS, first the joint names will be written, then the name of the point to store, and then the values. For each entry to be stored in the dictionary, first the name must be written and then the value. If no new name is used, it will use the previous name and overwrite the previous data. It will also always overwrite the data if the key already exists.

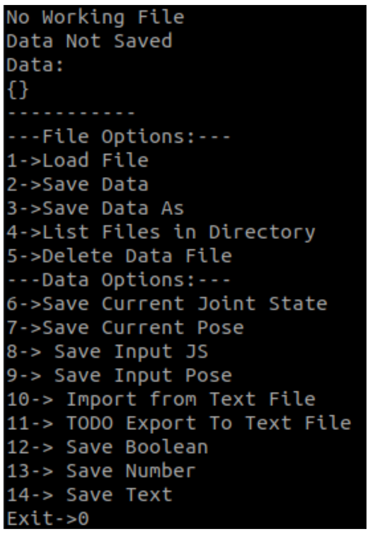
Deleting Data

This function will allow deleting dictionary entries. Will ask for the key of the value to delete, and then ask for confirmation. The key to delete and an option to skip confirmation can also be given as parameters to avoid the UI.

Using the Editing Menu

A function that displays a menu with different options that the user can select is set as a UI to allow interaction with the different functions. It offers the following options for files: Loading, Saving, Save as, List files in Directory, and Delete a Data file. For data handling, it offers the following functions: Save Current JS, Save Current Pose, Save Input JS, Save Input Pose, Save Input Boolean, Save Input Number, Save Input Text, Import From Text File, Delete Point, and List Keys.

These options execute the corresponding functions and, if needed, use the necessary robot information functions. When accessing the functions through this menu, the default options are presented, and no option to skip the UI is given. These options are only available when the functions are manually used in other parts of the code.



```
No Working File
Data Not Saved
Data:
{}
----------
---File Options:---
1->Load File
2->Save Data
3->Save Data As
4->List Files in Directory
5->Delete Data File
---Data Options:---
6->Save Current Joint State
7->Save Current Pose
8-> Save Input JS
9-> Save Input Pose
10-> Import from Text File
11-> TODO Export To Text File
12-> Save Boolean
13-> Save Number
14-> Save Text
Exit->0
```

Figure 10: Menu Editor with various options available

3.5 Moving the Robot

Now that the information about the robot can be stored, the next step is to actually move the robot into one of the desired positions. In order to move the robot, first all the different positions that it has to visit need to be calculated, as the information can be presented in multiple ways. Then the duration of the movement must be set. With this information, the object that the execution server can read needs to be set up, and then the information must be published.

Setting the Trajectory of Joint States

There are two main ways to store the state of a robotic arm: the state of the joints, or the position of the TCP. Also, multiple ways to store or input the information have been prepared. But in order to move the robot, the first step is to compile all the positions we want the robot to visit in a Joint State format.

First, a list of the types of points that will be input is set. This list contains the same number of elements as the number of points we want the robot to visit during the next move sequence. Each value of the list is either 'JS' for a JointState type, 'P' for a Pose type, 'F' for a point in a file, or 'L' for a point in the list given as a parameter. If no value is given, it will ask the user to input a value.

Then, the list will be iterated, and a Joint State object will be added to the list. If the input type is JS or P, it will ask the user to input the necessary values, and in the case of a P input, it will also calculate the Inverse Kinematics, all making use of the functions to obtain robot information. If the type is selected to be from a file, it will then ask the user for the file and the point, and if the point is of the Pose type, compute the IK to obtain the JointState. Finally, if it's a point on the list given as a parameter, it will read the point from the given data and do the IK if necessary.

With this, a list containing a sequence of Joint State configurations is obtained, which can then be given to the execution server.

Setting the Move duration

Another aspect that must be set for each point that the robot visits is the duration of the movement up to that point. During the iteration of the point-type list that yields the objective Joint State, a calculation for the move time is made for each movement and added to a list containing all the durations for that set of movements.

To calculate the move duration, a simplified approach has been taken. Supposing that a constant velocity is desired, the duration of a movement between two points has to be proportional to the distance between those points. If the distance is calculated in Cartesian space, then the trajectory would have to be be linear for the calculation to be correct, which is far from the truth. Instead, it is taken the approach to calculate the distance in the Joint Space. By using as a base the distance that the joints have to rotate, an accurate distance is obtained, and is faster when computing as no IK calculations need to be done. The distance is then multiplied by a factor, which the user can change by using an available function.

It is also given the option for the user to manually input the time in seconds that the move duration is desired to take.

Set Trajectory Points and Goal

Once all the Joint State points are set, as well as the duration of each movement, the next step is creating the object necessary for the server to execute the movements.

A Joint Trajectory Point object will be set for each desired point in the trajectory, given its Joint State and Duration. All the Trajectory Point objects will be added to a list belonging to a Joint Trajectory Goal Object. This object contains all the Joint Trajectory Point objects as well as other information related to the robot. This object now contains all the information necessary to be sent to the Execution server.

Publish the trajectory to Move the Robot

In order for the robot to move, the Joint Trajectory Goal object needs to be published for the execution server to execute it.

First, an action client to connect to the server is set up with an adequate controller for the robot. Once a connection to the server is established, the rest of the necessary parameters are set, and the goal is sent to the server. The server then executes the movement instructed and informs the program when it has finished correctly.

Utility functions

This module also offers some utility functions to facilitate its use:

The Go Home function moves the robot to a desired stored state. By default, this is the zero configuration, where all the joints are in the 0 position. It is useful, for example, to set the initial position of the robot or have a safe position in which to wait.

The RQT library offers a Joint Trajectory Controller interface that allows moving each joint independently. It can be started from within the program and allows for easily moving the robot to specific joint configurations.

The Move Menu offers a variety of options to move the robot to different Joint States or Poses. For example, it allows moving the Robot to an input JS, an Input Pose, a point in a File, an input JS trajectory, an input Pose Trajectory, an input File Trajectory, a trajectory combination of the previous types, Start/Stop the RQT Joint Trajectory Controller, changing the Duration calculation to Automatic or Manual, and changing the Duration conversion scale.

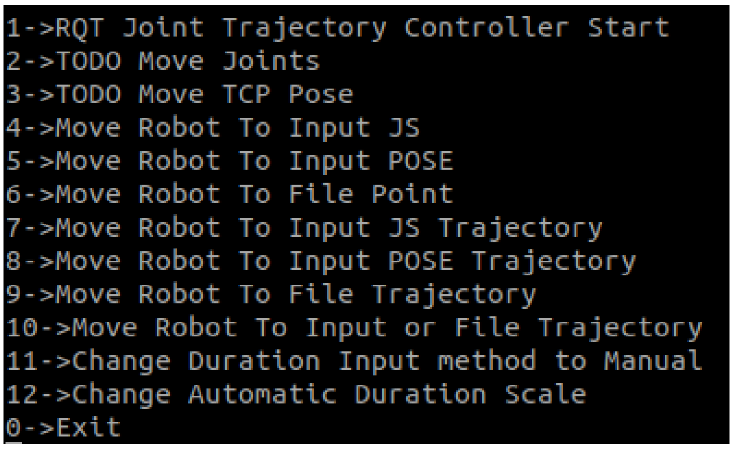

```
1->RQT Joint Trajectory Controller Start
2->TODO Move Joints
3->TODO Move TCP Pose
4->Move Robot To Input JS
5->Move Robot To Input POSE
6->Move Robot To File Point
7->Move Robot To Input JS Trajectory
8->Move Robot To Input POSE Trajectory
9->Move Robot To File Trajectory
10->Move Robot To Input or File Trajectory
11->Change Duration Input method to Manual
12->Change Automatic Duration Scale
0->Exit
```

Figure 11: Move Menu with different optind to indicate where to move the robot

3.6 Scripting Movements

With the ability to move the robot and store information, the next and last functionality to implement is the execution of scripts. Scripts are a sequence of instructions executed, not by the computer directly but by another program. In this case, we want commands to control the flow of execution, read and write variables, and move the robot, all while keeping the syntax easy to read and understand. This will allow the robot to operate autonomously through cycles of movements without having an operator imputing each command.

Data Needed

For the script to be executed, the instructions are not enough. It is also necessary for the program to know which variables are being referenced when executing some commands. For this reason, the information is divided into three different files:

Data About the Script: This contains the basic information about the script. Specifically, it contains the names of the rest of the files needed and the basic variables and initial values used during the execution of the script. The format is the same as for importing data from a text file, or it can be stored in a binary file. These variables will be loaded into a dictionary for their use.

Points used in the script: This file will contain the different points that the robot can be ordered to move to. It is decided to be a different file to allow more flexible data handling. Data can be changed mid-script, and the execution does not stop. For example, if a position is obtained externally, a placeholder point should be set up and modified when the data is read.

The Script: a text file containing the sequence of instructions. Each line is limited to a single instruction, and the commands available are limited, but enough to handle the common cases.

Loading or Reading the Data

First, the script information has to be the first to be loaded into the program, as it contains the rest of the files that have to be loaded. Two options are given: loading directly from a binary file or importing from a text file. Which one is used will depend on how the data has been stored.

Next, the corresponding file containing the points will be loaded or imported, depending on how the data is stored.

Finally, the text file containing the instructions will be read and stored.

Executing the Script

When executing the script, each line is read sequentially, one after the other. Each line contains one command word and the necessary parameters, if needed. There are two types of commands: those that perform a set action or function, and those that control the flow of the execution.

Function Commands:

-Load: it deletes the current data dictionary and then loads the information from a binary file.

-Import: Same as 'Load', but for text files that must be imported.

-Set: It changes the value of the named variable to the value provided.

-MoveTo: Moves the robot to the specified stored point.

-Wait: It pauses the execution until a condition is met. Can wait for a variable to become 'True', wait for any user input, or wait the given number of seconds.

Flow Commands:

-While: This function checks if the variable specified is ‘True’ or not. If it is, it stores the line in which this ’While’ is and proceeds with the execution normally. If the variable is set to ‘False’, it will not execute the following lines unless it’s an ‘EndWhile’.

-EndWhile: When encountering this command, if the last ‘While’ was being executed, it makes the execution jump to the location of that ’While’ and continue from there. If the last ‘While’ had a ‘False’ condition and was not executing commands, it allows from this point forward the execution of commands and deletes the last stored line number to return to, as it is no longer needed because the while loop is now finished.

-If: When the variable given is ‘True’, it allows the execution of the next commands and blocks it if the condition is ‘False’.

-Else: If execution was permitted from the last ‘If’, it now blocks it, and if it was blocked, it now allows it.

-EndIf: It eliminates the last variable that indicated if commands had to be executed or not. From now on, execution is not conditioned and can proceed normally.

Dummy PLC Simulator

In order to simulate Input/Output signals, a Python program is written to cyclically read the data file and modify it accordingly when some variables are found to meet some requirements. The script can then reload the file with the updated variables and act accordingly.

The Python program is refreshed every second and is hard-coded, meaning it is programmed to do a specific function and simulate a specific environment. It is not a formal part of the interface programmed in this project; it is just a tool used for testing.

4 Testing

Once the program has been set up, to examine the extent of its capabilities, a series of tests are conducted and later evaluated. This will determine if the objectives set have been successfully achieved, what parts can be improved, and how future work can do it. In this section, the steps to set up the different experiments will be explained.

4.1 Configuration

The first test will be, starting with an empty environment, how a user would use the program to create the files necessary for executing a script.

-When the program is started, the first thing to do is select the robot that we will be working with. By entering ’3’, the program will show us a list of all the available robots at the moment. We then enter the number corresponding to the robot desired. If no robot is selected, it will load the UR3 by default.

-We are returned to the previous menu, and from there the simulation is started by entering ‘1’ into the console. The Gazebo simulation will start, and the robot will move to the Home position. Once everything has loaded correctly, the next menu will appear.

-We enter the File Editor menu, and from here various options appear. To create the main file for the script, we can either import the information from a text file, or add the data one by one and save it in a binary document.

-For the first case, importing from a text file, we have to write according to the correct syntax. That is, first writing ‘Name:’ and stating the name we want to give to the next variable to import. Then we must state the type of value, followed by the actual value. Repeat this for every variable desired to be used in the script.

-For the second case, manually entering the values, we must select the option with the data type we want to store, then enter the value, and finally the name. Once the dictionary is set, we have to save it.

-The next file to set up is the one containing the possible points the robot can be ordered to move to. As before, we can import the data from a data file written in the correct syntax, or we can save the file manually. To manually store the information, various options are given: we can use the Editor interface to input the data manually if we know the values of the joints or pose; we can move the robot with external control and then store the current position; or we can move the robot using the built-in menu to position the robot and read the current position. With all the points in place, we save the file with the name specified in the script configuration.

-Finally, we create a text file with the name set in the configuration and write the operations and logic we want the script to have.

-Editing the text files must be done by the text editor on the computer; the program does not offer a text editing feature. Different Scripts will be shown in the next tests.

4.2 Pick and Place

Pick and place is one of the most basic robot arm operations. It consists of placing the robot in a position to grab an object, retreat in a controlled way, go to another position, approach carefully, and release.

This script will use 5 boolean variables and 7 robot positions. The Boolean Activate-Gripper, GripperActive, DeactivateGripper, and GripperInactive variables will control the state of the gripper, and Repeat ensures cyclic execution. The points are the pick and the place location, the pre-pick and pre-place location, 2 pass-through points, and the Home point.

```

Name: PointFile
Text: PAPpoints
Name: ScriptFile
Text: PAPscript

Name: Repeat
Bool: True
Name: ActivateGripper
Bool: False
Name: DeactivateGripper
Bool: False
Name: GripperActive
Bool: False
Name: GripperInactive
Bool: True
```

Listing 2: Pick and Place Configuration File

```

load cycledata
moveto p0
while Repeat
wait time 1
moveto PrePick
wait time 1
moveto Pick
wait time 1
set ActivateGripper True
wait GripperActive
wait time 1
moveto PrePick
wait time 1
moveto Pass1
wait time 1
moveto Pass2
wait time 1
moveto PrePlace
wait time 1
moveto Place
```

```
wait time 1
set DeactivateGripper True
wait GripperInactive
wait time 1
moveto PrePlace
endwhile
```

Listing 3: Pick and Place Script

The script will cyclically repeat the sequence of movements until the Repeat value is set to 'False'. It will also wait for the variables that express the state of the gripper to be set to the values requested.

This test shows the basic capabilities needed to perform pick-and-place operations.

4.3 Sorting

The next test will be an expansion of the Pick and Place algorithm. The pickup location will always be the same, but the place location can change depending on a Boolean variable. This variable represents, for example, a sensor that classifies the picked object as one type or another. This way, the robot will place the object in one location or another depending on the sensor, classifying it.

In this test, the ability to deal with conditional clauses and the capacity to connect information signals into the script are shown.

```

Name: PointFile
Text: SORpoints
Name: ScriptFile
Text: SORscript

Name: Repeat
Bool: True
Name: Type1
Bool: False
Name: Type2
Bool: False
Name: ActivateGripper
Bool: False
Name: DeactivateGripper
Bool: False
Name: GripperActive
Bool: False
Name: GripperInactive
Bool: True
```

Listing 4: Sorting Configuration File

```

load cycledata
moveto p0
while Repeat
```

```
wait time 1
moveto PrePick
wait time 1
moveto Pick
wait time 1
set ActivateGripper True
wait GripperActive
wait time 1
moveto PrePick
wait time 1
moveto Pass
wait time 1

if Type1
set Type1 False
moveto Pass1
wait time 1
moveto PrePlace1
wait time 1
moveto Place1
wait time 1
set DeactivateGripper True
wait GripperInactive
wait time 1
moveto PrePlace1

else
if Type2
set Type2 False
moveto Pass2
wait time 1
moveto PrePlace2
wait time 1
moveto Place2
wait time 1
set DeactivateGripper True
wait GripperInactive
wait time 1
moveto PrePlace2


else
set Type1 False
set Type2 False
moveto Pass3
wait time 1
moveto PrePlace3
wait time 1
moveto Place3
wait time 1
set DeactivateGripper True
wait GripperInactive
wait time 1
moveto PrePlace3

```

```
endif
endif
endwhile
```

Listing 5: Pick and Place Script

4.4 Bin Picking

It is a core problem in the fields of computer vision and robotics. The goal is for the robot to pick up pieces that are in an unordered state by finding the correct position for the pickup with sensors and cameras. The computer vision software will analyze an image taken of the unordered group of pieces and return the position of a piece to pick. That position will be sent to the robot, and the necessary calculations will be done to move the robot to that position, pick up the piece, and move it to the required location.

This problem combines multiple disciplines like robotics, computer vision, communications, and logic control. To adapt the problem and be able to test only the robotic part, the communications are simulated with a Python script that will modify the stored data according to the necessary logic.

The script will offer the option of two movements: loading the machine or unloading the machine. Variables to indicate the desired movement are set in place, and when the external logic sets one of them to 'True', the robot will then execute that movement.

Each movement has a predetermined number of positions the robot will sequentially visit. Except for the position to pick up the piece, where the script will wait until the external program informs it that it has set the pickup point, a placeholder pickup point must be set in place when stating the available points.

This script shows two pick-and-place sequences, where the external signal selects which to execute. Moreover, one of the points the robot has to move to is changed every cycle by the external script. This shows all the main capabilities the program has to offer.

```

Name: PointFile
Text: BINpoints
Name: ScriptFile
Text: BINcycle

Name: Repeat
Bool: True
Name: LoadMachine
Bool: False
Name: UnloadMachine
Bool: False
Name: AskPickupPoint
Bool: False
Name: PickupPointSet
Bool: False
Name: ActivateGripper
Bool: False
```

```
Name: DeactivateGripper
Bool: False
Name: GripperActive
Bool: False
Name: GripperInactive
Bool: True
```

Listing 6: Bin Picking Configuration File

```
load cycledata
moveto p0
wait time 1
while Repeat
load cycledata
wait time 1
if LoadMachine
set LoadMachine False
set AskPickupPoint True
wait var PickupPointSet
set PickupPointSet False
moveto pinput
wait time 1
set ActivateGripper True
wait GripperActive
moveto p0
wait time 3
moveto p1
wait time 3
moveto p2
wait time 3
set DeactivateGripper True
wait GripperInactive
moveto p2
wait time 3
moveto p1
wait time 3
moveto p0
wait time 3
else
if UnloadMachine
set UnloadMachine False
moveto p0
wait time 3
moveto p2
wait time 3
moveto p1
wait time 3
set ActivateGripper True
wait GripperActive
moveto p3
wait time 3
moveto p4
wait time 3
set DeactivateGripper True
wait GripperInactive
moveto p4
```

```
wait time 3
moveto p3
wait time 3
moveto p0
wait time 3
endif
endif
endwhile
```

Listing 7: Bin Picking Script

5 Results

With these tests executed, a full understanding of the program's capabilities and limitations can be obtained. From the general setup to the simple and complex scripts, the program has managed to execute properly and yield the expected results. Still, shortcomings have also been found.

During the configuration stage, one is able to easily store all the main information desired. Storing simple data is very straightforward through the presented interface. In the case of storing points for the robot, it gets a bit more complicated. Changing between menus multiple times makes the task tedious, and importing requires knowing the position beforehand. The interface for editing existing values is also lacking, as it is faster to simply overwrite the data.

In the scripting part, the robot is shown to be able to work cyclically, do pick-and-place operations, and act according to variables that are modified by an external input. It meets the basic requirements for the most common operations done with robots, but for more complex systems, the options provided may not be enough. Firstly, the syntax and requirements can be a bit confusing if one is not familiar with them. The script being in text shape does not help with readability, making it more complex to write longer programs. Finally, the number of commands available can be found to be limited, as no mathematical or Boolean operations are allowed, and there are only two functions that control the flow of the program. The functions provided are enough for simple programs, but complex ones may require some intricate use of these functions.

6 Impact

Industrial Application

This program is not yet fit for industrial application, but with more work and polishing, it could make its way into the industry. It would significantly reduce the training time for the robot operators, as learning one interface could be enough to program different manufacturers' robots.

Open Source

This project has been conceived in an Open Source context, as have most of the tools used during its development. With this, the community of robotic enthusiasts can edit and improve the program and expand the viewpoints, skills, manpower, and impact that this program can have.

Access to all the code can be done through my Github Repository [21].

Personal Learning

On a personal level, this project has allowed me to improve my programming skills, deepen my knowledge about ROS, and consolidate everything learned during this master's degree. It has also helped with soft skills like organization, synthesizing, research, analysis, commitment, and focus.

Environmental

This project's software does not have any direct impact on an environmental level, but its development and use require electricity, which needs to be generated, which in turn generates CO2 emissions. This project has been completed in approximately 600 hours, and the work has been done exclusively on a computer as simulations were run instead of real robots. The total energy consumption of an 80W computer is 48 kWh. With an average of 195 gCO2/kWh [4], this results in the emission of 9.36 kg of CO2 into the atmosphere.

Economic

The economic impact this project has had is calculated by adding all the costs of the different elements that have participated in the project. First, the cost of the dedication of an engineer has been estimated at 15€/h, which for the 600 hours of labor results in a cost of 9000€. The next term is associated with the materials used, in this case, the computer. Two different computers have been used: a desktop and a laptop computer. The first one had an initial cost of 1500€, and the laptop 1000€. With an amortization rate of 20% per year, the 6 months of this project yield 10% of amortization. As each computer was used for half the time, the amortization is actually half of that value. This yields 75 and 50 €in amortization costs, respectively. Finally, the consumption of electricity. The power consumption of both computers is equivalent, so the previous calculated electricity consumption is used. 48 kWh at 0.2966€/kWh [3] results in 14.23€spent on electricity.

Concept	Base	Amortization	Hourly cost	Hours	Subtotal
Engineer			15€/h	600 h	9000€
Desktop Computer	1500€	5%			75€
Laptop Computer	1000€	5%			50€
Electricity	48 kWh		0.2966€/kWh	600 h	14.23€
Total					9139.23€

Table 2: Summary of costs

7 Conclusions

Objectives Achieved

The objectives initially described started with the intention of creating an interface in order to control any type of robotic arm. The bulk of the project has been done in simulation with a UR3 robot, but limited testing with other robots has also been done. The UR series is provided with all the necessary files for launching the simulations, but Fanuc and ABB lack some simulation files. By manually creating these files, a workaround is found, but some parameters not provided make the simulation inaccurate. The pool of selectable robots is currently limited to UR and a couple Fanuc robots, but this amount can be easily expanded by completing the files for the simulation with the model described. This objective has been achieved, even if this feature is in a limited state, but it can be improved with ease.

The second main objective was creating a system to store and load data. It has been completed by allowing the storage of data in a dictionary, which can hold multiple data types. Also, an import file option makes it easier than using the rudimentary interface. Importing is preferred in cases with a lot of data, while the interface would be preferable for adding or modifying data units.

The scripting language created is also capable of solving the initial problems presented. The number of functions is limited, but sufficient to successfully solve the typical cases of robotic arm operations. An important feature is being able to communicate data through a very simple and rough method, but reading and writing nonetheless.

Further Work

The objectives accomplished represent a milestone, but improvements, optimizations, and expansions could be made. Firstly, expanding the robot models available and testing with real robots. It would increase the usability of the program and make it more accessible to whoever would like to use it.

Also, an improvement in the user interface would help. Currently, it is only a simple interface, as the focus of the project was to create the program without necessarily making it look good. The way the program is structured is to facilitate the later integration of a UI done with, for example, PyQt.

Finally, improving the scripting language by expanding the functions available and adding more options for the existing ones would facilitate writing scripts. Currently, some knowledge of programming is required to write complex scripts, but with more options, it could be reduced.

Doing so can be done by researchers or collaborators of the ROS project, as the code is open source and anyone can contribute to improving it.

Overall Conclusions

This project has served to do an analysis of the current state of robotic arms and the tools available to program and control them. ROS is a very powerful tool, and its flexibility allows creating environments capable of working with different robots, even from different manufacturers. This can serve as a starting point for great further developments, by allowing community and researches to have a solid starting point to continue developing this idea, if they see future in it.

Bibliography

[1] Brief Robotic Arm History "https://tedium.co/2018/04/19/robotic-arm-history-unimate-versatran/"

[2] Brief History of Industrial Robots, "https://www.wevolver.com/article/a-history-of-industrial-robots"

[3] Average Electricity prices in Spain "https://electricityinspain.com/electricity-prices-in-spain/"

[4] Emisions of CO2 per kWh, "https://www.nowtricity.com/country/spain/"

[5] Trac-IK webpage, "https://traclabs.com/projects/trac-ik/"

[6] ROS Documentation, "http://wiki.ros.org/"

[7] Python Documentation, "https://docs.python.org/3/"

[8] Git description, "https://git-scm.com/"

[9] ROS-2 Documentation , "https://docs.ros.org/en/rolling/index.html"

[10] Ubuntu Releases, "https://wiki.ubuntu.com/Releases"

[11] Main ROS webpage, "https://www.ros.org/"

[12] CIM-upc webpage, "https://www.cimupc.org/en/"

[13] ABB interface example, "https://control.com/technical-articles/introduction-to-abb-robot-programming-language/"

[14] ABB RoboStudio, "https://new.abb.com/products/robotics/robotstudio"

[15] Fanuc Programming example, "https://control.com/technical-articles/introduction-to-fanuc-robot-programming/"

[16] Unimate Picture, "https://www.techspot.com/images2/trivia/bigimage/2021/2021-12-17-image.jpg"

[17] Fanuc Robots Picture, "https://www.techspot.com/images2/trivia/bigimage/2021/2021-12-17-image.jpg"

[18] UR interface Picture, "https://www.universal-robots.com/media/1810404/1.png?width=800&height=500"

[19] Fanuc Interface Picture, "https://motioncontrolsrobotics.com/wp-content/uploads/2018/01/alarm-history.pngg"

ETSEIB

[20] ABB Interface Picture, "https://roboticsbook.com/wp-content/uploads/2018/11/Fig_67.png"

[21] Github page where to fing the code, "https://github.com/Lynkx95/TFM"