Final Master Thesis

MASTER'S DEGREE IN AUTOMATIC CONTROL AND ROBOTICS

Implementation and evaluation of movement primitives and a graphical user interface for a collaborative robot

MEMORY

Author : Roy Ove Eriksen

Supervisor : Prof. Dr. Cecilio Angulo

Date : January, 2021

Escola Tècnica Superior

d'Enginyeria Industrial de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Abstract

This master thesis's main objective is to develop an application that makes it easier for a user without prior robotic experience to control a robot. By focusing on intuitive, simplicity and safety for the end-user, the hope is to shorten small companies' path to invest and use robots. The environment consists of a standard computer and a Universal Robot 3 CB-series. The core of the application consists of acquiring a trajectory by Learning from Demonstration and converting these demonstrations to Dynamic Movement Primitives (DMPs). These DMPs are stored in a library and can generate robot trajectories at the users' discretion. The project consists of studying methods and investigating support libraries to develop a ROS application that follows the requirements of being: intuitive, safe and straightforward. A graphic user interface has been developed using the PyQT framework to facilitate the intuitive and straightforward requirements. While making the application safe, the generated trajectory can not be performed before a collision check is calculated. The developed application performs connection to the robot, recording and storing demonstrations, computing, and storing Dynamical Movement Primitives in a library while also generating and executing safe trajectories on the robot.

Contents

Page

Abstract i
List of Figures v
List of Tables vii
Acronyms ix

1 Introduction **1**
1.1 Motivation 2
1.2 Objectives 2
1.3 Scope 3

2 Background **5**
2.1 Learning From Demonstration 5
2.1.1 Teaching a Skill 6
2.1.2 Reproducing a Skill 7
2.2 Dynamic Movement Primitives 9
2.2.1 Point-attractive System 10
2.2.2 The Forcing Term 11

3 Resources **15**
3.1 ROS 15
3.1.1 ROS DMP Library 16
3.1.2 Universal Robots ROS Driver 18
3.1.3 MoveIt 18
3.2 Programming Languages 20
3.3 Universal Robots 21
3.3.1 UR3 22

4 Implementation **23**
4.1 Back-end Implementation 23
4.1.1 Preliminaries 24
4.1.2 Recording 26

4.1.3 DMP Learning . . . 29
4.1.4 Execution Phase . . . 31
4.2 Graphical User Interface . . . 33
4.2.1 Login Screen . . . 33
4.2.2 Recording Tab . . . 34
4.2.3 Learning Tab . . . 35
4.2.4 Execute Tab . . . 36

5 Results **41**
5.1 Evaluation of LfD Methods . . . 41
5.1.1 Kinesthetic Method . . . 42
5.1.2 Teleoperation Method . . . 45
5.1.3 Discussion . . . 46
5.2 Evaluation of DMP Execution . . . 48
5.2.1 Changes in DMP Parameters . . . 48
5.2.2 Time Consumption . . . 50
5.2.3 Discussion . . . 52
5.3 Evaluation of the GUI . . . 52
5.3.1 Discussion . . . 54

6 Impact **57**
6.1 Social Impact . . . 57
6.2 Environmental Impact . . . 58
6.3 Budget . . . 59

7 Conclusions **61**

8 Future Work **63**

9 Annex **65**
9.1 Videos . . . 65
9.2 UML Diagram of Code . . . 66
9.3 Code . . . 67

Acknowledgement **69**

Bibliography **71**

List of Figures

2.1 Illustration of the different levels of representation for describing the skill. Source: [1] . . . 8
2.2 Symbolic learning with constraints. Source: [2] . . . 9

3.1 Illustration of the communication between nodes. Source: [3] . . . 16
3.2 MoveIt pipeline. Source: [4] . . . 19
3.3 Image of the Universal Robots product series. Source: [5] . . . 21

4.1 Application pipeline. . . . 24
4.2 Robot description as seen in the Gazebo simulator. The new description includes a gripper and a table. . . . 25
4.3 Kinesthetic demonstration on the UR3 by using the freedrive option on the teach-pendant. . . . 27
4.4 Teleoperation using a program created on the teach-pendant Source: [5] . . . 28
4.5 Pipeline of the recording class in Python. . . . 29
4.6 Pipeline of the learning class in Python. . . . 31
4.7 Pipeline of the Execution class in Python. . . . 33
4.8 Initial login screen when nothing is connected . . . 34
4.9 Loginscreen when launchfiles has been launched and connection to robot established. . . . 35
4.10 Display of the “Recording” tab. . . . 36
4.11 Display of the “Learning” tab where users can edit parameters. . . . 37
4.12 Display of the “Execute” tab when retrieving joint space coordinate of the robot. No collision on trajectory is generated. . . . 38
4.13 Display of the “Execute” tab when retrieving Cartesian coordinates of the robot. . . . 39
4.14 Display of the “Execute” tab when retrieving joint space coordinate of the robot. Collision on trajectory is generated. . . . 39

5.1 End-effector path using the kinesthetic method 1 in the pick and place activity. . . . 42

5.2 Joint positions for the kinesthetic method 1 in the pick and place activity. 43
5.3 Joint velocities for the kinesthetic method 1 in the pick and place activity. 43
5.4 End-effector path for the kinesthetic method 2 in the pick and place activity. 44
5.5 Joint positions for the kinesthetic method 2 in the pick and place activity. 44
5.6 Joint velocities for the kinesthetic method 2 in the pick and place activity. 45
5.7 End-effector path for the teleoperation method in the pick and place activity. 46
5.8 Joint positions for the teleoperation method in the pick and place activity. 46
5.9 Joint velocities for the teleoperation method in the pick and place activity. 47
5.10 Resulting trajectory shape from changes in *k_gain*. 49
5.11 Resulting trajectory shape from changes in *num_bases*. 50
5.12 Resulting trajectory shape from changes in *dt*. 51
5.13 Video showing workflow from starting up application to executing trajectory on robot. Link: *https://youtu.be/PuuCp_ny7Ro*. 53
5.14 Video showing how to change between different motions stored in the motion library. Link: *https://youtu.be/ABXdjRbYoDA*. 53
5.15 Video showing how to change between different parameters when computing DMP weights. The newly computed DMP weights are converted to trajectories and played on the robot. Link: *https://youtu.be/Iqh39EYPuPE*. 54
5.16 Video showing recording and generating DMP from a demonstration. Furthermore, the video shows execution of different trajectories. Link: *https://youtu.be/LuKmQYCK*

9.1 Video of kinesthetic demonstration with real robot 65
9.2 Video of comparison between kinesthetic and teleoperated demonstration with real robot. 66
9.3 Video of teleoperated demonstrated with real robot. 66
9.4 UML diagram of the three core classes and the GUI class. 67

List of Tables

5.1 Comparison of value of *k_gain* and corresponding number of trajectory points. 48
5.2 Time comparison, in seconds, when using diffeerent *dt* values, for the number of trajectory points, collision check and computing end-effector path. . 51

6.1 Energy consumption of computer and robot during the project. 59
6.2 Estimated cost of the project. Materials cost are calculated using life expectancy divided on usage, while energy consumption uses energy used multiplied by energy price(0,25€/kW/h) 59

Acronyms

Cobot Collaborative Robot

DMP Dynamic Movement Primitive

FK Forward Kinematics

GUI Graphical User Interface

IK Inverse Kinematics

IL Imitation Learning

LfD Learning from Demonstration

REE Red Eléctrica de España

ROS Robot Operating System

RTDE Real-Time Data Exchange

RVIZ ROS Visualization

SME Small and Medium-sized Enterprises

TFM Treball Fi de Màster (Master Thesis)

UR Universal Robots

UPC Universitat Politècnica de Catalunya

Chapter 1

Introduction

Industrial robots have being used in manufacturing since the 1930s. Initially, an industrial robot had a crane-like shape and was powered by a single motor that controlled all the axes. It was not before 1981 that Prof. Takei Kanade created the first robotic arm with motors directly installed in each joint making this new modern robot much faster than the former versions. One of the main drawbacks for industrial robots was the size of the floor area they required. The protection of human workers is essential, so the robot had to be locked in a cage. Another drawback was the acquisition cost of the industrial robot. The price of the robot and the necessity to hire an external robot consultant to program the robot for a specific task made robots mostly unavailable for the small-scale manufacturer.

To solve this problem, Universal Robots (UR) introduced in 2008 the first collaborative robot (cobot). The UR cobot is a low-cost robot that could be placed directly on the production floor and did not need to be locked in a cage. The cobot came with a touch screen tool that made it easier to program the robot to do specific tasks, and the need to hire an external robot consultant almost vanished.

The method of using the touch screen to program is useful when the goal and the starting position are defined in advance, and the trajectory shape between points is not essential. However, when the shape of the trajectory must be precisely defined, the user must manually add the needed way-points to achieve the desired trajectory. Moreover, if goal or starting positions change, the user must program a new routine. In the cases where both the start or goal positions changes frequently, and the trajectory needs the desired shape, the programming of the robot will be a time-consuming affair.

In these complicated types of scenarios is where the power of imitation learning (IL) excels, combined with the use of dynamic movement primitives (DMP). In this situation, the user first demonstrates the desired movement to the robot. Then the designed algorithm can reproduce the same movement in the robot. The shape of the trajectory will be similar to the demonstrated movement, but the trajectory can start and end at different positions. Furthermore, since the DMP is a point attractor system, the trajectory is robust at converging to the desired goal position.

1.1 Motivation

The primary motivation for this project is to increase the accessibility that non-robotics experts have to use robots for their desired tasks. To make a system that increases non-expert usage includes making a framework that is robust and safe, both concerning the robot and the human operator. Nevertheless, the framework also needs to be intuitive and straightforward for the user to use. In this form a large number of small and medium companies can benefit of using cobots in this digital revolution associated to Industry 4.0.

There is also some personal motivation for this project. Moreover, that is to increase my knowledge working with robots and then more specifically, the ROS framework by using the knowledge I have acquired during my 2-year studies in the Master of Automatic Control and Robotics. Furthermore, to prove to myself that I can execute a larger individual research project.

1.2 Objectives

The main objective of this project is to develop a robust framework that a user without a prior robotic experience can use to make the robot perform desired tasks by using imitation learning combined with DMP. The robot should be safe to use, both concerning the robot itself, but also for environment around the robot. Furthermore, the usage of the system should be robust user errors, but also intuitive for the user to use. To achieve the main objective, a set of milestones have been designed and are listed below,

1. Research and evaluate available DMP libraries
2. Development of a framework for learning from demonstration

3. Research and development of safety features, specifically for collisions
4. Development of a Graphical User Interface (GUI)
5. Development of a intuitive and robust framework.

The final goal is to develop a software that is available open source that implements the previous mentioned objectives, so that other users can use this software to shorten the road they have into the robotic world.

1.3 Scope

This Master Thesis will develop a Dynamic Movement Primitive control system to be used on a Universal Robot UR3 CB-edition. External sensors such as a camera are not implemented, so the only sensors for feedback are the sensors integrated with the robot. Without the external sensors, there are limitations of the world perception and on the registration of human proximity. Unlike the newer e-Series of Universal Robots, the CB-Series does not have a Force/Torque sensor integrated. With this limitation, compliance control is not considered in this work.

The thesis will focus on developing an intuitive system for controlling the UR3 robot, while also providing safety for the robot so that it can not collide with itself or the table it stands on.

Chapter 2

Background

The main goal of this thesis work is to implement a system that is simple and intuitive to use for non-experts. For this to be a reality, the user must have a simple way of showing the robot what kind of motion it should perform. For the user to demonstrate a motion, the concept of learning from demonstration(LfD) will be implemented. Below, the general concept of LfD is introduced along with a more detailed explanation of the chosen algorithm for Dynamic Movement Primitives (DMP).

2.1 Learning From Demonstration

In the traditional approach to making a robot perform a task, the task is directly programmed by an expert. When programming a task, the desired positions of the robot has to be defined by the user [6]. The programming work can be very time-consuming, especially whether the trajectory should meet some desired specifications, beyond the default movement solution. The programmer then should manually add a number of way-points along the trajectory. When a task is directly programmed, it is not easy to scale or reuse the code for other similar tasks without making modifications to the program. Directly programming a task is also prone to errors and require that the user has full knowledge of the task to be performed.

Solving the complexity of reusing a programmed task is where learning from demonstration (LfD) excels. In LfD, the goal is to transfer human skills to the robot. One of the major contributions of LfD is that learning a specific task is not dependent on a

robotic expert to program the task. LfD is a part of the supervised learning research area in machine learning [7]. It can be divided into two phases, the teaching part and the reproducing part. The teaching phase concerns to the transfer of human skills to the robot. The reproducing phase refers to learning a presentation of the movement that is demonstrated. Below both, the teaching and reproducing phases are described in detail.

2.1.1 Teaching a Skill

Teaching a robot a new skill deals with the problem of transferring the knowledge of a teacher to the robot. When transferring a skill from human to robot, a problem occurs since the anatomy of the human body differs from the robotic body. This is known as the correspondence problem [8] and deals with the different morphology between teacher and robot. To teach a skill, that skill should be recorded so that it can be used in the learning process. In order to record skills for LfD, three different methods are considered here[54]. These methods are kinesthetic teaching, teleoperation and observational learning.

Kinesthetic teaching can be seen as a high-level method of doing teleoperation. In kinesthetic teaching, the joints of the robot are made passive and are then moved by the teacher to perform the desired motion. To be able to perform kinesthetic teaching, the robot must be small or be able to compensate for the robot's own weight with a gravity-compensation controller so that the teacher is able to move the robot[53]. Another drawback with kinesthetic teaching is that teaching complex motions can be complicated because the teacher must move multiple joints simultaneously. The main benefit for kinesthetic teaching is that the motion can be fully or partially demonstrated by a teacher, while also tuning the speed of the motion during the execution of the trajectory. Applications of kinesthetic teaching have been wood planning [9], playing the ball-in-a-cup game [10] and flipping pancakes [11].

In teleoperation, the robot is controlled from a distance by a user using a remote controller. This controller can be a joystick, haptic device (e.g. Phantom) or a touchpad. Teleoperation of a robot is a simple way to control a robot without needing the user to be close to the robot. The drawback of teleoperation is that controlling the robot is sensitive to delay. This sensitivity makes teleoperation of a robot that is far away (e.g. another planet) not feasible without compensating for the delay by reducing the robot speed to ensure safety. Another drawback is that if the user can not see the robot, the user is dependent on external sensors to ensure that the robot behaves as expected. Even with these limitations, teleoperation has been used for a wide variety of applications concerning

LfD. Some examples are flying a robotic helicopter [12], robotic assembly tasks [13], and object grasping [14].

Finally, another way to demonstrate a motion that is in contrast to the two previously mentioned methods that require direct control of the robot is observational learning. In observational learning, the teachers own body is making the motion, while the robot is shadowing that motion[5]. To obtain the motion demonstrated by the teacher, an external sensor is required. This can be motion sensors that are put on the teacher's body, or an image capturing device (e.g. Microsoft Kinect) that tracks the movement the teacher performs. The major drawback with observational learning is that it heavily relies on the correspondence problem. Unlike teleoperation and kinesthetic teaching where the robot's own sensor records the motion, observational learning needs to find a mapping from the teacher's body to the robot body. The environmental setup also requires more work than the two previously mentioned methods since the addition of external sensors must be set up and calibrated. Despite the limitations imposed by observational learning, the method has been used successfully in LfD algorithms such as object manipulation [15].

The three methods that have been described serves as input for the learning phase. The robot will then have to successfully learn how to reproduce a representation of the demonstrated motion. This reproducing part will be described in the next section.

2.1.2 Reproducing a Skill

Now that a skill has been demonstrated, the robot should learn to reproduce a representation of the demonstrated skill. In LfD, there are two methods of learning and reproducing the demonstrated skill, and that is at a symbolic level or at the trajectory level. At the symbolic level, the skill is learned at an abstract level. The skill is segmented into multiple primitive actions and is represented as symbols. For learning at trajectory level, the skill is learned at a low level and are represented as a trajectory. In Figure 2.1 the generalization of the two methods is shown. Below, first, symbolic learning is described, then the trajectory learning.

As a first approach for the LfD, symbolic learning was adopted in robotics[][][]. Here the position of the end-effector and the forces applied to the manipulated object are stored. The full skill is then segmented into sub-goals (way-points along the skill trajectory) and primitive actions that could reach these sub-goals. These primitive actions are usually point-to-point movements that the earlier industrial robots could perform.

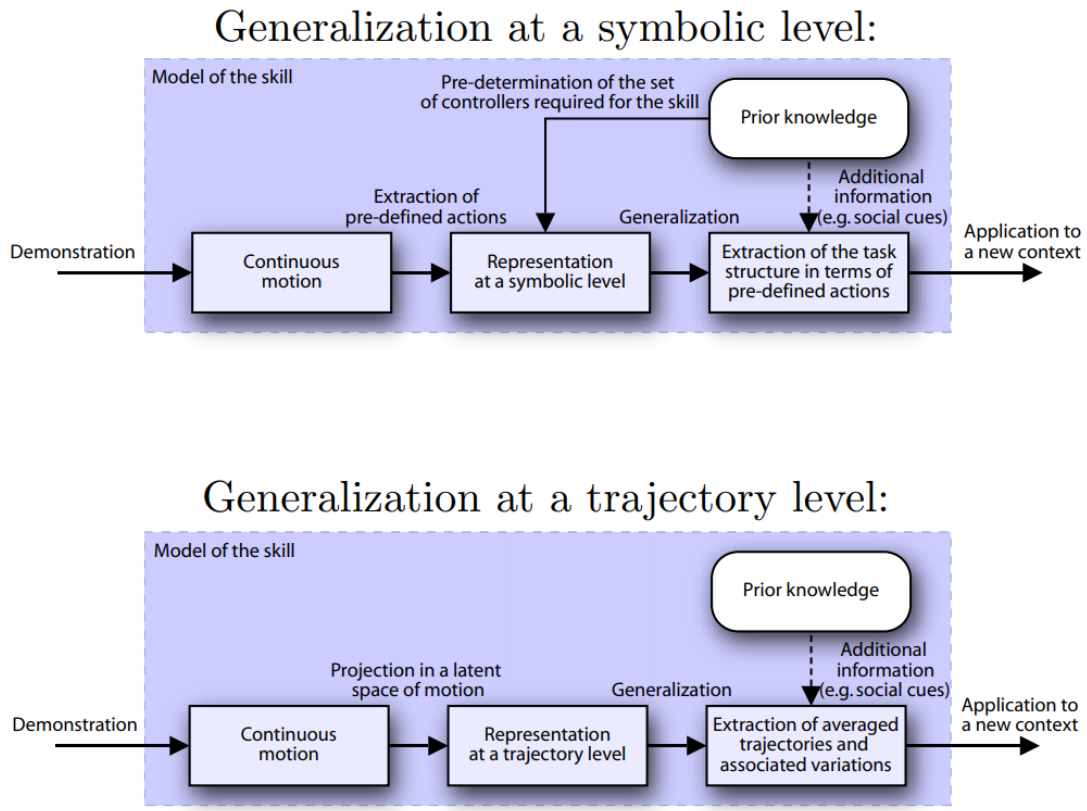


Figure 2.1: Illustration of the different levels of representation for describing the skill. Source: [1]

These would lead that the demonstrated task would be segmented into a sequence of state-action-state transitions. To account for noise in sensors and the human motion during the demonstrated motion, the state-action-state sequences are converted into "if-else" logic. The logic would describe the state and actions according to symbolic relationships, such as "close-to", "in-contact"... These symbolic relationships would then be fed with prior knowledge. This would be a numerical approximation so that system knows when something is "close-to" or not.

The symbolic learning is effective when learning on a high-level like to do a complex task such as baking a cake or setting up a table [2]. For doing a complex skill as doing laundry, it is important to insert clothes to the washing machine before starting the washing machine. This is solved in symbolic learning by adding constraints, as shown in Figure 2.2. Here a set of constraints are set as prior knowledge, that results in the correct order of performing the sub-goals. The drawback is that enough prior knowledge must be provided, and this can result in a time-consuming design process.

Unlike symbolic learning, which is learning a policy at a higher level, the trajectory learning seeks to derive a low-level representation of the demonstrated movement. That

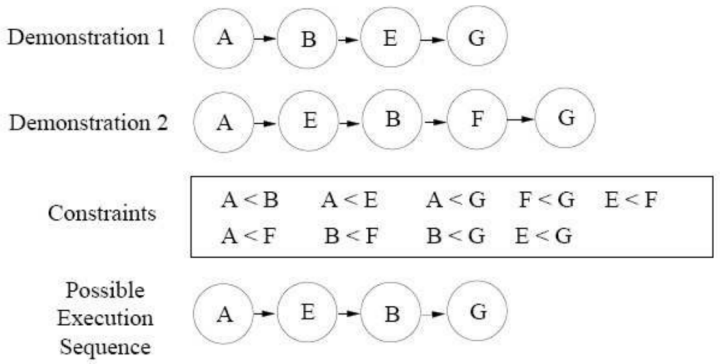


Figure 2.2: Symbolic learning with constraints. Source: [2]

is, representations of joint positions, motor commands or the Cartesian position. Originally encoding of demonstrated movements was done directly by using splines or Bézier's curves [16] [17]. The drawback of these methods is that they only accounts for the position over time, and not other variables such as velocity or acceleration. In recent research, the focus has been put on representing the motion as either, a statistical model or a dynamical model [1].

The idea behind the statistical model is to use mathematical regression. With regression, the goal is to map the relationship between input and output data. With respect to trajectory learning, the input data are the states of the robot, while the output data are the actions performed by the robot. Some of the most known regression methods that are used in trajectory learning are Locally Weighted Regression [13], Gaussian Process learning [18] and Gaussian Mixture Regression [19]. Instead of encoding the trajectories as statistical models, another solution is to use a dynamical system modelling. Dynamic movement primitives (DMP) is a stable well known dynamical system and are expressed as a spring-damper system that ensures convergence towards the goal state. DMP is explained in detail in the next section.

2.2 Dynamic Movement Primitives

Dynamic Movement Primitives, or DMPs, is a planning or trajectory control technique first introduced by Stefan Schaal in [20]. It is based on the principle that movements, or trajectories, are composed of simpler segments that may overlap, called primitives. Finding those primitives allows to recreate complex movements on the basis of easy and simple movements from a lower dimensional space whose convergence and stability is well known. Although Schaal proposed two separate techniques (albeit able to be combined),

due to the nature of the problem to be solved, work here will be focused that deal with non-rhythmic behaviour, or discrete DMPs.

The main benefits of the DMPs are the possibility to use trajectory imitation directly in the Cartesian space (or task space), where movements are easier to generate and interpret, and the lack of manual parameter tuning or previous expert's insight of the problem to be solved for it to work. Moreover, the primitives are found for each degree of freedom of the robot, decoupling the movement and, as stated, reducing the complexity of the problem. Thus, a single technique is able to handle a plethora of movements without further training or adjustments.

2.2.1 Point-attractive System

The proposed movement primitive is a point-attractive system, whose equation is:

$$\ddot{\mathbf{y}} = \alpha_y(\beta_y(\mathbf{g_y} - \mathbf{y}) - \dot{\mathbf{y}}) \tag{2.1}$$

where α_y and β_y are time constants, $\mathbf{g_y}$ is the goal state and $\mathbf{y}$ is the current system state. Note that the system proposed corresponds to an n-dimensional problem defined in the task space. Moreover, since it is defined in discrete time, time is implicit in the formulation. This means that velocity and accelerations can be extracted from the reference trajectory, which is a key point in the problem solution. In the case here presented, the problem is 6-dimensional, so the system would look like,

$$\begin{pmatrix} \ddot{y}_1 \\ \ddot{y}_2 \\ \ddot{y}_3 \\ \ddot{y}_4 \\ \ddot{y}_5 \\ \ddot{y}_6 \end{pmatrix} = \alpha_y\beta_y \left(\begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \\ g_6 \end{pmatrix} - \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} \right) - \alpha_y \begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \dot{y}_4 \\ \dot{y}_5 \\ \dot{y}_6 \end{pmatrix} \tag{2.2}$$

Note that each degree of freedom has its own primitive movement, hence the decoupling. The result is that n 1-dimensional systems are obtained from 1 n-dimensional problem.

This system is a globally stable linear system with exponential convergence, but there is not much room to tune it so it can follow more complex trajectories. Therefore, an

additional term f is added, called *forcing term*,

$$\ddot{\mathbf{y}} = \alpha_y(\beta_y(\mathbf{g_y} - \mathbf{y}) - \dot{\mathbf{y}}) + f \tag{2.3}$$

This makes the system nonlinear but, when chosen wisely, it allows to replicate any other trajectory while still maintaining exponential convergence.

2.2.2 The Forcing Term

The election of a point-attraction system to model the dynamics of the actual physical system had its bases on its simplicity and convergence. When adding a forcing term, it is of prime interest that the election of f is straightforward and it is easily generalized. To achieve that, a *canonical dynamical system* can be defined as,

$$\dot{x} = -\alpha_x x \tag{2.4}$$

whose solution for $x_0 = 1$ is:

$$x(t) = e^{-\alpha_x t} \tag{2.5}$$

This is a very simple system with well known dynamics. It converges to 0 with an exponential decay with a speed proportional to α. Moreover, it allows to transform any function defined in the $[0, +\infty)$ range, which is impractical, into a finite one.

The forcing function is defined as:

$$f(x, \mathbf{g_y}) = \frac{\sum_{i=1}^{N} \psi_i \omega_i}{\sum_{i=1}^{N} \psi_i} x(\mathbf{g_y} - \mathbf{y_0}) \tag{2.6}$$

being $\mathbf{y_0}$ the initial position of the system, ψ_i are bases functions,

$$\psi_i = \exp(h_i(x - c_i)^2) \tag{2.7}$$

and ω_i the weight of the corresponding basis function.

These basis functions are Gaussian functions, each centered at c_i with a variance h_i. They are weighted and added and the result normalized. As x decays from its starting

point (it can assumed to be 1), these Gaussian basis functions activate. The centers c_i are spread evenly across time to achieve an even distribution of the basis function. Since x does not decay linearly, but exponentially, the width of the basis functions cannot be a constant. Otherwise, the very first ones would be too narrow whereas the last ones too wide, yielding disproportional results. To achieve the desired distribution, the variances can be placed as,

$$h_i = \frac{\#BasisFunctions}{c_i} \tag{2.8}$$

Returning back to the forcing term, the x that appears there is to ensure that the forcing term does not disturb the point attraction dynamics of the system as it is reaching its goal. Effectively, as x decays and gets closer to 0, the importance of the forcing term in the system is reduced too.

The last part of the forcing term is $\mathbf{g_y} - \mathbf{y_0}$, which appears as an spatial scaling term. This allows to change the goal so that, for instance, the desired trajectory is twice the original one. By doing so, the forcing term (and hence the basis functions within it) is altered as well to follow this new trajectory. This is extremely useful because it allows the system to follow scaled up or down trajectories without any kind of tuning needed.

Temporal scaling can also be achieved via the introduction of an additional parameter τ. If it is greater than 1 the trajectory is executed faster, whereas if it is comprised between 0 and 1 it is done slower. The only changes needed are,

$$\dot{\mathbf{y}} = \dot{\mathbf{y}} + \tau\ddot{\mathbf{y}} \tag{2.9}$$

$$\mathbf{y} = \mathbf{y} + \tau\dot{\mathbf{y}} \tag{2.10}$$

$$x = x + \tau\dot{x} \tag{2.11}$$

The forcing term defined previously was added with the aim of imitating a desired trajectory while still maintaining point attraction dynamics. It was defined as a combination of weighted Gaussian basis functions and some other terms, but the weights have not yet been defined. From the original trajectory, the accelerations can be computed as,

$$\ddot{\mathbf{y}} = \frac{\partial}{\partial t}\dot{\mathbf{y}} = \frac{\partial}{\partial t}\frac{\partial}{\partial t}\mathbf{y} \tag{2.12}$$

From the original system in Eq. 2.1, f can be extracted as (considering now only a

single dimension, the process is the same for n dimensions),

$$f = \ddot{y} - \alpha_y(\beta_y(g_y - y) - \dot{y}) \tag{2.13}$$

From Eq. 2.6, the weights can be chosen so that the forcing term matches the specified trajectory. To achieve that, several solutions are available. Here it is proposed to use Locally Weighted Regression (LWR) to find these weights since it is fast and does not need training of any kind, albeit the use of Neural Networks or Reinforcement Learning techniques is also possible. In this case, the objective function, for each one of the n dimensions,

$$\min \sum_t \psi_i(t)(f_d(t) - \omega_i(x(t)(g_y - y_0)))^2 \tag{2.14}$$

The solution was found by Schaal as well, and it is,

$$s = \begin{pmatrix} x_{t_0}(g - y_0) \\ \vdots \\ x_{t_N}(g - y_0) \end{pmatrix} \quad , \quad \psi_i = \begin{pmatrix} \psi_i(t_0) & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & \psi_i(t_N) \end{pmatrix} \tag{2.15}$$

Chapter 3

Resources

A large part of this master thesis work is about integrating existing and well-tested tools and libraries to a complete system. Moreover, without these tools and libraries, the focus of this master thesis would have to be something different. In this chapter, a short introduction of these tools and libraries is provided to give a better understanding of why they are used in this project.

3.1 ROS

The Robot Operating System (ROS) [3] is a middleware, that is computer software. This is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms.

ROS is responsible for connecting different software components or applications. The reason for using middleware is to increase,

1. Portability: Middleware offers a standard programming model across languages and platforms.
2. Reliability: Middleware is developed and tested by itself, and are separated from the final application.
3. Managing complexity: Suitable libraries manage low-level aspects, such as hardware

control.

A robot is a complex entity that can have a wide variety of sensors and actuators. ROS helps the communication between them by having a common communication channel and are using interfaces that are consistent between different applications.

ROS provides utilities such as hardware abstraction, libraries, visualizers, message-passing, package management, and more.

The core of the ROS middleware is the use of nodes and topics and sending messages of a defined type between them as shown in Figure 3.1. A node is a process that performs some computation or a task. A type of computation can be to compute the inverse kinematics of a robot, and a task can be to retrieve images from a camera. Instead of directly sending this information to another application, the nodes then send the information to a specific place on the ROS network, which is called a topic. Another node that needs this information can then subscribe to this topic and receive this information. An analogy of this process would be a radio channel transmitting its broadcast to a specific frequency. Then if you have a radio, and want to listen to this broadcast, the radio has to listen in to that specific frequency.

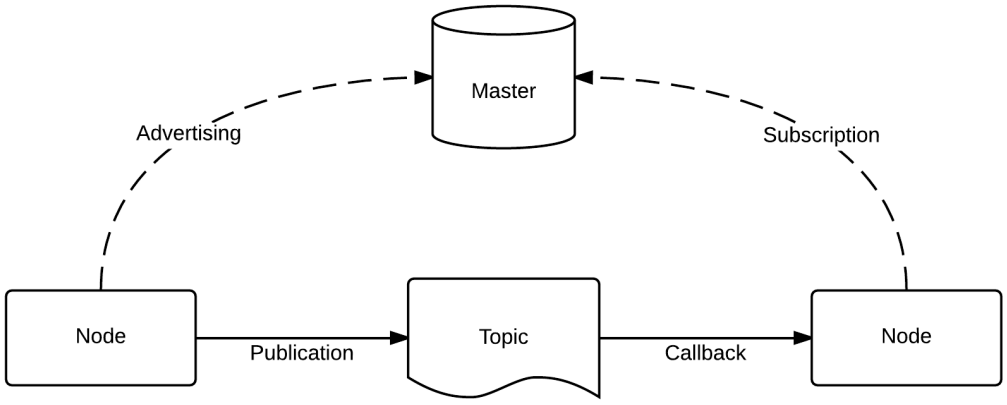


Figure 3.1: Illustration of the communication between nodes. Source: [3]

3.1.1 ROS DMP Library

In ROS there is available a DMP package that was developed by Scott Niekum [21]. This package is a general implementation of DMP and provides the more stable reformulation of DMP that is described in the paper [22]. The DMP package is able to provide learning

of multi-dimensional DMP from demonstrated trajectories and can generate both full and partial for arbitrary starting and goal positions. This package follows the formulation where N DoF are formulated as N separate DMPs with a common phase system. The package acts as a server and provides three services that can be requested by other nodes. These services are `learn dmp_from demo`, `set_active_dmp` and `get_dmp_plan`.

1. `learn_dmp_from_demo`. This service takes as input the demonstrated trajectory and together with the DMP parameters for the gains and number of basis functions, the service returns the learned multi-dimensional DMP.

2. `set_active_dmp`. The user can have demonstrated and learned multiple trajectories and stored them in a library. With this service, the user can find the desired motion and set that motion as active on the DMP server.

3. `get_dmp_plan`. When a learned DMP is active on the server, the user can request to generate a trajectory of this DMP based on some input parameters, and the output is a full or partial plan for the new trajectory. The input parameters that are needed to be specified are:

    - `x_0`: Starting state of the new trajectory.
    - `X_dot_0`: The first derivative of state. from which to begin planning
    - `T_0`: Time in seconds from when to begin planning. Usually, 0, unless piecewise planning.
    - `Goal`: The goal position that the DMP should converge towards.
    - `Goal_tresh`: A threshold for each dimension that the plan must come within before stopping planning.
    - `Seg_length`: The length of the plan in seconds. Can be used to do piecewise, incremental or replanning. It is set to -1 if planning until convergence is desired.
    - `tau`: It is used to temporal scale the DMP. Higher values increase the time the trajectory use to complete. A smaller value will make it faster.
    - `dt`: The time resolution of the plan in seconds.
    - `integrate_iter`: The number of times to numerically integrate when changing acceleration to velocity to position. This can usually be 1 unless dt is fairly large (i.e. greater than 1 second), in which case it should be larger.

3.1.2 Universal Robots ROS Driver

The Universal Robots ROS driver [23] is a ROS package that has been developed in a collaboration between Universal Robots and the Forschungszentrum Informatik (FZI) Research Center for Information Technology. This driver works for all CB3 and the newer e-Series versions of the Universal Robots. It uses the Real-Time Data Exchange (RTDE) interface for communication. RTDE interface uses the standard TCP/IP connection to provide a way to synchronize external applications with the UR controller without compromising any real-time properties the UR controller has.

Universal Robots ROS driver provides tools to make communication and developing applications for UR easier. It contains vendor-specific robotic models, parameters, interactions, communication interfaces, controllers and visual representation. One of the helper packages contains the Unified Robot Description Format (URDF) of the available UR robots. URDF describes the kinematics and dynamic properties of the robot manipulator and can also include the robot's mesh so that collision checks can be performed.

3.1.3 MoveIt

MoveIt [4] was developed in 2011 at Willow Garage. However, MoveIt is now an open-source project and results from an extensive collaboration between the international community and multiple organizations. MoveIt is a user-friendly platform for building flexible applications and provides modules for collision checking, kinematics, motion planning, perception representation, interfaces to controller, execution and monitoring, and kinematic analysis. In this master thesis, the modules of collision checking and kinematics will be used, as the DMP will take care of the trajectory planning and the default ROS action controller will execute the trajectory. In Figure 3.2 the pipeline of the Moveit library is showed. It starts with the user or AI request, and ends up in the robot controller for execution. In between, motion planning and collision checking are performed.

For the kinematics part, MoveIt is designed to use a plugin infrastructure. This is targeted towards users that want to develop their own inverse-kinematic (IK) algorithm. However, the default IK algorithm in MoveIt is the Kinematics and Dynamic Library (KDL) developed by the Orocos project and are proficient enough for this master thesis.

Collision checking in MoveIt is configured inside a planning scene. A planning scene is a representation of the world and the current state of the robot. MoveIt is set up so

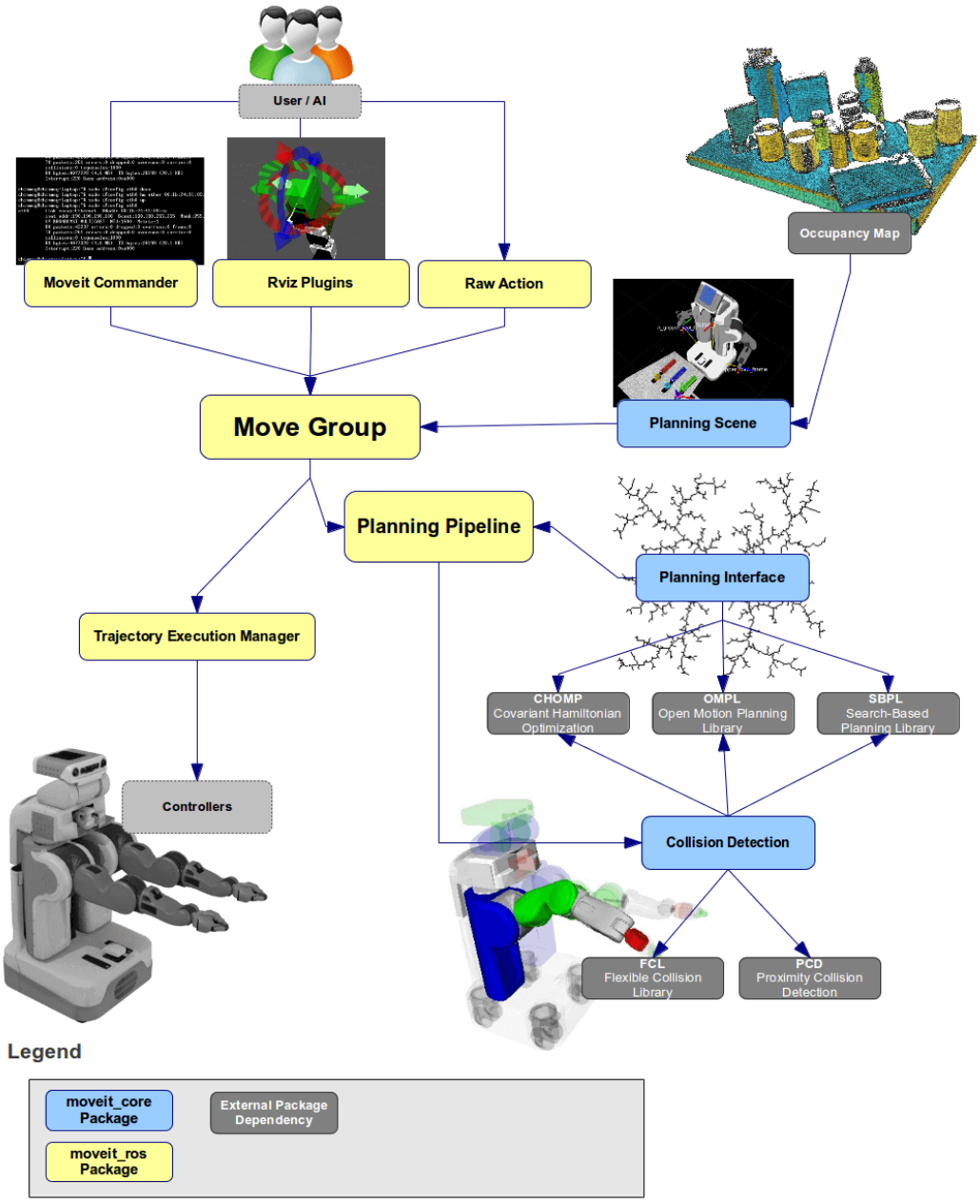


Figure 3.2: MoveIt pipeline. Source: [4]

that how the collision checking is performed is not something that the user has to worry. However, the collision checking in MoveIt is usually carried out by the Flexible Collision Library (FCL) package. MoveIt supports collision checks for different types of objects, including meshes, primitive shapes (boxes, cylinders, cones) and octomaps. Collision checks are one of the most expensive computational parts of the planning pipeline, and in motion planning, collision checks account for up to 90% of the computational expense. To solve this, MoveIt is using Allowed Collision Matrix (ACM). With a defined ACM, collision checks between links and joints that can not collide with each other are not performed.

3.2 Programming Languages

Applications in ROS are mainly developed in two languages, C++ and Python [24]. Other languages are, however, available, but they are not fully supported. To decide which language to use for developing ROS applications comes down to what kind of application to be developed. Moreover, what can be done in C++ can also be done in Python. However, they have their differences.

Python is an interpreted high-level language that does not need to be compiled before runtime. Furthermore, the design philosophy of Python is to emphasize human readability. With this philosophy, a user can develop a prototype application fast, and the program can be short, even for complicated things. Python is also the most preferred language for data science, resulting in a large amount of available third-party libraries. The drawbacks of the python language are that it runs slower than other languages. Python is also more prone to encounter errors during runtime since specific parts of the code are not checked before it is at that specific part of the code.

The significant drawbacks of the Python language are the strengths of the C++ language. C++ is an extension of the C language and created by Bjarne Stroustrup. C++ was designed to be used in systems programming and embedded systems that have resource constraints. This resulted in a design philosophy of making the language following the criteria of high performance, efficiency and flexible of use. With this design philosophy, the C++ programming language runs really fast, and since the language needs to be compiled, errors are caught before runtime. However, the resulting drawbacks are that C++ is a lot more complicated than Python and require much code to develop small programs.

However, the strengths of the ROS framework shines here. Since the communication between nodes uses standard messages, Python nodes can communicate with C++ nodes. Furthermore, because of the standard communication, expensive computational algorithms such as IK and collision checks can be programmed in the C++ language. While prototyping user interfaces can be developed using the Python language.

3.3 Universal Robots

Universal Robots (UR) [5] is a robot manufacturer located in Denmark. They produce small flexible industrial collaborative robot arms. The products range from the small table-top UR3 and UR3e through the UR5/UR5e and UR10/UR10e to the heavy-duty UR16e. The different models can be seen in Figure 3.3.

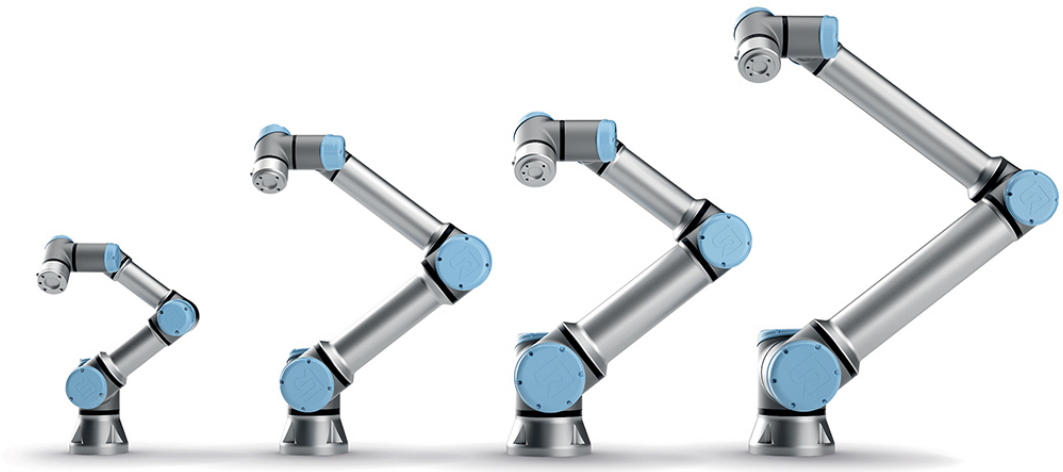


Figure 3.3: Image of the Universal Robots product series. Source: [5]

The number in the product names is correlated with the lifting ability of the robot. The UR3 can lift 3 Kg while the UR16 can lift 16Kg. UR's cobots are six-jointed arm robots and have a low weight ranging from 11 Kg for the smallest to 30 Kg for the largest including cables. In 2018 UR introduced the e-Series, and the robot can be identified by the name ending with an e such as UR3e. The new e-series improved the accuracy of the position, force and torque repeatability since it has a built-in force/torque sensor. The robot that is used in this thesis is the UR3.

3.3.1 UR3

UR3 is the smallest cobot from UR and has six rotating joints. The cobot has a reach of 500mm and can lift payloads of up to 3 Kg. Every joint has 720-degree of rotation except the end joint, which has infinite degrees of rotation and the speed of joint rotation is maximum 180 degrees/sec. The power consumption is stated in the datasheet to be between 90W and 250W, with typical use of 125W. For input/output ports, the UR3 provides two digital in, two digital out and two analogue input. The UR3 also provides a 12V/24V 600mA connection to connect a tool to the end-effector.

UR3 is a precise robot with repeatability of movement with errors ±0.1mm. The repeatability is a measure of the spreading of end-positions when the same task is performed multiple times.

Chapter 4

Implementation

This chapter aims to explain how a final solution has been implemented based on the theoretical background. The system developed will be explained in two parts. The first part is about the back-end, where the core of the system lies. Here, trajectory recording and saving are performed. Furthermore, DMP computation, trajectory generation, collision checks and finally execution of the trajectory are also a part of the core program. The second part, which is the frond-end will describe the graphical user interface and how this interface interacts with the core of the program.

4.1 Back-end Implementation

As it can be seen in Figure 4.1, the general pipeline follows an algorithm that starts with the user recording a trajectory and storing it in a rosbag format. The next step is then to compute the corresponding DMP weights for that trajectory and store them as YAML [25] files in a library for later use. From this library, the computed DMP weights are used to generate a trajectory. Finally, the collision check module is performed for all trajectory points. Now, the algorithm executes the selected trajectory on the real robot. Below a more detailed description of the process is explained.

Figure 4.1: Application pipeline.

4.1.1 Preliminaries

A connection with the robot is established before the user can start to record or send commands to the UR. The UR3 is connected to a WIFI router with an Ethernet cable. By connecting to the same WIFI network, a connection can be established between computer and robot. The Universal Robot ROS driver performs the handling of making the connection. The driver needs as input the IP address of the robot. This IP address is found in the settings of the teach-pendant and is made static. It is made static so the same IP can be used every time when connecting. To control the UR3 CB robot from the computer with ROS, a URCaps program must be installed on the robot, called "Externalcontrol". This is a small program that, when running on the robot, handles commands from ROS. When using it, the host IP address is entered into the teach pendant's external

control program.

The driver from UR comes with a robot description of the robot in a URDF format. Since the robot description contains the robot's dynamics and kinematics, the MoveIt package uses this description for the collision check. The URDF from UR includes the robot's description, however no description exists of external tools, grippers or environment. To make sure that collision check is performed for the volume that the gripper occupies and the table the robot stands on, the URDF file containing the robot description has been modified. A description of the robot gripper has been added to the UR3 URDF file. A planar surface of 1×1 meter is also included in the URDF to act as the table that the robot is standing on. As seen in Figure 4.2, the new robot description includes a gripper and the table where the robot is standing on.

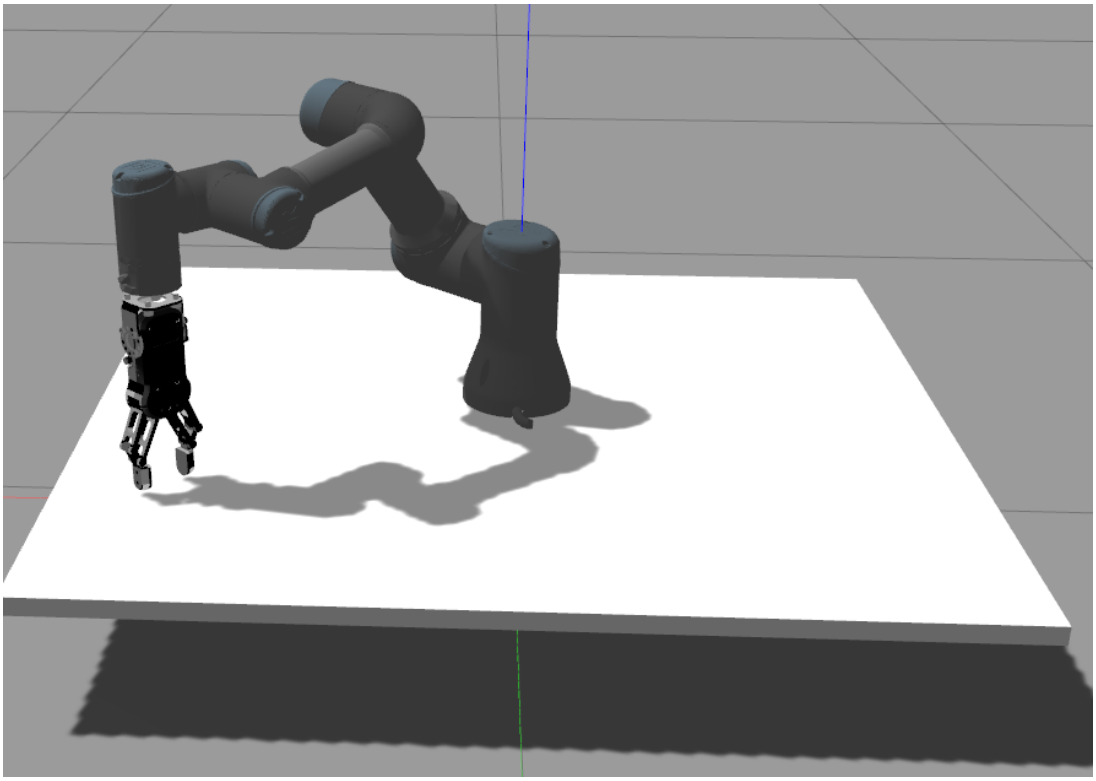


Figure 4.2: Robot description as seen in the Gazebo simulator. The new description includes a gripper and a table.

Furthermore, the application relies on the collision checking module from the MoveIt package. The package from UR comes with a MoveIt configuration, however since the robot description has been modified a new MoveIt configuration has been defined. The MoveIt configuration describes which joints and links are included in the robot and what joints and links the collision check should consider. However, since the collision check is an expensive computation, links and joints that can not collide are exempted from the collision check. An example of such links are the links connected to the elbow joint. These

links are in parallel, and will never collide with each other.

4.1.2 Recording

There are two ways to record a trajectory in this application. That is to record internally from the robot states or from an external source.

Internal Recording

This robot driver opens up a Real-Time Data Exchange (RTDE) between the robot and the ROS network. When the connection is established, the robot driver is publishing the values of the joints on the *joint_states* topic. The format of the messages that are published on the *joint_states* topic is of sensorMsgs/jointState. The jointState message consist of a header, name of the joints and a list for position, velocity and acceleration.

These messages are published by the robot driver at a speed of 125Hz and stored by the application as rosbags. For recording a trajectory with the joint states, there are three methods. The first method is to use kinesthetic force as shown in Figure 4.3 . In this method, an external force (usually a human pushing with hands) is put on the robot to make it move. To enable the UR robot to move with an external force, the user has to set the robot in "free drive". When the robot is in "free drive", only the gravity compensation is active on the joints. Making it possible for the user to move the robot joint around.

The second way of recording the joint states is to program a trajectory with the corresponding teach-pendant as seen in Figure 4.4. Teach-pendant is touch screen connected to the robot with a cable. With this screen, the user can control the robot and programs trajectories. When the user has programmed a trajectory, that trajectory can be played on the robot. When this trajectory is played, the application can record the movement.

The third method is to teleoperate the robot with an external device. This would be similar to the first method, but instead of exerting forces directly on the robot, the robot is controlled with a device such as a joystick.

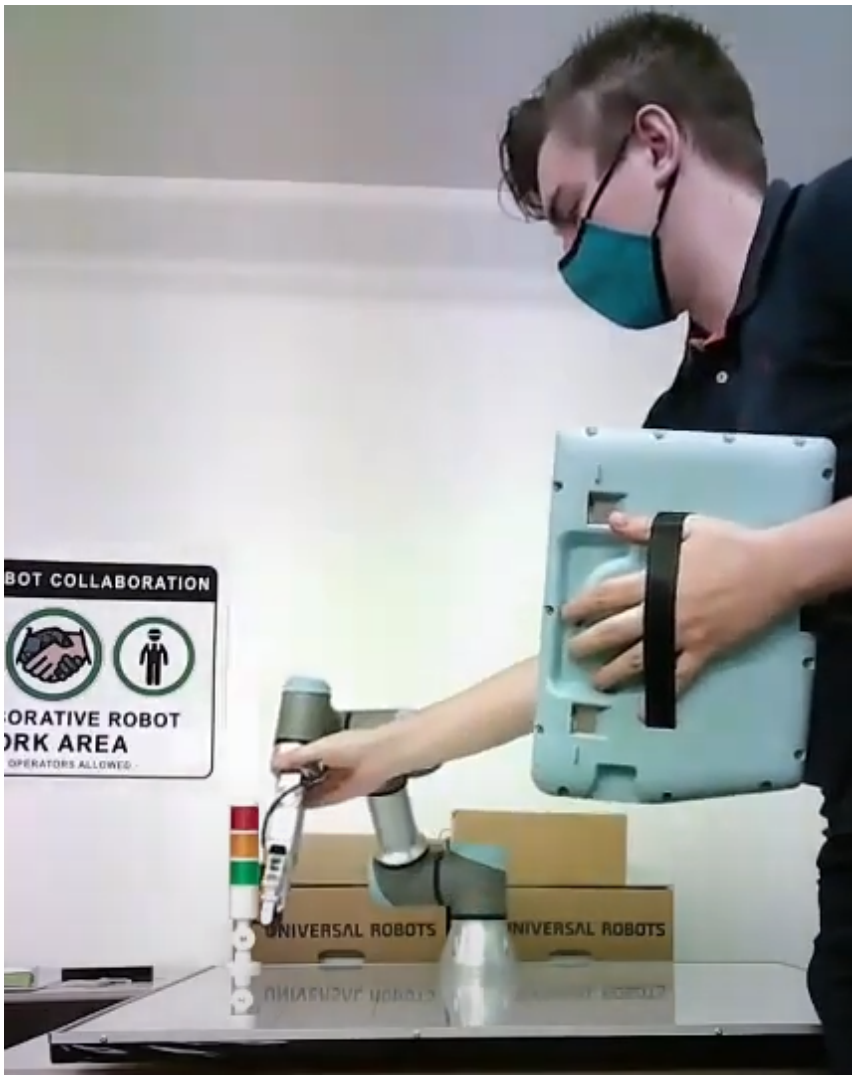


Figure 4.3: Kinesthetic demonstration on the UR3 by using the freedrive option on the teach-pendant.

External Recording

The application is set up to record from external sources. To enable recording from this external source, the external source needs to publish a ROS poseStamped message to the /end_effector_pose topic. A pose stamped message contains the Cartesian position of the end-effector in a format that describes the (x, y, z) position and the corresponding quaternion. The PoseStamped message also includes the timestamp for that given message. Although the feature is implemented, it is not fully tested and is out of the scope of the project.

When the recording is finished, the trajectory that is recorded is saved as rosbags in the Data/recordings folder inside the application package. These recordings are then used

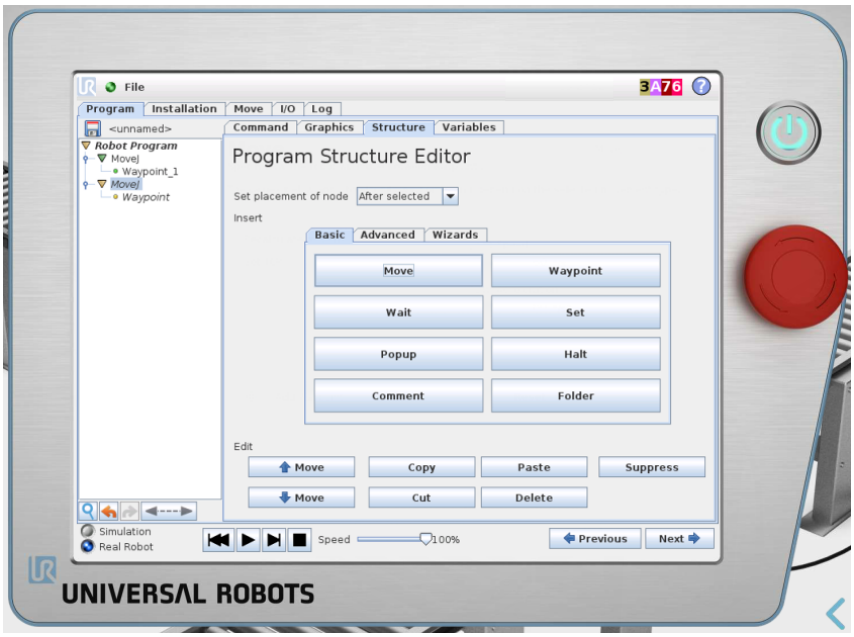


Figure 4.4: Teleoperation using a program created on the teach-pendant Source: [5]

in the DMP learning phase that is described in the next part.

Python Implementation

In Figure 4.5 the workflow of the recording module is shown. The recording module is made up of a Python object that contains four methods. These four methods are:

- Object initialisation: The initialisation method's main objective is to connect to the */joint_states* topic and set the recording variable as disabled.
- A ROS callback method: The objective of the callback method is to process the messages published from the robot. In this method, there is a conditional check. This conditional check is true if the recording variable is set to true. When true, the callback function will append all messages received into a list.
- Start recording: The main objective of this method is to set the recording variable true. External variable inputs are handled by this method. These variables are the motion's name, what joints to record, and the recording's desired filename.
- Stop recording: This method first set the recording variable back to false. Setting the recording variable false stops the appending of new ROS messages in the callback function. The method then proceeds to open up a rosbag with the name received in

the “Start” method. The appended list is then written into this rosbag, and when finished writing, the rosbag is closed.

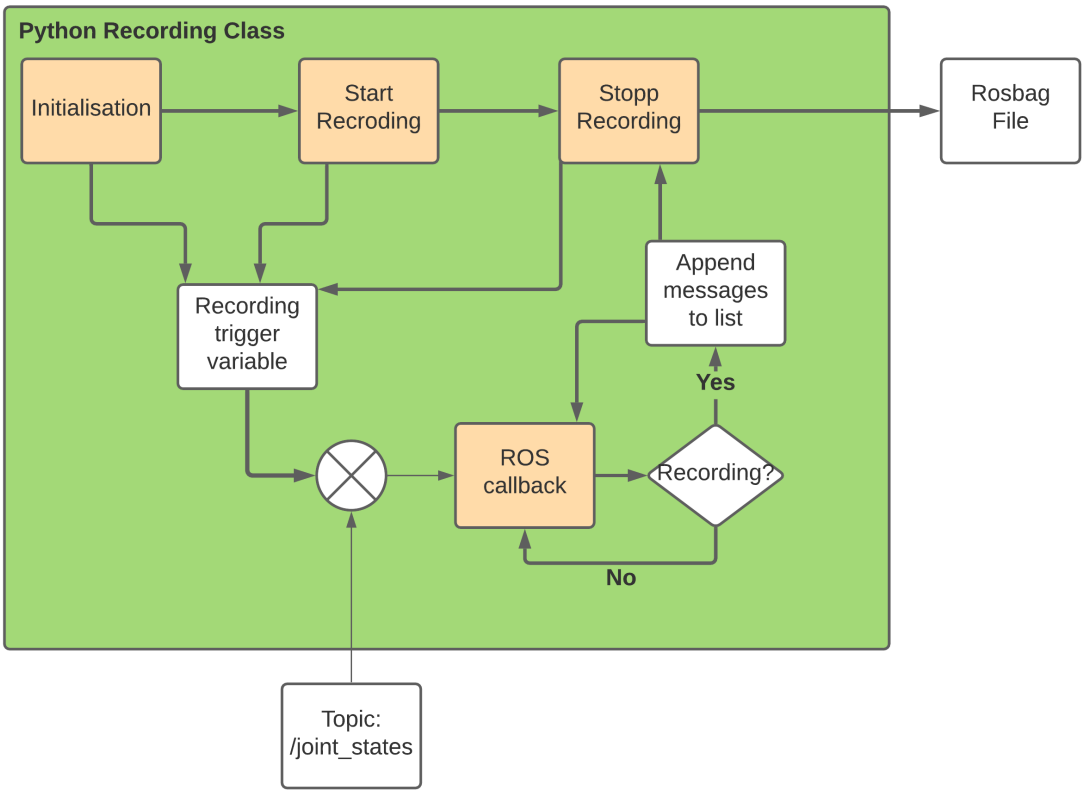


Figure 4.5: Pipeline of the recording class in Python.

4.1.3 DMP Learning

To compute the DMP weights, a request is sent to the DMP server that will perform the computations, and return the result. The DMP server from ROS_DMP package requires that the request is sent in a specific format. The specific format of the request requires that For learning the DMP weights of the trajectory, the trajectory are first loaded from the stored rosbag file. To compute the DMP weights from a trajectory, a service request has to be sent to the DMP server.

The DMP server is another node that has been started and is running on the local ROS network. This DMP server is performing the computation of converting the trajectory into DMP weights and returning these weights back to ros node that requested them. The required format of the service message contains the positions, velocities, time, gains and number of basis functions. When the DMP server has computed the corresponding weights, the server response with a list for each DOF with weights and a time constant

that will cause the DMP to replay at the same speed as demonstrated. The response from the DMP server is stored as YAML files. These stored YAML files will now act as a motion library and can be made the active motion on the DMP server. When a DMP weight file is active on the DMP server, a trajectory can be generated using the DMP weights and user-specified start and stop positions.

Python Implementation

In Figure 4.6 the workflow of the learning module is shown. The learning module is made up of a Python object that contains three main methods and some helper methods. These three methods are:

- Load a recording: The main objective of the method is to load a previously recorded trajectory, and compute the DMP weights of that trajectory. This is performed by extracting the joint positions in the rosbag containing the jointState messages from the recording. The extracted joint positions together with the input-parameters dt, K,D and num_bases are then sent to a helper method to construct a service request message. This service request then computes and returns the DMP weights. The DMP weights are then stored as a python dictionary in a YAML file. Also when a new DMP has ben computed of a recording, this new DMP is set as the active DMP on the DMP server.

- Load DMP: The objective of this function is to load previously computed DMP and make them the active DMP on the DMP-server. This is done by sending a service request to the DMP-server to make the loaded DMP active.

- Retrieve Trajectory: The goal of this method is to retrieve a trajectory plan by using the current active DMP on the DMP-server together with an initial and goal position of trajectory. This is performed by constructing a service request message containing initial and goal position, the threshold for successfully reaching the goal, tau and dt. The constructed service request message is then sent to the DMP-server. The DMP-server are then sending a response with the generated trajectory.

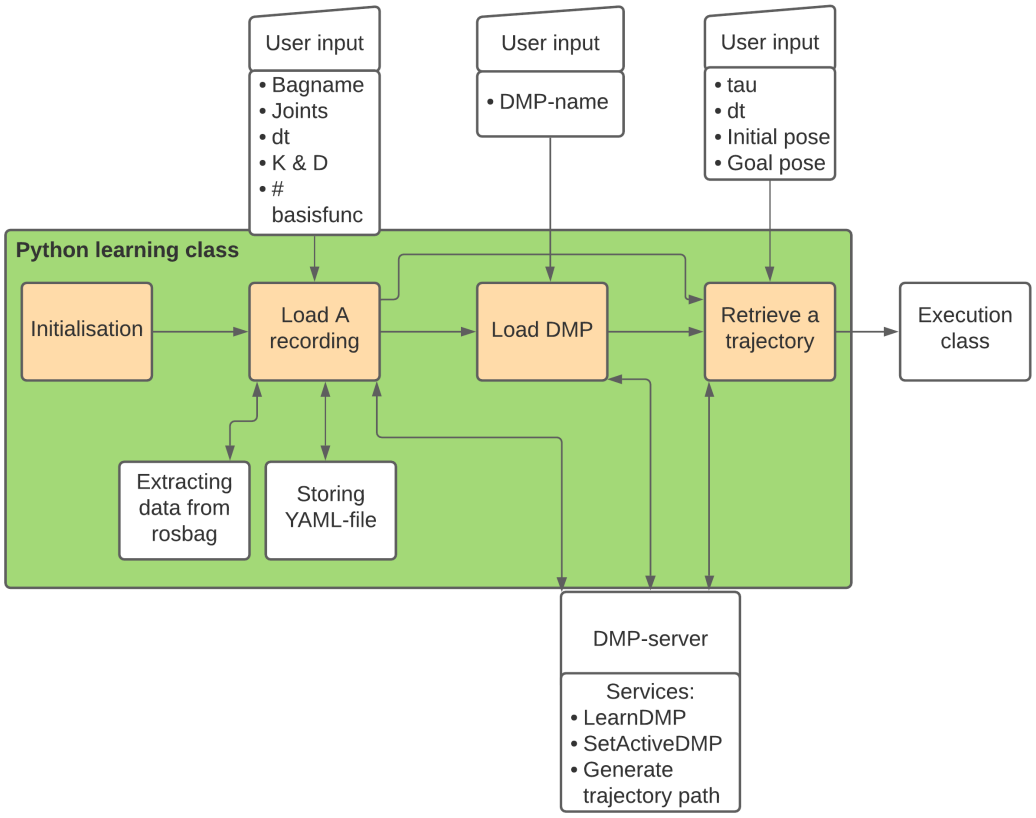


Figure 4.6: Pipeline of the learning class in Python.

4.1.4 Execution Phase

When a given DMP weight file is active on the DMP server. A service request can be sent to the DMP server to compute a trajectory based on the DMP weights. This new trajectory will follow a similar shape as the one demonstrated when recording the original trajectory but will start and end from user-defined positions. The requested service message needs to follow a specific format and is explained in the resource chapter. The response from the DMP server is a trajectory that includes positions and velocity for the joints, and also the timestamps for when each position should be reached. Before the responded trajectory are allowed to run on the robot, collision checking is performed.

In this application, the collision checking is performed by the external library MoveIt. The trajectory is converted to a special MoveIt message RobotTrajectory. A service request is then populated by this RobotTrajectory and sent to the MoveIt node. This node is then performing a collision check for each point in the trajectory to check if the robot collides with itself or the table it stands on. The response from the MoveIt node is an integer number corresponding to an error code. And the error code tells if the trajectory

is valid or not valid. If the trajectory is not valid, a new set of start and goal positions are needed to be defined and another trajectory must be generated. If the trajectory is valid, the trajectory can be sent to the robot.

When sending the trajectory to the robot, the UR robot driver has exposed an action server that requires a follow_joint_trajectory message. The trajectory that is computed from the DMP server is then used to populate a FollowJointTrajectoryAction message. This message is then sent to the robot over the action server. When the robot has finished the trajectory and is within the specified goal threshold, the action server sends a response back to the application that the trajectory is finished.

Python iImplementation

In Figure 4.7 the workflow of the execution module is shown. The execution module is a Python object containing three main methods and some helper methods. These main methods are:

- Check trajectory validity. This method is responsible for checking the validity of the trajectory. The method performs this by sending the trajectory as a service request to the MoveIt node. The MoveIt then performs collision on this trajectory and returns an integer number. This number represents an error code, and when the returned number is “1”, the trajectory is valid. If the trajectory is valid, this method returns a boolean value of “True”.

- Publish the end-effectors Cartesian path. This method performs the publishing of the end-effectors Cartesian path. To retrieve the Cartesian path, the FK is performed on all the trajectory points consisting of joint values. The FK is done by utilising a service provided by the MoveIt package. When FK is done, the resulting Cartesian path is published on the /imitated_path topic.

- Send trajectory action. The main goal of this method is to send the trajectory to the robot. First, the generated trajectory path is converted to a JointTrajectory message. This message is then sent as a command to the action server opened up by the UR ROS driver. The method is then waiting for a confirmation from the action server that the trajectory has finished. The methods return a boolean value “True” if trajectory has finished without error.

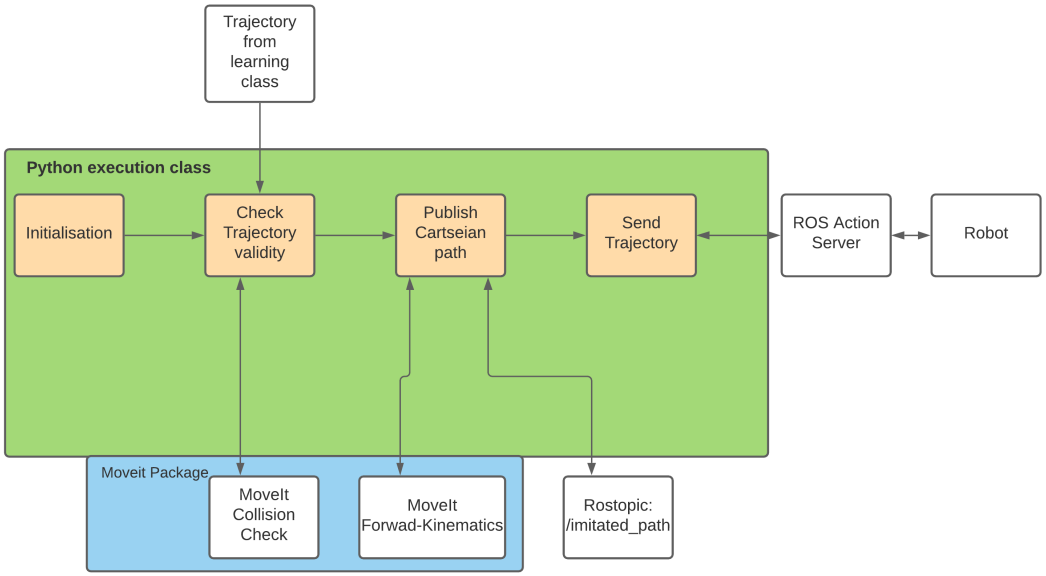


Figure 4.7: Pipeline of the Execution class in Python.

4.2 Graphical User Interface

In this section, the developed graphical user interface (GUI) will be explained. The first screen that appears when starting the application is the login screen. Next, when user is logged, a tab with three different screens appear. These three different screens correspond to "Recording features", "DMP learning", and "Trajectory execution", that is, the different steps in our application. A more detailed explanation is now provided about the different screens and their features.

4.2.1 Login Screen

The first screen that appears when starting the application is the login screen, as shown in Figure 4.8. Here the IP is set for the connected robot. There is also an option to run the application on a Gazebo simulator instead of a real robot. This GUI is developed for use with the UR robots and tested on the UR3. Therefore the user has an option to choose between robots UR3, UR5 and UR10. The user pushes the "Connect to Robot" button when the desired parameters have been selected. When the "Connect to Robot" button is pushed, a ROS launch file is called. This launch file includes the UR robot driver, ROS_DMP and the MoveIt packages. When the application receives a confirmation that these nodes are up and running by using the ROS command wait_for_service(), the "Start DMP Application" button is enabled and is possible to be pushed (see Figure 4.9). The

"Start DMP Application" button initializes the different classes from the core program. Its activation also changes the screen to tab screen that contains 'recording', 'learning' and 'execution' screens.

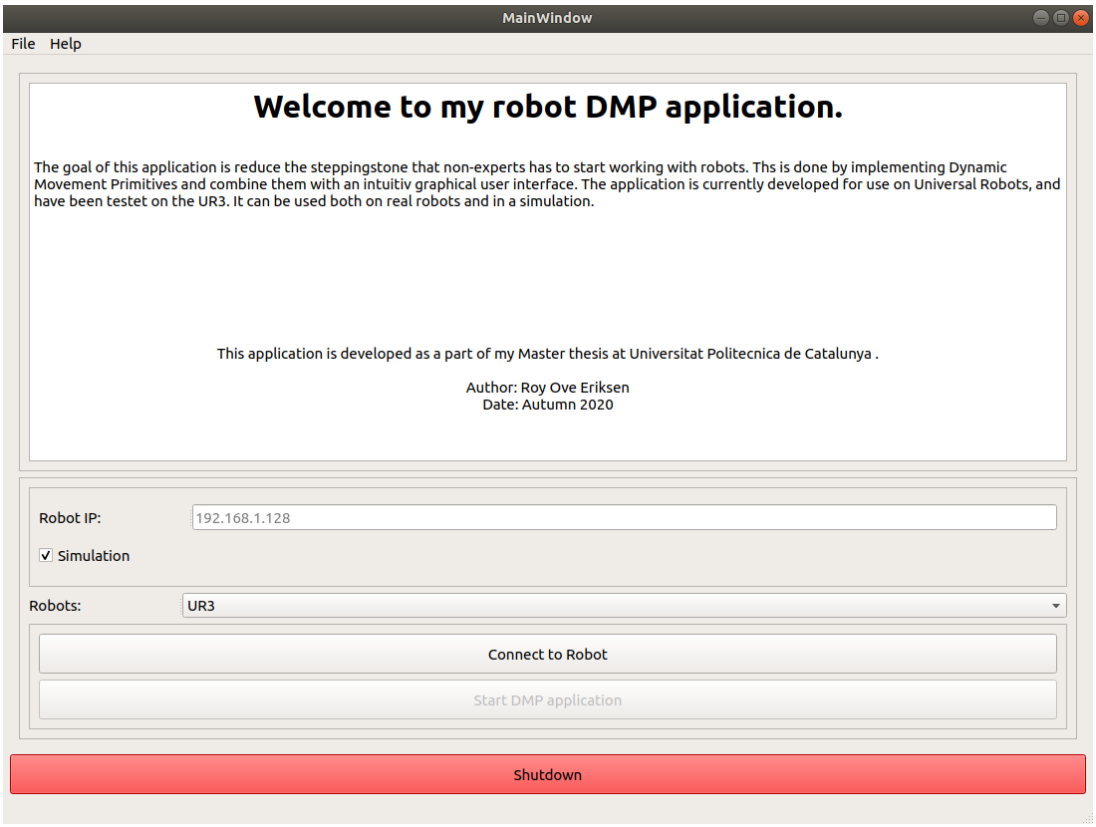


Figure 4.8: Initial login screen when nothing is connected

4.2.2 Recording Tab

The "Recording" tab in the "MainWindow" screen (see Figure 4.10) gives the user the possibility to record either, the end-effector's Cartesian position or the joint states of the robot. Both choices connect to the corresponding topic and record the given topic in a rosbag. When the user has chosen what type of recording will be performed, the user enters the recording name and pushes the "Start Recording" button to capture the motion to be later imitated. When the user wants to finish the recording, the "Stop Recording" button is pushed. This button will then change to respond to the user that the recording is stopped and saved with the provided name.

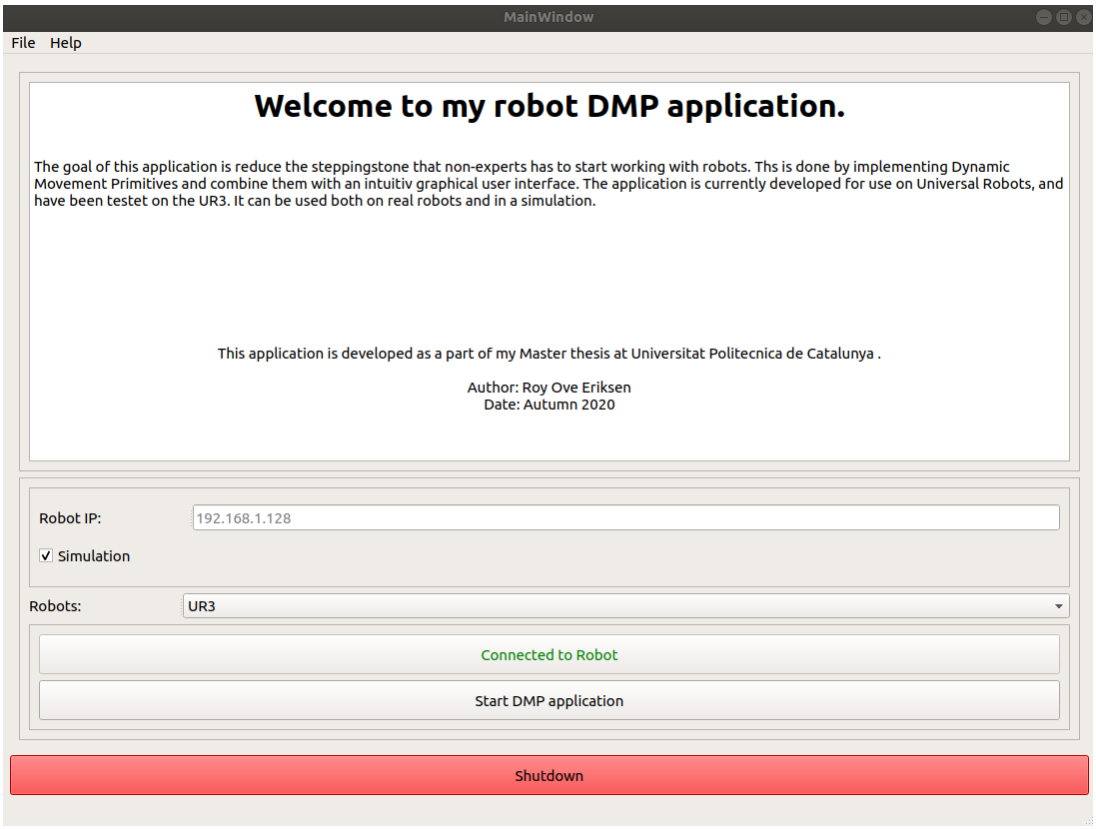


Figure 4.9: Loginscreen when launchfiles has been launched and connection to robot established.

4.2.3 Learning Tab

In the "Learning tab", the user first chooses what type of recording will use to generate DMP from, either the Cartesian positions of the end-effector or the whole robot's joint states. Currently, only the joint states option has been thoroughly tested out. After choosing the recording type, the user has the option to enter the different DMP parameters. The application is set up with default parameters that have been empirically tested to provide satisfactory results for most trajectories.

Whether the user chooses to tune the parameters, variables connected to the given textbox are updated every time a change is performed (see Figure 4.11). The textboxes, however, will register valid types of input. For dt, K and D variables, the accepted values are `FLOAT`, while the # basis_func variable accepts `INT`. The textboxes are encapsulated in try/catch, so if a wrong type is inserted, the variable will maintain the previously correct type as the current value.

After choosing the type of recording, and optionally tuning the DMP parameters, the user chooses which previously demonstrated recording to generate DMP from. Here a

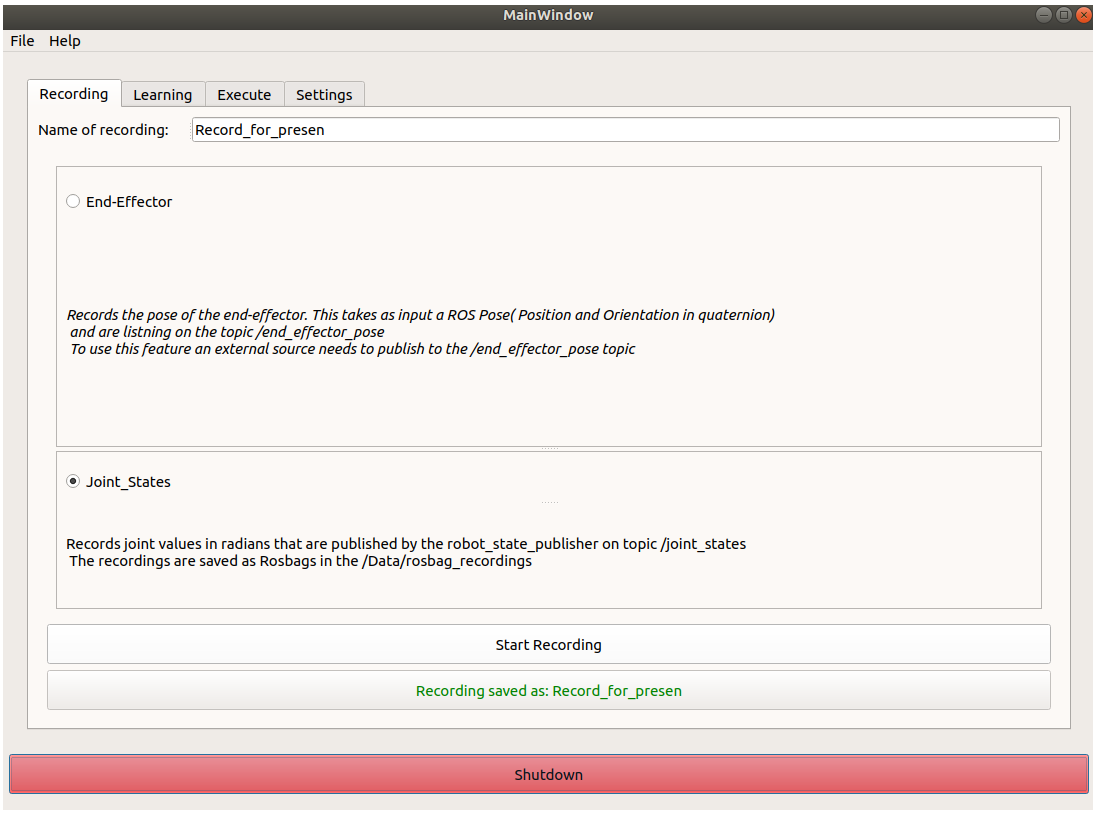


Figure 4.10: Display of the "Recording" tab.

dropdown menu of all previous recordings is available for the user to choose from. This dropdown menu is updated every time the user enters the "Learning screen". When the user has chosen all desired options, the user pushes the "Generate DMP button". When this is pushed, the application sends a request to the DMP server to compute the DMP weights. The button also changes text colour when the application receives the response, to give feedback to the user that DMP is generated.

4.2.4 Execute Tab

The "Execute" tab is where the user sets the active DMP on the DMP server, generate trajectory path and send that trajectory path to the robot to be executed. Firstly, the user chooses the available previously generated DMP's weight files from a dropdown menu and set this DMP weight file as active on the DMP server. When the user has set an active DMP on the DMP server, the initial and goal positions should be determined. There is an option to use joint values, as shown in Figure 4.12 or Cartesian coordinates, as shown in Figure 4.13. The option for Cartesian coordinates is implemented and can retrieve the robot's current position and oritentation (in quaternions). This option retrieves the Cartesian pose by first reading the joint states, and then populating a service request to

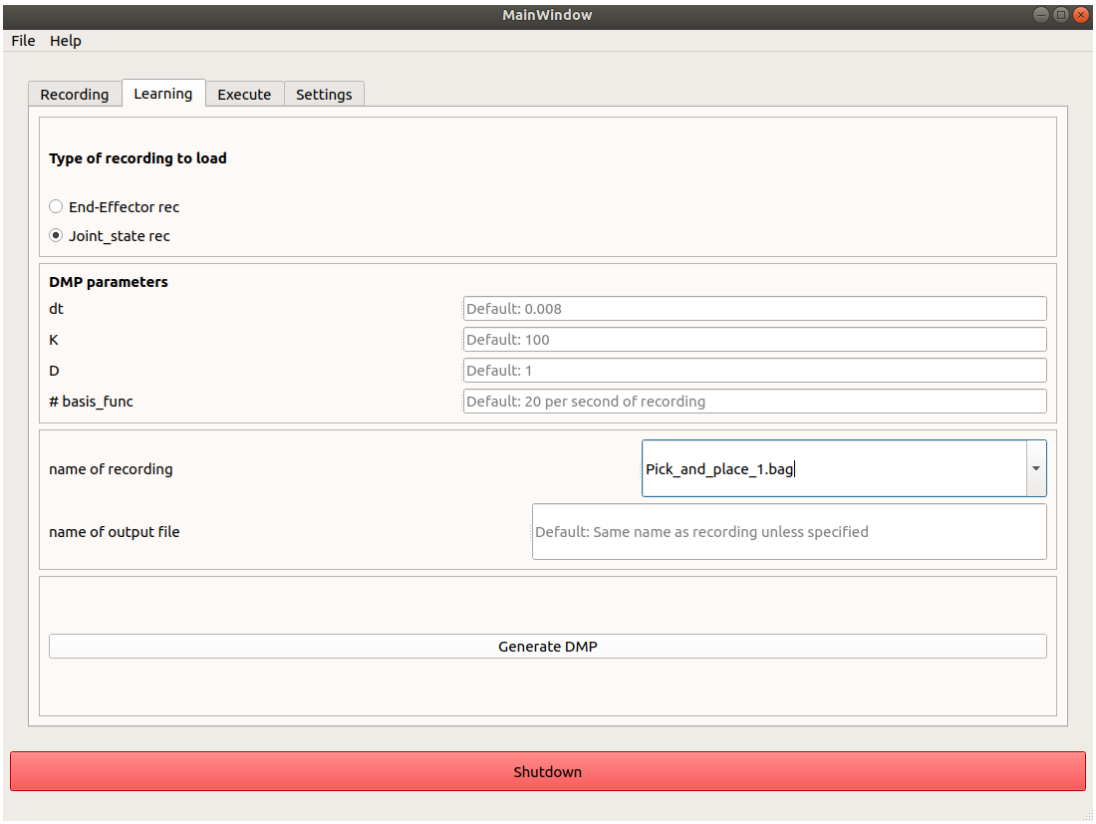


Figure 4.11: Display of the "Learning" tab where users can edit parameters.

MoveIt to retrieve the forward kinematics. However, the Cartesian control is currently not implemented in the application, and the values retrieved are store in variables not used.

For the joint state option, the different values of joint states, in radians, are entered. Either the user can manually enter the desired values, or the user can move the robot to the desired goal or initial position and then retrieve the robot's current joint values. These values are obtained by subscribing to a single jointState message on the */joint_states* topic. However, if the user enters the values manually, the variables are only accepting the `FLOAT` type. If an invalid type is entered, the variable will stay at the previously valid value. The user also has the option to tune the tau (τ) variable. Tau is the time constant for the trajectory. Hence, a higher value will yield a slower speed of the trajectory. The default value is set to one, which will replay the trajectory at the same speed as the demonstrated trajectory.

For the safety of the robot and the human operator, a collision check procedure must be executed. The collision check is performed by requesting the collision check service provided by the MoveIt package. When the user pushes the "Check Collision" button, a set of procedures are performed. Firstly, the trajectory path is derived by using the

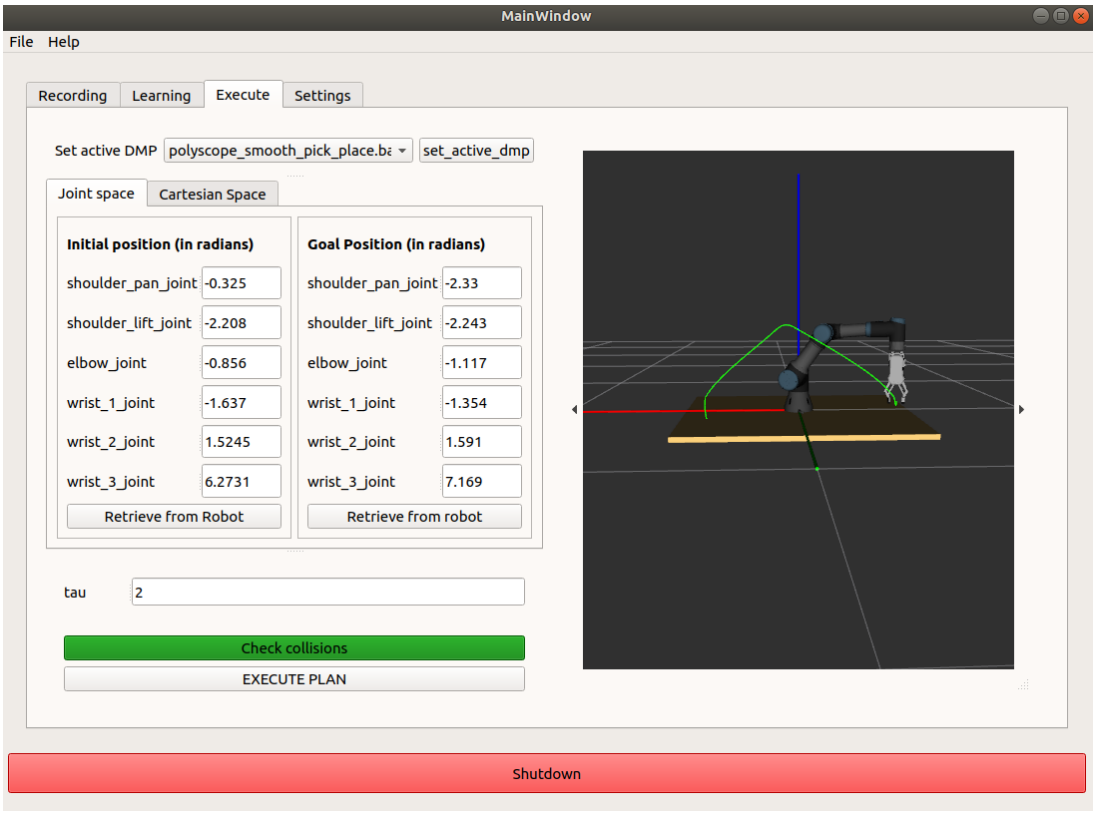


Figure 4.12: Display of the "Execute" tab when retrieving joint space coordinate of the robot. No collision on trajectory is generated.

desired initial and goal position and sending these positions to the DMP server. The response from the DMP server is then a joint trajectory. The returned trajectory is then undergoing a collision check by MoveIt. If the trajectory does not induce any collision, the "Check Collision" button colour changes to green, and the "EXECUTE PLAN" button is enabled. If a collision does occur as shown in Figure 4.13, the "Check Collison" button changes colour to red and the "EXECUTE PLAN" button remains/changes to disabled. After a collision check is performed, the end-effector path is published as a PoseStamped message on the topic /imitated_path.

In order to provide the user with a visual feedback of the new trajectory, a ROS Visualization (RVIZ) [26] plugin is integrated. In this application, only the rendered window (excluding all tools and options) is displayed. In this window, RVIZ extracts the robot model on the ROS parameter server, and RVIZ subscribes to the /imitated_path topic. With these two options, the user will better understand how the new trajectory will look. The RVIZ window is dynamic, so with the mouse-pointer, the user can change the orientation and look at the robot and generated path from different angles. When the user is satisfied with the generated path, and no collision occurs, the user can execute the trajectory by pushing the "EXECUTE PLAN" button.

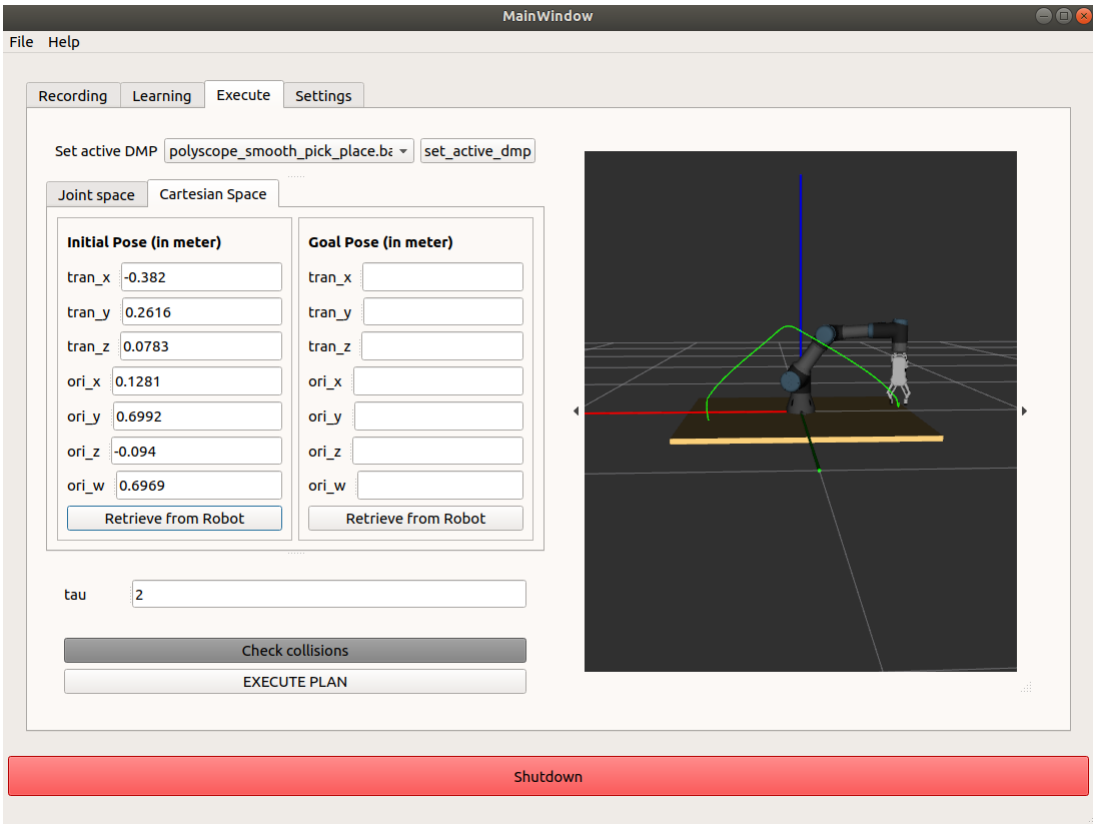


Figure 4.13: Display of the "Execute" tab when retrieving Cartesian coordinates of the robot.

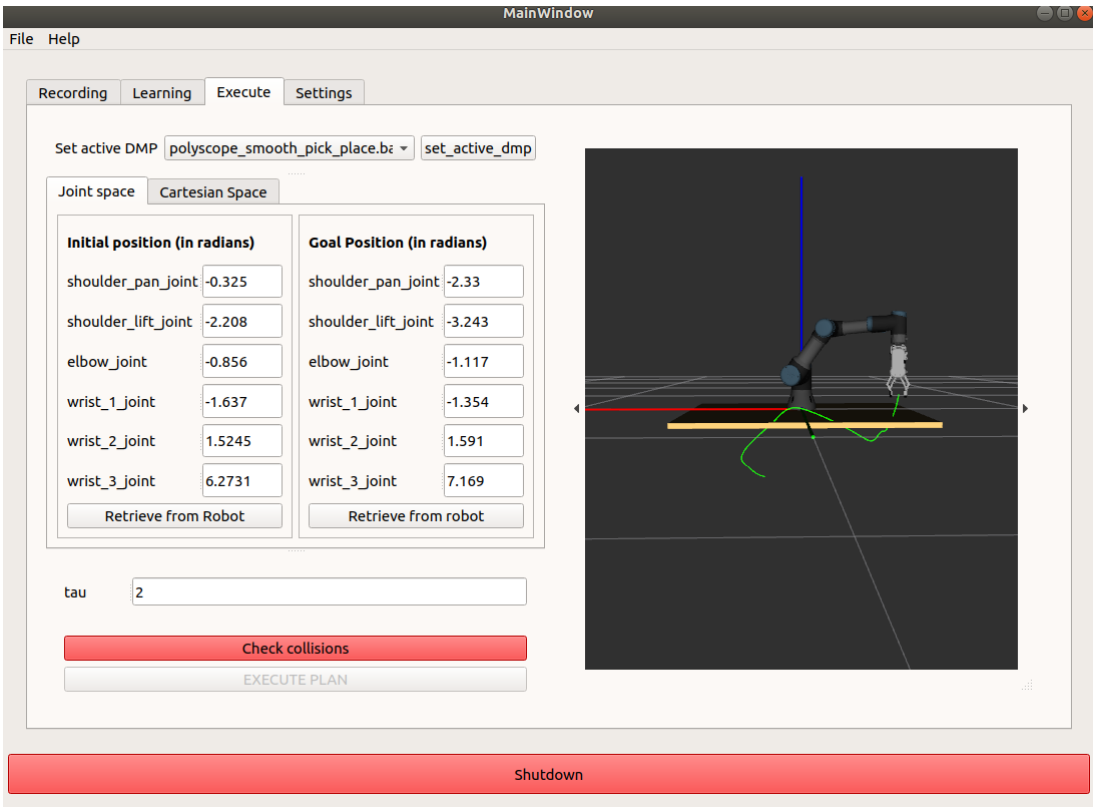


Figure 4.14: Display of the "Execute" tab when retrieving joint space coordinate of the robot. Collision on trajectory is generated.

Chapter 5

Results

In this chapter, the experiments with the corresponding results are explained. The experiments are developed to showcase the effect of the learning method, measure the computational time and check how the GUI reacts to different aspects. The tests in the first Section will analyse the trajectory obtained by the different learning methods discussed in Chapter 2. Next, the second Section focus on experimentation regarding DMP parameters. The experiments on DMP parameters selection are concerned with trajectory shape and computational time. Finally, the last experiment shows how the GUI responds to the different features it provides, mainly with the help of video recordings.

5.1 Evaluation of LfD Methods

In this section, the features of different types of LfD will be explained. The task to be performed is a simple pick and place activity. Moreover, only the trajectory is of concern and not the gripping action of the end-effector. There will be two experiments with kinesthetic reproduction, and one about teleoperation reproduction. The first kinesthetic reproduction is performed with the "freedrive" method available with the UR teach pendant. Freedrive is enabled by holding in a physical button on the teach-pendant down. The second kinesthetic reproduction is also performed with freedrive, but forces required to move the robot in x and z-axis is now null. The third and last method is teleoperation. In the teach-pendant, a short script with a few number of way-points is created. The robot is then performing this trajectory by playing the script.

5.1.1 Kinesthetic Method

As mentioned above, there are two experiments performed with the Kinesthetic method. The first method is performed by using only the freedrive option from the teach-pendant. Freedrive is executed by holding down a physical button on the back of the teach-pendant. While this button is pushed, the robot is enabled to move freely around by external forces. The freedrive of the UR has some resistance, so to move the robot requires some force. Furthermore, when an operator is performing this experiment alone, one hand is already occupied holding the teach-pendant. This results in that there is only one hand left to guide the robot. As it can be seen in Figure 5.1, the end-effector trajectory became very noisy. The end-effector trajectory shows a lot of sharp edges, and are not as smooth as desired. The lack of smoothness is also seen in Figure 5.1, where the joint position profile is shown. When the curve shows an abrupt change, the joint position changes quickly. This yields a large acceleration on the robot joints. The velocity profile indirectly shows the acceleration in Figure 5.3, where an abrupt change in velocity means a large acceleration.

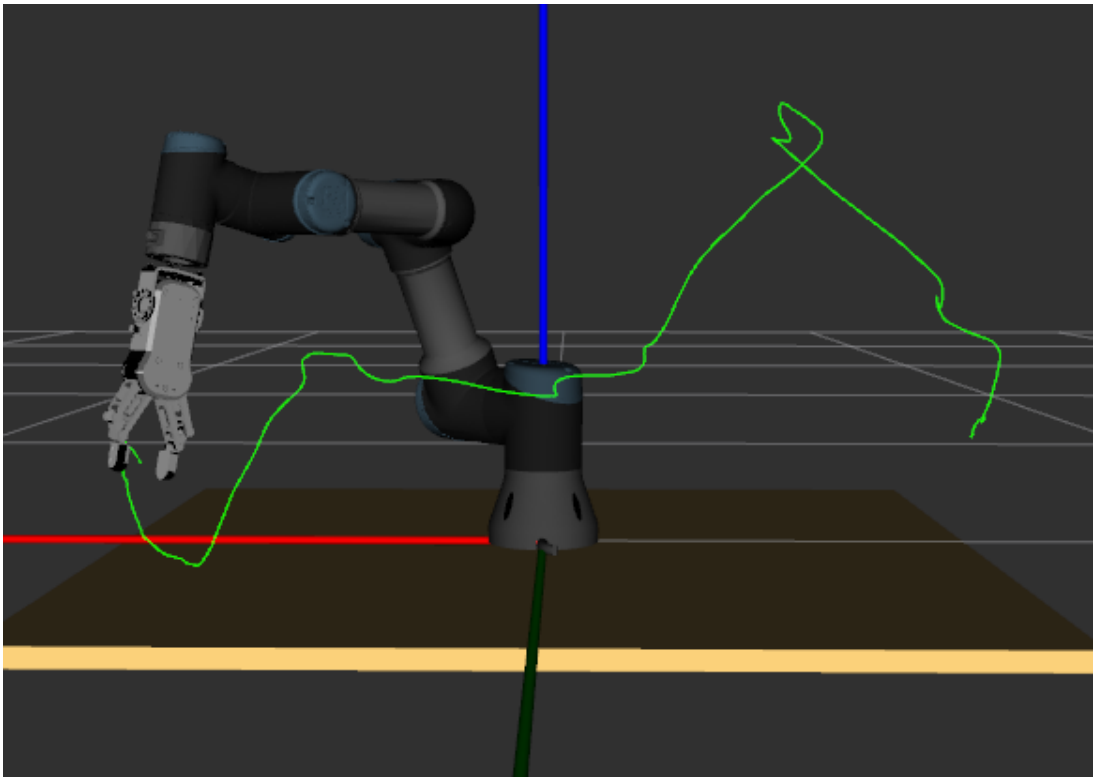


Figure 5.1: End-effector path using the kinesthetic method 1 in the pick and place activity.

Some parameters were changed in the teach-pendant to help with the relatively high resistance when using the freedrive mode. These parameters are about how much force is needed to move the robot in a given direction. When enabling more than two directions, the robot started to shake uncontrollable. So two directions were chosen to require zero

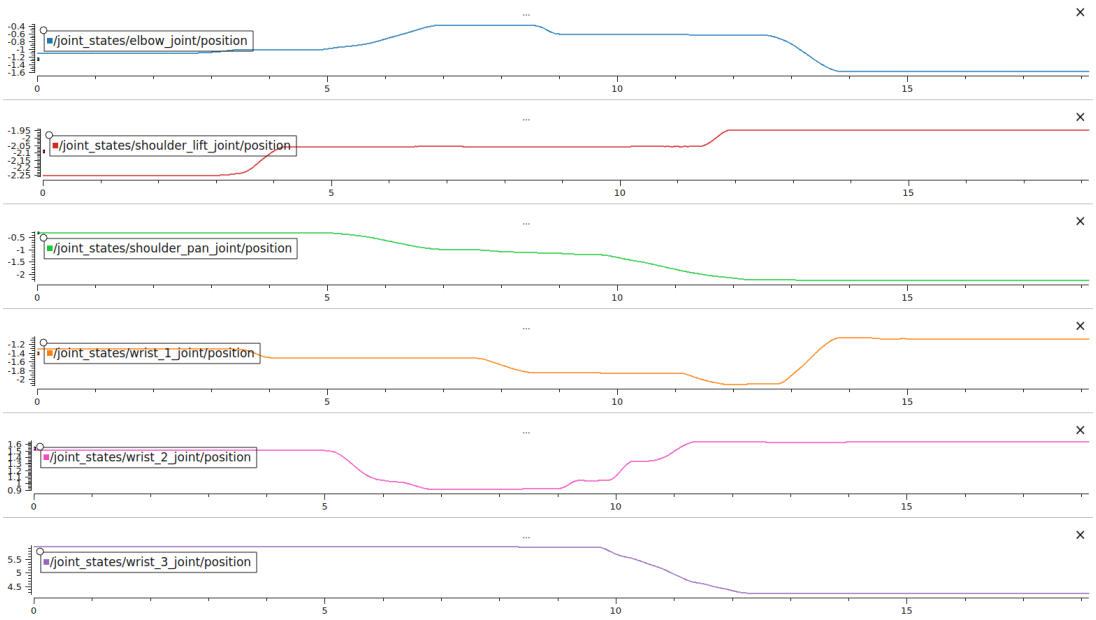


Figure 5.2: Joint positions for the kinesthetic method 1 in the pick and place activity.

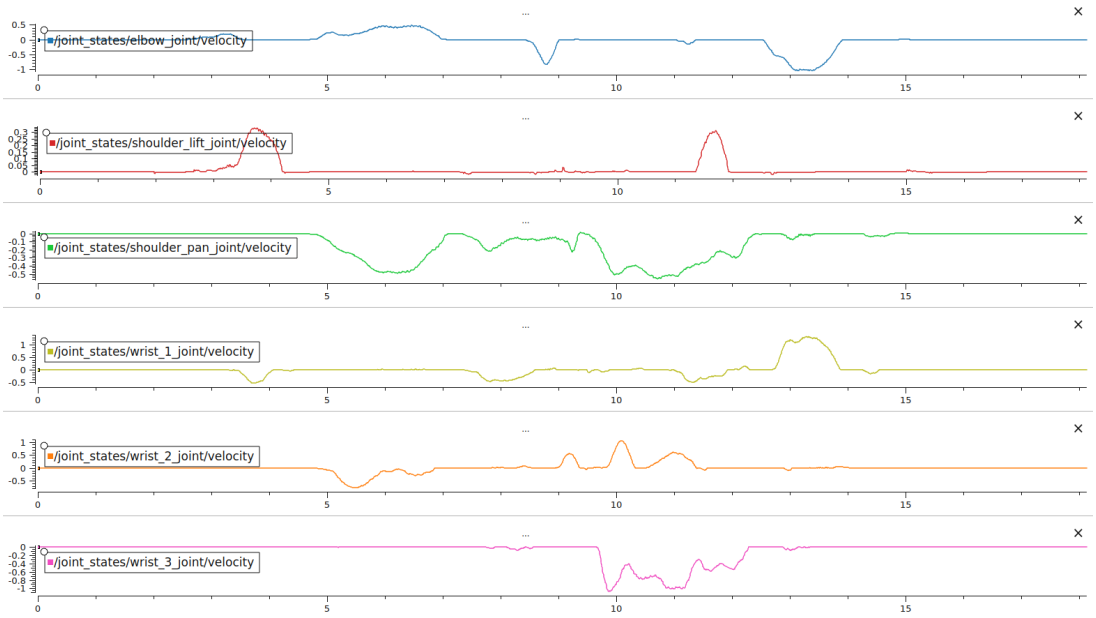


Figure 5.3: Joint velocities for the kinesthetic method 1 in the pick and place activity.

force to move. These directions are the z and y directions. These correspond to the up/down and back/forward direction in Figure 5.4.

As shown in Figure 5.4, the end-effector is much smoother than the one in Figure 5.1. The joint position profile that is displayed in Figure 5.5 shows a more relaxed change in positions during the trajectory. The graph shows an abrupt change for "wrist 1", but the changes in values from -1.5rad to -1.35 is small. The joint velocity profile in Figure 5.6 shows that there are still some noisy changes in the velocity during the trajectory. Given

that two of the directions require almost zero force to move, the movement can be abrupt sometimes. To combat the noisy trajectory, a trajectory is performed by teleoperation. This is explained more in the next section.

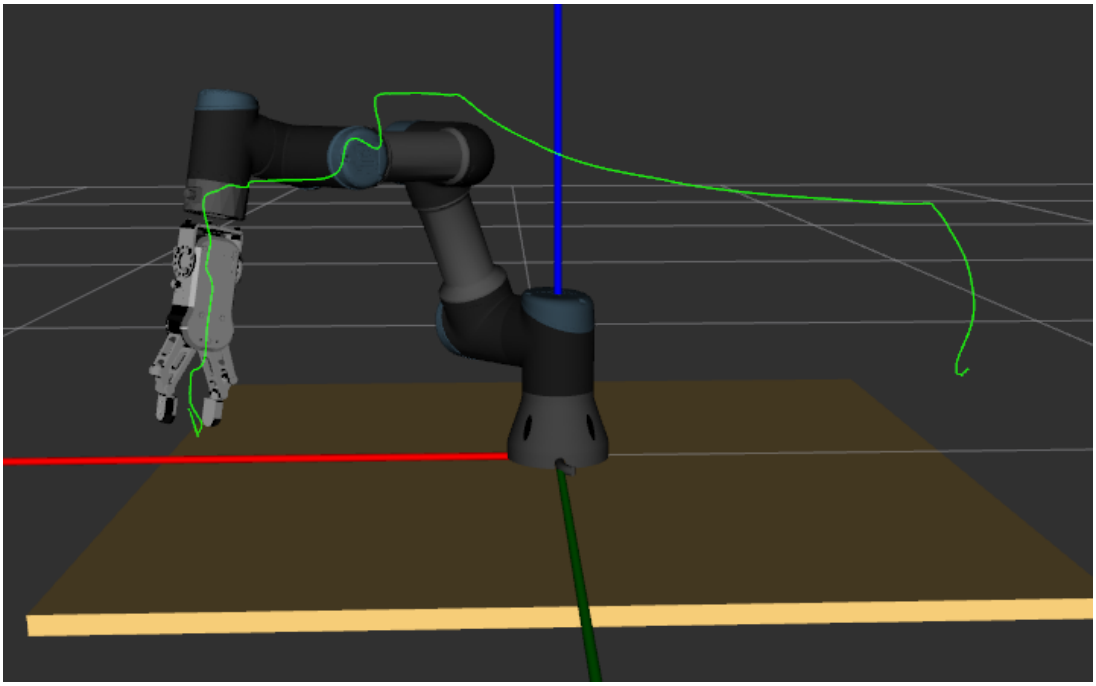


Figure 5.4: End-effector path for the kinesthetic method 2 in the pick and place activity.

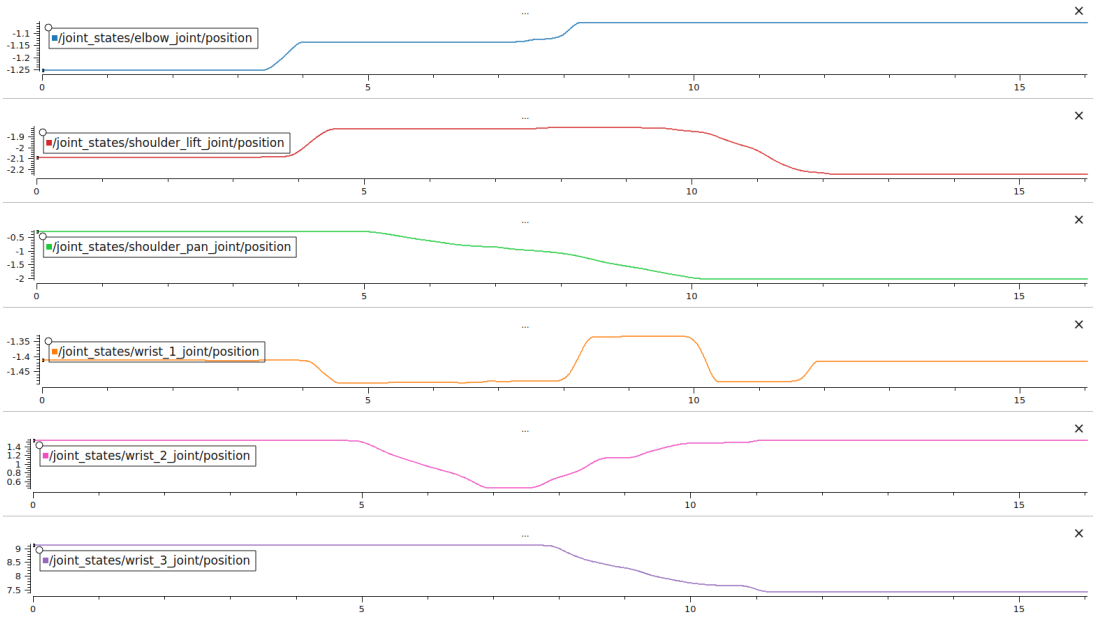


Figure 5.5: Joint positions for the kinesthetic method 2 in the pick and place activity.

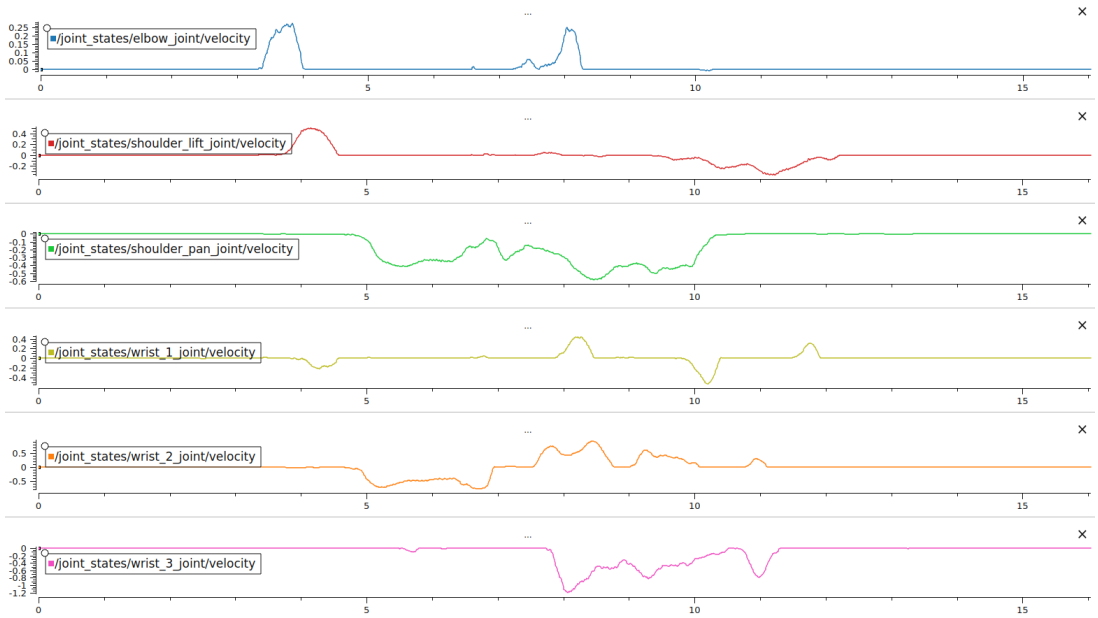


Figure 5.6: Joint velocities for the kinesthetic method 2 in the pick and place activity.

5.1.2 Teleoperation Method

The teleoperation is performed by first creating a script with a number of way-points, then replaying it on the robot. The script is made up of 5 way-points. These way-points are “start”, “upper start”, “high-midpoint”, “upper-goal” and “goal”. The way-points are saved by moving the robot to the desired position and storing the current robot configuration to the corresponding way-point. The movement between way-points is performed by the UR call “moveJ”. These movements are non-linear and do not need to follow a strict line between the way-points. This results in more relaxed changes in the robot joints during the trajectory. When the script is created, the script can then be played back on the robot.

Figure 5.7 shows how the demonstrated trajectory follows a smooth curve from start to goal position. The joint positions displayed in Figure 5.8 show that joints do not impose abrupt changes during the trajectory. This is further explained by the joint-velocity profile shown in Figure 5.9. In this profile, the joint-velocities are shown to increase and decrease steadily with no sudden changes.

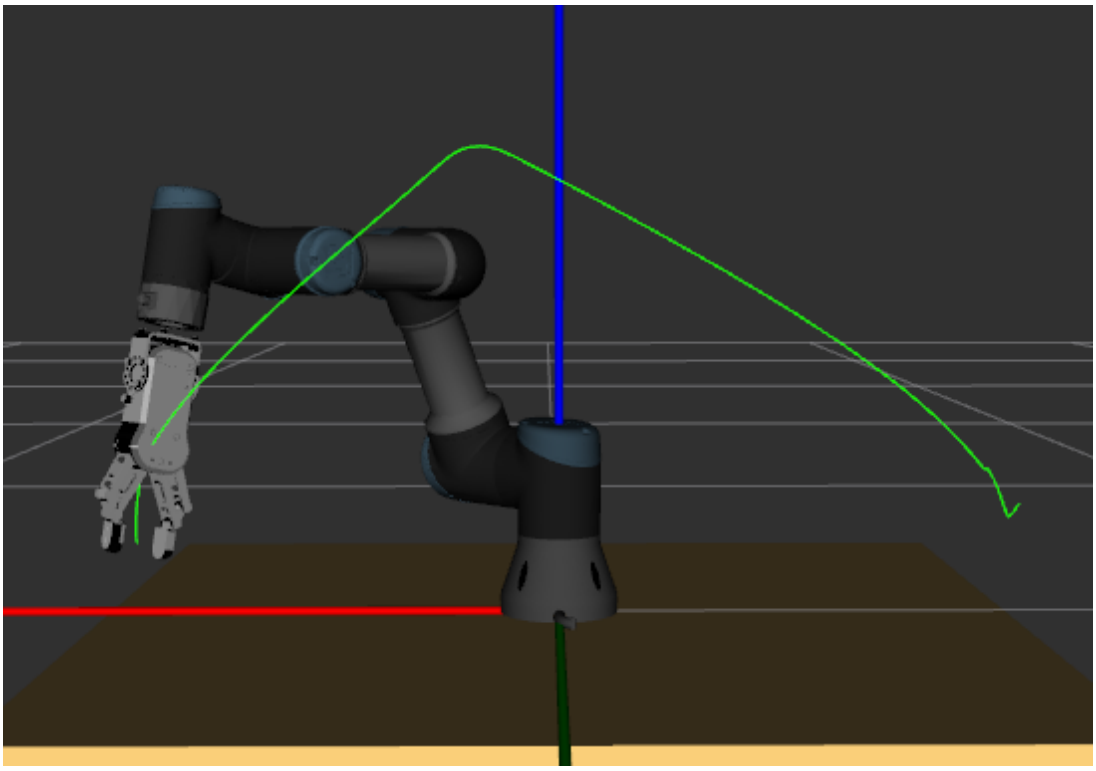


Figure 5.7: End-effector path for the teleoperation method in the pick and place activity.

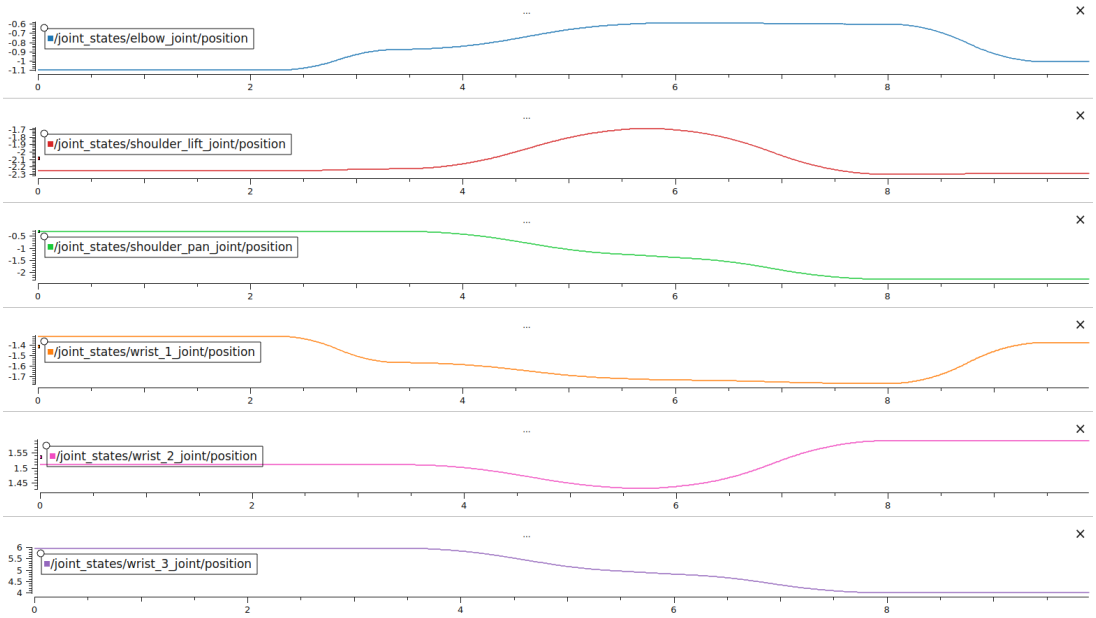


Figure 5.8: Joint positions for the teleoperation method in the pick and place activity.

5.1.3 Discussion

When we first compare shape and smoothness of the generated trajectory, the teleoperated script shows the best results. Smoothness feature makes sense since the teleoperated script uses the precise robot actuators to follow a predefined trajectory. Whereas the kinesthetic

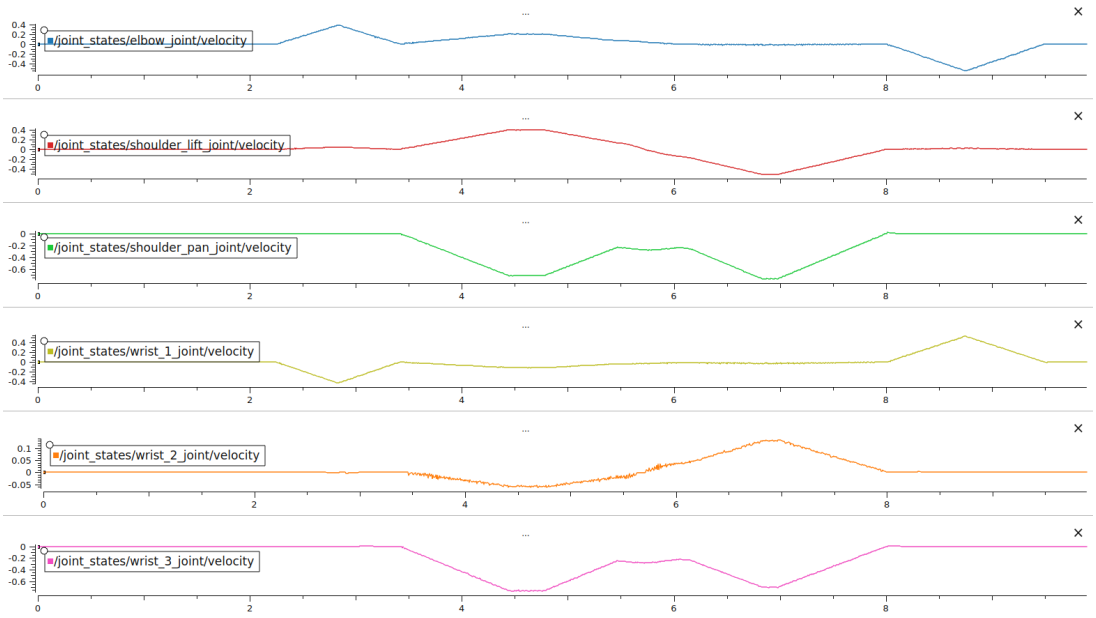


Figure 5.9: Joint velocities for the teleoperation method in the pick and place activity.

methods rely on the precision of the human movement. The freedrive mode of the UR presents a relatively large resistance when moving the joints. By considering that one hand is occupied in this methodology, enabling the freedrive on the teach-pendant, moving along a smooth trajectory becomes difficult. To overcome this problem, one person could hold the teach-pendant while another one moves the robot using both hands. Another possibility is to connect a switch to the I/O panel of the robot to act as an external freedrive button[1].

For the simplicity of demonstrating a trajectory, it comes back to how complex the trajectory is. Only a few way-points must be defined when demonstrating a simple trajectory such as the pick and place activity. A simple trajectory benefits the scripted teleoperation by requiring little previous work. If the trajectory becomes complex, the needed amount of way-points increase a lot. It is making the scripted teleoperation needing much prework before executing the trajectory. It is with these complex trajectories that kinesthetic demonstration shines. It does not rely on prework such as obtaining way-points but are demonstrated directly. The kinesthetic demonstration is though reliant on the teachers' expertise. The teacher should know the trajectory and be able to perform it well to obtain good results.

[1]https://www.universal-robots.com/articles/ur/external-freedrive-button/.

5.2 Evaluation of DMP Execution

In order to perform this evaluation we will analyze two elemeents: firstly, how changes in the DMP parameters affect on their execution. Next, a small study about time consumption is ellaborated.

5.2.1 Changes in DMP Parameters

The DMP parameters to be tuned, as explained in the chapter about resources, are:

- *dt*: Time constant representing the resolution of the plan in seconds.
- *k_gains*: List of proportional gains for each dimension of the DMP.
- *d_gain*: List of damping gains. They should almost always be set to get critically damping, $D = 2 \cdot \sqrt{K}$.
- *num_bases*: Number of basis functions used to approximate the trajectory.

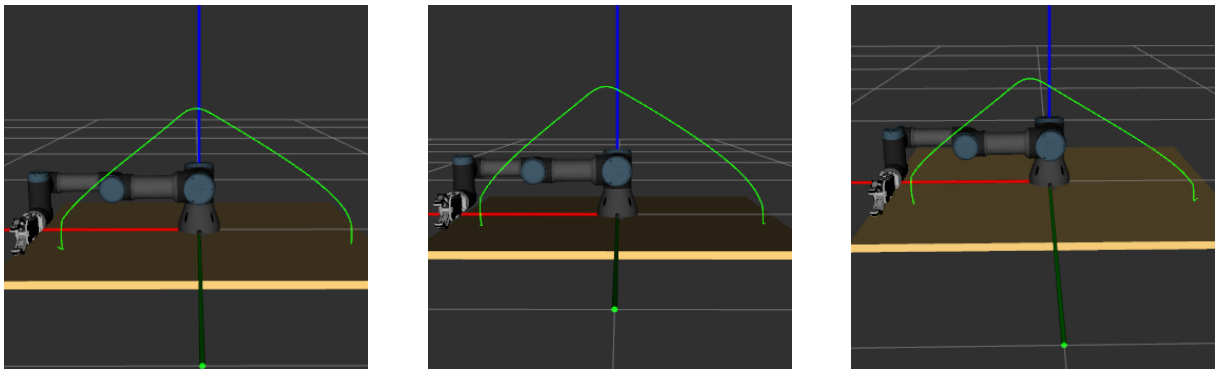
k_gain

For the gain values, the K gain was experimented with, while leaving the *d_gain* critically damped. Table 5.1 shows the correspondence between the K gain value and the number of trajectory points generated when reproducing the movement.

Value k_gain	Trajectory points
1000	1251
100	1251
50	1861
30	2639
10	7509

Table 5.1: Comparison of value of *k_gain* and corresponding number of trajectory points.

Table 5.1 shows that the lower the value of K, the more trajectory points there are. An increase in trajectory points will require a lot more computational time for collision checking. The required computational time is described further down in the next Section.



(a) $K = 10$. **(b)** $K = 30$. **(c)** $K = 100$.

Figure 5.10: Resulting trajectory shape from changes in *k_gain*.

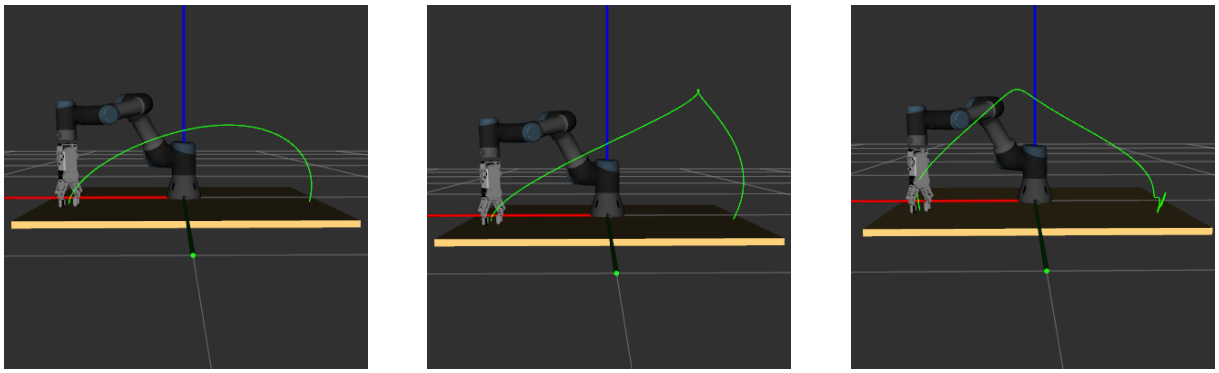
The shape of the generated trajectory, as it can be seen in Figure 5.10 does not change significantly when *k_gain* is modified. With these results, the K value is set to 100 as default, but the user has an option to change.

num_bases

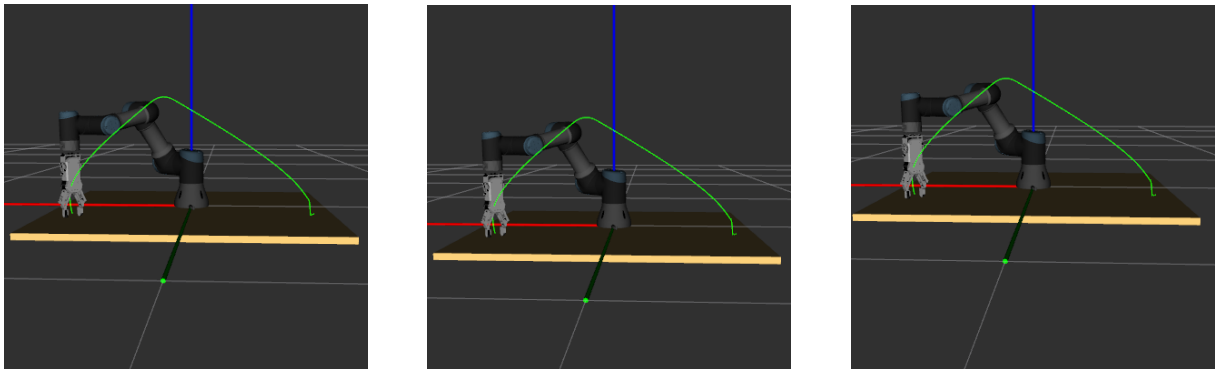
The DMP package from Niekum provides different methods for approximating the trajectory with DMP weights. The default method is a local linear interpolation. However, this method does not take as input basis functions. Another method that is shipped together with the DMP package is an option using univariate Fourier basis. An experiment using different numbers of basis functions for the trajectory associated to the pick and place activity was performed. The resulting shapes of the trajectories are shown in Figure 5.11.

From the image corresponding with *num_bases* equal to zero (see Figure 5.11a), the trajectory goes from start to end without any edges. In Figure 5.11b is displayed the result with *num_bases* = 1. The shape looks more familiar with the "pick and place" shape described previously. When using ten *num_bases* (see Figure 5.11c), the shape starts to get close to the demonstrated trajectory. However, when using 50 *num_bases*, the shape is practically identical to the demonstrated shape (see Figure 5.11b). Finally, an increase to 1000 *num_bases* does not increase the precision of the shape significantly, as it is shown in Figure 5.11b.

The increase in *num_bases* increases the time needed to compute the corresponding DMP weights. With these results, the *num_bases* are set to 50 as the default value.



(a) $num\_bases = 0$. **(b)** $num\_bases = 1$. **(c)** $num\_bases = 10$.



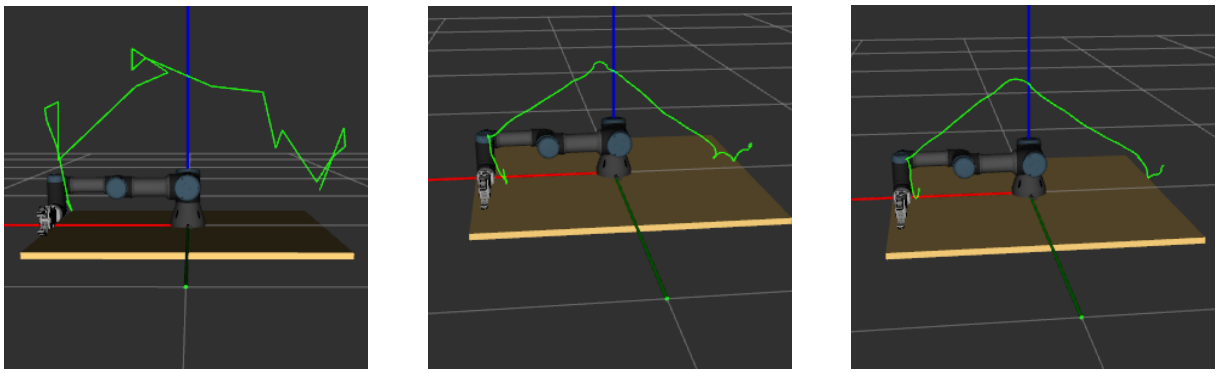
(d) $num\_bases = 50$. **(e)** $num\_bases = 100$. **(f)** $num\_bases = 1000$.

Figure 5.11: Resulting trajectory shape from changes in num_bases.

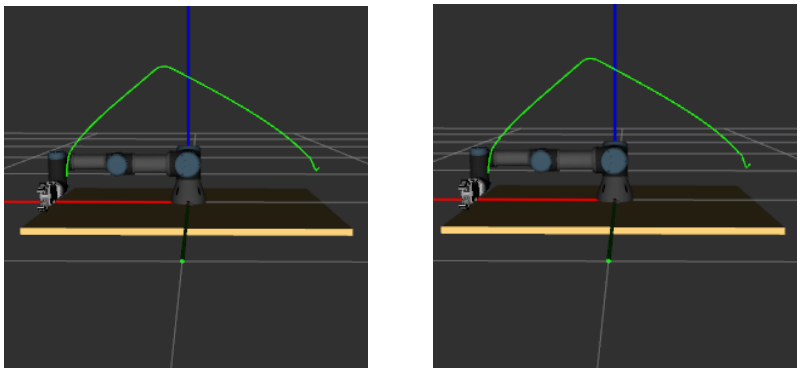
5.2.2 Time Consumption

The dt value corresponds to the time resolution of the trajectory path. A lower value of dt increases the number of trajectory points. In Figure 5.12, shapes generated with different values of dt are shown. The first shape considers a dt value of 0.5. With this value, the shape of the trajectory is edgy, as shown in Figure 5.12a. The next two images are corresponding to a dt value of 0.1 (Figure 5.12b) and 0.05 (Figure 5.12c), respectively. Here, the shape has smoothened out more; however, there are still some sharp edges presented. When looking at the shape using a value of 0.01, the shape looks smooth (see Figure 5.12d). Even with a decrease in dt value to 0.001, the trajectory shape is similar to the previous shape (see Figure 5.12e).

Based on the shape of the trajectory, a lower dt yields better results. However, as explained earlier, a lower value of dt increases the number of trajectory points. An increase in trajectory points has an impact on computation time. An experiment is performed to check time usage. This experiment checks the time the algorithm used when the user pushes the "Check Collision" button. This algorithm retrieves trajectory path, collision check, and does forward kinemathics (FK) on the trajectory to obtain an end-effector



(a) $dt = 0.5$. **(b)** $dt = 0.1$. **(c)** $dt = 0.05$.



(d) $dt = 0.01$. **(e)** $dt = 0.001$.

Figure 5.12: Resulting trajectory shape from changes in dt.

path for visualisation. A comparison between dt value, the number of "trajectory points" and computation time is shown in Table 5.2. When the value of dt gets lower than 0.01, the number of trajectory points increases high enough to impact the computational time significantly.

dt	Points	Path generation	Collision check	FK path	Total
0,5	30	0,00317	0,0320	0,0309	0,0627
0,1	132	0,00442	0,1350	0,1331	0,2597
0,05	285	0,00462	0,2906	0,3253	0,5501
0,01	1001	0,01151	1,0189	1,0210	1,9585
0,005	2000	0,01728	2,0300	2,0390	3,8585
0,001	10001	0,06839	9,9630	10,0500	19,3123

Table 5.2: Time comparison, in seconds, when using diffeerent dt values, for the number of trajectory points, collision check and computing end-effector path.

5.2.3 Discussion

From the experiments performed, some conclusions can be drawn. The parameters of *num_bases* and *dt* have a significant impact on the resulting trajectory shape. Even though K has some influence on the shape, the impact is not significant. For this reason, the default value of K is left to the same value as described in the documentation. When it comes to the computational time of the application, the number of trajectory points significantly impacts. The parameters that alter the number of trajectory points are *dt* and K. The *dt* value has an impact on the shape and number of trajectory points. A higher *dt* value yields fewer trajectory points; however, it reproduces a worse trajectory shape.

With the experiment results, the *dt* value is set to 0,008 (125Hz) as default value. The *num_bases* function is set to 50 as default. These default values serve as guidelines for the user. However, the user can change these values in the application.

5.3 Evaluation of the GUI

To demonstrate the abilities of the graphical user interface, a few video recordings have been performed. The first recording Figure 5.13 shows the process from the starting moment to the execution of a trajectory using the gazebo simulator. This recording starts by opening the application from the Linux terminal. Firstly “roscore” should run in the background. Then in the second terminal window, the application is started. This video is made for showing the process of the application. Therefore the recorded motion is for the robot standing still. The video then proceeds to the learning phase of generating DMP weights. Finally, the video shows the path generation and execution of the trajectory on the simulated robot.

The purpose of the second video Figure 5.14 is to showcase how to use the generated motion library. A set of trajectories has been demonstrated and recorded using the real robot. DMP weights have then been generated from these recordings and are now stored in a motion library. The user can quickly change between different motions.

The third recorded videoFigure 5.15 shows how to change DMP parameters in the application. The stored set of demonstrations are saved in a library. From the application, the desired recording to generate DMP weights can be loaded. Furthermore, the

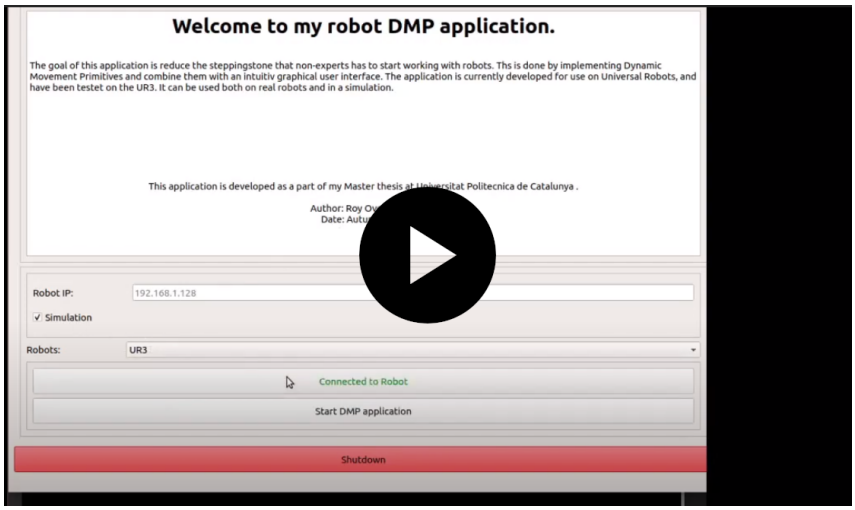


Figure 5.13: Video showing workflow from starting up application to executing trajectory on robot. Link: *https://youtu.be/PuuCp_ny7Ro*.

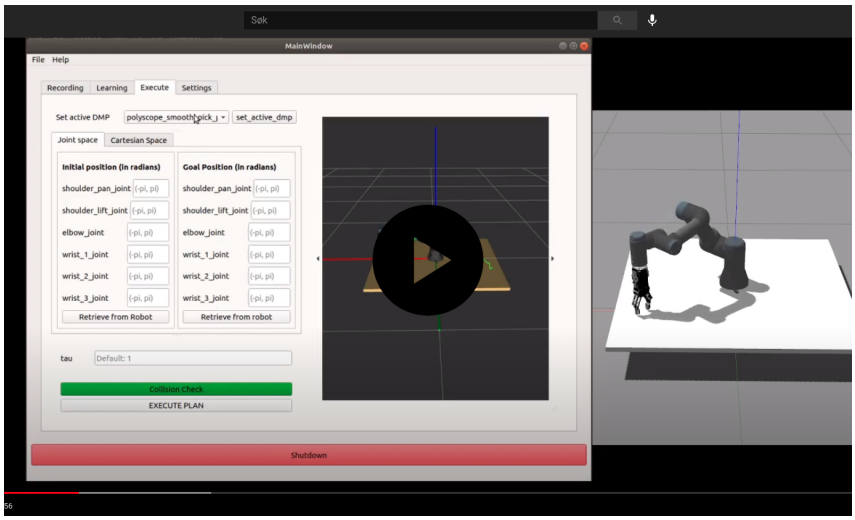


Figure 5.14: Video showing how to change between different motions stored in the motion library. Link: *https://youtu.be/ABXdjRbYoDA*.

parameters for generating DMP weights can be changed. In the video, the parameters *dt*, *k_gain*, *num_bases* and *tau* are experimented with. This video is intended to show the simplicity of changing the parameters, while also displaying the effects from them.

The fourth and last video shown in Figure 5.16 is a recording of using the application with a real robot. Here the scripted trajectory for the "pick and place" activity is recorded. Firstly, in the video, it is shown how to connect the application to the robot. Then the video proceeds to show how to record the trajectory. When the trajectory is recorded, DMP weights are generated from the demonstrated trajectory. Finally, the trajectory is played back on the robot. The video also shows what happens when the robot starts in a position that is leading to collisions.

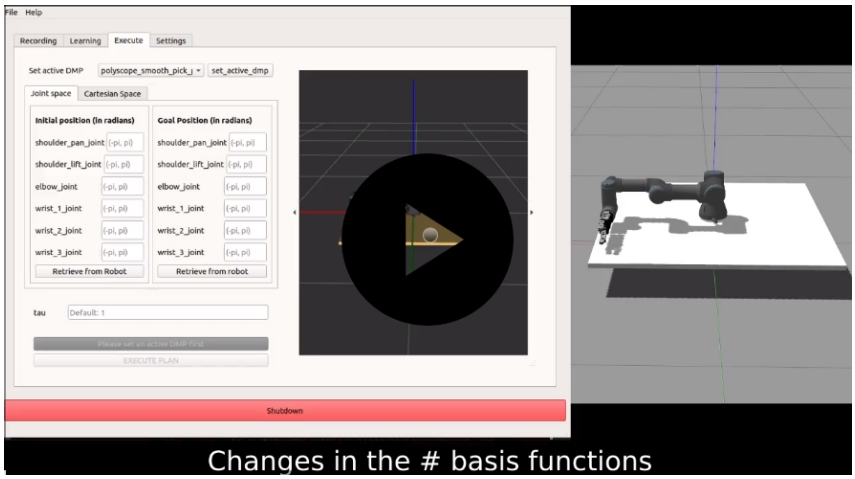


Figure 5.15: Video showing how to change between different parameters when computing DMP weights. The newly computed DMP weights are converted to trajectories and played on the robot. Link: *https://youtu.be/Iqh39EYPuPE*.

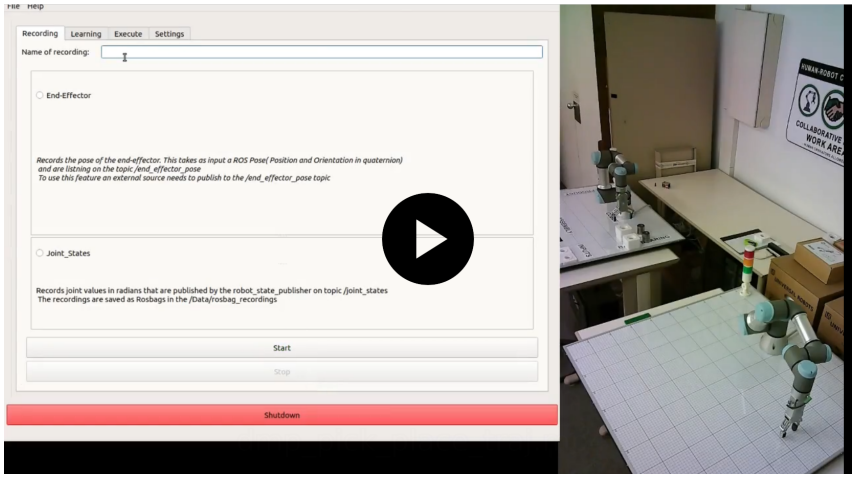


Figure 5.16: Video showing recording and generating DMP from a demonstration. Furthermore, the video shows execution of different trajectories. Link: *https://youtu.be/LuKmQYCKPJg*.

More videos with results are added in the annex.

5.3.1 Discussion

The recorded videos show that the GUI can do all of the features related to the application. They are starting with connecting to the robot and ending up executing the generated trajectory on the robot. The user can record multiple of trajectories, while computing DMP weights of the trajectories. Both, the recorded trajectories and the computed DMP associated weights are stored in a folder on the computer. The storing makes previous recordings and DMP weights available every time the application is started. Currently,

there is no option of deleting recording or DMP weights inside the GUI. These must be deleted locally in the corresponding folder. For the user inputs, a default value is given. These default values are put as placeholder text in each textbox and act as guidelines. The user-input is made fail-safe by discarding all invalid input. There is no error message given if an invalid input is given in the current state of the application. The value of the invalid input will stay at the last valid value.

With the integration of an RVIZ module, the user can have a good idea of how the generated trajectory is. With the visualization of the trajectory, sending a generated trajectory does not induce unwanted surprises. More visual feedback is implemented in all the actions the user can do. Most of the buttons that perform an action or computation give feedback to the user. Either by changing the colour of the button, or giving an explanatory text.

Chapter 6

Impact

In this chapter, the project impact is explained. First, the social impact is described. Here the possible impact of a system such as this can have, especially regarding the small manufacturing industry. Secondly, the impact the project has on the environment is explained, focusing on the greenhouse emission. The last part of this chapter elaborates on the cost concerning the development of the project.

6.1 Social Impact

In Europe, small and medium-sized enterprises (SME) make up 99,8% of all enterprises and two-thirds of the employment according to a report from the EU Commission [27]. Furthermore, with the focus on the manufacturing sector, SME accounts for 58% of employment and contribute 42% of the added value [28]. The benefits of SME is the agile work structure. SME can more quickly change direction than a large enterprise making them faster to adopt new technology. The disadvantages of SME is often the limited economic ability to invest in new technology. Furthermore, high unit production labour cost reduces the profitability of SME.

It is these SME, investments in a cobot can be beneficial. A cobot from Universal Robots cost around €20.000 for the UR3 and up to €45.000 for the UR10. Including a gripper and other tools, a system will cost around €40-50.000 [29]. By introducing a cobot to the manufacturing environment, a repetitive task performed by a human can be replaced by the acquired cobot. The cobot frees up the human resource to work on other,

possibly more complex tasks. A case-story from a Finnish SME [30] with a turnover of 1.3 million euros integrated a cobot in their manufacturing. The integration reduced the unit labour cost for applicable parts of the manufacturing of over 40%. Furthermore, they are estimating a repayment cost of one year for each robot. Another reason they stated for integrating the cobot is the lack of qualified personnel in Finland. Stating that repetitive task such as a CNC operator is not a career the youth want now.

Another environment to use cobots is for older people. According to a UN report [31], they are projecting that by 2050, 1 of 6 people worldwide, up from 1 in 11 in 2019 will be over 60 years old. With the increase in older people, elderly care requires more resources. Here is where a cobot system can help. A cobot can perform a task that requires more strength, such as lifting or carrying heavy things. A cobot can also perform the task that takes a longer time, such as cooking food. However, the current price of a cobot makes this solution not feasible. However, a reduction in cobot prices in the feature can make it more feasible to supplement healthcare works with cobots.

6.2 Environmental Impact

The project developed is a software application that is used in combinations with a real robot. The software will not have a direct impact on the environment. However, computer and robot use electricity. The project is developed in Catalunya in Spain where Red Eléctrica de España (REE) is the electric grid operator. According to an REE press release [32] renewable energy accounts for 43.6% of the electricity generation in Spain for 2020 and are increasing each year. Furthermore, the electricity generation that produces zero CO_2 emission accounts for 66,9% . This means that there is still some CO_2 emission by generating electricity. In 2019, CO_2 emission was estimated to be 241 g/kWh.

Table 6.1 shows the electricity consumed during the project. Because of the COVID situation, much telework was done, exchanging a real robot with a simulator. When the software was developed, the software was tested with the real robot. The project contributed to emitting 21 Kg CO_2. And when the system runs, it emits 43 g CO_2 per hour. The emission is changing, as the electric grid in Spain is changing over to more renewable energy sources.

Device	Consumption (W/h)	Time (h)	Total (kW)
Computer	80	840	67,2
UR3 CB	100	200	20,0
Total consumption	–	–	**87,2**

Table 6.1: Energy consumption of computer and robot during the project.

6.3 Budget

The project is a Master Thesis and is developed at the university campus. Because of this, the project uses already acquired material. However, the depreciation of these materials is calculated to estimate the cost of the project. As this is a final Master thesis of 30 ECTS credits, the student's workload accumulates to 840 hours. In Table 6.2 an overview of the cost is shown. Furthermore, below, the cost correlated with the project is explained.

Cost factor	Fixed (€)	Life (y)	Var (€/h)	Time (h)	Total (€)
Student salary	–	–	15,00	840	12.600
Tutor salary	–	–	30,00	30	900
Computer	1.800	5	0,20	840	168
UR3 CB	23.000	5	2,56	200	511
General consum	–	–	0,02	840	17
UR3 CB consum	–	–	0,025	200	5
Total Cost	–	–	–	–	**14.312**

Table 6.2: Estimated cost of the project. Materials cost are calculated using life expectancy divided on usage, while energy consumption uses energy used multiplied by energy price(0,25€/kW/h)

The personal cost of the project is divided between the student and the tutor. The student has a workload of 840 hours, with a base salary of 15€/h, this accumulates to €12.600. The tutor's workload regarding the project is 30 hours. Using a tutor salary of 30€/h, the cost of the tutor is €900.

For the material cost, both the computer and robot cost are based on the depreciation. A newly acquired computer cost €1.800. Furthermore, with a life expectancy of 5 years or 9000 working hours, the cost contribution the computer has to the project is €168. For the UR3, the same calculations are done. With a purchasing price of €23.000, the robot contributes €111 to the cost of the project. The energy cost is based on the energy consumption of the computer and the robot. Energy cost such as illumination and travelling to the workspace is not included. The energy cost is calculated by using the energy consumed (shown in Table 6.2) and multiply with an electrical price of 0,25€/kW

Chapter 7

Conclusions

The experiments' results show that the application can reproduce demonstrated motions on a real robot by using Dynamic Movement Primitives. The results also show that the GUI is intuitive and straightforward to use.

This project's experiments are designed to evaluate the optimal DMP parameters to use on the UR3 when using the DMP library developed by Scott Niekum. The chosen parameters that are retrieved from the experiments are set as default values. These default values are chosen concerning performance. The performance requirement considers the precision of reproduced trajectory and the computational complexity. These two requirements are polar to each other, so a middle ground between them is chosen. The middle ground makes sure that the trajectory is close in shape to the original one, while the algorithm's need for computational resources is not too high.

One of the goals was to make the developed system intuitive to use for non-robotic experts. Therefore a GUI was developed. When designing the GUI, the simplicity of the end-product was set as a requirement. The developed GUI considers user errors. Such errors could be entering a wrong datatype in input boxes or activating an action before a required prior action is performed. These errors are solved by discarding invalid inputs and disabling push buttons. The GUI also incorporates rendering of the robot with the generate end-effector trajectory. By adding this feature, the user knows how the trajectory will be and does not need to worry if another trajectory is made active before sending the robot's trajectory.

An important factor is safety for the environment and the human. Therefore collision

check is integrated. In this application, no trajectory may be executed without a prior collision check. Currently, the collision checking only accounts for the robot itself and the table it stands on. The environment can be included in the collision check by adding a camera or external sensors and some code configuration modifications.

The final goal was to develop software that is available open-source that implements the project's objectives. Furthermore, with this shorten the path other users has into the robotic world. With the basis of the final goal, the project performs to expectations. The developed application is safe and intuitive to use. Furthermore, it performs well on reproducing demonstrated trajectories. The developed application is available as open-source. Making the application open source and all used libraries are also open source, this should make it simple for other users to use and perhaps develop the application further.

Chapter 8

Future Work

The system is tested and performs well on the real robot and in a simulation environment. However, there are improvements to be done to improve the usefulness of this application. Some of these improvements and added features are explained in this chapter.

The application does accept both joint states and Cartesian coordinates for recording a trajectory. However, only the joint state part is implemented in the trajectory execution. A module that performs IK on the Cartesian trajectory could be implemented as a solution. The control structure currently implemented uses a "followJointTrajectory" action. This action uses input a list of joint positions and the corresponding time the robot should be at that given joint position. If the robot collides with an object and exerts a too high force, the safety mechanism in the UR3 stops the robot. This results in the trajectory, not finishing, and the robot must be enabled again. With a compliance controller implemented, the robot should not stop, and instead, continue the trajectory towards the goal position when the external force is gone. The application currently does not send gripper commands to the robot tool. To do a more advanced task than move the robot, this should be implemented.

Currently, the algorithm only does collision checking of the robot itself and the environment predefined in the URDF. With an external sensor such as Force/Torque sensor (implemented in the new UR e-series) or a camera, the collision checking can be performed on dynamic objects. Furthermore, with the implementation of a camera, the algorithm can check if the goal position has changed. If the position has changed, a control structure that recomputes the trajectory, based on the new goal position could be added.

Lastly, motion planning in the GUI only accepts one trajectory at the time. It would be an exciting feature to set up a list of desired motions to be performed in series. This will enable the construction of complex trajectories that include gripper commands during steps in the whole trajectory. Since a large part of the planning process is collision checking and FK on the trajectory, some improvements could be made. A solution on this could be a more effective algorithm on collision checking and FK.

Chapter 9

Annex

9.1 Videos

Some more videos of the application used on real robot

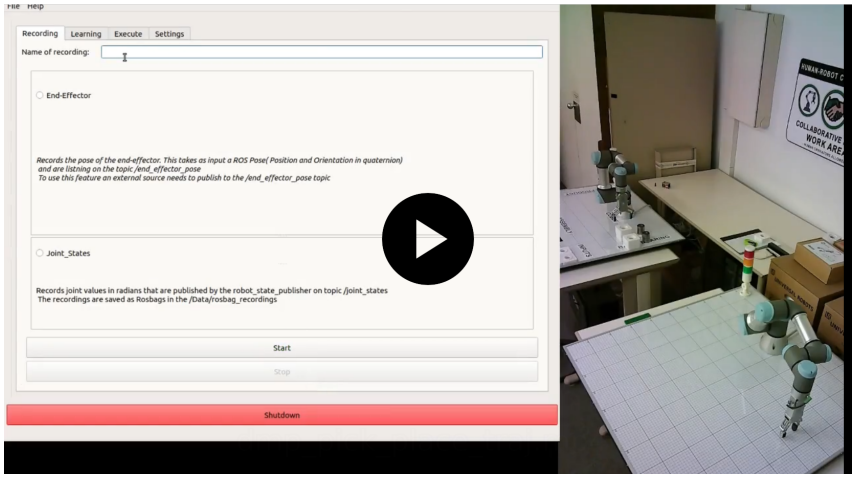


Figure 9.1: Video of kinesthetic demonstration with real robot

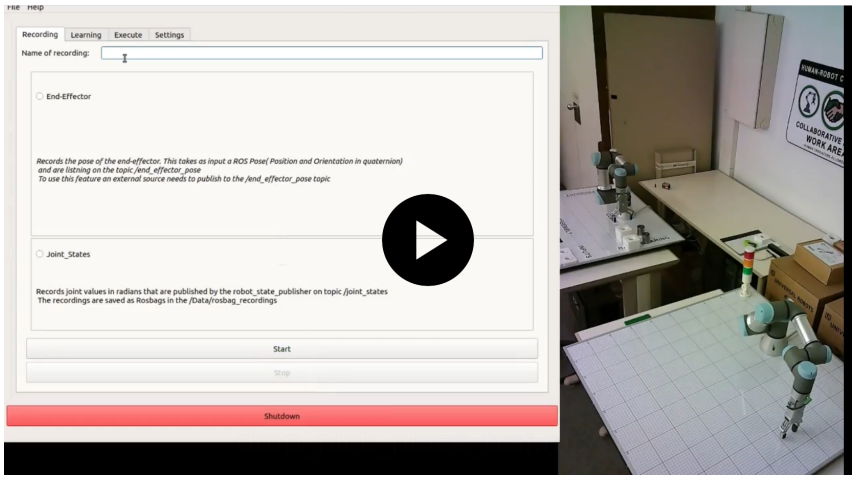


Figure 9.2: Video of comparison between kinesthetic and teleoperated demonstration with real robot.

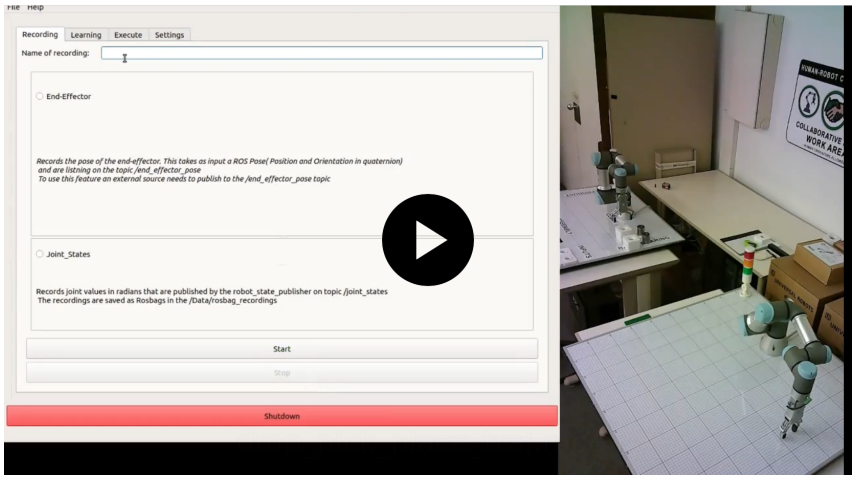


Figure 9.3: Video of teleoperated demonstrated with real robot.

9.2 UML Diagram of Code

The UML diagram of the core of the code containg the recording, learning and execution class together with the GUI class.

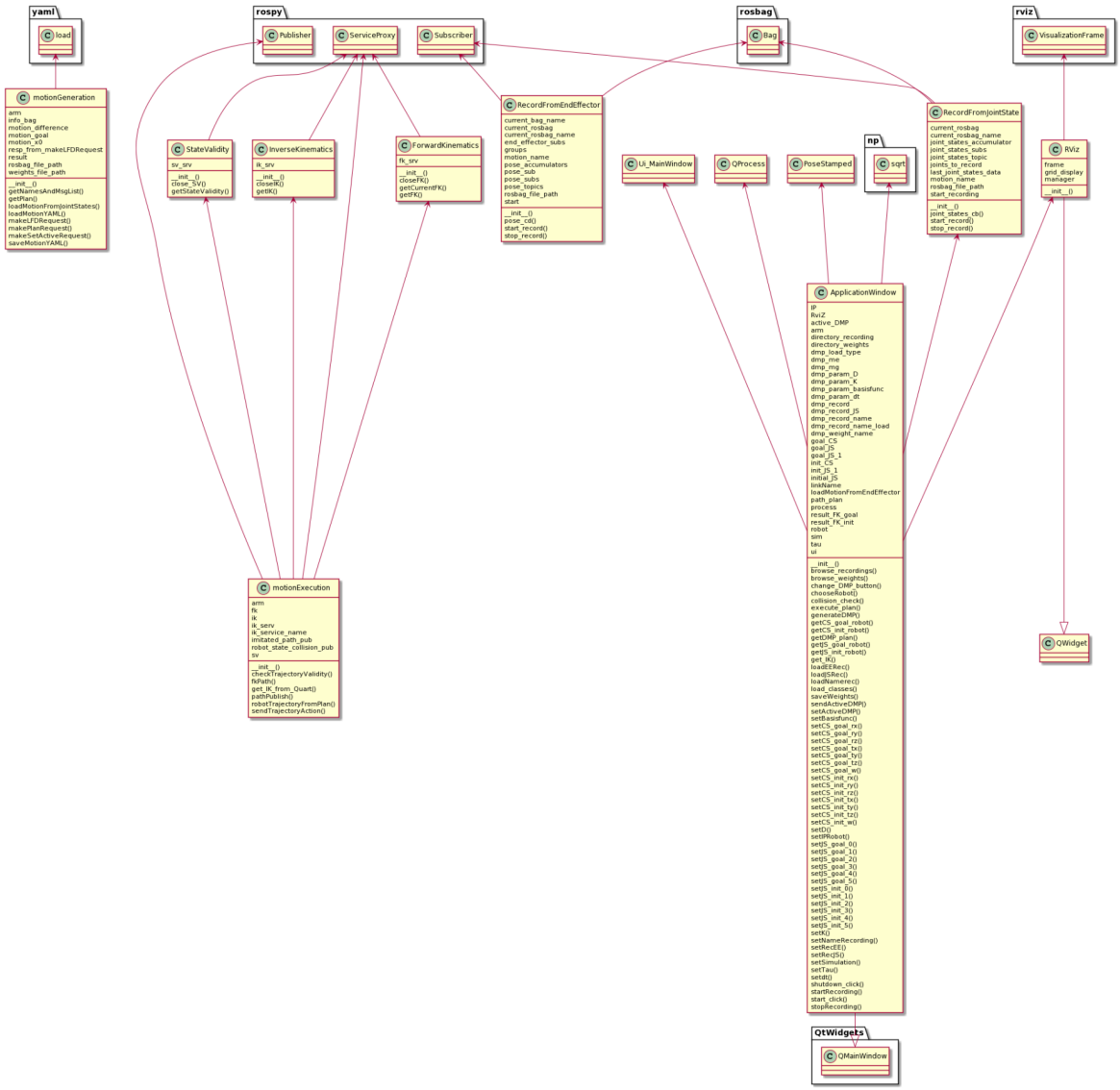


Figure 9.4: UML diagram of the three core classes and the GUI class.

9.3 Code

The developed code is available in github at: *https://github.com/eriksenroy/roy_dmp*

The project package dependencies is found at:

- *https://github.com/UniversalRobots/Universal_Robots_ROS_Driver*
- *https://github.com/sniekum/dmp*

Acknowledgements

I want to thank my tutor Cecilio Angulo for his support and guidance through this master thesis. While also allowing me to work with a real robot.

I will also thank Ester and her family that have welcomed me into their family. I deeply value the support I have received while being far away from my home country during the COVID crisis.

Lastly, I will thank my parents Bjørn-Ove and Toril for the support and guidance I have received when I decided to start up with my education after working for a few years.

References

[1] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Survey: Robot programming by demonstration. In *Handbook of robotics*, chapter 59. MIT Press, Cambridge, MA, 2008.

[2] Staffan Ekvall and Danica Kragic. Learning task models from multiple human demonstrations. In *ROMAN 2006. The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 358–363. IEEE, 2006.

[3] Roswiki.org. `https://www.ros.org/about-ros/`, 2021. Accessed: 2021-01-11.

[4] Ioan A. Sucan and Sachin Chitta. MoveIt package, 2021. URL `http://moveit.ros.org/`.

[5] Universal robots. `https://www.universal-robots.com/`, 2021. Accessed: 2021-01-11.

[6] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Imitation learning of positional and force skills demonstrated via kinesthetic teaching and haptic input. *Advanced Robotics*, 25(5):581–603, 2011.

[7] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5): 469–483, 2009.

[8] Chrystopher L. Nehaniv and Kerstin Dautenhahn, editors. *Imitation and social learning in robots, humans and animals: behavioural, social and communicative dimensions*. Cambridge University Press, 2007. doi: 10.1017/CBO9780511489808.

[9] Alberto Montebelli, Franz Steinmetz, and Ville Kyrki. On handing down our tools to robots: Single-phase kinesthetic teaching for dynamic in-contact tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5628–5634. IEEE, 2015.

[10] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine learning*, 84(1-2):171–203, 2011.

[11] Jens Kober, Katharina Mülling, Oliver Krömer, Christoph H Lampert, Bernhard Schölkopf, and Jan Peters. Movement templates for learning of hitting and batting. In *2010 IEEE International Conference on Robotics and Automation*, pages 853–858. IEEE, 2010.

[12] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX*, pages 363–372. Springer, 2006.

[13] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145, 1996.

[14] Polly K. Pook and Dana H. Ballard. Recognizing teleoperated manipulations. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 578–585. IEEE, 1993.

[15] Aleksandar Vakanski, Farrokh Janabi-Sharifi, and Iraj Mantegh. Robotic learning of manipulation tasks from visual perception using a kinect sensor. *International Journal of Machine Learning and Computing*, 4(2):163, 2014.

[16] Jung-Hoon Hwang, Ronald C Arkin, and Dong-Soo Kwon. Mobile robots at your fingertip: Bezier curve on-line trajectory generation for supervisory control. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, volume 2, pages 1444–1449. IEEE, 2003.

[17] Aleš Ude. Trajectory generation from noisy positions of object features for teaching robot paths. *Robotics and Autonomous Systems*, 11(2):113–127, 1993.

[18] Christopher K I. Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2(3). MIT press Cambridge, MA, 2006.

[19] Sylvain Calinon, Florent Guenter, and Aude Billard. On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):286–298, 2007.

[20] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: Learning attractor models for motor behaviors. *Neural Computation*, 25:328 – 373, 2013.

[21] Scott Niekum. dmp ROS wiki, 2021. URL `http://wiki.ros.org/dmp`.

[22] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation*, pages 763–768. IEEE, 2009.

[23] UR & FZI. UR robot ROS driver, 2021. URL `https://github.com/UniversalRobots/Universal_Robots_ROS_Driver`.

[24] Python software foundation. Python official homepage, 2021. URL `https://www.python.org/`.

[25] Clark C. Evans. YAML official website, 2021. URL `http://www.yaml.org/`.

[26] Dave Hershberger, David Gossow, and Josh Faust. rviz ROS wiki, 2021. URL `http://wiki.ros.org/rviz`.

[27] The European Commission. *Annual report on European SMEs 2018/2019*. The European Commission, 2019. URL `https://op.europa.eu/en/publication-detail/-/publication/cadb8188-35b4-11ea-ba6e-01aa75ed71a1/language-en`.

[28] The European Commission. *Annual report on European SMEs 2016/2017*. The European Commission, 2017. URL `https://op.europa.eu/en/publication-detail/-/publication/0b7b64b6-ca80-11e7-8e69-01aa75ed71a1/language-en/format-PDF`.

[29] Price of collaborative robots. `https://blog.robotiq.com/what-is-the-price-of-collaborative-robots`, 2021. Accessed: 2021-01-08.

[30] Case-story of implementing cobot. `https://www.universal-robots.com/case-stories/ket-met-oy/`, 2021. Accessed: 2021-01-08.

[31] United Nations, Department of Economic and Social Affairs. World population prospects 2019: Highlights. `https://population.un.org/wpp/Publications/Files/WPP2019_Highlights.pdf`, January 2021.

[32] Electricity report on spain. `https://www.ree.es/en/press-office/news/press-release/2020/12/renewables-account-43-6-per-cent-electricity-generation-2020-their-highest-share-since-records-began`, 2021. Accessed: 2021-01-08.