Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017

Project Title:	**Machine Learning for Humanoid Robot Programming through Virtual Reality Headsets**
Student:	**Filippo Baldini**
CID:	**00835932**
Course:	**EEE4**
Project Supervisor:	**Professor Y.K. Demiris**
Second Marker:	**Dr K.M. Mikolajczyk**

Abstract

This project concerns the research and design of a learning technique for the reproduction of human-demonstrated tasks by a humanoid robot. Robots are being progressively used to assist or even replace humans in a wide range of jobs, which often requires them to act like a person would do. Learning by demonstration is therefore the most direct and convenient way to train them. In order to fully exploit the human-like nature of the tasks considered here, teleoperation was chosen as the means of demonstration. This was achieved through virtual-reality hardware that allows the demonstrator to fully immerse in the robot's point of view and lead its movements as he would his own. Several demonstrations of a sequence of actions were then recorded, providing data that is broad enough to generalise a task while capturing its key features. These signals were first temporally-aligned, then encoded using a mixture of Gaussian distributions (GMM) to build models of the demonstrated motions. Generalised trajectories are then computed through Gaussian Mixture Regression (GMR) and used on-line in the reproduction phase by computing new, situation-specific trajectories that maximise the similarity between the obtained behaviour and the key features of the task at hand.

Acknowledgements

I would like to express all my gratitude to my final year project supervisor, Professor Yiannis Demiris, for considering me for this work and for all the support that I received during this last year at Imperial College. His advice and guidance have accompanied me throughout this journey, and inspired me to strive for the success of this project.

I would also like to express sincere gratitude to all the Ph.D. students from the Personal Robotics Lab at Imperial College London, for their tips and for their friendliness, which kept me going whenever I was faced with a problem. In particular, I would like to thank Fan Zhang, for his support with all aspects concerning the Baxter robot, and Tobias Fischer for his most valuable advice.

I would like to thank my friends and colleagues at Imperial College London, as well as my Basketball team-mates, who have been an enormous part of my life for the past four years and have put a smile on my face every day for the last few stressful months. Finally, I would like to thank my parents, for their unconditional support and the sacrifices they have made to give me this opportunity and make my academic journey as comfortable as possible.

Abbreviations

- **LbD**: Learning by Demonstration
- **HMM**: Hidden Markov Model
- **EM**: Expectation Maximisation
- **GMM**: Gaussian Mixture Model
- **GMR**: Gaussian Mixture Regression
- **DTW**: Dinamic Time Warping
- **DOF**: Degrees Of Freedom
- **IK**: Inverse Kinematics
- **VR**: Virtual Reality
- **HMD**: Head Mounted Display
- **ROS**: Robot Operating System
- **BIC**: Bayesian Information Criterion
- **AIK**: Akaike Information Criterion

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

Robots are becoming increasingly popular and present within our society. As their applications grow more widespread, non-robotics experts are often exposed to human-robot interaction (HRI). It is therefore desirable for someone with limited technical knowledge to be able to operate a robot, or cooperate with one. This trend is not limited to robotics, but rather has been observed multiple times in the history of technology, with cars, personal computers, mobile phones and many other examples of technologies that transitioned to user-friendly versions as they turned popular.

When it comes to teaching a robot what it needs to do, however, expert knowledge is required to apply the vast majority of teaching techniques to allow the robot to independently determine the next action from the current state of the environment and of the robot itself. This mapping is often referred to as a *policy* in machine learning.

This project explores *Learning by Demonstration* (LbD) as a more natural and less technical method for policy development. With LbD, a robot is able to learn a policy from demonstrations provided by a human, and to reproduce the displayed behaviour without

any need of technical knowledge or programming skills from the human teacher. Furthermore, demonstrations are an intuitive method of communication for humans, are commonly used to teach other humans.

The project is subdivided into two main parts, with relative objectives. The first goal of the project is to design and implement algorithms to allow remote control of humanoid robots through virtual reality headsets. This will allow human users to teleoperate the robot to perform tasks including handling objects and assisting other humans. A second goal of the project involves implementing machine learning algorithms to allow the robot to record human demonstrations, imitate their behaviour and assimilate knowledge to later perform similar tasks independently.

1.2 Project Achievements

The final deliverables of the project can be summarised as such:

- A platform to remotely teleoperate a humanoid robot.
- A system to record human demonstrations and perform Learning by Demonstration.
- A technique to reproduce the observed behaviours in generalised contexts.

The assignment has a strong hardware component. However, all physical components are provided for the development of the project and are available in pre-established forms. The challenge is to combine different pieces of hardware into a functioning system that maximises performance, and my contribution to the project is therefore mostly comprised of software.

The prime pieces of hardware involved are:

- A Robot: Baxter Research Robot;
- A development machine: a laptop running Linux to control the robot;

- A virtual reality headset: an HTC Vive;
- A Windows Machine: to control the VR headset;
- One or more cameras: to provide both visual and depth data from the robot;

1.3 Project Definition

The project can be subdivided into two parts: Teleoperation and Machine Learning.

For the Teleoperation part, the first step is to understand the overall functioning of the Baxter Research Robot, with particular attention to how the robot movements are controlled and how its visual inputs can be accessed. This stage aims at acquiring a good basis of control over the robot which will be crucial in later steps.

Then, I will proceed to gain access to the camera inputs of the chosen robot, and pipeline it to the VR headset. This stage requires knowledge of the HCT Vive, and providing a bridge between the robot's network and the Vive's SDK.

The goal here is to let a user wear the headset to see from the robot's point of view, therefore human head movements must be accurately mirrored by the robot. In order to do so, it is necessary to access information about the position and orientation of the headset, stream it to the robot, and apply control methods to move the appropriate motors accordingly. These first steps will provide the backbone for the teleoperation, by establishing a bi-directional remote communication between the robot and the VR headset.

The Vive also comes with two joysticks, which can be used to control the robot's arms and hands, mirroring the user's movements in a similar way to his head's. All this should be carried forward with the minimum lag, to ensure not only optimal performance, but limit a possible side effect created by the lag in visual responsiveness to head movements: motion sickness [1].

When the implementation of the teleoperation is complete, a system to record the stream of commands used to remotely control the robot will be constructed.

Once this is in place, the project will transition into its second part, involving machine learning. While the first part of the project is mostly well defined (i.e. there is a series of known things to do), the latter is somewhat more open ended. The goals have been defined, but various techniques must be evaluated before and during implementation, and there is room for experimentation.

Two objectives have been laid out for the second part of the project. For the first one, the robot should be able to re-play actions performed by humans. This can start from an identical reproduction of a sequence of commands, but should evolve into tuned adjustments based on the slightly different circumstances from time to time.

A second, more ambitious goal requires generalising the learned actions to entirely new situations. Such behaviour can be applied to tasks such as tiding up a room, cleaning, helping someone dress, or any other task which is narrow enough to have a closed set of specifications to be met, but that needs adaptation when attempted in different environments. Such behaviours must be much more generalized than the initial "action re-play".

This project can have multiple applications. It is primarily a research project, as it will attempt to implement machine learning algorithms in human-robot interaction. However it could also have commercial applications. Customers might want to buy a similar product to remotely help their older relatives with their daily tasks, and some degree of independence of the robot will be more attractive than a simpler remotely-controlled system. Moreover, robotic teleoperation is already utilised in field where humans cannot intervene, for practical reasons (e.g. with tasks involving heavy weights) or safety ones.

1.4 Structure of the Thesis

This thesis consists of background, requirements capture, design, implementation, testing, evaluation, and future work.

The *Background* chapter introduces the humanoid robot used for the project, the main concepts behind teleoperation and the current competition in this field, some examples on virtual reality applications in robotics, and a brief description of the field of machine learning, followed by a detailed account of its sub-field: Learning by Demonstration. For the latter part, the *Background* chapter will provide a basis for all the individual techniques considered for the implementation part.

The *Requirements Capture* describes the key deliverables. This part was included so that a reader always can keep the objectives in mind while reading the remainder of the thesis.

The *Design* and *Implementation* chapters provide a description of all the accomplished components involved with the project, along with justifications for the choices that were made along the way, and comparisons with the possible alternatives. The *Implementation* is subdivided into the two core parts of the project: Teleoperation and Learning by Demonstration.

The *Testing and Results* and *Evaluation* chapters provide a series of experimental results and data obtained after the implementation was completed. These parts also describe some of the choices made to optimize the performance of the imitation on the basis of the obtained results.

Finally, a *Further Work* section lists a few extra implementation steps that would aid the performance of the delivered system, but were not implemented at this stage. A *User Guide* is also provided to help operate the robot, both for teleoperation and learning.

Chapter 2

Background

2.1 Previous Experience

When I selected this project as one of my preferences, I had already had some experience with virtual reality. During my industrial placement with Sony, I worked on a project that involved the creation of a virtual environment for the PlayStationVR platform. The objective of the project was to stream live sports coverage to a playstation, with the intent to re-create a 360-degree environment which simulated the inside of a stadium. The VR headset would allow users to enter this environment and watch a game.

That project had many similarities with the current one. It involved programming through game engines, which is the way the HTC Vive is operated. The HTC Vive uses Unity, while for my previous project I worked with PhyreEngine. However, the two present many similarities and I was already quite familiar with most of the tools used by Unity, which will saved me quite a lot of background research and familiarising with the development tools.

It also involved streaming videos from a camera to be shown inside the VR world. This is exactly what the first part of this project involves. The manner in which this is carried over is very different, but my experience with rendering video frames onto textures to be

used inside the virtual reality certainly proved useful. In particular, I have learned a great deal about the distortion caused by virtual reality (as opposed to watching a video on a 2D screen) and how to cope with it.

2.2 Robot's Choice

Both the iCub and the Baxter were available as platforms for this final year project, and both presented comparative advantages.

As part of the Human Centered Robotics module, I had a chance to work with the iCub, and learn a great deal about its functionalities. This robot is very human-like, making it ideal to work in situations where humans are involved. Working with the iCub I could notice how users tend to be very comfortable with this type of robot, and engage easily with it. Part of this is influenced by the iCub's human-like head and its ability to show facial expressions. Moreover the iCub's head can be tilted to establish eye contact with the user, and its mouth's LEDs can be moved to extablish a more realistic verbal communication. All these aspects are key to human communication, and have been shown to greatly enhance human interaction with machines [2], [3], [4].

However the iCub presents a few issues which made it unsuitable for this project. Its joints are much weaker than Baxter's, and particularly its hand motors are very fragile. iCub can grasp objects, but would not be able to lift heavy items or apply forces, making it unsuitable for many tasks. On the other hand, Baxter has very strong joints and is capable of moving much more quickly and reliably than the iCub, making him ideal even for tasks involving human contact (such as helping someone dress up).

Moreover, Baxter has a much more established user base, making it much more likely to find ready-to-use tools to speed along the early stages. This project was ambitious, and I did not want to risk getting stuck in early issues and having to come up with solutions to basic problems myself. This is something that happened with the iCub (just installing the

software dependencies took two weeks) and I did want this to be a factor slowing me down.

After some background research over Baxter and the way it is operated (which will be discussed more in detail in the following sections) I have opted to use this robot for my project.

2.3 Baxter

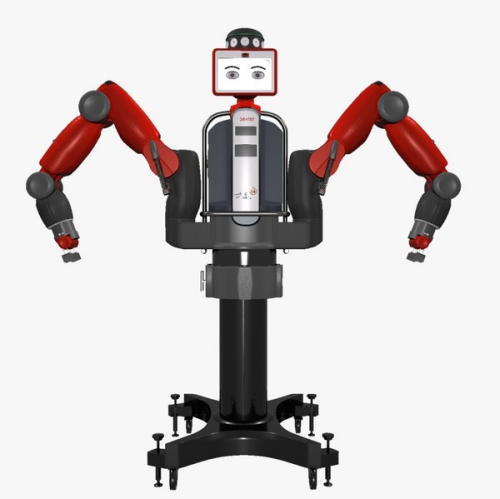

Figure 2.1: An image of Baxter

Baxter was developed by Rethink Robotics for automation applications. It was conceived as a intermediate entity between an automation machine and a human worker, being able to show characteristics of both. Apart from its appearance, shown in Figure 2.1, Baxter has a built-in ability to be trained by demonstration [5], as it can repeat a sequence of joint states recorded in one demonstration. The goals of the project go well beyond this, however this feature proved useful for the early stages of the learning implementation.

Each of Baxter's arms has 7 degrees of freedom, controlled by the joints shown in Figure 2.2. The names displayed can be used in software to set the angle of the corresponding joint, hence determining the arm's movements with high precision. However, for my application I almost never had such direct control over the angle values, but rather made use of the

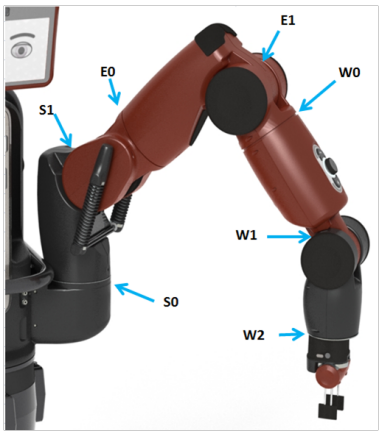

Figure 2.2: Baxter's Joints

pre-implemented inverse kinematics solver, which accepts a (x, y, z) input for the position and a (x, y, z, w) input for the orientation of the robot's hand [6].

Baxter runs on ROS (Robot Operating System) which acts as "middleware" between the software and the robot. ROS creates a network of Nodes, each controlling a particular function or part of the robot. *Nodes* can communicate with each other by sending messages to the network. For this communication to happen, the sender must *advertise* on a given *Topic*, while the receiver must *subscribe* to that topic. This way, no new direct connection must be implemented each time a new node wants to subscribe to a topic [7].

A standalone program can be run on a ROS network, after compiling it into a ROS *package*. These programs can be coded in C++ or Python, both compatible with ROS after installing the appropriate plugins.

2.4 Competition and Teleoperation

As part of my background research, I have looked for possible competitors or similar projects already under development. Robot teleoperation is certainly not a new topic, but I was interested to see how that was implemented and how I could learn from previous attempts. As it turns out, virtual-reality-teleoperated robotics is not well spread. Most teleoperation

applications indeed use 2D screens to display the robot's point of view, and use hardware joysticks to control robotic arms. If Virtual Reality is adopted, it is normally used in conjunction with simulators rather than with the real robot. This is normally done for training purposes before utilising the actual machine.

In the following sections, I will break down the relevant pieces of information that arose from my research trying to capture the pertinent aspects of the different approaches taken in teleoperation and virtual reality applications.

2.4.1 Humanoid Robots

Intelligent humanoid robots are growing increasingly popular, and have a great opportunity to make the jump from science fiction to reality. However humanoid robots first need to develop a series of characteristics that non-humanoid robots could lack, such as human-machine interactions involving touch, gestures and speech.

The rationale behind humanoid robots is their ability to fit into our current living environments, designed for humans. As such, they could become very useful in applications for homes, hospitals or factory floors. A successful development of humanoid robots will therefore prevent the infrastructure changes and corresponding costs which are linked to ensure the correct operation of non-humanoid machines [8].

2.4.2 Teleoperated Humanoid Robots

The first attempt to introduce "intelligent" humanoid robots into our society was done by trying to teleoperate them. This way, the intelligence was still retained by the operator, while other robotic characteristics could be exploited, such as strength, endurance, precision and possibility to work under dangerous conditions.

However, teleoperation of humanoid robots introduces some drawbacks (compared to

non-humanoid robots), such as the struggle to self-balance during locomotion [9]. Since humanoid robots have many DOFs, it is very challenging to control every joint respectively with buttons or joysticks. For this reason, researchers have attempted to develop systems where a human controls a robot that has feedback over its own state, so that the human provides intelligence and the humanoid robot is the one in charge of mobility [10] [11].

Particularly relevant to this project is the work carried forward by Miller, Nathan, et al. from the Korea Institute of Science and Technology [10]. They proposed a system where the humanoid robot tries to imitate a human walking. Using wearable devices, human movements are recorded, and used as control inputs for robotic walking imitation.

For their research, they used a sensor suit to record human walking. The suit was equipped with 16 inertial measurement units (IMUs), having 6-DOF at head, torso and each limb. For this project, the Vive's joysticks and the headset itself are used to record similar data.

Song, H., et al. [12] somewhat narrowed Miller's work by focusing on controlling the robot's arms via wearable devices (much more portable and lighter than an entire suit) and a Bluetooth module. This approach relates much more closely to this project, as it does not involve generating walking patterns for the robot, but only the control of its upper body.

This system implemented sensors which recorded the angles at the shoulder, elbow and wrist of each arm. This information was then transmitted via Bluetooth to the humanoid robot MAHRU [13], which had 6 DOF for each arm. As we have seen, Baxter's arms are controlled by changing the angles on its joints, making Song's approach very relevant to this project. Moreover, their sensors returned voltage measures, which had to be converted (and approximated) to angles. The Vive, instead, will provide very accurate angular measurements.

In their experiments, the slave robot is able to move following the transferred motions, following the trajectories of the desired joint position in real-time, so the robot mirrored the tele-operators movements.

Their main source of errors was that the robot controller module was planning the robot motions based information on angles which were less than the robot's DOF. If there is a mismatch with the robot's structure, the controller needs to provide alternative planning to achieve the desired motion.

Hasunuma, Hitoshi, et al. also attempted to find suitable applications for the humanoid robots, by implementing robots which drove lift trucks [14] and backhoes [15]. Their intent was to design a humanoid robot that could drive a construction machine at a dangerous place and operate it from a remote site.

They argued that a teleoperated humanoid robot able to maneuver machines made for humans would have been easier to carry to a disaster site, and it could have been re-used for different machines, saving quite a lot of effort and investment in designing multiple, very specialised, teleoperated non-humanoid machines.

Their setup consisted of the robot and a remote control cockpit which could send four commands: Biped Walk (not relevant to us), Grasp, Move Arm, and Control Camera. These commands two would accept parameters to attempt the desired movements in a very similar manner to what this project will involve.

Moreover, their control cockpit included a Head-Mounted-Display (HMD) with a head tracker and 3D audio. This was a very similar setup to the one characterised by the HTC Vive.

2.4.3 Teleoperated Baxter

After exploring robot teleoperation in general, some projects were found where Baxter had been teleoperated before.

In [16], Lu and Wen developed a human-robot cooperative control system to aid impaired individuals using Baxter. They developed a system for human-robot cooperation, comninig 3-DOF user inputs and autonomous control to operate the 14-DOF Baxter arms (7 DOF

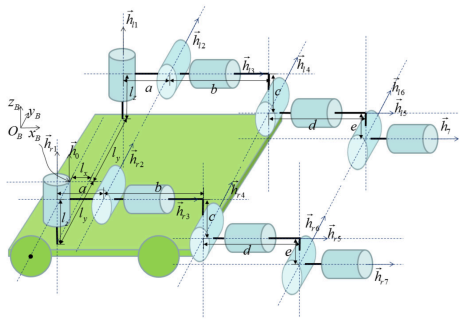

Figure 2.3: Schematic of Baxter's Joints

per arm).

In their work, Lu and Wen study how a 3-DOF input can be best exploited to control a Baxter robot mounted on a wheelchair. Their experiments vary what the three inputs control and test how the autonomous controller acts on the remaining degrees of freedom. In particular, they compare the performance controlling the end effector's velocity, and controlling the orientation of the arms. In our setup, the HTC Vive's controllers should be used to control the end effectors positions and orientation, but it should be possible to gather information about the velocity of the end effectors.

Although a control law must be re-thought given the Vive's higher capabilities than Lu and Wen's limited 3-DOF input system, some common points can be inferred from their work, such as their model for Baxter.

Figure 2.3 shows a simplified schematic of Baxter's joints. The numerical values for the model are:

$$a = 0.069m, b = 0.364m, c = 0.069m, d = 0.374m, e = 0.01m. \tag{2.1}$$

$$[lx, ly, lz] = [0.004, 0.259, 0.390]m. \tag{2.2}$$

The joint angles of the left and right arms are denoted by the vectors ql and qr, respectively, where ql consists of ql1,,ql7 and qr consists of qr1,,qr7.

The following equations describe the relationship between the change in rate of joint angles and the tasks:

$$\begin{bmatrix} \omega_{Tl} \\ V_{Tl} \end{bmatrix} = J_l \left[\dot{q}_{1l}, ..., \dot{q}_{7l} \right]^T$$
$$\begin{bmatrix} \omega_{Tr} \\ V_{Tr} \end{bmatrix} = J_r \left[\dot{q}_{1r}, ..., \dot{q}_{7r} \right]^T \tag{2.3}$$

where J_l and J_l are Jacobian matrices which relate the joint-angle inputs of each arm to the end-effectors angular and linear velocities. ω_{Tl} and ω_{Tr} are the left and right end effectors angular velocities, and V_{Tl} and V_{Tr} are the respective linear velocities.

The control law and the results obtained by Lu and Wen cannot be applied directly without further understanding of how angle and velocity inputs can be drawn from the HTC Vive's joysticks, but their work could turn out very useful to establish a more targeted control law to the presented system.

In [17], Reddivari H. et al. propose another, simpler approach to Baxter teleoperation. They used a Kinect Xbox 360 sensor to detect human motions and mirror it onto the robot. They designed two different strategies, using vectors and inverse kinematics to achieve a given task. Python and rospy scripts were used to calculate the joint angles of the Baxter robot with these two approaches.

Particularly relevant is the inverse kinematic approach, as it attempted to calculate all joint angles from the end effector's position and orientation. This is the situation faced with the Vive's joysticks, which can only provide information about the user's hands.

2.5 Virtual Reality and Robotics

Some applications of virtual reality have been implemented in the robotics field. For example, a company named *219 Design* created a prototype of a system where a virtual model of a

robotic arm is displayed in virtual reality, and a user can control it with the HTC Vive's joysticks [18]. These movements are then mirrored onto the actual robotic arm.

This strategy has the advantage to show the user an actual representation of the robotic arm, making it easy to see what are the constraints of the device and what positions are reachable.

The VR app they developed can also be used to teach the robotic arm. A "record" button was added so that the application will memorize the user's movements and will be able to reproduce them independently. This allows user to teach the robot without needing any programming knowledge, which is very valuable in the commercial applications.

Researchers from the *Computational Interaction and Robotics Laboratory* at John Hopkins University have also developed an application that combines robotics and virtual reality named *IVRE* (Immersive Virtual Reality Environment)[19].

They used an Oculus Rift to display a virtual model of a robot which they could then control (in the simulation or in real time). Their system allows them not only to control the robot's movements, but also to interact with a virtual user interface which allows them to spawn virtual objects, change viewpoint, and program the robot recording trajectories.

Peppoloni, Lorenzo, et al. [20] proposed a system where the user perceives a 3D augmented visual feedback, consisting of the robot's point of view in the remote environment with additional digital elements to help with the task. Moreover, for their experiments they used a Baxter robot.

Their work describes a system consisting of a framework for controlling a robot through a ROS network, with real-time augmented visual feedback. A control unit running on ROS calculates the end effector's pose from the user's hand position and orientation, and accordingly produces appropriate signals for Baxters. The robot's point of view is recorded from a Kinect camera, aumented with task-specific virtual features, and sent th the head-mounted-display for immersive visual feedback.

In order to mirror the operator's motions, noise is first removed from the incoming signals by low-pass filtering, then a linear envelope is computed by rectifying and high-pass filtering the signal. This envelope is then published to a ROS topic at 5Hz.

The augmented reality feedback is used to provide information about the robotic environment, and it's based on object recognition tools.

2.6 Machine Learning

With the term machine learning, computer scientists define algorithms that allow computers to learn autonomously, without being extensively programmed to solve tasks. This field explores the construction of algorithms able to learn from a set of data, making predictions about them [21]. These techniques are particularly useful in fields where a program cannot be specifically coded to solve a too-generalised problem. Examples of this could be tasks involving computer vision [22], game playing, language processing or robot locomotion [23].

For the latter, traditional approaches involve encoding model dynamics, and deriving mathematically-based policies, which are mappings used determine the next action from the current state of the environment and of the robot itself. Despite being theoretically well-founded, the accuracy of the applied models can heavily influence the performance of these approaches. These models require considerable expertise to develop, and often involve linearisation and approximations which can degrade the performance of the system [24].

Different methods like Reinforcement Learning attempt to learn policies through feedback (reward or punishment) upon visiting particular states. While reducing the amount of expertise required to develop a complete system, this approach retains a high degree of difficulty represented by the task of defining an accurate function to provide the reward. Furthermore, building complete policies requires visiting a very high number of states to obtain rewards, which is complex and time consuming [24]. An alternative approach requiring less expertise is therefore desirable, and the field of Learning by Demonstration is among

the most promising.

2.7 Learning by Demonstration

This project's work is mainly concerned with this sub-field of machine learning. The development of behaviours by hand-coding is often very challenging, therefore in fields such as robotics, researchers have been increasingly implementing ways of learning by following a demonstration of the desired behaviour from a teacher. LbD algorithms make use of datasets of examples to derive sequences of tasks to reproduce an observed behaviour.

In [24], Argall, Brenna D., et al. subdivide LbD into two smaller branches: *gathering* the examples, and *deriving* strategies from them.

Gathering is concerned with the construction of a dataset, made of state-action pairs recorded observing a "teacher" (human). The methods of recording vary from application to application, but it normally involves sensors placed on the learning robot. For LbD to be successful, the states and actions must be usable by the "student" (robot), therefore a database must account for the differences (in sensing abilities or mechanics) between the teacher and the student.

Deriving is concerned with the technique implemented to acquire the policy, and with improving the performance beyond the demonstrations. LbD techniques developed three main categories of approaches to derive policies: *Mapping Function* (gathered data maps directly a state to an action), *System Model* (gatehred data is used to determine a model of the required task, in conjuction with a reward function), and *Plans* (gathered data is used to to learn rules about the expected conditions before and after an action occurs).

Human teleoperation has been previously to provide demonstrations, and used for a variety of applications, including flying a robotic helicopter [25] , robot locomotion [26] , object grasping [27], and obstacle avoidance and navigation [28].

In [29], Field, Matthew, et al refer to the strategy of demonstration by teleoperation. In principle, a teleoperated robot could store the given trajectories and merely repeat them. However, with this simple method, small changes in the environmnet like the change of an object's position or the introduction of an obstacle to be avoided are not interpreted correctly, and new demonstrations are needed.

Some solutions involve sub-dividing a trajectory into a succession of key points. This way the database can simply store key points along the trajectory, and the control algorithm will interpolate to create a trajectory that can change every time while being sufficiently similar to the original demonstration.

Statistical models are normally implemented to determine the variations between multiple demonstration, to identify the key trajectory points. However, this method has shown performance drops in cases where the robot had high DOF, such as with humanoid robots.

2.8 Expectation-Maximisation

In machine learning, a maximum likelihood estimation is often used to fit the parameters of a model based on observations. Indeed, maximum likelihood estimation will compute the parameters that maximise the probability that the obtained model will have the observed data as outcome [30].

However, maximum likelihood estimation is sometimes infeasible to solve in closed form, because a given problem is missing a probability distribution from which to derive the log likelihood (more details in Section 2.14, where I discuss Gaussian Mixture Models and latent variables). For such situation, the Expectation-Maximisation (EM) algorithm was developed.

After setting some initial conditions for the parameters to be estimated, the EM algorithm is iterative and it consists of two steps [31]:

1. Expectation (E) step: the current estimate for the parameters are used to estimate an

unknown probability distribution and a corresponding log-likelihood function;

2. Maximisation (M) step: computes the parameters that maximise the log-likelihood found in the E step.

These steps are repeated until convergence. In order to improve the convergence properties of the EM algorithm, the parameters are sometimes initialized using k-means (See Section 2.12) [32].

2.9 Markov Models and Hidden Markov Models

Hidden Markov Models (HMM) are commonly used in Learning by Demonstration as they are a powerful tool to encode a subsequence of observations over time [33]. Moreover, HMM can be used to encode the probabilities to transition from one state to another, making them very suitable for learning how to reproduce the key steps in performing a demonstrated task.

Given a state of states $S = \{s_1, s_2, ..., s_N\}$, we can observe a time sequence of T observations $z = \{z_1, z_2, ..., z_T\}$ where $z_t \in S$. A Markov Model is then defined with two parameters:

- A set of probabilities of starting in state s_i, or priors, often indicated with the symbol π;
- A state transition matrix A, where each element A_{ij} defines the probability to transition from state i to state j.

Hidden Markov Models take this concept further, by considering the problem where the actual state s_i is not directly observable (hence the word hidden), but rather must be inferred from some observable output y_i. HMMs therefore also define a matrix B where each element B_{ij} defines the probability that the hidden state i generated the observed output j.

A Hidden Markov Model is then completely defined by the parameter set:

$$\lambda = (A, B, \pi) \tag{2.4}$$

From a set of temporal observation with known underlying states, algorithms have been defined to estimate the parameters of a HMM that make the observed data most probable (i.e. maximise the likelihood). In particular, a derivation of Expectation Maximisation called the *Baum-Welch* algorithm is commonly applied in HMMs, and was used for my exploration of applications of Hidden Markov Model in Learning by Demonstration.

Human actions are inherently stochastic, as not only they reflect mental states but also often different methods are often applied to solve the same problem. Moreover, patterns can be learned and exploited from human activity, such as we do recognising the same word when uttered by different speakers. To do that, we rely on past experience and compare the heard word to an abstract mental "model" of all the words we know, and probabilistically determine the most similar one.

HMMs are therefore particularly useful to represent human actions, and have been widely used in robotics to learn human skills [34] [35].

2.10 Activity Grammars

As an alternative to Hidden Markov Models, in [36], Lee, Kyuhwa, et al. proposed an approach to imitation learning based on probabilistic activity grammars. The aim was to show that learned grammars can be applied recursively to recognise complicated tasks, which all share similar structures. This research was mostly concerned with teaching robots to understand the goals of what is being demonstrated.

Their approach can be used to learn sequences of large number of actions. For example, tidying up a room requires a sum of lower level trajectories. In similar cases, Neural networks

are often not appropriate, and an approach based on stochastic context-free grammars should be implemented instead, such as in [37].

Activity grammars were eventually not implemented in the presented work. However, they provided an valid alternative throughout the design phase, where different strategies were compared before one was selected. Moreover, the method proposed here generalises actions by recognising their key features, and activity grammars could be used as future work to combine learned activities into a wider knowledge. This way sub-elements of different demonstrated tasks could be combined into new actions.

2.11 Dynamic Time Warping

Humans are very good at visually detecting patterns. Observing multiple time sequences of the same process, a human is able to identify key features even if the signals are not perfectly aligned in time (e.g. if the same signal is delayed by a few seconds), or if the same process was performed at a different speed. Machines however are not very good at pattern recognition, and comparing such signals as they are might cause an algorithm to misinterpret the variance across the observed data.

For this reason, it is often useful to perform some temporal alignment of the data before applying recognition algorithms. In [38], Dynamic Time Warping (DTW) is implemented for speech recognition, a process that involves matching a speech signal to a reference waveform. In order to do that as accurately as possible, DTW is used to stretch and compress the signal in time (in the appropriate portions) to minimize the *distance* between the template and the analysed speech signal.

The mentioned *distance* is defined as the squared difference between the observed sequence s and the template sequence t, which must be minimized by applying a warping path W. This warping path is a sequence of mappings between s_i and t_j, where, for example, w_5 might map s_2 to t_5.

$$\delta_{ij} = (s_i - t_j) \tag{2.5}$$

$$DTW = min_W \sum_i \delta(w_i) \tag{2.6}$$

The cumulative distance is computed recursively as such:

$$\gamma(i,j) = \delta(i,j) + min[\gamma(i-1,j), \gamma(i-1,j-1), \gamma(i,j-1)] \tag{2.7}$$

At the end of the iteration, the optimal warping path can be found by tracing this computation backward from the minimum computed distance [38].

2.12 K-Means

Clustering is a branch of machine learning that is concerned with grouping portions of a dataset into different regions, or clusters, according to some distinguishing features. In particular, K-Means clustering provides a method to cluster the data around a set of k "means", or average between the allocated data.

Achieving a global optimum is a non-trivial task, but k-means offer a fast method to reach a local minimum. This makes k-means an ideal pre-processing tool when used before other learning algorithm such as Expectation-Maximisation. K-means can indeed provide a fast method to initialise the necessary parameters and ensure a better performance of iterative algorithms like EM, which must be given some initial conditions and may converge to different local/global optima depending on them.

K-means is an iterative algorithm involving the following two steps [32]:

1. Initialise k cluster centroids randomly $\mu_1, ..., \mu_k$

2. Repeat until convergence

$c(i) := argmin_j ||x(i) - \mu_j||^2$

$\mu_j := \frac{\sum_{i=1}^{m} 1\{c(i)=j\}x(i)}{\sum_{i=1}^{m} 1\{c(i)=j\}}$

2.13 Gaussian Distributions

Gaussian (or Normal) distributions are very common in probability theory, as they are well suited to represent a variety of normal or social events and behaviours. Its probability density has the shape of a bell curve is entirely characterised with two parameters: a mean μ and a variance σ^2 (where σ is defined as the standard deviation):

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.8}$$

$$X \sim N(\mu, \sigma^2) \tag{2.9}$$

As suggested by its parameters, this distribution has two main features. First, the curve is symmetrical about its mean μ. Second, the variance defines how likely observations are to diverge from this mean, and by how much, such that approximately 68.3% of the observations will fall within one standard deviation from the mean. Figure 2.4 displays a few examples of how the mean and standard deviation affect the probability density of Gaussian distributions.

2.13.1 Multivariate Gaussian Distribution

A Multivariate Gaussian Distribution is a X is derived from the univariate distribution taken to a higher number of dimensions. A multivariate gaussian distribution with mean vector μ and covariance matrix Σ, written as:

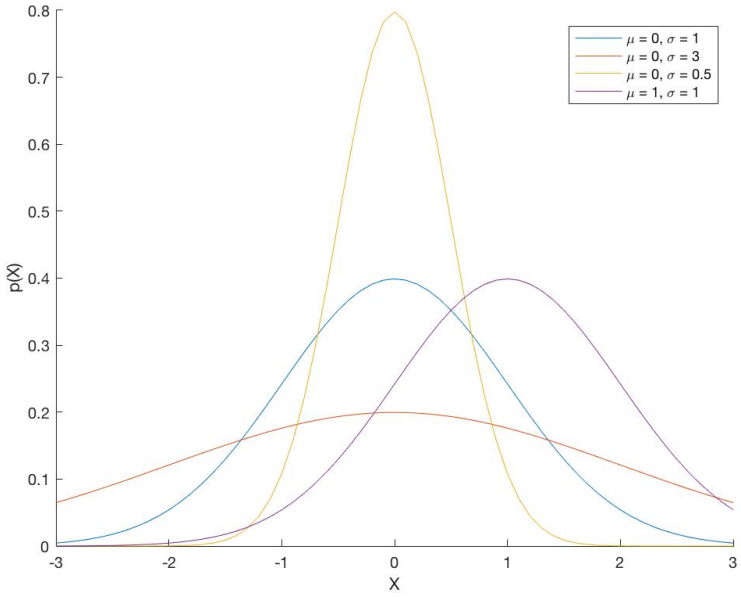

Figure 2.4: Examples of Gaussian Distributions with different mean and standard deviation

$$X \sim N(\mu, \Sigma) \tag{2.10}$$

If x has n dimensions, then $\Sigma \in \mathcal{R}^{n \times n}$ such that:

$$\Sigma := E[(X - \mu x)(X - \mu x)^T] \tag{2.11}$$

It can be shown that Σ is symmetric and that the elements in its diagonal are the variances of the individual dimensions of X. For example, in the bivariate case:

$$\mu = \begin{pmatrix} \mu_X \\ \mu_Y \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{pmatrix} \tag{2.12}$$

where $\rho = Cov(X, Y)/(\sigma_1\sigma_2)$ [39]. The covariance matrix defines the relationship between the dimensions of X, and will be extremely useful for Gaussian Mixture Models and Gaussian Mixture Regression that we will see in later sections, as well as at implementation of a

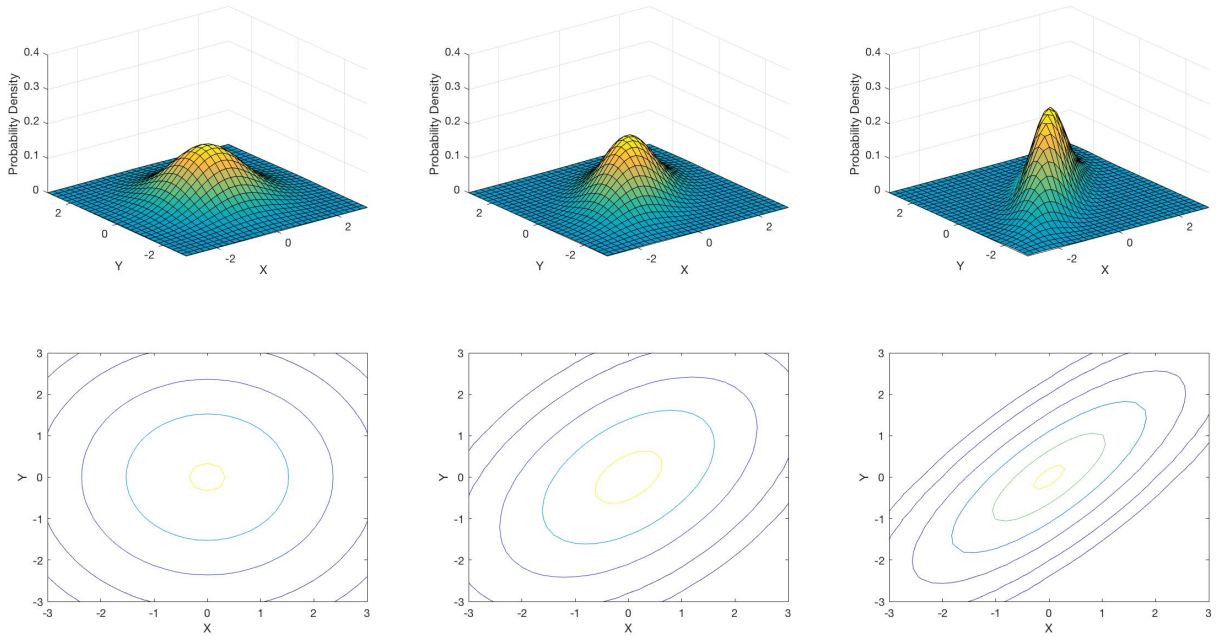

Figure 2.5: Multivariate Gaussian Distributions with varying Σ and their contours

trajectory controller. Hence, it is crucial to point out the properties of this matrix and its effect on the distribution, and the bivariate case can be understood most intuitively.

Figure 2.5 displays three examples of bivariate distributions with mean 0 and respectively $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$, and $\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$. As we increase the off-diagonal entry in Σ, the density becomes more compressed, an effect that is particularly noticeable from the contours.

2.14 Mixture Models

Given a training set of temporal sequences $\{x^{(i)}, ..., x^{(m)}\}$ obtained by demonstrations of a particular task, we wish to find a model that can represent the statistical properties of the entire sequence while allowing local parametrization of the different part of the gesture. This means finding a model that describes how likely it is to observe an outcome given not

only the overall distribution of the data during a demonstration, but rather based on what "portion" of the demonstrated gesture we are currently in.

Mixture Modelling provides a powerful tool to achieve this, as it allows to define a set of K states for a given dataset. Different portions of the data can be associated with a particular state, and a localized probability distribution can be calculated. This method somewhat recalls what we discussed for Hidden Markov Models, and indeed the two are often used in conjunction as in [40], [41], and [42].

In Mixture Models, a latent variable z is introduced, and $Z \sim Multinomial(\pi)$ according to the probability of being in any of the K states (or prior). Then, the probability of our original sequence $x^{(i)}$ is given by:

$$p(x^{(i)}) = p(x^{(i)}|z)p(z) = \sum_{k=1}^{K} p(z = i)p(x^{(i)}|z = k) \tag{2.13}$$

2.14.1 Gaussian Mixture Models

In a Gaussian Mixture Model, the model is constructed with K Gaussian distributions (possibly Multivariate Gaussian distribution, as in my case). The following parameters define a Gaussian mixture model [32]:

- $p(z = k) = \pi_k$;
- μ_k: the mean of the Gaussian distribution of each component k;
- Σ_k: the mean of the Gaussian distribution of each component k
- $p(x^{(i)}|z)p(z) = \mathcal{N}(\mu_k, \Sigma_k)$

Expectation-Maximisation is used to estimate the first three parameters listed above, which are used to model the remaining conditional probability of x given z.

Notably, if Multivariate Gaussian distributions are used in a GMM, the result is a mixture of K *joint* Gaussian distributions.

2.15 Gaussian Mixture Regression

Once a GMM is in place, a regression algorithm can be used to retrieve a path that joins the K Gaussian distributions into a generalised trajectory that goes through the means of the distributions. Such trajectory must follow a path determined by the covariance matrices which, as discussed in Section 2.13.1, provide information about the correlations between variables, hence indicating how multiple variables behave together.

Gaussian Mixture Regression (GMR) [43][44], is a method that can be used to generalise the motion for reproduction. GMR, compared to other regression algorithms, returns a *joint* probability density function of the dimensions of the Multivariate Gaussian distributions obtained with the GMM. Only at a later stage, one can specify which of those dimensions to use as input to retrieve the other (missing) ones [40].

For example, suppose a GMM was used to encode a set of m demonstrations where 8 variables were recorded: a 3D position of a robot's hand, a 4D quaternion indicating the hand's orientation, and a 1D temporal sequence indicating at what time the other variables where in what state. By applying GMR, one can decide to input a new temporal sequence and obtain an expected (i.e. averaged across the m demonstrations) trajectory for the 3D hand position. Similarly, one can decide to generate a 3D trajectory for the hand, and use GMR to output an estimate of how much time has passed. Moreover, one could read a real-time value for the 3D hand position, and use it as input for GMR to estimate what the 4D hand orientation should be, and then use the data to apply the necessary rotations.

This versatility of Gaussian Mixture Regression makes it ideal for testing different ways of implementing the reproduction stage of a demonstrated task.

Figure 2.6 shows the process of obtaining a trajectory from three demonstrations of a

2D path. A 4-state GMM is first generated from the raw data (shown in the middle in the Figure). Here, we can appreciate how the covariance matrices assimilate the relationship between the x and y variables as time progresses. As discussed in Section 2.13.1, the off-diagonal entries of the covariance matrices determine the shape of the ellipses, so that in portions (time intervals) of the trajectories where the pair (x, y) did not change much across demonstrations, a higher off-diagonal entry is obtained in GMM and a stronger relationship is represented by a narrower ellipse. This factor is exploited in GMR, and will also be used when implementing the reproduction of a learned trajectory.

As demonstrated in [43], given a joint Gaussian distribution, the conditional density of a variable X given another variable Y (or more than one) is also Gaussian, and the conditional expectation and covariance are:

$$\hat{\mu_X} = E[X|Y = y] = \mu_X + \Sigma_{XY}\Sigma_Y^{-1}(y - \mu_Y) \tag{2.14}$$

$$\hat{\sigma_X^2} = Var[X|Y = y] = \Sigma_X - \Sigma_{XY}\Sigma_Y^{-1}\Sigma_{XY} \tag{2.15}$$

Since each component k of the GMM presents one such joint distribution, K expected means and variances $\hat{\mu_{kX}}$ and $\hat{\sigma_{kX}^2}$ can be generated as above. To obtain an overall conditional mean and variance, these are mixed in [43] according to the probability w_k that state k contributed to the observed x. Moreover, the concept was extended in [45] to multidimensional x and y so that the full GMR equations are:

$$w_k(x) = \frac{p(x|k)}{\sum_{i=1}^{K} p(x|K = i)} \tag{2.16}$$

$$\hat{\mu_X} = E[X|Y = y] = \sum_{k=1}^{K} w_k \mu_{kX} \tag{2.17}$$

$$\hat{\Sigma_X} = Var[X|Y = y] = \sum_{k=1}^{K} w_k^2 \Sigma_{kX} \tag{2.18}$$

Since the above equations are used to calculate both $\hat{\mu_X}$ and $\hat{\Sigma_X}$ at every time step, the former can be treated as the generalised trajectory (i.e. the mean of the demonstrated trajectories *is* the generalised trajectory) and the latter as the constraints around the trajectory. An example of a regression using the above method is displayed in the lower part of Figure 2.6.

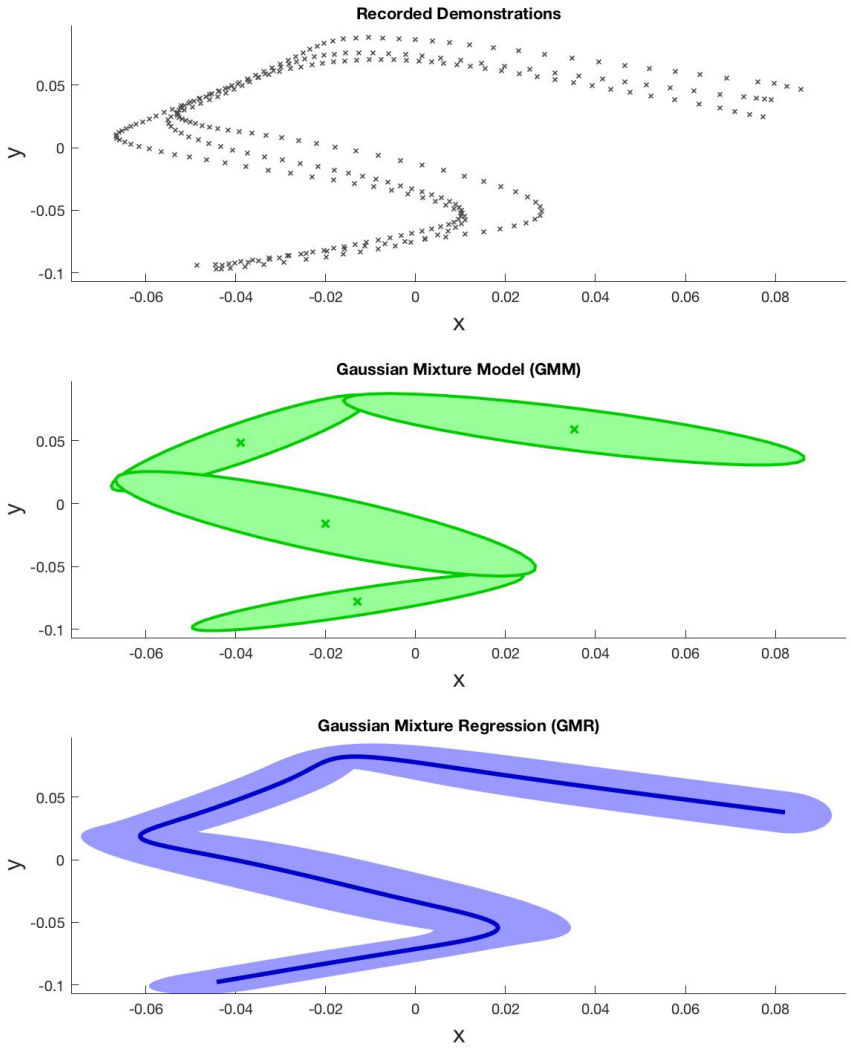

Figure 2.6: Learning process for a 2D trajectory. Top: three demonstrations of the same trajectory. Middle: A GMM of 4 states obtained from the demonstrations, where the ellipses represent the covariance matrices Σ for each gaussian component, centered at the mean μ of each component. Bottom: a generalised trajectory obtained via GMR; the light-blue contour represents the trajectory limits, obtained from the variance in the demonstrations at every (x,y) position. The code to generate these images was obtained from [46][45].

Chapter 3

Requirements Capture

3.1 The Project Deliverable

The objective of the project is to deliver a platform that allows teleoperation of a humanoid robot as means of demonstrating tasks. The robot should then learn these tasks, and reproduce them independently, removing the need of a human operator.

The project proposes using Virtual Reality technology for the teleoperation, as a method to provide immersive teleoperation. The operator should be able to wear a VR gear and embody the robot. The goal is to allow the him to see what the robot sees and to map the operator's movements to the robot's limbs.

The project deliverables can be subdivided into two main parts: Teleoperation and Learning by Demonstration.

3.2 Part 1: Teleoperation

This first phase concerns the delivery of a platform to control a humanoid robot, which ensures reliable communication between the robot side and the virtual reality side, and it can be treated independently from the Learning phase. Indeed, this stage should deliver a system that can be used for a variety of purposes, not limited to Learning by Demonstration.

For this part, the main focus should be maximising user experience by removing lags and addressing issues with the visual feedback.

The following features should be delivered, which represent the backbone of the teleoperation:

- Environment Bridge: interfacing the virtual reality world with the robot's operating system.
- Camera Connection: establishing a connection between the robot's camera and the VR controller.
- VR Display: displaying the camera feed onto the VR headset without distortion. This step involves generating a stereo signal from a mono feed.
- Head and Hands Position Extraction: obtaining information about the spacial position and orientation of the user's head and hands from the headsets and the joysticks.
- Movement Control: designing a control algorithm to best map the information obtained in the previous point to the robot's movements.
- Controller Buttons: allow the user to manage some aspects of the teleoperation via the Vive controller buttons.
- Performance Enhancements: removing any lags, bugs, or any other factor that might limit the usability of such system.

3.3 Part 2: Learning by Demonstration

The second stage of the project will be concerned with machine learning. Here, the steps are less structured than in the previous part, as a great deal of research is required before implementation, and different techniques might involve different courses of action. There are, however, some guidelines, or higher level functions that must be covered:

- Recording demonstration data: most likely as a sequence of state-action pairs determining the desired behaviour. The exact implementation and the choice of data to store will depend on the chosen learning technique.
- Trajectory learning: as partially discussed in Section 2.6, this type of learning can teach the robot to do simple, relatively repetitive actions. Examples could be moving objects or picking up items. Here, the use of neural networks could be explored. This type of learning can deal very well with noise, and allows quick training. Therefore I believe it to be ideal for my first approach to machine learning.
- Robust Learning: depending on the performance of trajectory learning algorithms, I would like to expand my research, perhaps exploring some useful principles such as the use of Probabilistic Activity Grammars in [36] and [37] or Gaussian Mixture Models in [40], [45], and [47].

Chapter 4

Design

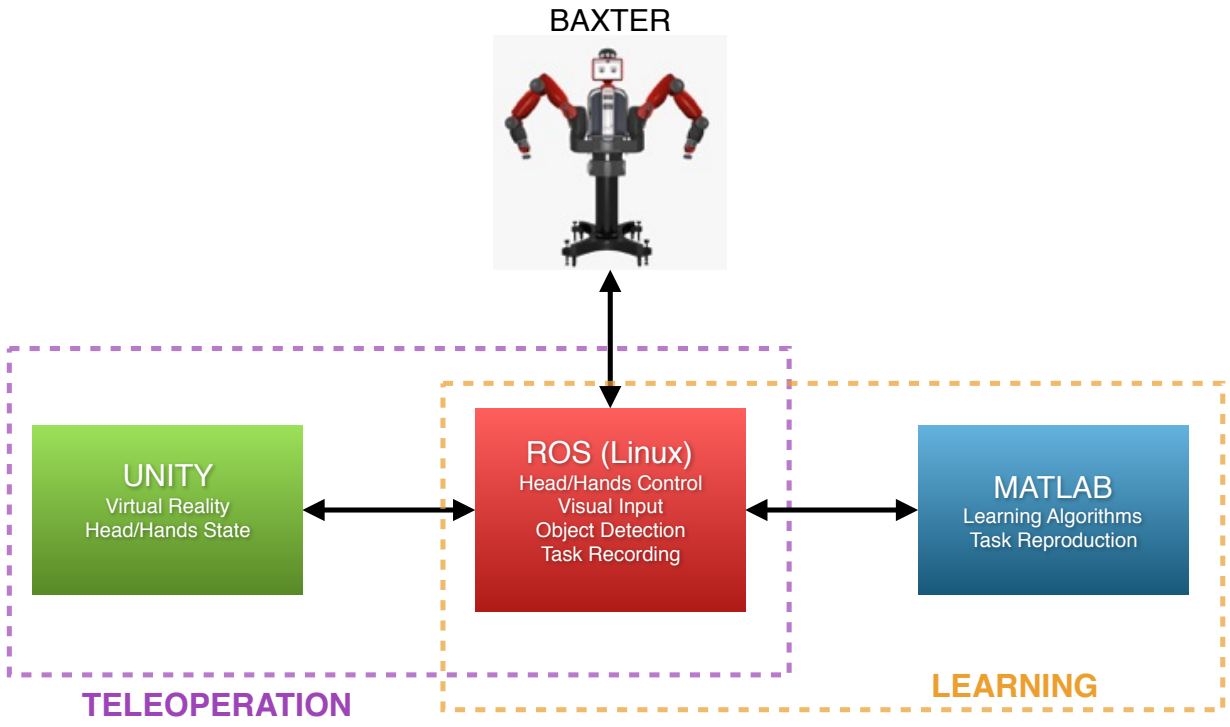

Figure 4.1: Functional breakdown of the project's components. The ROS network, running on a Linux machine, acts as a low-level interface between the robot and the Teleoperation and Learning units, represented respectively by Unity and Matlab

The following chapters will account for the implementation of the two core components of the project: Teleoperation and Learning by Demonstration. Figure 4.1 displays an overview

of the final system as split into its functional components.

The Unity side will take care of the input part of the teleoperation, by recording the operator's movements and displaying what the robot sees onto the Head Mounted Display. Its input is a video stream from a camera on the robot's head, while its outputs represent the current states of the teleoperator's head, arms and and hands, as well as a list of parameters for the teleoperation.

The Matlab side will perform the probabilistic analysis involved in learning and reproduction of the demonstrated task. Its inputs are represented by a series of demonstrations for a specific task, as well as a series of robot current state readings during task reproduction, while its outputs will be in the form of direct commands for the robot's movement to reproduce the task.

The Linux side will instead take a lower-level control of the robot itself, serving as a bridge between the other components and Baxter.

Figure 4.2 displays a more comprehensive overview of the implemented system. All the major components of Teleoperation and Learning are shown here, and their relationship with each other is indicated by the arrows.

ROS nodes and topics (shown in red) perform the following:

- Stream visual content over to the Unity side from an Asus Xtion Camera.
- Convert both the teleoperator state and the reproduction commands into robot inputs. This is done through the nodes shown in the middle of Figure 4.2. For the hand controllers, an Inverse Kinematics solver transforms each hand pose (position and orientation) into joint angles.
- Record robot limb states as demonstrations, to be used by the learning algorithms.
- Manage object recognition, to save new objects or retrieve previously stored ones. This information is streamed over the Matlab side for automatic object detection during

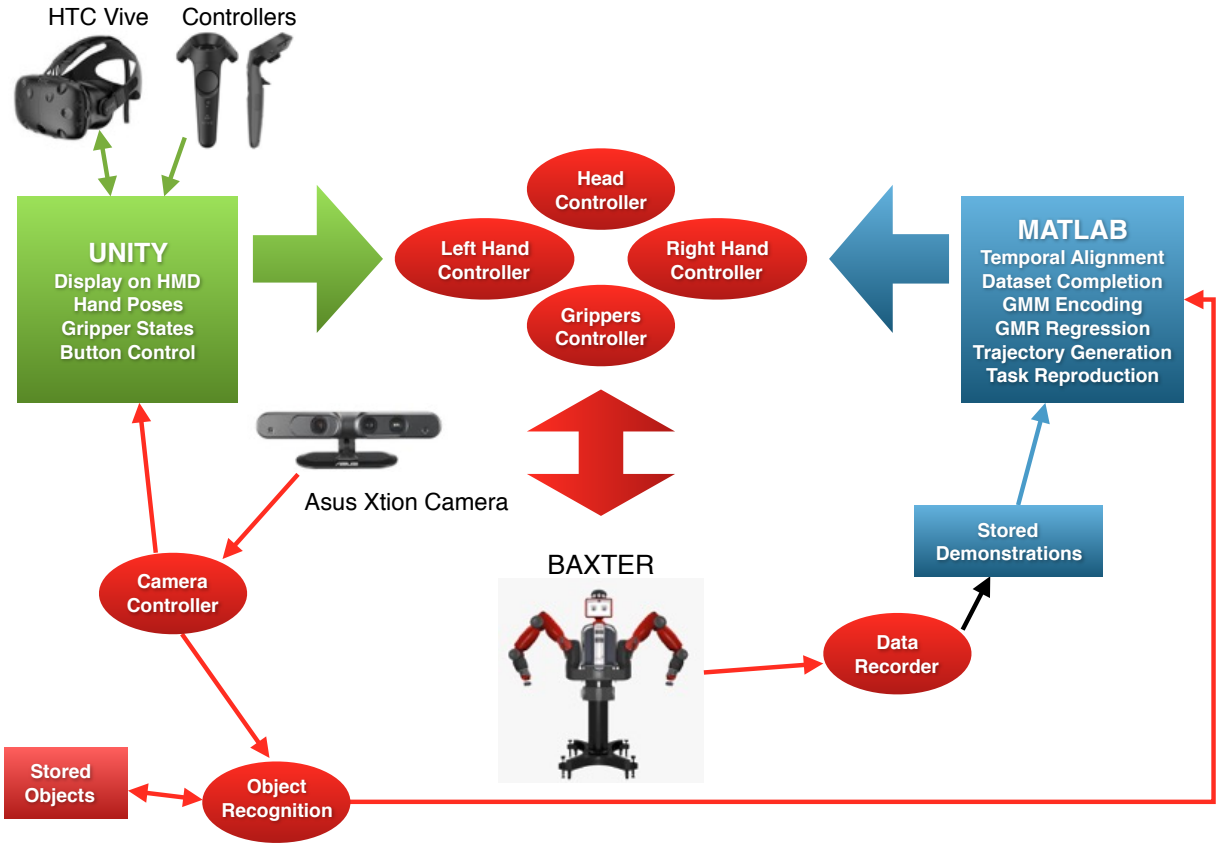

Figure 4.2: A more detailed overview of the designed system. In red are the ros nodes and topics, handling and translating messages between different parts of the system. These nodes convert teleoperation and reproduction inputs into robot commands, manage the camera, record the demonstrations and perform object recognition. In green is the Unity side, used for teleoperation through an HTC Vive's VR headset and two controllers. In blue is the Matlab side, which performs the learning from the stored demonstrations. Multiple steps are shown here between the reading of the demonstrations and the task reproduction.

reproduction.

The messages streamed from Unity and Matlab will be in the same format, so that the same four controller nodes can interpret and translate them into robotics inputs. As far as the robot is concerned, the source of the commands is irrelevant, and a high level of modularity is retained.

For the same principle, the data recorder stores a sequence of states instead of teleoperation inputs. In this manner, multiple methods can be used as means of demonstration (e.g. one might simply drag the robot's arms), and teleoperation is just one of them.

On the Unity side, the spatial information from an HTC Vive headset and two controllers is used for teleoperation. Messages about the head's orientation, hands position and orientation and gripper state (open or closed) are streamed over the ROS network. Moreover, some additional functionalities are controlled via the buttons on the Vive's controllers.

On the Matlab side are the learning algorithms. Here, a sequence of actions is performed starting from the recorded raw demonstrations. A probabilistic Gaussian Mixture Model is trained, which is then used to retrieve generalised trajectories through Gaussian Mixture Regression. These are used as targets generate context-based optimal trajectories, and reproduce the learned task in new environments. Some of the reproduction algorithms will involve automatic object detection.

More details of each element will be discussed in the following chapters.

Chapter 5

Implementation Part 1: Teleoperation

Figure 5.1: Teleoperation at work

This chapter accounts for the design and implementation of all components concerning robotic teleoperation. This system was built to serve as an independent component of the project, so that it can be used on its own. Learning by Demonstration is only one example

application.

The system was designed to interface the Virtual Reality hardware represented by an HTC Vive headset and two controllers with a Baxter Research Robot, in order to allow a human to take the robot's perspective and directly control the robot's limbs as if they were his own.

5.1 ROS and Baxter Overview

The first step of the implementation required getting acquainted with Baxter and its tools.

As mentioned in the Background chapter, Baxter runs on an operating system called ROS. ROS handles a number of independent *nodes* which can be treated as black boxes and serve one or more functions. These nodes exchange messages over *topics*, to which they can *publish* and *subscribe*.

ROS is started up by calling a *roscore* terminal command. The machine that calls this becomes the master (which for this project will always be Baxter), and other machines must be able to connect to it. This is done on the slave machines by setting two environment variables appropriately: ROS_MASTER_URI, with the master's address or hostname, and $ROS\_HOSTNAME$ with the current machine's hostname. Every machine in the network that shares the same ROS_MASTER_URI can publish to the same topics, while individual $ROS\_HOSTNAME$ values are used by the master to send messages to the slaves, such as when the slaves subscribe to a topic.

On top of this, all Linux machines need to match hostnames to actual IP addresses inside the *etc/hosts* file.

ROS messages can manage a list of different data formats, including strings, numbers, and images. Video streams are represented as a sequence of pictures, which can be transmitted over a topic as RAW, or compressed as either JPEG or PNG.

Baxter has three built in cameras, one on the head and two on its hands. However power constraints only allow two of them to be active at the same time. The head camera can be access through the topic *cameras/head_camera* and its video stream is published to *cameras/head_camera/image*. This is a video stream in RAW format, or in ROS terms it comes in *sensor_msgs/Image.msg* packets.

Along the ones for its cameras, Baxter also publishes topics about the current angle of each joint, as well as a full description of the position and orientation of its two end effectors, calculated internally via direct kinematics.

A number of pre-compiled tools and code examples can be obtained from Baxter's SDK, which include tutorials on how to set up a ROS workstation, perform basic functions such as enabling and disabling the robot, obtaining access to its state, control its joint angles, and more.

5.2 Unity and Interfacing with ROS

The HTC Vive was conceived for gaming, and it is therefore commonly programmed through game engines. These are frameworks for the development of gaming software, which allow programmers to define three-dimensional objects inside a gaming scene. These scenes are then rendered at every frame onto a 2D screen, for desktop games, or into a stereo stream (with adjustable disparity) to be experienced through a head mounted display such as the HTC Vive. One of the most popular game engines is Unity, and its Windows version was utilised for this project.

In Unity, objects can be assigned C# (C-sharp) scripts which are evaluated at every frame. These define whatever object behaviours are needed for the game, such as movements, physical properties, rendering textures, etc. Every program in Unity has the same two core functions: `start()`, which is called once and it is used to initialise the script, and `update()`, which is called once per (rendered) frame.

Game engines are not commonly used in robotics, therefore there is no built in interface between Unity and ROS. Moreover, ROS cannot be directly installed on Windows. However, students from the *University of Massachusetts Lowell Robotics Lab* developed a program called *ROS.NET*, which allows to develop C# scripts to run ROS nodes on a Windows machine [48]. Furthermore, the same developers have created another tool, $ROS.NET\_Unity$, which ports all of the features in ROS.NET to the Unity game engine.

In order for ROS.NET to work properly, the machines running on the network must be configured to be able to communicate with each other in the same manner as two ROS machines. Therefore, the lab's Windows machine was set up as such:

- ROS_MASTER_URI = http://011401P0008.prl:11311 (Baxter)
- $ROS\_HOSTNAME$ = ViveMAGIC.prl

ROS.NET works in a slightly different manner than ROS, but it follows the same overall principle: nodes can be created, which publish messages onto topics so that other nodes can listen to them by subscribing to the same topic.

These last steps concluded the workstation set up, which were necessary before any functional program could be developed.

5.3 Camera Connection and Virtual Reality Display

One of the key requirements of the project is to allow the robot teleoperator to acquire the robot's point of view, and see through its eyes. This can be obtained by streaming a feed from Baxter's head camera to Unity, and then render it inside the Virtual Reality to replace any virtual scene.

As mentioned before, Baxter publishes a stream of RAW images of type $sensor\_msgs/Image.msg$ corresponding to the feed from its head camera onto the topic $cameras/head\_camera/image$.

This type of file is quite heavy, and causes unnecessary delays when transmitted over a network. Therefore a compression of the files before streaming was implemented. ROS has a built-in tool, *republisher*, which accepts video frames in one format and re-publishes them in another. This allowed to convert the video stream to a $sensor\_msgs/CompressedImage.msg$ format. Figure 5.2 displays the new network structure. The ellipses represent ROS nodes, while the arrows represent the topics and indicate which node is publishing and which one is subscribing.

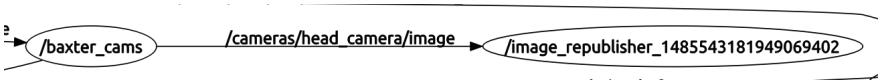

Figure 5.2: Part of Baxter's network showing the format conversion of its head camera's feed

On the Unity side, it was necessary to define a node which can subscribe to the topic published by the above *republisher*. Once this connection is secured, the stream of images is converted into a different texture for each rendered frame, and applied to a rectangular object. This way, inside the virtual reality world, a user has the impression that a video is being played onto an object. If this object is then rendered as a full-screen mesh, then the user can only see what is being played on the video, and he ends up seeing what the robot sees.

The code below shows part of the implementation of this program. It first initialises a node, *teleoperator*, which is then used to subscribe to a topic. The name of the topic is passed as a command line argument, and it is stored in the `topicName` variable. This feature was introduced to make it easier to switch between camera feeds, without hardcoding a specific one into the program. In the `update()` function, the program simply checks if new data arrived from the robot's camera, and executes a call-back function to process the frame, which is turned into a texture and applied to some object in the scene.

```
1 //Called when a new image arrives from the robot's camera
2 void leftCallback(sm.Image s){//data received
3    lastImage = s;
4    newData = true;
5 }
6 //Update is called once per frame
7 void Update () {
```

```
    if (newData){
        tex.LoadImage(lastImage.data);
        GetComponent<Renderer>().material.mainTexture = tex;
        newData = false;
    }
}
```

The above script takes care of "playing" a stream of images onto a rectangular plane, however it does not yet anchor this object in front of the teleoperator's eyes as to replace the virtual reality scene with the robot's visual input. In order to do this, a new script was attached to the plane.

The video must always be in front of the user, therefore it is positioned just beyond the near clipping plane. In game engines, this clipping plane represents a distance from the camera before which objects are not rendered. It can be thought of as the lens of a camera: only objects behind it are recorded.

The video plane is then rotated to face the user, and scaled to cover his entire field of view, by exploiting the virtual camera's field of view and aspect ratio. The code below shows a portion of such script.

```
Camera cam = Camera.main;//obtain handle for main camera
//anchor video plane in front of the user's eyes
float pos = (cam.nearClipPlane + 0.62f);
transform.position = cam.transform.position + cam.transform.forward * pos;
transform.LookAt(cam.transform);
transform.Rotate(90.0f, 0.0f, 0.0f);

//scale it to full screen to cover entire field of view
float h = (Mathf.Tan(cam.fov * Mathf.Deg2Rad * 0.5f) * pos * 2f) / 10.0f;
transform.localScale = new Vector3(h * cam.aspect, 0.1f, h);
```

Figure 5.3 shows the updated network with the new *teleoperator* subscribing to the compressed video stream, while Figure 5.4 shows what the teleoperator sees while wearing the VR headset.

The video quality provided by Baxter's camera is not satisfactory (as observable from Figure 5.4). Moreover, this camera does not provide any depth information about what it

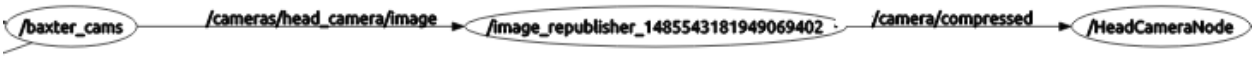

Figure 5.3: Nodes and topics involved in the visual part of the teleoperation

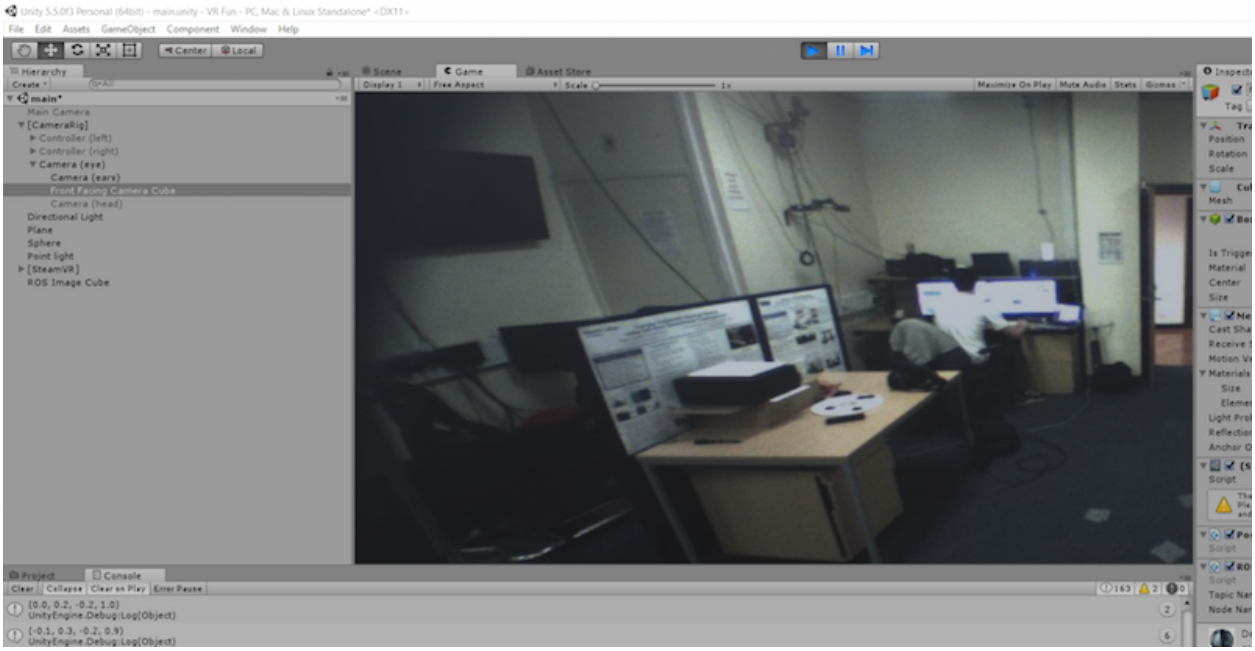

Figure 5.4: A screenshot of Unity showing what the HTC Vive is displaying to the user

sees, which is necessary for object detection (see Section 6.10). Finally, Baxter's joint do not allow to move its head up and down, meaning that the vertical angle of any head camera is fixed, and with the built-in camera it would be very hard to perform tasks with objects on a table.

These reasons justified testing with another camera, and an ASUS Xtion was considered as it provides depth information and its vertical angle is manually adjustable.

In order to interface the Xtion with ROS, it was necessary to install two ROS packages: *Openni_Camera* and *Openni_Launch* [49] [50]. These packages are drivers for Kinect-like OpenNI cameras, which provide both RGB and depth data.

Openni_Launch automatically looks for connected devices, and publishes a list topics with data from the cameras, including $camera/rgb/image_color$ which transports $sensor\_msgs/CompressedIma$ messages. A republisher was therefore not needed for the Xtion camera, and a simple replacement of the topic name in Unity sufficed to substitute the old camera feed for the new one.

The use of the Xtion camera proved very useful for object detection as discussed in Section 6.10, but it also improved the image quality while drastically reducing the lag between the teleoperator's head movements and the camera feed display on the HMD. Indeed, the lag was mainly caused by the need of a republisher to compress the image (although it was still faster than transmittin the RAW image).

5.4 Head Movement

In order to mirror the operator's head movements to the robot ones, a C# script was coded in Unity to obtain angles from the VR headset and publish them over a topic. Then, a Python script in Linux subscribes to such topic, reads in the angles, and applies them to Baxter's neck. Unfortunately, Baxter's neck only allows horizontal movements (i.e. it can look right and left, but not up and down). Therefore the following implementation is designed to such

limited situation, but can easily be expanded to a more general case.

5.4.1 Unity Side

In order to access the orientation of the headset, a node called *teleoperator* is created, and a topic called *head_movement* is attached to it, onto which data can be published inside the `update()` function. Note that for simplicity, the same node is used here as with the camera connection described in Section 5.3.

The horizontal angle is first obtained from a combination of Euler angles: $Y - Z$. Then, it is adjusted so that it can only range from -180 and +180 degrees (this is the range accepted by baxter's neck), choosing where to place the 0 degree position (facing in front). Finally this angle is converted into radians and streamed over the *head_movement* topic. The code below shows a partial implementation of the program.

```
void Update () {
   //obtain horizontal angle
   float horizontalAngle = transform.eulerAngles.y - transform.eulerAngles.z;

   //not shown:
   //adjust angle to go from -180 to +180 with the desired 0 in front
   //convert into radians

   //publish obtained angle
   Messages.std_msgs.Float32 data = new Messages.std_msgs.Float32();
   data.data = horizontalAngle;
   pub.publish(data);
}
```

5.4.2 Linux Side

On the Linux side of the system, it was necessary to implement a Python script in order to create a node which can subscribe to the *head_movement* topic, extract the information about the head's angle, and apply it to the robot.

Figure 5.5: The new network with the connections between the VR headset and the robot

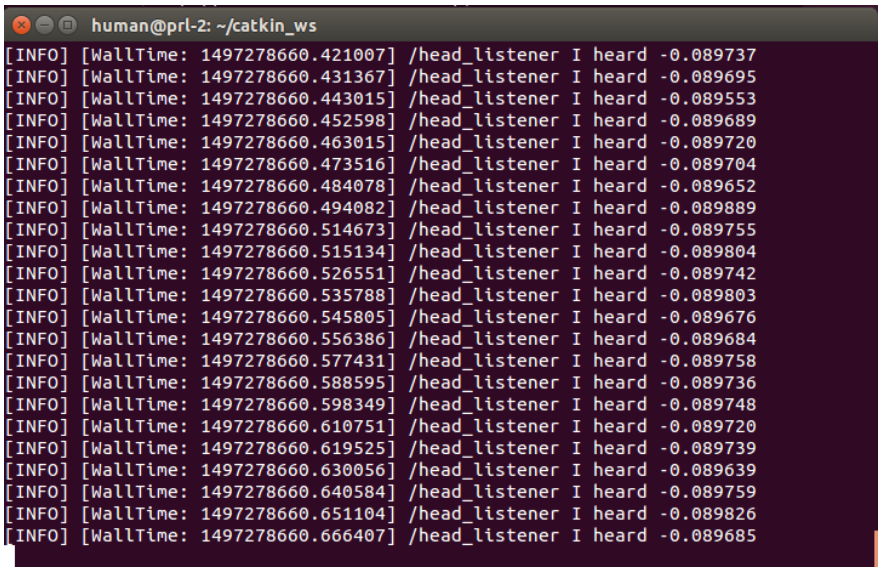

Figure 5.6: A screenshot of Baxter's subscription to the /head_movement topic

Below is a partial implementation of such program, showing the relevant parts of the code used for such tasks.

```
1  def callback(head_angle):
2      #Set new pan angle if received data is different enough from current angle
3     if not (abs(head.pan() - head_angle.data) <=
           baxter_interface.HEAD_PAN_ANGLE_TOLERANCE):
4        head.set_pan(head_angle.data, speed=1, timeout=0)
5
6  def main():
7     rospy.init_node('head_listener')
8     head = baxter_interface.Head() #Obtain handler for Baxter's head
9     rospy.Subscriber('head_movement', Float32, callback, queue_size=1)
```

Figure 5.5 shows the updated network, which involves both the camera connection and the head movement parts. Figure 5.6 instead displays how Baxter successfully receives data about the orientation of the VR headset.

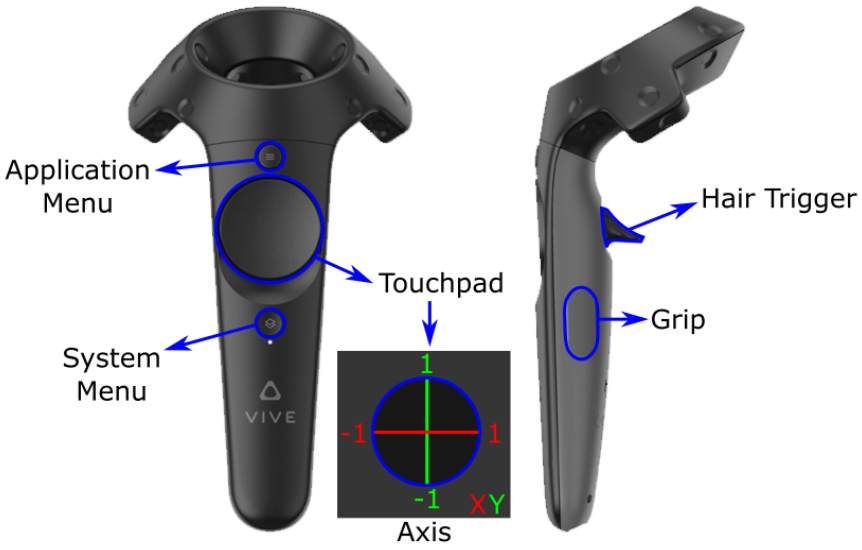

Figure 5.7: A depiction of a HTC Vive controller and its buttons

5.5 Hand Movements

Similarly to what was done with the head movement, one of the requirements of the project is to allow the teleooperator to guide the robotic arms by holding two VR controllers, one in each hand. This time, however, a much more comprehensive state of the controllers must be published, including a 3D position of both hands, as well as a 4D orientation. Moreover, one more state is required to determine when the operator wishes to open and close the robot's hands.

Figure 5.7 displays the type of controller used for this purpose. In the following sections, this figure will be used as reference when accounting for how the buttons were assigned different functions.

5.5.1 Unity Side

For this part, a program was implemented to perform a number of tasks. The main functionality of the program is to obtain the positions of the hands in the right frame of reference and publish them over the $right_movement$ and $left_movement$ topics. Indeed, it is not enough to

publish the hand poses, as Baxter will need positions and orientations relative to the origin of its frame of reference (situated roughly inside Baxter's belt).

To overcome this problem, it was assumed that Baxter's coordinate origin lies directly below its head, and a variable called `headToOriginDistance` is used to fine tune an appropriate distance. Once this is in place, the hands positions are calculated relative to the VR headset, then `headToOriginDistance` is subtracted from the vertical axis to obtain the hand positions relative to Baxter's axis origin.

During this process, it was noted that the orientation of the three axis (X, Y, Z) is different in the two frames of reference. In particular, in Baxter's world the Y axis points to the left, while the Z points upwards. These are inverted in the Virtual Reality world, and the code accounts for this discrepancy.

Beyond the hand position, an attempt was made to reproduce the wrist orientation of the teleoperator onto the orientation of Baxter's grippers. This would allow a more flexible operation than with fixed gripper orientation.

Orientations are represented as four-dimensional quaternions both in ROS and in Unity. However, quaternion conversion between two frames of reference is a non-trivial task. Indeed, quaternions are not intuitively readable by humans, and after a substantial research over their usage, a working solution was eventually found by trial and error by changing the mapping between the four dimensions until the correct conversion was obtained. Once both position and orientation were correctly reproduced by Baxter's arms, the following functions were assigned to controller buttons:

- Grip Button: enables and disables the teleoperation of the respective arm, and was introduced for practical and security reasons. When it's pressed, a variable `shouldStream` is toggled;
- Touchpad Button: enables fixed gripper orientation. In this mode, the wrist angle is no longer published, and Baxter's gripper is fixed to point downwards, in the most comfortable position to grab objects.

```
void Update () {
   //store poses of controller and head (not shown)
   //adjust head's position to be above user's belt
   headPose.position.z = headPose.position.z - headToOriginDistance; //(0,0,0) is
       at belt height,

   //find hand position in the right frame of reference
   //scale operators movements to exploit full Baxter arm range
   handPose.position.x = 1.3*(handPose.position.x - headPose.position.x);
   //(same for other dimensions)

   //read button inputs
   gripButtonDown = controller.GetPressDown(gripButton);
   touchpadButtonDown = controller.GetPressDown(touchpadButton);
   if (gripButtonDown){//Grip Button was pressed, toggle teleoperation
      shouldStream = !shouldStream;
   }
   if (touchpadButtonDown){//Touchpad Button was pressed, toggle writs angle
      shouldUseQuaternion = !shouldUseQuaternion;
   }

   if (!shouldUseQuaternion){
      //use fixed orientation: gripper points downward (not shown)
   }

   if (shouldStream){//only stream data if teleoperation is enabled
      pub.publish(handPose);
   }
}
```

A similar program is in charge of informing about the desired state of Baxter's gripper: open or closed. This time, the Trigger button was utilised. The *teleoperator* node publishes commands onto a topic named $gripper_movement$. The messages are in the $std_msgs.String$ ROS format. Four different strings are used to open or close one of the grippers:

- "q": close left gripper
- "Q": close right gripper
- "w": open left gripper
- "W": open right gripper

Below is a portion of the code used to control the right gripper. The rest of the code is omitted as it closely resembles the previous program.

```
pub = nh.advertise<Messages.std_msgs.String>("/gripper_movement", 1, false);

if (triggerButtonDown){//Trigger Button was just pressed
   message.data = "Q";
   pub.publish(message);
}
if (triggerButtonUp){//Trigger Button was just unpressed
   message.data = "W";
   pub.publish(message);
}
```

5.5.2 Linux Side

On the Linux side of the system, two Python scripts create nodes which can subscribe to the *right_movement* and *left_movement* topics, extract the information about the hands state, and apply it to the robot. A third script then takes care of listening for signals to open or close Baxter's grippers. This modular design choice was opted for easier debugging as well as retaining the possibility to reuse these scripts independently in a future application.

The scripts that read a hand position and orientation must be able to convert this information into joint angles for the respective 7-DOF arm. This can be done through Inverse Kinematics (IK). Indeed, while Direct Kinematics calculates the position of an end effector based on joint angles and link lengths, IK performs the opposite calculation, and outputs a set of angles from an end effector's pose. Multiple solutions might be found by an IK solver, and different implementations will favour one over the others.

As discussed in Section 2.3, Baxter has built in IK tools. They can be accessed through the *ExternalTools* topic, published by Baxter at startup. For example, the IK solvers for the right arm can be accessed at *ExternalTools/right/PositionKinematicsNode/IKService*. This tool takes the form of an IK-ServiceProxy which accepts IK-Requests containing the desired pose. The response of the service proxy can be unpacked into a list of joint angles

(provided the IK solver returned a valid solution) and applied directly to one of Baxter's arms. Below is an extract of the program used to control Baxter's right hand.

```
def ik_solver(pose): #called when new data appears on /righ_movement
   ikRequest = SolvePositionIKRequest()
   ikRequest.pose_stamp.append(pose)
   try:
      resp = ikServiceProxy(ikRequest)
   except (rospy.ServiceException, rospy.ROSException), e:
      rospy.logerr("Service call failed: %s" % (e,))
      return 1

   if (resp[0] != resp.RESULT_INVALID): # if a valid solution was found
      # reformat the solution arrays into a dictionary
      joint_solution = dict(zip(resp.joints[0].name, resp.joints[0].position))

      # set arm joint positions to found solution
      arm.set_joint_positions(joint_solution)
   return 0
```

Some experimentation was carried out using a third-party IK solver called *MoveIt*!. This is a package that can be installed on a ROS network and used directly on Baxter after some parameter settings. However, *MoveIt*! comes with an extensive list of tools and functionality that were not needed for the purposes described in this section, and were perceived as an over-complication of a simple task. Since the performance of the built-in IK solver was very satisfactory, it was eventually resolved to fall back on the original tool.

Regarding Baxter's grippers, a much simpler program was implemented to accept commands as strings from the Unity side over the *gripper_movement* topic, which handles both the right and left grippers. Baxter's grippers need to be calibrated before they can be used, therefore the following program takes care of this initial step before allowing gripper control.

The code below shows a partial implementation of such program, which focuses on commands for opening and closing the grippers. Not shown here, many more commands are accepted by the program, mostly regarding parameter settings such as gripper holding force, moving velocity, and exact position. No support for these extra commands was implemented in the Unity part, but this way this Python script retains a higher degree of modularity and

reusability.

```
def respond(data): #callback function handling received commands
   #A list of accepted commands (most commands are not shown here)
   bindings = {
      #   key: (function, args, description)
      "c": (left.calibrate, [], "left: calibrate"),
      "C": (right.calibrate, [], "right: calibrate"),
      "q": (left.close, [], "left: close"),
      "Q": (right.close, [], "right: close"),
      "w": (left.open, [], "left: open"),
      "W": (right.open, [], "right: open"),
   }

   if data.data in bindings:     #if the received command is recognised
      cmd = bindings[data.data]  #extract command from received string
      cmd[0](*cmd[1])          #send command to Baxter's gripper
```

5.6 Summary of Teleoperation

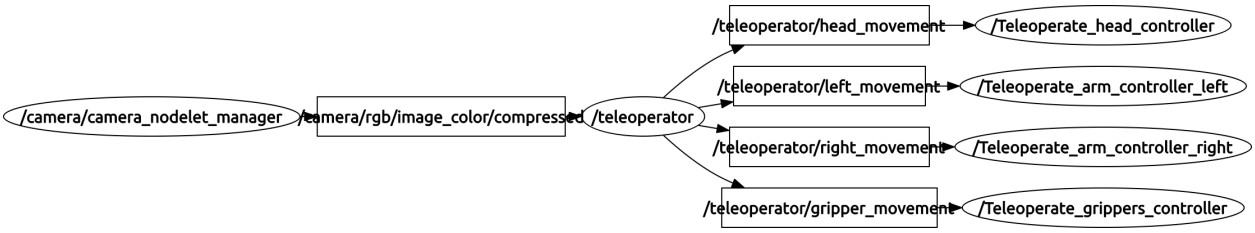

Figure 5.8: A graph of the main ROS nodes and topics involved in the teleoperation

Figure 5.8 displays the complete network of ROS nodes and topics involved with the Teleoperation part of the implementation. The nodes are represented by ellipses, while the rectangles represent the topics, with the arrows indicating which node publishes the topic and which node subscribes to it.

The *teleoperator* node is the only one running in Unity, and handles all functions within the Virtual Reality world. All other nodes run on the Linux side of the Teleoperation.

The *camera/camera_nodelet_manager* publishes a compressed video stream, which is picked up by *teleoperator* to display what the robot sees inside the HMD of the HTC Vive.

The node *teleoperator* then publishes four topics, corresponding to the poses of the teleoperator's head and arms, as well as a state of the teleoperators grippers (open/closed). These messages are read by four nodes on the Linux side, which parse the data and apply it to the robot to mirror the teleoperator's movements.

Chapter 6

Implementation Part 2: Learning by Demonstration

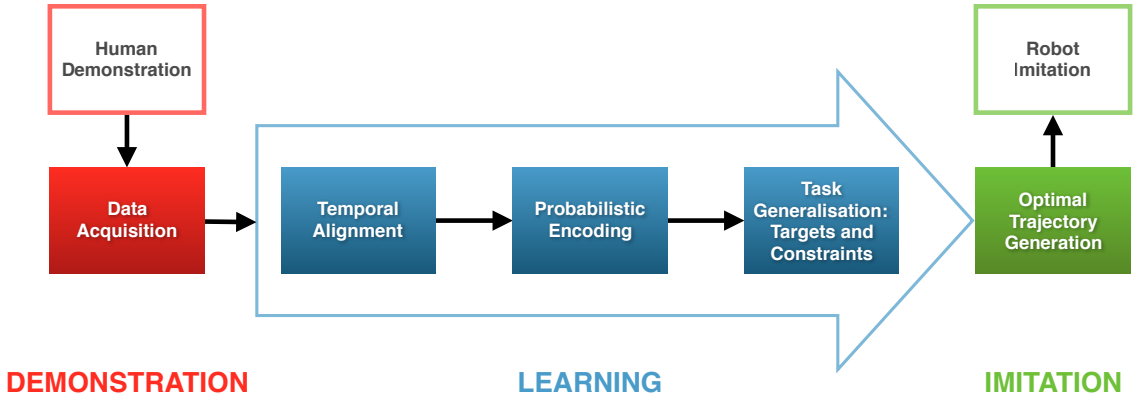

Figure 6.1: A diagram displaying the subcomponents of the Learning by Demonstration chain

This chapter accounts for the design and implementation of all components concerning the learning of a demonstrated task. In principle, this system should be independent of the first implementation part, and teleoperation is only one of the way to teach a robot a task (See Section 6.1).

The system was designed to generalise a task by analysing a set of demonstrations, represented by a sequence of temporal and spacial data recoded during the teaching. The

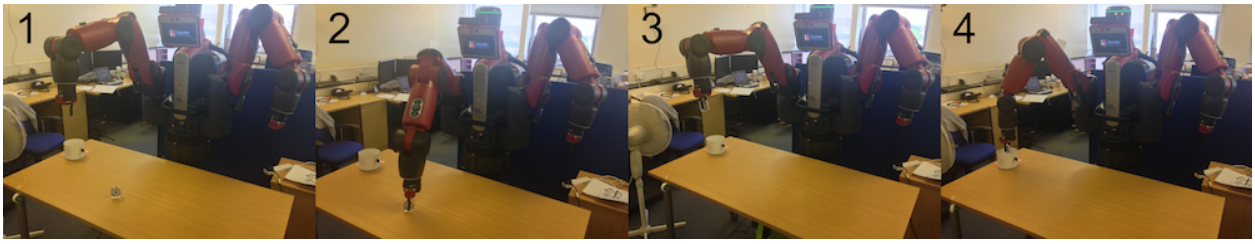

Figure 6.2: Four snapshots demonstrating the task of picking up an object somewhere on a table, and placing it inside a box.

following sections will provide a detailed analysis of all the tools used to achieve robust learning, accounting for possible alternatives and motivations for the chosen ones.

Figure 6.1 displays the proposed steps for Learning by Demonstration. The three main components are *Demonstration*, *Learning*, and *Imitation*. The first step consists on performing the same task multiple times, and recording spatio-temporal data of the demonstrations. This data is then passed to the Learning phase, where patterns in the data are used to align the demonstrations in time. These are then encoded into a probabilistic model, which captures the key features of the demonstrated task, and their relative importance for the task. This information is used to generate a generalised task trajectory, with target states and constraints to be met in different portions of the same task. Finally, the generalised trajectory is used on-line to produce a sequence of inputs for the robot's limbs. These inputs are calculated as to generate an optimal trajectory, based on the similarity with the generalised trajectory at key portions of the task. The *Demonstration* and *Learning* steps are done once a priori. Their outputs are then used every time for *Imitation* under different initial conditions.

Figure 6.2 shows an example task, consisting in grabbing an object from a table and placing it inside a box somewhere else. For this project, this task was used for reference to keep track of the learning performance, and was eventually expanded into clearing the table of all objects by finding them and moving them all inside the box.

For the remainder of the chapter, the following notation will be used:

- X: Training set
- M: number of variables (dimensions) in a demonstration
- T: Number of time steps in a demonstration
- N: number of demonstrations
- K: number of model components

6.1 Kinesthetics vs Teleoperation

Although teleoperation is the focus of this project as a means of demonstrations, it is not the only method that can be used with the proposed system. Any technique that allows an operator to show the robot the correct movements can be applied here.

Kinesthetics is an example of this. It consists in directly moving the robot's arms to the desired positions, by dragging its limbs. With Baxter, kinesthetic teaching is particularly easy to carry out, as the robot has a built-in *zero-gravity* mode, where it lets the user drag its end effectors without applying any resistance, and counterbalancing the gravitational force.

This feature of Baxter was exploited in the early stages of the implementation of a learning algorithm, given its practicality. In particular, Kinesthetics represent a quick and reliable method to inform the robot about the positions of objects in absence of a visual input for object recognition, which was only implemented at a later stage. Thus, object positions were initially taught by dragging a robot's hand over them.

6.2 Choice of parameters

For the task of grabbing and placing an object into a box, the following set of state data was recorded during demonstration, as it was deemed to cover all important information:

- A 1D temporal sequence, denoted by x_t
- A 3D spatial sequence representing the position of the robot's end effector, denoted by x_s
- A 3D spatial sequence representing the distance between the position of the robot's end effector and the initial position of the object to be moved, denoted by x_o
- A 3D spatial sequence representing the distance between the position of the robot's end effector and the final position of the object to be moved (i.e. the position of the box), denoted by x_f
- A 1D binary state sequence of the robot's gripper (0 for open and 1 for closed), denoted by x_g

It should be noticed the constant relationship between x_s and both x_o and x_f. As the initial object position p_o and final box position p_f are constant across one demonstration (but vary across demonstrations), the three parameters are related by the following:

$$\begin{aligned} x_{o,i,j} &= x_{s,i,j} - p_{o,i} \qquad where \qquad i = 1, ..., N; j = 1, ..., T \\ x_{f,i,j} &= x_{s,i,j} - p_{f,i} \qquad where \qquad i = 1, ..., N; j = 1, ..., T \end{aligned} \tag{6.1}$$

6.3 Data Acquisition: Demonstration Recording

A ROS program is used to record the demonstrations, which creates a a comma-separated-value (CSV) file for each demonstration. The program accepts as inputs the name of the file and two 3D positions, respectively p_o and p_f. Once these are provided, the program gets handlers for Baxter's limbs and stores the following state readings, in order, to the CSV file:

- A 1D time stamp (x_t).
- Two 3D end effector positions, one for each arm (x_s).

- Two 1D gripper state variables (x_g).
- p_o.
- p_f.

These 11 values are recorded at a rate that can be passed as argument to the program (or 100Hz by default), and written in consecutive rows of the CSV file. Table 6.1 displays the first 4 lines of an example demonstration.

Table 6.1: A table showing the first 4 lines of a recorded demonstration.

time	right_x	right_y	right_z	right_grip	p_o_x	p_o_y	p_o_z	p_f_x	p_f_y	p_f_z
0.485	0.654	-0.548	0.124	0.000	0.737	-0.081	-0.187	0.482	-0.676	-0.074
0.486	0.654	-0.548	0.124	0.000	0.737	-0.081	-0.187	0.482	-0.676	-0.074
0.496	0.654	-0.547	0.124	0.000	0.737	-0.081	-0.187	0.482	-0.676	-0.074

Once N demonstrations have been recorded, their files are passed to the temporal alignment algorithm described in Section 6.4, which, among other things, re-shapes the demonstrations to last equally long. Then, this program replaces the temporal values with a sequence of integers from 1 to T for each demonstration. After that, a separate program calculates x_o and x_f and replaces the 6 columns containing p_o and p_f with them.

Such formatted dataset is now ready to be passed for model training.

6.4 Temporal Alignment

However accurately one might demonstrate the same task multiple times, some degree of temporal misalignment is bound to be shown across demonstrations. This is because the same task could be (for example) performed faster or a few seconds later compared to other demonstrations. This is an issue when learning must be performed observing multiple demonstrations.

For example, the most naive learning approach might try to simply average out the recorded signals, and misalignments would clearly corrupt the correctness of the output. Fortunately, various approaches have been developed to solve this problem.

A common technique to encode temporal data is by constructing a Hidden Markov Model. As discussed in Section 2.9, HMMs not only provide temporal alignment by encapsulating observations over time into a single model, but also provide a probabilistic encoding of the states and their transitions. Because of this, HMMs are very commonly used in Learning by Demonstration [33], [34], [35].

However, Calinon et al. [45] argue that, when used in conjunction with Gaussian Mixture Models, the information encoded in the two models is redundant and even hurtful. Although HMMs can deal very well with temporal misalignment, their representation of a task as a set of transition probabilities makes it very hard to retrieve a smooth trajectory from the model.

In [51], Inamura et al. attempted to average the output sequences obtained from a HMM. Their approach, however, required a very large number of sequences.

To solve this issue, in [45] and [52], the temporal signals are treated as an extra state variable, and used as inputs to retrieve a generalised trajectory via Gaussian Mixture Regression. The temporal alignment is carried out independently, before the probabilistic encoding step, using Dynamic Time Warping.

With their approach, GMM and GMR can be used to learn a task directly from a pre-aligned sequence of demonstrations. Therefore, as a first step, a naive temporal aligment algorithm was developed for the specific task at hand (grabbing and placing an object in a box), and more general techniques were attempted at a later time.

The algorithm in question exploited a constant recurrence across the demonstrations: the robot must close its gripper just before grabbing the object, and open it just before releasing the object inside the box. These key steps are particularly apt for temporal alignment as they subdivide the task into four recurring subtasks: reaching for the object (under given

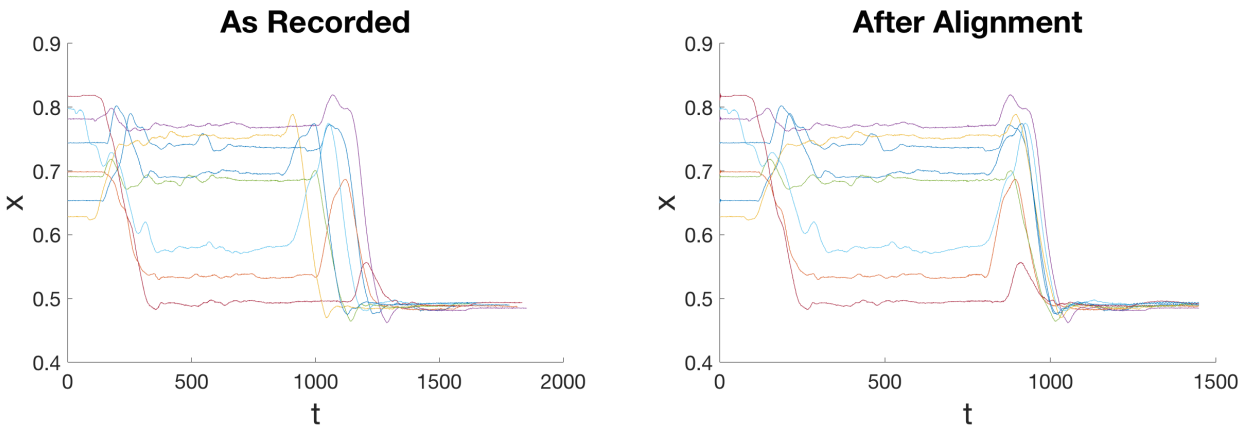

Figure 6.3: Alignment of 8 demonstrations for the task of grabbing and placing an object in a box, obtained by re-sampling the three portions of the signals before, in between, and after the robot grabs and releases the object. Only the x position of the end effector is displayed here.

initial conditions), grabbing it, carrying it over the box, and releasing it.

By taking note of the temporal instance when the robot closes and opens its gripper for each demonstration, the algorithm then splits each signal into three sections, and re-samples them independently in order to provide three segments of known lengths l_1, l_2, and l_3, where $l_1 + l_2 + l_3 = T$. These segments are then concatenated back together to obtain N temporally-aligned demonstrations of length T.

Figure 6.3 shows the performance of such algorithm (only the x position of the end effector is displayed). For the eight demonstrations shown here, the initial position of the object was always changed, while the position of the box remained constant. As observable, the resulting signals are much better aligned than the original ones.

This algorithm was used throughout most of the implementation phase. However, a less naive solution was developed, which can be applied to more general problems, and makes use of Dynamic Time Warping [42]. As discussed in Section 2.11, DTW is capable of distorting two signals temporally in order to minimise their Euclidean difference. The resulting signals have the same length but are generally longer than the two originals, as portions of both are repeated for consecutive time instances.

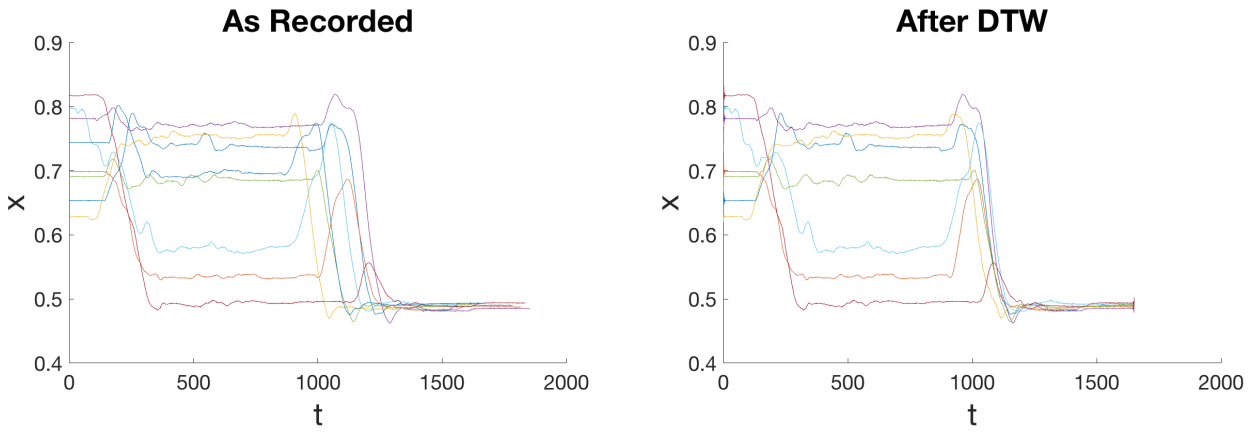

Figure 6.4: Alignment of 8 demonstrations for the task of grabbing and placing an object in a box, obtained through Dinamic Time Warping followed by targeted re-sampling. Only the x position of the end effector is displayed here.

This presents a problem when more than two demonstrations are provided, as comparing demonstrations two by two generates N signals of different length. An additional step is therefore applied.

One of the demonstrations is chosen as reference, and its length l_r is saved. The reference is thus used in pair with a second demonstration in a DTW algorithm. Finally, the distorted second demonstration is resampled to be l_r long. The process is repeated for all demonstrations, obtaining N, l_r-long, temporally-aligned sequences. Figure 6.4 shows the result of such algorithm. The output signals are very similar to those obtained by the original algorithm, but this time no case-specific information was needed to obtain the temporal alignment.

6.5 Gaussian Mixture Models

As discussed in Section 2.14.1, Gaussian Mixture Models provide a powerful tool to encode a sequence of spatio-temporal observations into a probabilistic framework. A Gaussian Mixture Model is made of K multivariate gaussian distributions, with means and covariance matrices μ_k and Σ_k. These parameters are estimated through Expectation Maximisation.

EM algorithms are a good tools to estimate GMM parameters, however, as discussed in Section 2.8, they tend to converge to local optima, and give different outputs depending on initial conditions. Therefore, to attempt reaching a global optimum, a k-means clustering is applied first to the data, to find K estimates of μ and Σ. These are then used as initial conditions for the EM algorithm.

6.6 Number of components K

EM algorithms have no way of estimating the optimal number of components in a GMM. Therefore other tools must be used to choose K. Indeed, adding components to a model only increases the likelihood until a certain point, after which overfitting occurs. Two common approaches are the Akaike Information Criteria (AIC) and the Bayesian Information Criteria (BIC), which are both penalized-likelihood criteria.

AIC determines the quality of a model by summing a constant component, representing the cost of adding more model parameters, and an estimate of the maximum likelihood for the model. Therefore a the model with the lowest AIC is considered to be closer to the truth [53]. If K is the number of model components, and $\hat{L}$ is the maximum likelihood, then AIC is defined as:

$$AIC = 2K - 2ln(\hat{L}) \tag{6.2}$$

BIC is very similar to AIC, and it estimates posterior probability of a model being correct. Once again the model with the lowest BIC is considered to be closer to the truth [54]. If n is the number of datapoints in the demonstration set X, then:

$$BIC = ln(n)K - 2ln(\hat{L}) \tag{6.3}$$

Both methods make various assumptions and asymptotic approximations [55]. Their

main (and arguably only) difference is that BIC penalizes model complexity more than AIC. Therefore AIC tends to choose larger models than BIC.

Both these models were taken into account when choosing the optimal number of components.

6.7 Gaussian Mixture Regression

The theoretical rationale behind GMR and its usefulness in Learning by Demonstration was discussed in Section 2.15. In particular GMR shows a high degree of versatility, as it returns joint distributions of the encoded variables, and only at a later stage one can decide which are inputs and which are outputs. GMR was therefore chosen as the algorithm to produce a generalised trajectory from the demonstrations, once a GMM is in place.

GMR outputs both $\hat{\mu_X}$ and $\hat{\Sigma_X}$ at every time step. The $\hat{\mu_{X,t}}$ can be directly treated as the generalised trajectory, or target trajectory during reproduction. The $\hat{\Sigma_{X,t}}$ represent constraints around the trajectory for each variable of the dataset X, at different portions of the task. They can therefore intuitively be thought of as how important it is to keep different state variables close to their targets at different moments in the reproduced task. As we will see in Section 6.9, this characteristic will play an important role in the optimal trajectory generation.

6.8 Time vs Space Prediction

In previous works (e.g. [45] and [47]), separate Gaussian Mixture Models are used to encode different parameters, by pairing them up with the temporal data individually. In the current case, this would mean generating different GMMs for datasets composed of x_t, x_s, x_t, x_o, x_t, x_f, and x_t, x_g. Then, temporal data is generated during reproduction as an estimation of x_t and always used as input for GMR to retrieve the other parameters independently.

This is however very limiting, and wasteful of the versatility of GMR. For this project, the entire dataset X is used to obtain a single GMM. This way, the regression is not limited to temporal data inputs, but can instead use any of the other variables as both inputs and outputs.

For example, in determining when the robot should close its grippers, it makes much more sense to ask "has the hand reached the object?" than "have 10 seconds passed, which is what it normally took the demonstrator to reach the object?". With this approach, it was possible to control the arm's movements based on time such as in previous approaches, while deciding to close/open the gripper based on the distance between the hand, the object, and the box.

6.9 Optimal Trajectory Generation

GMR returns a set of generalised target states and trajectory constraints. The next step is to implement a method to generate new trajectories on-line each time the task needs to be reproduced under varying initial conditions. One way to do this is to define a cost function to represent how closely the generated trajectory is to the target one.

To do that, we first define X as the candidate state, and $\hat{X}$ as the target state. Then, the distance between X and $\hat{X}$ is:

$$\sum_{m=1}^{M} w_i(x_i - \hat{x}_m) = (x_m - \hat{x}_m)^T W (x_m - \hat{x}_m) \tag{6.4}$$

That is, for all M variables in X, find the distance between x_m and $\hat{x}_m$ and weight its importance with a time-varying weight matrix W. Not all variables in X should affect the trajectory generation, therefore equation 6.4 can be slightly modified to better separate the different parameters of the dataset X involved, namely x_s, x_o and x_f, and generate a const function C:

$$C = (x_s - \hat{x}_s)^T W_s (x_s - \hat{x}_s) + (x_o - \hat{x}_o)^T W_o (x_o - \hat{x}_o) + (x_f - \hat{x}_f)^T W_f (x_f - \hat{x}_f) \tag{6.5}$$

To find an optimal controller, the cost function defined in equation 6.5 must be minimised subject to the constraints between x_s, x_o and x_f:

$$\underset{x_s}{\text{minimize}} \quad (x_s - \hat{x}_s)^T W_s (x_s - \hat{x}_s) + (x_o - \hat{x}_o)^T W_o (x_o - \hat{x}_o) + (x_f - \hat{x}_f)^T W_f (x_f - \hat{x}_f)$$

subject to:

$$x_o = x_s - p_o$$

$$x_f = x_s - p_f$$

The problem can be solved by defining a Lagrangian:

$$L(x_s, x_o x_f, \lambda_1, \lambda_2) = C + \lambda_1^T (x_s - x_o - p_o) + \lambda_2^T (x_s - x_f - p_f) \tag{6.6}$$

and solving for $\Delta L = 0$:

$$\Delta L_s = 2W_s(x_s - \hat{x}_s) + \lambda_1 + \lambda_2 = 0$$

$$\Delta L_o = 2W_s(x_o - \hat{x}_o) - \lambda_1 = 0$$

$$\Delta L_f = 2W_s(x_f - \hat{x}_f) - \lambda_2 = 0$$

then:

$$W_s(x_s - \hat{x}_s) + W_o(x_s - \hat{x}_o - p_o) + W_f(x_s - \hat{x}_f - p_f) = 0$$

$$[W_s + W_o + W_f]x_s = W_s\hat{x}_s + W_o(\hat{x}_o + p_o) + +W_f(\hat{x}_f + p_f)$$

and finally:

$$x_s = [W_s + W_o + W_f]^{-1}[W_s\hat{x}_s + W_o(\hat{x}_o + p_o) + W_f(\hat{x}_f + p_f)] \tag{6.7}$$

Equation 6.7 represents an optimal controller for the candidate trajectory for the robotic arm $x_s = (x, y, z)$ at each time step.

As discussed in Sections 2.15 and 6.7, the GMR algorithm returns a covariance matrix $\hat{\Sigma}$ at every time step t as well as the $\hat{x}$ used in the above derivation. Each of these matrices indicates how similar each variable m was at time t to the same variable m across the demonstrations. The higher the variance, the less similarly a particular variable m behaved at time t. This is also observable in the output of the GMR as wider constraints on the variable at time t.

Alternatively, this characteristic of $\hat{\Sigma}$ can be interpreted as such: the higher the variance of the variable m at time t, the less important it is to keep it close to its target trajectory at time t. By contrast, the smaller the variance, the more important it is. Therefore, it is intuitively understood that a good way to define the weightings is:

$$W = (\hat{\Sigma})^{-1} \tag{6.8}$$

Or, more specifically, each of the weighting matrices can be obtained as a subset of $\hat{\Sigma}$ as such:

$$W_s = (\hat{\Sigma}_s)^{-1}, \quad W_o = (\hat{\Sigma}_o)^{-1}, \quad W_f = (\hat{\Sigma}_f)^{-1} \tag{6.9}$$

This way the weightings will vary with time, increasing in portions of the reproduction where little variance was observed in the demonstrations, and decreasing in portions of the reproduction where high variance was observed.

For example, suppose that in the grab-and-place task the operator grabs the object at time t_{grab}. x_o will show little variance at t_{grab}, when $x_o = [0, 0, 0]^T$. This is because in every demonstration this was the case right before grabbing the object. Therefore W_o will have a higher value than W_s and W_f at $t = t_{grab}$, and equation 6.7 simplifies to $x_s \approx x_o + p_o = p_o$, i.e. the robot's hand moves to where the object is.

6.10 Object Detection and Recognition

So far it was possible to implement a system that accurately reproduces a demonstrated task, such as grabbing an object in position p_o and placing it in position p_f. However, these object position and final position need to be passed as parameters. For a full autonomous task reproduction, some form of computer vision should be implemented as to allow the robot to automatically detect these positions.

The ASUS Xtion camera has a mono rgb camera, and a depth sensor. The two can be used in conjuction for object recognition. A package called $find\_object\_2d$ [56] was created by third party developers to run on ROS and recognise saved objects on a 2D rgb video stream. It does so by mapping a number of features (i.e. patterns on the objects) to a saved object, and recognising them in the video frames.

This information can be used in conjunction with the data provided by the depth sensor on the Xtion camera to estimate a 3D position of a recognised object via trigonometry. For this purpose, a module called $find\_object\_3d$ (part of $find\_object\_2d$) allows a user to manually indicate where an object is, saves its features, and publishes the objects 3D positions onto a ROS topic called $/objectsStamped$, as messages of type $find\_object\_2d/ObjectsStamped.msg$.

These positions, however will be in the frame of reference of the camera, which has the origin on a frame called $/map$. Figure 6.5 displays part of the tree of frames published by $find\_object\_3d$. Objects are always positioned with respect to $/camera\_depth\_optical\_frame$, and it's up to the user to define the relative position of the branches to correspond to the distances between the rgb and depth cameras on the Xtion. These frame-to-frame transformations are published onto a topic called $/tf$ as messages of type $tf2\_msgs/TFMessage$, which consist of an array of 7 values comprised of a 3D translation and a 4D (quaternion) rotation from the parent frame to the child.

It is also up to the user to define the relative position of the camera with respect to Baxter, by publishing a message on $/tf$ defining a transform from the frame $/base$ (Baxter's origin) to $/map$. This part is particularly challenging, as quaternions are involved to define

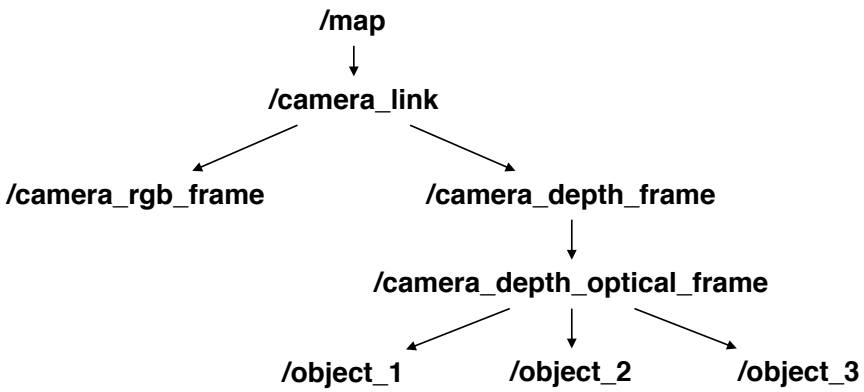

Figure 6.5: Tree of frames published by $find\_object\_3d$.

the relative orientation of the camera with respect to the robot. To help the user, a tool called *rviz* is built in ROS that allows to visualise how the frames on the ROS network are positioned with respect to each other according to the messages on $/tf$.

Figure 6.6 displays an *rviz* visualisation of a successful object detection from a camera placed on top of Baxter's head. It is visible how the manually-adjusted transformations accurately represent the real orientation of Baxter's surroundings.

The following transform is applied from Baxter's $/base$ to $/map$:

- Translation: x = 0.250, y = 0.08, z = 0.075
- Rotation: x = 0.0, y = 0.605297, z = 0.0, w = 0.796

Once a stable object recognition was in place, a program was developed to test the automatic reading of object positions in front of Baxter. This program performs the following:

1. Finds and stores the names of all recognised objects, by subscribing to the $objectsStamped$ topic.
2. For every object:

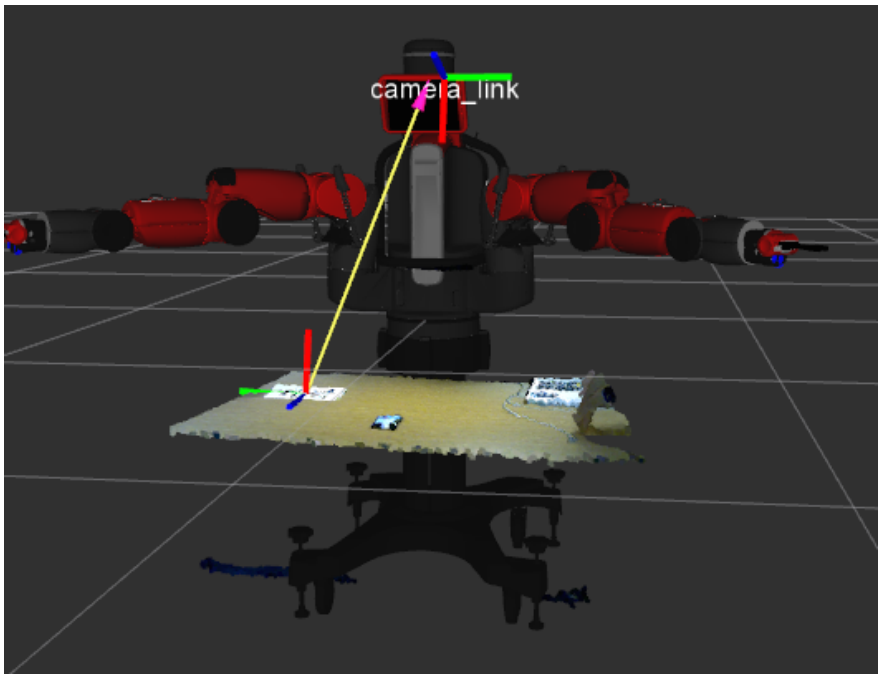

Figure 6.6: Object detection as observed inside rviz.

(a) Finds the objects position with respect to Baxter's origin. This is done with a ROS tool called $tflistener$, which returns the complete transform between two frames (e.g. $/base$ and $/object$) by iterating through the tf tree.

(b) Publishes it on a topic called $initial\_position$, to which a separate node, in charge of the task reproduction, subscribes.

(c) Waits for the robot to place the object in the box, by listening to a $subtask\_done$ topic.

This program expands the task of grabbing and placing an object in a box, to a more general scenario where all objects on a desk must be put away. Moreover, the structure of this program was kept general enough to apply to any task which involves subtasks. It is up to the receiving node to perform the appropriate subtask. Figure 6.7 displays how this modularity is applied in this specific case.

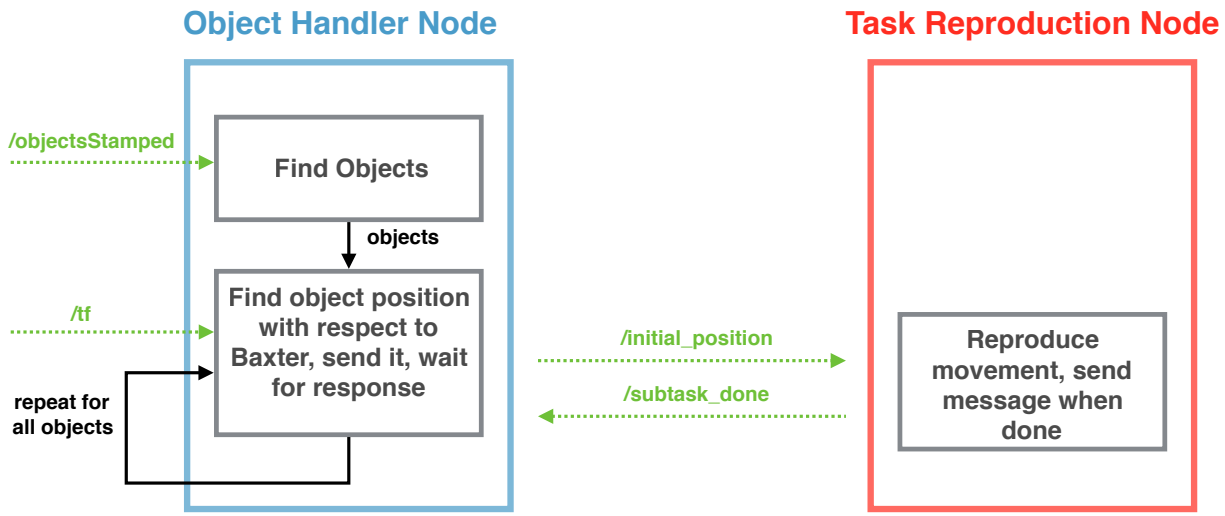

Figure 6.7: Diagram showing two nodes working together for repeated reproduction of a subtask. This program clears the table from all detected objects, by placing them all into a box. The dotted green lines represent topics.

Chapter 7

Testing and Results

The following sections will account for the series of experiments carried out to test the implemented systems. As a result of these tests, some parameters of the underlying implementations were fine tuned to optimise the overall performance. Given the higher degree of uncertainty and variables, these experiments focused primarily on testing the learning part of the project implementation.

7.1 Teleoperation

Figure 7.1 displays the relationship between the inputs generated by monitoring the teleoperator, and the actual response of the robot. Shown here is data for the three dimensions of Baxter's right arm. The data shows a lags ranging around 0.6-0.8 seconds before the desired state is reached, which is the result of the delay generated by inverse kinematic solver (minor component) and by the time it takes for given joint angles to be reached (major component). Interestingly, the teleoperation shows higher accuracy for the y and z axes, compared to the x axis. This is most likely a result of the configurations of Baxter's joints. Horizontal and vertical movements are mostly controlled by one joint each, corresponding respectively to the first two joints from Baxter's shoulder. Moving in the x direction (i.e. in front of Baxter,

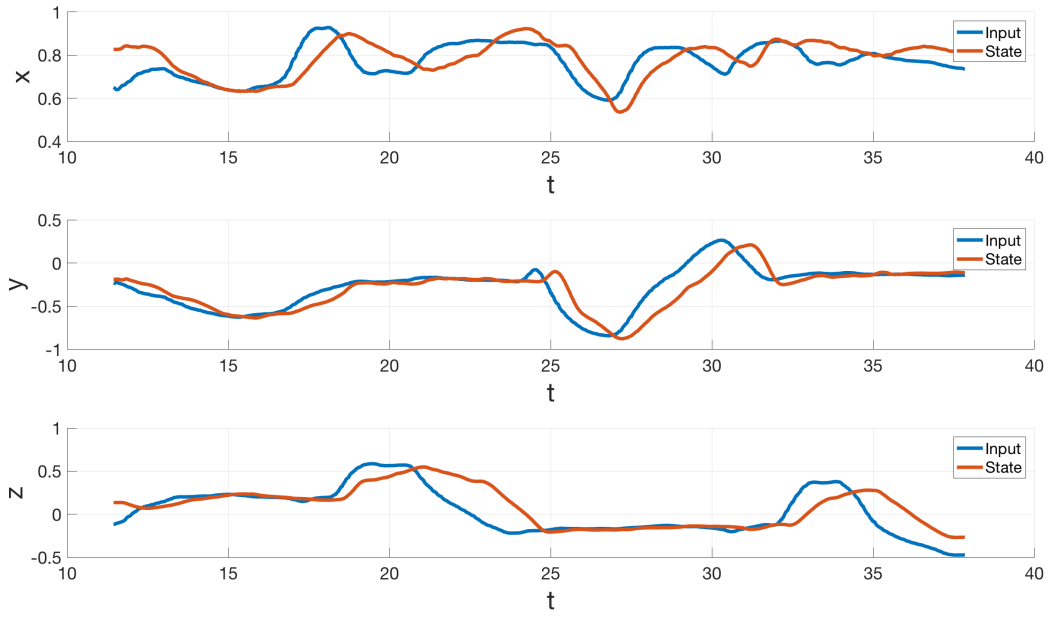

Figure 7.1: Diagram showing the relationship between inputs and actual states for Baxter's Right arm. The inputs are taken as streamed from the VR environments, while the states are read directly from Baxter's joints at the time the input is applied. The data shows a lags ranging around 0.6-0.8 seconds before the desired state is reached. The time axis is displayed in seconds.

away from and towards its torso) involves a combination of 6 out of 7 joints, possibly causing slower responses and lower accuracy.

Similarly, the responsiveness of the head control was tested, and the results are displayed in Figure 7.2. This is a much simpler input-state relationship, with no processing done after the message is streamed from the VR Headset. Indeed, all adjustments and angle calculations are performed in Unity, before streaming a final angle to be applied to Baxter's head, and on the listening ROS node, the message is simply converted into a compatible command. This implementation choice was adopted as an attempt to reduce the visual lag as much as possible, as this type of lag has a much stronger effect on the perceived quality of the teleoperation, and could even cause motion sickness if there is too much discrepancy between the user's movements and his visual feedback [1]. Because of this choice, the head movement lag is reduced to an average of 0.25 seconds, which makes the teleoperation quite

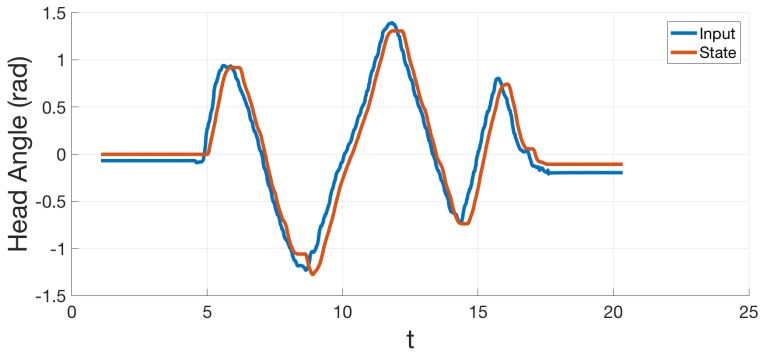

Figure 7.2: Diagram showing the relationship between input and actual state for Baxter's head. The input is taken as streamed from the VR environments, while the state is read directly from Baxter's joints at the time the input is applied. The data shows an average lag of 0.25 seconds. The time axis is displayed in seconds.

comfortable.

7.2 Learning by Demonstration

A series of experiments was conducted to assess the performance of the learning framework. The underlying task always involved grabbing an placing an object, but controlled variations were applied to test the robustness of the learning technique.

For the first experiment, the robot was shown the task of grabbing an placing an object in a box seven times. The initial and final positions were kept constant for these demonstrations, and 6 components were used for GMM encoding. Figure 7.3 displays the results of the learning stage. Because p_o and p_f are constant, all demonstrations resulted in similar values between 300 and 600 milliseconds, as well as after 1000 milliseconds. This characteristic was correctly picked up by the probabilistic encoding (middle column), which shows little uncertainty (variance) in those portions of the model. In turn, this resulted in generalised trajectories that are highly constrained around those regions, as observable on the right column. During reproduction, these target trajectories will be imitated, and the narrow constraints around p_o and p_f will result in weightings that will reward staying as close as

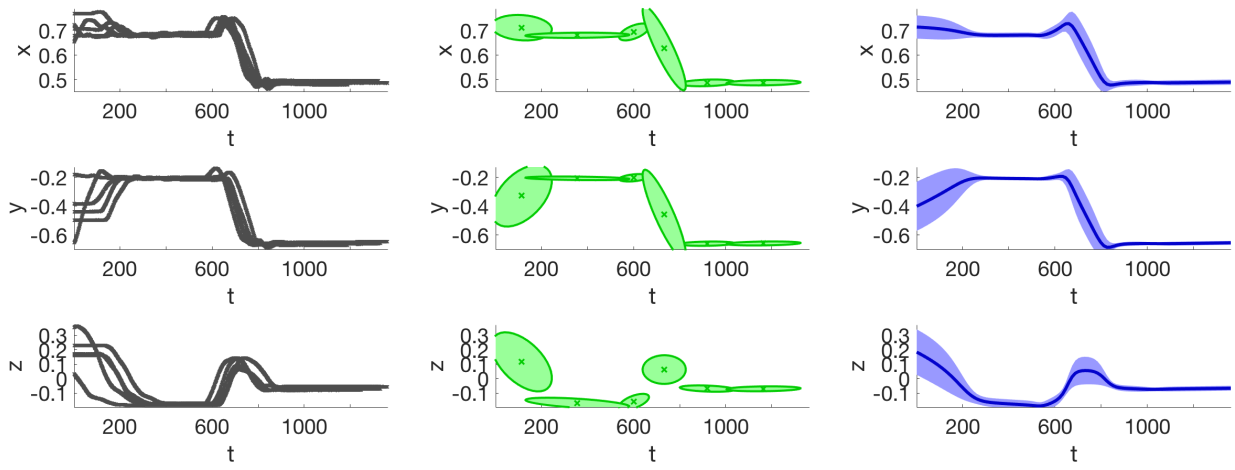

Figure 7.3: Diagram showing the steps of the learning algorithm for the 3D right-hand end-effector position. Time here is shown in milliseconds. In the left column, the data is displayed as recorded. The middle column shows the Gaussian Mixture Model for each components, where the crosses indicate the means μ_k while the ellipses represent the covariance matrices Σ_k. The right column shows the result of generalising the trajectories through Gaussian Mixture regressions, where the dark blue line represents the target trajectory, while the light blue contour represents the trajectory constraints. The time axis is shown in milliseconds

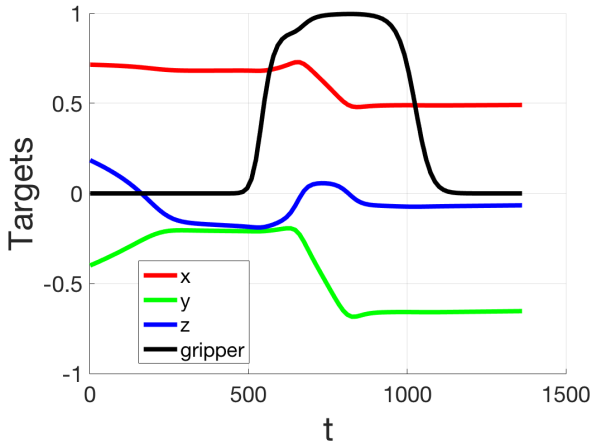

Figure 7.4: Targets for the 3D x_s and the 1D x_g. The gripper correctly learns to close when the end effector is over the object, and open when it is over the box.

possible to the target trajectory. This effect will be even more clear with the next experiment, where the initial position was varied across demonstrations.

Figure 7.4 displays the target trajectory for the gripper state x_g. The target trajectories for x_s are also displayed to show how the algorithm correctly learned to close the gripper when the end effector is over the object, and open when it is over the box. Note that encoding a binary state x_g with a Gaussian Mixture Regression results in smooth transitions. Therefore at reproduction, the gripper is closed when it is currently open and $\hat{x}_g > 0.6$ and it is opened when it is currently closed and $\hat{x}_g < 0.4$.

For the second experiment, the initial position of the object was changed for each of the 8 demonstrations. Given that the objects lie on a table, the x and y values of p_o changed, while the z value remained constant as the hight of the table is fixed. This will reflect on the results.

Figure 7.5 displays the targets obtained through GMR for the state variables x_s, x_o, and x_f. By observing the left column, it is noticeable how the margins around the x and y trajectories have increased compared to the first experiment. This is because different initial conditions resulted in very different arm movements, and only the final part of the demonstrations showed some degree of similarity, because the same final position was reached.

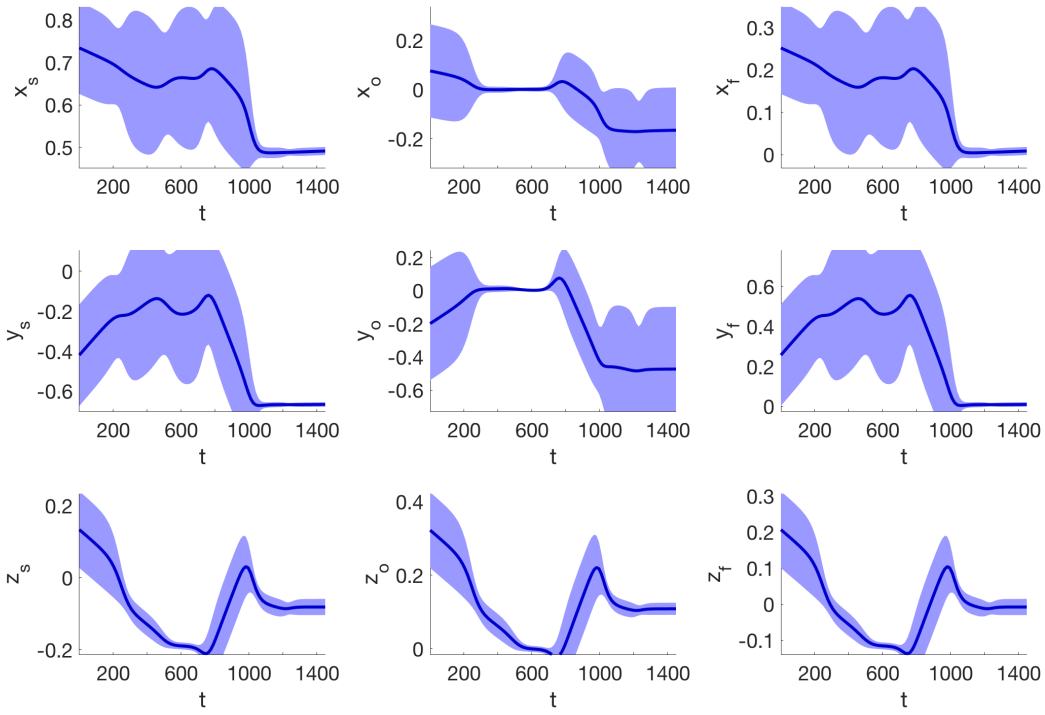

Figure 7.5: Output of the Gaussian Mixture Regression for x_s (left column), x_o (middle column), and x_f (right column). The dark blue line represents the target trajectory, which can be though of as the average of the demonstrations, while the light blue contour represents the trajectory constraints calculated from the Σ_k of the GMM model.

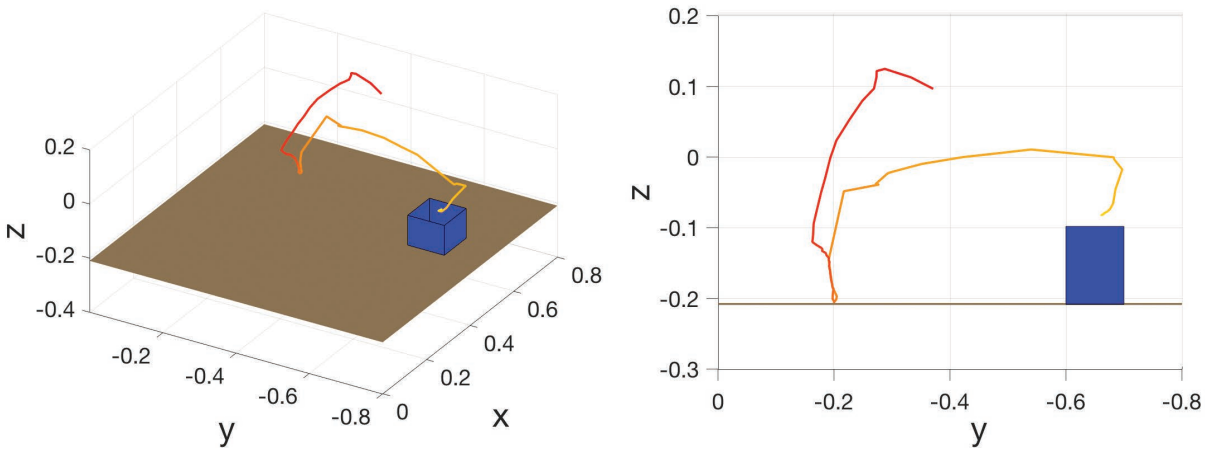

Figure 7.6: A display of the actual trajectory followed during one reproduction by Baxter. The trajectory was obtained by recording data during reproduction, while the table and the box where added manually for reference. The change in colour represents time: from red to yellow.

Given that the final position is constant, $\hat{x}_f$ is expected to simply look like a shifted version of $\hat{x}_s$, so that at the end of the task $\hat{x}_f = [0, 0, 0]^T$. Indeed, this is exactly what is observed here.

On the other hand, different results are expected for the x and y components of $\hat{x}_o$, as they vary for each demonstration. Here it is visible how the algorithm correctly understood the importance that x_o is equal to $[0, 0, 0]^T$ between 400 and 600 milliseconds (hence the narrower constraints), independently of where p_o is. Note that the displayed target trajectories for x_s and x_o are not achievable simultaneously, as they are not shifted versions of each other (recall the constant relationship $x_o = x_s - p_o$). However, it is visible how x_o becomes much more important than x_s right before grabbing the object, hence the controller described in Section 6.9 will read p_o and yield a x_s such that $x_o = [0, 0, 0]^T$.

Figure 7.6 displays the trajectory obtained during one reproduction from different points of views, while Figure 7.7 compares how the robot behaved when the initial positions of its end effector and of the object were changed. As expected, the trajectory adjusts for the new p_o, and successfully brings the object from p_o to p_f.

Figure 7.8 compares the target state $\hat{x}_s$ and the optimal trajectory generated by equation

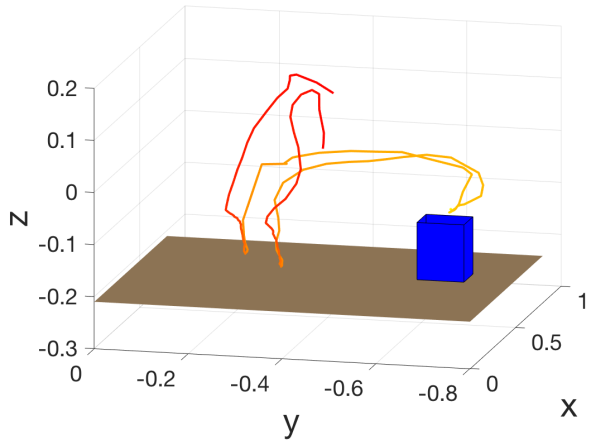

Figure 7.7: A display of the trajectories followed during multiple reproductions (two shown here). The position of the object was changed between demonstrations to observe the effect on the generated optimal trajectory, and assess if the task is still completed correctly. The change in colour represents time: from red to yellow.

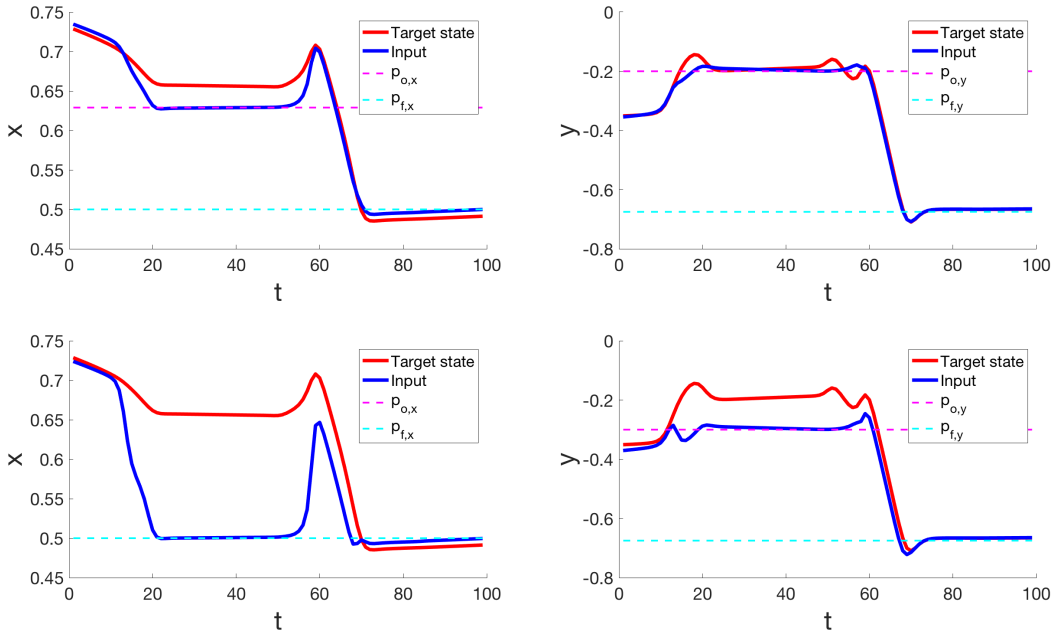

Figure 7.8: A comparison between the target state $\hat{x}_s$ and the optimal trajectory generated during two reproductions (one per row) .

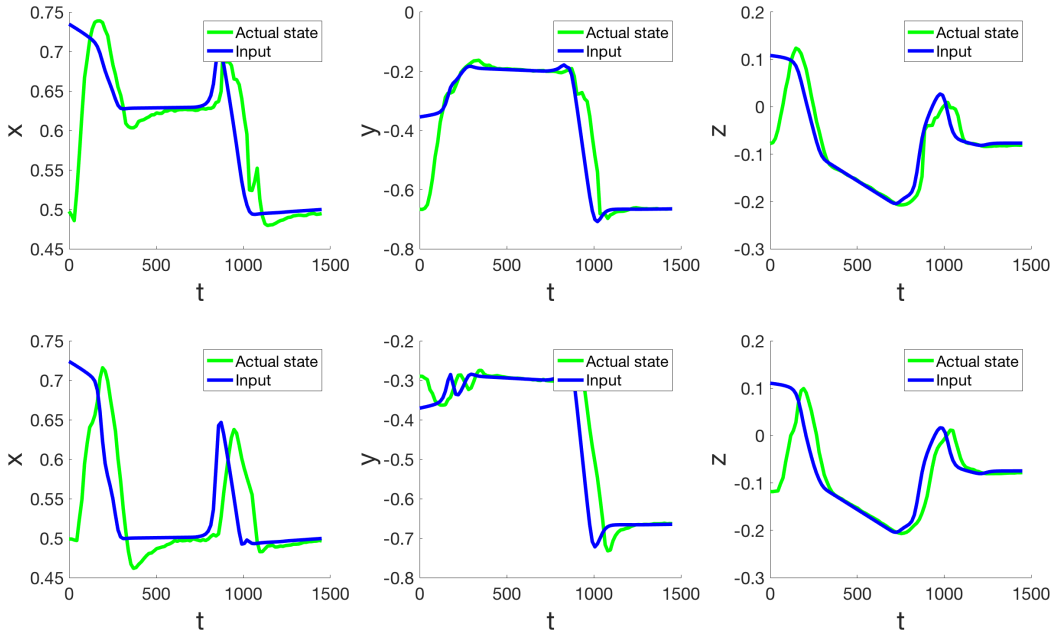

Figure 7.9: A comparison between the input (optimal trajectory) and the actual trajectory followed during two reproductions (one per row). The algorithm selectively chooses which of the $\hat{x}_s$, $\hat{x}_o$, or $\hat{x}_f$ are more important in each portion of the reproduction, to ensure reaching p_o and p_f when appropriate. Because of this, the target state $\hat{x}_s$ is not followed when grabbing and releasing the object.

6.7. The algorithm selectively chooses which of the $\hat{x}_s$, $\hat{x}_o$, or $\hat{x}_f$ are more important in each portion of the reproduction, to ensure reaching p_o and p_f when appropriate. Because of this, the target state $\hat{x}_s$ is not followed when grabbing and releasing the object.

Figure 7.9 shows the relationship between the applied inputs and the actual trajectory followed by the robot as a result. Here it is observable how the inverse kinematic solver is accurately generating joint angles from 3D coordinates, so that the end effector follows the desired trajectory. The imperfections are mostly due to infeasible positions given Baxter's joints, or by inertia in the movements (hence the overshoots). A small lag is also present, which ranges between approximately 60 and 100 milliseconds. This is the time it takes to solve for the joint angles, apply the resulting input, and move to the desired position.

As discussed in Section 6.8, temporal data is not always the best choice of input for

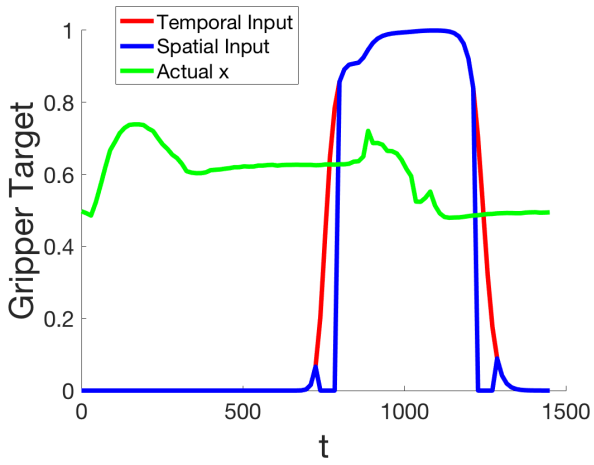

Figure 7.10: A comparison between the gripper state targets when temporal and spatial data are used as inputs for the Gaussian Mixture Regression algorithm. The actual x trajectory is shown in green to help establishing when the end effector was at p_o and p_f.

Gaussian Mixture Regression, such as when attempting to predict a gripper's state. For this case, indeed, it makes more sense to decide to close or open Baxter's grippers based on the position of the end effectors. Figure 7.10 displays a comparison between the target states for the gripper when temporal and spatial values are used as inputs to retrieve it via GMR.

For the spatial input, the current values of x_o and x_f are passed as input to the GMR, as deemed to contain all information relevant to the gripper state. The results however were not accurate until the current x_g was introduced as input as well. Indeed, without x_g the robot could not distinguish between hovering above p_o just before and just after grabbing the object, the grippers would open, and the object would be dropped. After introducing the current x_g as input, the blue target in Figure 7.10 was obtained. The plot shows how the new strategy retains the accuracy of the temporal input almost perfectly. As a result, in all subsequent tests the object was always grabbed and released correctly.

As discussed in Section 6.6, the number of optimal components for the Gaussian Mixture Model cannot be known in advance. An EM algorithm must therefore be applied for a range of K values, after which an information criterion can be applied. Figure 7.11 displays the results of applying both the Akaike Information Criterion and the Bayesian Information

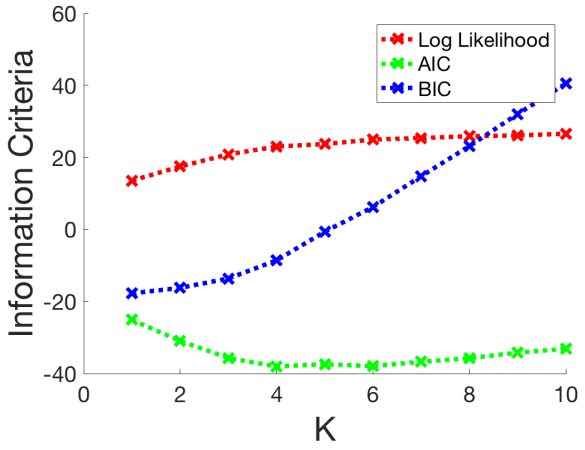

Figure 7.11: Comparison of AIC and BIC for GMMs encoded with a number of components ranging from 1 to 10. The log likelihood of each model is also shown for reference. These results were used to choose the optimal number of components.

Criterion after modelling with K ranging from 1 to 10.

As predicted in Section 6.6, the BIC favours models with a lower number of components compared to AIC. This is because a higher penalty is given for K by the [number of parameters] * log(number of observations) term in BIC. Given the very high number of 11592 observations obtained across 8 demonstrations, the BIC can no longer be considered an accurate measure, as it over-penalises models with many components. Therefore the AIC was taken into account for choosing K.

Figure 7.11 shows that the optimal number of components according to AIC is 6. To reinforce this result, Figure 7.12 displays the GMM and GMR encoding results for the variable x of x_s. Here it is observable how for $k > 5$ the generalised trajectory assumes the right shape, showing the desired behaviours at p_o and p_f. Not shown here, similar analysis was performed for the other dimensions, and eventually 6 was confirmed as the optimal number of components.

Figures 7.13 and 7.14 display the results of the tests performed to assess the timing at reproduction. The first image shows the maximum rate at which the reproduction algorithm can generate inputs for the robotic arms, given the time it takes to calculate such input

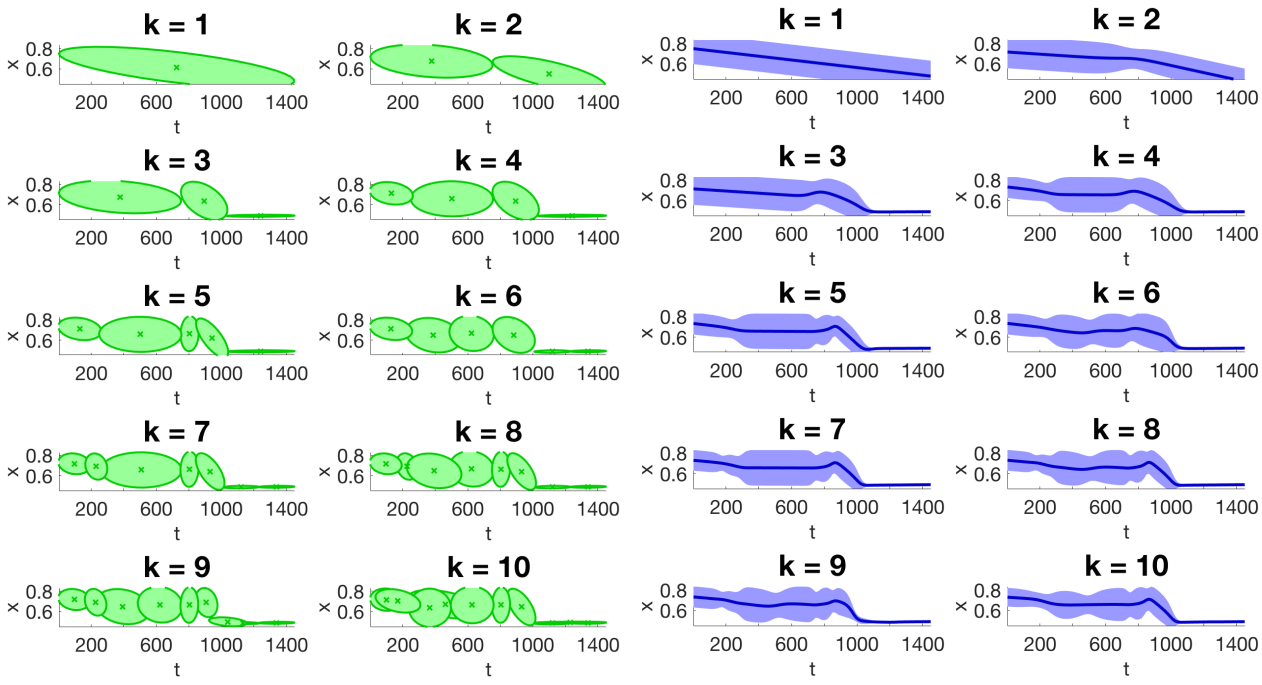

Figure 7.12: A plot displaying the results of GMM (left) and GMR (right) for model components ranging from 1 to 10 (only the x component of x_s is shown). By comparing these signals with the results shown in Figure 7.11, one can discard BIC as a reliable measure, as choosing a small K does not yield an accurate trajectory. This is most likely due to the high number of observations (11592) used, which causes BIC to overly penalise higher K values. On the on the other hand, AIC results are confirmed here, with optimal values ranging around K values between 4 and 6.

at each iteration. The data shows the generation of 100 input sequences, where most of the time inputs could be generated at 70-80Hz, while sometimes the computation can slow down the input generation to as low as 10Hz. Figure 7.14 instead displays the timing for the subcomponents of the above iterations. It is clear how the most timely expensive component is acquiring a reading of the current joint states from the robotic arms. Since the current state is only used to obtain the gripper target trajectory, reverting to using only temporal inputs in GMR could speed up the input generation to up to 6 times faster.

Throughout testing, a rate of 10Hz was used as default, which resulted in visibly smooth reproduction trajectories. Moreover, the reproduction algorithm was built to be robust against fluctuations in input generations, so that it never exceeds 10Hz while slowing down

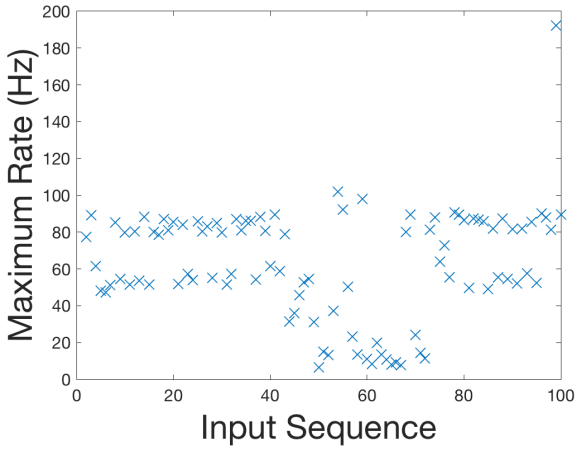

Figure 7.13: A diagram showing the maximum input generation rate achievable. The data was obtained timing how long it takes to perform each iteration of optimal trajectory generation at reproduction, for 100 inputs. We can see here that for most of the time inputs could be generated at 70-80Hz, however sometimes the computation can slow down the input generation to as low as 10Hz.

to the appropriate rate whenever the input generation required longer than 100 milliseconds.

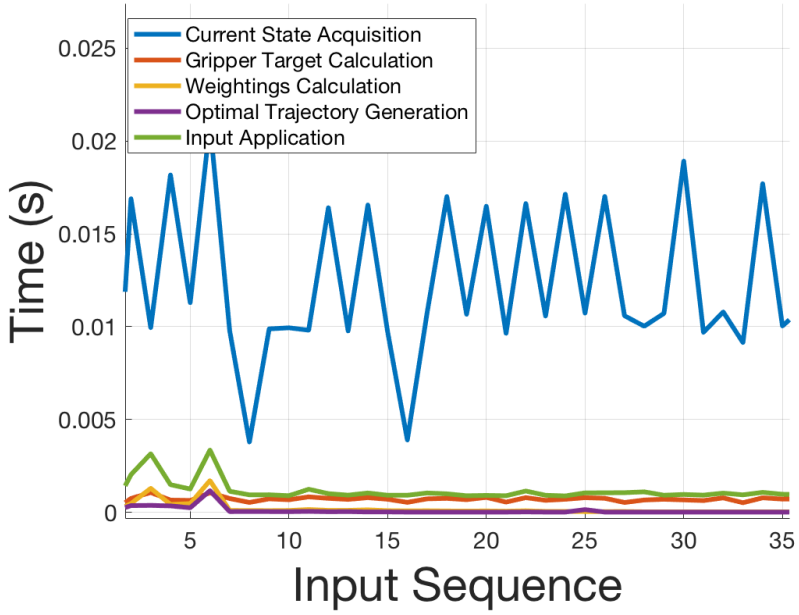

Figure 7.14: A diagram showing the time it takes to perform the 5 main sub-components of reproduction: acquiring the current state, recalculating the target trajectory for the gripper based on spatial information, obtaining the weightings for the cost function, calculating the optimal 3D hand trajectory, and sending the complete input over the ROS network. Notably, acquiring the current state is the slowest process. Since the current state is only used to obtain the gripper target trajectory, reverting to using only time-inputs in GMR could speed up the input generation to up to 6 times faster.

Chapter 8

Evaluation

This project introduced a learning framework in which a robot is teleoperated to provide demonstrations of a task, which the machine can learn and reproduce by extracting key aspects of the task and generating new, context-dependent trajectories to replicate all of them.

For the teleoperation part, all prerequisites listed in the requirements capture were carried forward. A successful bridge between Baxter and the VR environment was created, which allowed to stream over commands from the HTC Vive to ROS nodes controlling the robot and vice versa. With the presented system, an operator can wear the HTC Vive's headset and hold two controllers to take the robot's point of view, seeing what the robot sees, and moving Baxter's limbs as if they were his own arms. A series of scripts takes care of streaming a video originated on Baxter's head to the teleoperator with minimal lag, and of sending over the user's head orientation, hands position and hands orientation.

The latter three messages are analysed by ROS nodes which transform these signals into joint commands, and apply them to Baxter to mirror the user's movements. An Inverse Kinematic solver transforms the hands pose (position and orientation) into joint angles to control the 7-DOF robotic arms, whenever a feasible solution is found. Because Baxter does not have human-like wrists, controlling hand orientation is a non-trivial task. A small

wrist movement by the operator might need extensive joint variations to maintain the same end-effector position while changing its orientation.

Experiments have shown that with the system in place, this aspect could cause lags in the arm's movements and an overall loss of smoothness and responsiveness from the robot, making teleoperation less comfortable. To cope with this limitation, a button on the HTC Vive's controller is configured to toggle wrist control. When deactivated, Baxter wrists always assume an orientation such that its grippers point downwards, which is the most suitable pose to grab objects. Having this extra control option, the operator can chose to switch between a more fluid teleoperation and a more precise one at any point during task demonstration.

One more limitation of the presented system is the absence of a stereo video input. The Xtion camera used for this project was chosen as it can acquire depth information, necessary for automatic object recognition. However, it only has a mono RGB camera, which is used to let the operator see from Baxter's point of view. This mono feed makes it challenging for the operator to perceive depth, posing a challenge for demonstrations involving grabbing objects. A stereo feed would ease the teleoperation, as discussed in more detail in Section 10.

For the learning by demonstration part, the task of grabbing an object from one place and carrying it to another was considered. This task was chosen as it encompasses characteristics that are common in a wider number of jobs, as it involves an initial, an intermediate, and a final state, automatic object recognition, and handling of the recognised objects. The results obtained from this experiment can therefore easily be adapted to other tasks, most of which would not even require to change the parameters considered in the presented experiments.

The cost function used in the experiments attempts to find an optimal trajectory that tries to match the demonstrated hand paths x_s while prioritising the relationship between the end effector and the objects (x_o, x_f) for certain portions of the reproduction. Depending on the portion of the task, different variables have different importance, which is extracted automatically from the demonstrations by a human teacher. For the task of grabbing and

placing an object, x_o shows the highest importance just before an object is grabbed by the robot, while x_f shows the highest importance at the end of the trajectory, before releasing the object. This concept can be expanded to a more general series of tasks, where the same variables will show different importance at different times, depending on the task at hand.

Furthermore a separate controller monitors the activity of the robot's hand grippers, to determine when it is necessary to open and closed them. Contrary to what observed in previous works, where temporal inputs control all aspects of the reproduction, the presented implementation uses spatial information to control the gripper state. This way, the full versatility of GMR encoding is exploited by the presented implementation. The gripper was used as a way to demonstrate this feature, but the same concept could be applied to other variables depending on the task at hand.

The presented system deals with task generalisation by:

1. Aligning different demonstrations of the same task in time, to group key task features in specific time slots
2. Encoding demonstration data into a GMM, which accounts for the variations in behaviours across demonstrations
3. Exploiting the variation to extract information about the time-varying relevance of each subset of recorded data for the demonstrated task
4. Generating an optimal signal from such obtained information, to maximise the likelihood of reproducing the task correctly

This system is robust against considerable variations during demonstrations. For example, different initial conditions during demonstration for the hand position or the object position resulted in substantially different x_s recordings, but the implemented system was able to interpret these variations as portions where x_s was not relevant to complete the task, and focused instead on x_o when attempting to generate an optimal trajectory. In-

deed, throughout the testing phase the task has consistently been reproduced correctly, for a variety of initial conditions.

Throughout the experiments, it was assumed that 3D positioning information was sufficient to describe a task. This may not always be correct, as other factors such as speed and force could be crucial for other types of tasks. However, the implemented system merely accetps a list of state variables coupled with timing information, and it can therefore already perform learning involving other types of data other than 3D positions. After GMR, however, new cost functions and trajectory controllers would have to be designed for problems involving speed, force, or any other similar state type.

The system presents some limitations, mainly represented by the strong influence of timing. Indeed, for most of the reproduction, a time sequence is generated and used as input to retrieve the target state for all other variables through GMR. This is quite a robust strategy, since time is inherently linear and invariant independently of the task, and it is therefore very intuitive to use this variable as a "known-a-priori" element of the reproduction. However, this also means that the controller will try to reproduce a task always within the same time frame. Unfortunately, there exist a set of initial conditions under which the allowed time might not be enough to reproduce the task correctly, and the robot might skip some crucial steps. In other works, this issue is resolved by adopting Hidden Markov Models as means of temporal alignment, and removing the temporal component altogether. However as discussed, HMMs often result in less accurate generalised trajectories, presenting discontinuities or other non-smooth behaviour. Therefore in this work, this issue was tackled by simply allowing more time for a given task to be reproduced. This means that under most initial conditions it might take longer than needed to reproduce a task, but it also ensures a very high success rate.

Chapter 9

Conclusion

The initial objectives of the project were successfully achieved, and this work provided a system where a robot is teleoperated via VR hardware, and tasks are learned after demonstration. For the teleoperation, this system allows to see from the robot's point of view via a head camera, and to control the robots head an hands by directly mapping the operator's movements onto the robot. For the learning by demonstration part, the acquired data from the demonstrations is processed to extract all important features of a task, and reproduce the job in a new context using a metric that evaluates the accuracy of the imitation.

A series of experiments have validated the proposed method, highlighting its strengths and drawbacks, but ultimately demonstrating high performance and versatility. The learning method was heavily inspired by combinations of previous works, but also introduced some novel techniques which contributed to the overall robustness of the system.

Chapter 10

Further Work

10.1 Camera Distortion

With the system in place, no processing is done to the robot's camera feed before displaying it on the HMD. Since every camera presents some level of distortion due to its lens, a more realistic display could be achieved by correcting for these imperfections. The transition to the Xtion camera already removed the visible distortion presented by the built in Baxter head camera, but some artefacts are still noticeable at the edges of the video stream. To obviate for this factor, the distortion parameters of th Xtion camera could be used to modify the plane onto which the image is streamed. Instead of using a flat surface, information about the lens' focal distance and optical axis could be extracted to turn it into a curved shape, so that from the user's point of view no distortion is visible.

10.2 Stereo Vision

As discussed in Section 8, only a mono video was used to stream Baxter's point of view to the HTC Vive. This aspect makes it challenging for a human operator to perceive depth. The disparity of a stereo vision system (like the ones humans have) is instead interpreted by

human brains to determine how far objects are, and would therefore be much more suitable for teleoperation than a mono feed. During this project, a Videre Design stereo camera was considered for this task. However, a number of software and hardware incompatibilities made it challenging to utilise this camera. Moreover, a working system (including correct object recognition) was in place and the Videre Design camera was eventually dropped, as a considerable amount of time would have had to be spent to interface this camera with ROS and to implement an accurate object recognition and camera calibration, necessary to obtain object positions in Baxter's frame of reference.

10.3 Reduction of Dimensionality

Given the strong relationship between x_s, x_o and x_f, some of the information encoded in the GMM is redundant. In machine learning, there exist techniques to reduce the dimensionality of a dataset before using it for model training. Two popular ones are Factor Analysis and Principal Component Analysis [32].

These methods map a set of M dimensions to a space of dimension $D < M$ by projecting each datapoint in the M-dimensional space to a D-dimensional latent space. This is done by choosing the axis of the latent space as to maximise the variance of the projections [32].

After this reduction, the new latent dataset encapsulates roughly the same amount of information, but in a more compact way. The benefits of doing so lie primarily in faster training of the GMM, as well as faster regression through GMR.

Chapter 11

User Guide

This section will provide instructions on how to use the delivered system. Since the Teleoperation adn Learning parts are entirely independent, the guide is also subdivided in these two components.

11.1 Code

11.1.1 ROS Programs

The following programs running on ROS can be found inside a ROS package called *teleoperate_baxter*:

- Scripts:
  - `head_movement.py`: Initialises a node that listens to topic $teleoperator/head\_movement$ for head angles, and applies them to Baxter's neck.
  - `right_movemebt.py`: Initialises a node that listens to topic $teleoperator/right\_movement$ 3D positions and 4D orientations, and applies them to Baxter's right end effector.
  - `left_movement.py`: Initialises a node that listens to topic $teleoperator/left\_movement$ 3D positions and 4D orientations, and applies them to Baxter's left end effector.

- `grippers.py`: Initialises a node that listens to topic *teleoperator/gripper_movement* for gripper command (open/close gripper), and applies them to Baxter's right and left-hand grippers.
- `data_acquisition.py`: This program is used to record the demonstrations. It has one mandatory parameter, a comma-separated-value file name, and an optional one, the recording rate (default is 100Hz). The program queries for an initial position p_o and a final position p_f. It then proceeds to record x_t, x_s, x_g, p_o, and p_f at the specified rate to the specified CSV file.

- Nodes:
  - `clear_all_objects_matlab`: This program is used in pair with a task reproduction program to remove all recognised objects from a table, and put them in a box. The program accesses information from a depth camera to obtain a list of objects and store their positions. These positions are then streamed over a topic called *initial_position* to a separate program, which performs the task. A message is returned over the topic */subtask_done* after the object was placed in the box. This process is repeated for all recognised objects. `clear_all_objects_matlab` should be run after the paired task-reproduction program is ready to accept object positions.

- Launch files (called via `roslaunch`):
  - `openni.launch`: This program initialises all necessary software to operate OpenNI-compatible depth cameras. One such camera should be connected via USB before running this launchfile. This program will publish a list of topics under the namespace */camera*. These include colour and depth information, and are used for teleoperation and object recognition.
  - `find_object_3d.launch`: This launch file should be launched after `openni.launch`, and it initialises all necessary object recognition software. It automatically looks for previously saved objects through the depth camera, and publishes their position relative to the camera on $/tf$. This launchfile has a default camera calibration

to transform object positions in Baxter's frame of reference, which can be edited as needed. An optional parameter `gui:=true` can be applied when launching, to enable the graphical user interface. This tool can be used to visualise the recognised objects, add new one, and change the default directory for the stored objects.

- `receivers.launch`: This program simply runs `head_movement.py`, `right_movement.py`, `left_movement.py`, and `grippers.py` at once, in order to save time.
- `teleoperate.launch`: This program simply launches all previously listed launch files in one command.

11.1.2 Matlab Programs

The following programs running on Matlab were delivered:

- `demo`: This script shows an example usage of most other programs listed here, demonstrating the complete chain of steps taken from data acquisition to task reproduction.
- `temporalAlignment`: This function performs the temporal alignment through Dynamic Time Warping. For more details see Sections 2.11 and 6.4.
- `datasetCompletion`: This function intakes temporally-aligned demonstrations and expands their dataset to include x_o and x_f: the 3D components representing the hand-object relationships.
- `learnGMM`: This function performs the training of the GMM model through a k-means clustering parameter initialisation, followed by an Expectation-Maximisation algorithm.
- `reproduceTaskManually`: This function demonstrates the steps taken for reproduction, including Gaussian Mixture Regression, cost function calculation and optimal trajectory generation. This function connects to Baxter's ROS network, and utilises

a number of topics to obtain the current state of Baxter's limbs as well as to send commands.

- `plot_GMM_GMR`: This scripts demonstrates the results f GMM encoding and GMR by plotting their outputs.

11.1.3 Unity Programs

The following C-sharp scripts are provided in Unity, inside the Unity project *BaxterTeleOp*:

- `ROSCamera`: This script subscribes to ROS topic */camera/rgb/image_color/compressed* to obtain video frames from the Xtion camera mounted on Baxter's head. It then applies these frames as a texture (one per rendered VR frame) to an object in the scene, so that it appears to the user as a video stream played on an object
- `FullScreenPlane`: This script takes the object to which `ROSCamera` applies the video stream, and transforms it as to cover the entire user's field of view. This way the user can only see what is streamed through Baxter's camera, and perceives Baxter's point of view.
- `HeadPoseStreamer`: This script latches onto the VR headset and obtains its orientation. It then streams this information over the topic */teleoperator/head_movement*.
- `RightHandPoseStreamer` and `LeftHandPoseStreamer`: These scripts latch onto the HTC Vive's controllers, and obtains their poses (position and orientation). They then stream this information over the topic */teleoperator/right_movement* and */teleoperator/left_movement*. Controller buttons can be used to turn on and off some features of teleoperation.
- `RightGripperController` and `RightGripperController`: These scripts stream open/-close gripper commands for both Baxter arms over the topic */teleoperator/gripper_movement* whenever a controller's Trigger button is pressed/released.

11.2 Teleoperation

A project called *BaxterTeleOp* was saved on the Windows machine connected to the VR headset in the Personal Robotics Lab. This file comes with a list of objects (called a "scene" in Unity) which are used during the teleoperation for a number of functions.

To correctly configure the Windows machine to communicate with the ROS network, the following environmental variables should be set:

- ROS_MASTER_URI = http://011401P0008.prl:11311 (Baxter)
- $ROS\_HOSTNAME$ = ViveMAGIC.prl

BaxterTeleOp comes with the full list of scripts described above, which are already applied to the appropriate objects in the scene. The project can be played as is. However, the appropriate nodes should be run on ROS for correct functioning. The quickest way to do this is to run two launch files (in any order) on ROS:

1. `openni.launch`: This program initialises all necessary software to operate OpenNI-compatible depth cameras, which should be connected via USB before running this launchfile.
2. `receivers.launch`: This program runs all the nodes that subscribe to the topics that are published in Unity, and convert position/orientation information into appropriate baxter commands.

Once everything is running, the user can wear the VR headset to acquire Baxter's point of view. The robot's head should already react to the user's head movements. The Vive's controllers have grip buttons which activate/deactivate arm teleoperation. When activated, the trigger buttons can be used to open/close Baxter's grippers. Baxter's grippers will point downards by default. If necessary, wrist orientation can be activated through the

touchpad button. It should be noted however that Baxter lacks a proper wrist, and complex movements might be required to reproduce small user wrist changes, which might deteriorate the performance of the teleoperation.

11.3 Learning by Demonstration

11.3.1 Data Acquisition

A ROS script called `data_acquisition.py` was provided to record the demonstrations as explained in Section 6.3. This program has one mandatory parameter, a comma-separated-value file name, and an optional one, the recording rate (default is 100Hz). The program queries for an initial position p_o and a final position p_f. It then proceeds to record x_t, x_s, x_g, p_o, and p_f at the specified rate to the specified CSV file.

The demonstrations should be saved onto CSV files named `1.csv`, `2.csv`, `3.csv` and so on, and saved into the same folder.

IMPORTANT NOTE: This program records robot states, not inputs. Because of this, demonstration does not necessarily need to be done through teleoperation, but could for example be achieved by manually dragging Baxter's limbs in zero-gravity mode.

11.3.2 Learning

Such acquired data must undergo a series of steps before it can be used to train a GMM. Please refer to the Matlab script `demo.py` for an example of how to perform these steps.

The first step consists in temporally aligning the signals, which can be done trhough the provided Matlab function `temporallAlignment`. This function takes as argument the path to the folder containing the CSV files with the demonstrations, along with the number of demonstrations provided. It then performs Dynamic Time Warping of the signals, returning

the aligned data, along with variables containing the list of initial and final object positions for all N demonstrations.

For the second step, the dataset is expanded to include x_o and x_f. This is done automatically by the provided Matlab function `datasetCompletion`, which takes the aligned data and the initial/final positions as inputs, and returns the complete dataset, with the correct format for GMM training.

This dataset can thus be passed to the `learnGMM` function for model training, after specifying the number of Gaussian components for the model.

11.3.3 Reproduction

An example for reproduction is provided by the Matlab function `reproduceTaskManually`, which requires to manually specify the desired initial and final position.

For before running, the launch file `receivers.launch` must be running on ROS, for the generated trajectories to be applied to Baxter.

Any reproduction algorithm should contain at least the following steps:

1. Initialise the ROS environment, by setting the correct environment variables, registering on the network, and preparing the relevan ROS publishers and subscribers.
2. Load the desired GMM model (previously saved during training).
3. Evaluate the target states from the GMM through GMR.
4. For each generated Baxter input:
   (a) Obtain the robot's current state (optional).
   (b) Re-calculate gripper targets based on spatial information (optional).
   (c) Obtain the weightings for the cost function.

(d) Obtain the ideal trajectory.

(e) Publish the appropriate inputs on the ROS network.

5. Shut down ROS.

For reproduction routines that wish to automatically detect objects in front of Baxter, two additional launch files must be run: `openni.launch` and `find_object_3d.launch`. The former must have previously been configured to correctly position objects into Baxter's frame of reference. The latter must have been configured (by enabling the user interface through the `gui` parameter) so that the objects to be recognised are stored somewhere, and the correct path is set to automatically retrieve information about these stored objects.

Bibliography

[1] Lawrence J Hettinger and Gary E Riccio. "Visually induced motion sickness in virtual environments". In: *Presence: Teleoperators & Virtual Environments* 1.3 (1992), pp. 306–310.

[2] Terrence Fong, Illah Nourbakhsh, and Kerstin Dautenhahn. "A survey of socially interactive robots". In: *Robotics and autonomous systems* 42.3 (2003), pp. 143–166.

[3] Volker Klingspor, John Demiris, and Michael Kaiser. "Human-robot communication and machine learning". In: *Applied Artificial Intelligence* 11.7 (1997), pp. 719–746.

[4] Candace L Sidner et al. "Explorations in engagement for humans and robots". In: *Artificial Intelligence* 166.1-2 (2005), pp. 140–164.

[5] *Baxter — Redefining Robotics and Manufacturing*. URL: `http://www.rethinkrobotics.com/baxter/` (visited on 16/10/2016).

[6] *Baxter — Inverse Kinematics Solver*. URL: `http://sdk.rethinkrobotics.com/wiki/API_Reference#Inverse_Kinematics_Solver_Service` (visited on 16/10/2016).

[7] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe. 2009, p. 5.

[8] Kazuhiko Kawamura et al. "Humanoids: Future robots for home and factory". In: *International symposium on humanoid robots*. 1996, pp. 53–62.

[9] Sung-Kyun Kim, Seokmin Hong, and Doik Kim. "A walking motion imitation framework of a humanoid robot by human walking recognition from IMU motion data". In:

Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on. IEEE. 2009, pp. 343–348.

[10] Nathan Miller et al. "Motion capture from inertial sensing for untethered humanoid teleoperation". In: *Humanoid Robots, 2004 4th IEEE/RAS International Conference on*. Vol. 2. IEEE. 2004, pp. 547–565.

[11] Shuji Hashimoto et al. "Humanoid robots in Waseda universityHadaly-2 and WABIAN". In: *Autonomous Robots* 12.1 (2002), pp. 25–38.

[12] H Song et al. "Tele-operation between human and robot arm using wearable electronic device". In: *17th IFAC World Congress*. 2008, pp. 2430–2435.

[13] Young-Su Cha et al. "MAHRU-M: A mobile humanoid robot platform based on a dual-network control system and coordinated task execution". In: *Robotics and Autonomous Systems* 59.6 (2011), pp. 354–366.

[14] Hitoshi Hasunuma et al. "A tele-operated humanoid robot drives a lift truck". In: *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. Vol. 3. IEEE. 2002, pp. 2246–2252.

[15] Hitoshi Hasunuma et al. "A tele-operated humanoid robot drives a backhoe". In: *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*. Vol. 3. IEEE. 2003, pp. 2998–3004.

[16] Lu Lu and John T Wen. "Human-robot cooperative control for mobility impaired individuals". In: *American Control Conference (ACC), 2015*. IEEE. 2015, pp. 447–452.

[17] H Reddivari et al. "Teleoperation control of baxter robot using body motion tracking". In: *Multisensor Fusion and Information Integration for Intelligent Systems (MFI), 2014 International Conference on*. IEEE. 2014, pp. 1–6.

[18] *Watch a Real Robotic Arm Controlled with Virtual Reality*. URL: `http://motherboard.vice.com/read/watch-a-real-robotic-arm-controlled-with-virtual-reality` (visited on 05/11/2016).

[19] *IVRE An Immersive Virtual Robotics Environment*. May 2016. URL: http://cirl.lcsr.jhu.edu/research/%20human-machine-collaborative-systems/ivre/ (visited on 05/11/2016).

[20] Lorenzo Peppoloni et al. "Augmented reality-aided tele-presence system for robot manipulation in industrial manufacturing". In: *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology*. ACM. 2015, pp. 237–240.

[21] Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[22] Miles N Wernick et al. "Machine learning in medical imaging". In: *IEEE signal processing magazine* 27.4 (2010), pp. 25–38.

[23] Jun Nakanishi et al. "Learning from demonstration and adaptation of biped locomotion". In: *Robotics and Autonomous Systems* 47.2 (2004), pp. 79–91.

[24] Brenna D Argall et al. "A survey of robot learning from demonstration". In: *Robotics and autonomous systems* 57.5 (2009), pp. 469–483.

[25] Andrew Y Ng et al. "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[26] Brett Browning, Ling Xu, and Manuela Veloso. "Skill acquisition and use for a dynamically-balancing soccer robot". In: *AAAI*. 2004, pp. 599–604.

[27] John D Sweeney and Rod Grupen. "A model of shared grasp affordances from demonstration". In: *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*. IEEE. 2007, pp. 27–35.

[28] William Donald Smart. "Making reinforcement learning work on real robots". PhD thesis. Brown University, 2002.

[29] Matthew Field et al. "Learning trajectories for robot programing by demonstration using a coordinated mixture of factor analyzers". In: *IEEE transactions on cybernetics* 46.3 (2016), pp. 706–717.

[30] John Aldrich et al. "RA Fisher and the making of maximum likelihood 1912-1922". In: *Statistical Science* 12.3 (1997), pp. 162–176.

[31] Arthur P Dempster. "Maxmum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal Statistical Society, B* 39 (1977), pp. 1–38.

[32] Andrew NG. *Machine Learning*. Stanford University Lecture. 2014.

[33] Daniel Ramage. "Hidden Markov models fundamentals". In: *Lecture Notes. http://cs229. stanford. edu/section/cs229-hmm. pdf* (2007).

[34] Geir E Hovland, Pavan Sikka, and Brenan J McCarragher. "Skill acquisition from human demonstration using a hidden markov model". In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*. Vol. 3. Ieee. 1996, pp. 2706–2711.

[35] Jie Yang, Yangsheng Xu, and Chiou S Chen. "Human action learning via hidden Markov model". In: *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 27.1 (1997), pp. 34–44.

[36] Kyuhwa Lee et al. "A syntactic approach to robot imitation learning using probabilistic activity grammars". In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1323–1334.

[37] Harold Soh and Yiannis Demiris. "Learning Assistance by Demonstration: Smart Mobility With Shared Control and Paired Haptic Controllers". In: *Journal of Human-Robot Interaction* 4.3 (2015), pp. 76–100.

[38] Donald J Berndt and James Clifford. "Using dynamic time warping to find patterns in time series." In: *KDD workshop*. Vol. 10. 16. Seattle, WA. 1994, pp. 359–370.

[39] Timo Koski. "Multivariate Gaussian Distribution". In: *Auxiliary notes for Time Series Analysis SF2943, Spring* (2013).

[40] Sylvain Calinon et al. "Learning and reproduction of gestures by imitation". In: *IEEE Robotics & Automation Magazine* 17.2 (2010), pp. 44–54.

[41] Jacopo Aleotti and Stefano Caselli. "Robust trajectory learning and approximation for robot programming by demonstration". In: *Robotics and Autonomous Systems* 54.5 (2006), pp. 409–413.

[42] Aleksandar Vakanski et al. “Trajectory learning for robot programming by demonstration using hidden Markov model and dynamic time warping”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42.4 (2012), pp. 1039–1052.

[43] Hsi Guang Sung. “Gaussian mixture regression and classification”. PhD thesis. Rice University, 2004.

[44] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. “Active learning with statistical models”. In: *Journal of artificial intelligence research* 4.1 (1996), pp. 129–145.

[45] S. Calinon, F. Guenter, and A. Billard. “On Learning, Representing and Generalizing a Task in a Humanoid Robot”. In: *IEEE Transactions on Systems, Man and Cybernetics, Part B* 37.2 (2007), pp. 286–298.

[46] S. Calinon. *Robot Programming by Demonstration: A Probabilistic Approach*. EPFL Press ISBN 978-2-940222-31-5, CRC Press ISBN 978-1-4398-0867-2. EPFL/CRC Press, 2009.

[47] Thomas Cederborg et al. “Incremental local online gaussian mixture regression for imitation learning of multiple tasks”. In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. 2010, pp. 267–274.

[48] Eric McCann. *UML-Robotics/ROS.NET*. Sept. 2016. URL: `https://github.com/uml-robotics/ROS.NET` (visited on 18/10/2016).

[49] *Openni Camera*. URL: `http://wiki.ros.org/openni_camera` (visited on 09/05/2017).

[50] *Openni Launch*. URL: `http://wiki.ros.org/openni_launch` (visited on 09/05/2017).

[51] Tetsunari Inamura et al. “Intent imitation using wearable motion capturing system with on-line teaching of task attention”. In: *Humanoid Robots, 2005 5th IEEE-RAS International Conference on*. IEEE. 2005, pp. 469–474.

[52] Leonel Rozo et al. “Learning physical collaborative robot behaviors from human demonstrations”. In: *IEEE Transactions on Robotics* 32.3 (2016), pp. 513–527.

[53] Torrey Botanical Club. “Akaike, H. 1973. Information theory and an extension of the maximum likelihood principle. Pp. 267-281 in 2nd International Symposium on Information Theory, Tsahkadsor, Armenia, USSR, September 2-8, 1971, eds. BN Petrov and F. Cski. Budapest: Akadmiai Kiad. Bankevich, A., S. Nurk, D. Antipov, AA Gurevich, M. Dvorkin, AS Kulikov, VM Lesin, et al. 2012. SPAdes: A new genome assembly algorithm and its applications to single”. In: *Evolution, Ecology and Population Biology* 1001.1 (2016), p. 139.

[54] Gideon Schwarz et al. “Estimating the dimension of a model”. In: *The annals of statistics* 6.2 (1978), pp. 461–464.

[55] David R Anderson and Kenneth P Burnham. “Avoiding pitfalls when using information-theoretic methods”. In: *The Journal of Wildlife Management* (2002), pp. 912–918.

[56] *FindObject2D*. URL: `http://wiki.ros.org/find_object_2d` (visited on 15/05/2017).