Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2019

Project Title:	**Machine Learning for Robot Programming using Virtual Reality Headsets**
Student:	**Karoly T. Horvath**
CID:	**01088730**
Course:	**EEE4**
Project Supervisor:	**Professor Yiannis K. Demiris**
Second Marker:	**Dr Tae-Kyun Kim**

Acknowledgements

I would like to express my special thanks and gratitude to my supervisor for this Master's project - Professor Yiannis Demiris - for his support, valuable advice and providing me with the opportunity to work on this project. His encouragement and feedback kept me motivated to achieve the best outcome for this work.

Additionally, I would like to highlight my gratefulness for the PhD students and Research Assistants at the Personal Robotics Lab at Imperial College London for their friendliness, for letting me join the Lab for the last couple of months, and for providing me with the best practical advice. Especially, I am thankful for the help I received from Vinicius Schettino concerning the HTC Vive and the Virtual Reality part of the project, Rodrigo Chacón Quesada for his help with Unity, Fan Zhang for advising me with all concerns regarding the Baxter Research Robot and Daniel Varnai for his general advice and helping me to set up ROS#. In addition I would like to say special thanks to Filippo Baldini, whose fantastic Master's thesis titled *Machine Learning for Humanoid Robot Programming through Virtual Reality Headsets* gave me motivation and demonstrated that a stunning outcome is achievable for this project. His work provided useful references and proved to be an amazing inspiration for the basis of this work.

I would also like to thank my friends at Imperial College and at my home city of Budapest for enriching my life with many memorable moments throughout the last four years. Lastly but not least, I would like to thank my parents for giving me the opportunity to study at Imperial College London and live in London and for their unconditional support, that allowed me to perform my best at university.

Abstract

Humanoid robots increasingly assist with household activities and started to replace workforce in a wide range of jobs. However, teaching new tasks to them still often requires tedious computer programming. Inspired by the way human infants learn, Imitation Learning provides an innovative approach for teaching a robot a desired task, simply based on a couple of demonstrations.

This project is concerned with the research and implementation of an Imitation Learning technique that allows a robot to perform the task of putting a ball into a cup. Using a set of demonstrations, the robot should be able to generalise its knowledge to reproduce the action under different circumstances. Virtual Reality teleoperation provides an immersive way to remotely take the robot's point of view and control its head and limbs through the action. For each demonstration a time-sequence of data is recorded and the temporal sequences are first aligned using Dynamic Time Warping. The data is then encoded with Gaussian Mixture Models, and finally Gaussian Mixture Regression is used to generate a trajectory for the reproduction of the task.

Table of contents

List of figures

Chapter 1

Introduction

1.1 Motivation

In the last decades, technological innovations revolutionized society in many ways. Television changed the way people access news, cars gave us immense mobility and the internet made communication easier with our loved ones, and is bringing the world together in many ways. It is enough to think about modern factories to realise that Robotics has the potential to improve life quality. Household robots are getting more and more widespread, however there has not been a large breakthrough in their commercialization yet. This is largely due to the fact that as of now, most household robots are custom-made for only a very specific task, and human interaction with them is limited, or non-existent. Also, even the more generic humanoid robots are very cumbersome to teach new activities. In the majority of the cases, teaching robots includes computer programming, which is tedious and too complicated to be practical for an average household. Manufacturers are developing learning methods for teaching robots based on physically guiding their arms through the movements. However as of today, most of the time robots still need to be installed and programmed by the developer company.

Robotics is not a new area of research, with the first humanoid robots emerging in the 1930s; however they are still not as commercially widespread as other inventions such as personal computers, mobile phones, or televisions. Every technological invention has the tendency of becoming more and more user-friendly in their life-cycle. Robots are on their ways to being more and more usable by humans, however they are not viewed by the public as being user-friendly yet, which is a barrier they have to overcome before larger commercial success. Human-robot interaction (HRI) is also under heavy investigation by researchers [44]. As more and more people are exposed to robots - in situations such as working by an assembly line or getting help from a household robot - manufacturers are thriving to

make robots easy to operate or cooperate with, even for someone with limited technological knowledge.

Therefore, one of the main objectives of this Master's project is to investigate how to overcome this barrier and make using robots more user-friendly. For this, a humanoid robot will be teleoperated using a Virtual Reality (VR) system. Then the problem of teaching new activities using Imitation Learning (IL) will be explored. These machine learning algorithms allow the robot to reproduce and generalise the operator's actions, based on a number of demonstrations provided previously. This would bypass the necessity for computer programming for teaching new tasks, which could speed up factory or household processes. Imitation Learning is a human-inspired method of learning. Humans also learn from looking at and mastering an act demonstrated by others, and thus it could prove to be a more natural and user-friendly way of teaching for robots.

My previous experience with machine learning motivated me to select this work as my Master's Thesis. During my academic studies I have become increasingly interested in machine learning and its real-life applications such as for Robotics, and found that this project would allow me to combine these interests to create a useful and interesting product.

1.2 Project Overview and Goals

This Master's project is jointly an implementation and research project, focusing on machine learning and human-robot interaction through teleoperation. As mentioned before, it consists of two mainly independent parts with different goals and objectives: 1) Teleoperation and 2) Imitation Learning. In the first part of the project an innovative method of controlling a humanoid robot's movements (the Baxter Research Robot - "Baxter") and obtaining its vision from a remote location using a VR system (the HTC Vive) will be implemented. Teleoperation could be used for other applications than providing demonstrations for Imitation Learning as well, for example for handling objects in toxic or dangerous environments remotely. Vice versa, VR Teleoperation is not the only way for demonstration. Methods like dragging the robot's hands (Kinesthetic Teaching) could also be used, as explored in Section 5.2.2.

1. For effective progress, firstly a reliable work environment and sufficient knowledge and practice with the development tools and frameworks is needed. This allows exploiting all the capabilities of the robot, the 3D development environment and ROS (Robot Operating System) and thus speeds up development, which leaves time for further research. Having prior experience with Ubuntu, C++ and Python, emphasis was put on exploring ROS and VR development environments, namely the game engine Unity, Steam and SteamVR, details of rendering, game development including GameObjects

and details of artificial 3D vision - including notions of the near or far clipping plane. Focus was also put on on C# scripting in Unity and particularly establishing a bi-directional communication channel between the Windows-based Unity environment and the Ubuntu-based ROS. This is necessary, as information will be both streamed from the VR system to the robot and from the robot to the VR system. Sections 2.2.5 and 2.3 will introduce these tools.

2. With the development environment in place, the first step is to understand the Baxter's functionalities including control of the limbs and gripper, limitations and accessing the data published by the robot. Camera images, gripper states and the position of its joints are accessible via various ROS topics. The success of the teleoperation relies on a good knowledge of the Baxter's capabilities and precise control of them.

3. Next, the Baxter's camera feed needs to be displayed on the Vive's Head Mounted Display (HMD) to allow the user to see from the robot's point of view. An Ubuntu-Windows cross-platform bridge needs to be established between the two devices. This will be achieved using ROS#, a framework developed by Siemens that relies on the ROSBridgeClient and Server services made for C# .NET applications to communicate with ROS systems. To place the operator in the robot's point of view, the head movements of the user should be mirrored by the robot. The orientation and position of the HMD will be sent to the robot which will then calculate the required angle of its head and move the head-motor accordingly.

4. For complete teleoperation, the Baxter's arms and grippers need to be controlled to follow the user's hands' movements. This will be achieved similarly, by sending the position and orientation of the Vive's controllers to the robot, which will then calculate the necessary movements of its limbs to achieve the desired position.

Teleoperation will provide means of demonstrating a task to the robot, which could include any simpler household or factory tasks. It is an implementation task, which includes optimising performance for usability. The Baxter should be controlled in a reliable, quick and precise manner and the delay between the human action and the robot's reaction should be minimised, as the visual delay could cause nausea and motion sickness in the operator. [33] However, as Imitation Learning is given more emphasis, optimisation of this part will not be taken to the extremes due to limited time, Chapter 9 - *Future Work* explains the left-out features.

The second part of the project implements the Imitation Learning algorithms that let the robot record the the input signals from the user's demonstration through the teleoperation

system implemented in the first part and reproduce the action and generalise its knowledge so that it is able to perform the task under changed circumstances. This part of the project is more research-based and therefore the objectives are less well-defined and more open-ended. Various Imitation Learning techniques will be explored and there is bigger room for experimentation. There are two stages of this part.

1. First, the movement of the human demonstrator needs to be recorded. Saving different state and action information - such as position, angle or force data - will be explored as the best data to use might be the position and orientation of each of the robot's joints, or the torque applied by each of the motors at consequent moments in time. After couple of demonstrations, the robot shall be able to replay the action on its own without human input. Initially identically to the teleoperator's movements and then in a more generalised way, such that the robot shall be able to reproduce the actions even when the surroundings are slightly modified. The robot will be taught to pick up a ball from a tabletop and place it into a coffee cup. This action could be later evolved to pouring milk or putting sugar cubes in one's coffee - or even preparing the entire drink autonomously.

2. Secondly, the robot should generalise from the previously learnt demonstrations to apply the knowledge to new situations. In reality this could be useful in several situations where there are a set of concrete specifications for the action, but there is large variance between situations. Examples of this might include cleaning plates from a tables or loading the dishwasher, or making a coffee as mentioned. In the case of this project, this means finding the coffee cup and the ingredients in different locations on the table. For these situations the robot would need to adapt to the environment every time it is presented with a new scenario, therefore the algorithms in this part should allow the robot to generalise its behaviour for the different scenarios.

The final product - the combination of the first and the second parts - could be easily applied in industry in dangerous or toxic environments, for dealing with heavy objects or for the completion of tasks that are too long for humans to do in one standing. In households it could make daily chores easier by teaching the robot to cook, clean or tidy. It could be used for assisting immobile or elderly relatives with everyday tasks such as eating, or dressing up, which is being researched by another project of the Personal Robotics Lab of Imperial College. Also, since robots can be designed for continuous operation - such as the Baxter - this project could be used to instruct the robot to do tasks in one standing that a human would have to do over several days. However the applications are not limited to these scenarios.

Every well-defined, but generalizable action could be taught to the robot that is more complex than just repeating the same motion in the same environment.

1.3 Project Resources and Deliverables

Success of this project relies on the combination of hardware and software tools. The backbone of the first part is the hardware, including the Baxter and the Vive and the development machines running the software written for them. The hardware components are provided by the Personal Robotics Lab, therefore the author's contribution will be mostly software based. In the first part it will mostly comprise of combining the right platforms and tools to achieve an operational system for teleoperation. In the second section it will comprise of research and software implementation of innovative machine learning solutions.

The resources used are:

- An HTC Vive Virtual Reality system
- A Windows development PC with a GPU to satisfy the requirements for running the Vive (See requirements in [7])
- Unity, Steam and SteamVR on the PC
- The Baxter Research Robot
- An Ubuntu development PC to control the Baxter
- ROS Kinetic on the Ubuntu machine
- Network connection between the Baxter, the Windows and the Ubuntu machines
- An Intel D435 RGB-D camera for improved camera quality (and depth feed for object recognition)
- Various other software packages

The following deliverables are expected to be produced upon successful completion of the project. The evaluation criteria for these deliverables is discussed in Chapter 7 - *Evaluation*.

- A platform to control the Baxter with the HTC Vive
- A platform that records human demonstrations
- Imitation Learning algorithms to to reproduce the actions
- Further algorithms to generalise the robot's knowledge

1.4 Report Structure

This report is presented to introduce the project, show appropriate understanding of the background and show in detail the work and testing done to achieve the desired outcomes. Following the Project Guideline [23] provided by the Department of Electrical and Electronic Engineering, the report will include the following chapters:

- This *Introduction* chapter serves to help the reader understand the reasons behind the design choices made. It presents the motivations for this project, that is to implement a more user-friendly way of robot teleoperation - using Virtual Reality headsets - and to research a method to teach robots various tasks without the need for computer programming - using Imitation Learning algorithms. The two-part structure of the project is introduced: The following chapters are divided according to Part 1: Teleoperation and Part 2: Imitation Learning. The list of hardware resources used are listed and their combination using software tools will achieve the project deliverables.

- The *Background* chapter introduces the tools, theory and the robot that this work is based on. This includes a brief description of ROS, Unity, Steam, and the connection between the various systems with ROS# and a quick introduction to similar project done in the field of humanoid robots, teleoperation and VR teleoperation. The relevant topics of Machine Learning and Imitation Learning are built-up from the basics. The theory of Gaussian Mixture Regression is established from the notion of Normal Distributions through Multivariate Normal Distributions and Gaussian Mixture Models. The later sections of the project refer to the *Background* chapter for the introduction to tools and explanation of theory.

- The *Requirements Capture* chapter summarises the project deliverables for the two parts of the project. Namely, the necessary functionalities for successful teleoperation are mimicking the user's head and arm movements with the robot and displaying the robot's head camera image in the user's HMD, so that the user can completely take the robot's point of view. For Imitation Learning, recording the demonstrations, learning the key characteristics of the action, generalising the knowledge to new situations, creating a trajectory and replaying the action are the important functionalities that need to be implemented. This chapter also aims to establish the structure of the succeeding chapters.

- The *Design* chapter serves to introduce the final product in more technical detail and its modules from a high-level point of view. It shows that the final implementation of

the system can be divided into three functional components, with a couple of external peripheries connecting to it. These three functional units are the VR control (in Unity - Windows), the control of the Baxter robot (in Ubuntu) and the Imitation Learning algorithms implemented in Python (and partially Matlab). The external peripheries are the hardware, namely the Baxter robot, the VR HMD and controllers, the Intel D435 external RGBD camera with its driver and the object recognition modules.

- The *Implementation* chapter is included to show how the final product is built up from its core components and to show the design and interaction of these units in detail, including justification of the design choices and selection of the final components. First, the detailed setup of the Ubuntu and Windows development environment created is presented, including their bidirectional communication channel through ROS#. Secondly, the ROS and C# scripts and the Unity scene created to take care of moving the head and limbs and streaming the video from the Intel D435 camera to the HMD are analysed. Then, in the second part of the chapter, the six robot-state variables captured during the recording of the demonstrations are introduced and the implementation of Gaussian Mixture Modelling and the Gaussian Mixture Regression part of the Imitation Learning algorithm is explained and finally, the online reproduction of the generalised task is shown.

- In the *Testing and Results* chapter, the successful experimental test results prove that the overall system introduced in the *Implementation* chapter satisfies the standards for the deliverables laid out in the *Requirements Capture* chapter. The most important criteria to fulfill in the Teleoperation part is the minimisation of the delay in the head and arm movements, therefore this section will outline how the three most important components of the delay can be minimised: the delay from the arm's inertia, the Inverse Kinematics solver and the communication channel delays. It will be proven that the 0.2-0.5 second delay in the arms and the 0.1 second second delay in the head movements are satisfactory enough not to cause motion sickness for the user. Three experiments are laid out to test the functionality and robust generalisation of the Imitation Learning part. All of them are based on moving a ball on a table into a coffee cup. To test the overall functionality, the first experiments shows that the robot is able to pick up and move the ball into the cup based on demonstrations taken with fixed initial position of the ball and the cup. In the second and third experiment, the initial position of the ball and of both objects respectively are changed, and the results of these experiments prove that the robot is still able to determine which of the recorded state variables are important to follow at each segment of the demonstrations through

the Gaussian Mixture Regression algorithm and thus its knowledge is generalised enough to find the ball even in a changed environment. The chapter also shows how the test results were used for the optimisation of the system for better performance.

- Consequently, the *Evaluation* chapter presents the system's limitations to behave like a human and the reasons behind the observed test results and explain the nuances of the system's behaviour. The limitations mostly originate from the difference between the robot's and a human's body structure, for example the lack of some human joints in the Baxter, including a proper wrist; and from the limitations of the Gaussian Mixture Model encoding and Gaussian Mixture Regression algorithms.

- Therefore, the *Future Work* chapter will present how the implemented system could be improved by adding object recognition and stereo vision to make the teleoperation smoother, and how probabilistic activity grammars and recurrent neural network could be experimented with to achieve better learning results. The next stage of the project is to implement these further improvements.

  The *Conclusion* chapter will present a short summary to show that the finished project successfully completed the requirements set out.

- A *User Guide* is provided with all of the Unity and Python (Learning and ROS) scripts that are necessary to test the implemented Teleoperation system physically and to reproduce the experimental results. It lists the steps and scripts needed for the robot's setup and operation and it helps the user to use the final implemented system by first starting it up, finding the Unity project and the ROS launch files necessary for operation.

  The *Appendices* contain all the supplementary material that might be interesting but does not add to the understanding of the project. It lists the Evaluation criteria and the Ethical, Legal and Safety considerations taken into account and mentioned in the Interim Report.

Chapter 2

Background

This chapter introduces the related similar projects that have been completed before and the tools and background research that was necessary for this project. First, the tools and frameworks used for programming the Teleoperation part are introduced, then a broad overview of Machine Learning and its sub-field - Imitation Learning - is given. This chapter builds up the background knowledge from a low level, however the more complex concepts are heavily reliant on the basics, therefore it was decided that leaving the foundation would result in an incomplete project report. Nonetheless, the very basics are assumed to be known.

2.1 Similar Projects

It is important to learn the lessons from the similar projects completed in this field before, therefore before the first steps, a thorough background research was conducted. Teleoperation is an old idea, and robots have been teleoperated in numerous creative ways. Virtual Reality teleoperation has also been implemented before, however previous systems are different from this project.

Humanoid robots relate to this project the most, and they are the most useful for real-world applications, therefore a lot of experiments have been done to make robots follow human motion and cognitive abilities. The first step towards making intelligent robots is teleoperation, which ports the intelligence of the human user to the robot. The first teleoperated robots had issues with balancing themselves, as it has been mentioned by Hong, Kim et al. [43] Many approaches of teleoperation for the Baxter Research Robot have been considered, using various different sensors, primarily motion sensors. Cheng, Peng et al. proposed a system where the Baxter was teleoperated using human motion capture equipment. [59] A similar project was carried out by Reddivari et al. who proposed a solution for teleoperating the Baxter using the Kinect sensors of the Kinect Xbox 360 system.

[67] They used the sensors to detect human motion and then control the robot to follow the detected movements. Similarly to this project, they also used inverse kinematic approaches and *rospy* to calculate and control the joint angles, as it is mentioned in Section 2.2.1. Liu, Zhang and Fu used the Leap Motion Sensor system combined with the Myo armband to experiment with Teleoperation of the Baxter using Kalman filter based sensor fusion. [46] Lu and Wen used the collaborative functionalities of the Baxter to create a system that helps mobility impaired people do daily tasks using the sip-and-puff control method, and a Baxter robot mounted on a wheelchair. [49] Zhangfeng Ju, Chenguang Yang, et al. experimented with haptic-feedback based teleoperation [40], however the closest research work was carried out at the Computer Science and Artificial Intelligence Laboratory of MIT, where they used the Oculus Rift virtual reality system to control the robot. The project additionally created representation of the robot in the virtual space, allowing the user to control the robot in the real world while interacting with the virtual world at the same time. [48]. Similarly, The Computational Interaction and Robotics Laboratory of John Hopkins University created an "Immerse Virtual Reality Environment" [32], which is a system that uses the Oculus Rift create a virtual representation of the real world based on the robot's surroundings. It augments the real objects with certain characteristics which is then displayed in the virtual world. Similar system was developed by Peppoloni, Lorenzo et al. [60], where the Kinect camera was used for augmenting reality in the robot's point of view with additional objects that the robot is using at its original location. There are many applications and experiments where VR is used for simulations and controlling models of robots in a simulator. The built-in functionality of ROS# that makes it possible to import URDF (Unified Robot Description Format) descriptions of robots to Unity that is mentioned in Section 2.3.3 was created exactly to ease these kinds of applications. Other than these projects and a couple of personal project, not many papers have been published on virtual reality teleoperation of the Baxter, as most of the projects use motion capture systems and display the robot's actions on regular screens. [42] This gives hope that this project has the capability to improve the currently existing techniques.

Imitation Learning has been applied to many fields in Robotics. The industrial usage has already been described, but in addition there are papers published on how to use Imitation Learning to teach a helicopter to fly [54], grasping objects [79], closing and opening lids [5] or to avoid obstacles during navigation [77].

2.2 Robotics

2.2.1 Inverse Kinematics

In Robotics, inverse kinematics is the process of calculating the joint-angles required to achieve a particular state of a robot using mathematical expressions. The input is usually the position and orientation of one of the robot's fixtures and the output is the joint angles. Motion planning is the process of determining the states that the robot needs to move through in order to get to the desired end state. For the control algorithms, the necessary forces applied by the actuators also need to be calculated from further kinematic equations. [57] [50] Inverse kinematics is used in computer-generated imagery as well. For example in a video-game, the image acquired from a camera mounted on a moving object within the virtual world is also calculated with similar equations. The difference between forward and inverse kinematics is that forward kinematics uses the joint parameters (angles for example) to compute the position of the robot's limbs or grippers. Inverse kinematics reverses the process, and determines the joint parameters from different input sources. The Baxter Research Robot has a built-in inverse kinematics solver, which will be utilised to save time invested into writing complex control and motion planning algorithms for driving the arms.

2.2.2 Kinesthetics

Kinesthetic teaching is a an area of Imitation Learning where the demonstrations are recorded by physically guiding the robot's arms through the desired task, usually from start to finish. [4]

2.2.3 Available Robots

The Personal Robotics Lab of Imperial College London has three available humanoid robots suitable for this project: the iCub, the YuMi and the Baxter (See Figure 2.1), all of which have relative advantages and disadvantages in appearance and functionality. Table 2.1 compares the interesting properties of the three candidate robots.

The iCub is an open source humanoid robot released in 2009, with its hardware and software developed by a collaboration of several European universities. With its height of 1m, the iCub is relatively small and it closely resembles a 3 year-old human child. The creators designed the iCub as a research robot in the field of human-robot interaction and artificial intelligence, especially for research on cognitive learning and how a child's interaction with the world leads to learning. [56] [81] Its eyes have stereo cameras and microphones are built

(a) The iCub *(b) The ABB YuMi* *(c) The Baxter*

Fig. 2.1 This Figure shows the three available robots: the iCub, the ABB YuMi and the Baxter. They are not shown proportional in size, the Baxter is significantly bigger in real life than the other two. It can be observed how the iCub's face is very humanoid, however its joints are too weak for this project. The YuMi is stronger, but smaller than a human. The best fit for real-time human teleoperation therefore is the Baxter

Table 2.1 The list of important characteristics, including the joint torques, size and resources available justify using the Baxter for this project

Characteristics	iCub	YuMi	Baxter
Height	3'1"	1'11"	5'10" – 6'3"
Gripping Force	10N	25N	35N
Peak Joint Torque	40Nm	100Nm	60Nm
Introduced	2009	2015	2011
Intended Use	Research	Industrial	Industrial & Research
Open Source	Yes	No	Partially

close to its ears. The iCub can show various facial expressions using its LED-light mouth and eyebrows, and its head can keep eye-contact by moving its neck. Research shows that such human-like resemblance and emotions tends to make it more comfortable for humans to interact with the robot. [76] Even though some application for this project may contain human interaction, these human-friendly features will not benefit the project hugely and thus are not crucial for the candidate robot. On the other hand, it is important to take into consideration the mechanical properties of the iCub. For mechanical tasks such as lifting heavier objects, the iCub is less suitable compared to the two candidates, as its joint actuators are weaker and it is not designed for industry use. It is not able to exert bigger forces and can only grab small and light objects. The project could exclude lifting heavy objects, however it is not desired to restrict the project based on the robot's mechanical properties, as for real-life industrial use, the robot would most likely need strong limbs.

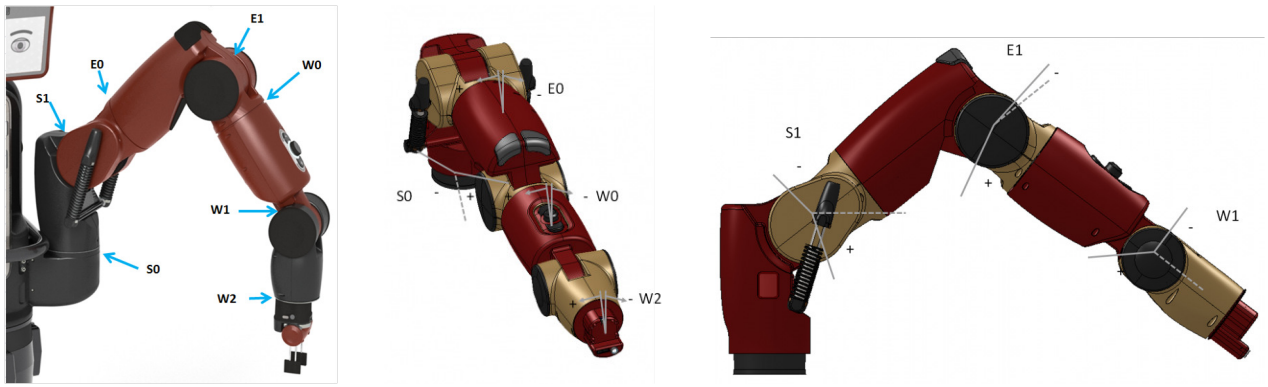
The YuMi is another humanoid robot introduced in 2015 by ABB as a collaborative robot designed to revolutionize the interaction of industrial machines and humans. It is meant to investigate how humans can behave with industrial robots and vice versa to create a flexible assembly environment where the robots' endurance and precision complement the humans' intelligence and responsiveness to change. With its 38 kilograms it is bigger than the iCub, but still smaller than an average person. As it is designed for industrial use, it is strong enough to grab and lift heavier objects. However, due to its size the YuMi cannot cover the entire arm span of a human - the same of which is true for the iCub. This undermines the usefulness of Virtual Reality teleoperation by creating a distorted virtual environment and discomfort in the operator. ABB equipped the robot with built-in kinesthetic teaching features, which means that users can teach the YuMi by physically guiding its arms thought the desired positions. This could anchor the project, however as the YuMi is an industrial robot, is has limited amount of online open-source resources, therefore this potential could not be exploited. Also, few examples and tutorials are available, as companies using the YuMi prefer not to share their intellectual property.

On the other hand, the third competitor, the Baxter Research Robot has plenty of useful guides, tutorials, descriptions and example programs available online. Even though - similarly to the YuMi - it was also meant to work next to assembly lines in cooperation with humans, due to its lower price it became popular with the academic community, hence the online resources. These readily available tools will allow quicker progress in the initial stages of the project and avoid unnecessarily spending time debugging simple errors. The Baxter has significantly stronger joints than the iCub, therefore it can move heavier object and exert bigger forces quicker and more reliably. It has an LCD screen as its face, which allows friendlier human interaction, even though the iCub's head is more human-like. Section 2.2.4 will introduce the Baxter in more detail.

Due to the variety of online resources and its physical abilities, the Baxter was selected as the choice of robot for this project.

2.2.4 The Baxter Research Robot

The Baxter is a humanoid robot, introduced by Rethink Robotics in 2011 for simple industrial tasks, such as sorting or object handling - to take repetitive workload off from assembly line workers. [69]. It is a collaborative robot, meaning that it was designed to connect human factory workers with the automatic robots, and it is capable of behaving as both parties. [69] Even though Rethink Robotics intended it for industrial use, many educational institutions conduct research with it for various research areas in Robotics, Mechanics or Computer Science. [70] Figure 2.1c shows that the Baxter has two arms and an animated face displayed



(a) Labelling *(b) Twist Joints* *(c) Bend Joints*

Fig. 2.2 This Figure shows the seven joints in the Baxter's limbs that it uses to set a particular position. The arms can be moved precisely by setting the angles of these joints to achieve seven degrees of freedom motion. The names on this graph are equivalent to the name of the joints used in the robot's SDK

on a screen that allows it to show various facial expressions. The arms have 7 degrees of freedom controlled by 7 joints. Their names used in the Baxter's pre-written libraries are shown on Figure 2.2a, and Figure 2.2b and 2.2c show their range of motion. The arms can be moved precisely by setting the joint angles, which will allow precise teleoperation. It has a pre-implemented inverse kinematics solver, which calculates the set of joint-angles required to set given the 3D coordinates and a 4D quaternion for the robot's gripper's position and orientation. The robot implements the complex control algorithms needed to precisely drive the joints to the desired gripper position. [71] The research edition of the Baxter has an additional head-mounted camera, alongside with sonar head sensors and IR hand lighting. The programs run on the open-source Robot Operating System on a personal computer inside the robot's chest. ROS is not by definition an operating system, but a multi-platform "middleware" [27] that achieves the tasks of an operating system - running between the low-level implementation of the robot and its software. The operation of ROS is introduced in Section 2.2.5. Similarly to the YuMi, Rethink Robotics ships a kinesthetic-teaching tool for the Baxter. [70] The built-in PC can memorize the limbs' movements and the associated states of the joints during the demonstrations and it can command the Baxter to repeat the task. The scope of this project goes further than this, however this might be a useful aid in the initial stage. Currently the Baxter is also being used in the Personal Robotics Lab to develop an aid that can help people with immobility to dress up.

2.2.5 ROS

The Robot Operating System is an open-source software created by Stanford academics that provides hardware abstraction, drivers and includes libraries and tools for robotic applications. [63] ROS is called "middleware", because it is not an operating system in the classical sense, but instead an operating-system-like software framework that provides inter-process communication and low-level control of various devices. It is widely used in the Robotics community and it works with C++, Python, Matlab and a huge range of robots and sensors smoothly. The ROS executables are arranged into *packages* and are called *nodes*. A *node* can be started in the ROS network after being compiled using *rospy* or *roscpp*, which enable quick interfaces to ROS using C++ or Python, respectively. The Baxter also uses ROS to provide easy access of its hardware. The running ROS system contains a network of ROS *nodes*, which are simply executable programs which have their own functionalities. The *rosmaster (roscore)* is a special master node responsible for managing the other nodes, specifically the communication between nodes. When a *node* is executed, first it needs to register with the *master*. Then it can communicate with other nodes by publishing messages of a particular type onto a predefined *topic*. The receiver must be subscribed to the same *topic* to be able to read the messages. The publisher-subscriber protocol makes communication between nodes simpler by eliminating the need for direct communication channels between each of the *nodes* that desire to speak with each other. [63] [6] [27]

2.3 Virtual Reality

2.3.1 Virtual Reality Systems

Virtual Reality is a simulated environment created by computers though 3D visual and audio feedback to the user, occasionally paired with haptic feedback as well. The created environment is supposed trick the users to believe that they are physically in the new environment, which is usually completely independent of the real surroundings (as opposed to augmented reality, where only an extra layer of virtual information is rendered on top of the real world). The perceived environment may or may not be realistic; many creators experimented with creating fantastical environments. Most of the Virtual Reality systems include at least a headset to observe the created surroundings, a device for audio feedback and controllers to interact with the virtual world's objects. The controllers can occasionally use vibration as feedback to the user. The main use of Virtual Reality is gaming and entertainment,

Fig. 2.3 The Head-Mounted Display (HMD) of the HTC Vive Virtual Reality system is shown in the middle of this image, surrounded by the controllers (first row) and the Base Stations (back row). The base stations track the HMD and the controllers by emitting infrared light and have to be placed in the corners of the play-area to allow fluent operation

however future improvements might enable the use of VR in medical, educational or industrial fields.

2.3.2 HTC Vive

An HTC Vive and an Oculus Rift VR systems are available in the Personal Robotics Lab. Due to its widespread support and numerous online development tools, the HTC Vive will be used to virtually re-create the surroundings of the Baxter in the system. This way the user will be able to take on the robot's point of view even from a remote location. The HTC Vive is a fully-immersive room-scale VR system developed by HTC in conjunction with Valve Corporation. It was first released in 2015 for developers and became available in 2016 for gaming consumers. [68] The system tracks its components on a room scale and lets the user move in the 3D space and interact with the virtual objects using hand-held controllers. [35] Originally, the Vive can be run using Steam from the Windows operating system, however in 2017, support for Linux and Mac came out. Since these still contain certain bugs, a Windows PC was chosen to run the Vive in this project. The system consists of the following main components:

- Head Mounted Display: The HMD provides a 110° view to the user with a refresh rate of 90Hz using two screens - one per eye -, each having a display resolution of 1080x1200. A camera mounted on the headset provides safe operation by detecting physical objects blocking the operator's way. The program can react by displaying an object in the virtual environment, or giving a warning.

(a) Head Mounted Display *(b) The Controllers* *(c) System Setup*

Fig. 2.4 The left image shows the strap that can be used to ensure that the HMD does not fall off the user's head. The image in the middle shows the controllers and introduces the naming convention of its buttons that will be used for the rest of the report. The system's setup is visible on the right picture, indicating that for safety reasons, the user should not have any obstacles in the gaming area

- Controllers: Two hand-held controllers are available to interact with the virtual environments using the trackpad, grip button or dual-stage trigger of each handpiece. (See Figure 2.4b for the positions and names of the controller's buttons)
- Base Stations: The base stations are two black boxes placed in the corners of the room outside of the game-play area. These create the 3D space for the virtual environment and track the controllers and the headset. The controllers and the HMD have infrared sensors built-in to detect the base station's emitted infrared light to allow the base station to determine the devices' positions.
- Tracker: Optionally a motion tracker can be attached to the controllers.

2.3.3 Unity

Unity is a popular game engine that was developed by Unity Technologies and first released in 2005. It was originally intended for Mac OS, however by now it supports 27 platforms. A game engine is a software development environment mainly used to build video games by software engineers and artists for PCs, mobile phones, VR systems and many other platforms. A game engine includes a rendering engine, or "renderer" that creates the 2D or 3D images on the displays from the GameObjects defined in the software. It also has a Physics engine which is responsible for collision detection, and other necessary tools, such as audio and animation management,

thus speeding up the development process by making parts reusable. Unity is widely used to create 2D or 3D animations, simulations and applications. [26]

2.3.4 Steam and SteamVR

Steam is a digital video game distribution platform developed by Valve Corporation used for buying and downloading games alongside its social networking and matchmaking services. Steam helps installation and update management and provides community features such as friends, groups and chat. SteamVR is a plugin required to run VR applications on the HTC Vive from Steam or Unity. These tools were selected to run the HTC Vive, as these are the most widely used ones.

2.3.5 ROS#

ROS# (or ROSSharp) is an open-source C# library developed by Siemens in 2017 that makes communication with a ROS network possible from a non-ROS environment (such as a .NET application in Windows). [13] It was developed with Unity in mind, therefore various functionalities are Unity specific. ROS# makes it possible to define ROS nodes outside of the ROS environment. It provides implementation of the publish-subscribe communication protocol and its nodes can advertise and use ROS services and get access to the ROS parameter server to read or set ROS parameters. It uses the ROSBridge suite and relies on a ROSBridge server node running on the ROS network. It also allows importing Unified Robot Description Format (URDF) models to Unity as a modifiable GameObject. This makes development of simulations using a virtual robot easier by visualising the robot's state more accurately, but this functionality will not be exploited in this project. However, ROS# will prove to be very useful as the bridge between the Vive's Windows based development system and Baxter's Ubuntu based ROS software. As implemented in [9], an alternative solution could have been implemented using the *ROS.NET* system created by students at the Lowell Robotics Lab of the University of Massachusetts. ROS.NET is similar to ROS# in functionalities and also has a Unity specific *ROS.NET_Unity* package. After investigation of both solutions ROS# was chosen, as it is newer, simpler to use and faster support is provided by Siemens.

2.4 Machine Learning

Machine learning is an application of artificial intelligence (AI) that enables computers to learn and adjust their behaviour without being explicitly programmed. This field of research is concerned with the design and implementation of algorithms and statistical models that allow computers to acquire and use already available data - the training set - to perfect their tasks. Machine learning is especially useful in fields where the problem is too general and a computer program cannot be explicitly written to take into account every possible scenario. This includes for example computer vision (for object recognition, self-driving cars, etc.), speech recognition, search algorithms, spam filtering, or anomaly detection, for example to flag unusual activities on a computer network. Machine Learning builds upon techniques from Statistics, Computer Science and Mathematics to create the most accurate and efficient models and algorithms taking into the account the available time and computational resources used for training. [52] [14] [73]

Similarly to the above mentioned fields, Robotics can also have several aspects where the problem is too general and a computer program could not be programmed to deal with all of the scenarios, therefore machine learning is also often used in conjunction with Robotics. For example, when planning a robot's movement (locomotion), no algorithm can be programmed to take into account all of the possible obstacles the robot could face. Section 1.1 mentioned that robots are being increasingly used to collaborate with humans. A person is unpredictable, and has infinitely many actions, so machine learning is useful to create algorithms for human-robot interaction. Also not to forget object detection, object grabbing, and interactive skills, such as speech and display of emotions, all of which are too general for an ordinary computer program. [51] In general, the desired outcome of machine learning in Robotics is a model that can give the next state or action (in terms of movement, speech, or other reaction) of the robot based on its current state, that includes its current environment (such as the obstacles around it, or the humans it needs to deal with).

Robots could learn using traditional machine learning approaches like reinforcement learning or by being guided by a human leader (called Imitation Learning). [8]

A popular traditional approach is reinforcement learning, where the robots learn by receiving feedback after each decision. If the decision is correct, the robot gets a reward, and if it is wrong, a punishment. This approach is also inspired by human learning, however there are several problems with it when implemented in Robotics. Even though it needs less computer programming compared to explicitly writing a program for the action, it is still very challenging to create an appropriate function for deciding whether the robot should be punished or rewarded, which basically includes taking into account infinitely many scenarios. On top of this, the amount of training time is high, as the robot needs to perform a lot of

iterations of the same action, until it can confidently learn the correct and precise action in many scenarios from the feedback. In general the problem with traditional (non-IL) methods is their complexity. To achieve satisfactory accuracy of the tasks, they require large training sets to generalise well and more complicated algorithms. They also tend to take very long to train from the large training sets, therefore often approximations and linearizations are used, which tend to worsen the performance of the system. Hence, Imitation Learning - an alternative solution that deals with this disadvantages is necessary.

2.4.1 Imitation Learning

As it has been established, teaching robotic behaviour by computer programming is difficult and tedious. Imitation Learning (IL) (or Learning by Demonstration - LbD, Programming by Demonstration - PbD) is a sub-field of machine learning, where the algorithms allow the robot to learn through demonstrations of the behaviour from a human guide. Imitation Learning implements algorithms that use the datasets constructed from the demonstrations (the training set) to recreate a sequence of commands to the robot to follow the desired behaviour with changed initial conditions. In Robotics, most of the time it is desired to learn a mapping between the state of the surroundings and the next action of the robot. This mapping has been referred to as a *policy* by Argall et al. [8] Learning and applying the policy allows the robot to select the next action based on its state (position, velocity, etc) without the need to write complicated computer code.

In [8] Argall et al. defined two phases for any Imitation Learning task:

1. Gathering examples (Recording the demonstrations)

2. Deriving strategies from the gathered examples (Implementation of the actual algorithms)

Gathering is the phase where the human demonstrations are recorded, therefore the states, or state-action pairs of the robot are saved, ready for training. Recording methods are different in every application, but normally involves saving physical properties of the robot (position, velocity, torque), or information recorded from external sensors placed on the robot, to more precisely save its state. In some cases, as mentioned in [75], the human demonstrator and the robot has different physical abilities. Such cases could be when robots have more degrees of freedom in their limbs than humans. Or in another scenario, the robot could be more limited than the demonstrator - as in the case of the Baxter: The Baxter does not have wrists in the traditional sense, while the teleoperator does. For Imitation Learning to be effective, the demonstrations and the recording method should account for the differences

(mechanical, or sensory) between the robot and the demonstrator. This is especially relevant when the human demonstrator's states are recorded. In this case it is important to make sure that the robot has a similar physical abilities, or the data is translated to fit the robot's physical build in an appropriate manner.

Deriving is the stage where the algorithms are applied to learn the behaviour and reproduce the action in different environments (generalisation).

There are several examples in the literature for Imitation Learning using teleoperation or kinesthetics - by physically guiding through the robot's arms through the action. The mention of demonstration using teleoperation by Field et al. [29] proposed that the robot could store the states touched during *Gathering* and replay them. However, as already discussed, this simple store-replay approach would lack in generality. Small changes in the surroundings would render the algorithms useless new demonstrations would need to be recorded. Thus in this work, the difference between the demonstrated action and the new surroundings are evaluated by various statistical models to extract the key points from the demonstrated states and discard variance between the models that is noise. [29] The key points would allow freedom for the robot to plan its trajectory without having to take into account the surroundings, other than the key points.

According to Calinon et al. [75], it is an established custom in Imitation Learning to select a sequence of key points of interest from the demonstrations. The algorithms need to decide which points are the key points and the robot's control algorithms simply need to include those in the planned trajectory. The trajectory can change every time depending on the surroundings of the robot, which means that if the key points are selected properly, the trajectory will follow the desired path even in a modified environment. [11]

In [11] the authors implemented statistical models to identify the key points and discard the variation in the demonstrations due to noise. This method was proven to work well for simpler tasks and smaller robots, however the performance deteriorates with higher degrees of freedom and more complex tasks. In spite, statistical algorithms are the baseline for Imitation Learning, and this work will predominantly focus on this solution, namely Gaussian Mixture Modelling and Regression [20].

The research on Imitation Learning could be divided to two prevailing areas. On one of them is the above mentioned statistical way, that uses Gaussian Mixture Regression, Probabilistic Activity Grammars, Partially Observable Markov Models, etc. [21] [36] Alternatively, the use of Recurrent Neural Networks (RNN) is also investigated by researchers. RNN is a type of neural network that forms a graph from along a temporal sequence. [30] Using their internal state (memory), they are useful for learning time-dependent tasks [64] -

such as speech recognition, or trajectory generation. [47] [72] However, this method is slow and lots of training data is required in general for good performance.

Based on more promising results in the literature, this work will predominantly concentrate on implementing and improving on statistical methods, such as Gaussian Mixture Regression, and RNNs will not have that much attention.

2.4.2 The Normal Distribution

The rest of this chapter consists of the introduction to the background theory of the Statistical Imitation Learning Techniques explored during the course of the project. The information is built up from a low level, which is necessary to establish the mathematical notions and foundation of the more complex concepts, however basic statistical knowledge is assumed. Normal Distributions need not be introduced, however the since rest of the chapter is heavily based on them, the report would lack the concepts and notations if left out.

The Normal Distribution (or Gaussian Distribution) is the most well-known and most commonly used probability distribution in the field of Statistics, Probability Theory and Machine Learning. Many of the real-world's events and vast amount of the human behaviour can be modelled with the normal distribution. Its probability density function is the bell curve (or *normal curve*) and it is well-known that it can be uniquely characterised by two parameters: the mean μ of the probability density function and its variance σ^2 (where σ is the standard deviation of the distribution). According to the parameters, the distribution has two main characteristics. The pdf is symmetrical and the mean defines the location of the axis of symmetry. The variance defines how far away the observations tend to lie from the mean. It is well-known that about 68% of the samples lie within one standard deviation away from the mean, 95% of them lie within two, and 99% within tree standard deviations away.

In mathematical notation, the distribution of the random variable X that follows the normal distribution with mean μ and variance σ^2 can be written as $X \sim N(\mu, \sigma^2)$ and its probability density function f given the two parameters is formulated as:

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.1}$$

Probability density functions of different Normal Distributions with different means and variances are shown on Figure 2.5 as shown to demonstrate how the mean and variance change the shape of the bell curve.

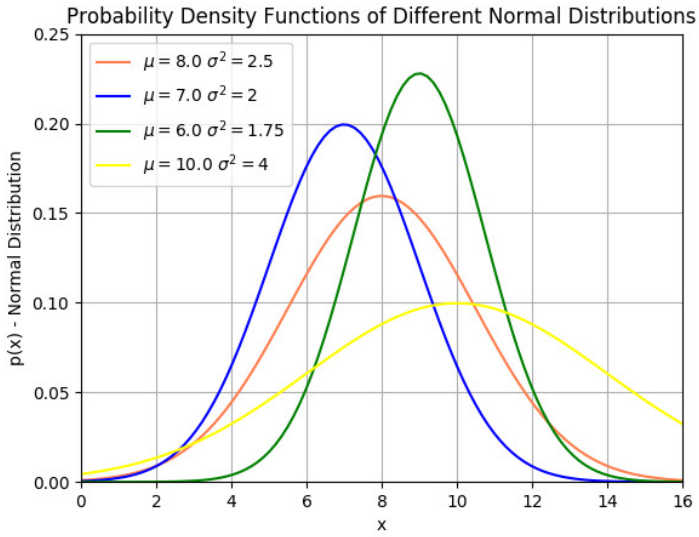


Fig. 2.5 This graph shows the probability density function of four different normal distributions. It is observable how the function gets more and more compressed with increasing variance, and how the mean defines the position of the graph

2.4.3 The Multivariate Normal Distribution

The samples of the previously described Normal Distribution are observed in the one-dimensional space. The Multivariate Normal Distribution (or Joint Normal Distribution) is a generalisation of the previously described one-dimensional *(univariate)* Normal Distribution to a higher dimensional space. By definition a random vector (higher dimensional random variable) is k-variate if every linear combination of its k components has a one-dimensional Normal Distribution. The Multivariate Normal Distribution is very similar its one-dimensional equivalent in the sense that it can also be described by two parameters, a mean vector μ and a covariance matrix $\mathbf{C}$. Similarly to the univariate distribution, in mathematical notation, the Multivariate Normal Distribution can be written as: $X \sim N(\mu, C)$. If the sample space is k-dimensional, then $C \in \mathbb{R}^{k \times k}$ and it can be calculated as:

$$C = \mathbb{E}[(X - \mathbb{E}[X])([X - \mathbb{E}[X])^T] \tag{2.2}$$

The covariance matrix is symmetric and it describes the relationship between the different dimensions of the random variable X. [1] Its diagonal entries are the variances of the individual dimensions of the random vector X. [38]

For the Gaussian Mixture Models (Section 2.4.8) Gaussian Mixture Regression (Section 2.4.10) it is important to understand Multivariate Normal Distributions and the effect of the

covariance matrix on the distribution. The two-dimensional case is the easiest to demonstrate with graphs, therefore the following example will use the bi-variate normal distribution.

In the bi-variate case $X \sim N(\mu, C)$ and $C \in \mathbb{R}^{2\times 2}$ such that

$$\mu = \begin{pmatrix} \mu_{X1} \\ \mu_{X2} \end{pmatrix} \text{ and } C = \begin{pmatrix} \sigma_{X1}^2 & \rho\sigma_{X1}\sigma_{X2} \\ \rho\sigma_{X2}\sigma_{X1} & \sigma_{X2}^2 \end{pmatrix} \text{ and } \rho = \frac{Cov(X_1, X_2)}{\sigma_{X1}, \sigma_{X2}} \tag{2.3}$$

The effect of the mean and the covariance matrix on the probability density function of Multivariate Normal Distributions can be seen on Figure 2.6, where the 3D and the contour graphs of different distributions are displayed. The mean is $(0,0)$ on all three graphs and the covariance matrix C is $\begin{pmatrix} 1 & -0.5 \\ -0.5 & 1 \end{pmatrix}$ $\begin{pmatrix} 1 & -0.8 \\ -0.8 & 1 \end{pmatrix}$ $\begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix}$ respectively. The graphs show that as the off-diagonal entries get larger, the probability density function gets more and more compressed, which is nicely noticeable from the contours below the graphs.

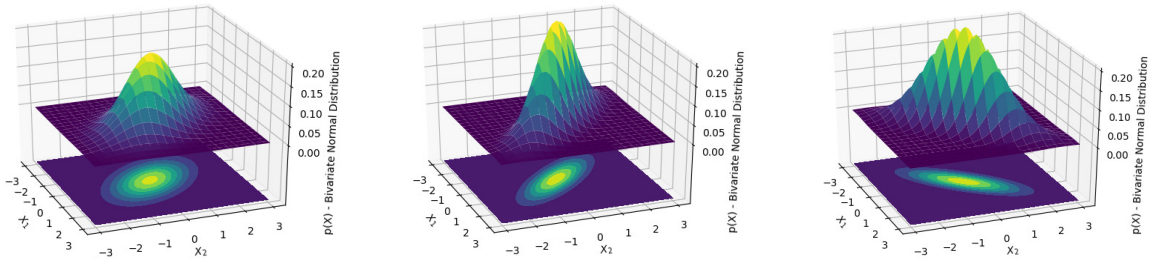


Fig. 2.6 The three pictures show the contour graphs and probability density functions of three different bi-variate normal distributions with different covariance matrices: It is observable how larger off-diagonal entries in the matrix result in more compressed contour graphs, and that the sign of the entries determines the direction of the compression [22]

2.4.4 Expectation-Maximization Algorithm

In the basic scenario of Statistics, samples are taken from a known probability density function, with known parameters, however sometimes the parameters are unknown. In this case, maximum likelihood estimation provides a way to fit a model on the observed samples as closely as possible, giving the parameters that observations were most likely taken of. The process of maximum likelihood estimation involves the maximisation of the log likelihood function of the probability density function of the distribution with respect to the unknown parameter, to find the maximum likelihood estimator of the unknown parameter. In machine learning, this method of maximum likelihood estimation is often used, as usually the model parameters are not given, just samples from a distribution, which are used to figure out

information about the underlying distribution. Maximum likelihood estimation will calculate the estimation of the missing parameter that maximises the probability that the samples were taken from that model. [12]

However, in some cases, the probability density function of the underlying distribution is unknown, therefore it is not possible to write the log likelihood function of the distribution in closed form. Section 2.4.8 introduces Gaussian Mixture Models, where this is exactly the scenario presented. This problem can be solved by using the Expectation-Maximisation Algorithm, which is an iterative algorithm, consisting of the next two steps.

Before starting the algorithm, the unknown parameters need to be initialised with some initial conditions. For the best convergence properties of the algorithm, research [15] shows that a good practice to set the initial conditions is using K-Means clustering. (See Section 2.4.5 for the introduction to clustering.)

- Repeat until convergence:

    1. Expectation - Estimate the unknown probability distribution and its corresponding log-likelihood function from the current estimate of the parameters (the initial conditions in the first step)
    2. Maximisation - Find the parameters that maximize the log-likelihood function that was estimated in Step 1. (Maximum likelihood estimation)

2.4.5 K-Means Clustering

In Statistics, clustering means dividing the observed data samples into groups based on some similar properties. In Machine Learning clustering usually means dividing the data into different regions based on their Euclidean location in the n-dimensional sample space. This usually means that the data samples that are close to each other in the Euclidean sense are placed in the same cluster, while the ones further away should be rendered to another cluster.

The K-Means clustering algorithm divides the data points into k clusters around k cluster centers - k number of cluster averages - hence the name K-Means. The algorithm selects the cluster centres and the points around them in a manner such that the Euclidean distances between the data points and the cluster centres are minimised. This can be explored on Figure 2.7 where the 2D data samples taken from a distribution are clustered around 3 cluster centres.

The K-Means algorithm is an iterative algorithm consisting of the following two steps [61]:

- Randomly initialise k cluster centers $(\mu_1, \mu_2 \ldots, \mu_k)$

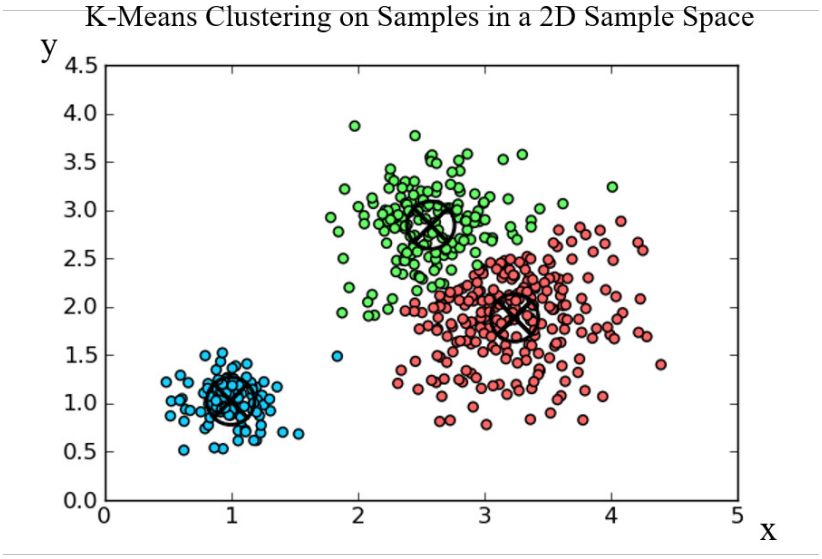


Fig. 2.7 This graphs shows a set of 2D data samples divided into three clusters by the K-Means algorithm. The cluster centres are marked with black crosses and the samples belonging to the same clusters are filled with the same colour. [55] It is observable how the samples physically closer to each other are clustered together, minimising the overall distance of the samples from the cluster centres

- Repeat the following calculations until convergence to local or global optimal solution:

  1. For each i set
  $$c^{(i)} = \underset{j}{\operatorname{argmin}} ||x^{(i)} - \mu_j||^2$$
  2. For each j set
  $$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}$$

Using this algorithm, convergence to a global optimum is not guaranteed, however it is a quick algorithm to reach (at least) local optimal solution. Therefore it is often used to initialise or pre-process data used by other machine learning algorithms such as the Expectation Maximisation algorithm, as it has been mentioned before in Section 2.4.4. This quick method of initialisation can lead to quicker convergence for other iterative algorithms, whose performance heavily depends on the initial condition (like the Expectation Maximisation algorithm). It is important to note that iterative algorithms (including K-Means clustering and Expectation Maximisation) can lead to different (local or global) optimal solutions depending on the initial conditions they are given.

2.4.6 Dynamic Time Warping

Each demonstration is a time-sequence of data. People can recognise patterns in time-variant data very well, and are very good at ignoring the time dependency of a pattern in a sequence. For example, people are very good at noticing similarities even if the patterns are not at the same time in the two time-signals, or if one of the patterns is performed quicker/slower than the other one. In speech recognition for example, humans are very good at understanding each other, no matter when the words are pronounced and at what speed. However, robots and computers find it difficult to notice patters in time-varying data if the signals are not aligned, for example if they are delayed compared to each other, or if the pattern lasts longer/shorter in one of the signals. Computers can compare time signals only with complicated algorithms. Pattern recognition is a difficult task for machines, no matter whether the pattern to be noted in an image, or a time-sequence and it is a heavily researched topic. The perturbation of a signal by a delay or a change of speed might lead to the computer's algorithms interpreting the signal completely differently.

In the case for this project, the time-sequence is the recorded demonstrations, which can be performed slower, faster, starting a bit later or earlier in time, therefore they are definitely not aligned. The rest of the models will be heavily dependent on well-aligned demonstrations, therefore it is very important to find an appropriate way of aligning the demonstrations as closely as possible.

Our goal is thus to make sure that the same part of the movements happen at the same time in all of the demonstrations. There are several algorithms that could be implemented to achieve this, one of them is the Dynamic Time Warping (DTW) algorithm. This problem is called temporal alignment, and the Dynamic Time Warping algorithm provides a non case-specific, general algorithm that is applicable to any dimensional signal for pre-processing before the recognition algorithms. Initially DTW was implemented by D. J. Berndt et al. for the purposes of speech recognition, for matching a query signal to a reference speech waveform.

The main idea of Dynamic Time Warping is to produce an algorithm that distorts a signal in time (in the right proportions) to minimise the distance between the reference signal and the query signal. The distance is the squared distance from the two time-sequences at every time-point added. This distance is to be minimised by applying a *Warping path*, which is a mapping between the elements of the two signals that should happen at the same time-instance. [10]

$$\delta_{ij} = (s_i - t_j)^2 \tag{2.4}$$

$$DTW(S,T) = min_W \sum_{k=1}^{P} \delta(\omega_k) \tag{2.5}$$

Where s_i and t_j refer to a recorded time instance of the two signals and W is the Warping path, which is calculated by the minimisation of the distances. This can be formulated as a nice recursive dynamic programming algorithm [10]:

$$\gamma(i,j) = \delta(i,j) + min[\gamma(i-1,j), \gamma(i-1,j-1), \gamma(i,j-1)] \tag{2.6}$$

The best *Warping path* can be found by iterating the solution backwards from the minimum distance found by the algorithm. [10]

2.4.7 Mixture Models

In Statistical Analysis, often the population is made up of several sub-populations, meaning that the original distribution of samples is a combination of several smaller distributions. For these scenarios, a mixture model is used to represent the presence of the sub-populations within the original population without the need to identify which group each individual sample was taken from. It is a probabilistic model that represents the mixture distribution - the distribution that represents the properties of the sub-distributions. [28]

For example let's consider the house-prices in a particular city. Different type of houses (flats, terraced housed, detached houses, etc.) have significantly different distribution of prices. However, the prices of one type of house, let's say the the price of flats in a selected neighbourhood is likely to cluster close to each other with a defined mean and variance. All of the different kind of properties are subpopulations of the overall population. The distribution of the different house prices in the city could be described individually with a mean and variance, and the combination of all of these distributions would make up the entire city's house price distribution. Then, the model constructed from the individual distributions is a mixture model, with K different components. It is important to mention that in general the individual components can have different type of distributions (normal, log-normal, etc.), but usually identical distributions are assumed. [28]

Using Mixture Models is going to be useful for this work as well. In the case of Imitation Learning, the demonstrations - a time-sequence of recorded data - need to be divided to sub-tasks. Segmenting the overall task can be achieved by mixture models. Based on the time-dependent data obtained from the demonstrations, a model could be fitted to the data points that represents the statistical properties of the entire sequence - the full action - while dividing it into smaller parts, and giving a more accurate statistical representation (mean, variance) of the smaller tasks locally. Building this model could describe the likelihood of a

particular state assumed during the action not only based on the overall distribution of the demonstrations, but in addition to what portion of the overall task - what sub-task of the action the robot is currently performing. In the case of this project, when the overall task is to put a ball into a coffee-cup, an example of the sub-task could be for example reaching for the ball, lifting the ball, or dropping it in to the cup itself.

In this case, a Mixture Model of K components could be defined and used to divide the original dataset into K separate sub-tasks. After dividing the dataset into K different portions - each sample is associated with a separate sub-task - the probability distribution and pdf of each individual component/model needs to be calculated.

In order to mathematically describe the Mixture Model, another variable, z needs to be introduced to describe the probability of the sample to be drawn from a each component. [31] The random variable $Z \sim Multinomial(\pi)$, where π is a vector of probabilities, called the mixing proportions. Therefore, the probability of being in a state from the original sequence's point of view is

$$p(x^{(i)}) = p(x^{(i)}|z)p(z) = \sum_{k=1}^{K} p(z=i)p(x^{(i)}|z=k) \tag{2.7}$$

where $x^{(i)}$ is the state in question.

2.4.8 Gaussian Mixture Models

Gaussian Mixture Models (GMM) are a special case of Mixture Models, where all K components are Normal Distributions with a different mean and variance. They can also be Multivariate Normal Distributions and be described by a mean-vector and a covariance matrix - as it will be used in this work - and in this case the resulting model is a mixture of K joint Normal distributions.

Gaussian Mixture Models can be defined with the following equations and parameters [53]:

- $\mu_{\mathbf{k}}$: The mean vector of the Gaussian Mixture Model, specifying the means of each component k
- Σ_k: The covariance matrix of each of the components k
- $p(z=k) = \pi_k$
- $p(x^{(i)}|z)p(z) = \mathcal{N}(\mu_k, \Sigma_k)$

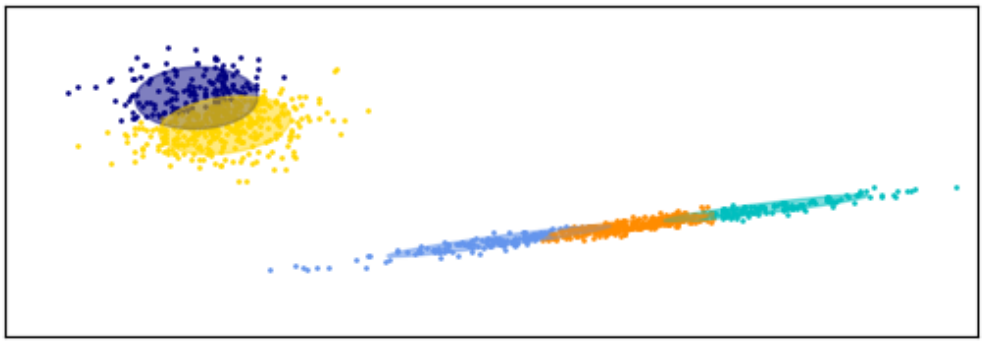


Fig. 2.8 This graph demonstrates a set of badly fitted Gaussian Mixture Model components on a set of observations. The use of five model components is not well-selected, as it can be clearly seen that the data should be modeled with a mixture of two models according to the line and the cluster of samples on the bottom and top. The line should be classified as one sub-task, instead of three

As it has been mentioned in Section 2.4.4, the Expectation Maximisation Algorithm is useful to estimate the parameters that describe the models, which can then describe the conditional probability $p(x^{(i)}|z)$. [53]

2.4.9 Number of Components in the Gaussian Mixture Model

So far, the number of components in the model (k) was given as a model parameter. However in general, it is unknown how many sub-tasks the demonstrations should be divided to. When the number of components is not selected properly, frames that should belong together can get be grouped to separate sub-tasks, or too many frames could get joined together into one sub-task unnecessarily. Increasing the number of components increases the likelihood calculated, but after some points, the data starts to become heavily overfitted by the model. This is shown on Figure 2.8, which is an example code taken from the *sklearn* Python library [58], showing that the line is encoded in three separate models, even though it should be only one. In comparison, Figure 2.9 shows that two models could fit the data better.

The Expectation Maximisation algorithm does not know how many components it should divide the data into, therefore it needs to be selected prior to the EM (and K-means) algorithm. This can be achieved with two criteria, called the Akaike Information Criterion (AIC) [3] and the Bayesian Information Criterion (BIC) [74]. The Python library *sklearn* contains a readily implemented function that returns the scores, once a dataset is provided.

Both of these criteria calculate a score for the model optimality. Their equations include the likelihood plus a loss function that penalises more complex models. Therefore a lower score represents a better mixture model.

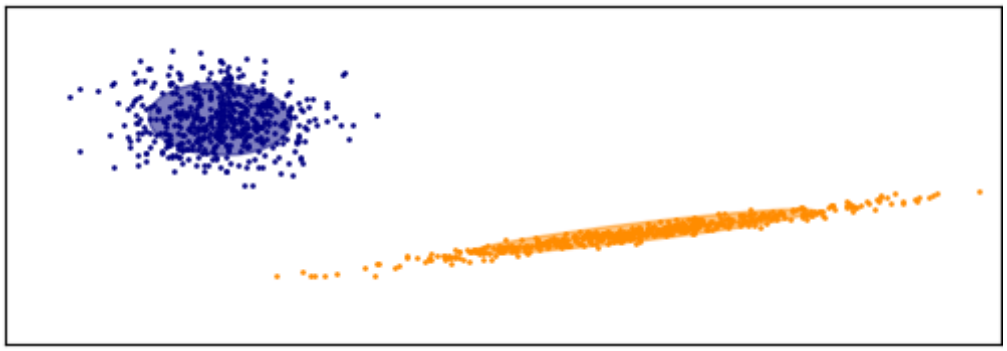


Fig. 2.9 A well-formed Gaussian Mixture Model: The model uses the necessary two model components to represent the two sub-actions properly. This can be achieved by selecting the number of components that yields the lower Akaike or Bayesian Information Criterion

Mathematically, the two information criteria can be formulated as summarized in [2]:

$$AIC = 2k - 2ln(L) \tag{2.8}$$

$$BIC = 2ln(n)k - 2ln(L) \tag{2.9}$$

where k is the number of parameters, L is the likelihood function and n is the number of observations (sample size)

It is observable that that while AIC simply subtracts the likelihood function from a constant term representing the number of components in a model. The BIC is very similar, but the constant cost of the model complexity is weighted by the number of observations that the model is fitted on. Since $ln(n)$ - the weight is always bigger than 1, the BIC score of a model will be higher than the AIC. This means that the number of components is penalised more in BIC than AIC. Both of the metrics will be considered, when the number of models is chosen in the implementation phase.

2.4.10 Gaussian Mixture Regression

In order to generate a trajectory for a robot, it is not enough to have the Gaussian Mixture Models, an algorithm is necessary that produces a sequence of states from the model that the robot could follow. This sequence represent the generalised trajectory of the demonstrations. To achieve this, Gaussian Mixture Regression (GMR) can be used, which is a regression algorithm that generates an averaged path that connects the means of the k GMMs. The exact path produced is controlled by the k covariance matrices. (See Section 2.4.3 for discussion on the properties of the covariance matrix of a multivariate distribution.) Encompassing

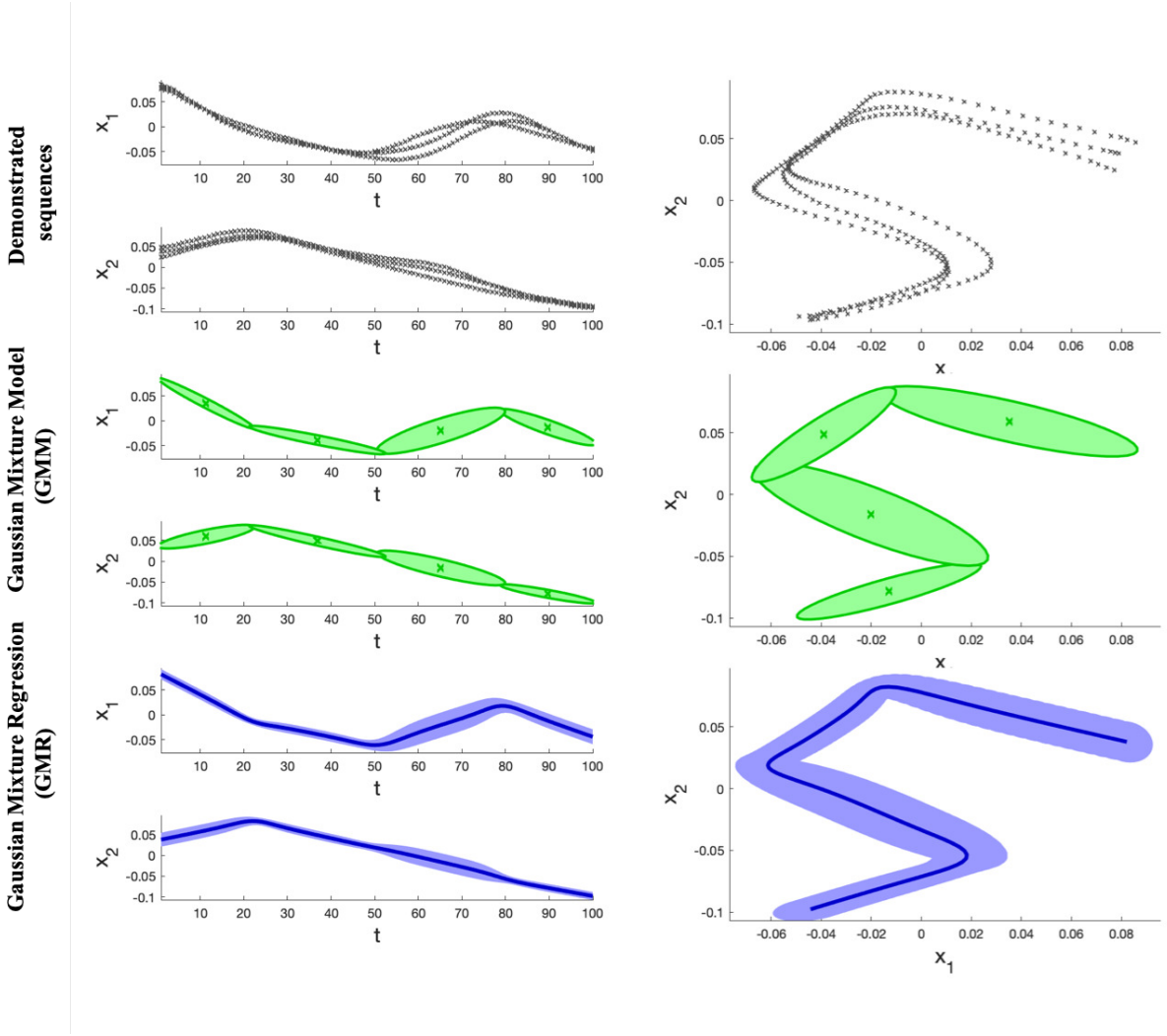


Fig. 2.10 This example shows the original demonstrations, the Gaussian Mixture Models and the Gaussian Mixture Regression generated from a 2D example dataset found in [16]. The top row shows the original demonstrations, the middle row shows the GMM trained to represent the dataset using four components. The ellipses represent the covariance matrix, and the crosses the means of the Multivariate Gaussian Models. The bottom row displays the generalised trajectory generated using Gaussian Mixture Regression. Overall it is clearly visible that when the demonstrations were closer in space, the algorithm generated the trajectory with a bigger confidence (The light-blue region on the bottom-row graphs is narrower)

the correlation between the underlying variables, the matrix expresses how the random variables affect each other. Using GMR, one could generalise the motion recorded in the demonstrations for a generalised reproduction of the action. [66]

It is important to understand the properties of GMR, as described in [78]. The GMR algorithm, implemented in [16] simply produces the joint probability density functions of the K multivariate components of the GMMs. At this state, this is a multi-purpose function, and one (or more) of the variables can be selected as input to the regression algorithm and the rest of them as outputs. [18] The simplest example that still relates to this project (based on and implemented in the *demo1.mat* file in [16]) is a 3 dimensional problem, with the three dimensions being: a time-sequence, and a stream of 2D coordinate data corresponding to the time indices. A GMM could be created from several demonstrations, and then GMR could be used to generalise the data and retrieve a generalised version described by the joint probability density function of the 3 variables. Then, the user could decide for example to feed in a sequence of temporal values as input to the model, which would in turn produce the generalised output of the missing variables, in this case: an expected sequence of 2D coordinates. Figure 2.10 demonstrates and describes this example in more details. (The code obtained to generate this image is found in [16].) It shows how a possible generalised trajectory could be obtained from a set of demonstrations (in this case three demonstrations). The images on the top show the original demonstrations (training set) in 2D, where the coordinates were recorded with respect to time. Then, the graphs in the middle show the GMMs (means as crosses, and the covariance matrices as ellipses) obtained through using the Expectation Maximisation algorithm. The number of components (individual Normal distributions in the model) was selected to be 4 in this case.The GMM captures the relationship nicely between the two coordinate variables x_1 and x_2. Also, the middle graphs show nicely the characteristics of the covariance matrix described in Section 2.4.3. The higher the off-diagonal value of the covariance matrix, the narrower the ellipse is. To interpret this from an Imitation Learning point of view, the GMM divides the action into k - in this case four - different sections, where the coordinate points (x_1 and x_2) are most likely to align, or most likely to come from the same sub-population in a more mathematical sense. This is when the coordinate points do not change a lot from demonstration to demonstrations and there is a stronger Normal relationship between the two variables. This is when the off-diagonal values of the covariance matrix are bigger, and narrower ellipse is drawn. This is represented in the path generated by the GMR algorithm (shown on the bottom graphs in Figure 2.10) which returns a generalised trajectory with a higher confidence (narrower light-blue area around the dark-blue trajectory) if the covariance matrix of the GMM is narrower.

Another example that is more closely related to this project is uses the same algorithms, but with higher-dimensional demonstration-data. In this example, a GMM is trained to encode a collection of demonstrations, all of which could consist of 15 variables: a time-sequence, the position (3D coordinates) and orientation (Quaternions) information of the Baxter's left and right hand recorded at every time instance. Again, using Gaussian Mixture Regression, any of these variables could be selected as input and the rest of them would be returned by the regression algorithm based on the joint probability distribution function. For example, if a new time-sequence is given as an input, an averaged trajectory of the hands (orientation and position) would be returned. Also, let's assume that a time-sequence and the recorded position and orientation of the left hand is given, GMR could be used to predict the right hand's motion (sequence of position and orientation coordinates)

However, of course, instead of feeding in an entire sequence of input, one could feed in the input variables of a single sample as well, and the GMR will return the rest of the corresponding variables. For example in the 2D case, feeding a value of the x coordinates at a time instance, the model returns the missing corresponding y coordinate that is most likely given the time instance and the x coordinate (which is equivalent to saying which of the 4 sub-actions we are in). This y value could be drawn immediately for example, and the next time-x pair could be provided. Considering the second example, the real-time position and orientation coordinates of the Baxter's left hand could be fed into the GMR algorithm as an input, to return the corresponding expected data of the right hand, which could be driven to the calculated state immediately - by the robot. This trick will be particularly useful in the implementation part, when the robot's movements will be calculated on-line based its current state and not just simply based on pre-recorded data. This way, a more precise algorithm could be implemented that makes use of knowing the distance between the robot's hand to the objects that it is dealing with. Note that the distance changes as the robot moves its hands. The flexibility of the Gaussian Mixture Regression Algorithm allows testing with different input and output data variations and experimenting with the best method to reproduce a generalised version of the demonstrations. Further discussion of the testing and implementation will be provided in Section 5.2.3 of the *Implementation* Chapter - Chapter 5

To formulate Gaussian Mixture Regression more mathematically, it is important to note, that in a Joint Gaussian Distribution, the conditional density function of a variable Y given X, another variable, is also Gaussian and is given by:

- The conditional expectation, that can be calculated as:

$$m(x) = E[Y|X = x] = \mu_Y + \Sigma_{YX}(\Sigma_X)^{-1}(x - \mu_X) \tag{2.10}$$

- And the covariance matrix, that can be calculated as:

$$\sigma^2 = Var[Y|X = x] = \Sigma_Y - \Sigma_{YX}(\Sigma_X)^{-1}\Sigma_{XY} \tag{2.11}$$

(Equation 4.2 and 4.3 in [78])

Generally a GMM consists of K components, and thus there are K joint probability distributions corresponding to each component k. Therefore K conditional expectations and covariance matrices are calculated and these are mixed based on the probability ω_k that the sample x was produced from a state k. Using multidimensional distributions and variables x and y, the following equations formulate the problems and can be used to calculate ω_k (the probability that a sub-population created the sample), the means $m(x)$ and covariance functions $v(x)$:

- $$\omega_k(x) = \frac{p(x|k)}{\sum_{i=1}^{K} p(x|k=i)}) \tag{2.12}$$

- $$m(x) = E[Y|X = x] = \sum_{i=1}^{K} \omega_k(x) m_k(x) \tag{2.13}$$

- $$v(x) = Var[Y|X = x] = \sum_{i=1}^{K} (\omega_k(x))^2 \sigma_k(x) \tag{2.14}$$

(Simplifications of Equation 4.11, 4.12, 4.13 in [78])

For every input sample, both the mean and the covariance functions have to be evaluated using the above equations. The mean calculated using the given variables gives the point along the trajectory and the covariance matrix the constraints around it. [16] implements this algorithm, and generates the graph in Figure 2.10. This technique will be implemented for the reproduction of the learnt action; the generalised trajectory from the demonstrations.

Chapter 3

Requirements Capture

This chapter is included to summarise the project objectives and set the structure of the following chapters. This is useful to keep the main goals and the project deliverables in front of the reader's mind.

3.1 System-level Deliverables

The main goal of this project is to create a platform that can be used remotely to teleoperate the Baxter Research Robot. This system would be then used to demonstrate a task several times to the robot, which should learn this action. After a couple of demonstrations, the robot needs to reproduce the task independently, without the human user. It should be also capable of generalising its knowledge to perform the desired action in a different environment as well.

The human input for the teleoperation is provided through a HTC Vive Virtual Reality system. The operator should be able to wear the HTC Vive and completely take the robot's point of view. Successful implementation would allow the user to see what the robot sees and move the robot's limbs and head around just like his own by moving the HMD and the controllers, respectively.

The project has two main parts: Teleoperation and Imitation Learning. The two phases have different structures and work packages. The main project deliverables are divided and outlined accordingly.

3.2 Part 1: Teleoperation

The first part of the project implements the VR Teleoperation platform, which will be used by the human operator to control the robot. The final product should provide quick and reliable teleoperation, including communication between the robot and the VR system and quick processing of information on both sides. The main point to keep in mind is that the user should feel like he is under the robot's skin. The user experience could be enhanced by providing quick reactions from the robot to the user's actions. Particularly important is the visual feedback, which can cause motion sickness in the user, if it is not quick enough.

The system deliverables can be divided into the following smaller work packages. The network of these components will build up the teleoperation system. Most of these components are computer programs that have their own functionality. Some components depend on the successful completion of previous packages, therefore they are completed in the following order.

- **Baxter Setup:** An Ubuntu development environment should be created that allows effective use of the Baxter's SDK to create applications with specific functionalities.
- **Vive Setup:** A Windows development environment should be set up that allows writing applications that are executable on the HTC Vive. This includes setting up Unity, Steam and SteamVR.
- **VR-Robot Connection:** A bi-directional communication channel should be set up that allows communication between the Baxter's operating system and the Virtual Reality world. The Baxter should be able to interface and communicate with the VR world and data needs to be streamed from the robot to the Vive and vice versa.
- **Video Streaming:** The camera on the Baxter's head should have connection to the VR headset and the camera image should be streamed to the Vive's development environment.
- **Video Display:** The Vive should receive and render the camera feed onto an object that is displayed in the headset with as little distortion as possible.
- **Head Position Streaming:** The Vive should obtain and send the position and orientation of the HMD (and hence the user's head) to the Baxter.
- **Head Position Control:** The Baxter should read the head position send from the Vive and move its head to match the orientation of the user's head.

- **Hand Position Streaming:** The Vive should obtain and send the position and orientation of the controllers (and hence the user's hands). This is to be done for both the left and right controllers.
- **Limb Movement Control:** An inverse kinematics algorithm should be implemented that sets the Baxter's arms to mimic position and movements of the user's arms using the hand positions streamed from the Vive. This is again, to be done for both the left and right limb.
- **Trigger Streaming:** A program should be provided that streams the state of the right and left controllers' Trigger button, which will be used to open or close the robot's grippers. The Trigger buttons were selected, out of all available buttons on the controller, because the motion of pressing the Triggers resembles grabbing an object the most.
- **Gripper Control:** The Baxter should close its left or right Gripper when the Trigger button is pressed on the left or right controller, respectively. The Grippers should be closed when the buttons are pressed and opened when they are unpressed.
- **Safety Features:** To provide more control over the robot's behaviour and safety for the researchers working in the lab and surrounding equipment, a button on the controller should enable or freeze the robot's arm-movements, like an on-off button.
- **Optimisation:** The system's performance should be enhanced to decrease the lag between the operator's action and the robot's reaction. The precision of the movements and the streamed image quality should be maximised. This is not a separate work package, but optimal solutions should be implemented in all of the previous parts. Research in the second part of the project is more important than the optimisation of the teleoperator, therefore as it has been mentioned before, more emphasis will be put into finding a suitable machine learning solution than optimising this system.

Figure 3.1 shows the dependencies of the work packages and the order in which they should be completed.

The teleoperation platform is independent from the Imitation Learning phase, and the delivered product should be able to be used for more than just being a way of providing demonstrations for Imitation Learning experiments.

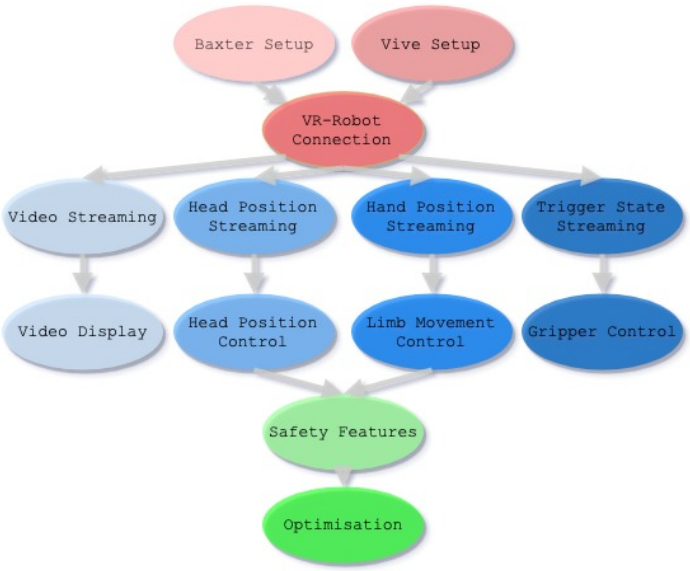


Fig. 3.1 This graph shows the work packages of Part 1: Each bubble is a well-defined deliverable and the arrows indicate the dependencies between them. As observable, some of the tasks can be completed simultaneously, however most of the tasks are dependent on previous packages

3.3 Part 2: Imitation Learning

The second part of the project implements the machine learning algorithms necessary for the learning and reproduction of a particular task. This stage is less structured and laid-out, because research and the implementation of the previously presented background material is necessary. Experimenting with different techniques might lead to better or worse results, therefore the path of the implementation cannot be determined in advance. On a higher level, the guidelines for the necessary functionalities of the overall system are quoted below, based on the three previously established stages of this part.

- **Recording Demonstrations** A script should be written and integrated into the system that records and stores data acquired from the demonstrations. As it is unknown what data is going to be useful for the rest of the experiments, this script should store different data types, including, but not limited to the time-dependent position of the grippers, joint angles and joint velocities. To retain modularity of the Imitation Learning part from the teleoperator, the stored data should come from the robot's states and not the teleoperator's input. In this way other ways of demonstrating the task can be considered. The data type used in the final model will depend on the results of future investigation and the selected machine learning algorithms.

- **Replaying Tasks:** Firstly another script should be implemented that commands the robot to replay the recorded actions. Then trajectory learning algorithms should be implemented that could be used to teach a simple and repetitive task to the robot. The first steps of Imitation Learning should allow the reproduction of the task with a moderate change of environment. The task is going to be as simple as grabbing an object and moving it to a particular position. The robot's first task will be putting an object to a coffee cup using its arms and grippers.

- **Further Generalisation:** As discussed in Sections 2.4.6, 2.4.8 and 2.4.10 , Gaussian Mixture Models and Gaussian Mixture Regression can provide robust methods of training from the robot's demonstrations. They are robust to training noise and are quick to implement and run, therefore first these techniques will be explored to provide a more generic algorithm for trajectory generation. To further improve on generalisation, more research will be put into new methods of Imitation Learning, including Probabilistic Activity Grammars, depending on the performance of the already created system.

Chapter 4

Design

This chapter provides a high-level and a functional overview of the core components of the Teleoperation and the Imitation Learning part, emphasizing their interactions with each other. Most of the implementation comes from the Teleoperation part, as the Imitation Learning part is more research based. Hence this chapter may seem skewed towards introducing Part 1 of the project. The design of the complete system is inspired by and based on [9], however the overall system is changed and improved in many places.

4.1 Functional-level Overview

Figure 4 shows a higher-level overview of the three core components and functionalities of the final design. The three higher-level functional units are related to the VR headset *(Unity)*, the *Baxter* and the Imitation Learning algorithms *(Python)*.

This graphical representation shows how the functional components that have been described before will connect to each other.

The *Unity* part takes care of integrating the Vive into the system. This part takes care of streaming the HMD and controller positions and displaying the Baxter's camera image to the user via the HMD. The source of the camera images will come from the *Baxter* module, initially originating from the Baxter's head camera, and then it will take as input a stereo feed from the Intel D435 camera. This module's output it the head and hand position and orientation coordinated and the gripper states.

The *Baxter* part's main job is to control the robot's movements based on the position information received from the *Unity* module. Understandably the *Baxter* is the most important piece of hardware, and it is in the centre of the modules. It connects the rest of the system together. This component takes as input the head, hand and gripper information to control the robot's arms and grippers and records the demonstrations. Its output is the camera image

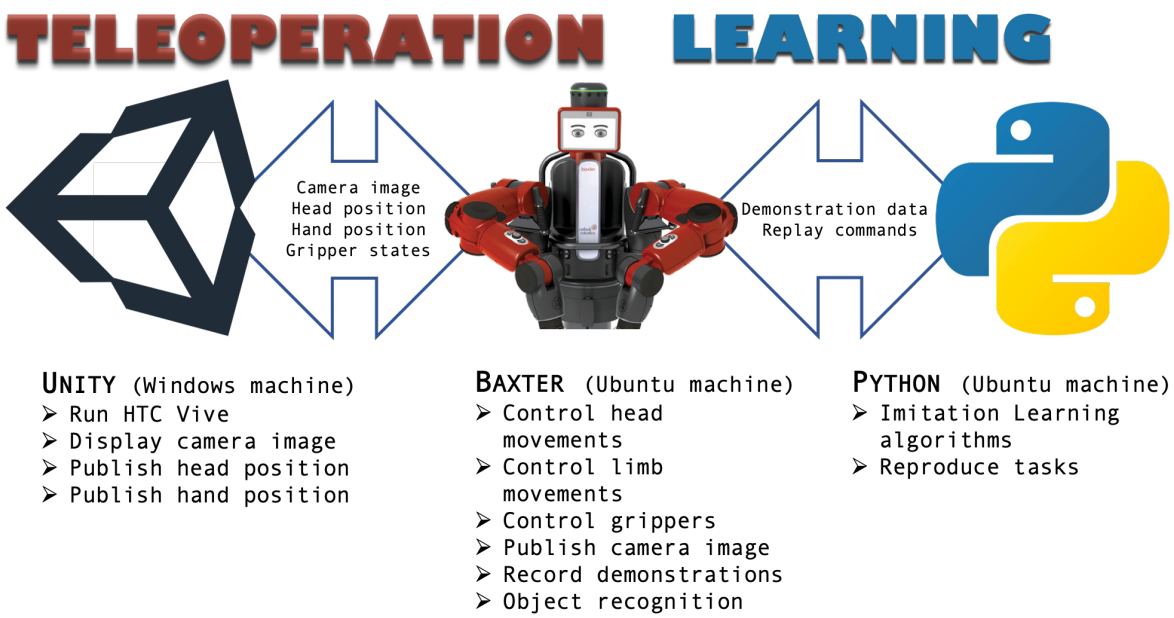


Fig. 4.1 The project is composed of three functional components: the Unity, Baxter and the Python parts. The Unity part is the collection of scripts and scenes defined on the Windows machines that is responsible for controlling the VR system and interfacing the robot. The Baxter unit sets the robot's joints based on the values received from Unity and communicates with the Python part to provide demonstrations and replay trajectories. The Python part represents the Imitation Learning algorithms and necessary data manipulation

feed from the head camera (or the Intel D435 mounted on the head) to be used by Unity. Its output to the *Python* component is its data that it stored during the demonstrations.

The *Python* part implements and trains the Imitation Learning algorithms for trajectory generation. It should be closely interfaced with the *Baxter* to ease the recording and replaying of the task based on the data recorded and calculated. It takes as input and processes the recorded data by performing machine learning and statistical analysis and gives as output the commands to the robot to instruct it how it should move to reproduce the demonstrated task under a new environment. It is important to note that this module will read the robot's states while it is performing the learnt action. It will use the state information to adjust its trajectory, to provide even more accurate reproduction of the task. The outputs are in the format of gripper positions or joint angles in order to make the *Baxter* module compatible with both the *Unity* and the *Python* modules. More on this in Section 4.3.

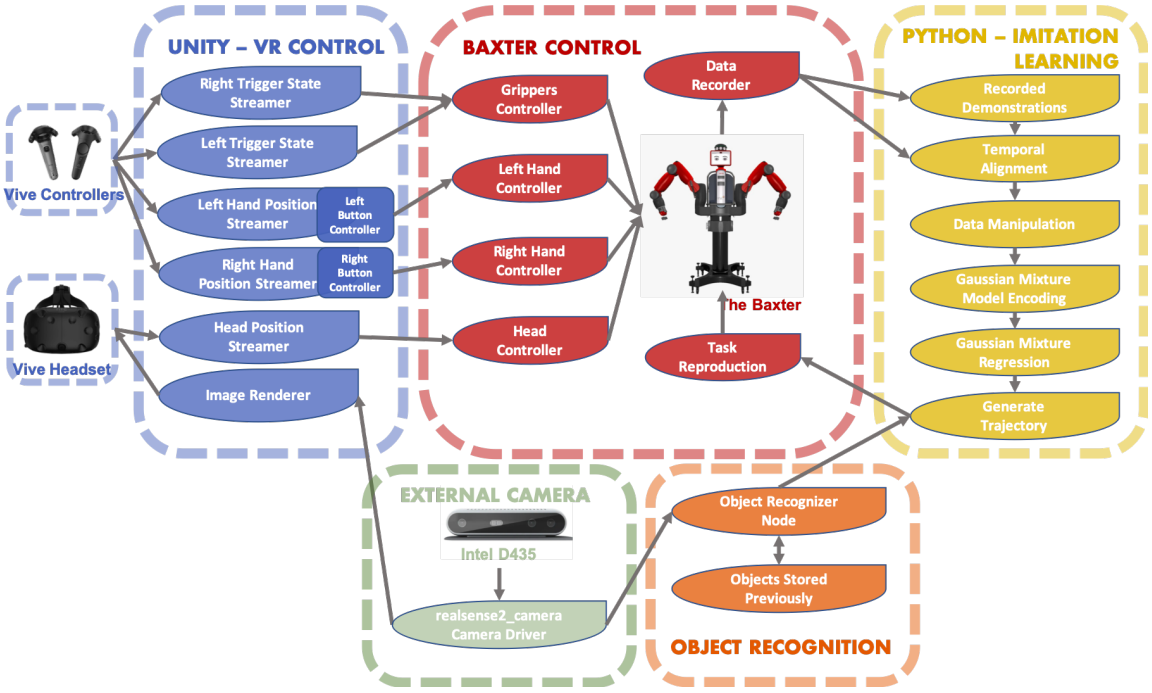


Fig. 4.2 This graphs provides the ROS-level overview of the project components. The modules in the Unity, Baxter and Python parts are clearly visible in each bubble. The Peripherals (Vive controllers, Vive HMD, Intel D435 RGBD camera and the object recognition modules) are connected to the main modules. The communication is indicated by the gray arrows, with the direction of the arrow pointing in the direction of the data flow

4.2 ROS-level Overview

Figure 4.2 provides a lower-level overview of the functional units of the system from a ROS level overview. All of the modules and peripheries of the final system for the teleoperation and Imitation Learning are included with a highlight of the communication between them.

The ROS nodes running on the Baxter (Ubuntu machine) are coloured red, and the "ROS nodes" created in the ROS# system are coloured blue. The yellow part shows the learning part, while the gray arrows indicate the communication channels (topics) where the ROS messages flow. The arrows point in the direction of the data flow. The peripherals are coloured green and orange.

The following ROS nodes are created in the system and summarised on Figure 4.2. Note that this closely resembles the list of the work packages given in Chapter 3, since a functional unit in ROS is usually implemented as a node.

- **Camera Display:** A ROS# node subscribing to the camera image feed and rendering the images to a texture in front of the user's eyes.

- **Camera Publisher:** A node needs to publish the camera images on a ROS topic. The feed of the Baxter's head camera is published automatically by the SDK. In the final design, when the Intel D435 is used, the realsense_camera package is used to stream the data over to the *Camera Display* node.
- **Hand Position Publishers:** Two ROS# nodes are initialised to publish the hand position data (one for each controller).
- **Hand Position Subscribers:** Two ROS nodes will run on the Baxter to subscribe to the hand position information and use the Baxter's built-in inverse kinematics solver service to get the joint angles needed to be set. These nodes should drive the arms and to reproduce the commands of the teleoperator.
- **Head Position Publisher:** A ROS# node that publishes the position of the head.
- **Head Position Subscriber:** A ROS node that reads the published head position data and controls the robot's head accordingly.
- **Recorder:** A script (node) that stores the demonstration data used by the learning algorithms.
- **Object Recognition:** If time allows, an object recognition algorithm should complete the system. It would be attached onto the learning part, which could use the input from the object recognition node to initialise its states, instead of requiring the user to do so. The object recognizer should automatically find the objects that the robot needs to interact with and calculate their positions. The object information should be sent to the Python side of the project to help with the reproduction of the task.

The more in-depth overview of these modules work is given in Chapter 5.

4.3 Modularity of the Design

The system's design is modular, which is useful for easier testing, and it a module could be used for several purposes. Each of the modules could be treated as a black box, and tested for their functionality individually, which allows for a safer development process. The modules could be used for different purposes, which might be necessary, as teleoperation and the action-replay deals with very similar variables and actions, the only difference between them is where the data is coming from. For the former, it is live-streamed from the controllers. for the latter, it is read from a file (and generated on-line in the more advanced algorithms).

To follow modularity, as it has been mentioned before, Part 1 (Teleoperation) and Part 2 (Imitation Learning) phases of the project are independent, and could serve as finished products on their own. This means that instead of storing the commands from the teleoperator, the data recorder should store the robot's states (joint angles, positions, etc.) independent of the teleoperator, so that recordings could be taken using different methods, like kinesthetics.

Wherever possible, the ROS messages will be of the same type, so that they could be routed to other modules more easily. This is especially useful during the implementation of the learning part, because this way, the controller modules of the Baxter can be reused to read the messages streamed from Python. The exact message types are mentioned in Chapter 5 - *Implementation*. For the Baxter, this way, it does not matter where it gets the messages from, it will control the robot's movement from both sources. For the Teleoperation part of the project, deliberately many scripts were used instead of a large one, so that they are reusable for more applications.

Whenever it was possible, the code was written as generally as possible so that it could be changed and adapted to other robots as well, not just the Baxter, or other functionalities could be easily added, such as the vertical head movements.

Chapter 5

Implementation

This chapter shows the implementation of the final design introduced in Chapter 4. The structure of this chapter follows the structure outlined in Chapter 3 - Requirements Capture. The work done to complete the objectives of each work package will be shown from a low-level point of view. Wherever design decisions were made between possible alternative implementations, the justification is provided and the final path taken is discussed. As the previous chapters, this chapter is also divided into two parts, according to the two sections of the project: Teleoperation and Imitation Learning.

5.1 Teleoperation

First, the design of the Teleoperation system will be introduced. As it has been mentioned before, this is an independent system - Imitation Learning is just used as one of the applications of the teleoperated Baxter. The overall goal of this part is to connect Virtual Reality with ROS to achieve the teleoperation of the Baxter Research Robot using the HTC Vive's HMD and controllers. Upon successful project implementation, the user will be able to completely embody the robot and the users actions are completely mirrored by the robot.

For the majority of this project, computer programming was necessary. It would be unnecessary and extensive to include all of the code written, therefore most often only the part of the code will be shown that is relevant for understanding the concepts and design decisions. Imports, includes and repetitions are excluded.

5.1.1 Baxter Setup

The first step of the project is to create an efficient working environment that doesn't set back the development process, and to get used to the tools used.

The Personal Robotics Lab has been using the Baxter for other projects, therefore the installation of the hardware and the software of the robot had been already completed. This saved some time by avoiding having to go through all the setup. It has already been connected to the Lab's internal network, therefore the robot was almost completely ready to go for the project.

However, to develop applications for the Baxter, a development PC or laptop is also necessary. It was decided to use the author's laptop for this purpose, because it fulfils all the hardware requirements for the Baxter's development and it allows a portable development station.

To be able to interface the robot, ROS needs to be installed. First, the laptop had to be dual booted with Ubuntu, because ROS cannot be installed on the already installed Windows 7 operating system. To match the Lab's research tools, it was decided that instead of installing the newest long-time supported (LTS) Ubuntu distro (the 18.04 LTS - Bionic Beaver), the Xenial Xerus (16.04 LTS) was installed. This way, the other researchers can be consulted in case an unexpected error happens relating the operating system during the installation of ROS or the development phase. For similar reasons, instead of the newest ROS (Melodic Morenia), ROS Kinetic Kame was installed.

After installing Linux and ROS, only two steps are left to create an operational development station. The software development kit (SDK) of the Baxter and the laptop's network setup needs to be finished.

The Baxter's SDK contains many code examples, tutorials, tools and scripts. There are numerous tutorials that help setting up and personalising the ROS workstation of the Baxter, many scripts that can be executed to perform basic actions - like enabling the robot (using the enable_robot tool), moving the robot to a neutral position (using the tuck_arms tool), and many others. The tools and example scripts provide help for the developers to understand the structure of the Baxter's codebase and quickly get control of the robot with their own code (move the joints, get access to the states, etc). As a first step of the implementation, some time was spent getting familiar with the Baxter using these tools.

The Baxter also runs on ROS - the Robot Operating System. As it has been already mentioned, ROS is a network of executables called *nodes*, that are black boxes from the system's point of view with their own functionalities. They communicate with each other based on a subscriber-publisher model. Each node can advertise messages of a particular type on a *topic*, and other nodes can listen to those messages if they are subscribed to the node. A particular node - called the *rosmaster* is in charge of keeping account of the other nodes and handling the communication between them. The rosmaster is started with the *roscore* bash command. This needs to be executed on the machine that is to become the master, which is in our case the Baxter. To call this command on the Baxter, it is simply enough to source the **baxter.sh** bash script. Executing this scripts initialises Baxter as the master. After the *rosmaster* is initialised, other machines on the same network can connect to the master as slaves. However, first two environment variables on the slave machines need to be set. Setting the ROS_MASTER_URI variable to the master's IP address (alternatively the master's hostname is also accepted) and the ROS_HOSTNAME variable to the slave's hostname allows the slaves to communicate with the master and create nodes running on them. All of the machines on the network with the same ROS_MASTER_URI variable can publish to the same topics created by their master. The master uses the ROS_HOSTNAME variable to send the messages published on a topic to the right slave machine (the ones that are subscribed). The Ubuntu operating system matches each machine's hostname to their actual network IP address based the /etc/hosts file, therefore it is important to check if all the hostnames are defined and match the IP addresses in that file.

5.1.2 HTC Vive Setup

As it is introduced in Chapter 2 - *Background*, the Vive was developed jointly by HTC and Valve Corporation as a gaming device. Therefore it can be programmed with game engines, such as Unity. Steam and SteamVR (a popular plugin for Unity to develop VR applications) is developed by Valve Corporation, one of the creators of the Vive, Unity has good integration and support for Vive developers. This, and the fact that Unity is a popular game engine

with an established user base, it was decided that the VR part of the project is going to be created with Unity. SteamVR contains the classes and methods necessary to control and obtain information about the VR hardware, and methods that can take inputs from VR controllers and apply them to the scene's objects. The development PC that fulfils the Vive's hardware requirements laid out in Section 2.3.2 runs the Windows 10 operating system and the Windows version of Unity. This is ideal, as SteamVR can only be installed to Windows systems. There is only a beta version available for Linux or Mac operating systems, which unfortunately proved to be extremely buggy. The Windows machines have Unity installed, as it is used by other researchers as well. Therefore starting the project was quick, the time to install Unity was saved. Before any development, it was important to learn the basics of Unity - as the author did not have any experience with the concepts of 3D development.

Unity lets developers create 3D objects in a virtual gaming scene, which is then rendered to the user's platform, based on the virtual location of the player by the rendering engine of Unity. Rendering determines the image that the user should see from the gaming scene, depending on where he is in the virtual space and the point of view it takes. If the user's platform is a mobile phone, or a desktop monitor, Unity renders the game scene at every frame onto a 2D plane, however, if the user is playing with a VR device, the scene needs to be rendered into a stream of stereo images (with adjustable disparity) that is displayed in the HMD of the system (the HMD of the HTC Vive in our case).

Learning the basics of Unity included learning scripting for Unity. Even though Unity has a graphical interface with endless functionalities, it is still very important to know how scripting works in Unity. Attaching C# scripts to the GameObjects allows the behaviour or the structure of the objects to be changed. These can be anything from movements, colours, textures, physical characteristics, colliders around the object or rigid body characteristics and many more. Scripting makes the game scene easier and quicker to personalise and provides another wide range of possibilities for the objects. All C# scripts rendered to Unity objects are executed at every frame and they consists of two core functions:

- The `start()` function is called once upon the start of the game, and it initialises the script. It is often used to set up variables and characteristics of the game object that do not change.

- The `update()` function is called before a frame is rendered to the users eyes, therefore with a frequency of 90Hz when rendering for the Vive's headset. This is used for rendering characteristics that change at every frame, therefore the images coming from the Baxter will be rendered to a plane inside the update function.

5.1.3 Development Environment Overview

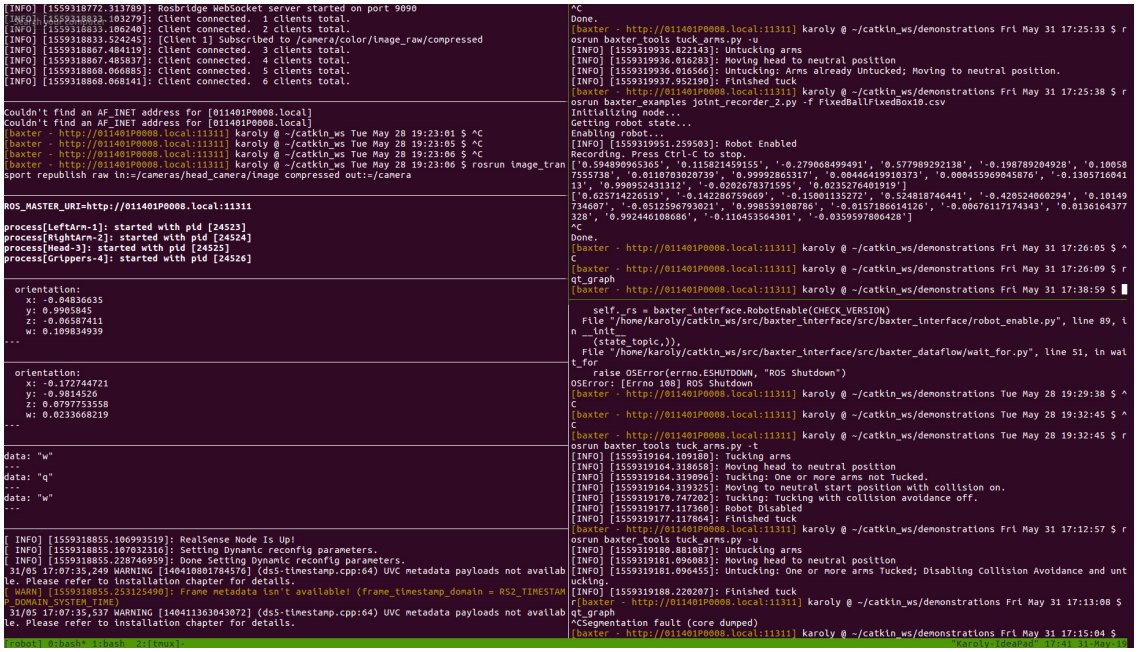


Fig. 5.1 Since the ROS part of the project requires opening a lot of terminal processes, the tmux terminal multiplexer provides a flexible and clear way of organising the necessary terminal windows. This graphs shows the most practical layout for development (the left side contains the processes running, while the right side allows development of two separate files)

Developing with ROS requires starting many terminal windows, which are easy to manage efficiently. For proper terminal window management *tmux*, a terminal multiplexer utility for Linux-based systems was set up. It allows multiple terminal session to be started simultaneously in the same windows, and therefore it is very useful for running more than one command line program at the same time, which is exactly the problem to be solved. Alongside many other functionalities, *tmux* allows to define many sessions, split the window to many panes, and quickly change between the panes using an action key.

See Figure 5.1 for a screenshot of the most efficient Baxter ROS development environment set up in tmux. The screenshots in Figure 5.3 and 5.2 show the VR development environment, including Visual Studio (a C# editor), *Unity* and the physical environment is shown on Figure 5.4. The GameObjects defined in Unity will be highlighted later, in Section 5.1.5.

Figure 5.5 shows a picture of the overall development station including the laptop with Linux (left), the Windows machine running Unity and the VR headset (middle) and the Baxter (right). It is observable that the development environment is safe to the people working in the lab.

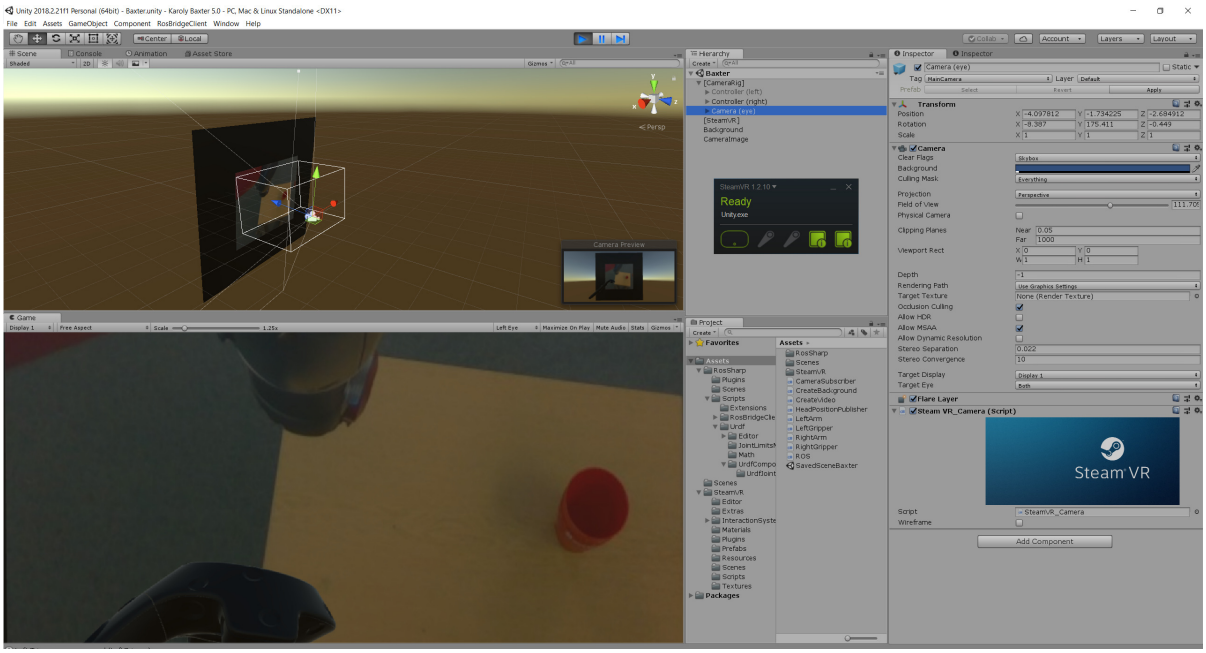
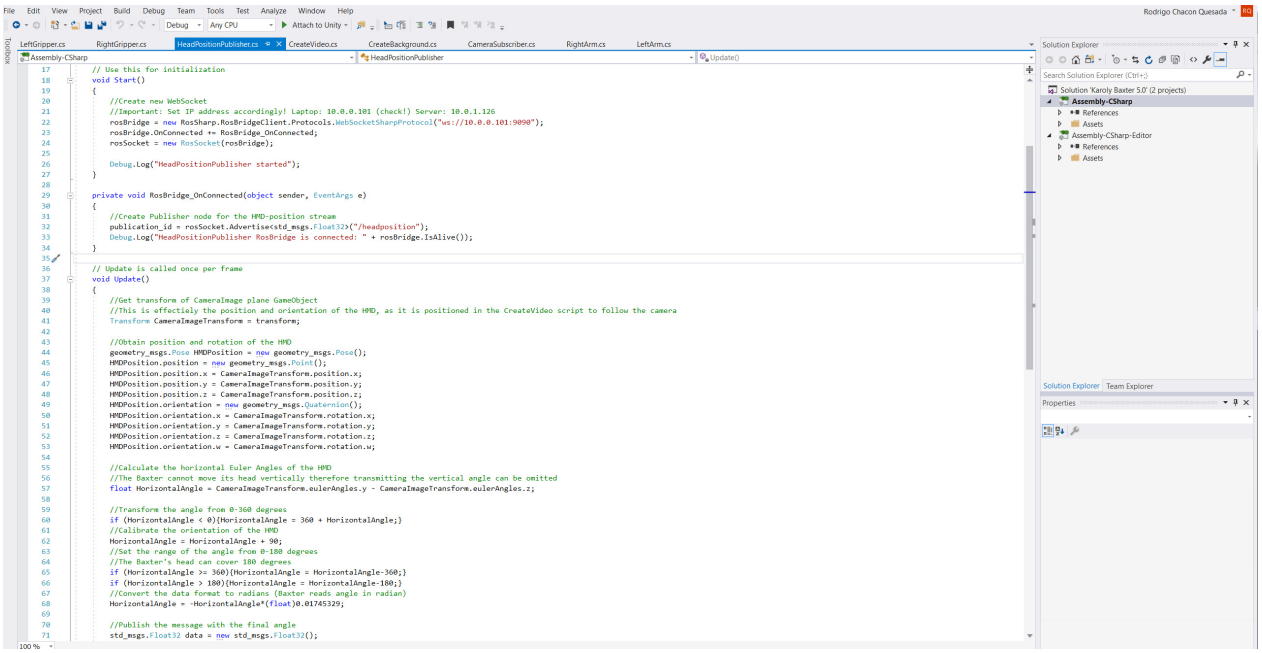


Fig. 5.2 This screenshot shows the scene created in Unity. The GameObjects created are visible on the top-left window (including the Background plane, the Video plane, and the camera and controller GameObjects). The bottom-left image shows what the user can see through the Vive's HMD (the screenshot was taken when the robot was moving a ball (in the robot's grippers - not visible) to the red coffee cup)

5.1.4 Baxter-Unity Connection

Virtual Reality development is rarely connected with Robotics, and even when so, it is mostly for simulations, or testing. Therefore it is not usual to use game engines in Robotics. Unity doesn't have a built-in interface to connect to ROS systems. ROS can only be installed to Ubuntu systems, and it is not trivial to interface a ROS network from a Windows machine, especially not from Unity, as it is necessary for this project.

There are a couple of tools made for such applications however. Two of them were considered, one of them is called ROS.NET developed by the researchers at the Lowell Robotics Lab of the University of Massachusetts, and the other one is a new system created by Siemens in 2017 called ROS#. ROS.NET is similar to ROS# in functionalities and also has a Unity specific *ROS.NET_Unity* package. After investigation of both solutions ROS# was chosen, as it is newer, simpler to use and faster support is provided by Siemens. As it is introduced in Section 2.3.5, after importing ROS# in the project as an *asset*, it lets developers create ROS nodes in C# scripts using Unity or another platform, and run them on a Windows machine (.NET environment is necessary). It also has functionalities tailored towards Unity,



```
// Use this for initialization
void Start()
{
    //Create new WebSocket
    //Important: Set IP address accordingly! Laptop: 10.0.0.101 (check!) Server: 10.0.1.126
    rosBridge = new RosSharp.RosBridgeClient.Protocols.WebSocketSharpProtocol("ws://10.0.0.101:9090");
    rosBridge.OnConnected += RosBridge_OnConnected;
    rosSocket = new RosSocket(rosBridge);

    Debug.Log("HeadPositionPublisher started");
}

private void RosBridge_OnConnected(object sender, EventArgs e)
{
    //Create Publisher node for the HMD-position stream
    publication_id = rosSocket.Advertise<std_msgs.Float32>("/headposition");
    Debug.Log("HeadPositionPublisher RosBridge is connected: " + rosBridge.IsAlive());
}

// Update is called once per frame
void Update()
{
    //Get transform of CameraImage plane GameObject
    //This is effectiely the position and orientation of the HMD, as it is positioned in the CreateVideo script to follow the camera
    Transform CameraImageTransform = transform;

    //Obtain position and rotation of the HMD
    geometry_msgs.Pose HMDPosition = new geometry_msgs.Pose();
    HMDPosition.position = new geometry_msgs.Point();
    HMDPosition.position.x = CameraImageTransform.position.x;
    HMDPosition.position.y = CameraImageTransform.position.y;
    HMDPosition.position.z = CameraImageTransform.position.z;
    HMDPosition.orientation = new geometry_msgs.Quaternion();
    HMDPosition.orientation.x = CameraImageTransform.rotation.x;
    HMDPosition.orientation.y = CameraImageTransform.rotation.y;
    HMDPosition.orientation.z = CameraImageTransform.rotation.z;
    HMDPosition.orientation.w = CameraImageTransform.rotation.w;

    //Calculate the horizontal Euler Angles of the HMD
    //The Baxter cannot move its head vertically therefore transmitting the vertical angle can be omitted
    float HorizontalAngle = CameraImageTransform.eulerAngles.y - CameraImageTransform.eulerAngles.z;

    //Transform the angle from 0-360 degrees
    if (HorizontalAngle < 0){HorizontalAngle = 360 + HorizontalAngle;}
    //Calibrate the orientation of the HMD
    HorizontalAngle = HorizontalAngle + 90;
    //Set the range of the angle from 0-180 degrees
    //The Baxter's head can cover 180 degrees
    if (HorizontalAngle >= 360){HorizontalAngle = HorizontalAngle-360;}
    if (HorizontalAngle > 180){HorizontalAngle = HorizontalAngle-180;}
    //Convert the data format to radians (Baxter reads angle in radian)
    HorizontalAngle = -HorizontalAngle*(float)0.01745329;

    //Publish the message with the final angle
    std_msgs.Float32 data = new std_msgs.Float32();
```

Fig. 5.3 This image is a screenshot of Visual Studio: Unity's connected C# editor, which provides advanced features for C# development

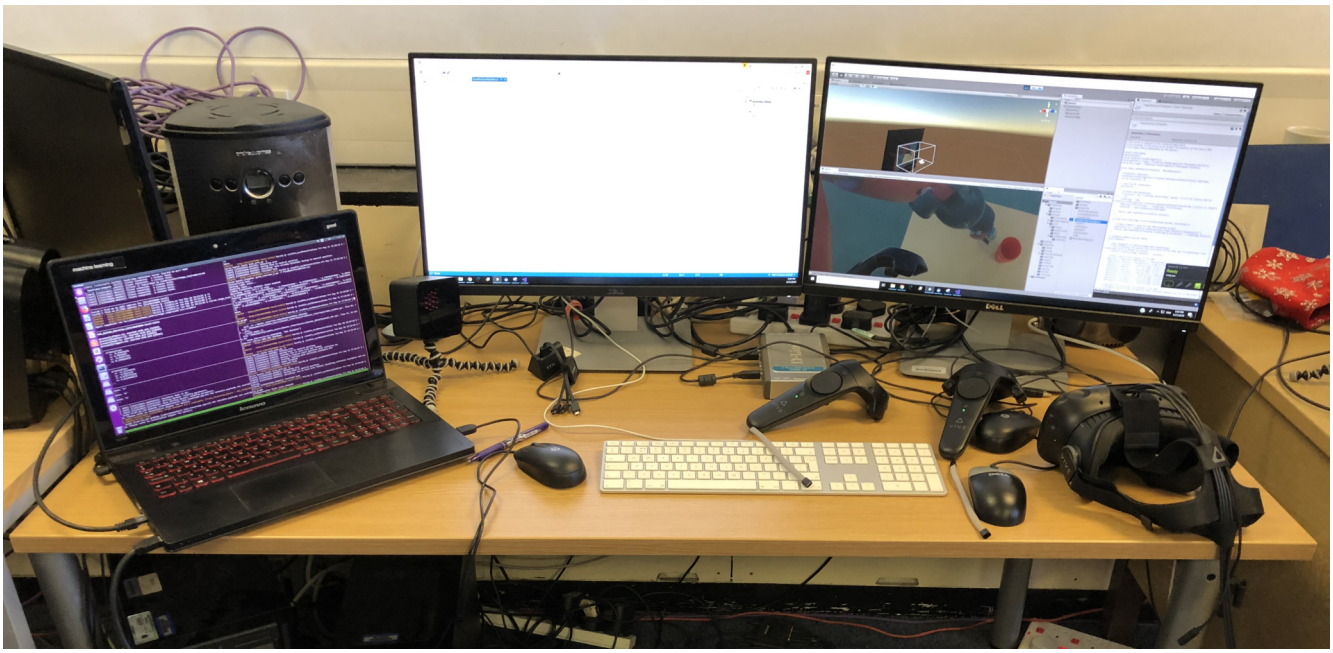


Fig. 5.4 The development environment: The laptop on the left is running Ubuntu to control the Baxter. The computer with the two screens is running Windows 10 and the necessary programs to drive and control the HTC Vive set (visible on the table)

Fig. 5.5 The placement of the Baxter close to the development environment provides an easy way to check the operation of the robot. The surroundings are cleared for safety reasons (except the tabletop that the robot uses for the completion of its task)

for example ROS# allows importing models of robots using URDF format (Universal Robot Description Format), however, this functionality will not be utilised in this project.

The general workflow of ROS# is very similar to an ordinary ROS node, and from the system's point of view it will be completely treated as one. First nodes are created, then they can publish messages on topics, or subscribe to topics to read the messages sent on that topic. The Windows PC needs to be set up just as a computer on the ROS network to be able to communicate with the rest of the nodes. Similarly to Ubuntu based ROS, the machine running ROS# needs to be connected to the network, and two environment variables must be set to allow communication with the *rosmaster*. Therefore the following environment variables of the Windows development PC's were set:

- ROS_MASTER_URI: http://011401P0008 (to point to the Baxter's IP address on the network)
- ROS_HOSTNAME: ViveMAGIC (the hostname of the development PC in the Personal Robotics Lab)

After setting up the connection between ROS and Unity, the necessary development environment is completely set up, and project-specific applications can be created.

5.1.5 Camera Image Streaming

The first task necessary for successful teleoperation is to allow the user to see what the robot's "eyes" can see. This is achieved by streaming a video - a sequence of images from the Baxter's head camera - to Unity, which then renders is onto a full-screen plane in front of the user's eyes in the Vive's HMD. This task could be implemented in three stages. First the images from the Baxter's built-in head camera will be streamed to Unity. Then, for better image quality, depth information necessary for object detection and better camera orientation, the images will be obtained from an Intel D435 camera, mounted onto the head panel of Baxter. The Intel D435 is a stereo camera, and in the future, the Vive's display could display the view from the camera in 3D, by rendering the stereo feed in front of the user's eye onto the HMD, however due to time limitation and the relative importance of the Imitation Learning part of the project, this step had not been implemented yet.

The Baxter comes with three built-in cameras. The main camera is on its head, and there are two additional cameras placed on the grippers for a closer view of the objects that the Baxter is dealing with. In this project, the head camera feed is the only one necessary for teleoperation. However, the three cameras cannot be running simultaneously, because the Baxter can only power at most two of them as the same time. Therefore after running the **baxter.sh** bash scripts from the root of the catkin workspace, the two hand-cameras need to be turned off, and the head camera has to be turned on. Luckily the Baxter's SDK comes with scripts written to quickly turn on-and-off these cameras. Running the following scripts achieves this step:

- `rosrun baxter_tools camera_control.py -c left_hand_camera` - closes the left hand-camera
- `rosrun baxter_tools camera_control.py -o head_camera` - opens the head-camera
- `rosrun baxter_tools camera_control.py -c right_hand_camera` - closes the right hand-camera

After these steps, the various camera information and camera feeds are published automatically to the cameras/head_camera topic by the publisher *baxter_cams*. Figure 5.6 shows the topics published related to the head camera on the top and a couple of other topics published by the Baxter automatically. The image feed is accessible on the cameras/head_camera/image topic. It is simple to pre-view the video using a built-in ROS tool, image_view, by running the following command in the terminal (also shown on Figure 5.6):

Fig. 5.6 The image on the left shows the topics that are automatically published by the Baxter regarding its head camera. These topics can be subscribed to for reading important information about the camera and of course the camera image. The right image shows the image_view tool, which is a small ROS program to display video streams on the screen

- `rosrun image_view image_view image:=/cameras/head_camera/image`

As it has been mentioned before, the messages on the ROS topics can be of different types, which are defined in *.msg* files. There are various pre-defined message types in ROS, but any custom message types can also be defined. These include numerical data, like integers, floating point numbers, etc., strings, images, 2D, 3D position and orientation information, alongside many other formats. Video is naturally sent as a stream of consecutive images, and the images can be sent either in RAW format, or they can be compressed into JPEG or PNG format, with various compression ratios before transmission. The image stream published on the cameras/head_camera/image topic is in RAW format, which is defined as sensor_msgs/Image type in ROS.

However, when this image feed was rendered onto a plane in Unity, the program kept crashing, and the renderer could not display the image feed properly as a video most of the times. When the renderer did manage to display an image, the feed was stuttering and there was a large delay between the image rendered and the actions in front of the camera. The delay was very high - in the order of multiple seconds - which is clearly not tolerable for teleoperation. After some debugging and reading about image compression methods, it was noted that images in RAW format are very large. The large files cannot be transmitted on the network quick enough to provide a video stream. This caused the large delays, and Unity trying to render images that were not fully transmitted was the reason behind Unity crashing. Thus it was clear that compression of the images before transmission is necessary.

Luckily, ROS has a tool, called the republisher, which can "re-publish" an image feed in RAW format, to the message type sensor_msgs/CompressedImage. The republisher can be simply treated as a black box, that takes as input an image feed in RAW format, and publishes the compressed image feed to another one. Later, it is was noted that every external camera's driver uses the sensor_msgs/CompressedImage type to provide the camera feed, therefore using the republisher node is justified to confirm to industry standards.

After running the following command in the terminal, the compressed image feed is accessible on the /camera/compressed topic:

- `rosrun image\_transport republish ...`

  `raw in:=/cameras/head\_camera/image compressed out:=/camera`

The republisher is a new node in the network, therefore the ROS network after starting the republisher is shown on Figure 5.7. The ellipses show the ROS nodes, and the arrows connecting them represent messages published on the topics indicated above the arrows. The direction of the messages is defined by the arrows.

Fig. 5.7 The bubbles show ROS nodes and the arrow shows the rostopic sending data between the nodes. The left bubble shows the node that publishes the head camera image stream in RAW format onto the /cameras/head_camera/image topic. The right node shows the Republisher node, which is a tool in ROS that accepts as input a video stream in RAW format and publishes as output the same stream in a compressed format (std_msgs/CompressedImage format in ROS terms)

So far, no code needed to be written to get the camera feed published, as it is done by the Baxter automatically. This feed is available to be read on the Windows PC, however a node needs to be initialised by ROS# that subscribes to this topic published by the republisher. After obtaining the images, the same node should convert them onto a texture that can be applied to a flat GameObject in the game scene, like a plane. This way, when the scene is rendered at every frame in front of the user's eyes, it will seem that the video is being played onto a plane in the scene. This object needs to be then enlarged and positioned in front of the users eyes such that it fills up the entire screen. This way the video plane will block out everything else that might be going on in the screen, so that the teleoperator will only be able to see the video.

```
1    void Start()
2    {
3        //Create new WebSocket
4        rosBridge = new RosSharp.RosBridgeClient.Protocols.WebSocketSharpProtocol("ws://10.0.0.101:9090");
```

```
5           rosBridge.OnConnected += RosBridge_OnConnected;
6       }
7
8       private void RosBridge_OnConnected(object sender, EventArgs e)
9       {
10          string subscription_id = rosSocket.Subscribe<sensor_msgs.CompressedImage>("/camera/color/image_raw/
                compressed", SubscriptionHandler, 1);
11      }
12
13      void Update()
14      {
15          //If a new image arrived on the topic render it to the plane
16          if (NewImage)
17          {
18              VideoTexture.LoadImage(ReceivedImage.data);
19              GetComponent<Renderer>().material.mainTexture = VideoTexture;
20              NewImage = false;
21          }
22      }
23
24      private void SubscriptionHandler(sensor_msgs.CompressedImage image)
25      {
26          ReceivedImage = image;
27          NewImage = true;
28      }
29
30      private void OnDestroy()
31      {
32          //Unsubscribes and closes the rosBridge connection
33          rosBridge.Close();
34      }
```

Listing 5.1 This script initiates a ROSBridge connection, and creates a node in ROS# that subscribes to the camera image feed. When there is a new image on the topic, it renders the images as the texture of the plane GameObject that this code is applied to. Upon exiting the Unity project, the OnDestroy function takes care of properly shutting down the ROSBridge connection

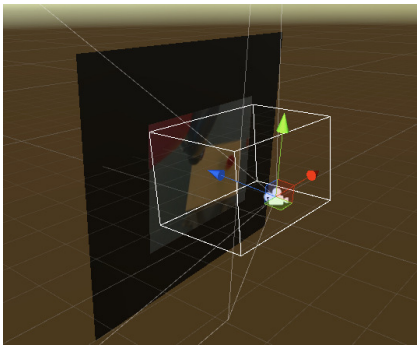


Fig. 5.8 This picture displays the GameObjects of the Unity scene. The black plane is the background, which is behind the video plane. It is visible how the Baxter's camera image is rendered onto the video plane. The rectangular box represents the head-mounted display, technically where the user's eyes are positioned in the system. Not shown are two SteamVR controller objects

Listing 5.1 shows the code that implements the previously mentioned functionalities. This script is rendered to a plane in Unity. This plane can be observed on Figure 5.8. First,

it initialises a ROSBridge connection with the ROS network. This immediately creates a ROS node in the network, which then subscribes to topic /camera/compressed. Later, when the feed is to be taken from the Intel camera, the node should subscribe to topic /camera/color/image_raw/compressed, therefore the name of the topic needs to be changed. The `start()` function of this script only initialises the WebSocket and sets up the variables to be used later. Whenever a new image is published on the topic, the function called SubscriptionHandler sets a boolean to true, that indicates that a new image has arrived. In the `update()` function, the program checks if a new image was published on the robot's camera by reading this boolean. If there is a new image, it applies the image to the plane that the script is attached to, by setting it as the texture of the plane, and it sets the boolean to false, so that this process can start from beginning. Then, when the image is rendered to the user, it will seem that the video is playing on the plane.

With this script, the most difficult task of playing the actual camera image is done, however at this point, it is still just being played to a rectangular plane, and it is not resized and fixed in front of the user's eyes yet.

```
1   void Update () {
2
3       Camera MainCamera = Camera.main; //Obtain main camera object
4
5       //Position: the Near Clipping Plane of the camera plus an offset
6       //Offset: Sets the distance of the Video plane from the Near Clipping Plane
7       //Important! Offset has to be less than the offset of the Background plane!
8       float position = (MainCamera.nearClipPlane + 3.0f);
9
10      //Move the GameObject forward by 'position' amount
11      transform.position = MainCamera.transform.position + MainCamera.transform.forward * position;
12
13      //Position the plane perpendicular to the viewers eyes
14      transform.LookAt(MainCamera.transform);
15      transform.Rotate(90.0f, 0.0f, 0.0f);
16
17      //Scale the plane to fill the Vive's screen
18      transform.localScale = new Vector3(0.5f, 0.001f, 0.45f);
19  }
```

Listing 5.2 This script replaces the virtual reality scene with the camera feed by first positioning the underlying plane in front of the near clipping plane to anchor the video in front of the user's eyes. Then it rotates the plane to face the user and scales it to fill the Vive's HMD

Listing 5.2 shows another script that is attached to the plane, that enlarges the plane to replace the virtual reality scene with the camera image. To make the scene realistic, the horizon needs to be kept as the user moves his head. To achieve this illusion, the image needs to be rotated within the scene such that the actual horizon matches the image's horizon. The user also might move within the screen. However, the camera image should be always rendered in front of his eyes, therefore the object it is rendered on needs to be placed with respect to the user's location. To make sure that the video is always in front of the user's

eyes, it is placed just beyond the Near Clipping Plane, which represents the camera's lenses in game engines. The objects behind this plane are visible and in front of it are not. The Near Clipping Plane luckily moves in Unity's coordinate system as the user moves the Head Mounted Display, therefore it is enough to position the video with respect to it.

At every frame, the script simply first sets the position of the plane object, so that it is slightly behind the near clipping plane. To replace the rest of the virtual scene with the video, it needs to be close to the camera (the user's eyes)). This distance is represented by an offset set to 3.0. Then, the plane is rotated to keep the horizon, and be in front of the user and enlarged to fill the entire screen of the HMD. The ratio for enlarging the script is found by testing which ratio looks the best on the HMD.

Additionally, another plane was added in the scene, which provides a black background behind the video plane. This is important to change the blue background of Unity to black. In theory the video plane completely covers everything else in the Unity screen. When the image is rotated to keep the user's horizon, the edges of the image cannot cover the corners of the screen. This could be solved by enlarging the video plane enough to cover the screen even when the rotation is applied, however this would zoom in and cut too much of the image. Instead, when the background it changed to black instead of Unity's very intense blue background, this slight lack of image in the corners is barely noticeable. A trade-off is needed to be made here, and since the advantages of a less zoomed in image outweigh the deterioration in the system caused by the black corners, it was decided that this is the better design choice.

A very similar script (not shown) that scales and rotates the video plane needs to be applied to the background to also keep that in front of the user's eyes. It is important that the background needs to be behind the video plane, so that it does not cover the images. Thus the offset from the Near Clipping Plane of this plane is slightly larger than the offset from the video plane. The rest of the script is the same, therefore to avoid repetition, it is omitted from this report. Figure 5.8 shows the Unity scene, and it shows how now the user can see the video on full-screen when wearing the VR headset.

With this, a functional system for the camera streaming and display is implemented. However, there are several reasons why the camera image from the Baxter's head camera is not satisfactory for the rest of the project.

First of all, the main camera's video quality is not great, as it can be seen on Figure 5.6. (Max resolution: 1280 x 800 pixels, effective resolution: 640 x 400 pixels, frame rate: 30 frames/sec.) A better image quality would improve the user experience greatly. Secondly, the orientation of the head camera is not adjustable, it is fixedly built into the head. The head can only be moved left and right horizontally, but since the Baxter's neck does not have a

motor to move the head vertically, the robot cannot look up and down. The fixed vertical angle of the camera does not show a good view of what the hands are doing. This is very disadvantageous for teleoperating the robot to interact with smaller objects on a table in front of the robot, which is the requirement for Part 2 of the project. Lastly, the built-in camera does not provide neither depth information about about the objects is can see, which would be beneficial for object detection in the later stage (See Section 9.4 for further information). Nor does it provide a stereo image stream, which could be used to create a 3D image of the surroundings in the headset.

Due to the above reasons and following the recommendations of the PhD students working with the Baxter, it was decided that the camera feed will be changed to an external camera's feed. Two cameras were available and suitable: The Asus Xtion and the Intel D435. The Intel D435 was selected as the camera of choice, because it fulfils all extra requirements - it provides depth and a stereo feed and the image quality is significantly better, and it was already mounted onto the robot's head with a case in the desired angle by another researcher. The case allows adjustment of the vertical angle in case that is needed.

To connect the camera to the ROS network, two ROS tools were installed - librealsense2 and realsense2_camera - which are necessary to access the camera's video. These tools are drivers for stereo and RGBD cameras made by Intel that supply a 3D feed and depth data.

The different data feeds can be accessed independently by installing the command-line tool called realsense-viewer. This tool proved to be useful during the setup of the camera, to check that everything works. After plugging the camera into the Ubuntu development Laptop, various feeds can be accessed from the realsense-viewer.

The realsense2_camera tool works similarly to the Baxter from the prospective of the rest of the system. It is a device on the ROS system can be treated as a black box that publishes various image feeds onto topics that can be subscribed by the other nodes. First it initialises the node, then queries the system for connected cameras, and when it finds one, it publishes the following topics from the camera.

It publishes a topic named camera/color/image_raw/compressed, which contains the images of type sensor_msgs/CompressedImage needed to replace the head-camera's image. Therefore from the Unity script's point of view, the only thing that needs to be changed is the name of topic that the camera subscriber subscribes to. With this setup, the republisher node can be shut down, as it does not need to publish the camera images anymore.

The Intel camera's image quality is significantly better, and its depth data feed is crucial for successful object detection, as it is shown in Section 9.4. Even though transporting the images in the compressed and not the RAW format eliminated most of the delays in the display, the republisher still takes some time to compress and publish the new feed therefore

there was still some delay between the camera image and the actions in front of the camera. Since the republisher is not needed anymore, the delay between the head movements and the displayed video was even further reduced.

To further improve the system, the camera's stereo feed could be used to create a 3D view in the headset. Rendering both of the streams with adjusted disparity would create the illusion of a 3D video to the teleoperator, and could potentially improve teleoperation by providing a better sense of distances to the user.

This concludes the complete implementation of the camera streaming and the video display part of the Implementation.

5.1.6 Head

The next step is to mimic the users head movements with the robot's head. Th Baxter automatically publishes not just the camera image, but many other useful information about itself. For example it also publishes on different topics all of its joint angles, the position and orientation coordinates of its grippers calculated by the robot using direct kinematics, and many more. To follow the user's head movements, first another simple script was created in Unity. It first creates a publisher node then it obtains the angles of the headset and publishes the angle data on topic /headposition.

```
1   private void RosBridge_OnConnected(object sender, EventArgs e)
2   {
3       //Create Publisher node for the HMD-position stream
4       publication_id = rosSocket.Advertise<std_msgs.Float32>("/headposition");
5   }
6
7   void Update()
8   {
9       //Calculate the horizontal Euler Angles of the HMD
10      float HorizontalAngle = CameraImageTransform.eulerAngles.y - CameraImageTransform.eulerAngles.z;
11
12      //Transform the angle from 0-360 degrees
13      if (HorizontalAngle < 0){HorizontalAngle = 360 + HorizontalAngle;}
14
15      //Calibrate the orientation of the HMD
16      HorizontalAngle = HorizontalAngle + 90;
17
18      //Set the range of the angle from 0-180 degrees
19      if (HorizontalAngle >= 360){HorizontalAngle = HorizontalAngle-360;}
20      if (HorizontalAngle > 180){HorizontalAngle = HorizontalAngle-180;}
21
22      //Convert the data format to radians (Baxter reads angle in radian)
23      HorizontalAngle = -HorizontalAngle*(float)0.01745329;
24
25      //Publish the message with the final angle
26      data.data = HorizontalAngle;
27      rosSocket.Publish(publication_id, data);
28  }
```

Listing 5.3 This Unity script first obtains the horizontal angle of the Vive's HMD, and manipulates by restricting it to range between +-90°and converting the value to radians to be readily readable by the Baxter. Then it publishes the ready head angles on a rostopic to the robot

Listing 5.3 shows the script coded to achieve this. Then the head angles of the Vive's HMD will be obtained and they will be continuously published to this topic in the `update()` function. But before publishing, the format of the angle data needs to be converted to be readily readable by the Baxter's SDK. In Unity's coordinate system to read the horizontal angle of the headset, Euler angle Z of the camera object's transform needs to be subtracted from angle Y. (The coordinate system of Unity is different than of the robot's - this will be mentioned in detail in Section 5.1.7. Then these angles need to be restricted to range between -90 and +90 °, because the Baxter's head can only pan 180 . Adding an offset to this angle data calibrates the value such that 0 °corresponds to the Baxter's head facing forward. Converting the final value to radians, the angle data can be published to the /headposition topic and be readily processed by Baxter.

```
1  def setposition(data):
2      #Set Baxter's head angle to the value received
3      if not (abs(head.pan() - data.data) <= baxter_interface.HEAD_PAN_ANGLE_TOLERANCE):
4          #Adjust angle so that the Vive's and the head's orientation is the same
5          head.set_pan(data.data+1.570796, speed=1, timeout=0)
6
7  def main():
8      rospy.init_node('HeadPositionSubscriber')
9      rospy.Subscriber('/headposition', Float32, setposition, queue_size=1)
10     head = baxter_interface.Head()
11     rospy.spin()
```

Listing 5.4 This very simple Python script initialises a ROS node that subscribes to the head angle data published by the Baxter. A simple interrupt listens to the topic, and when there is a new head angle data on the topic, it sets the orientation of the robot's head to follow that of the headset

On the Baxter, a very simple Python script shown on Listing 5.4 creates a node that subscribes to the /headposition topic, reads the angles values and moves the head accordingly. Figure 5.9 shows the angle data that is successfully received by the /HeadPositionSubscriber node.

As it has been mentioned in Section 2.2.4 the Baxter can only move its head horizontally - left and right, but not up and down. This code only accounts for this movement, however for another robot, it could be very easily extended for vertical angles as well.

5.1.7 Limbs

The next requirement of the project is to implement a system that - similarly to the head movements - lets the teleoperator take control of the robot's limbs simply just by holding and moving the Vive's controllers. This is implemented very similarly to the head movement controller nodes. The user should be able to move the robot's limbs simply just by moving the two VR controllers, one in each hand. The right and left hand both hold one of the controllers. Compared to the head-angle data published for the head controllers, now a

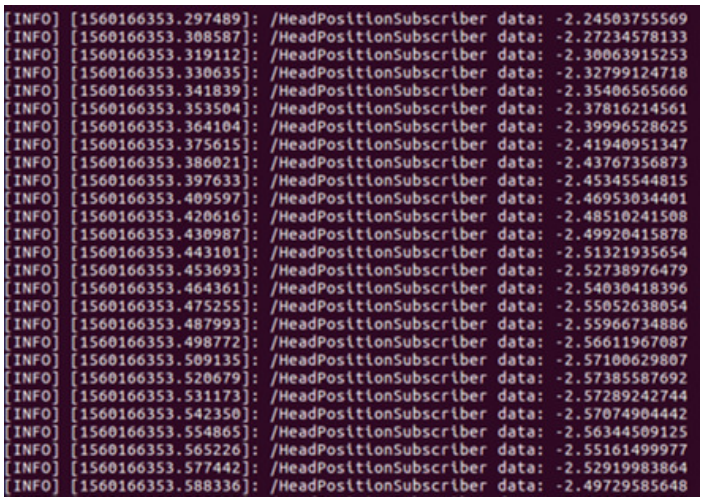


Fig. 5.9 The image shows that the /HeadPositionSubscriber ROS node successfully receives and prints the angle of the Vive's HMD. THis value will be used to set the orientation of the Baxter's head and proves that the ROS# connection between the Ubuntu and Windows machines works properly

much better description of the controllers' state needs to be published. Namely the 3D position coordinates and the quaternion orientation coordinates of both controllers in Unity's coordinate system. In addition to these, the controller button states also need to be published. As mentioned in Section 3 and 4, the Grip button will serve as an on-off button for the arm movements, and the Trigger button will be used to command the robot to grab an object - to open or close its grippers. However, the gripper control will be discussed later, in Section 5.8.

An image of the HTC Vive's controller is shown on earlier on Figure 2.4. The labels on the figure establish the names of the buttons that will be used in further discussion, when describing the functionality of each button in the system.

To achieve these, a Unity script was written and attached to the two controller objects. This script performs a number of different task, most importantly, it obtains the position of the controller object in Unity's coordinate system, and streams it to the ROS network on ROS topics - /leftarmposition and /rightarmposition for the left and right hand respectively. The position of the controllers are calculated with respect to the Vive's headset, and then the coordinates are adjusted to be readable by the Baxter (correct coordinate systems - align head of the user to head of the robot) The Baxter would then read the value, and move the arms accordingly. Listing 5.5 shows the code written in Unity.

It is important to mention that Unity's and the Baxter's coordinate systems are different, in various ways. This was noted, when the movement of the Baxter's limbs were first implemented. The arms were trying to move to the values in Unity's coordinate system,

which caused incomprehensible limb-reaction to movement of the controllers. During the development process, it was realised quickly that the wrong arm-actions are due to mismatch in the coordinate system. The three axes - x, y and z - are different in Unity's and the Baxter's coordinate system. This had to be corrected by mapping and correcting the orientation of the coordinates in Unity so that it matches to that of the Baxter's using the following mapping:

- Unity's **x** position coordinate is mapped to Baxter's **z** coordinate
- Unity's **y** position coordinate is mapped to Baxter's **x** coordinate
- Unity's **z** position coordinate is mapped to Baxter's **y** coordinate
- Unity's **x** orientation coordinate is mapped to Baxter's **z** orientation coordinate
- Unity's **y** orientation coordinate is mapped to Baxter's **-x** orientation coordinate
- Unity's **z** orientation coordinate is mapped to Baxter's **-x** orientation coordinate
- Unity's **w** orientation coordinate is mapped to Baxter's **w** orientation coordinate

Line 10-19 of Listing 5.5 shows this alignment.

```
1   private void RosBridge_OnConnected(object sender, EventArgs e)
2   {
3       //Create Publisher node for the Controller-position stream
4       publication_id = rosSocket.Advertise<geometry_msgs.PoseStamped>("/leftarmposition");
5   }
6
7   void Update()
8   {
9       //Align the coordinate systems of the Baxter and the Vive
10      ControllerPosition.position.x = -ControllerTransform.position.z;
11      ControllerPosition.position.y = ControllerTransform.position.x;
12      ControllerPosition.position.z = ControllerTransform.position.y;
13      CameraPosition.position.x = -CameraTransform.position.z;
14      CameraPosition.position.y = CameraTransform.position.x;
15      CameraPosition.position.z = CameraTransform.position.y;
16      ControllerPosition.orientation.x = Quat.z;
17      ControllerPosition.orientation.y = -Quat.x;
18      ControllerPosition.orientation.z = -Quat.y;
19      ControllerPosition.orientation.w = Quat.w;
20
21      //Apply head-to-arm offset to the Controllers and set position relative to the head
22      CameraPosition.position.z = CameraPosition.position.z - HeadOffset;
23      //Subtract the two camera's position from the controller's position and rescale them
24
25      //Create position message to be sent
26      geometry_msgs.PoseStamped data = new geometry_msgs.PoseStamped();
27      data.pose = ControllerPosition;
28      data.header = new std_msgs.Header();
29      data.header.frame_id = "base";
30
31      //On-off button for sending the position of the Left Controller
32      if (GripDown){SendData = !SendData;}
33
34      //On-off button for sending the rotation of the Left Controller
35      if (TrackPadDown)
36      {SendRotation = !SendRotation;}
37      if (!SendRotation){//Set pre-defined orientation coordinates}
38      //Send the data if it is turned on
```

```
39            if (SendData){rosSocket.Publish(publication_id, data);}
40        }
```

Listing 5.5 This Unity script adjusts and streams the left controller's coordinates. After initialising a subscriber node in ROS#, it obtains the position and rotation of the controller, changes the coordinate system to match that of the Baxter and sets the head's position to be the the origin. This way only the relative distance from the headset matters when the controllers are moved. A PoseStamped object is created, which is the format of the rosmessage that is sent over the topic, which is suitable for this case, because it can contain position and orientation coordinates. Depending on whether the TrackPad button was pressed, or not, the actual, or a predefined set of orientation coordinates are stored in the object alongside with the position. After the Trigger button was pressed (SendData=True), the data is published over the topic. A very similar script is applied to the right controller

After the coordinates are aligned, a minor improvement in the user experience can be achieved by adding an offset to the vertical coordinate of the coordinates. The controllers' coordinates have to be calculated with respect to the headset, this way it does not matter where the user stands, the robot's limb positions will only be dependent on the user's arm position and not absolute position. However the origin of Baxter's coordinate system is inside its waist. This means, that without subtracting the offset, when the user's hands are in line with the head, that's when Baxter's arms are in "normal" position - in line with the waist. To correct this misalignment, for the rest of the system it is assumed that the Baxter's coordinate system's origin is in its head, and then the variable *HeadOffset* is subtracted from the z coordinate to correct this misalignment. The offset in the final system was set to 0.3, however this is an approximate value. The waist-head distance varies from person to person, and the exact value has little significance from the usability point of view, however a value of around 0.3 seemed to be suitable for most teleoperators.

Apart from the hand positions - for more precise and flexible teleoperation and better user experience, the user's wrist orientation also should be mirrored by the Baxter's grippers. This is a challenging task, since the Baxter does not have wrists in the traditional sense, therefore it uses all of its limb joints to recreate a correct orientation. The orientation information is represented in as quaternion both in Unity and in the Baxter's SDK. Quaternions are a mathematical tools that are defined as "the quotient of two directed lines in a three-dimensional space or equivalently as the quotient of two vectors." [25] They can be used to represent 3D orientation of an object and they are generally represented in the form: $a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k} a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k}$

Unity and the Baxter both use the 4 number quaternion representation to store information about the object's grippers' orientation.

To correctly mimic the orientation, the quaternion axes also needed to be aligned, as it was done for the coordinate systems. Aligning two frames of reference is not as simple as for the position coordinate case, however following the change of coordinates described from line 10 and 19 the Baxter follows the correct orientations.

After the Baxter was able to follow the user's hand-position and wrist-orientation, some control over the teleoperation was implemented using the buttons on the Vive controllers.

- **Grip buttons:** Pressing the Grip buttons turns on and off the arm movements. This is an important safety feature, and makes development more practical by allowing to turn off the feature when it is not being tested.

- **TrackPad Button:** The inverse kinematics solver sometimes finds it difficult to find the joint angles required to reproduce the orientation of the angles. The robot sometimes freezes. To avoid these situations, Pressing the touch button turns off sending the orientation information. When it is turned off, the gripper orientation is fixed. In this mode, the wrist angle is not reproduced, and the Baxter's gripper points down, in the best orientation to grab objects.

To read the position and orientation values published in Unity, two Python scripts are created in Linux, one for controlling the left arm and one for the right arm. They subscribe to the /lefthandposition and the /righthandposition topics and read the values published and control the hand state accordingly. Since the Baxter is controllable by setting its joint angles, this script needs to convert the information into join angles first. As discussed in Section 2.2.1, this can be done using Inverse Kinematics, which is a method to calculate the set of joint angles required to set a particular gripper position and orientation. The output of the IK solver is not unique, one might be better than another, and it is possible that the solver does not return a valid solution.

```
1  def setposition(data):
2      #Use the Baxter's Inverse Kinematics Solver service to get the desired joint angles
3      ns = "ExternalTools/left/PositionKinematicsNode/IKService"
4      iksvc = rospy.ServiceProxy(ns, SolvePositionIK)
5      ikRequest = SolvePositionIKRequest()
6      ikRequest.pose_stamp.append(data)
7      try:
8          resp = iksvc(ikRequest)
9      except (rospy.ServiceException, rospy.ROSException), e:
10         rospy.logerr("Service call failed: %s" % (e,))
11
12     if (resp.isValid[0]):
13         #If a valid solution is found, reformat the answer and set the joint positions
14         limb_joints = dict(zip(resp.joints[0].name, resp.joints[0].position))
15         arm.set_joint_positions(limb_joints)
16
17 def main():
```

```
18      rospy.init_node('LeftArmSubscriber')
19      rospy.Subscriber('/leftarmposition', PoseStamped, setposition, queue_size=1)
20      arm = baxter_interface.Limb('left')
21      rospy.spin()
```

Listing 5.6 This listing shows the Python controller that drives the left arm. It creates a subscriber node to read the arm positions and orientations published by Unity and when a new value is received it calls the Baxter's IK service to get the joint angles. If the IK solver returns a valid solution, it controls the Baxter's arms to the right positions. A very similar script takes care of the right arm

After looking at other Internal Kinematics solvers by external developers, the Baxter's built in IK solver was chosen to obtain the joint angles, for simplicity. The other solvers did not work any quicker than the Baxter's and provided no advantages.

The Baxter's IK solver is an IK-Service Proxy that can be accessed through the External-Tools topic, for example through the topic ExternalTools/right/positionkinematicsnode/ikservice for the right arm. It returns the calculated joint angles, which needs to be than unpacked and values have to be matched to the right angles, if the solver could return a valid joint solution for the particular position. The joint angles need to be than passed onto the Baxter as a command to set them. This can be achieved as shown in Listing 5.6

5.1.8 Grippers

The last part of the Teleoperation part is to make the controllers actually control the grippers and implement the previously mentioned safety functions. Similar implementations were found in the baxter_examples folder of the SDK, the gripper control example served as basis of this implementation.

The gripper of the Baxter will be operated (on-off) by the Trigger button of the controller. The robot will be instructed to squeeze or unsqueeze its grippers by pressing or unpressing the button.

The code strings are quoted below. There are more codes defined by the Baxter's SDK, however they are not necessary for this project, but the script could be easily extended to include further functionalities.

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

```
    private void RosBridge_OnConnected(object sender, EventArgs e)
    {
        //Create Publisher node for the Gripper-state stream
        publication_id = rosSocket.Advertise<std_msgs.String>("/gripperstates");
    }

    void Update()
    {
        //Get button actions
        TriggerDown = controller.GetPressDown(Trigger);
        TriggerUp = controller.GetPressUp(Trigger);

        if (TriggerDown)
        {
            State.data = "q";
            rosSocket.Publish(publication_id, State);
        }
        if (TriggerUp)
        {
            State.data = "w";
            rosSocket.Publish(publication_id, State);
        }
    }
```

Listing 5.7 This Unity script commands the robot's grippers. It creates a publisher node in Unity, and publishes a "q" character on a rostopic when the Trigger button is pressed and and a "w" character when the Trigger button is unpressed. These values are read by the robot and the grippers are closed/opened accordingly. A similar script is created to publish the right gripper states, but using the equivalent upper-case characters

Listing 5.7 shows the the implementation of this functionality in Unity, of the left gripper. It is a very simple script that gets the controllers objects, and the Trigger button attributes. Then the script simply streams a set of pre-defined commands when the button is pressed and when it is unpressed over the /gripperstates topic in a simple string format (std_msgs/String). The right gripper's implementation is almost identical and is therefore omitted.

```
def setstate(data):
    bindings = {
        "q": (left.close, [], "left:close"),
        "Q": (right.close, [], "right:close"),
        "w": (left.open, [], "left:open"),
        "W": (right.open, [], "right:open")
    }
    #If the command is defined set the Baxter's grippers
    if data.data in bindings:
        cmd = bindings[data.data]
        cmd[0](*cmd[1])

def main():
    rospy.init_node('GripperStateSubscriber')
    rospy.Subscriber('/gripperstates', String, setstate, queue_size=1)

    left = baxter_interface.Gripper('left', CHECK_VERSION)
    right = baxter_interface.Gripper('right', CHECK_VERSION)

    #Calibrate grippers before open or close commands are accepted
    bindings = {
        #key: (function, args, description)
        "c": (left.calibrate, [], "left:calibrate"),
        "C": (right.calibrate, [], "right:calibrate")
    }
    cmd = bindings["c"]
    cmd[0](*cmd[1])
    cmd = bindings["C"]
    cmd[0](*cmd[1])
```

```
30      rospy.spin()
```

Listing 5.8 This Listing shows the ROS script to control the Grippers. A subscriber node is created that reads the characters published in Unity when the buttons are pressed. A dictionary is used to match the characters to the corresponding robot commands (left/right, open/close) to open or close the grippers. This script also takes care of calibrating the grippers upon startup, which is necessary for proper gripper operation

The Python scripts shown in Listing 5.8 simply reads these commands from the topic, and send the corresponding command to the Baxter to close or to open its grippers. It is important to mention that this script takes care of sending a set of calibration commands to the robot upon startup, which is necessary for correct operation.

5.1.9 Teleoperation Overview

Figure 5.10 shows the rqt graph of the nodes and topics implemented for the full teleoperation system. The ROS# nodes cannot be displayed by the rqt_graph command, as they are not officially part of the ROS system. On the graph they are "behind" the /rosbridge_websocket node. From the ROS point of view, this is the node that handles the functionalities described in the previous Sections.

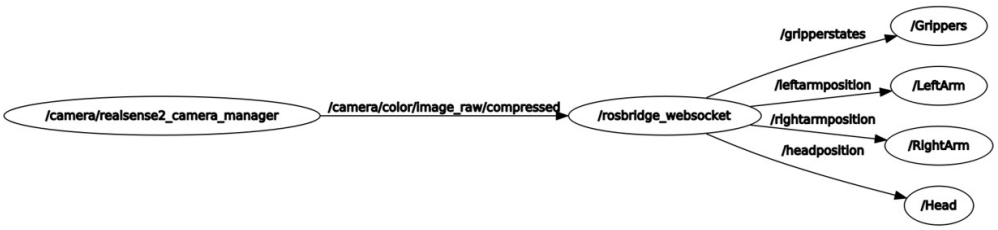


Fig. 5.10 This rqt graph shows the ROS nodes created in the complete system. The Intel D435 camera's driver (node display on the left) publishes the camera image on the /camera/color/image_raw/compressed topic, which is subscribed to by a ROS# topic. The ROS# nodes are represented on this graph by the /rosbridge_websocket node, because from ROS's point of view, the ROS# nodes are shielded by the rosbridge connection and the communication with the outside world (ROS#) happens through the Websocket node. The right side shows the nodes created to control the head (/Head) the arms (/LeftArm, /RightArm) and the grippers (/Grippers) and the rostopics that communicate with these nodes are also visible

For simpler operation, a launch file was created in ROS, that allows to start the ROS nodes easier, by just calling one command. This launch file is a ROS-specific XML file that allows to list the nodes that should be started when the file is launched. The nodes in Unity can already be started by simply running the Unity project containing the defined scene with the GameObjects and the scripts rendered to it.

5.2 Imitation Learning

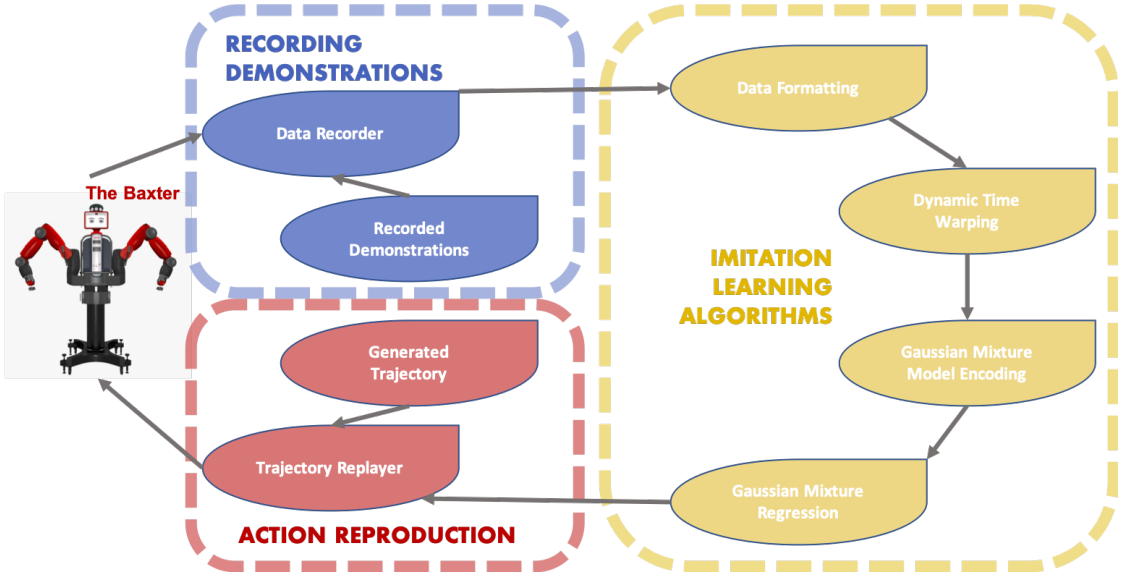


Fig. 5.11 The imitation learning system can be divided into three parts: Recording Demonstrations, Imitation Learning Algorithms ad Action Reproduction. The first two sections are performed offline. The data recorder module will be used to record 10 demonstrations of a task. Then the Imitation Learning algorithms will perform the temporal alignment, the probabilistic encoding and building the model for the generation of trajectories. Then, the action reproduction system feeds data to the model in an on-line manner, to create a generalised trajectory for the Baxter and set its joint angles accordingly

This Section introduces the the work carried out for the design and implementation of the second part of the project - Imitation Learning. This is the part that will allow the robot to learn, generalise and reproduce the task - which will be to find a ball on a table and place it inside a coffee cup. As it has been mentioned before, this stage is independent of Part 1: Teleoperation. In theory, teleoperation could serve other purposes than just to provide demonstrations, and the vice versa, the demonstrations could be provided by other methods, for example by Kinesthetics, as it will be discussed in Section 5.2.2. The system works based on the principle introduced in the Background chapter (Chapter 2); it learns the task, by manipulating and generalising the time-sequence of data recorded of the demonstrations. Section 5.2.2, Section 5.2.3 and Section 5.2.4 describes the tools used to complete the second part of the project, including design justifications and possible alternatives.

Figure 5.11 provides an overview of the implemented system for Imitation Learning. As it can be seen, this system is simpler than the Teleoperation system and is divided into three parts, called *Recording Demonstrations*, *Imitation Learning Algorithms* and *Action Reproduction*. The first part serves as the basis of the next steps. It accounts for the recording

various different states of the robot, as the demonstrations are performed several times. The result will be a time-sequence of data consisting of the states, which is then utilised in the second part, in which, various algorithms will detect the similarities in the time-sequences (after alignment) and create a probabilistic model that represents the important parts of the demonstration. Alternatively, it could be phrased as: it "learns" the action. The output of this stage is a generalised trajectory that is passed to the robot in the last "*Action Reproduction*" stage to follow. As it was introduced in Section 2.4.10, the generated trajectory will be the means of the returned probability distributions, and a constraint (the covariance matrix) will be provided to specify how much variance there is between the demonstrations. Based on the generated trajectory, and a control algorithm, the robot will step through the states that are provided by the Imitation Learning algorithms and the control algorithms on-line. The robot's limbs will be guided through the action step by step, and the its actions will be fed into the algorithms instantaneously. Therefore, it can be noticed that the first two steps (*Recording Demonstrations* and *Imitation Learning Algorithms*) are done off-line, and the last step (*Action Reproduction*) is running on-line based on the outputs produced by the previous two steps.

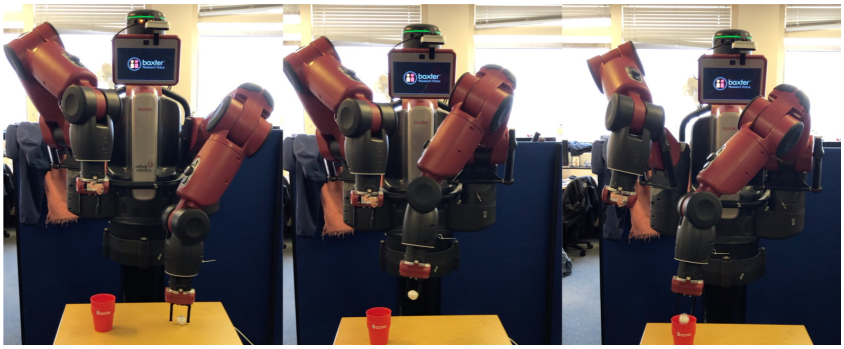


Fig. 5.12 This Figure shows three frames of the robot performing its task from the learnt demonstrations. It first picks up the ball (left image), moves it above the coffee cup (middle image) and opens its grippers to let it go (right image)

Figure 5.12 shows three frames of the robot performing the action - putting a ball in a coffee cup. The action was chosen, because initially a simple action was necessary to implement to make sure that the different parts of the implemented algorithms work. This action is simple, but can be easily extended for example to the more complicated action of making a coffee, by pouring the liquid in the cup, putting sugar cubes in it (instead of the ball) and stirring them to mix it. Ideally, the robot would have to find the ingredients, and prepare the morning coffee of the user independently. It is an ambitious goal, and various steps are required before it could work. Putting the ball in the cup is a simple task that is

useful to perform measurements and compare different methods to decide which is better to implements.

5.2.1 Notations

It is important to establish the notations used in this Section for mathematically describing the scenarios.

- The number of demonstrations taken: N
- The length of the demonstrations (defined by how many times the states of the robot were saved): T
- The training set composed from the demonstrations: X
- The number of components chosen in the Gaussian Mixture Model: k

5.2.2 Recording Demonstrations

As it has been mentioned before, the Baxter has a built-in zero gravity mode, for the purpose of making dragging the robot's limbs easier for humans. In the zero gravity mode the robot applies only enough force in its limbs to keep them in place, and not fall due to gravity. Research Robotics intended to make research concerning Imitation Learning easier for researchers. Even though the teleoperation part of this project was meant to be used to provide the demonstrations, it was easier to use the built in zero gravity mode of the Baxter to take the demonstrations. Kinesthetic teaching was preferred, because using the VR teleoperation, the demonstrations were less precise, took longer to record, and presented more variability, just because of the limitations introduced by the framework. One of these limitations was the camera perspective: The Baxter's arms often covered vision of the table that the objects were placed on, therefore the arms had to be first moved away to see the ball and the box, and then do the action. This presented unnecessary noise in the demonstrations, with which the initial models could not deal with. Therefore, the recordings were taken by dragging the robot's arms and setting the grippers in the desired position.

Also, in the earlier stages, kinesthetics will be used to show the robot the position of the objects - the ball and the coffee cup by dragging the hands there. Later, the positions will be read by the object recognition module.

In order to store the recordings, a *Python* script will be used, running in the ROS network on the Ubuntu laptop. The Baxter's SDK provides a similar functionality that is necessary for the recordings. One of the example scripts takes a recording of the action, by writing

the states that the robot passes through into a Comma Separated Value (CSV) file *(baxter_examples/joint_recorder.py)*. Another script is provided to replay this action by moving the robot's limbs through these states. *(baxter_examples/joint_trajectory_file_playback.py)*. Another ROS script was created based on these files that takes care of recording the variables necessary for the later steps. It also creates a CSV file for each demonstration, and it takes in two inputs: the name of the file it should save and the frequency at which is should take the recordings. (As it is for the example file, the default rate is 100 Hz for the recordings). Then it takes the necessary handlers for the Baxter's limbs, creates the CSV file and stores the chosen variables in each line.

When the recordings were taken, it was not clear which state variables will provide the most useful data for the learning algorithms, therefore it was decided that all of the possibly useful states of the robot will be recorded. The following set of states were recorded during the demonstrations, which covered all of the useful information for learning:

- The time the recording was taken $t_1, t_2 ... t_T$ (1-dimensional)
- The position and orientation of the right hand at each moment in time (3+4=7-dimensional)
- The joint-angle of the 7 joints of both the right and the left hand (14-dimensional)
- The position and orientation of the left hand at each moment in time (3+4=7-dimensional)
- The state of the left and the right grippers (2-dimensional)
- The position and orientation of both hands necessary to grab the ball (The "position of the ball") ((3+4)x2=14-dimensional)
- The position and orientation of both hands necessary to drop the ball into the coffee cup (The "position of the cup") ((3+4)x2=14-dimensional)

All of these values are recorded at the default 100 Hz rate of the new recorder script. It is important to notice that the states mentioned in the last two data points are constant across all of the demonstrations, as they are the initial position of the ball, and the final position of it, when it is dropped into the cup. These are recorded in the demonstration file, as these values will be used in the later stage to calculate the distance of the hand from the initial and the final position on-line.

These values are recorded by first calibrating the values before each demonstration. This means that the robot's hands need to be dragged to the desired initial and then final position.

When they are at the desired position, a script is run, that stores the position of the arms into a file. The values are read from the file by the recorder scrips.

A snippet of the CSV file created as a demonstration is shown on Figure 5.13. A row represents a state reading taken as a time instance.

time	left_x	left_y	left_z	right_x	right_y	right_z	rot_left_x
0.470738	0.590570377984	0.157882706711	0.09436222532	0.579165671055	-0.188073357345	0.106746208052	0.12652960297
0.472204	0.590570377984	0.157882706711	0.09436222532	0.579165671055	-0.188073357345	0.106746208052	0.12652960297
0.481605	0.590839846542	0.158063592304	0.094805459483	0.578660496218	-0.188019472488	0.106619859332	0.126446864602
0.492389	0.590160700444	0.157773375576	0.0943857056929	0.578747986191	-0.187933120214	0.106528041174	0.12712534379
0.501588	0.590725555033	0.157450945762	0.0946387077466	0.57853519371	-0.188664385022	0.106331635073	0.127212640181
0.511589	0.591087562225	0.158310087959	0.0944127401569	0.578591514901	-0.188016453606	0.106542187682	0.126031139429
0.521719	0.590749020182	0.157694966098	0.094179415784	0.578711352154	-0.188306505346	0.106634060064	0.126280081
0.531801	0.590497371038	0.157893600396	0.0942623210864	0.578582736701	-0.18799333069	0.106724672982	0.126631344448
0.541788	0.590652342872	0.157769463696	0.0943818113483	0.578412332014	-0.188458595823	0.106340373528	0.126635329243
0.55159	0.590649138781	0.157551904417	0.0946486879481	0.578756676115	-0.188486319861	0.106730346032	0.127031430207

Fig. 5.13 The recordings are stored in CSV files with each row containing the value of all of the robot's state variables captured in a time frame. The recordings are taken with a 100 Hz rate, which can be observed on the left-most column of this graph. The variables shown here are the position coordinates (x, y, z) of the left and right hand of the Baxter, but several other variables are also recorded

Three different variants of the same action are recorded. A set of 10 recordings are taken of the robot putting the ball in the cup, when both of the objects are in the same position throughout the 10 demonstrations. Next, 10 recordings are taken when the ball is moved, but the cup stays in the same location. For the last set of 10, both of them are moved across the table. This is necessary so that the generalisation of the algorithm could be implemented in stages. The first few algorithms will not necessarily generalise well, but at later stages, the action could be reproduced even with the initial positions changed.

5.2.3 Imitation Learning Algorithms

Data Manipulation

After taking the set of 10 demonstrations in all three scenarios, the CSV files need to be read by the first Python script, which takes care of the data manipulation to reshape the variables to a state that could be used by the actual Learning algorithms.

Out of the recorded variables, not all of them are actually necessary for efficient learning, many of them are redundant. For example, the right arm's data can be immediately disregarded, since during all of the recordings, it was left in one place, therefore the recordings of

its states do not contain any useful information. For the later learning algorithms, only the following data was needed:

- The indices of each recordings, ranging from 1 to T. (1D)
- The 3D position and orientation of the Baxter's left hand recorded at each time instance. (1D)
- The state of the robot's left gripper recorded at each time sequence (1D)
- The 3D distance between the initial position of the ball and the current position of the Baxter's hand recorded at each time instance (denoted by d_b) (3D)
- The 3D distance between the position of the ball when it is released into the cup and the current position of the Baxter's hand recorded at each time instance (denoted by d_c) (3D)

It is important to note, that the distance between the initial position of the ball and the current position of the hand can be calculated by subtracting the position of the ball recorded by the recording script, and the current hand position. Similarly, the distance between the final position of the ball and the current position of the hand is calculated by the subtraction of the current position of the hand from the cup's position. It is important to note that this constant relationship holds even when the position of the ball and the box are changed from demonstration to demonstration.

Therefore, the data manipulator script first selects the above listed variables useful for learning, then replaces the time values with indices ranging from 1 to T. Then it calculates the distances by performing the subtractions.

After performing these steps, the reformatted data is ready for temporal alignment. Useful Python Machine Learning libraries can be found in [58].

Temporal Alignment

The demonstrations cannot be performed 10 times perfectly aligned, some of them will be longer, some of them will be shorter, and some of them start a bit later than the others, therefore some misalignment will be present from demonstration to demonstration. Many of the learning models would be corrupted by misaligned signals. The simplest model is just one that takes the average of the signals, which is clearly corrupted misalignment. The approach chosen for this project - Gaussian Mixture Modelling - also needs correctly aligned signals.

Temporal alignment of the samples is implemented in the same Python script after the manipulation has been performed. Once the data is ready for alignment, the script applies Dynamic Time Warping and makes sure that all of the signals last equally long, based on the work in [24].

There are multiple ways to align the demonstrations temporally.

- [65], [34] and [82] used Partially Observable Markov Models to encode the sequence of data, which takes case of the temporal alignments. Encoding them into this model gives a probability transition from one state to other, and it is tricky to create a trajectory from them [80].

- [9] implements task-specific alignment algorithm for a similar action. This algorithm divides the task into three parts, and re-samples the three portions to the same length. The first task ends when the gripper grabs the ball, and the second ends when the gripper releases the ball. A similar algorithm could be used for this project, however Dynamic Time Warping provides a non-case-specific and more precise way of aligning the signals.

- Dynamic Time Warping is the best solution, as it is applicable for general scenarios as well. As it has been discussed in Section 2.4.6, DTW returns a *Warping path*, that when applied to the signals, distorts them, so that their distance is the smallest. The two signals will be equally long after applying the distortion to them, but they will be longer (or equally long) than the original time-sequences. This is because parts of the signals generally have to be repeated for multiple time frames. Dynamic Time Warping was used in [80] as well.

The Python script therefore performs Dynamic Time Warping to align the signals, using the fastdtw library. [62]

The demonstration with the median length was selected as a reference demonstration, and its length T was saved as the reference length. In order to produce ten demonstrations with the same length, the Dynamic Time Warping algorithm aligned all of the signals with the reference demonstration, and resampled them to the reference length.

The demonstration with the median length was selected as a reference demonstration, and its length T was saved as the reference length. In order to produce ten demonstrations with the same length, the Dynamic Warping algorithm aligned all of the signals with the reference demonstration, and resampled them to the reference lengths. This way all ten demonstrations are equally long and are aligned in time. This can be explored nicely on Figure 5.14. On the left side of the Figure, the y coordinate of the left hand's position is shown before alignment

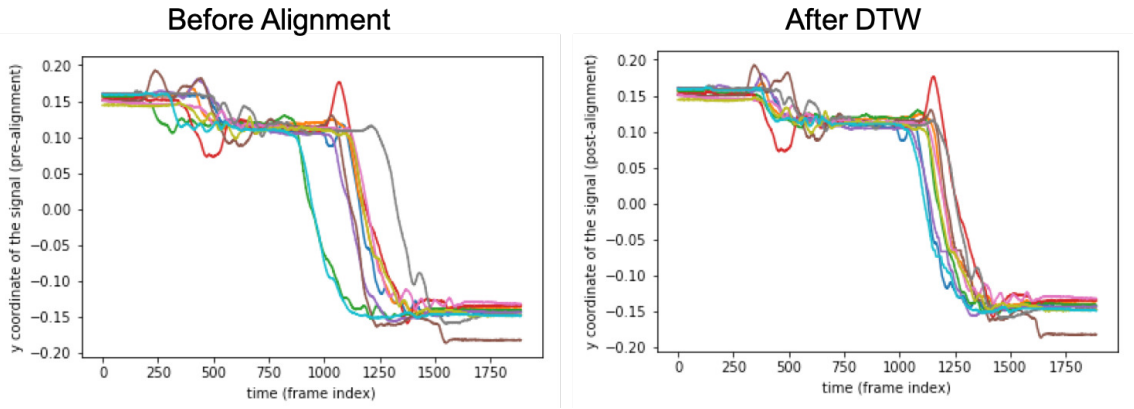


Fig. 5.14 This Figure shows the y coordinate of the signals before and after Dynamic Time Warping is applied. As it can be observed, before alignment (left graph) there is some delay between the recorded demonstrations, which are inevitable when repeating the same action ten times. After the Dynamic Time Warping algorithm distorted them according to the Warping path, they are observably more similar to each other (right graph)

of the ten demonstrations, while on the right hand, it is shown after alignment. It is very well observable that the algorithm brought the signals closer.

Gaussian Mixture Models

Section 2.4.8 introduced the usefulness of Gaussian Mixture Models to encode the set of demonstrations to a probabilistic model to represent the sub-tasks and provide basis for Gaussian Mixture Regression. The models are composed of k multivariate Gaussian Distributions, where k was selected taking into account the Akaike and the Bayesian Information Criteria, as implemented in *sklearn*. As introduced in Section 2.4.4 and 2.4.5, the GMMs were trained using the Expectation Maximisation algorithm, with k-means initialisation. The models are represented by the mean and the covariance matrix of the distribution.

Calinon et al, provided an implementation of the Gaussian Mixture Models and Gaussian Mixture Regression readily available in Matlab based on [17] and [19]. Therefore after the data manipulation and temporal alignment, the Python script writes the created demonstrations into a *.mat* file and passes it onto the Matlab script for training. The selection of k is still implemented in the Python file, based on *sklearn*'s built-in AIC and BIC function.

Figure 5.15 shows that the optimal number of parameters for the GMM were selected in sklearn to be 9. The GMM model is shown on the lower graph, by indicating how the x coordinates of the demonstrations were divided into the 9 separate models that they are most likely to come from. The rest of the implementation is covered by the Matlab script, which calculates the means and covariance matrices using the EM algorithm with

k-means initialisation. The script also provides easy visualisation tools for the models and the regression paths, which will be used later.

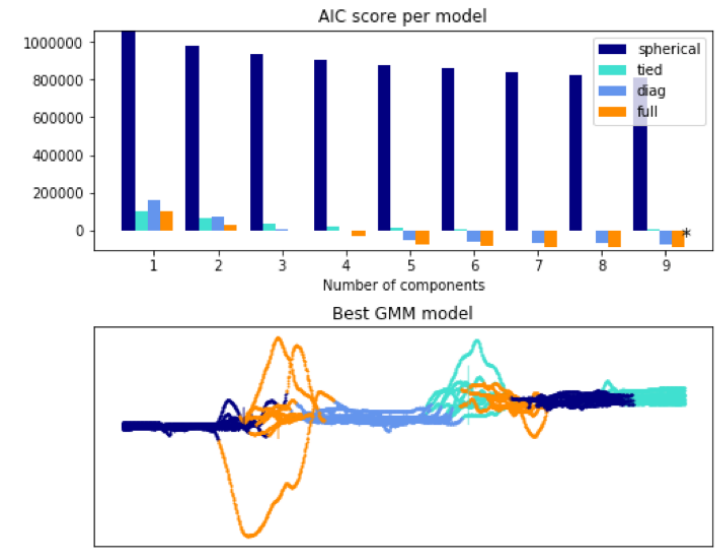


Fig. 5.15 This Figure shows the selection of k, the number of model parameters for the GMM. The graph is generated while training GMMs with increasing number of components (from 1 to 10) at every iteration the AIC and BIC score of the model is evaluated, and the AIC scores are displayed and are shown on the top graph. Each colour represents a different shape-constraint for the covariance matrix. The lowest (best) AIC shore was calculated with k=0, using a full (no-constraints) covariance matrix

Gaussian Mixture Regression

As described in Section 2.4.10, Gaussian Mixture Regression provides a flexible way of creating a generalised trajectory from the trained GMMs. The Gaussian Mixture Regression algorithm returns the joint probability distribution of the variables that are encoded in the GMMs. Then, the input and output variables are selected. In the final implementation, the best solution is found, if the input variables are: the time instance, the distance of the Baxter's hand from the ball in the initial and in the final position. This way, the algorithm will output the Baxter's hand position and orientation coordinates, and a Python script in ROS can be used to drive the arms to the specified position. The Gaussian Mixture Regression training is implemented in the same Matlab project, written by Calinon et al..

Chapter 6 shows the results and the experiments carried out with three different training scenarios: Experiment 1: the initial location of the ball and the coffee cup were both fixed throughout the 10 demonstrations. Experiment 2: The location of the coffee cup was kept fixed, but the initial location of the ball was changed throughout the 10 demonstrations.

Experiment 3: Both locations were changed. The regression algorithm returns a mean vector and a covariance matrix for every time instance. The returned mean is going to be treated as the generalised trajectory and the covariance matrix as the constraint. At every time instance, the output (the location of the Baxter's hand) is determined from the inputs. If there is a stronger constraint on one of the inputs (the covariance matrix is a narrower matrix), the generated trajectory weigh that variable more when constructing the averaged trajectory.

This is the reason why the distance from the two objects is important to record. For all of the demonstrations, when the Baxter closes its grippers and pick up the ball, the distance of the hand from the initial position of the object is very small. When it drops it off, the distance of the hand from the coffee cup is very small, and this is the case for all of the demonstrations. Therefore the constraint on the distance variable is going to be very harsh in the time frames when the ball is going to be picked up or dropped off. This means that the GMM and GMR will generate a trajectory that takes this into account and goes close to the two objects when the distances and the time frame is given as an input to the model. Further exploration of this idea can be found within the Test and Results chapter.

5.2.4 Action Reproduction

Overall, the workflow of the implemented system can be observed on Figure 5.16 During the training phase, the Gaussian Mixture Models are trained, and the joint probability density function of the variables is returned. Then, during the reproduction phase, a Python script in the ROS network queries the location and orientation of the Baxter's hands, to calculate the distance between the initial position ball and the distance between the coffee cup. The scripts also keeps count of the time frames (as a simple counter). This script uses the time frame number and the distance data to give the combination of these variables as input to the GMR algorithm. The output of the GMR algorithm is the position and orientation of the Baxter's hand where is should be driven to. The scrips therefore technically acts as a smart control algorithm and drives the hands to the desired position and that knows the desired movements.

5.2.5 Imitation Learning Overview

In conclusion, the implementation of the Imitation Learning system consists of on-line calculation of the next states of the Baxter's hand position based on the Gaussian Mixture Regression model trained from the demonstrations (as a controller). The algorithm needs as input the time and the distance between the Baxter's current hand position and the initial position of the objects, and returns the next state (hand position). Therefore, a trajectory for the Baxter's hand will be generated.

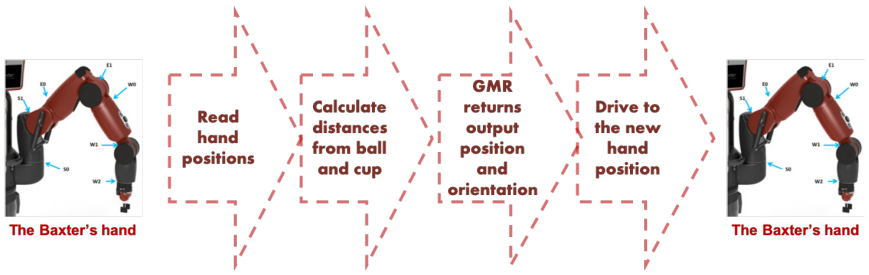


Fig. 5.16 This flowchart explains how the code for the action reproduction part works. The reproduction of the learnt task is implemented in a script in an online manner. The script calculates the distance of the gripper from the ball and the coffee cup, feeds the distance and time data into the previously trained GMR model and controls the Baxter's arms to the returned new position coordinates. The location of the ball and the cup need to be set before starting the action by moving the Baxter's hand over them and taking a reading by the other scripts. The Future Work chapter explains how object detection can eliminate this step

Chapter 6

Testing and Results

This chapter shows the experiments created to test and further improve the implementation of the project's components. Some of the ideas discussed in the previous chapter initiated during the testing phase. As expected, this section is also divided into two parts according to the two project phases. The Teleoperation part of this Chapter is shorter, because most of the system's characteristics were fine-tuned during the implementation phase. Setting up a unit-testing system for the teleoperator is unnecessary, as long as it can be used without errors to provide demonstrations. many test results were explained in the *Implementation* chapter and thus the main extra bit of information is provided in this chapter by the three experiments on Imitation Learning.

6.1 Teleoperation Experiments

Other than making sure that every module described in Chapter 5 works, the most important aspect to investigate here are the delays in the system, as significant delays can have negative effect - as for example motion sickness - in the user.

The delays in the arm movements are shown on Figure 6.1. The graph compares the 3 coordinates of the controller and of the Baxter's left arm. The delay on average between the systems is between 0.2-0.5 seconds, and this delay can be attributed to three effects. The first component - the delay due to the processing and transport - is negligible to the latter two. The second component is how long the IK solver takes to return a valid solution, if found. This delay is a result of the inverse kinematics solver trying to calculate the correct joint angles for the 7 degrees of freedom arm, which is a long and complicated calculation, made more difficult by the fact that the Baxter does not have proper shoulders or wrists. [41] [37]. The IK Service Proxy can take between 0.01-0.05 seconds to return the results of the calculations. The rest of the delay is due to the robot's arms having inertia, and the motors in

the joints cannot move them immediately. The delay mostly shows how long it takes for the arm to reach the desired position. The delay in the x coordinate seems to be slightly higher than for y and z coordinates and the signal is slightly noisier. This might be due to the fact that x coordinate is set my a combination of joints, while the other two joints can be simply controlled by 1 and 2 joints respectively, which makes the movement easier.

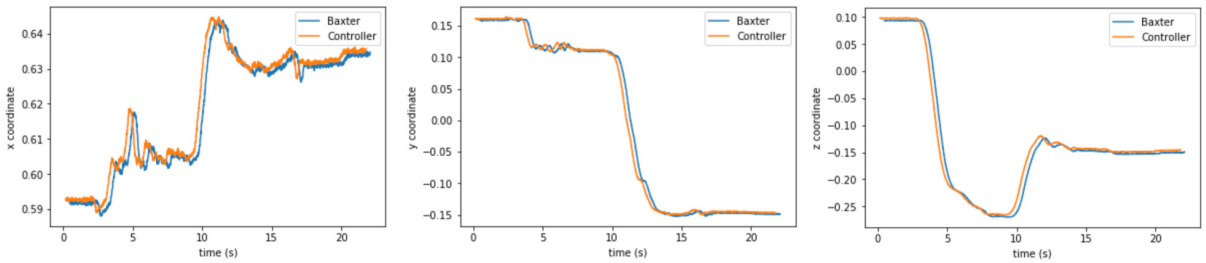


Fig. 6.1 This Figure shows the delay between the Vive's controllers and the Baxter's arms. The graphs show the recorded (x, y, z) coordinates of the two objects and the 0.2-0.5 second delay between them. The delay in the x coordinate is seemingly higher, and the signal is more noisy. This is most likely due to the fact that the x coordinate can be only set by moving more than one joint, while the others can be controlled by a single joint directly

The delay in the head movements is observably less than that of the arms, as it can be observed on Figure 6.2. This is most likely due to the fact that the setting the head angle is very simple, does not need an inverse kinematics solver to calculate the angle, the processing is happening on the Unity side, and the robot only needs to control one motor to the right position. This is advantageous because the delay in the head movement is most prone to leading to sickness in the user, therefore the results of an average delay of 0.1 seconds is promising. It is for the reduction of this delay that the scripts produced are as basic as possible, and that most of the processing is happening on the Windows development machine.

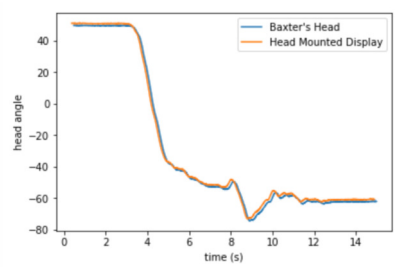


Fig. 6.2 This Figure shows the delay between the Vive's HMD and the Baxter's head. It can be observed that the delay is smaller that the delay in the arms and the noise in the Baxter's head is significantly smaller. This is due to the smaller amount of processing needed for the head angle. The lack of Inverse Kinematics solver and the simpler joint movement of the head allows smaller delays in the order of about 0.1 seconds

6.2 Imitation Learning Experiments

More experimentation was carried out to fine-tune the Imitation Learning system put into place. This is the more research based part of the project, and the number of design choices in the system is bigger, more parts are optimisable. Three experiments were set up to come up with the final design that was proposed in the Design chapter. The task of the experiments was putting a ball into a coffee cup, which is easily generalisable in the later experiments by changing the colour/position/general form of the object. d

6.2.1 Experiment 1

In the first experiment, 10 recording were collected using the ROS script to record the action of the robot moving its left arm and grippers over to the ball, gripping it, moving it to over the coffee cup and opening its grippers over it. For this experiment, the position of the cup and the ball were both kept constant. The Gaussian Mixture Model and the trajectory generated by the Gaussian Mixture regression can be seen in Figure 6.3 (only coordinates x and y are shown).

It is observable on the first graph that all of the 10 demonstrations lasted around 18 (hence the 1800 time frames using 100 Hz recording rate), and several parts of the action are explainable. The first flat region corresponds to the state before the robot is moved. The robot always starts from the same initial position, and stays there for a couple of seconds before the action is started. This is seen on the graph in the first 350 time instances. Then the robot is moved in the next 250 instances, and the 10 demonstrations start to differ, only to get well aligned again from about 600 to 1100 time instances, when the gripper picks up the ball. The signals are similar, since the position of the ball is always the same. Next, the recording shows that the robot is moved again, and that it drops the ball into the cup, which is again always in the same spot throughout the demonstrations.

The GMM shown on the graph only contains 4 components for a clearer demonstrations, as opposed to the 9 components in the final implementations. It can be seen on the lower graphs, that the fact that the robot should move to the ball and the cup was correctly identified by the algorithm, shown by the fact that the generated trajectory is more confident around those regions. This means that during the reproduction of the action, the Baxter will move the initial positions, and connect them with an averaged trajectory of the demonstrated movements.

This experiment produced the least of the interesting results, so the rest of the algorithms' and models' characteristics are shown in the other two experiments.

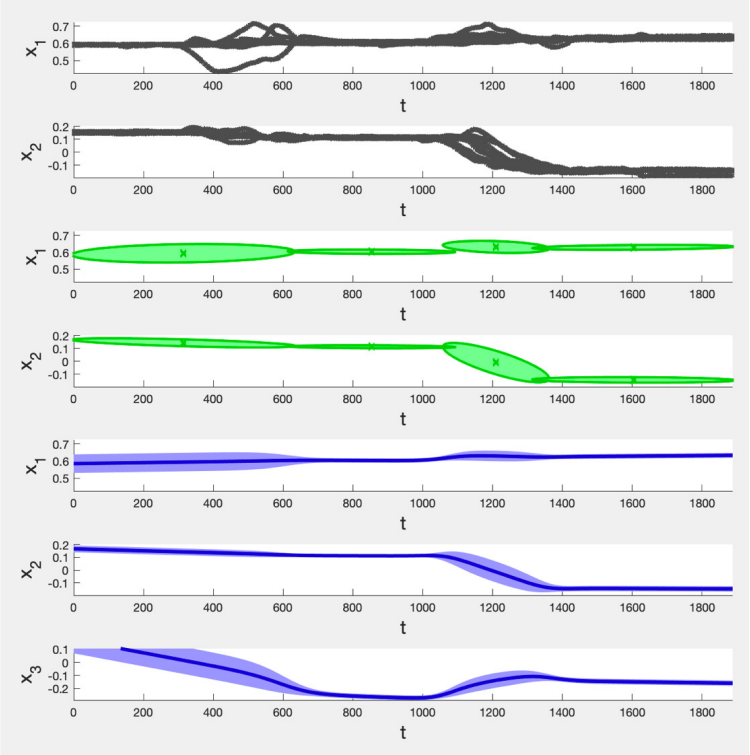


Fig. 6.3 This Figure shows the results of the first experiment, when the GMM and the GMR were trained from from ten demonstrations where the position of the cup and the ball is kept constant. It can be seen by the narrower constraints (light blue area) that the models learnt the initial position of the box and the ball and the generated trajectory includes them with a higher confidence

6.2.2 Experiment 2

In the second experiment, the initial position of the ball was varied when recording the demonstrations, but the position of the coffee cup always stayed the same. The recorded (and temporally aligned) demonstrations can be observed on Figure 6.4. Note that the signals are further away from each other than in Experiment 1, since different paths are taken in all of the demonstrations. The beginning and the final part of the graph shows that the signals start off at the same location (the initial position of the robot is assumed before all demonstrations are taken) and that the signals finish at the same location (since the coffee cup's location always stays the same. Because of this, we expect the covariance matrices of the GMM model to be wider, indicating less confidence in the generated trajectory.

Let's observe Figure 6.5. It shows the GMM model and the generated trajectory of the GMR, when the input is simply a new time sequence of data. The output are the targets of the distance variables, namely the targets for the distance between the Baxter's grippers and

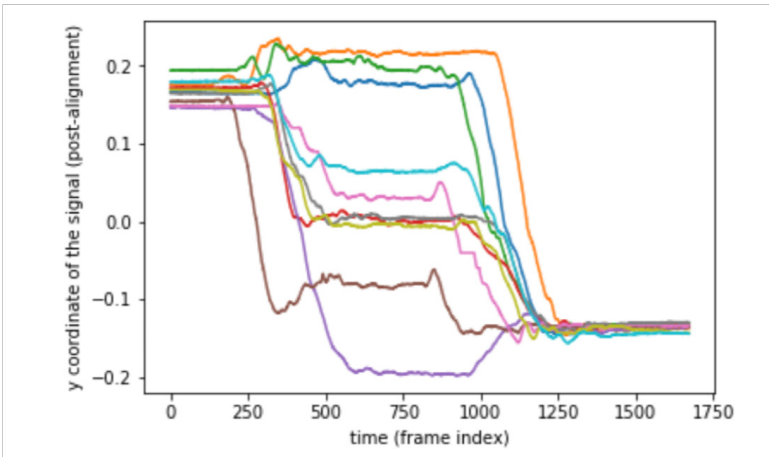


Fig. 6.4 In the second experiment, the location of the ball is changed before starting the recordings. This is nicely reflected by the path taken by the robot's arms to reach the cup. The cup's position is kept constant and this is why the signals start to align at around 1200 milliseconds, when the ball reaches the cup

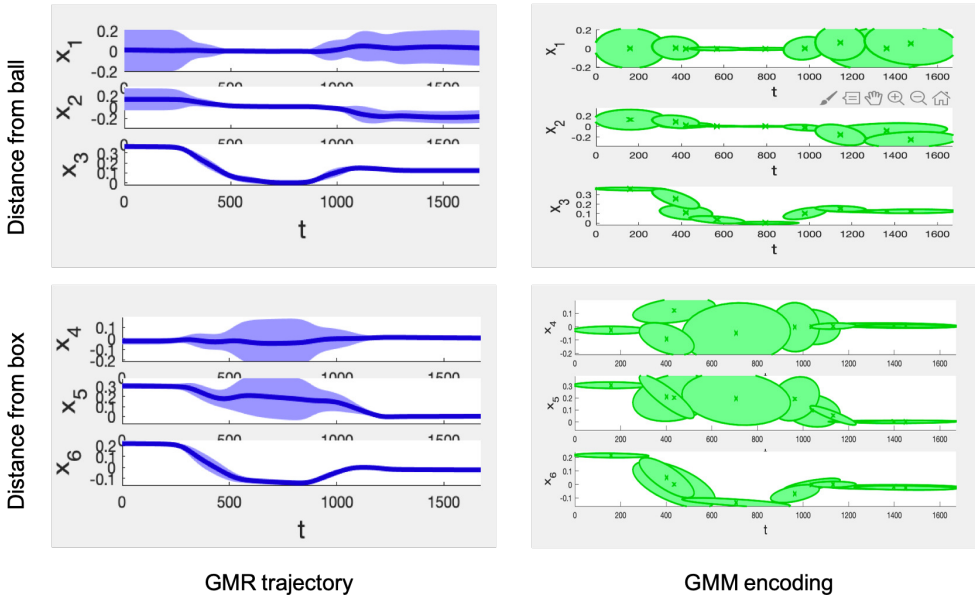


Fig. 6.5 On the top two graphs, it is nicely observable that the robot learns with high confidence that between around 600-1000 milliseconds, it is important to keep the distance from the ball's initial location around 0. However, the bottom two graphs show that between around 1200 and 1400 milliseconds, the distance from the cup became more important to keep minimum. Based on these, the controller will control the arm to go close to both of these destinations at the right segment of time

the ball in the first row and the target for the distance between the grippers and the cup in the second row.

First of all, it is observable on the graphs that the covariance matrices got wider at several locations. This is due to the fact that at these points it does not really matter where the hand moves, and it is a result of the big variance in the data at these time instances. The signals are less aligned, and the generated trajectory has a broader light blue surrounding, indicating less confident trajectory generation in certain areas. This is due to the fact that the robot's hand took different paths because of the different initial positions and thus only the final and the resting (initial) position of the robot are the same.

With this experiment, it is even easier to demonstrate that the Gaussian Mixture Model encoding and Gaussian Mixture Regression picks up which variables are more important to follow at each instance of time. Between around 600 and 1000 milliseconds, the top row of the graphs shows very constrained trajectory. This is the time when the gripper moved over to the box and closed its grippers to pick up the ball. Around this time, it is not that important how far away the hand is from the ball, which is why the bottom row shows very high variability in the GMM covariance matrices and the generated trajectory. On the other hand, the model understood well that it is important to control the robot close to the ball's position and, after around 1300 milliseconds. Around this time, exactly the opposite effect is observable. The distance from the box is very constrained, but it does not really matter what the distance between the object and the initial position of the ball is. This way the controller will control the Baxter's gripper at reproduction to go close to initial position of the ball in the beginning of the reproduction, and go closer to the cup at the end.

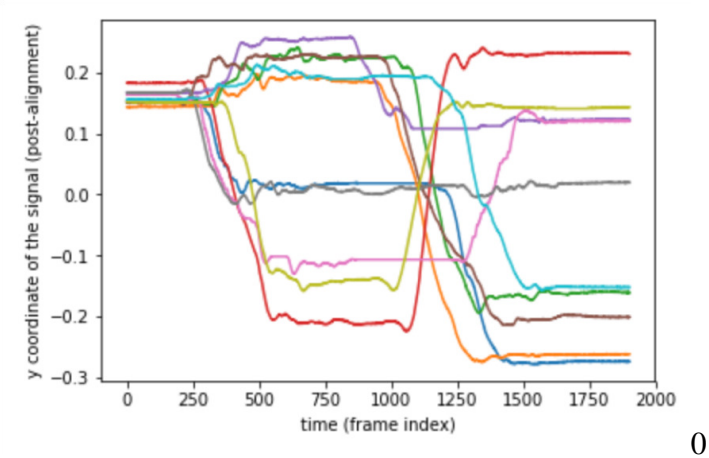
Overall, the algorithm learns which of the variables are important at each time segment of the demonstrations. Based on this, the controller will generate a trajectory that follows closer the more constrained variables are reproduction instead of just simply following the average trajectory of the Baxter's gripper position, as it was the case in Experiment 1.

It is also interesting to note that the third (z) coordinate of the trajectories stayed very similar in all of the demonstrations, which resulted in a very constrained trajectory along this coordinate. This is due to the fact that the objects are placed on a tabletop and the z coordinate of the objects is the same. Since all of the demonstrations last around the same time, and the ball is lifted to around the same heights for all of the demonstrations, the z coordinate of the demonstrations is going to be about the same, but there is bigger variability in their x and y coordinates.

6.2.3 Experiment 3

In the third experiment, both the cup and the ball were moved before recording each demonstrations. In this Section it is desired to show that the encoding still picks up which variables are important at each time frame.

Firstly, Figure 6.6 shows the demonstrations. It can be observed that they are very dissimilar, other than the initial starting position, they move on completely different trajectories.



0

Fig. 6.6 In the third experiment, the location of both the ball and the cup is changed before starting the recordings. This is nicely reflected by the path taken by the robot's arms to reach the cup. The initial position of the recording is the same throughout all of the demonstration, that's why all the signals start from the same value

Figure 6.7 shows the generated trajectory for the distances, similarly to the graph produced for Experiment 2. Here it can be explained that the covariance matrices are even wider, except where the object is picked up or dropped off in the ball. At these time instances the model learns that it is important to keep the grippers close to the ball and the cup respectively.

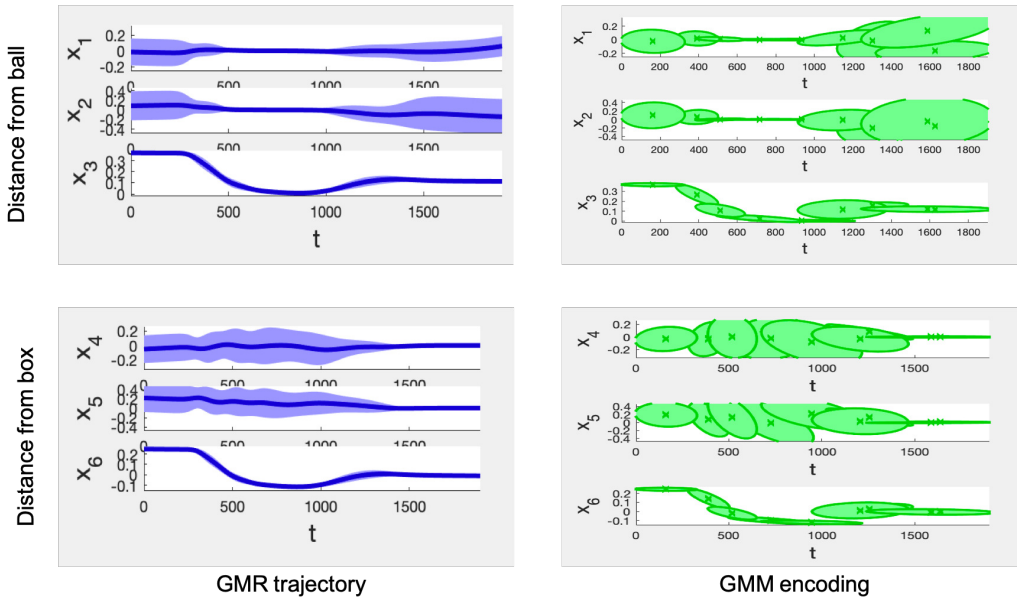


Fig. 6.7 On the top two graphs, it is nicely observable that even in this more generalised scenario, the robot learns with high confidence that between around 600-1000 milliseconds, it is important to keep the distance from the ball's initial location around 0. However, the bottom two graphs show that between around 1200 and 1400 milliseconds, the distance from the cup became more important to keep minimum. Based on these, the controller will control the arm to go close to both of these destinations at the right segment of time

Chapter 7

Evaluation

Overall, this project created a system that can be used to teleoperate the Baxter Research Robot using a Virtual Reality headset as a mean of providing demonstrations of an action. The implemented algorithms allow the robot to learn this action, ad be able to reproduce it. Learning the key characteristics of the action, the robot is able to generalise its knowledge, so that changed initial conditions don't change the quality of reproduction, and the robot is able to generate general trajectories that completes the action even under changed circumstances.

The first part of the project - Teleoperation - implemented all of the requirements laid out in the *Requirements Capture* chapter necessary for a comfortable user experience. The Baxter and the HTC Vive were connected through the ROS# project developed by Siemens. This allowed streaming information from Unity to the Baxter and vice versa. The data flowing between the two systems were the arm position and orientation and head angle information that is necessary for the ROS scripts running in the Baxter to control the robot, and the video image that is rendered in front of the user's eyes. The final system allows the operator to take on the robot's point of view by seeing from the robot's eyes by wearing the headset and being able to control the Baxter's limbs by holding the Vive's controllers to follow his own arm movements. The robot's vision is represented by an Intel D435 camera placed onto the robot's head. The image from it is streamed in a compressed format, and a bundle of GameObjects and scripts are created to render the images in front of the user's eyes with as little delay as possible.

The head angle data, the arms' position and orientation coordinates and the gripper command data is used in the Baxter's ROS scripts to command the joints to move to the right angles to mimic the user's hand movements. The position and orientation coordinates are fed into the Inverse Kinematics solver of the Baxter to obtain the joint angles from the arm position and orientation coordinates (if a solution can be returned).

Controlling the orientation of the grippers is a more problematic task, because the robot does not have human-like wrist and shoulders. Therefore, to mimic the user, the robot's joints have to move significantly even for a small change in the user's hand orientation, even at the same location. This resulted in a loss of user-friendliness of the system when the orientation was set, due to the delay introduced by the arm's movements. Often, the Inverse Kinematics solver could not return a value, which blocked the robot's arm from moving, and the system became less responsive, less sensitive and less smooth. To keep the teleoperation more comfortable a button control was implemented to switch off setting orientations, giving the user the decision between the trade-off of having a smoother or more precise control over the arm, even when the robot is running. The TrackPad button of the Vive's controller can turn on and off this functionality. When it is off, the Arm moves in a fixed orientations - with the grippers pointing towards the table next to the robot - which allows it to deal with objects, like picking and dropping them easily.

As introduced in Section 9, the system could be further improved by rendering a stereo image in front of the users eyes. The camera used for the robot's vision (the Intel D435) provides depth information and can provide a colour stereo feed as well. The depth image is useful for object recognition, and the stereo feed could be used to render the two images to create 3D vision for the user inside the HMD. The two feeds would have to be rendered with a disparity between them, which would mean the creation of another camera and another plane in Unity. The currently used mono video is satisfactory, but makes it difficult for the user to estimate distances based on this image, and therefore makes object handling a bit more challenging, and the recorded requirements a bit more variable. This functionality was however not implemented yet and left as a future task as it is not a crucial requirement, and more time was spent with the investigation of Imitation earning algorithms.

Another limitation of the project is the lack of automatic detection of the underlying objects (the ball and the cup). Once the object detection system could determine the position of the objects, the system could work autonomously, without the necessity to capture and feed in the positions manually before reproduction.

The Imitation Learning part of the project put a system in place that learns the action of lifting a ball and dropping it into a coffee cup. This task is simple for the robot to complete and can be used as a benchmark for more difficult task experimented with later. This would allow easy adaptation to more complex experiments without too much change in the learning algorithms and the model parameters. This task has elements that are present in a wide variety of household tasks, such as recognising, lifting or carrying an object. It can also be easily expanded to the useful household chore of creating a morning cup of coffee for the

user. The final implementation of the Imitation Learning algorithm obtains its knowledge in five stages.

1. First, the demonstrations are recorded by guiding the robot's arms over the task ten times
2. Then, the demonstration are aligned in time using Dynamic Time Warping, so that the same key features happen at the same time in all of the demonstrations
3. A Gaussian Mixture Model is trained from the variables recorded to learn the variation of the recorded data
4. The Gaussian Mixture Regression algorithm generates a trajectory on-line taking into account which variables are important to follow at each time-portion of the demonstrations, using the Gaussian Mixture Model trained
5. Controlling the robot to the required trajectory

This way the algorithms outlined above generate a trajectory that follows the demonstrated stated, but it also prioritises the most important variables captured in the right time frames. At each time-segment, a different variable is important to follow more closely, which is detected automatically by the above implementation. For example when the task in question is moving the ball into the coffee cup, it is very important to move the the position of the ball, so in the time frames where the ball was grabbed in the demonstrations, the generated trajectory has to monitor closely the variable recorded to represent the distance of the ball and the robot's grippers. When the ball is released, it is important to follow the variable representing the cup's position. Therefore overall different variables are important at different times. This algorithm is useful for more general tasks where the variables are not necessary position coordinates. The robot's gripper states are also fed into the Gaussian Mixture Model, and the regression returns the necessary state value that needs to be set to grab the ball at the right place and right time. The state is dependent on the time of the action and the current position of the Baxter's hand position.

This method allowed a fair place for experimentation as the Gaussian Mixture Model's flexibility allows to define and experiment with different inputs to the model. The regression algorithm returns the missing variables as outputs, based on on the joint probability distributions of the underlying variables. Gaussian Mixture Regression could be used trained with other variables as well, and therefore allows the generalisation of this project for more tasks.

The system in place allows changing the environment up to a certain level while reproducing the actions correctly. The Imitating Learning experiments showed that with the proper

demonstrations used, the learning algorithm tackled the problem of moving the ball into the cup even when the position of the ball and the cup were changed. The demonstrations recorded with changing initial positions for the ball and the cup were significantly different, but the Gaussian Mixture Regression implemented still extracted the important variables at the right time segments and ignored the less important (more varying) components. The task was reproduced with a high success rate, even under changing initial conditions, however sometimes the gripper did not manage to catch the ball properly.

The generalisation of this project was kept in mind throughout the implementation and experimentation phase. This particular implementation only deals with 3D positions (and orientations), as the variables that carry all important information for this task. However, for other tasks, other variables can be important, such as the physical width, height, depth, colour, speed, force, etc. GMR deals with the generalisation of these variables very well, as it does not matter what are the underlying variables, it simply takes as input a list of variables, with a sequence of time indices, and thus it can learn the joint probability density function of other variables than 3D positions as well. The learning system was mostly created from already implemented code and based on previous research, but the on-line controller could serve as a unique feature, not considered by other papers yet. This controller allows more flexible experimentation and easier generalisation of the system, but comes with the disadvantage of a slower overall reproduction system in place.

Overall, the system has some limitations, originating mostly from the fact that the robot's structure is not completely the same as of a human body. Also, the precision of the Imitation Learning is constrained by the quality of the demonstrations, the interdependence of the variables, and the overall dependence of the system on timing. For the reproduction phase, a sequence of the temporal values is used to query the GMR and generate a trajectory for the other variables. Also, unfortunately there are some positions that cannot be set by the Baxter's hands. When the trajectory generated includes such a position, the controller usually gets confused, and the action is blocked. It is also important to place the object to a position that the Baxter can reproduce, but all the positions on the table used in the experiments are reachable.

Even though, most of the delays were eliminated, some lags are still present and cannot be decreased to zero, which could make experimenting and using the robot a bit uncomfortable. The most bothering delays are the delay from the robot's inertia, and the inconvenient movements necessary the set the correct gripper orientation. Since the reproduction introduces additional processing, transportation and control delays, the task reproduction is significantly slower than the demonstrations, however this is acceptable because it doesn't require human interaction, and the slower reproduction ensures a higher success rate.

With all of the above points considered, the project implements all of the deliverables and fulfils all of the requirements mentioned in the *Requirements Capture* chapter, and is therefore considered successful.

Chapter 8

Conclusion

Overall, the project successfully implemented the requirements listed, namely the teleoperation of the Baxter robot with a VR system, and the learning of a task from a set of demonstrations achieved using the teleoperation system. The teleoperation system allows the robot to mimic the user's head and arm movements through the headset and the controllers respectively, and lets the user place himself into the robot's position by showing in the Head Mounted Display the video stream of a camera placed on top of the Baxter's head.

In the second part of the project, demonstrations were provided and the time-dependent data was manipulated and encoded to learn the key characteristics of the demonstrated task. An online reproduction algorithm feeds into allows the robot to reproduce the task autonomously, and the Imitation Learning algorithms ensure that the robot's knowledge is generalised enough to allow reproduction of the task under changed environmental circumstances.

Experiments were conducted to show and optimise the performance of the implemented system. The methods are implemented to a high standard, optimised for performance, and their flexibility is taken into account for further generalisation of the project scope. The experiments showed that the proposed teleoperation system eliminates major lags and the proposed learning method can generalise reasonably well. The major strengths and disadvantages are outlined, and the design choices made are verified.

Overall, the project was ambitious, as the creating of the VR teleoperation system presents the developer with numerous obstacles and it not simple. For a perfect system, the given resources present limitations - such as the Baxter's body structure and the camera's distortion, the delay in the Inverse Kinematics solver and transportation methods. The implemented learning methods were satisfactory for this project, but overall significant further research is needed in the field of Imitation Learning, before household robots could be programmed by their owners. Using Virtual Reality present a feasible and practical way of teleoperation,

however more advanced systems need to be implemented, using even more human-like robots than the Baxter.

Chapter 9

Future Work

9.1 Improved Camera Image

The quality of the Intel D435 camera is significantly better than that of the Baxter's head camera and its delay is also better. However the image quality could be even further improved for more realistic display. More importantly there are various distortions in the camera's image inherently that could be corrected for in the Unity scene. An image taken by every camera is distorted by the lens, mostly around the edges where the objects are further away, which could be counteracted digitally. The distortion of the Intel camera is milder than that of the Baxter's built-in head camera, as the Intel's lens is slightly bigger. At the moment, the system does no image processing to the video streamed to the headset. In modern handheld devices (like mobile phones, tablets) the camera distortion is corrected digitally by post-processing the image. The possible solution against the distortion would be to counter-distort the video plane of the Unity scene, using the distortion data from the camera's data-sheet. The plane would have to be curved by the edges to bring the edges closer to the user, with the curvature calculated by the focal distance and axis position of the lens. However the implementation would require significant further research in the operation of lenses and imaging, and the implementation of this improvement would take long for little improvement in the system, thus this task was left as future work.

The pan of the camera's lens covers slightly less than a human's angle of vision. Another camera with a larger angle could be tested to display a wider image. Another related improvement could be to zoom in further to the camera image Unity when rendering the images to the plane, to make sure that the image fills the edges of the plane when the headset is rotated, to eliminate the edges appearing in front of the user. This could be achieved by zooming in even further to the camera image (definitely wider angle would be required) and adjusting the curvature of the plane accordingly. Additionally, the periphery could be

blurred to make the vision more realistic. This could be done using eye gaze information to determine what the user focuses on and determining on the image what's closer and what's further. This is very advanced and less crucial, so it was not considered for implementation.

9.2 Stereo Image Display

As it has been mentioned before, the system could be ameliorated by introducing stereo vision for displaying a 3D image in the HMD. At the moment only the mono feed of the camera is used to allow the user to see through the robot's eyes, but the camera published a stereo feed too. Using a mono feed is a disadvantage, because it makes id hard for the user to feel distances. A human also sees through a "stereo stream". The disparity between the eyes can be used by the brain to decode depth information of the objects. This could be mimicked by a stereo vision system that would allow much better user experience for teleoperation. The Intel D435 readily published a stereo feed suitable for the ROS network to read it and render it, therefore the ROS system would not have to be changed. Three feeds are also readable by the camera, so the separate depth feed could still be used for object detection. In Unity another camera object and another video plane would need to be added to the scene, and some code modification would have been needed to let the headset render a 3D image - a stereo image to the display with adjustable disparity. This task is not difficult in principle, and it is definitely one of the next steps in the improvement of the teleoperation part, but the implementation and debugging takes some time. Due to the Imitation Learning part being more important, the implementation of the stereo image display was suspended and delayed.

9.3 Improvement of the Imitation Learning Algorithms

This project was not concerned with the use of Recurrent Neural Networks and Reinforcement Learning techniques for the Imitation Learning part. These methods are out of the scope, and constitute a separate project. However, the current system could be further experimented with, for example by recording even further robot state variables to see if the key characteristics of the action could be better captured by other variables. Also, more experimentation could be run to select a better GMM parameter k more wisely than using the Akaike Information Criterion, and a more advanced controller could be implemented than the current online GMR controller to keep the trajectory closer to the task, and implement a gripper state controller that misses grabbing the ball even fewer times. The flexibility of the GMR algorithms would allow for further testing which input variables would lead to more reasonable regression outputs and better trajectory generated. Training the system with joint angles was considered,

and the results were worse, but would need further investigation. For some output variables it is possible that temporal data is not the best basis input, for others it is. For example for the gripper state output, it might be better to base the decision whether the robot should grab or release based on spatial information (whether the robot's hands have reached the target object) than temporal data (has enough time passed to when the robot's had usually reached the target object during the demonstrations). Other methods, including Probabilistic activity grammars, such as the system implemented in [45] could be also investigated.

The most useful variables that capture most of the input information could be selected using Principal Component Analysis, which is a method that projects the variables to the lower n dimensional space that keeps the most variance in the data. [39] Using less dimensions and thus less complex spaces could speed up the GMM training and would lead to faster GMR regression. This could lead to less delays in the implemented controller and a faster reproduction rate, however this improvement is minor. The HTC Vive is fitted with an eye-gaze recorder. The recorded data could be fed into the GMM and thus maybe the experiments would results in an improved novel learning system.

9.4 Object Detection

Another feature that could be implemented is automatic detection of the objects. With this system in place, the robot could be able to read the position of the cup and the ball automatically and based on the learnt GMM model, it could autonomously find the ball, pick it up and place it in the cup. At the moment, before the reproduction algorithm can be started, the positions need to be manually read by moving the robot's gripper over the ball and then the cup and taking readings using the two scripts provided for this. The readings - stored in a file - are read by the reproduction algorithm to calculate the current distance of the robot's grippers from the object and uses that as input to the encoded model. This step could be eliminated by implementing automatic reading of the objects' coordinates.

This step could improve the flair of the project significantly and is considered to be one of the next steps. Readily available packages can be found that store object models and can return the positions of the object to be published on a ROS network.

The implementation of this system was researched, and the the following system is proposed: A readily available object detection node is initialised on the ROS network. This script would be responsible for detecting the cup and the ball and storing the positions in the files automatically instead of manually moving the Baxter over them. Of course calibration of the algorithm needs to be achieved to make sure that the object detection software's coordinate system matches that of the Baxter, but this simple addition is achievable.

Chapter 10

User Guide

This user guide shows how to operate the implemented system for Teleoperation and Learning. First, the code parts written are introduces, and then the necessary steps for running the Teleoperation system and recreating the experimental results are given.

10.1 Code

10.1.1 ROS

A ROS package called *baxter* in the catkin workspace contains all the ROS scripts written for this project.

The following programs are delivered:

- `Head.py`: Reads the head angle data published on the */headposition* topic by Unity and turns the Baxter's head accordingly
- `LeftArm.py`: Reads the left Vive controller's position and orientation data published on the */leftarmposition* topic, calls the IK solver service to get the joint angles and moves the Baxter's left arm.
- `RightArm.py`: Reads the right Vive controller's position and orientation data published on the */rightarmposition* topic, calls the IK solver service to get the joint angles and moves the Baxter's right arm.
- `Grippers.py`: Reads the gripper commands published on the */gripperstates* topic, and opens/closes the robot's grippers.
- `BallPosition.py`: With the Baxter's hand over the ball, it reads the position of the hands and stores it in a file.

- `CupPosition.py`: With the Baxter's hand over the coffee cup, it reads the position of the hands and stores it in a file.
- `DataRecorder.py`: A modification of *joint_recorder.py* in the *baxter_examples* ROS package, used to store the record the demonstrations by storing the robot's states. It takes as input the name of the file to write and optionally the rate of recording (100 Hz is the default). It reads the files created by the `BallPosition.py` and the `CupPosition.py` scripts and stores the values in the file it created alongside various other robot states, including the position and orientation coordinates of both of the robot's hands and the time of the data-snap.
- `Reproduce.py`: Reads the position of the Baxter's hand, and feeds it into the Gaussian Mixture Regression model with a time value to get the next hand position. Then it sets the arms to the new position.

These launch files should be run (with *roslaunch*) in order to test Teleoperation:

- `baxter.launch`: Launches the necessary ROS scripts listed above (`Head.py`, `LeftArm.py`, `RightArm.py`, `Grippers.py`) for Teleoperation from a single command.
- `realsense2_camera.launch`: This launch file starts up the scripts necessary to interface the Intel D435 (or equivalent) Stereo RGBD cameras. The scripts looks for connected RealSense cameras and publish the camera feeds onto the corresponding topics, under the /camera/... topic, including stereo colour image and depth information. The camera needs to be connected to one of the USB ports of the Ubuntu laptop in order for successful image transmission.
- `init_all.launch`: Launches the above two launch files, and makes it easier to start up the entire project.

10.1.2 Unity

A Unity project called *Baxter* is created and the following scripts are attached to the right GameObjects within the scene.

- `CameraSubscriber.cs`: It subscribes to the camera image feed (on topic /camera/-color/image_raw/compressed) to get the video published by the Intel D435 camera on the Baxter's head. Then it renders the images onto a plane defined within the scene, as a texture.

- `CreateVideo.cs`: This script is rendered to the same plane as `CameraSubscriber.cs`. It reshapes, resizes and rotates the plane so that it fills up the Vive's screen in the HMD, so that the user can only see the camera image. This reshapes, resizes the plane that the image is rendered onto, so that the user can only see the the camera image.

- `HeadPosisionPublisher.cs`: This script publishes the horizontal angle of the Vive's HMD onto the /headposition topic.

- `LeftArm.cs`: This script publishes the position and orientation coordinates of the Vive's left controller onto the /leftarmposition topic. (The Grip and the TrackPad button's states respectively determine whether it should publish data and whether the orientation published should be a pre-defined value or the actual orientation of the controller).

- `RightArm.cs`: This script publishes the position and orientation coordinates of the Vive's right controller onto the /rightarmposition topic. (The Grip and the TrackPad button's states respectively determine whether it should publish data and whether the orientation published should be a pre-defined value or the actual orientation of the controller).

- `LeftGripper.cs` This scrips simply publishes a command to the /gripperstates topic when the Trigger button on the left controller is pressed and another command when it is unpressed.

- `RightGripper.cs` This scrips simply publishes a command to the /gripperstates topic when the Trigger button on the right controller is pressed and another command when it is unpressed.

10.1.3 Imitation Learning

The following Matlab and Python scripts were written for various tasks of the Imitation Learning part of the project.

- `LearningPrep.py`: This Python script reads the demonstration data recorded and prepares it for the learning algorithms. The recorded files already contain the initial position of the ball and of the coffee cup. This script first calculates the distance of the Baxter's hand from these object at every time frame. Then it temporally aligns the ten demonstrations using Dynamic Time Warping. Finally, it selects the optimal number of components k for the Gaussian Mixture Model to be trained in the next Matlab script.

It also graphs several results, including the pre-and post-aligned demonstration data, and the graph showing the AIC/BIC scores of the different Gaussian Mixture Models.

- `GMM_GMR`: This Matlab function (adapted from Calinon et al. [19]) trains the GMM with the parameter *k* from the `LearningPrep.py` script using the Expectation Maximisation algorithm with k-means initialisation. Then this function performs the reproduction steps using Gaussian Mixture Regression. It reads the Baxter's current gripper position by establishing a connection to ROS and it calculates the trajectory to be taken, by feeding in the current position and the time indices to the GMR. The returned next positions are sent over to ROS on the /leftarmposition and /righarmposition topics, so that the Teleoperation nodes can be reused for action reproduction. This script can also be instructed by setting `showGraphs := true` to plot the generated GMM and GMR models and tajectories along the way.

- `demo1, demo2, demo3`: These files are identical to the files given by [17], with the input changes to match the project's dataset. This way these Matlab scrips show step by step all of the functionalities of the `GMM_GMR` script.

10.2 Running the Project

This part of the *User Guide* will show the necessary steps to run all of the project, including Teleoperation, data recording and action reproduction.

10.2.1 Teleoperation

First, a couple of setup steps need to be completed in order to be able to launch the correct Unity and Baxter files.

The Vive should be correctly set up an connected to the Windows development machine of the Personal Robotics Lab through the link box provided with the Vive's package. The Baxter should be turned on, and both of the devices should be connected to the same network. The same is true for the Ubuntu development laptop. In order for the devices to communicate over the ROS network, two environmental variables need to be set on the computers:

1. ROS_MASTER_URI needs to be set to: http://011401P0008:11311

2. ROS_HOSTNAME needs to be set to *ViveMagic* on the Windows machine and *Karoly_Lenovo* on the Ubuntu laptop

After the Baxter finished starting up, before running the project in Unity, the following scripts and launch files needs to be run on the Ubuntu laptop.

1. `. baxter.sh`: Script to initialise the Baxter's necessary environmental variables (found under /catkin_ws folder).

2. `rosrun baxter_tools enable_robot.py -e`: enables movement of the robot's arms.

3. `rosrun baxter_tools tuck_arms.py -u`: moves the robot's arms into a comfortable initial position for the Teleoperation to begin.

4. `roslaunch rosbridge_server rosbridge_websocket.launch`: Initialises the ROSBridge server on the Ubuntu laptop which is necessary for ROS# to send and receive messages from the Baxter. It is important to check in the Unity scripts, that the web sockets initiated contain the IP address of the Ubuntu laptop (the server) in order for the connection to work properly, and the clients to find the server.

5. `roslaunch realsense2_camera rs_camera.launch`: This launch file starts up the scripts necessary to interface the Intel D435 (or equivalent) Stereo RGBD cameras. The scripts looks for connected RealSense cameras and publish the camera feeds onto the corresponding topics, under the /camera/... topic, including stereo colour image and depth information. The camera needs to be connected to one of the USB ports of the Ubuntu laptop in order for successful image transmission.

6. `baxter.launch`: Launches the necessary ROS scripts listed above (`Head.py`, `LeftArm.py`, `RightArm.py`, `Grippers.py`) for Teleoperation from a single command. This initiates all of the necessary nodes for teleoperation.

The nodes can be launched and run independently as well, but the easiest way to start the project is by typing the above commands into terminal windows.

Then, the Unity part of the project can be started on the Windows machine. Is is quite simple to get that running, as the Unity project called *Baxter* already has all of the necessary GameObjects defined in a scene and all of the necessary scripts attached to them. Therefore it is sufficient to simply click on the play button to start the project. If the VR headset is connected properly to the computer, the user should already be able to see the Intel camera's image rendered to the HMD, and when moving the headset, the Baxter's neck should also already move the head, but the arm will not follow the controllers' movement yet. For that, the grip buttons will need to be pressed on the Vive's controllers. When both the left and the

right grip button is pressed, the arms will start to move, but with fixed gripper orientation. To turn on sending and setting the orientation coordinates, the two TrackPad buttons should be pressed as well on the Vive controllers. It is important to mention that turning on the orientation coordinates, the teleoperation experience will get worse and slower. The Baxter sets the wrist movements with all of its joints, therefore large movements can be requires for the robot to follow a small change in the orientation of the controllers. Also, the IK solver finds an achievable solution for less position-orientation couples, so the teleoperation might freeze sometimes for a couple of moments.

This concludes running the Teleoperation part. The next Section will describe how to perform Imitation Learning, with using the Teleoperation system or any other means of moving the robot to record demonstrations, for example using Kinesthetics.

10.2.2 Imitation Learning

Recording Demonstrations

The demonstrations can be recorded using the script `DataRecorder.py`. This script simply writes the values for the robot state readings into a CSV (comma separated value) file. The name of the file has to be provided as an input, with optionally a rate of recordings as well. The default value is 100Hz if no other rate input it given. It is important to call this script from the folder catkin_ws/demonstrations to store the demonstration CSV files in this location. This is important as the other learning algorithms read the files stored here. Before the recording is started, the position of the ball and the position of the cup has to be updated with the `BallPosition.py` and `CupPosition.py` scripts. These write the position of the ball and the cup into a file, which is then read by `DataRecorder.py`, which stores the values at every time instance alongside with the other robot state readings. The other readings include the Baxter's hand positions, orientations, gripper states, time instances and joint angles, which are taken at the rate provided in the input. Since the recordings are taken of robot states and are unrelated to the teleoperation commands published by Unity, other methods of moving the robot's hand are provided. It is recommended to use kinesthetic teaching (dragging the Baxter's hand in zero-gravity mode over the action) for more precise demonstrations.

Imitation Learning Algorithms

Once the demonstrations are recorded, the `LearningPrep.py` script needs to be called to prepare the data ready for Gaussian Mixture encoding. This Python script first reads the demonstration values, reformats them, selects the variables to be used and then performs

temporal alignment of the signals using Dynamic Time Warping. After the demonstrations are aligned, the script calculates the distance of the Baxter's grippers from the ball and the cup at all times, using the gripper position readings and the stored initial position of the ball and the cup. The parameter k used for the number of components in the Mixture Model is calculated by checking which k had the lowest AIC score. After concatenating all of the time instances, the array of data is ready to be passed on to the GMM_GMR Matlab script for obtaining the Gaussian Mixture Model encoding.

Then, the Gaussian Mixture Models should be then trained with the GMM_GMR Matlab script. The number of model components needs to be specified manually. This generates the GMMs and the trajectory to be followed by the robot. It reads the robot's hand inputs and controls it through the trajectory online, by reading the current hand position and returning the next.

`Demo1`, `Demo2`, `Demo3` demonstrates all of the steps of data preparation and various different aspects of the GMM encoding.

Action Reproduction

For the the action reproduction to work properly the position of the ball and the cup needs to be stored by running the `CupPosition.py` and the `BallPosition.py` files. The GMM_GMR file reads these values, and online controls the Baxter's hand through the generated trajectory. Since the ROS scripts from the teleoperation part are reused here for the control of the Baxter's arm, before running the Matlab script, it is important to launch the Baxter.launch file to initialise the head and limb controllers of the robot

The reproduction algorithm can be tested as a standalone program as well, using already recorded data and without the need to go through all of the above listed steps. In this case, the following steps are required to get the robot moving.

- All of the components should be connected and the environmental variables should be set as it has been mentioned in Section 10.2.1. The `baxter.sh` script should be run to complete the necessary setup so that the Baxter can run on the network.
- The `baxter.launch` file should be run to initialise the nodes that are used to control the various part of the robot.
- Reset the initial position of the cup and the ball by running the `CupPosition.py` and `BallPosition.py` scripts if necessary.
- Run the `GMM_GMR` Matlab script that loads the previously trained GMM. This generates the desired trajectory by performing the following steps at each time frame:

1. Read the Baxter's current hand positions.
2. Query the GMR controller for the target hand position.
3. Query the GMR controller for the target gripper state.
4. Control the robot to the desired state by publishing the next hand positions and gripper states to the topics to be read by the teleoperator nodes.
5. Increase time instance count (otherwise the hand position would not move). After the demonstration is finished and there are no more time frames, the system unsubscribes, unadvertises and shuts down ROS.

Note that for this part it is not necessary to launch the realsense2_camera launch file, because the image does not need to be rendered inside the headset for the user. In case the teleoperator wishes to watch the action from the robot's point of view through the Vive headset, it can also be launched.

References

[1] Ahrendt, P. (2005). The multivariate gaussian probability distribution. *Technical University of Denmark, Tech. Rep*.

[2] Akaike, H. (1974). A new look at the statistical model identification. In *Selected Papers of Hirotugu Akaike*, pages 215–222. Springer.

[3] Akaike, H. (1998). Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*, pages 199–213. Springer.

[4] Akgun, B., Cakmak, M., Yoo, J. W., and Thomaz, A. L. (2012). Trajectories and keyframes for kinesthetic teaching: A human-robot interaction perspective. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 391–398. ACM.

[5] Akgun, B. and Subramanian, K. (2011). Robot learning from demonstration: kinesthetic teaching vs. teleoperation. *Unpublished manuscript*.

[6] Anon (2019a). ROSWiki: Introduction. http://wiki.ros.org/Introduction. [Online; accessed 5-Nov-2018].

[7] Anon, H. V. (2019b). What are the system requirements? [online].

[8] Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483.

[9] Baldini, F. (2017). Machine learning for humanoid robot programming through virtual reality headsets.

[10] Berndt, D. J. and Clifford, J. (1994). Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA.

[11] Billard, A. and Siegwart, R. (2004). Robot learning from demonstration. *Robotics and Autonomous Systems*, 2(47):65–67.

[12] Bilmes, J. A. et al. (1998). A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. *International Computer Science Institute*, 4(510):126.

[13] Bischoff, M. (2017). ROS is a set of open source software libraries and tools in C for communicating with ROS from .NET applications, in particular Unity3D. https://github.com/siemens/ros-sharp. [Online; accessed 2-Dec-2018].

[14] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[15] Bradley, P. S. and Fayyad, U. M. (1998). Refining initial points for k-means clustering. In *ICML*, volume 98, pages 91–99. Citeseer.

[16] Calinon, S. (2009a). Gaussian Mixture Model (GMM) - Gaussian Mixture Regression (GMR). https://uk.mathworks.com/matlabcentral/fileexchange/19630-gaussian-mixture-model-gmm-gaussian-mixture-regression-gmr. [Online; accessed 8-Mar-2019].

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

[18] Calinon, S., D'halluin, F., Sauser, E. L., Caldwell, D. G., and Billard, A. G. (2010). Learning and reproduction of gestures by imitation. *IEEE Robotics & Automation Magazine*, 17(2):44–54.

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

[20] Cederborg, T., Li, M., Baranes, A., and Oudeyer, P.-Y. (2010). Incremental local online gaussian mixture regression for imitation learning of multiple tasks. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 267–274. IEEE.

[21] Chernova, S. and Veloso, M. (2007). Confidence-based policy learning from demonstration using gaussian mixture models. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 233. ACM.

[22] Christian (2016). Visualizing the bivariate Gaussian distribution. https://scipython.com/blog/visualizing-the-bivariate-gaussian-distribution/. [Online; accessed 6-May-2019].

[23] Clarke, T. (2019). Final year individual projects: Student guide. [online].

[24] Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1996). Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145.

[25] Conway, J. H. and Smith, D. A. (2003). *On quaternions and octonions*. AK Peters/CRC Press.

[26] Craighead, J., Burke, J., and Murphy, R. (2008). Using the unity game engine to develop sarge: a case study. In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*.

[27] Demiris, Y. (2018). Intro to ROS: EE4-60 Human-Centred Robotics. https://bb.imperial.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_1480674_1&course_id=_13674_1. [Online; accessed 7-Nov-2018].

[28] Everitt, B. S. (2014). Finite mixture distributions. *Wiley StatsRef: Statistics Reference Online*.

[29] Field, M., Stirling, D., Pan, Z., and Naghdy, F. (2015). Learning trajectories for robot programing by demonstration using a coordinated mixture of factor analyzers. *IEEE transactions on cybernetics*, 46(3):706–717.

[30] Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., and Schmidhuber, J. (2008). A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868.

[31] Grosse, R. and Srivastava, N. (2018). Lecture 16: Mixture models. http://www.cs.toronto.edu/~rgrosse/csc321/mixture_models.pdf. [Online; accessed 17-Jan-2019].

[32] Guerin, K. and Hager, G. D. (2017). Robot control, training and collaboration in an immersive virtual reality environment. US Patent 9,643,314.

[33] Hettinger, L. J. and Riccio, G. E. (1992). Visually induced motion sickness in virtual environments. *Presence: Teleoperators & Virtual Environments*, 1(3):306–310.

[34] Hovland, G. E., Sikka, P., and McCarragher, B. J. (1996). Skill acquisition from human demonstration using a hidden markov model. In *Proceedings of IEEE international conference on robotics and automation*, volume 3, pages 2706–2711. Ieee.

[35] HTC (2016). VIVE SPECIFICATIONS. https://www.vive.com/uk/product/#vive-spec. [Online; accessed 5-Nov-2018].

[36] Hui, J. (2018). RL - Imitation Learning. https://www.vive.com/uk/support/vive/category_howto/what-are-the-system-requirements.html. [Online; accessed 5-Mar-2019].

[37] II, R. L. W. (2017). Baxter Humanoid Robot Kinematics. https://www.ohio.edu/mechanical-faculty/williams/html/pdf/BaxterKinematics.pdf. [Online; accessed 20-Dec-2019].

[38] Janakiev, N. (2018). Understanding the Covariance Matrix. https://datascienceplus.com/understanding-the-covariance-matrix/. [Online; accessed 19-Feb-2019].

[39] Jolliffe, I. (2011). *Principal component analysis*. Springer.

[40] Ju, Z., Yang, C., Li, Z., Cheng, L., and Ma, H. (2014a). Teleoperation of humanoid baxter robot using haptic feedback. In *2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI)*, pages 1–6. IEEE.

[41] Ju, Z., Yang, C., and Ma, H. (2014b). Kinematics modeling and experimental verification of baxter robot. In *Proceedings of the 33rd Chinese Control Conference*, pages 8518–8523. IEEE.

[42] Kim, S., Kim, Y., Ha, J., and Jo, S. (2018). Mapping system with virtual reality for mobile robot teleoperation. In *2018 18th International Conference on Control, Automation and Systems (ICCAS)*, pages 1541–1541. IEEE.

[43] Kim, S.-K., Hong, S., and Kim, D. (2009). A walking motion imitation framework of a humanoid robot by human walking recognition from imu motion data. In *2009 9th IEEE-RAS International Conference on Humanoid Robots*, pages 343–348. IEEE.

[44] Klingspor, V., Demiris, J., and Kaiser, M. (1997). Human-robot communication and machine learning. *Applied Artificial Intelligence*, 11(7):719–746.

[45] Lee, K., Su, Y., Kim, T.-K., and Demiris, Y. (2013). A syntactic approach to robot imitation learning using probabilistic activity grammars. *Robotics and Autonomous Systems*, 61(12):1323–1334.

[46] Li, C., Yang, C., Wan, J., Annamalai, A. S., and Cangelosi, A. (2017). Teleoperation control of baxter robot using kalman filter-based sensor fusion. *Systems Science & Control Engineering*, 5(1):156–167.

[47] Li, X. and Wu, X. (2015). Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4520–4524. IEEE.

[48] Lipton, J. I., Fay, A. J., and Rus, D. (2017). Baxter's homunculus: Virtual reality spaces for teleoperation in manufacturing. *IEEE Robotics and Automation Letters*, 3(1):179–186.

[49] Lu, L. and Wen, J. T. (2015). Human-robot cooperative control for mobility impaired individuals. In *2015 American Control Conference (ACC)*, pages 447–452. IEEE.

[50] McCarthy, J. M. and McCarthy, J. M. (1990). *An introduction to theoretical kinematics*, volume 2442. MIT press Cambridge.

[51] Nakanishi, J., Morimoto, J., Endo, G., Cheng, G., Schaal, S., and Kawato, M. (2004). Learning from demonstration and adaptation of biped locomotion. *Robotics and autonomous systems*, 47(2-3):79–91.

[52] Ng, A. (2014). Machine learning - introduction. [online].

[53] Ng, A. (2018). Mixtures of Gaussians and the EM algorithm. http://cs229.stanford.edu/notes/cs229-notes7b.pdf. [Online; accessed 6-April-2019].

[54] Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX*, pages 363–372. Springer.

[55] Pacula, M. (2011). k-means clustering example (Python). http://blog.mpacula.com/2011/04/27/k-means-clustering-example-python/. [Online; accessed 7-May-2019].

[56] Parmiggiani, A., Maggiali, M., Natale, L., Nori, F., Schmitz, A., Tsagarakis, N., Victor, J. S., Becchi, F., Sandini, G., and Metta, G. (2012). The design of the icub humanoid robot. *International journal of humanoid robotics*, 9(04):1250027.

[57] Paul, R. P. (1981). *Robot manipulators: mathematics, programming, and control: the computer control of robot manipulators*. Richard Paul.

[58] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830.

[59] Peng, G., Yang, C., Jiang, Y., Cheng, L., and Liang, P. (2016). Teleoperation control of baxter robot based on human motion capture. In *2016 IEEE International Conference on Information and Automation (ICIA)*, pages 1026–1031. IEEE.

[60] Peppoloni, L., Brizzi, F., Ruffaldi, E., and Avizzano, C. A. (2015). 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*, pages 237–240. ACM.

[61] Piech, C. (2013). K Means. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html. [Online; accessed 7-May-2019].

[62] PyPI (2017). fastdtw 0.3.2. https://pypi.org/project/fastdtw/. [Online; accessed 17-Mar-2019].

[63] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

[64] Rahmatizadeh, R., Abolghasemi, P., Behal, A., and Bölöni, L. (2018). From virtual demonstration to real-world manipulation using lstm and mdn. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[65] Ramage, D. (2007). Hidden markov models fundamentals. *CS229 Section Notes*, 1.

[66] Rasmussen, C. E. (2003). Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer.

[67] Reddivari, H., Yang, C., Ju, Z., Liang, P., Li, Z., and Xu, B. (2014). Teleoperation control of baxter robot using body motion tracking. In *2014 International conference on multisensor fusion and information integration for intelligent systems (MFI)*, pages 1–6. IEEE.

[68] Ripton, J. and Prasuethsut, L. (2015). The vr race: who's closest to making vr a reality. *. . . -tech/the-vr-race-who-s-closest-to-. . . .*

[69] Robotics, R. (2018a). Baxter – Redefining Robotics and Manufacturing. http://sdk.rethinkrobotics.com/wiki/Home. [Online; accessed 10-Nov-2018].

[70] Robotics, R. (2018b). Rethink Robotics: Baxter Overview. http://sdk.rethinkrobotics.com/wiki/Baxter_Overview. [Online; accessed 2-Dec-2018].

[71] Robotics, R. (2018c). Rethink Robotics: Inverse Kinematics Solver Service. http://sdk.rethinkrobotics.com/wiki/API_Reference#Inverse_Kinematics_Solver_Service. [Online; accessed 10-Dec-2018].

[72] Sak, H., Senior, A., and Beaufays, F. (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*.

[73] Samuel, A. L. (2000). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 44(1.2):206–226.

[74] Schwarz, G. et al. (1978). Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464.

[75] Siciliano, B. and Khatib, O. (2016). *Springer handbook of robotics*. Springer.

[76] Sidner, C. L., Lee, C., Kidd, C. D., Lesh, N., and Rich, C. (2005). Explorations in engagement for humans and robots. *Artificial Intelligence*, 166(1-2):140–164.

[77] Smart, W. D. (2002). *Making reinforcement learning work on real robots*. Brown University Providence, USA.

[78] Sung, H. G. (2004). *Gaussian mixture regression and classification*. PhD thesis, Rice University.

[79] Sweeney, J. D. and Grupen, R. (2007). A model of shared grasp affordances from demonstration. In *2007 7th IEEE-RAS International Conference on Humanoid Robots*, pages 27–35. IEEE.

[80] Vakanski, A., Mantegh, I., Irish, A., and Janabi-Sharifi, F. (2012). Trajectory learning for robot programming by demonstration using hidden markov model and dynamic time warping. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(4):1039–1052.

[81] Vernon, D., Metta, G., and Sandini, G. (2007). The icub cognitive architecture: Interactive development in a humanoid robot. In *2007 IEEE 6th International Conference on Development and Learning*, pages 122–127. Ieee.

[82] Yang, J., Xu, Y., and Chen, C. S. (1997). Human action learning via hidden markov model. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 27(1):34–44.

Appendix A

Evaluation Plan

It is difficult to define mathematical measures for the teleoperation part of the project. However, with human judgement, some considerable points:

1. Is there lag between the video displayed and the robot's actions?
2. Do the robot's arms move accurately and as instructed by the user?
3. Can the grippers be controlled by the controller easily and accurately?

Visually, the he machine learning part be judged by:

1. Whether the robot is able to record the actions and demonstration in a compact and usable way
2. Whether the proposed solution can reproduce the action
3. Whether enough different Learning by Demonstration methods were experimented with.

It is harder to judge whether or not the selected model generalises well. The main question asked here, whether the robot can learn an action that is further generalizable and how well can the robot generalize this action in comparison to the non-generalized method and in comparison to other research presented in the background search that was done in the field of Imitation Learning.

Appendix B

Ethical, Legal, and Safety Plan

B.1 Ethical Considerations

"Being ‘ethical’ means acting in accordance with a set of core values and principles, in particular integrity, compliance with the law, respect for human rights and avoiding unnecessary risk to people’s safety and well-being. Imperial College London seeks to ensure that any potential ethical risks arising from research are limited strictly in proportion to the importance of the intended benefits. A researcher must therefore consider the ethical implications of any work that:

- has the potential to damage the mental or physical health of human participants, (e.g. volunteers, College staff and students, or patients,) or others who may be affected has the potential to jeopardise the safety and liberty of people affected by the research (e.g. volunteers working in sensitive situations abroad)
- has the potential to compromise the privacy of individuals whose data is involved in the work
- otherwise involves methods (e.g. genetic research) or subject matter (e.g. recreational and controlled drugs) that are sensitive and therefore need to be managed consistently
- with the College’s high public reputation
- carries a risk of an actual or perceived conflict of interest on the part of researchers and/or the College
- has potential for environmental impact"

Accessed from Imperial College Research Ethics

B.2 Ethical Concerns

According to Imperial College's Research Ethics, the project does not raise an issue from the ethics point of view. See Appendix B.1 for the Ethics guideline. Even though technology can be always misused, ethical obstacles are unlikely to be faced for this project.

B.3 Legal Considerations

Similarly to the ethical considerations, in theory technology could be always misused, however in this project no legal obstacles will be faced. The robot is not going to be taught anything that would go against law.

B.4 Safety Assessment

The following points summarize what could possibly go wrong during the experiments and hurt other members of the lab, visitors, me or the lab's equipment and what are the precautions taken to prevent such accidents.

Crushing and trapping kind of accidents: The robot could injure someone by trapping and squeezing a body part. Impact or Collision accidents: The robot could also hit and cause damage resulting from quick motion. The possible types of injuries can include bruises, cuts, pierces skins or more seriously broken body parts or blinded eyes. To avoid these kinds of accidents, the robot will be paces into a fenced-off area where unauthorised people, who aren't working on the project will not be allowed to enter. The robot will be only operated when nobody is around. The robot will be places at a clearly visible place, where it is impossible to enter accidentally.

- Trip hazards: Since the project requires wired connection between different hardware components, it is necessary to always make sure that nothing is left on the ground and no wires pose tripping hazard.
- Flying objects: For safety reasons, the robot will not throw anything.
- Electrical safety: All of the electrical equipment used is tested by Imperial College for electrical fire and shock safety.
- Robot moving around: Optionally, a movable pedestal can be installed to the robot to move it around, manually. Moving the robot around is dangerous, therefore once in place, it will not have to change its position.

- The Baxter is a collaborative robot, and as such it has built-in safety features. It has sensors built in its hands and around the arms to adapt to this surroundings. Therefore it had the ability to sense and avoid collisions and reduce the impact. Precise hand sensors and cameras on the Baxter's hand allow it to be particularly sensitive while working with humans. The squeezing force of the arms and grippers is limited.

- Since it is impossible to sense the outside world when wearing the VR headset, the utilisation is dangerous. The objects from the playing area need to be removed, otherwise it is possible to collide when the headset is on. The VR headset will only be operated in a clear area following HTC's setup and usage guidelines, even though this project does not include walking around with the Vive.

- Also, the headset has a camera mounted on it that can be used for safety. It detects if there are physical objects in the the playing area and displays virtual objects to block the way of the operator. The system can be configured to either display the object in the virtual environment, or to show a warning message.

- Motion sickness, or as nowadays called VR sickness is a usual side effect of inexperienced individuals using Virtual Reality for the first time. Official guidelines say that usage should be increased gradually and as soon as nausea is felt, the game/application should be paused and the user should take the headset off.

- No further Electrical, Physical, Fire, Chemical, Biological Animal, Appliance, Airspace Study Participant or Data Infrastructure safety is posed by this project.