Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Simon Julian Nagelsaker Lexau

Masteroppgave

Combining Reinforcement Learning and Robotic Vision for Human-Machine Cooperation

Masteroppgave i Kybernetikk og Robotikk
Veileder: Anastasios Lekkas
Juni 2022

NTNU
Norwegian University of Science and Technology

Simon Julian Nagelsaker Lexau

Masteroppgave

Combining Reinforcement Learning and Robotic Vision for Human-Machine Cooperation

Masteroppgave i Kybernetikk og Robotikk
Veileder: Anastasios Lekkas
Juni 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Preface

This master's thesis is submitted as a part of the requirements for the master's degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The work presented in this thesis has been carried out under the supervision of Associate Professor Anastasios Lekkas, NTNU.

This master's thesis is a continuation of a specialization project I conducted during the autumn of 2021. As is customary, the specialization project is not published. This means that important background theory and methods from the project report will be restated in full throughout this report to provide the best reading experience. Below, a complete list of the material included from the specialization project is listed.

- Chapter 2, sections 2.1.1, 2.3.2, 2.5
- Chapter 3, sections 3.2, 3.3
- Chapter 4, section 4.3

During the project, I have been provided multiple tools through Omega Verksted who also gave me access to their workshop and 3D printers. The *Reinforcement Learning Environments* package was developed by Ph.D. candidate Sindre B. Remman, Department of Engineering Cybernetics, NTNU, and was an invaluable starting point for training of the reinforcement

learning agent. During the master´s project, I have further extended this training platform to be compatible with the MuJoCo simulator. Anastasios Lekkas, NTNU, provided me with the *OpenManipulator-X*, two stereoscopic depth sensors of the type *Intel RealSense D435*, a lever, and a *Raspberry Pi 3 Model B 1GB RAM*. Parts were designed and 3D-printed as part of this work to facilitate robotic vision in the physical setup. The Human-Machine Cooperation Interface is based on the Human-Machine Interface from the project thesis and was developed on a desktop computer supplied by NTNU, with an *Intel i7-8700 CPU*, *32 GB RAM*, and running *64-bit Ubuntu 20.04*. Open-source Python, and ROS, packages were used to build the system, and a comprehensive list is given in Table 3.1

Unless otherwise stated, all figures and illustrations have been created by the author.

I would like to thank my fellow graduates for the many lunches, quizzes, Sudoku riddles, and the support they have provided, my supervisor for setting me on the right track, Sindre Remman for answering all of my technical questions, and my family for the encouraging words towards the end of my last semester as an M.Sc student at NTNU.

Simon Lexau

Simon Julian Nagelsaker Lexau
Trondheim, June 2022

Abstract

Machines have affected our lives for millennia, arguably since the Romans first constructed water turbines for powering mechanical devices used in the production of flour on industrial scales. Advancements in science and technology keep bringing new appliances into our society with promises to alleviate humans from tedious, repetitive, and dangerous jobs. With the combined increase in computational power and research within artificial intelligence, the possibility of automating even more advanced and dynamic jobs have emerged. Tesla and other car manufacturers are developing self-driving cars. Simultaneously, a global race is ongoing between companies and governments to create the first autonomous ships. These advances do not come without problems, however. It is often difficult for humans to understand the "thought process" of artificial intelligence, which leads to a problem of trust and responsibility. This thesis focuses on solving these problems through active cooperation between artificial intelligence and humans. The solution presented includes a hand-tracking powered Human-Machine Interface, structured as a finite state machine, which an operator uses to collaborate with a reinforcement learning agent. The system is tested in a cooperation challenge, where a robotic manipulator is used to pull a lever in a specific angular position while relying on visually estimated angle values. By assigning full responsibility to the human overseer who plays an active role in the loop, one avoids designing a bulletproof system that also deals with risk assessment.

Sammendrag

Maskiner har påvirket livene våre i årtusener, helt siden romerne først konstruerte vannturbiner for å drive mekaniske enheter brukt i produksjon av mel i industriell skala. Fremskritt innen vitenskap og teknologi bringer stadig nye apparater inn i samfunnet vårt med løfter om å frigjøre mennesker fra kjedelige, repeterende og farlige jobber. Med den kombinerte økningen i beregningskraft og forskning innen kunstig intelligens, har muligheten for å automatisere enda mer avanserte og dynamiske jobber dukket opp. Tesla og andre bilprodusenter utvikler selvkjørende biler. Samtidig pågår et globalt kappløp mellom selskaper og myndigheter for å skape de første autonome skipene. Disse fremskrittene kommer imidlertid ikke uten problemer. Det er ofte vanskelig for mennesker å forstå «tankeprosessen» til kunstig intelligens, som fører til et problem innen tillit og ansvar. Denne oppgaven fokuserer på å løse disse problemene gjennom aktivt samarbeid mellom kunstig intelligens og mennesker. Løsningen som presenteres inkluderer et håndsporingsdrevet menneske-maskin grensesnitt, strukturert som en endelig tilstandsmaskin, som en operatør bruker til å samarbeide med en forsterkningslæringsagent. Systemet er testet i en samarbeidsutfordring, der en robotisk manipulator brukes til å trekke en spak i en spesifikk vinkelposisjon mens vinkelen estimeres visuelt. Ved å legge hele ansvaret til den menneskelige tilsynsmannen som spiller en aktiv rolle i loopen, unngår man å designe et skuddsikkert system som også omhandler risikovurdering.

Contents

List of Tables

List of Figures

Acronyms

ADC	Analog to Digital Converter. xiii, 61, 62, 64
AI	Artificial Intelligence. 2, 4, 7, 73, 75, 120
ANN	Artificial Neural Network. 27
AUV	Autonomous Underwater Vehicle. 134
BFGS	Broyden, Fletcher, Goldfarb and Shanno. 17
CNN	Convolutional Neural Network. 29, 77
DDPG	Deep Deterministic Policy Gradient. 29, 30, 32, 59
DH	Denavit-Hartenberg. 15, 50, 67, 84
DLS	Damped Least Squares. 16
DOF	Degrees of Freedom. 12, 16

FABRIK Forward And Backward Reaching Inverse Kinematics. 17, 18

FK Forward Kinematics. x, 13, 50, 51

FOV Field Of View. 52

FSM Finite State Machine. x–xii, 21, 73, 75, 79, 81, 84, 88, 103, 104, 112, 113, 116

GUI Graphical User Interface. 89

HER Hindsight Experience Replay. 35, 125

HMC Human-Machine Cooperation. 2, 4, 5, 72

HMCI Human Machine Cooperation Interface. x, 5, 45, 48, 49, 51, 73, 95, 106, 109, 132, 134

HMI Human Machine Interface. 2, 5, 6, 8, 71, 76, 95, 96, 98, 99, 112, 132

IK Inverse Kinematics. 16, 17, 19, 50

IR InfraRed. 52

ML Machine Learning. 81

MPC Model Predictive Control. 126, 133

MSBE Mean-Squared Bellman Error. 31, 32

OM-X OpenManipulator-X. 5, 75

OS Operating System. 95

PnP Perspective-n-Point. xii, 11, 37–39, 60, 65, 128

RGB Red Green Blue. xiii, 53, 60, 66, 73, 131

RL Reinforcement Learning. x, 2, 5–8, 17, 25, 29, 45, 49, 53, 54, 72, 75, 92, 94–96, 99, 103, 106, 108–110, 112, 113, 118, 120, 127, 129, 132–134

ROS Robot Operating System. 5, 6, 17, 42, 48, 50, 53, 75, 77, 78, 86, 92, 94–99, 101

SDK Software Development Kit. 53

SDLS Selectively Damped Least Squares. 16

SPI Serial Peripheral Interface. 62, 64

STL Standard Triangle Language. 50

SVD-DLS Damped Least Squares with Singular Value Decomposition. 16

UDP User Datagram Protocol. 65

URDF Unified Robot Description Format. 101

Chapter 1

Introduction

1.1 Background and Motivation

The first machines that significantly improved our lives were driven by water or wind and ground our grain [1]. Later machines revolutionized large-scale production by reducing the human workload on assembly lines and simultaneously increasing productivity [2]. Today, machines and robots are becoming increasingly capable of solving more complex problems. The Tesla car factories are great examples of how robots are being used in modern assembly lines [3].

Human-Machine Interaction is important today and will become even more important in the future as more and more jobs begin utilizing the advancements within robotics. These interactions happen mainly through Human-Machine Interfaces (HMIs) which include traditional interfaces such as keyboard, mouse and buttons, to more unconventional interfaces like speech control [4], gesture control [5] [6], and brain interfaces [7]. These newer systems allow for more intuitive interactions, which is advantageous as the human operator interacts with increasingly capable and intelligent robots.

The last decade has seen a boom in innovation related to artificial intelligence, with [8] revealing a tremendous increase in artificial intelligence (AI) related patents since 2013. A great deal has been done on implementing reinforcement learning, a branch of AI, in simulations, but the same cannot be said for physical robotic manipulators. In [9], Deep Reinforcement Learning for vision-based robotic manipulation is utilized to grasp unknown objects with a 96% success rate. Yet robots can still only perform simple manipulation tasks given enough samples to learn [10], and the demonstrations of model-based reinforcement learning (RL) techniques in real applications remain of limited practical use [11]. Another challenge with AI in robotics is the task of gathering enough experience from which the agents can learn. [12] suggests overcoming this obstacle by training on multiple robotic manipulators in parallel. In contrast, [13] delves into the sim-to-real transfer, which is the approach taken in this thesis.

In [14] the term Human-Machine Cooperation (HMC) is introduced and debates the necessity of humans and autonomous machines to cooperate more closely to combat unforeseen challenges. Concerns about the loss of expertise due to automation, the dangers of over-reliance, and issues regarding trust in the system, and self-confidence of the human operators were discussed. In a case where pilots were first confronted with an automatic system for landing, they either used the system blindly or shortcutted it. Advancements have been made since the paper was written in 2000, and the need for cooperation between humans and machines has not decreased. Self-driving cars and autonomous ships are some of the groundbreaking innovations around the corner, yet these technologies often have different levels of automation, which demands different levels of cooperation. A total of seven (0 - 6) levels of autonomy for ships is presented by Lloyd's Register [15]. Even though the work presented in this thesis does not fathom marine vessels, it can be classified as a level 4: "Human on the loop - operator/supervisory." See Appendix A.3 for all levels of autonomy.

When developing an autonomous system, it is crucial to evaluate the ethical concerns. In 2017 teachers in Houston had their performance assessed by an AI [16]. The agent made predictions based on their students' test results compared to the average score in Texas. The teachers who received a good performance were awarded bonuses, while those with low scores risked getting fired. The company responsible for the AI refused to reveal how it made predictions, calling it a trade secret. Thus the teachers could not tell if the predictions were fair or faulty. Later, a federal judge ruled that the AI program could be violating their civil rights. The school district stopped using it and paid the teachers' fees.

Some critical aspects of AI technologies employed in the real world are accountability, explainability, robustness, and safety [17]. The points of accountability and safety are most relevant to this thesis. Accountability is *the fact of being responsible for your decisions or actions and expected to explain them when you are asked*, as defined in [18]. Explainability is linked to the definition of accountability and describes to which degree the AI can explain its decisions. Robustness is the *the quality of being strong and unlikely to break or fail* [19]. The safety aspect fathoms not causing damage to any equipment and, most importantly, not harming humans and animals that may be affected by the decisions of an AI algorithm.

Not much work has been done on cooperative control of robotic manipulators to solve complex real-world tasks. Relevant work includes [20], where robotic manipulators are used during surgery. The system can seamlessly execute automated tasks before giving control back to the surgeon. Other examples of HMC (Human-Machine Cooperation), such as [21] describe situations with humans inside the manipulator's workspace and the facilitation of safe work environments. Thus, this thesis will explore the notion of Human-Machine Cooperation in robotics, tested on a practical lever-pull challenge.

1.2 Objectives and Research Questions

The scope of the thesis is defined in this section through three research questions and a list of objectives. These questions will be revisited in Chapter 6.

- Can a human operator cooperate with an RL agent through an HMCI to solve a manipulation task with a robotic manipulator?
- How transferable is an RL agent trained in a simulated environment to a physical one?
- Can a noisy visually estimate of the lever pose replace reliable direct measurements as input signals for an RL agent?

Note that HMCI is an acronym introduced in this thesis, and is a combination of the acronyms HMI and HMC, it is short for Human-Machine Cooperation Interface. To answer the research questions, the objectives listed below are outlined.

1. Convert the *OpenManipulator RL Environments* developed by the author of [22] to ROS Noetic and Python 3. The package was initially built as a framwork to train RL agents on the OpenManipulator-X.
2. Create a gym environment for the OM-X in MuJoCo, which was re-released as open-source software in 2021.

3. Port the HMI of [23] into a ROS Noetic package and extend it to be compatible with an RL agent. The HMI allows operators to control the OpenManipulator-X with hand gestures through an intuitive interface.

4. Design a method for continuous measurements of the physical lever angle.

5. Design an algorithm for estimating the lever angle and position from camera images.

6. Train an RL agent in the MuJoCo environment, and transfer it to the physical system.

1.3 Contributions

The thesis contributes to the fields of robotic control systems, reinforcement learning, robotic vision, Human-Machine Interface and Human-Machine Cooperation. More specifically, the thesis introduces:

- A novel, intuitive, and responsive Human-Machine Cooperation Interface for improved Human-Robot Interaction
- A vision-based estimation technique for lever poses
- A reinforcement learning environment for the OpenManipulator-X in the MuJoCo simulator

The Human-Machine Cooperation Interface is based on the Human-Machine Interface developed by the author in the project thesis [23], but the extended and more advanced version presented in this thesis allows for direct and intuitive cooperation between a reinforcement learning agent and a human operator.

The vision-based estimation technique is designed to work as the RL agent's eyes and extracts meaningful information about the lever in the manipulator's workspace. The lever is a part of the cooperation challenge, meant to illustrate how a human can cooperate with an AI to accomplish complex goals; it is presented in Section 5.2.

The OpenManipulator-X had to be ported into the MuJoCo simulator as no official or unofficial model known to the author exists. The reinforcement learning environment is based on the work done in [22], but extended and altered to suit the research questions of this thesis.

1.4 Outline

The thesis is partitioned into six chapters. Chapter 1 is the introduction, and Chapter 2 introduces the most relevant theory, which is useful for understanding the later chapters. Chapter 3 describes the physical setup, as well as the simulation environment for the training of the RL agent. Two methods of measuring the pose of physical lever are also presented here. In Chapter 4, the problem is formulated and a physical test is introduced. A summary of the system is given in the System Overview section. Following, the HMI from [23] and the *OpenManipulator RL Environments* package from [22] is presented. Further, Chapter 5 presents and discusses the results from the cooperation challenge.

Figure 1.1 outlines the thesis as a high abstraction block diagram, displaying the interactions between the most important parts of the system, and their relevant sections.

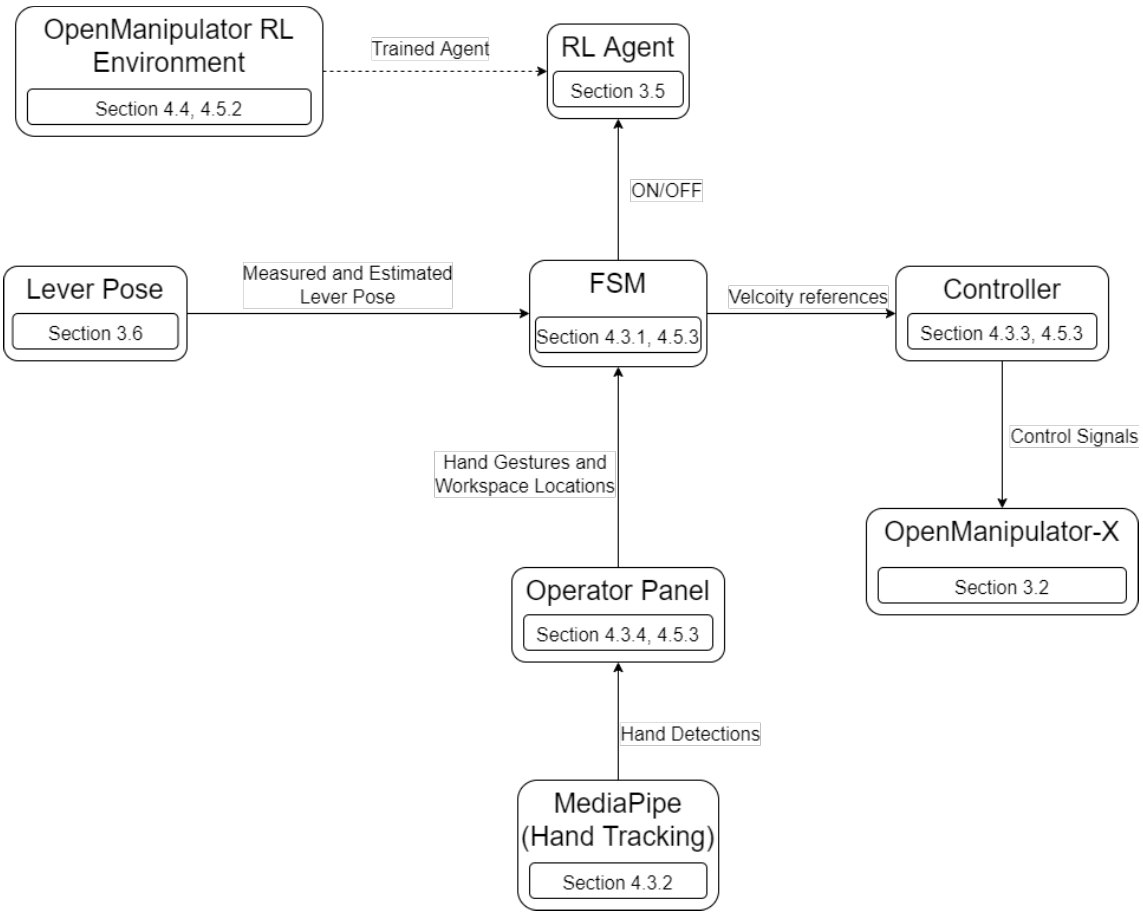


Figure 1.1: A high absraction map of the thesis, with relevant sections

Chapter 2

Theory

This chapter presents theory which is relevant for the solution introduced in the next chapters. Firstly, the forward- and inverse kinematics problems needed to control the robotic manipulator are reviewed. Next, the concept of Finite State Machines are described. Further, the most important parts of Reinforcement Learning is presented, together with the DDPG algorithm used to train the agent in this thesis. The next section introduces the PnP algorithm, which is necessary during the visual estimation of the lever pose. The final section describes the most relevant parts of the Robot Operating System, which is used for communications and package structuring.

2.1 Robotic Manipulators

Material from this section has been reused from the project thesis [23] and modified to better fit this thesis.

Robotic manipulators are mechanical devices designed to perform a wide range of automated tasks. Most manipulators consist of a series of rigid links connected by joints with one adjustable parameter, commonly referred to as one DOF (Degrees of Freedom). DOF are explained in [24] as *the number of scalar variables that are necessary and sufficient to describe the locations of all the components in a mechanical system.* Thus, a robotic manipulator with 4 movable joints will have four DOF. In some cases, the manipulator might be restricted, however. For instance, a pendulum attached to a rigid body can only move in a spherical space and has only two DOF instead of three. Its DOF is the difference between its DOF in an unrestricted system and the number of constraints. In this case, the pendulum has originally three DOF, but loses one since the body it is attached to is rigid. Furthermore, typical robotic manipulators have six DOF, which means the end-effector of the arm has free translational and rotational movement in the three axes X, Y, and Z.

There exist many different types of joints, but the two most common are prismatic and revolute joints.

Prismatic joint
Often also labeled linear joints, the movements from these joints are translational along a single axis, where the axes of the connected links remain parallel.

Revolute joint
The relative movement between the connected links is rotational, perpendicular, or parallel to the input link.

2.1.1 Forward Kinematics

Forward kinematics (FK) is the process of computing the end-effector's position by using the geometry of the robotic manipulator together with the varying link orientations. These varying angles are commonly known as generalized coordinates.

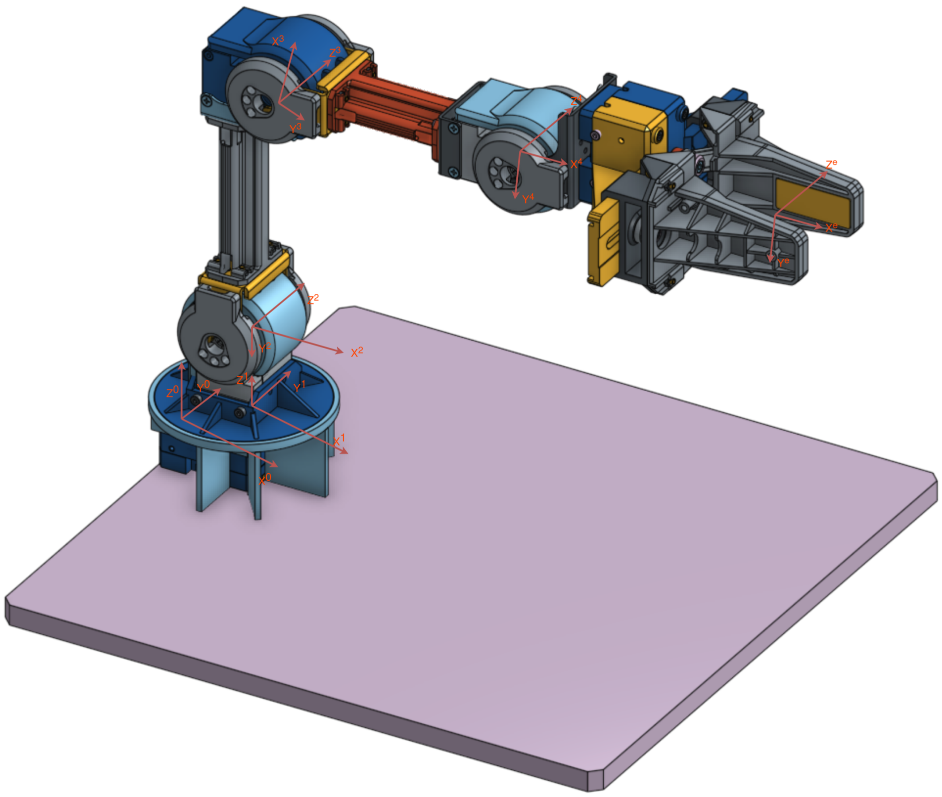


Figure 2.1: A model of the OpenManipulator-X with coordinate systems. Borrowed from [25] and modified.

In Figure 2.1 a total of six coordinate frames represented by vectors of the type $\vec{X}_i, \vec{Y}_i, \vec{Z}_i$ has been drawn. The world frame is represented by the $i = 0$ frame, while the $i = 1$ frame has been drawn in the first joint and is static within the world frame. The subsequent three frames move in relation to each other and are drawn to represent the movement from all joints in a straightforward manner. Lastly, the sixth frame has been drawn suitably on the end-effector. The frames have all been assigned according to the Denavit-Hartenberg (DH) convention [26].

$$\boldsymbol{H}_{\boldsymbol{EF}}^{\boldsymbol{W}} = \boldsymbol{H}_{\boldsymbol{n}}^{\boldsymbol{0}} = \begin{bmatrix} \boldsymbol{R}_{\boldsymbol{n}}^{\boldsymbol{0}} & \boldsymbol{o}_{\boldsymbol{n}}^{\boldsymbol{0}} \\ \boldsymbol{0} & 1 \end{bmatrix} \in SE(3) \tag{2.1}$$

By using the representation depicted in Figure 2.1, we can introduce homogeneous transformations between the frames. These transformation matrices can be multiplied to produce a single transform from zero frame to end-effector frame, as presented in Equation (2.1), where $\boldsymbol{R}_{\boldsymbol{n}}^{\boldsymbol{0}}$ is the rotation from 0-frame to end-effector frame, $\boldsymbol{o}_{\boldsymbol{n}}^{\boldsymbol{0}}$ is the position of the end-effector in 0-frame, and n is the number of movable joints. Thus we can acquire the end-effector coordinates in the static zero frame if we know the DH-parameters. In short, forward kinematics involves calculating the $\boldsymbol{H}_{\boldsymbol{n}}^{\boldsymbol{0}}$ matrix which transforms coordinates from 0 frame to n frame. The $\boldsymbol{H}_{\boldsymbol{EF}}^{\boldsymbol{W}}$ notation will be used in this thesis instead of 0- and n-frames. The reverse transformation can be found by inverting the matrix, see Equation (2.2).

$$\boldsymbol{H}_{\boldsymbol{W}}^{\boldsymbol{EF}} = \left(\boldsymbol{H}_{\boldsymbol{EF}}^{\boldsymbol{W}}\right)^{-1} \tag{2.2}$$

2.1.2 Inverse Kinematics

While forward kinematics can be used to calculate the end-effector position, given joint and link parameters, IK (Inverse Kinematics) is the opposite operation: estimating the variable joint parameters needed to move the end-effector to a pre-determined position. The inverse kinematics problem is usually not as straightforward as the forward kinematics.

$$\boldsymbol{x}_{\boldsymbol{EF}}^{\boldsymbol{W}} = f(\boldsymbol{\theta}) \tag{2.3}$$

$$\boldsymbol{\theta} = f^{-1}(\boldsymbol{x}_{\boldsymbol{EF}}^{\boldsymbol{W}}) \tag{2.4}$$

Analytically, the solution is to invert the direct kinematics equation, given by Equation (2.3) where x_{EF}^{W} is the end-effector position in the world frame and f is a function of rotations and translations dependant on the joint values $\boldsymbol{\theta}$, and calculate all possible solution branches, see Equation (2.4). However, this only works if the number of constraints is the same as the number of DOF of the robot. A range of methods that overcomes the challenge exists, where the most popular class utilizes the Jacobian matrix to find a linear approximation to the inverse kinematics problem. A few of these algorithms include DLS (Damped Least Squares) [27], SVD-DLS (Damped Least Squares with Singular Value Decomposition) [28], and SDLS (Selectively Damped Least Squares) [29]. They produce smooth postures but suffer from high computational costs, singularity issues, and complex matrix calculations.

Another class of IK solutions are based on Newton's method. The most well known algorithms include Powell's method [30], Broyden's method [31], and the BFGS (Broyden, Fletcher, Goldfarb and Shanno) method [32]. These methods avoid erroneous discontinuities while producing smooth motions. However, they suffer from complexity and high computational costs per iteration and are difficult to implement.

Further, [33] proposes another method called FABRIK (Forward And Backward Reaching Inverse Kinematics) which is the method that will be used to solve the IK problem in the MuJoCo simulator environment. The IK solver in MuJoCo does not have to produce exactly the same poses as the solver implemented by Robotis for the OpenManipulator-X in ROS. Thus the same technique for IK does not have to be implemented in MuJoCo. Nevertheless, an implementation is needed to control the manipulator to different initial positions for the RL training.

2.1.3 FABRIK

FABRIK [33] is an iterative method that finds each joint position on a line and readjusts the manipulator accordingly. See Figure 2.2 for an intuition of the process. The method removes the need to find rotational and translational matrices, resulting in convergence within a few iterations, and does not suffer from singularities. Another advantageous feature of FABRIK is the possibility of adding joint restrictions as well.

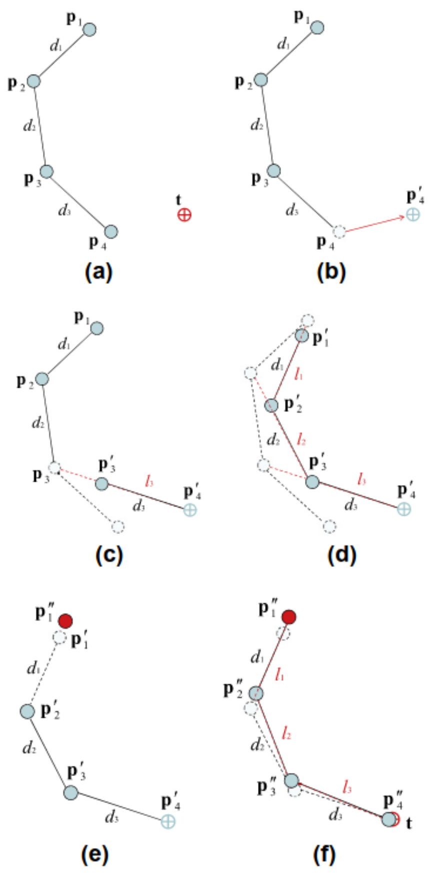


Figure 2.2: Forward reaching (a) - (d), and Backward reaching (e) - (f) of the FABRIK method visualized in a 2D plane. Image borrowed from [33].

The algorithm divides the IK problem into two phases: forward and backward reaching. During the forward reaching stage, the end effector is translated to the target, and the previous joint is moved to a new position along a straight line to the new location of the end-effector. The procedure repeats itself for all joints. The backward reaching stage is necessary as the root joint cannot move its position and must be held in place. It follows the same rules as forward reaching, except it starts at the root and iterates outwards to the end effector. A tolerance is set to end the loop when the error between target- and end-effector position is small enough. The algorithm is presented in Algorithm 1.

Algorithm 1 The FABRIK algorithm

Input: The joint positions $\mathbf{p}_i$ for i = 1,...,n, the target position $\mathbf{t}$ and the distances between each joint $d_i = |\mathbf{p}_i - \mathbf{p}|$ for i = 1,...,n-1.

Output: The new joint positions $\mathbf{p}_i$ for i = 1,...,n.

```
dist = $|\mathbf{p}_1 - \mathbf{t}|$                    ▹ The distance between root and target
# Check whether the target is within reach
if dist > $d_1 + d_2 + ... + d_{n-1}$ then
    for i = 1,...,n-1 do
        $r_i = |\mathbf{t} - \mathbf{p}_i|$          ▹ Distance between target and the joint position
        $\lambda_i = d_i / r_i$
        $\mathbf{p}_{i+1} = (1 - \lambda_i)\mathbf{p}_i + \lambda_i \mathbf{t}$          ▹ Find the new joint positions $\mathbf{p}_{i+1}$
    end for
else                                                  ▹ The target is reachable
    $\mathbf{b} = \mathbf{p}_1$
    $dif_A = |\mathbf{p}_n - \mathbf{t}|$
    while $dif_A > tol$ do
        # STAGE 1: Forward reaching
        $\mathbf{p}_n = \mathbf{t}$
        for i = n-1,...,1 do
            $r_i = |\mathbf{p}_{i+1} - \mathbf{p}_i|$
            $\lambda_i = d_i / r_i$
            $\mathbf{p}_i = (1 - \lambda_i)\mathbf{p}_{i+1} + \lambda_i \mathbf{p}_i$          ▹ Find the new joint positions $\mathbf{p}_i$
        end for
        # STAGE 2: Backward reaching
        $\mathbf{p}_1 = \mathbf{b}$                    ▹ Set the root $\mathbf{p}_1$ to its initial position
        for i = 1,...,n-1 do
            $r_i = |\mathbf{p}_{i+1} - \mathbf{p}_i|$
            $\lambda_i = d_i / r_i$
            $\mathbf{p}_{i+1} = (1 - \lambda_i)\mathbf{p}_i + \lambda_i \mathbf{p}_{i+1}$          ▹ Find the new joint positions $\mathbf{p}_{i+1}$
        end for
        $dif_A = |\mathbf{p}_n - \mathbf{t}|$
    end while
end if
```

2.2 Finite State Machine

An FSM (Finite State Machine) is a mathematical model of computations that can be in exactly one state out of a limited number of states. The FSM can change between states in response to some inputs, in what is called a transition [34]. An FSM is defined by a list of its states, the inputs that trigger each transition between all states, and its initial state. Figure 2.3 displays an FSM of an elevator, where the black dot indicates the starting state. All states are represented by circles and a description. The arrows indicate all transitions between states, and the triggering inputs are labeled. State machines can be designed at a high abstraction level as the elevator example, but they can also be far more complex and detailed.

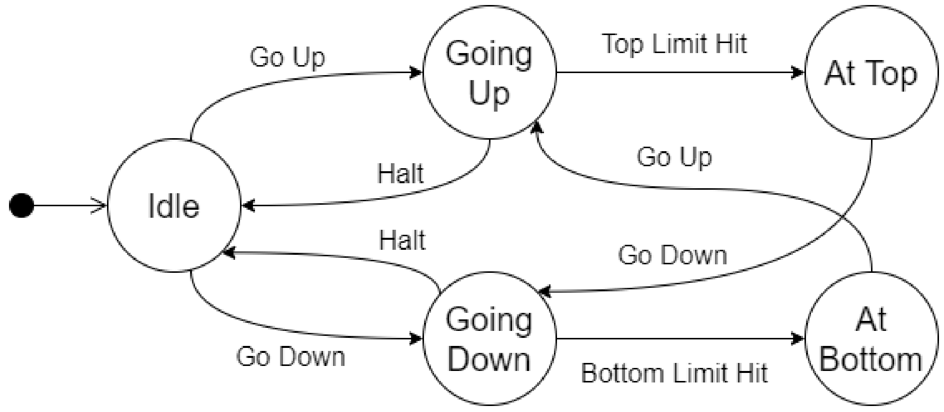


Figure 2.3: An example of an FSM description of an elevator system

2.3 Reinforcement Learning

Reinforcement Learning is a relatively new field within optimization and control problems and has shown great potential. A task in the RL task space is generally defined by the following elements: agent, policy, actions, states, and rewards. Actions and states can either be discrete or continuous.

2.3.1 Elements in an RL algorithm

Agent

The agent is the entity in the environment that the computer is allowed to control. In the autonomous vacuum cleaner example, one would assume that the agent could be the robotic cleaner, but this is a simplification. In reality, the robot is a part of the environment, as it consists of physical components such as wheels, motors, and a battery, enabling the agent to get around. These parts can exist in different states, impacting the agent's decision-making process. The agent cannot be allowed to directly control its environment, such as increasing the battery percentage without first charging.

Policy

The agent's policy, π, is the strategy that the agent follows when deciding on which action to take. The challenge is that the agent must visit all the states before being able to choose the actions leading to the optimal states.

In most cases, it will be too computationally demanding for the agent to explore all possible states, so a middle ground between exploration and exploitation must be found.

Action
For every accessible state in the environment, a set of actions is available to the agent. Taking an action can result in the agent transitioning to another state.

State
A state defines the environment in a given timestep and can be both dependent and independent on time. States contain information describing the agent's spatial and temporal location in the environment and the environment itself, such as the battery percentage for a robotic vacuum cleaner.

Reward
Some states are defined to give rewards to the agent, both positive and negative. The rewards are designed such that the agent can learn to do intended tasks. A robotic vacuum cleaner is supposed to clean the entire floor and return to its charging station before running out of battery. Thus a negative reward is given each time the robot fails to return before the battery depletes, and a positive reward is calculated based on how much of the floor area was swept. Perhaps it is desired to clean the floor as efficiently

as possible. The robot could get a small negative reward for each time step spent cleaning, incentivizing accurate and faster cleaning sessions. It is important to stress that the reward is defined in the environment and not in the agent, as the agent would then be able to modify its reward, achieving optimal accumulated reward while not doing the intended tasks. That is the reasoning behind moving physical parts such as battery status from the agent to the environment.

Off- and On-Policies

Off-Policy models differ from On-Policy models in the sense that Off-Policy models rely on one policy to choose which action to take, while another policy provides the most rewarding action based on current knowledge of the environment. We introduce the exploration-exploitation trade-off problem to understand why one would desire a suboptimal policy instead of the optimized one.

Exploration - Exploitation

The multi-armed bandit problem [35] describes the issue neatly. Imagine a row of slot machines, each with its own possibly unique reward distribution. The challenge is to choose which device to gamble on, as some of them will have a higher expected reward. The agent could be unlucky when selecting the best machine and lucky when choosing the worst performing one. A consequence of too little exploration means too much exploitation of the wrong devices. Thus, if it was lucky and got good rewards from the slot machine with the lowest expected reward, too much exploitation would cause it to not explore the other, better options. On the other hand, too much exploration would suggest choosing a random slot machine each time, and the agent would miss out on most of the rewards from the better ones, leaving it to chance.

A common technique within RL is the ϵ-greedy method [36], which involves choosing the optimal action out of a set of n actions at a rate of $(1 - \epsilon)$. Further, the method will also select one of the other actions at a rate of ϵ / n, increasing the exploration of the environment.

Model- VS Model Free Methods

Model-free methods do not use predictions of the environment's response during training or acting. Typical model-based methods include the classical Policy Iteration and Value Iteration algorithms [37], which use models to calculate the reward signals.

Q-Value and Bellman Equation

Multiple value functions exists, but the most relevant is called the optimal action-value function, Equation (2.5). It gives the expected reward if you start in state s, take an arbitrary action a, and follow the optimal policy π forever, resulting in the trajectory $\tau = (s_0, a_0, s_1, a_1, ...)$. $\tau \sim \pi$ is short for $\tau \sim \pi(\cdot|s)$.

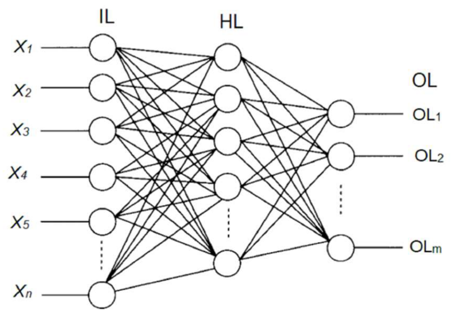
$$Q^*(s,a) = \max_{\pi} \underset{\tau\sim\pi}{E}[R(\tau)|s_0 = s, a_0 = a] \tag{2.5}$$

All value functions have their special self-consistency equations, known as Bellman equations. The idea is to let the reward be distributed over larger areas from some reward-yielding source. Thus, if an agent finds a path that leads to the reward, it is known which parts of the path were most vital for reaching the goal. More generally, Equation (2.6) describes the value of the starting state, plus the value of wherever you land next, s'. a' is the next action. $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, where P describes the environment's transition rules. $\gamma \in (0,1)$ is a discount factor.

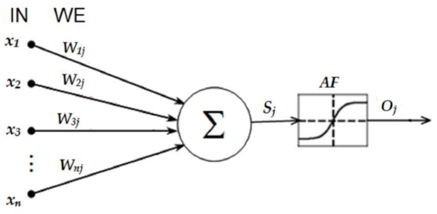
$$Q^*(s,a) = \underset{s'\sim P}{E}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{2.6}$$

2.3.2 Artificial Neural Networks

Material from this section has been reused from the project thesis [23] and modified to better fit this thesis.



(a)



(b)

Figure 2.4: (a): A neural network. (b): A close up of a single neuron in a fully connected network
Image borrowed from [38]

ANNs (Artificial Neural Networks) are computational models that intend to simulate biological neurons. A typical ANN, shown in Figure 2.4, consists of an input layer, n hidden layers, and an output layer. The network in the figure has exactly one hidden layer but they can have many more. Looking at figure (b), we can see that for all the neurons in a layer, all the inputs from the previous layer are multiplied by weights, summed, and passed through an activation function. A bias, not shown in the figure, is normally multiplied by a weight and summed together with the inputs before the activation function. The activation function mimics whether the neuron is activated or not, like in a physical brain. The ReLU activation function, for instance, makes all negative inputs zero while returning the positive ones. See Equation (2.7), where f is the activation function, and x the input signal.

$$f(x) = max(x, 0) \tag{2.7}$$

Another commonly used activation function is the Sigmoid, which maps all real values into the range $[0, 1]$. See Equation (2.8).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

While calculating the output from some input is called forward propagating, the distinctive learning algorithm is called backpropagation. For example, assuming we have a set of images of either cats or dogs which are labeled, we perform a forward propagation on a single image and compare the output with the corresponding label. The output is a vector with two decimals, each having a value between 0 and 1, where 1 means that the network is certain that a cat or a dog is present. Thus we can calculate the error between prediction and label with a loss function. The weights are updated according to how much they contributed to the error metric in question. This thesis will not delve into further details concerning the backpropagation algorithm.

A class of neural networks commonly referred to as convolutional neural networks (CNN) is popularly utilized to analyze visual imagery. [39]. These CNNs consist of hidden convolutional layers, where the term convolutional comes from the convolving process. A filter, or kernel, is initialized with random numbers for each convolutional layer and slides across the input, often an image. The dot product between the input image and the kernel is thus outputted from the convolutional layer. These filters are good at pattern recognition. By having multiple such layers consecutively, one layer can detect corners, circles, or squares while the next can detect more complex shapes such as hands, eyes, or hair. Further, each filter is trained, removing the need for designing each one by hand, which is a substantial advantage over traditional filtering methods.

2.3.3 DDPG

The DDPG algorithm is used to train the RL agent in this thesis.

Deep Deterministic Policy Gradient (DDPG) [40] is an off-policy, model-free, critic-actor reinforcement learning algorithm. It uses the Bellman equation and off-policy data to learn the Q-function, which is used to learn the policy π. See Algorithm 2 for the pseudo code of the DDPG algorithm.

It is similar to Q-learning in the sense that if you know the optimal action-value function $Q^*(s, a)$ you can acquire the optimal action by solving

Equation (2.9).

$$a^*(s) = arg\max_a Q^*(s, a) \tag{2.9}$$

The issue is thus to find an estimate of the optimal action value, which DDPG learns concurrently with the optimal action $a^*(s)$ approximator. As DDPG is designed specifically for continuous action spaces, it is not straightforward how we should compute the max over actions in Equation (2.9).

For finite action spaces, to find the optimal action is simply computing the action value for all actions and choosing the action which yields the highest value. There is no such trivial solution for continuous actions spaces, and common optimization techniques become far too computationally demanding to be calculated at each iteration.

$Q^*(s, a)$ is regarded as differentiable with respect to its action argument, allowing us to set up an efficient, gradient-based learning rule for a policy $\pi(s)$. Thus, instead of computationally demanding subroutines, we can take advantage of the approximation Equation (2.10).

$$\max_a Q(s, a) \approx Q(s, \pi(s)) \tag{2.10}$$

Going one step further, suppose that the approximator is a neural network, $Q_\phi(s, a)$. Where ϕ are the parameters, and we have collected a set $\mathcal{D}$ of transitions (s, a, r, s', d). d notes if s' is a terminal state. Using such an approximation, we can set up Equation (2.11), an MSBE (Mean-Squared Bellman Error), which indicates how close Q_ϕ comes to satisfying the Bellman equation.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{E}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right] \tag{2.11}$$

Replay Buffers

The set $\mathcal{D}$ is an experience replay buffer and contains previous experiences. The buffer should be large enough to include a wide range of experiences to ensure that the algorithm exhibits stable behavior. However, it is not wise to store everything due to computational limits. If you only keep the most recent data, the model will overfit. Tuning is needed to find a good trade-off.

Target Networks

Equation (2.12) is called the target since the minimization of the MSBE loss involves making the Q-function as similar as possible to this target.

$$r + \gamma(1-d)\underset{a'}{max} Q_\phi(s',a') \tag{2.12}$$

Unfortunately, the target depends on the same parameters we are trying to train, making the training unstable. To overcome this challenge, we can use a set of parameters that come close to ϕ, called ϕ_{targ}. In practice, we can achieve this by updating the parameters of the target network once per main network update by Polyak averaging [41]. See Equation (2.13), where ρ is a hyperparameter between 0 and 1.

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi \tag{2.13}$$

Due to the fact that the parameters of the target network are lagging behind ϕ, we say that the DDPG algorithm is off-policy. Similarly, the parameters of the target policy network, θ_{targ} is updated according to Equation (2.14)

$$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta \tag{2.14}$$

Due to the difficulties mentioned above when optimizing on continuous action spaces, a target policy network is used to compute an action that approximately maximizes $Q_{\phi_{targ}}$. The Q-learning in DDPG is performed by minimizing the MSBE given in Equation (2.15) with stochastic descent. $\pi_{\theta_{targ}}$ is the target policy.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{E}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)Q_{\phi_{targ}}(s', \pi_{\theta_{targ}}(s'))\right)\right)^2\right] \tag{2.15}$$

Policy Learning

We want to find a deterministic policy $\pi_\theta(s)$ that maximizes $Q_\phi(s, a)$. Assuming the Q-function is differentiable with respect to the action, we can perform gradient ascent on Equation (2.16), since the action space is continuous. The parameters of the Q-function is treated as constants during this step.

$$\underset{\theta}{max} \underset{s\sim\mathcal{D}}{E} \left[Q_\phi(s, \pi_\theta(s)) \right] \tag{2.16}$$

Algorithm 2 The DDPG algorithm

```
Input: Initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
  Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ} \leftarrow \phi$
  while no convergence do
      Observe state s and select action $a = clip(\pi_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
      Execute a in the environment
      Observe $s', r, d$
      Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
      if s' is terminal then
          Reset environment state
      end if
      if it's time to update then
          for however many updates do
              Randomly sample a batch of transitions, $\mathcal{B} = (s, a, r, s', d)$ from $\mathcal{D}$
              Compute targets
              $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{targ}}(s', \pi_{\theta_{targ}}(s'))$
              Update Q-function by one step gradient descent using
              $\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d)\in\mathcal{B}} (Q_\phi(s, a) - y(r, s', d))^2$
              Update policy by one step of gradient ascent using
              $\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} Q_\phi(s, \pi_\theta(s))$
              Update target networks with
              $\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi$
              $\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$
          end for
      end if
  end while
```

2.3.4 Hindsight Experience Replay

Hindsight Experience Replay (HER), developed by the authors of [42], seeks to deal with the problem of sparse rewards, which is a considerable problem for reinforcement learning, and especially so for robotics challenges. The method allows sample-efficient learning from sparse and binary reward signals, neglecting the need for complex reward functions. It can be implemented with any off-policy RL method.

The reward function must be carefully designed to describe the task at hand but also guides the policy optimization. Sometimes we do not know which behavior to expect, making it challenging to develop a proper reward function. In such cases, sparse rewards are preferred, as everything is left for the agent to decide.

HER will, for some of the state sequences in the replay buffer, replace the goal with the last state of that sequence. The trajectory does not teach us how to reach g, but it does tell us how to reach the last state, S_t. Algorithm 3 provides a pseudo code for the HER method.

Algorithm 3 The HER algorithm

```
Given:
  • an off-policy RL algorithm $\mathbb{A}$,
  • a strategy $\mathbb{S}$ for sampling goals for replay,
  • a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$
Initialize $\mathbb{A}$
Initialize replay buffer $R$
for episode=1, $M$ do
    for t = 0, T-1 do
        Sample an action $a_t \leftarrow \pi_b(s_t||g)$
        Execute the action $a_t$ and observe a new state $s_{t+1}$
    end for
    for t = 0, T-1 do
        $r_t := r(s_t, a_t, g)$
        Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in $R$
        Sample a set of additional goals for replay $G := \mathbb{S}(\textbf{current episode})$
        for $g' \in G$ do
            $r' := r(s_t, a_t, g')$
            Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in R
        end for
    end for
    for t = 1, $N$ do
        Sample a minibatch $B$ form replay buffer $R$
        Perform one step of optimization using $\mathbb{A}$ and minibatch $B$
    end for
end for
```

2.4 Robotic Vision

Robotic vision encompasses visual sensor information, advanced object detection, and orientation algorithms. SINTEF provides the following description: "The aim of robot vision is to make a wide range of robot platforms able to interact with the world around them through visual inputs" [43]. One such detection algorithm used for finding the camera's orientation and position is the PnP (Perspective-n-Point) algorithm [44].

2.4.1 Perspective-n-Point Pose Estimation

Perspective-n-Point (PnP) is the problem of estimating the camera pose from a set of n image points and corresponding object points, whose coordinates are usually measured. It is tightly linked with the camera calibration problem, where one would want to estimate a set of distortion parameters describing the physical imperfections of the camera lens. Mainly two types of distortions are considered: radial and tangential. Numerous methods for estimating these parameters exist, but Zhang's method [46] is perhaps the most widespread one. Figure 2.5 shows an overview of the PnP problem. The X-axis points to the right in the camera coordinate system, the Y-axis points downwards, and the Z-axis points outwards.

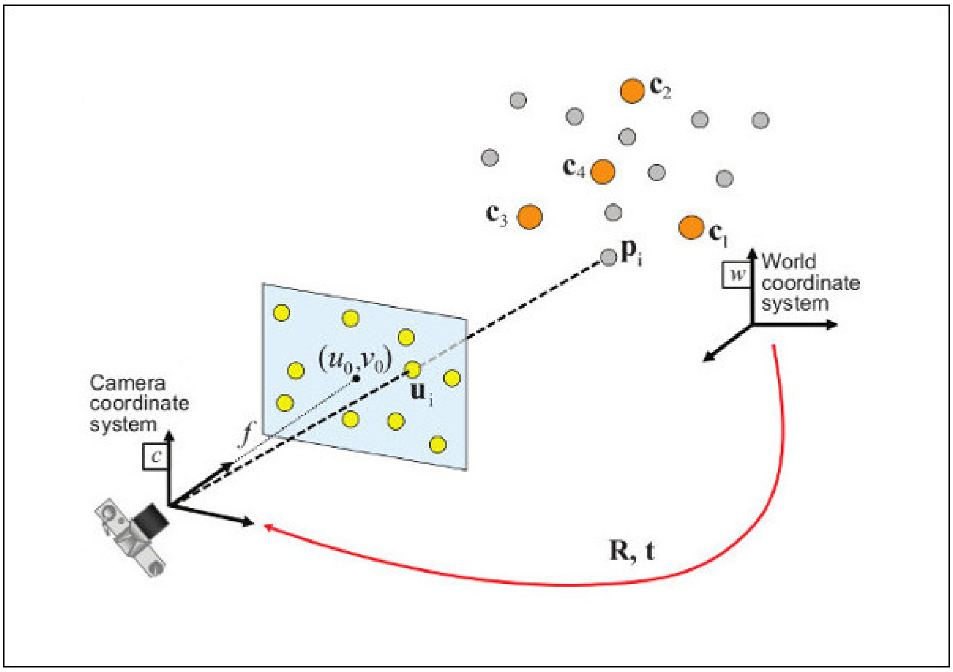


Figure 2.5: Work flow for the PnP pose computation problem. Image borrowed from OpenCV [45]

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \boldsymbol{H}_C^O \begin{bmatrix} x_{obj}^W \\ y_{obj}^W \\ z_{obj}^W \\ 1 \end{bmatrix} \tag{2.17}$$

In Equation (2.17) object points $[x_{obj}^W \; y_{obj}^W \; z_{obj}^W]^T$ are transformed into image points $[u, v]^T$ by the homography $\boldsymbol{H}_C^O$, which is based on the pinhole camera model [47]. The object points describe known points in world coordinates, often corners in checkerboard patterns or QR-codes, while the image points are pixel coordinates. Thus, one can estimate the homography by knowing the image- and object points given in world coordinates.

$$\boldsymbol{H}_C^O = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R}_{3x3} & \boldsymbol{T}_{3x1} \\ \boldsymbol{0}_{1x3} & 1 \end{bmatrix} \tag{2.18}$$

$\boldsymbol{H}_C^O$ is given by Equation (2.18), where $\boldsymbol{K}$ is the camera's intrinsic parameters and the $[\boldsymbol{R}_{3x3} \; \boldsymbol{T}_{3x1}]$ matrix is referred to as the extrinsics. For the PnP problem, we are mainly interested in obtaining the extrinsic parameters that describe the pose of the camera.

$$\boldsymbol{K} = \begin{bmatrix} f_x & \gamma & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.19}$$

For camera calibration, the intrinsic parameters $\boldsymbol{K}$ see Equation (2.19) are also estimated along with a set of distortion parameters. f_x and f_y are the scaled focal lengths, while u_0 and v_0 is the principle point. γ is the skew parameter, typically assumed at 0. s is a scale factor that arises because of the homogeneous vectors.

Due to the limitation of measuring the object coordinates by hand, patterns with known object coordinates are usually photographed. In this thesis, Aruco indices are utilized, see Section 3.6.2. The intrinsics are given by the camera manufacturer. We assume the camera sensor has been calibrated correctly, and that no distortion is present.

Once all points have been detected in the image, usually by extensive filtering, and the object points are measured, the transformation matrix $[\boldsymbol{R}_{3x3}\ \boldsymbol{T}_{3x1}]$ can be found. In Zhang's camera calibration method, the object points on checkerboards are always defined with zero Z^W_{obj} value. We will use the same trick here such that the expression is simplified as shown in Equation (2.20), reducing the number of $[\boldsymbol{R}_{3x3}\ \boldsymbol{T}_{3x1}]$ parameters we need to find from 12 to 9.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x^W_{obj} \\ y^W_{obj} \\ 0 \\ 1 \end{bmatrix} = K\begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x^W_{obj} \\ y^W_{obj} \\ 1 \end{bmatrix} \tag{2.20}$$

r_{ij} and t_i are the elements of the rotation and translation matrices. Each image-object point correspondence will then yield three equations in the form of Equation (2.21). Following these equations, we see that a minimum of three image-object point correspondences are needed to acquire the sufficient amount of equations to solve for r_{ij} and t_i.

$$\begin{aligned} su &= (f_x r_{11} + u_0 r_{31})x^W_{obj} + (f_x r_{12} + u_0 r_{32})y^W_{obj} + f_x t_1 + u_0 t_3 \\ su &= (f_y r_{21} + v_0)x^W_{obj} + (f_y r_{31} + v_0 r_{32})y^W_{obj} + f_y t_2 + v_0 t_3 \\ s &= r_{31}x^W_{obj} + r_{32}y^W_{obj} + t_3 \end{aligned} \tag{2.21}$$

2.5 ROS - Robot Operating System

This section has been reused from the author's project thesis [23] with minor adjustments.

The Robot Operating System is a general open source framework for writing robotic applications. It contains conventions, libraries and tools that are helpful when working with a robotic manipulator. In this project, ROS is used due to its tools for communicating with the manipulator, which has custom built packages specifically for use with ROS. The framework is available in multiple distributions, but here *ROS Noetic* is used due to its compatibility with Python, the OpenManipulator-X, and the Ubuntu 20.04 operating system.

To communicate with the manipulator, one should have a general understanding of the following modules in ROS: *Nodes, Topics, Services* and *Actions.*

Nodes
A node is responsible for a single modular operation, which could be to control the wheels in a car, or in our case to control a robotic manipulator. Another node could handle user input, and would probably want to transmit this information to the controller node. Which is what *Nodes, Topics* and *Services* are used for.

Topics
Nodes use topics to either publish data, or subscribe to data being published by other nodes. Topics can either be one-to-one, one-to-many, many-to-one or many-to-many communication and are meant for continuous streams of data. There are however no handshakes between the nodes, and one node does not know how many nodes has successfully received the data being pushed on a topic.

Services
A service is a more direct way of communicating than a topic, and is based on a call-and-response model. Each Service has exactly one server node that waits on requests from one or more client nodes, and transmits a response to the client that made a request. A client can for example ask for the joint positions of a robotic arm with one dedicated service.

Actions
Actions combine topics and services into a third communication type, meant for longer running tasks. Every action consists of three parts, a goal, a feedback and a result. The goal and result are services while the feedback is a topic. An action client first sends a goal to the server, which responds and acknowledges the goal. Next, the client asks for a result, and while the action is being performed, the server streams data though the feedback topic. Once the goal has been achieved, or is cancelled, a result is transmitted to the client.

Chapter 3

Experimental Setup

To test the system developed in Chapter 4, a robotic manipulator is used to pull a lever into a specific angular position. This section describes the experimental setup of that test: The lever-manipulator setup, how the HMCI from Section 4.5 records the operator's hand, the third-party software needed to run the system, the robotic manipulator and depth sensors, the different simulators and how the RL agent is trained, how the lever angle and position is measured.

3.1 Overview

The setup consists of the *OpenManipulator-X*, two *Intel RealSense D435* depth sensors, a lever connected to a potentiometer, and a *Raspberry Pi 3 Model B 1GB RAM*, as illustrated by Figure 3.1. Three Aruco indices were attached to the lever by a 3D-printed holder, as seen in Figure 3.1. A custom-made 3D printed adapter also attaches one stereoscopic sensor to the manipulator's end-effector. Both attachments were designed in FreeCAD and printed using a Prusa i3 MK3S 3D printer.

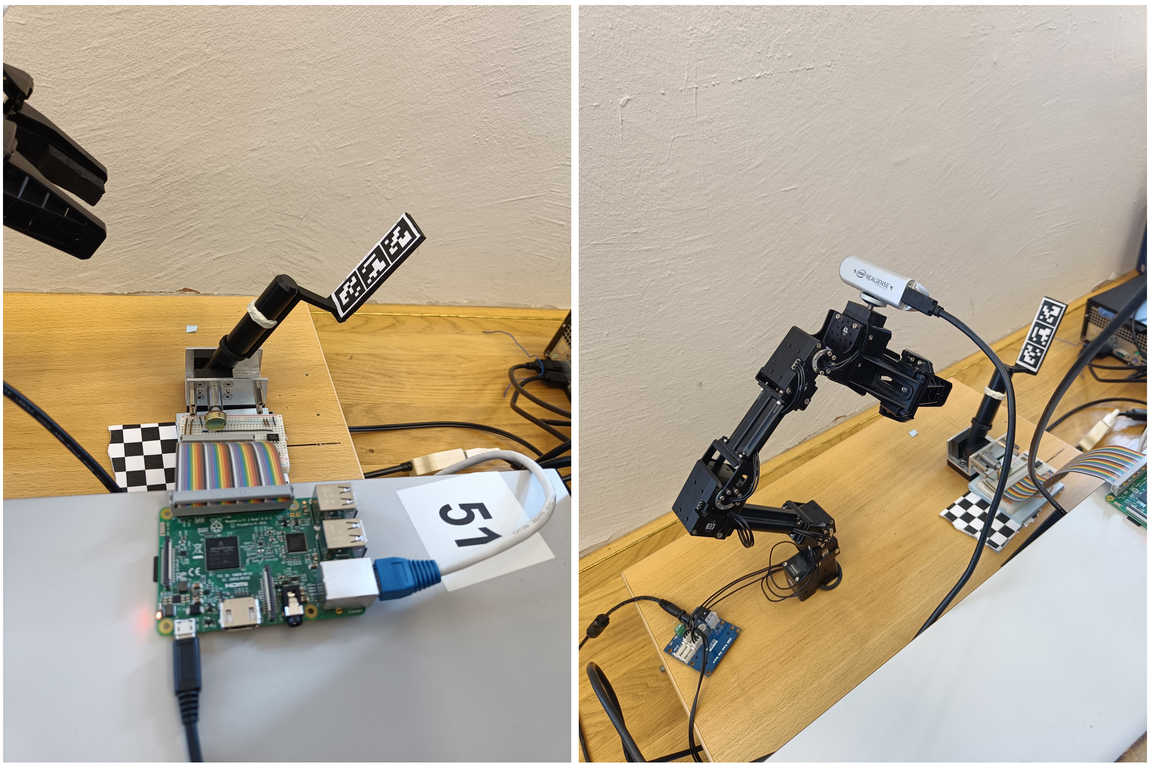


Figure 3.1: Left: The lever setup with a potentiometer and Aruco indices. Right: the OpenManipulator-X with an Intel RealSense D435 stereoscopic sensor mounted on its end-effector.

The manipulator was mounted on a wooden platform, together with a lever constructed by the author of [22]. The lever is approximately $24cm$ in front of the manipulator, along the X^W-axis. Figure 3.1 depicts both the lever and the manipulator. Further, Figure 3.2 exhibits a stand for the stereoscopic depth sensor, made during the project thesis [23]. Three grooves can be seen in the vertical arm, at approximately $58cm$, $72cm$, and $85cm$. Experiments were carried out to determine at what distance the depth sensor worked optimally, but in the end, the middle notch yielded the most reliable hand tracking.

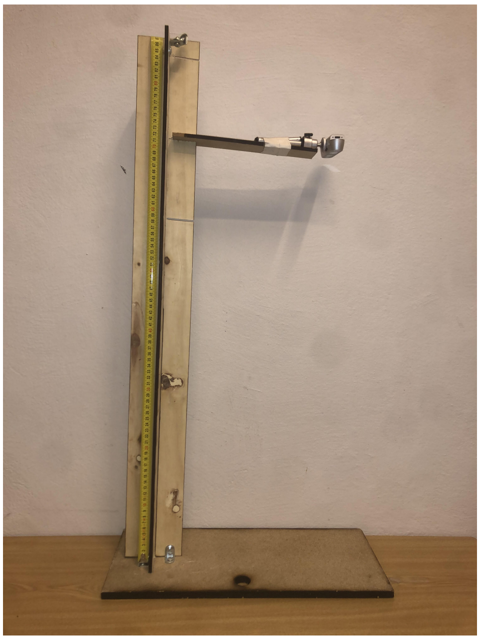


Figure 3.2: The plywood camera stand for hand gesture detection.

Ubuntu 20.04.3 LTS was selected as the operating system due to dependencies by the provided control system for the OpenManipulator-X and ROS. Table 3.1 lists all the dependencies necessary to run the HMCI developed in this thesis.

Name	Version
Ubuntu	20.04.03 LTS
ROS	NOETIC
OpenManipulator	2.0.2
OpenManipulator Msgs	1.0.1
OpenManipulator Dependencies	-
Robotis Manipulator	1.1.1
Python	3.8.10
Rospy	1.15.14
PyQt5	5.14.1
QDarkStyle	3.0.3
Gym	0.21.0
Numpy	1.20.0
Matplotlib	3.1.2
OpenCV2	4.5.5
MediaPipe	0.8.9.1
Scipy	1.8.0
PyRealSense2	2.50.0.3812
Pillow	9.0.1
mpi4py	3.0.3
PyTorch	1.10.2
rospkg	1.4.0
mujoco-py	2.1.2.14

Table 3.1: The packages used to develop the HMCI and RL training environment.

3.2 Robotic Manipulator

This section is reused, with modifications, from the author's project thesis [23].

The robotic manipulator OpenManipulator-X was used in the experimental setup. The manipulator is cost-effective, consisting of both open-source software and hardware. The software uses ROS packages for communication and control, while most of the manipulator parts are available as STL CAD models, allowing for 3D printing. The manipulator is mounted on a wooden platform, as seen in Figure 3.1.

In Section 2.1, the manipulator was briefly introduced as part of the Forward Kinematics section. Both FK and IK are handled by the supplied ROS packages for both the physical model, and the Gazebo environment. Appendix A.1 displays the dimensions of the manipulator, while fig. 2.1 shows the coordinate frames drawn by the author, according to the DH convention. The DH parameters are given in Table 3.2, and are used by the controller developed for the MuJoCo environment.

Link	θ_i	d_i	a_i	α_i
1	$\theta_1 = \beta$	0.077m	0	$\frac{\pi}{2}$
2	$\frac{\pi}{2} - arcsin(\frac{0.024}{0.130}) + \theta_2$	0	0.130m	0
3	$\theta_3 - \frac{\pi}{2}$	0	0.124m	0
4	θ_4	0	0.126m	-$\frac{\pi}{2}$

Table 3.2: The DH parameters used for FK

Further, the manipulator is connected to the computer running the HMCI through a USB communication converter, U2D2.

3.3 Depth Sensor

This section is partly reused from the author's project thesis [23].

A depth sensor is used together with the hand tracking software to utilize depth information from the hand. Seen in Figure 3.3 is the stereoscopic sensor, Intel RealSense Depth Camera D435. It is powered by a USB 3.0 cable and consists of a pair of depth sensors, one plain RGB camera, and an IR (Infrared Projector). Further, the camera has a wide depth FOV (Field Of View) of 87° x 58° and a range of 30cm to 10m. But since we are relying on the RGB sensor information when tracking hands and Aruco indices, we are constrained by the RGB sensor's FOV of 69° x 42°. The IR projector is handy for improving depth accuracy in environments with lower texture gradients.

Figure 3.3: The stereoscopic depth camera

Because the depth and RGB sensor are not perfectly overlapping with the depth sensors, we cannot directly extract depth information from pixels in the RGB image. Luckily, Intel has supplied the depth camera with an SDK: Intel RealSense SDK 2.0 for Python.

Since the minimum range of the sensor is $30cm$, the depth information cannot be scrutinized in the lever estimation process, as the lever is usually closer than $30cm$. Thus, only the RGB sensor on the camera mounted on top of the manipulator's end-effector is used.

3.4 Simulator environments

3.4.1 Gazebo

Gazebo is a popular open-source simulation environment maintained by Open Robotics. The environment is a collection of libraries targetting robot developers, designers, and educators. Gazebo has a set of ROS packages that provide wrappers around the standalone Gazebo version, allowing for easy integration with ROS. Unfortunately, it is difficult to train an RL agent in Gazebo, as increasing the timesteps of the simulation to speed up training yields an unstable simulation [48].

3.4.2 MuJoCo

MuJoCo (Multi-Joint dynamics with Contact) [49] is a general-purpose physics engine. It is a platform for research and development within robotics, machine learning, and graphics, to name a few. Fast and accurate simulations are crucial, especially within reinforcement learning, as models are often first trained in simulators before being applied in the real world. MuJoCo was initially developed by Roboti LLC but was acquired by DeepMind in 2021 and made open-source and free of charge.

The library is written in C/C++, but Python bindings have been made available by OpenAI. A built-in XML parser and compiler preallocates low-level data structures, tuning the runtime simulation module to maximize performance. Models are user-defined in the native MJCF scene description language - a humanly readable XML file format.

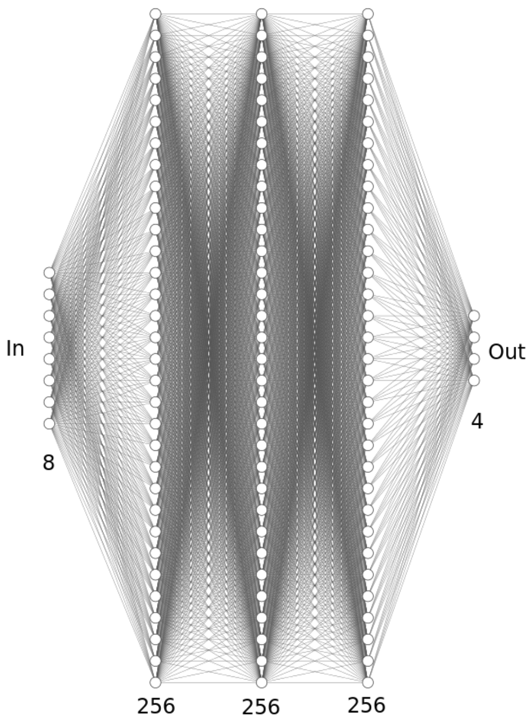
MuJoCo was chosen as the simulation environment for the training, due to its computational speed and stability [48]. Gazebo was initially tested for this purpose, but the author arrived at the conclusion that the training took too long, deeming Gazebo subpar for RL-training applications. The author of [22] chose to train the model in PyBullet and transfer the model to Gazebo for final training before testing it on the real-world manipulator. In this thesis, MuJoCo is used for all RL training.

3.4.3 Gym Environments

Gym is an open-source Python library by OpenAI and facilitates the development and comparison of reinforcement learning algorithms. It provides a standard API for communicating between environments and learning algorithms. The API includes methods for resetting the environment and stepping one action at a time. Reset is used at the start of each episode, while step ensures that the intended action is successfully performed in the environment.

3.5 RL Agent

An environment was set up in the MuJoCo simulator, as discussed in Section 4.5.2, to train the reinforcement learning agent to pull the lever correctly. The DDPG algorithm was used with the same set of hyperparameters used in [22], listed in Table 3.3. The DDPG method employs an actor-critic neural network pair, with the structure outlined in Figure 3.4. The eight inputs of the actor network are the joint positions, lever angle, goal, and relative position to the lever along the X^W and Z^W axes. The outputs are control signals in the range of $[-1, 1]$ which are multiplied by a tuned scale factor of 0.2 before being applied on the manipulator's joints. Further, the inputs to the critic network are the same inputs the actor uses, including the outputs from the actor, a total of 12 values. The critic output is an estimate of the optimal action-value function, Equation (2.6).



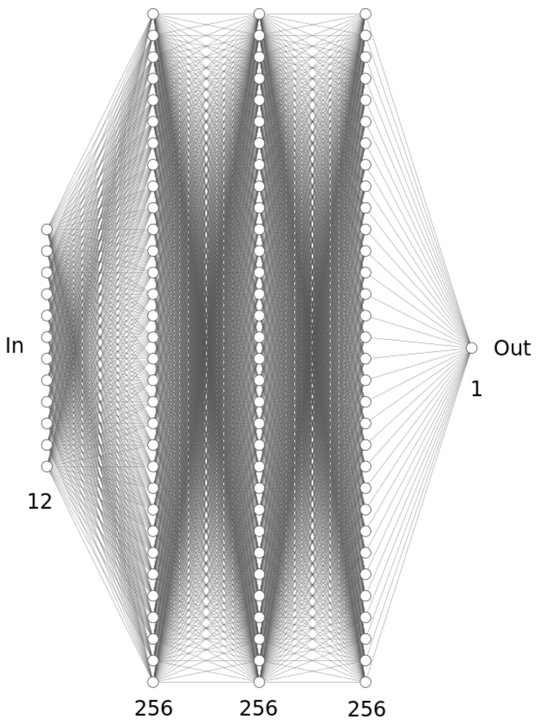


Figure 3.4: Left: Actor Network. Right: Critic Network. Drawn with NN-SVG. The hidden layers have been simplified in this drawing due to size constraints.

Hyperparameter	**Value**
Nr. of epochs	50
Nr. of cycles	50
Nr. of batches	40
save interval	5
replay strategy	future
clip return	50
noise ϵ	0.2
random ϵ	0.18
buffer size	1000000
replay k	4
clip obs	200
batch size	128
γ	0.98
action l2	1
lr actor	0.001
lr critic	0.001
polyak	0.95
n test rollouts	30
clip range	5
num rollouts per mpi	2

Table 3.3: List of hyperparameters

The manipulator has two different starting scenarios for each training cycle, inspired by the work from [22]. In one case, the manipulator starts by grabbing onto the lever. The gripper is locked for the first 75 timesteps of this initial position, ensuring that the manipulator will not release the lever too early. This initial position is implemented to increase exploration.

In the second initial position, the manipulator is adjusted to the joint values of $[0.0, 0.0, -1.05, 0.35, 0.70]$ given in radians. The first value is the gripper position, while the remaining four are the other joints, from *Joint 1* to *Joint 4*. *Joint 1* controls the horizontal angle β. Gaussian noise of $+/-0.2$ rad is then appended to the initial position's last three joint values to further increase exploration. The manipulator is assumed to already point towards the lever, meaning that *Joint 1* always has an actuator value of 0. Either of these initial positions has a 50% chance of occurring.

For each episode, the lever is spawned at a random location in the range $[0.23m, 0.28m]$ along the X^W-axis, Y^W-position is always $0m$, and Z^W-position is always $0.04m$. This is done to mimic measurement errors from the physical setup.

The agent is trained for 50 epochs with 50 cycles in each epoch and two rollouts per cycle, meaning that the manipulator is tasked with placing the lever in $50 * 50 * 2 = 5000$ different goal angles for each training session. The validation success ratio is produced on 30 new cycles.

During training, it was noticed that there was a gap between validation errors and training errors. The agent successfully achieved the goal in every training cycle but had a lower success rate during evaluation. Likely due to the exploration noise added to the steps taken during training and not during evaluation, which is standard procedure in the DDPG algorithm. The DDPG implementation used in this thesis is based on an implementation developed by OpenAI [50]. In an effort to minimize the difference in training/evaluation success, the constant Gaussian noise was replaced by the diminishing version given by Equation (3.1).

$$\epsilon_i = \frac{0.2}{i}, i \in [1, 51] \tag{3.1}$$

i is the current epoch number, and ϵ_i is the Gaussian noise appended to the action in that epoch. With a diminishing action noise, we utilize the exploration advantage but also push the agent to be more precise with its actions when it becomes more confident. The validation success rate steadily increased to adequate levels afterward.

$$r(s,a) = -1 - ||\omega_k - \omega_g|| - ||\boldsymbol{X}^W_{EF} - \boldsymbol{X}^W_{lever}|| \tag{3.2}$$

Two reward functions were tested, one sparse and one dense. The dense reward function is given by Equation (3.2), where ω_k is the measured ($k = m$) or estimated $k = e$ lever angle, while ω_g is the goal angle. $\boldsymbol{X}^W_{ef}$ is the position of the manipulator's end-effector in world coordinates and $\boldsymbol{X}^W_{lever}$ is the position of the lever case also in the world frame.

$$r(s,a) = \begin{cases} 0, & \text{if } \|\omega_k - \omega_g\| \leq 0.025 \\ -1, & \text{otherwise} \end{cases} \tag{3.3}$$

The second reward function is sparse and given by Equation (3.3). It is either 0 if the goal angle ω_g has been reached, or -1 otherwise.

3.6 Lever Pose Estimation

The physical lever, depicted in Figure 3.1, shall be controlled by the manipulator to a pre-determined angle. Thus, we introduce two different ways of acquiring values for the lever angle and position. First, potentiometer readings give an angle estimate which can be useful in scenarios where the RL agent has access to continuously measured lever or valve positions. The second method uses an RGB camera lens together with the PnP algorithm and Aruco indices physically attached to the lever, to estimate both the lever angle and relative position to the manipulator's end-effector.

3.6.1 Potentiometer

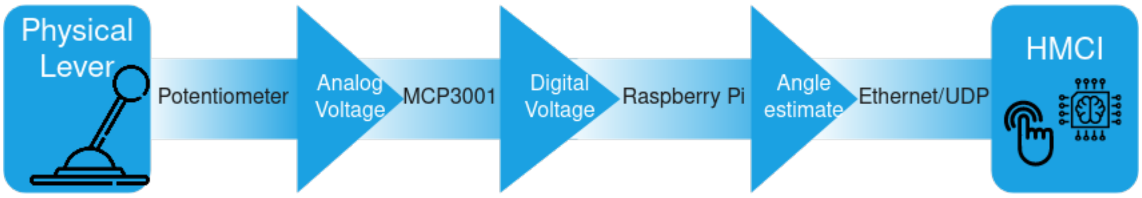


Figure 3.5: Information flow of the lever angle measurement using a potentiometer

A 10$k\Omega$ potentiometer was connected to the lever axis as depicted in figure Figure 3.1, similar to what was done by the author of [22]. The resistance in the potentiometer changes linearly as the lever angle varies. Applying a DC current on the potentiometer, the voltage over the resistor will also vary linearly with the lever angle, according to Ohms law, Equation (3.4), where I is the current and R the resistance. Figure 3.5 describes the information flow from physical lever angle to measured lever angle being sent to the Control System.

$$V = I * R \tag{3.4}$$

A *Raspberry Pi 3 Model B 1GB RAM* is used to power the potentiometer and record the digital voltage signal. The Pi, not able to read analog signals, requires the use of an ADC (Analog to Digital Converter). In this project the MCP3001 [51] is used, see Figure 3.6 for the pin layout. The ADC represents the voltage signal with 10 bits, but due to physical limitations, not all bits are used. The accuracy of the measured lever angles is estimated to be 0.24 deg.

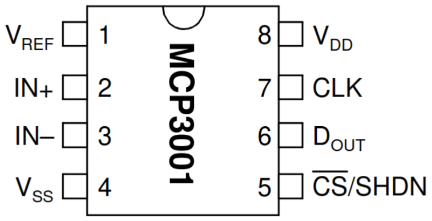


Figure 3.6: Pins of the ADC MCP3001. Borrowed from the datasheet

The complete circuit is presented in Figure 3.7. The Raspberry Pi is not drawn in, but all 20 even-numbered GPIO pins are given, where GPIO2 is the leftmost pin. Three of the odd-numbered pins are shown explicitly in the figure as SPI pins. The correct pins were found using the schematic provided on the official Raspberry Pi webpage [52].

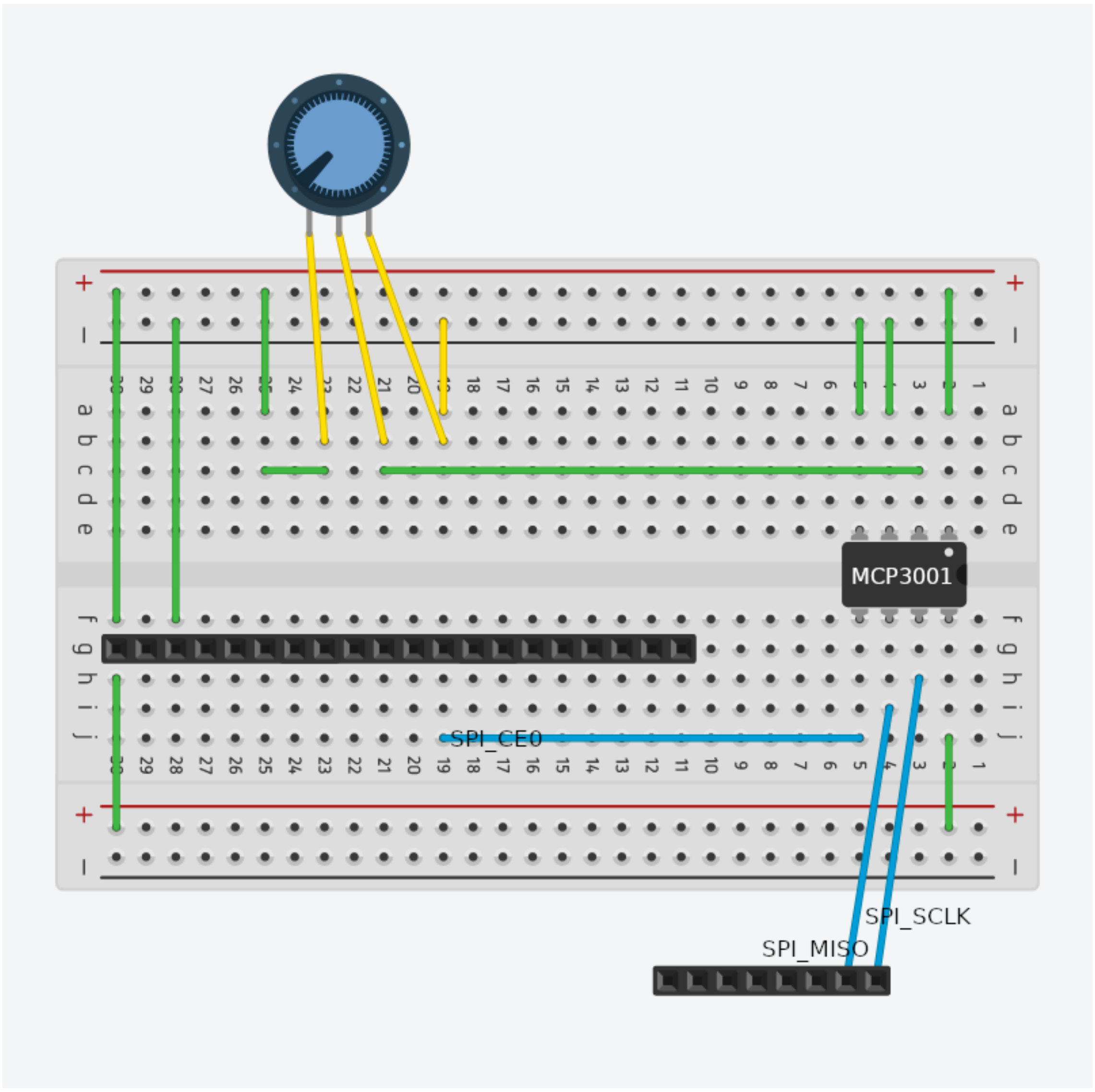


Figure 3.7: Circuit design for the potentiometer readings of the lever angle

The circuit is powered by the $+5V$ and ground pins from the Raspberry Pi. Both voltage values are sent to the plus and minus columns on the breadboard such that both the potentiometer and the ADC receive power. The voltage over the variable resistor is presented to the ADCs $IN+$ pin, while ground is sent to $IN-$ pin. The converted voltage signal is sent out from the D_{OUT} to the SPI_MISO pin on the Raspberry using an SPI (Serial Peripheral Interface) [53].

SPI is primarily used for short-distance communication such as embedded. The SPI bus specifies the following four logic signals:

- SCLK: Serial Clock
- MOSI: Master Out Slave In
- MISO: Master In Slave Out
- CS: Chip Select

Only SCLK, MISO, and CS are necessary for our circuit. The data pin can send one bit at a time, which happens at either falling or rising SCLK edges. Instead of reading each bit manually, Python's spidev library was used together with the Raspberry Pi's dedicated SPI pins in a script added to the Pi's startup procedure. The digital voltage signal, a number between 124 and 917, is linearly related to the analog voltage. The lever angle was measured to have a maximum value of 98° and a minimum value of -96°. The lever angle can be calculated readily by utilizing this relationship. Next, the lever angle, given in radians, is sent over an ethernet cable connected to the host computer with the communications protocol UDP [54]. The script is attached in the appendix, see Appendix A.4. The notation for the lever angle measured by this method is ω_m.

3.6.2 Visual Estimation

The other method for estimating the lever angle can be helpful for robots altering actuators on systems where the actuator position is not directly measured. We rely on a live stream of images from a camera on the end-effector to estimate the lever angle from afar. Three Aruco indices are mounted on the lever as seen in Figure 3.1. They provide points with known relative distances and are extracted from the images with OpenCV's Aruco framework [55]. Next, the detected image points and their corresponding object points, which are measured physically, are sent to the OpenCV's implementation of the PnP algorithm.

$$\boldsymbol{R}_C^O, \boldsymbol{t}_C^O = \text{solvePnP}(\boldsymbol{X}_{obj}^O, \boldsymbol{X}_{img}^I), \quad \boldsymbol{X}_{obj}^O \in \mathbb{R}^{3xn},\ \boldsymbol{X}_{img}^I \in \mathbb{R}^{2xn} \tag{3.5}$$

The output given by Equation (3.5) is a translational, $\boldsymbol{t}_C^O$, and a rotational, $\boldsymbol{R}_C^O$, vector, describing the pose of the object points related to the camera. To find the angle of the Aruco board in the $\boldsymbol{X}^W Z^W$-plane and thus the angle of the lever, we have to transform the detected points into our world frame.

The object points $\boldsymbol{X}^{O}_{obj}$ are measured and given in the object frame, while the corresponding image points $\boldsymbol{X}^{I}_{img}$ are pixel coordinates, outputted by the Aruco framework, see the visualized points in Figure 3.8.

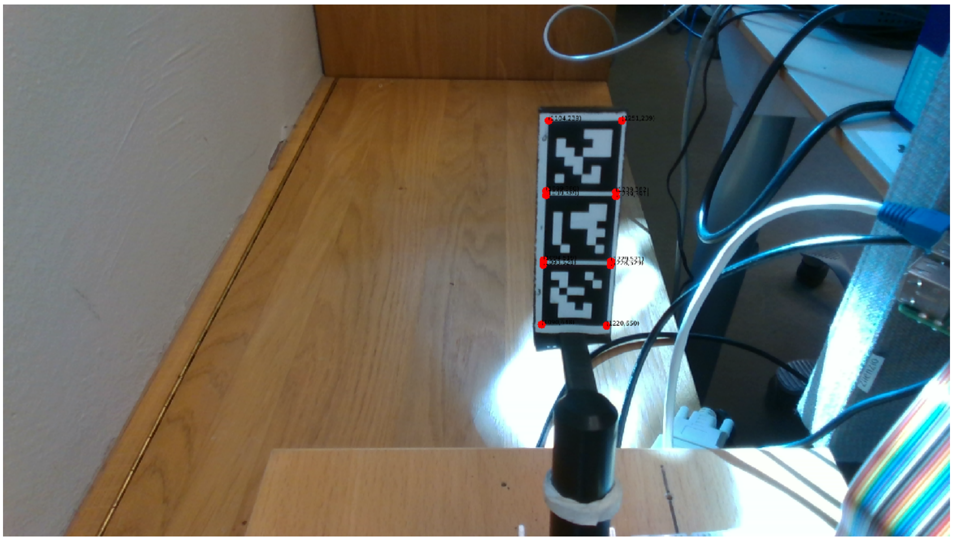


Figure 3.8: Detected corners of the Aruco indices, as seen from the RGB sensor mounted on the manipulator's end-effector.

$$\boldsymbol{H}^{O}_{C} = \begin{bmatrix} \boldsymbol{R}^{O}_{C} & \boldsymbol{t}^{O}_{C} \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3.6}$$

$\boldsymbol{R}^{O}_{C}$ and $\boldsymbol{t}^{O}_{C}$ are used to construct the homography $\boldsymbol{H}^{O}_{C}$ from object frame to camera frame in Equation (3.6).

$$\boldsymbol{H}_{EF}^{C} = \begin{bmatrix} 0 & 0 & 1 & -0.067 \\ 0 & 1 & 0 & -0.076 \\ -1 & 0 & 0 & 0.0325 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

Next, Equation (3.7) gives us the transformation from camera frame to end-effector frame. The translational parameters were measured with a ruler and are prone to be inaccurate.

$$\boldsymbol{H}_{W}^{EF} = \boldsymbol{H}_{W}^{J1}\boldsymbol{H}_{J1}^{J2}\boldsymbol{H}_{J2}^{J3}\boldsymbol{H}_{J3}^{J4}\boldsymbol{H}_{J4}^{EF} \tag{3.8}$$

Further, to obtain a transformation from the end-effector to world coordinates, the DH transforms of the manipulator are used in Equation (3.8). These matrices are dependent on the joint positions and physical dimensions of the manipulator, see Section 3.2.

$$\boldsymbol{H}_{W}^{O} = \boldsymbol{H}_{W}^{EF}\boldsymbol{H}_{EF}^{C}\boldsymbol{H}_{C}^{O} \tag{3.9}$$

Finally, by combining the results from Equation (3.6), Equation (3.7), and Equation (3.8) we arrive at a transformation from object space to world space, Equation (3.9).

$$\boldsymbol{X}^{W}_{obj,i} = \begin{bmatrix} x^{W}_{k,i} \\ y^{W}_{k,i} \\ z^{W}_{k,i} \end{bmatrix} = \boldsymbol{H}^{\boldsymbol{O}}_{\boldsymbol{W}} \boldsymbol{X}^{O}_{obj,i} \tag{3.10}$$

Equation (3.10) gives us the object points represented in the world frame, $\boldsymbol{X}^{W}_{obj}$, where k is either l for the corners detecetd on the left column, or r for the points detected on the right column.

$$\begin{aligned} \omega_{e,l} &= \arctan\left(\frac{\sum_{i=1}^{n} \left(x^{W}_{l,i} - x^{W}_{l,i-1} \right)}{\sum_{i=1}^{n} \left(z^{W}_{l,i} - z^{W}_{,i-1} \right)} \right) \\ \omega_{e,r} &= \arctan\left(\frac{\sum_{i=1}^{n} \left(x^{W}_{r,i} - x^{W}_{r,i-1} \right)}{\sum_{i=1}^{n} \left(z^{W}_{l,i} - z^{W}_{l,i-1} \right)} \right) \\ \omega_{e} &= \frac{1}{2} \left(\omega_{e,l} + \omega_{e,r} \right) \end{aligned} \tag{3.11}$$

An estimate for the lever angle is found by taking *arctan* of the mean of the differences in x^W coordinates of adjacent points, over the mean of the differences in z^W coordinates as shown in Equation (3.11).

Finally, an estimate of the lever position in the world frame is given by Equation (3.12). The lever position in object coordinates, $\boldsymbol{X}^{O}_{lever}$, is measured with a ruler and may be prone to some inaccuracies.

$$\boldsymbol{X}^{W}_{lever} = \boldsymbol{H}^{\boldsymbol{O}}_{\boldsymbol{W}} \boldsymbol{X}^{O}_{lever} = \boldsymbol{H}^{\boldsymbol{O}}_{\boldsymbol{W}} \begin{bmatrix} -0.05m \\ 0.0085m \\ -0.095m \end{bmatrix} \tag{3.12}$$

Chapter 4

Problem Formulation and System Design

This chapter starts by describing the problem addressed in this thesis. The system is briefly presented as a whole before each part is dissected into subsections. The first parts of the system design fathoms the HMI developed in the author's project thesis [23], and the *OpenManipulator RL Environments* package developed in [22]. The final section weaves the last two parts into a complete system.

4.1 Problem Formulation

The research questions in Section 1.2 generally outline the scope of the thesis. A challenge to test if a human and a trained RL agent can cooperate to solve a complex task is presented in this section. In short, the manipulator is used as a tool to manipulate a lever into a pre-determined goal angle. The setup is tested in two scenarios:

1. The RL agent receives measurements of the lever pose, and the manipulator is automatically guided such that the relative y-position between end-effector and lever is close to zero. $y_{EF}^{W} - y_{lever,m}^{W} \approx 0$. The agent is triggered by a hand signal from the human operator and disabled when the hand signal is no longer present.
2. The RL agent relies on visually estimated values for lever pose, and the manipulator is automatically guided such that $y_{EF}^{W} - y_{lever,e}^{W} \approx 0$. The agent is triggered by a hand signal and stopped when the operator discontinues the signal.

An essential aspect of HMC is the notion of safety. Who is responsible? For instance, who would be accountable if the agent did something dangerous, like turning the valve on an offshore installation in the wrong direction and, instead of lowering the pressure, increasing it? This situation serves as an example, and will not be further addressed. These are nevertheless difficult ethical questions, and they need practical answers.

4.2 System Overview

A simplified information flow diagram of the HMCI system is presented in Figure 4.1. From a system design perspective, the FSM (Finite State Machine) is the logical decision-making part of the system. It receives processed signals from a stereoscopic camera sensor and decides the system's state based on that. The camera sensor outputs both RGB and depth images. The former is sent to the Hand Detection module, which relies on MediaPipe's AI-powered hand tracking software. Next, the hand detection, represented by 3D points, is sent to the Hand Model module. Here the angles of each finger joint are calculated, and threshold values are used to decide on the current user-inputted hand gesture. These modules are further elaborated in Section 4.3.

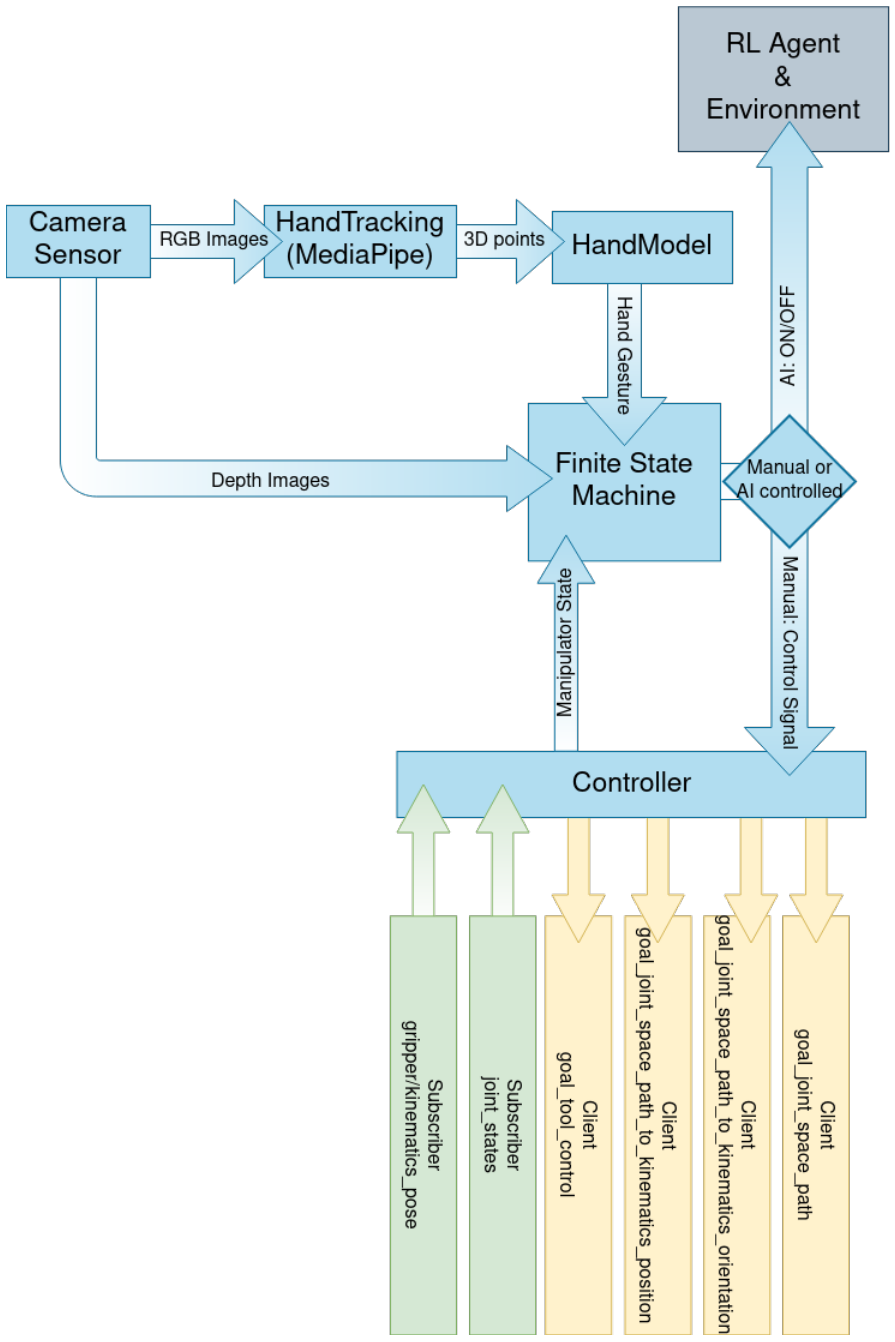


Figure 4.1: An overview of the Mimir system. Drawn with draw.io

The estimated hand gesture is sent to the FSM, which again decides on the state based on this information. The manipulator pose is used by the FSM to look for collisions with obstacles, which will stop the FSM from sending dangerous control signals to the Controller module. Meanwhile, the depth images are used to generate a control signal for the manipulator's Z-position during the *Move Height* gesture.

The Controller module receives control signals and function calls, which calls one of the four listed ROS clients. The module sends the end-effector pose and joint state values to the FSM by subscribing to the two given ROS topics. ROS clients and topics are described in Section 2.5.

Finally, the *Flip Hand* gesture sets the FSM in the AI-controlled state for as long as the hand gesture is detected. Therefore it is labeled as an ON/OFF signal in the figure. The RL agent and its environment are displayed as a black box in this diagram and will be further elaborated in Section 4.4. In short, the RL environment has a separate controller of its own, which talks to the OM-X through a similar set of topics and services as the Controller module shown in Figure 4.1. The control signals are reference angles and positions for motor joints, or reference poses for the end effector for both control modules.

The low-level control system is fully integrated into the OpenManipulator-X framework. It is a standard PID controller that utilizes both velocity and positional control. When the RL MuJoCo environment was developed, it was necessary to recreate the low-level control system for the modeled manipulator, as no OpenManipulator-X model existed for the MuJoCo physics engine.

4.3 Human Machine Interface

Listed below are the research questions from the author's project thesis of August - December 2021 [23].

- Can a hand tracking software powered by CNNs, such as the MediaPipe Hands framework, be applied to construct an intuitive HMI system based solely on hand movements and gestures?
- Can the HMI be used to perform any meaningful work with a robotic manipulator?
- How can XAI methods be used online to help users understand the system?

The HMI developed to answer these questions is introduced in this section. The following material is reused with adjustments, from the author's project thesis.

Figure 4.2 depicts the information flow and overall workings of the control system. The camera sensor is the red block in the upper right corner and records the operator workspace, outputting both RGB- and depth images.

The black box CNN is the MediaPipe Hands machine learning model, which receives the RGB image on its input, and yields estimated locations for 21 3D points where the third dimension is a synthetic height. The points are used to produce 2D and 3D skeleton hands, visualized in the operator panel. Meanwhile, the points are fed into a finite state machine, which also receives the depth image and tunable parameters from the operator panel. The information is scrutinized to determine the system state, which the controller uses to activate the correct proportional velocity controller. The controller communicates with the manipulator through the ROS 2 framework. Further, a feedback loop is incorporated into the system by the presence of a human operator. The user sees the movement of the robotic manipulator and the interpretations of their hand gesture in the operator panel. Based on this information, the user provides the system with new gestures to achieve the intended manipulator movement.

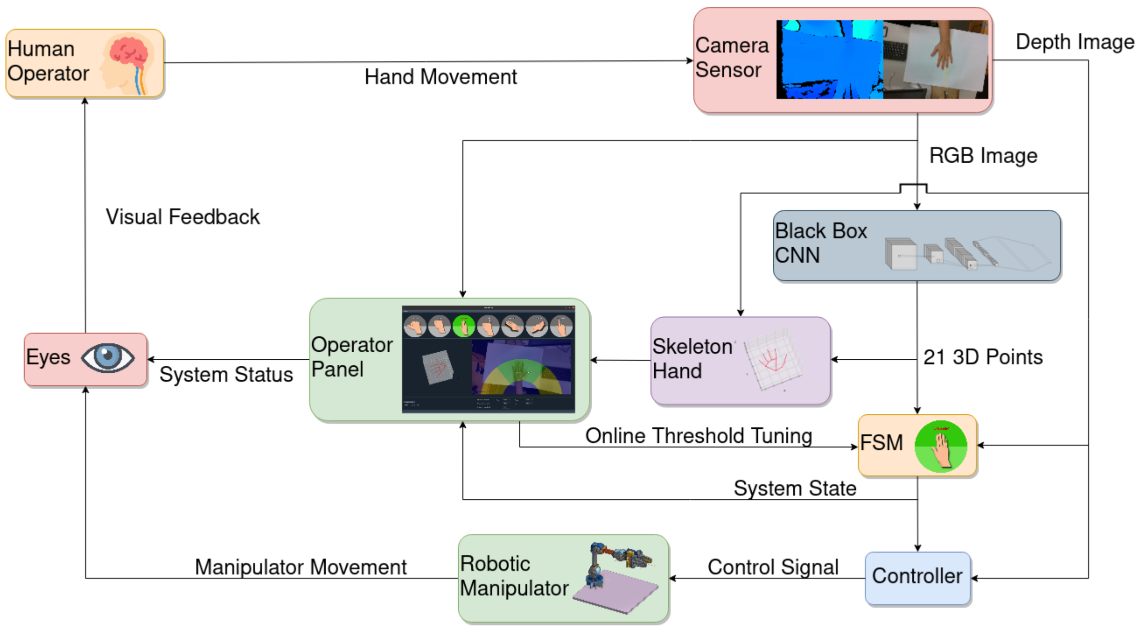


Figure 4.2: Information flow, and system overview

The FSM has exactly one Controller object, one **HandTracking** object, and one **HandModel** object. There is no limit for how many **Obstacle** objects it can have, but the implementation discussed in this thesis has five obstacles. The **Controller** module communicates with the robotic manipulator through ROS with two topics and four services. The **HandTracking** module, which produces the 21 3D landmarks, has one **CameraStream** object, which communicates with the depth sensor. The **HandModel** module estimates current gestures based on the landmarks from the **HandTrack-**

ing module. In the project thesis, a class diagram with further information on these modules was presented [23], it is available in Appendix A.2.

4.3.1 Finite State Machine

The states of the FSM implemented in this thesis are visualized in Figure 4.3. The diagram is a simplified state diagram, where transitions between the major states are not drawn for readability. The removed transitions are, however, easily described by an extra transition within *STOP*. The text along the arrows represents the input necessary to trigger the transition. Should these signals no longer be active, the state will transition to the *STOP* state. The signals are generated in the HandModel class, Table 4.1 displays all different signals.

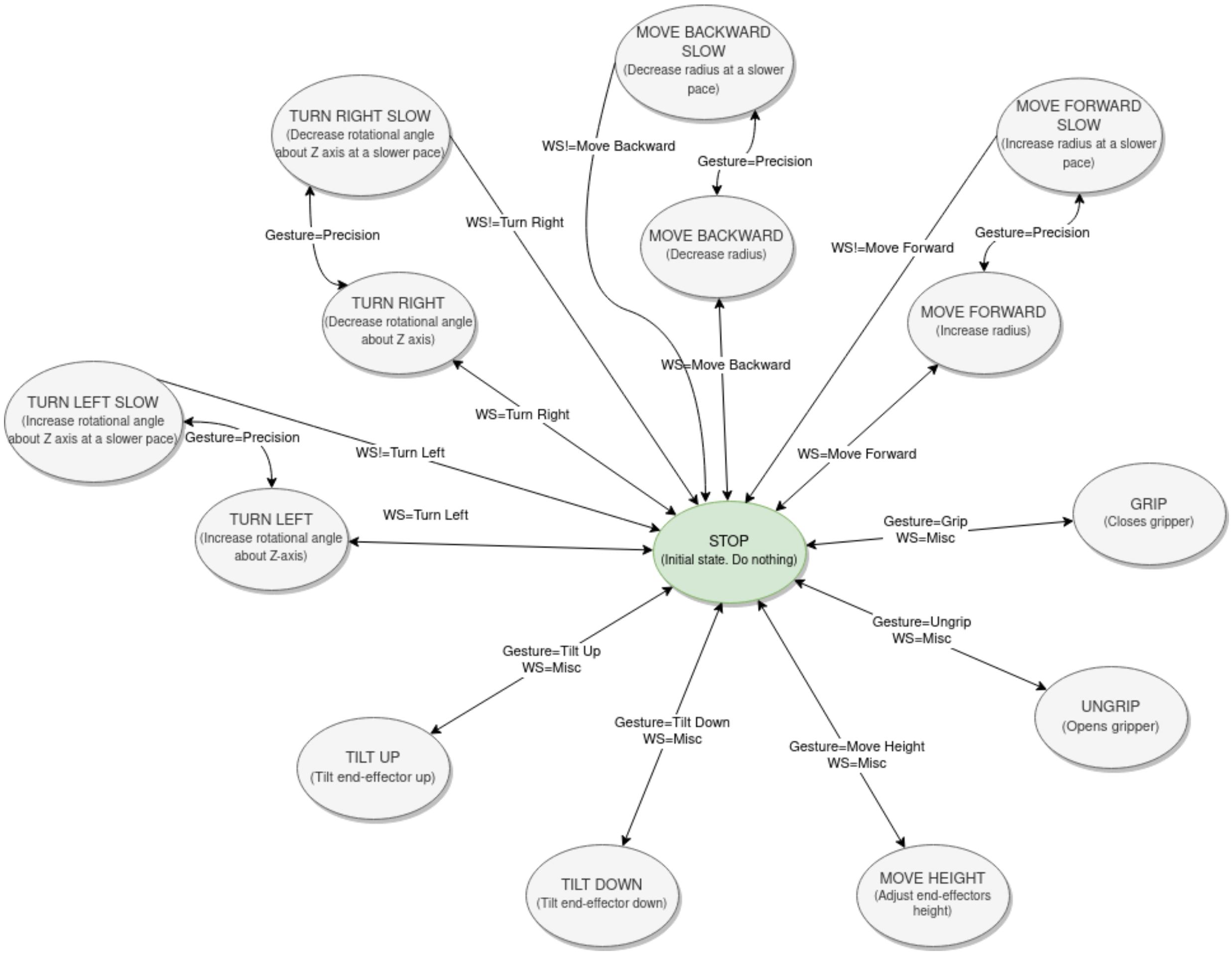

MOVE BACKWARD SLOW (Decrease radius at a slower pace)
TURN RIGHT SLOW (Decrease rotational angle about Z axis at a slower pace)
MOVE FORWARD SLOW (Increase radius at a slower pace)
WS!=Move Backward
Gesture=Precision
MOVE BACKWARD (Decrease radius)
WS!=Turn Right
Gesture=Precision
WS!=Move Forward
Gesture=Precision
TURN RIGHT (Decrease rotational angle about Z axis)
MOVE FORWARD (Increase radius)
WS=Move Backward
TURN LEFT SLOW (Increase rotational angle about Z axis at a slower pace)
Gesture=Precision
WS=Turn Right
WS!=Turn Left
WS=Move Forward
TURN LEFT (Increase rotational angle about Z-axis)
WS=Turn Left
STOP (Initial state. Do nothing)
Gesture=Grip WS=Misc
GRIP (Closes gripper)
Gesture=Ungrip WS=Misc
UNGRIP (Opens gripper)
Gesture=Tilt Up WS=Misc
TILT UP (Tilt end-effector up)
Gesture=Tilt Down WS=Misc
Gesture=Move Height WS=Misc
TILT DOWN (Tilt end-effector down)
MOVE HEIGHT (Adjust end-effectors height)

Figure 4.3: FSM diagram depicting all system states and transitions. WS is short for WorkSpace location, see Table 4.1

Hand gestures	Workspace locations
Stop	Turn Left
Grip	Turn Right
Ungrip	Move Forward
Precision	Move Backward
Tilt Up	Misc
Tilt Down	Hand Not Present
Move Height	

Table 4.1: The input signals determining the state of the FSM

4.3.2 Hand Tracking

MediaPipe

MediaPipe [56] is an open source ML library provided by Google, where the Hand Tracking API is the part we are interested in. Researchers at Google have trained a neural network to recognize human hands, and estimate the position of joints from a single image frame. The software detects initial hand locations, with the ability to recognize occluded hands in various sizes and environments. Next, a hand landmark model localizes 21 3D coordinates inside the detected hand via regression. According to the MediaPipe team, the model "learns a consistent internal hand pose representation and is robust even to partially visible hands and self-occlusions."

$$\boldsymbol{X}_k^W(t) = \begin{bmatrix} x_k^W(t) \\ y_k^W(t) \\ z_k^W(t) \end{bmatrix}, k \in [0, 20] \text{ and } t > 0 \tag{4.1}$$

The joint positions are outputted as 21 3D landmarks on the form given by eq. (4.1). The notation for the points is as follows: A hand point in coordinate system of joint $(i-1)$, with an index $k \in [0, 20]$ is written as $\boldsymbol{X}_k^{(i-1)}$. x_k^W and y_k^W are normalized pixel coordinates. z_k^W is a representation of the depth of the landmarks, where $z_0^W = 0$ at the location of the wrist. Note that the information given by z_k^W only describes the estimated depth of the joints in relation to each other. Further, since the depth information is artificially constructed by a machine learning model, it is referred to as synthetic information. Figure 4.4 illustrates how the landmarks are structured.

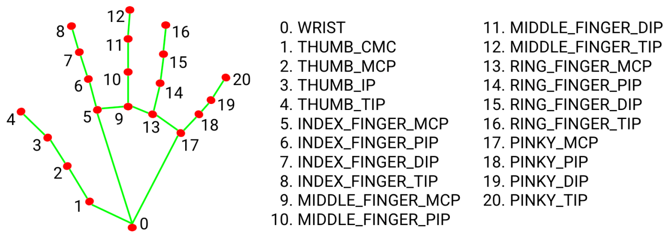


Figure 4.4: How the hand landmarks from MediaPipe are structured. Image borrowed from [56]

According to [57], the machine learning pipeline is a two step Convolutional Neural Network consisting of a single-shot detector, followed by a regression model. The input can either be a single image frame, or a video stream, while the outputs are: "21 3-dimensional screen landmarks", "A float scalar represents the handedness probability of the predicted hand", "21 3-dimensional metric scale world landmarks." Note that for both sets of predicted 3D points, the z-screen value and z coordinate, are provided by synthetic data based on the GHUM hand model ([58]). Due to the synthetic nature of the depth information outputted from the model, both the synthetic- and measured depth from a stereoscopic depth sensor can be used.

The detector model detects the palm location(s) in the input data. A crop of the input data, containing the hand, is then used as input to the regression model, that outputs a hand skeleton as the 3D points discussed above.

The authors have done extensive evaluations on the method, where the hand tracking algorithm was tested on 14 different groups of peoples from around the world, divided into groups based on the United Nations geoscheme. The results from this test yielded no error pattern with respect to regions, but showed that the error metric was smaller at the base of each finger, with larger values closer to the finger tips. Another test did not confirm any error pattern with respect to skin tone or gender.

Hand Gestures

Based on the hand tracking information, seven different gestures and six hand locations are defined in the operator workspace. These are the input signals listed in Table 4.1, which are used to transition between states in the FSM. To separate the gestures from one another, a coordinate system for the hand representation was designed and implemented in the HandModel module. A human hand consists of revolute joints, where the joints between fingers and palm are more complicated, rotating around two axes. Transformation matrices for each finger were designed, inspired by the convenient DH-convention. The landmarks from the hand tracker are given in workspace coordinates, Equation (4.1). We are, however, more interested in the relative angles between the finger links.

For every finger, each transformation is represented by an angle $\delta_i^{(i-1)}$, from joint $(i-1)$ to i in the $X_i^{(i-1)}Y_i^{(i-1)}$-plane, an angle $\gamma_i^{(i-1)}$ in the $X_i^{(i-1)}Z_i^{(i-1)}$-plane, and a translation $t_i^{(i-1)}$ from joint $(i-1)$ to i.

$$\boldsymbol{H}_{\boldsymbol{0}}^{\boldsymbol{W}} = \begin{bmatrix} \boldsymbol{R}(\delta_0^W) & \boldsymbol{t}_{\boldsymbol{0}}^{\boldsymbol{W}} \\ \boldsymbol{0} & 1 \end{bmatrix}, \boldsymbol{t}_{\boldsymbol{0}}^{\boldsymbol{W}} = \begin{bmatrix} x_0^W \\ y_0^W \\ 0 \end{bmatrix} \tag{4.2}$$

The first transformation matrix, from workspace coordinates W to point 0 coordinates (from Figure 4.4) is given by Equation (4.2), and is the same for all fingers. The general transformation matrices from joint $(i-1)$ to i, are on the form Equation (4.3).

$$\boldsymbol{H}_{\boldsymbol{i}}^{\boldsymbol{(i-1)}} = \begin{bmatrix} \boldsymbol{R}(\delta_i^{(i-1)})\boldsymbol{R}(\gamma_i^{(i-1)}) & \boldsymbol{t}_{\boldsymbol{i}}^{\boldsymbol{(i-1)}} \\ \boldsymbol{0} & 1 \end{bmatrix}, \boldsymbol{t}_{\boldsymbol{i}}^{\boldsymbol{(i-1)}} = \begin{bmatrix} x_k^{(i-1)} \\ y_k^{(i-1)} \\ z_k^{(i-1)} \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} x_8^7 \\ y_8^7 \\ z_8^7 \end{bmatrix} = \boldsymbol{H}_7^W \boldsymbol{X}_8^W = \boldsymbol{H}_7^6 \boldsymbol{H}_6^5 \boldsymbol{H}_5^0 \boldsymbol{H}_0^W \begin{bmatrix} x_8^W \\ y_8^W \\ z_8^W \end{bmatrix} \tag{4.4}$$

As an example, Equation (4.4) shows how the coordinates for the fingertip of the index finger, represented in the second to last joint, can be acquired. The angles are the essential results from this process, however, and are found iteratively by Equation (4.5) and Equation (4.6). To determine whether a finger is extended, we are interested in $\delta_i^{(i-1)}$ and $\gamma_i^{(i-1)}$ of the two outer joints for each finger.

$$\delta_i^{(i-1)} = arctan\left(\frac{y_i^{(i-1)}}{x_i^{(i-1)}}\right) \tag{4.5}$$

$$\gamma_i^{(i-1)} = -arctan\left(\frac{z_i^{(i-1)}}{\sqrt{(x_i^{(i-1)})^2 + (y_i^{(i-1)})^2}}\right) \tag{4.6}$$

By trial and error, threshold values were found to separate between an open and a closed finger. The threshold values were the same for all fingers and angular variables, except the thumb, which responded better to other thresholds. To check if the thumb is extended, $\delta_i^{(i-1)}$ for both joints must be greater than $-15°$. Meanwhile, for the remaining fingers, if either of $\delta_i^{(i-1)}$ and $\gamma_i^{(i-1)}$ for any of the outer two joints are larger than 25°, the finger is marked as closed.

Further, the mean position of landmarks $\boldsymbol{X}_0^W$, $\boldsymbol{X}_1^W$, $\boldsymbol{X}_5^W$, $\boldsymbol{X}_9^W$, $\boldsymbol{X}_{13}^W$ and $\boldsymbol{X}_{17}^W$ is used when deciding on where in the operator workspace the hand is located.

4.3.3 Controller

This section describes the Controller module. The Robotis OpenManipulator-X comes equipped with controllers dealing with motor control and inverse kinematics. The results are positional controllers for the end-effector, abstracted into ROS topics. Thus, the controllers presented in this section use these topics to control the manipulator to velocity references. The Controller object consists of four proportional velocity controllers, each with a default reference velocity of $0m/s$. The controllers set the velocity of the manipulator by requesting a new pose and a path time, which is set at $0.6s$.

One controller sets the reference velocity for the horizontal radius of the manipulator, and is activated in the *MOVE FORWARD*, *MOVE BACKWARD*, *MOVE FORWARD SLOW*, and *MOVE BACKWARD SLOW* states. For the former two, the speed is set at $8.33cm/s$, while the latter controllers operate at $1.67cm/s$. Another controller sets the reference velocity for the horizontal turning angle β, which is used by the states *TURN LEFT*, *TURN RIGHT*, *TURN LEFT SLOW*, and *TURN RIGHT SLOW*. As for the previous controller, the reference speed is set at $19.10°/s$ for the former states, and $3.82°/s$ for the latter two. A third controller is used for the second to last revolute joint, effectively adjusting the tilt angle of the end-effector. When active, the end-effector will tilt by $14.32°/s$. The last controller, responsible for setting the end-effector's velocity along the Z^0-axis, calculates the reference velocity based on the positional height of the operator's hand, using information from the depth sensor. The numerical values described here, were found through tuning of the proportional constants, $K_{p,\beta}, K_{p,r}, K_{p,Z^W_{EF}}, K_{p,\theta}$. β and r represents the position of the manipulator's end-effector in polar coordinates, while Z^W_{EF} is the end-effector's relative altitude. θ is the angle of tilt between the end-effector and the ground level X^wY^w-plane.

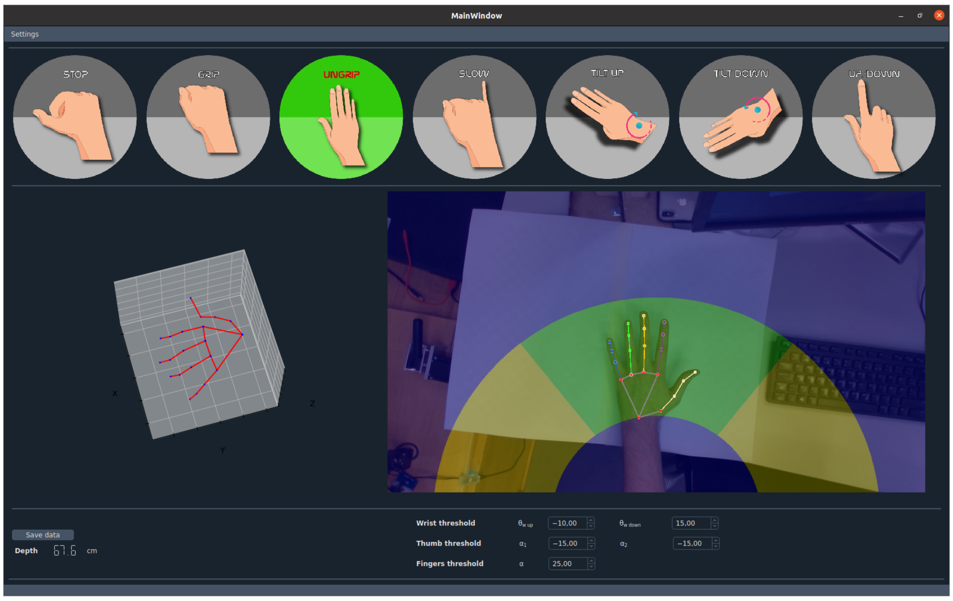
Meanwhile the end-effector, also known as gripper, is controlled in a binary fashion, and is either closed or opened. Unlike the previously discussed controllers, it has a positional reference, the distance, $d_{gripper}$. Possible values are of $-1cm$ to close the gripper, and $1cm$ to fully open it.

In Table 4.2 the states of the FSM, and their respected controllers are listed together with the velocity reference signals.

State	**Controlled variable**	**Reference**
MOVE FORWARD	r	$8.33 cm/s$
MOVE BACKWARD	r	$-8.33 cm/s$
MOVE FORWARD SLOW	r	$1.67 cm/s$
MOVE BACKWARD SLOW	r	$-1.67 cm/s$
TURN LEFT	β	$19.10°/s$
TURN RIGHT	β	$-19.10°/s$
TURN LEFT SLOW	β	$3.82°/s$
TURN RIGHT SLOW	β	$-3.82°/s$
TILT UP	θ	$14.32°/s$
TILT DOWN	θ	$-14.32°/s$
MOVE HEIGHT	Z_{EF}^{W}	Dynamic
GRIP	$d_{gripper}$	$-1.0 cm$
UNGRIP	$d_{gripper}$	$1.0 cm$

Table 4.2: The states of the FSM, the variables that are controlled when they are active, and the respective references.

4.3.4 Operator Panel



Depth 67.6 cm

Wrist threshold $\theta_{w\ up}$ −10,00 $\theta_{w\ down}$ 15,00
Thumb threshold α_1 −15,00 α_2 −15,00
Fingers threshold α 25,00

Figure 4.5: The operator panel in action, with the *Ungrip* gesture active.

The GUI was designed using the Qt 5 Designer and implemented with PyQt 5, a set of Python bindings for the Qt Company's Qt application framework [59]. The result can be seen in Figure 4.5. The GUI, or operator panel, consists of a video stream, known as the operator workspace, where a depth sensor is recording the operator's hand. Colored, curved boxes are drawn in the video frame to provide the user with a set of different commands by placing their hand inside these boxes. Further, a 2D hand skeleton is drawn on top of the detected hand, visualizing where the machine learning model estimates the position of the hand's joints.

Above the video stream, seven hand gestures are displayed on grey backgrounds. When the system selects a given hand gesture based on the information provided by the machine learning model, the gesture's background switches from grey to green, notifying the user how their hand gestures are being interpreted. This information is not needed to control the manipulator but certainly elevates the user's trust in the hand tracking model and overall system. The control panel also works as a user interface for new users unfamiliar with the different commands. For more experienced users and developers, a 3D hand skeleton and a metric of the measured distance to the hand palm from the stereoscopic sensor are provided. The additional information is toggled in the settings file.

Just below the operator workspace, three tunable parameters are accessible. The wrist thresholds are used to differentiate between *Tilt Up* and *Tilt Down*, where the angle between the wrist and fingertips must surpass the thresholds for the gestures to be detected. Similarly, the thumb thresholds are angles that the two outer thumb joints must surpass before the thumb is registered as extended. The finger threshold functions similarly to the thumb threshold, but it was experimentally determined that the same threshold worked for both outer joints for all remaining fingers.

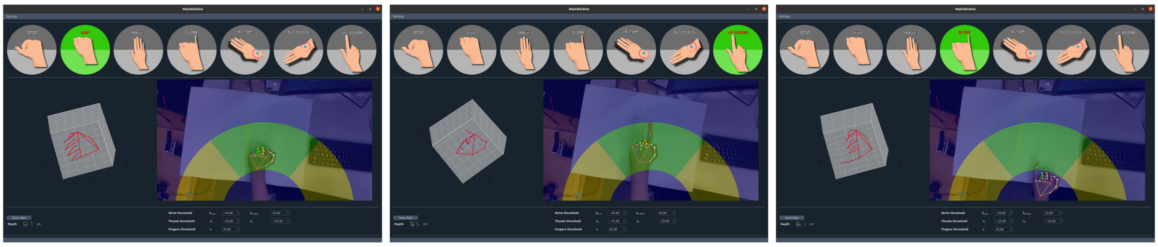


Figure 4.6: More examples of the operator panel in action

With the exception of *Slow* and *Stop* gestures, all commands using hand gestures are only in effect while the palm is located within the green section of the operator workspace. The green and yellow sections are shaped like half-circles, allowing the operator to keep his shoulder and elbow more or less stationary when performing rotations in a natural movement. The yellow areas control the manipulator's polar coordinate β, turning the robot arm left or right. Further, the blue sections control the manipulator's radius, r, where the upper blue area increases it, and the smaller region reduces it. Figure 4.6 displays a few user commands in action.

4.4 OpenManipulator RL Environments

To train an RL agent on the lever pull challenge, the *OpenManipulator RL Environments* package, developed by the author of [22] was extended and updated. In this section, the original package will be explained. Section 4.5 will elaborate on the extensions and changes made to adapt it for this thesis.

The software is designed as a ROS Kinetic package, since ROS is used for communicating with the OpenManipulator-X. It is structured hierarchically, with the three following main modules:

- Gym Environments
- Robot Environment
- Task Environments

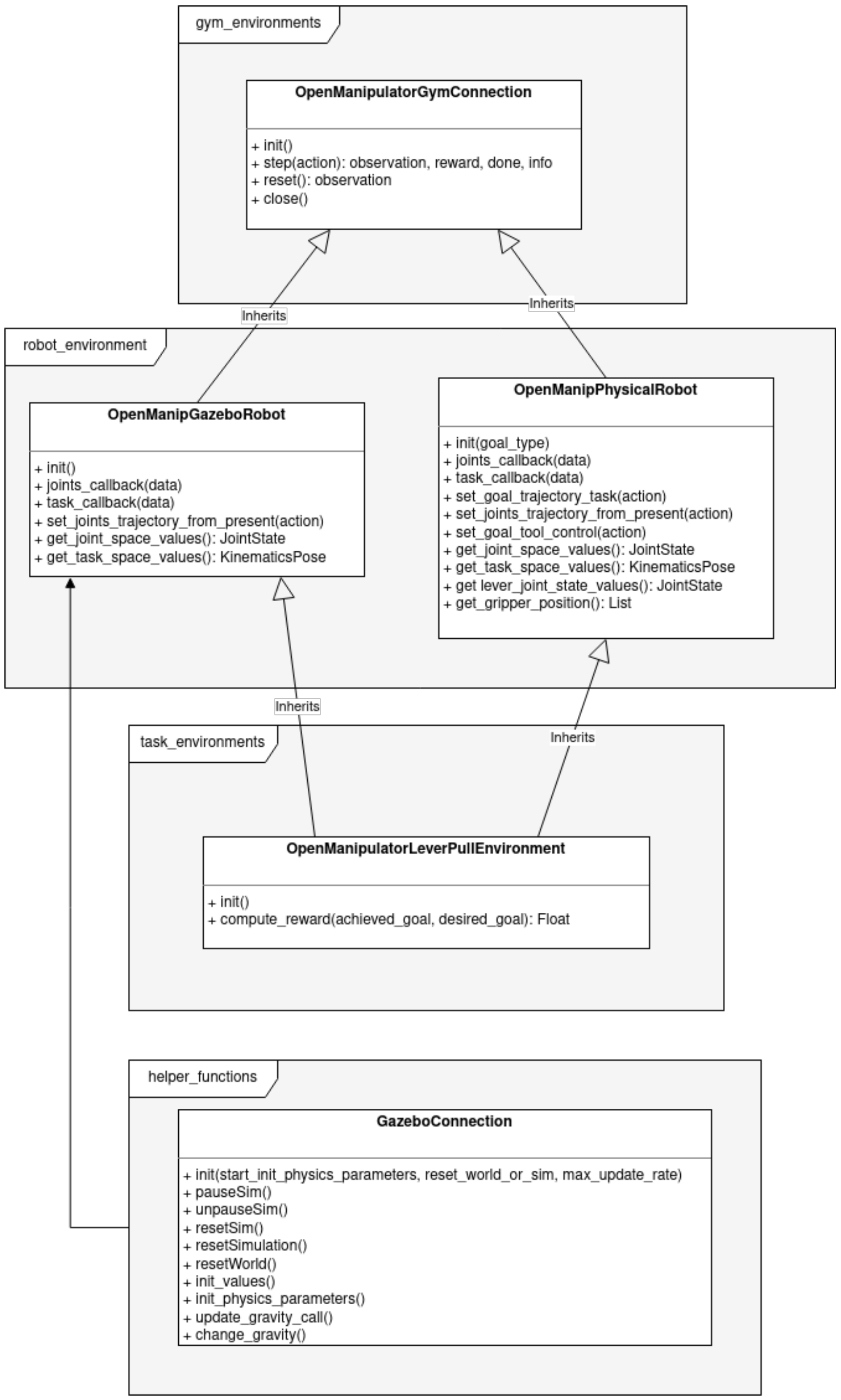

gym_environments
OpenManipulatorGymConnection
+ init()
+ step(action): observation, reward, done, info
+ reset(): observation
+ close()
Inherits
Inherits
robot_environment
OpenManipGazeboRobot
+ init()
+ joints_callback(data)
+ task_callback(data)
+ set_joints_trajectory_from_present(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
OpenManipPhysicalRobot
+ init(goal_type)
+ joints_callback(data)
+ task_callback(data)
+ set_goal_trajectory_task(action)
+ set_joints_trajectory_from_present(action)
+ set_goal_tool_control(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
+ get lever_joint_state_values(): JointState
+ get_gripper_position(): List
Inherits
Inherits
task_environments
OpenManipulatorLeverPullEnvironment
+ init()
+ compute_reward(achieved_goal, desired_goal): Float
helper_functions
GazeboConnection
+ init(start_init_physics_parameters, reset_world_or_sim, max_update_rate)
+ pauseSim()
+ unpauseSim()
+ resetSim()
+ resetSimulation()
+ resetWorld()
+ init_values()
+ init_physics_parameters()
+ update_gravity_call()
+ change_gravity()

Figure 4.7: Class diagram of the *Open Manipulator RL Environments* package

Not all classes from the environments are mentioned here, as they are not relevant to this project. Figure 4.7 provides a class diagram of all pertinent modules that will later be extended to encompass the needed functionality for this thesis. All public methods are listed, and all private methods have been excluded for readability.

OpenManipulatorGymConnection inherits from the gym.Env class to make it compatible with OpenAI's Gym framework, which was briefly reviewed in Section 3.4.3. The module defines the important step() and reset() methods. They are used by RL agents to interact with the environment. Next, **OpenManipGazeboRobot** and **OpenManipPhysicalRobot**, from the Robot Environment module, inherits from **OpenManipGymConnection**. For both classes, the methods joints_callback() and task_callback() update the manipulator pose and joint values by listening to ROS topics defined by the official OpenManipulator Controller provided by Robotis. A set of ROS services are used for maneuvering the manipulator to requested positions. The Gazebo class has another service for spawning the lever into the simulation. The simulated model and the physical manipulator have the same topics and services, making these classes very similar. **OpenManipPhysicalRobot** has a few more methods used to control the manipulator to starting positions. These were not needed for the Gazebo case, as the simulator can turn off physics and instantaneously move the manipulator to the desired location and pose, which is why the **OpenManipGazeboRobot** has a relation with the **GazeboConnection** from the helper functions module.

In the Task Environments module, **OpenManipulatorLeverPullEnvironment** sets up the action- and observation spaces used by the gym environment to define inputs and outputs to the machine learning model. It establishes the reward function, accessed through the compute_reward() method. Further, it handles setting the lever angle goal and determining when the task is finished. The observation outputted by the step() and reset() methods from the **OpenManipulatorGymConnection** is computed in this module.

By using these modules to set up an RL learning environment, one can train a model in a simulated environment before applying the very same model with only minor adjustments to the physical setup. This package was, however, written in Python 2 for ROS Kinetic on an Ubuntu 18.04 OS, and thus incompatible with the HMI from Section 4.3 - which was written in Python 3 for ROS 2 Foxy on an Ubuntu 20.04 OS. Thus work was put into porting the *OpenManipulator RL Environments* package over to ROS Noetic.

4.5 Human Machine Cooperation Interface

The complete system can be considered an HMCI (Human Machine Cooperation Interface) since it is a graphical interface that allows for greater cooperation between a human operator and a machine - in this case, the OpenManipulator-X.

4.5.1 Porting the Code

The graphical interface is based on the HMI from the author's project thesis [23], while the machine learning part builds on the *OpenManipulator RL Environments* package by the author of [22]. As mentioned in the last section, the HMI was built with Python 3 for ROS 2 Foxy, which is compatible with Ubuntu 20.04. Meanwhile, the RL environments were designed for ROS Kinetic with Python 2, which are incompatible with Ubuntu 20.04. Thus it was decided to convert the RL Environments to ROS Noetic and Python 3, as it would have taken too much time to convert it over to ROS 2. The HMI, being best known to the author, was ported from ROS 2 to ROS Noetic.

Porting the RL Environments
Python 3 is not backward compatible with Python 2. Thus, much of the Python 2 code from the RL package had to be rewritten and adequately tested. The Robotis supplied OpenManipulator-X packages renamed its topics and services for ROS Noetic, thus all service calls and subscribers in the RL environments needed to be renamed.

Porting the HMI
ROS 2 was created for several reasons. New use cases that were not apparent back in 2007 when work first began on ROS have since then appeared. For example, in ROS 1, there is no standard approach to having

multiple robots communicate with each other. Further, ROS 1 device driver dependencies do not allow microchips to communicate over ROS. See [60] for more examples. To overcome these challenges, ROS 2 does not use the TCPROS protocol employed in ROS 1 but instead builds its communications on top of the already existing middleware interface DDS, leveraging existing and well-developed standards. A few more differences between ROS 1 and ROS 2 are listed in Table 4.3.

	ROS 1	**ROS 2**
Python	ROS 1 is targeting Python 2	Supports Python 3.5 and higher
Communications	ROS 1 uses the custom TCPROS protocol	ROS 2 uses a middleware interface on top of the well developed DDS solution
Build system	catkin (CMake)	ament (CMake or Python)
Python packages	Restricted set of features from the setup.py file	Python packages can use anything in the setup.py file
Python client library	rospy	rclpy

Table 4.3: Some practical differences between ROS 1 and ROS 2, see [61] and [62]

Thus, to port the HMI from ROS 2 Foxy to ROS Noetic, more extensive changes were going to be made, compared to porting between two ROS 1 versions. The module that needed the most significant changes was the Communication module, which handles topics and service calls. Previously, in ROS 2, the Python client library *rclpy* was used for these purposes, but it is not available for ROS 1. *rospy* is a similar library meant for ROS 1 versions, thus the Communication module was properly ported by adapting it for *rospy*. Finally, the code was restructured into a ROS Noetic package, as the HMI was not previously fully integrated with ROS.

4.5.2 Extensions to the RL Environments

The most notable new feature added to the *OpenManipulator RL Environments* package was support for the MuJoCo simulator environment, presented in Section 3.4.2. MuJoCo was used instead of PyBullet, as the PyBullet environment from [22] would have to be ported to Python 3, which would likely take some time. Further, MuJoCo used to be licensed but became open-source and available for everyone in late 2021. It was chosen as the simulator in this thesis, as it runs faster, and is more stable, than Gazebo - allowing for faster training of the RL agent [48]. MuJoCo is not integrated with ROS to the same extent as Gazebo is, and the OpenManipulator-X does not officially support MuJoCo. The new modules that integrates MuJoCo into the codebase is illustrated in Figure 4.8.

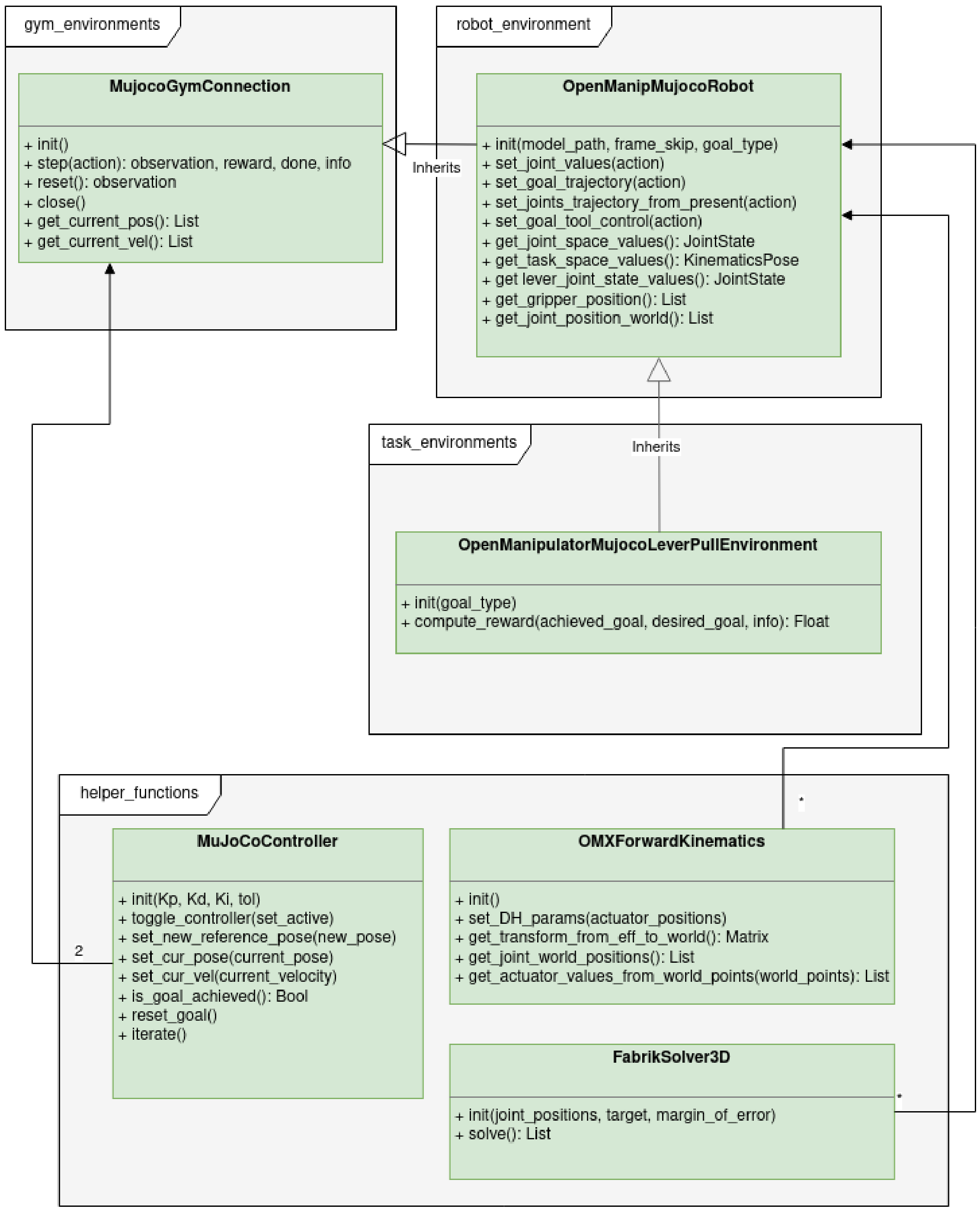

gym_environments
MujocoGymConnection
+ init()
+ step(action): observation, reward, done, info
+ reset(): observation
+ close()
+ get_current_pos(): List
+ get_current_vel(): List
robot_environment
OpenManipMujocoRobot
+ init(model_path, frame_skip, goal_type)
+ set_joint_values(action)
+ set_goal_trajectory(action)
+ set_joints_trajectory_from_present(action)
+ set_goal_tool_control(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
+ get lever_joint_state_values(): JointState
+ get_gripper_position(): List
+ get_joint_position_world(): List
Inherits
task_environments
Inherits
OpenManipulatorMujocoLeverPullEnvironment
+ init(goal_type)
+ compute_reward(achieved_goal, desired_goal, info): Float
helper_functions
MuJoCoController
+ init(Kp, Kd, Ki, tol)
+ toggle_controller(set_active)
+ set_new_reference_pose(new_pose)
+ set_cur_pose(current_pose)
+ set_cur_vel(current_velocity)
+ is_goal_achieved(): Bool
+ reset_goal()
+ iterate()
2
OMXForwardKinematics
+ init()
+ set_DH_params(actuator_positions)
+ get_transform_from_eff_to_world(): Matrix
+ get_joint_world_positions(): List
+ get_actuator_values_from_world_points(world_points): List
*
FabrikSolver3D
+ init(joint_positions, target, margin_of_error)
+ solve(): List
*

Figure 4.8: Simplified class diagram of the additions made to the *Open Manipulator RL Environments* package

The **MujocoGymConnection** handles the MuJoCo simulator environment and implements methods for getting the position and velocities of the angular joints, as well as the standard step() and reset() methods. No MuJoCo compatible model of the OpenManipulator-X had previously been made. In Gazebo, the model is represented by two XML files, one on the URDF format describing the parts of the robot and a DAE file for collisions and visuals. MuJoCo does not support either of the URDF or DAE files from Gazebo without some workarounds. The URDF file first needed to be converted to an MJCF file with the *compile* script from MuJoCo. Next, a few more additional MuJoCo tags were added to specify the actuators of the model, light and a surface plane. Friction in the lever joint also needed tuning to prevent it from succumbing to the simulated gravity. The DAE files were converted to STL files by using the open-source software MeshLab [63].

Further, the manipulator controller also needed to be recreated in MuJoCo. In Gazebo and the physical environment, the pre-built controller is accessed by sending reference values through ROS. The class **MuJoCoController** was designed to be iterative, and compute a new control signal, u, that would be summed with the previous actuator value, u_{prev}, and assigned as a new actuator value u_{act}, at each time step of the simulation, see Equation (4.7).

$$u_{act} = u_{prev} + u \tag{4.7}$$

The purpose of the control signal is to guide the joint to its reference angle with a tolerated error of 0.8°. Two instances of the class are used by **MujocoGymConnection**, where one controls the lever, and the other - being six-dimensional - controls the joints of the manipulator. The simulated manipulator has two actuators representing the physical end-effector actuator, giving rise to the sixth dimension.

OpenManipMujocoRobot encompasses methods for setting joint values, goal pose of end-effector, and manipulation of the lever angle. Further, the module has one object of the **FabrikSolver3D** class, which is necessary to calculate reference values for the controllers from the goal pose. The solver implements the algorithm discussed in Section 2.1.3. **OMX-ForwardKinematics** is used to calculate the spatial coordinates of each joint.

Finally, **OpenManipulatorMujocoLeverPullEnvironment** was created as a new module instead of adding the new features to the **OpenManipulatorLeverPullEnvironment** due to implementation differences between simulated and physical measurements of the lever angle, and MuJoCo controller.

4.5.3 Extensions to the HMI

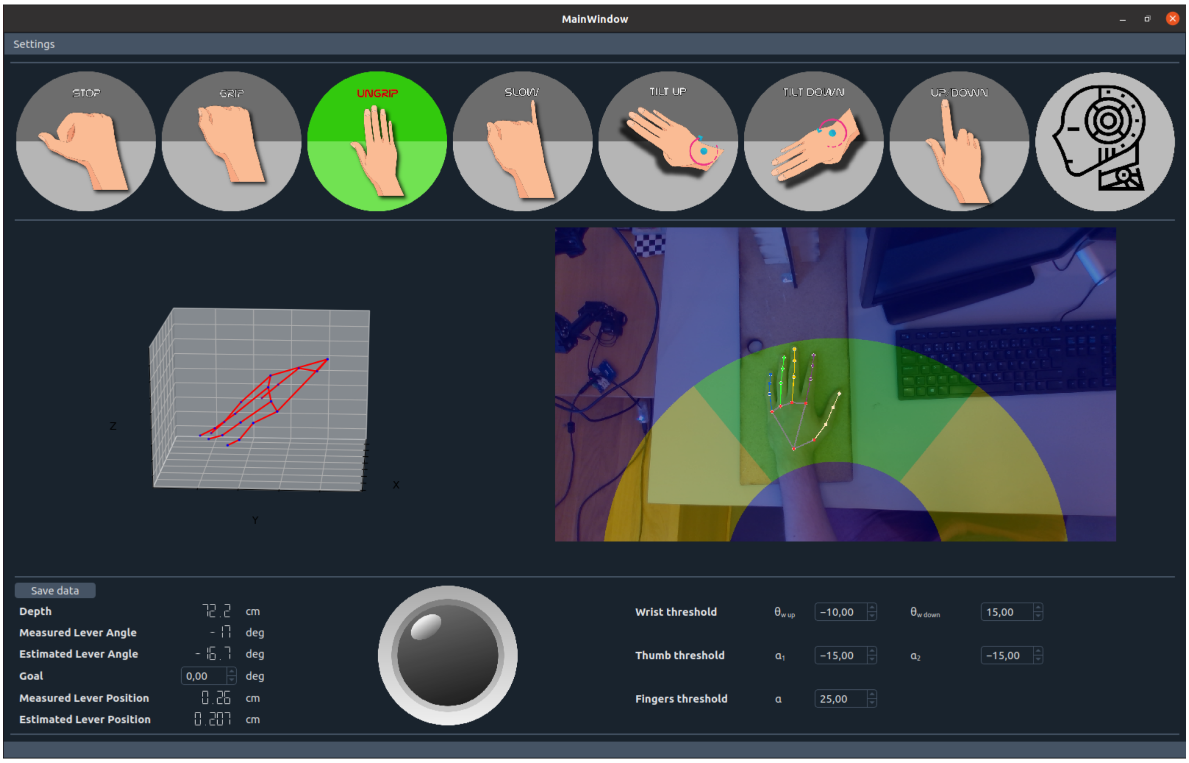
Table 4.4 displays all valid hand gestures, and lists the new hand gesture *Flip Hand*, meaning the operator's palm is facing down. Table 4.5 shows the states of the system, now extended to include the *AI Control* state, which is activated by the *Flip Hand* gesture. During the state, the four last joints of the manipulator are controlled by an RL agent, which can send control signals in the range specified. The agent stops sending control signals should the operator's hand gesture change to something else. Should the hand gesture return, the agent will continue controlling the manipulator.

Hand gestures	**Workspace locations**
Stop	Turn Left
Grip	Turn Right
Ungrip	Move Forward
Precision	Move Backward
Tilt Up	Misc
Tilt Down	Hand Not Present
Move Height	
Flip Hand	

Table 4.4: The input signals determining the state of the FSM

State	**Controlled variable**	**Reference**
MOVE FORWARD	r	$8.33cm/s$
MOVE BACKWARD	r	$-8.33cm/s$
MOVE FORWARD SLOW	r	$1.67cm/s$
MOVE BACKWARD SLOW	r	$-1.67cm/s$
TURN LEFT	β	$19.10°/s$
TURN RIGHT	β	$-19.10°/s$
TURN LEFT SLOW	β	$3.82°/s$
TURN RIGHT SLOW	β	$-3.82°/s$
TILT UP	θ	$14.32°/s$
TILT DOWN	θ	$-14.32°/s$
MOVE HEIGHT	Z_{EF}^{W}	Dynamic
GRIP	$d_{gripper}$	$-1.0cm$
UNGRIP	$d_{gripper}$	$1.0cm$
AI Control	joint states	$\in [-1, 1]$

Table 4.5: The states of the FSM, the variables that are controlled when they are active, and the respective references.



Depth	72.2	cm
Measured Lever Angle	-17	deg
Estimated Lever Angle	-16.7	deg
Goal	0,00	deg
Measured Lever Position	0.26	cm
Estimated Lever Position	0.207	cm

Wrist threshold	$\theta_{w\,up}$	−10,00	$\theta_{w\,down}$	15,00
Thumb threshold	α_1	−15,00	α_2	−15,00
Fingers threshold	α	25,00		

Figure 4.9: The updated operator panel

The new operator panel can be seen in Figure 4.9. New features include lever angle measurements and an option for setting the goal position of the lever angle. A bulb icon was added to indicate when the goal angle has been reached by lighting up.

4.5.4 System Execution

To get the system running one would first need to run *open_manipulator_controller.launch*, then *lever_angle_pose.launch*, *open_manipulator_lever_pull_load_params.launch*, and finally start *mimir.launch*, in four different terminals. The first script starts the manipulator controller, while the second starts publishing measured and estimated lever poses to the topic *mimir/lever_angle_pose*. The third script loads parameters describing the environment for the RL agent. The final script starts the HMCI system.

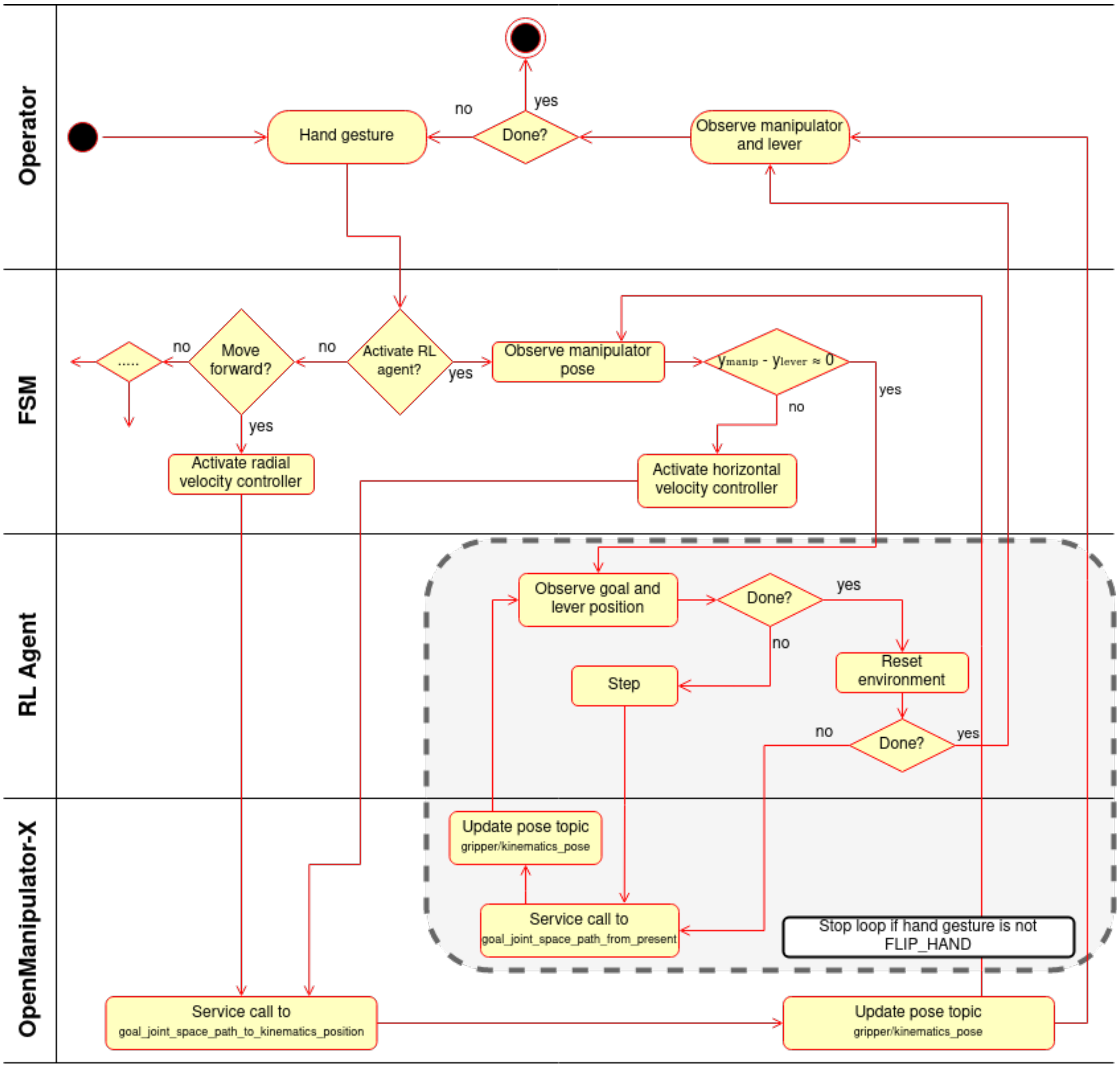

Operator
FSM
RL Agent
OpenManipulator-X
Hand gesture
Done?
no
yes
Observe manipulator and lever
Move forward?
Activate RL agent?
Observe manipulator pose
y_{manip} - y_{lever} ≈ 0
.....
Activate radial velocity controller
Activate horizontal velocity controller
Observe goal and lever position
Step
Reset environment
Update pose topic gripper/kinematics_pose
Service call to goal_joint_space_path_from_present
Stop loop if hand gesture is not FLIP_HAND
Service call to goal_joint_space_path_to_kinematics_position

Figure 4.10: A UML activity diagram of the complete system developed in this thesis

The diagram in Figure 4.10 illustrates how the system works, using one of the manual control states, and the special AI Control state as examples. The diagram divides the system into four different actors, which does a good job of explaining the system logic. For a deeper insight into the system, the reader can study the class diagrams in Appendix A.2, or visit the GitHub repositories [64]

The operator initiates the program and controls the system states by performing hand gestures. A set of pre-determined velocity control signals, see Table 4.5, are sent to the manipulator through service calls upon activating corresponding hand gestures. When the operator starts the RL agent, the manipulator is automatically steered such that the manipulator points in the direction of the lever. Once in position, the lever pose is evaluated, and the agent decides on control signals for the joints in the *Step* activity. These controls are sent to the manipulator through a service call, and the lever pose is evaluated once more. This loop continues until the goal is reached, or the operator halts the agent by switching to a different hand gesture. When the goal has been reached, the agent resets the manipulator and moves it safely away from the lever without accidentally adjusting the lever position.

Chapter 5

Results and Discussion

The first section of this chapter presents the results from the training of the RL agent in the simulated environment. Next, the trained agent is integrated with the HMCI, and the system is tested on the manipulator-lever setup, at two difficulty levels.

5.1 Training the RL Agent

Figure 5.1 displays the simulated environment during training. Training takes far less time when the MuJoCo window is not rendered, so the author had to rely on statistics printed in the terminal to evaluate ongoing training sessions. The images are from an evaluation of a trained model.

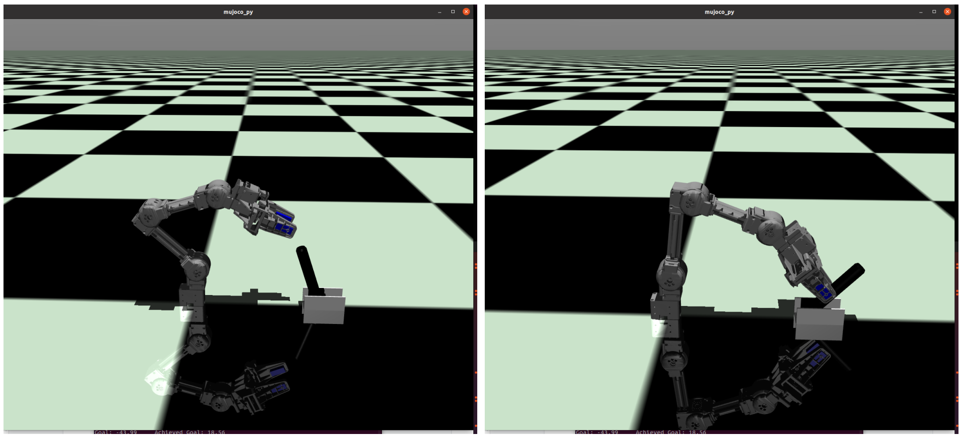


Figure 5.1: The OpenManipulator-X in the MuJoCo environment during training. Left: Manipulator approaches the lever. Right: The agent is successfully placing the lever in its goal position.

The agent was first trained for 50 epochs with a 50% chance of grabbing the lever in the initial position and diminishing action noise, as seen in Figure 5.2. During the collaboration challenge, however, the RL agent will never take control of the manipulator when it is already grabbing the lever. With this intuition, the agent was trained for another 50 epochs with a 0% grip chance. The results from the training are displayed in Figure 5.3. A sparse reward function was used for this training session as well. Figure 5.4 plots the statistics from a training session using dense rewards.

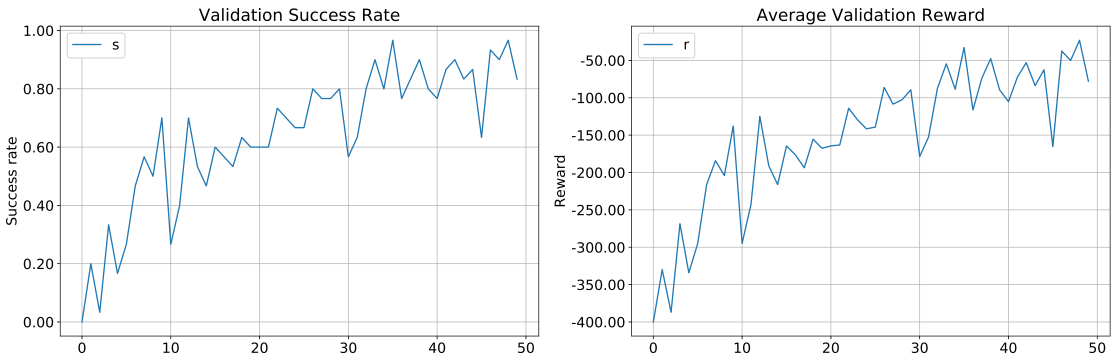


Figure 5.2: Model trained on a sparse reward. Left: Success rate. Right: Reward

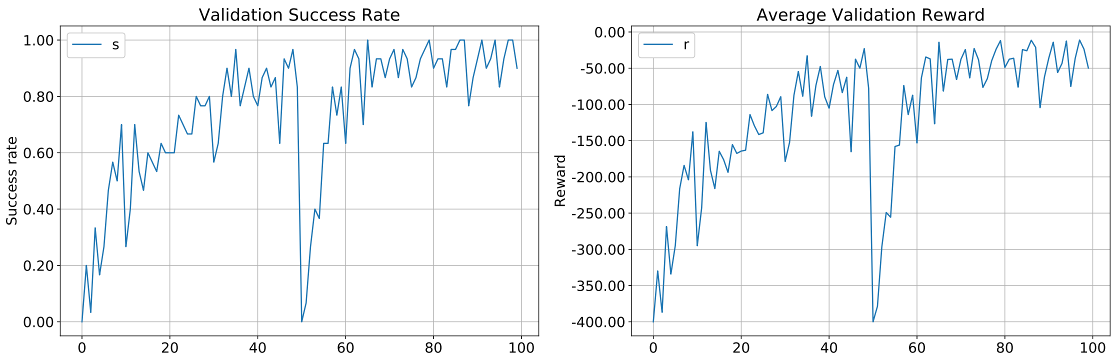


Figure 5.3: Model trained on a sparse reward with 50% grip chance, then further trained with a 0% grip chance. Left: Success rate. Right: Reward

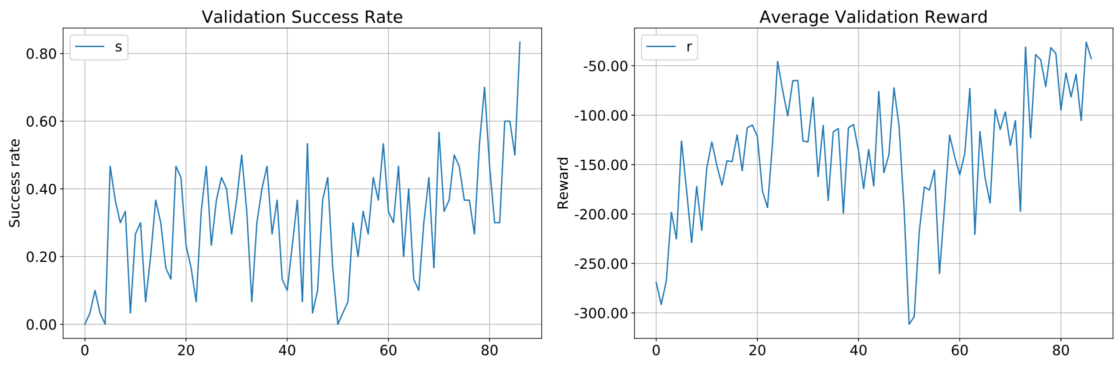


Figure 5.4: Model trained on a dense reward. Left: Success rate. Right: Reward

5.2 Cooperation Challenge

In the cooperation challenge, the system developed in this thesis is tested. A human operator takes control of the manipulator through the HMI system and controls it to arbitrary positions, demonstrating that it works. Next, the operator sets a goal angle for the lever through the operator panel before performing the *Flip Hand* gesture and setting the FSM in the *AI Control* state. Ideally, the RL agent should at this point take control over the manipulator and pull the lever into the goal angle position. Should anything go wrong, the operator can always switch hand signal, stopping the RL agent instantly.

5.2.1 Measured Lever

In this scenario, the measured lever pose is used as input to the RL agent. The position of the lever base was measured using a ruler, while the angle measurements were produced by the potentiometer setup discussed in Section 3.6.1. The reader is advised to study Figure 5.7 before continuing to read. The operator first controlled the manipulator to tilt downwards and lower its height by about $8cm$. Next, the manipulator turned left 77° and reduced its elevation by another $3cm$. The operator typed a goal angle of -45° into the operator panel before setting the FSM in the *AI Control* state.

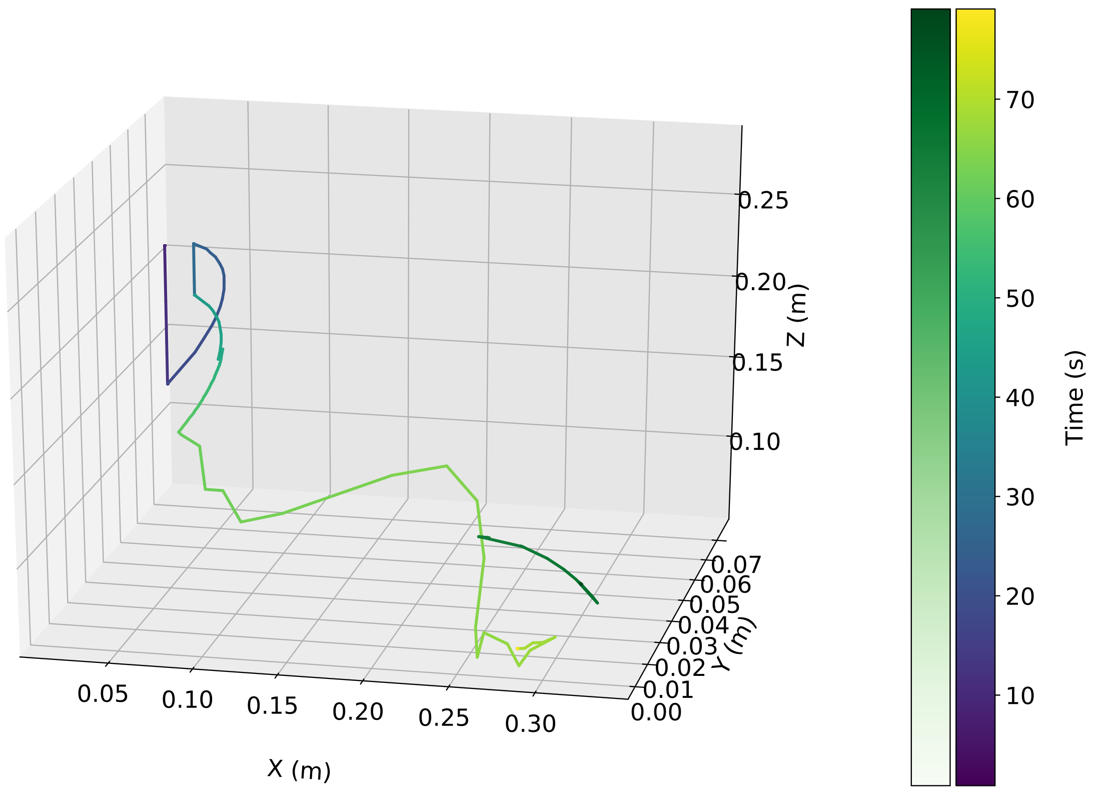


Figure 5.5: The path of the manipulator's end-effector plotted together with the path of the uppermost point of the lever (green). The paths are represented as gradients with respect to time.

The path of the end-effector during the test is shown in Figure 5.5 as a line that transitions from purple into yellow. Yellow being the position it had at the end of the test. Meanwhile, the path of the uppermost point on the lever is drawn in the same plot and can be seen as a green curve. The two polar plots in Figure 5.6 visualize the planar movement of the end effector in the $X^W Y^W$-plane and the planar movement of the lever in the $X^W Z^W$-plane. The goal angle is $-45°$, and the success threshold of $1.4°$ has been drawn as two dotted lines. Both the lever and end-effector have been colored in gradients related to the relative time, and the lever's end position can be seen as a yellow line.

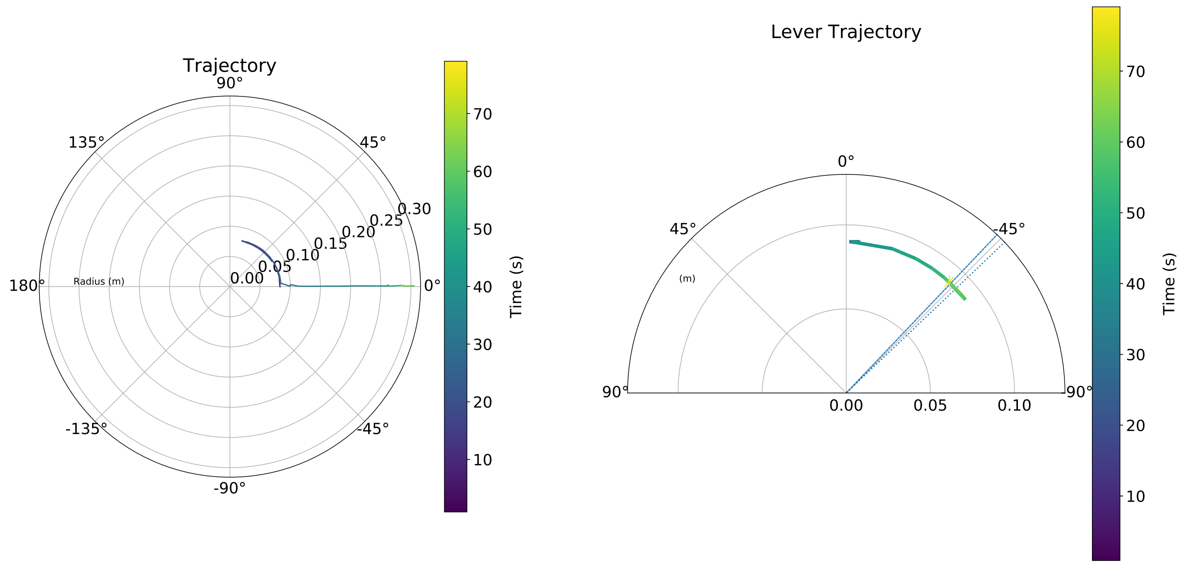


Figure 5.6: Left: Path of the end-effector in the $X^W Y^W$-plane, represented with polar coordinates. Right: Path of the lever in the $X^W Z^W$-plane, represented with polar coordinates. The threshold is plotted around the goal angle as two dotted lines.

Further, Figure 5.7 display the end-effector's pose through time.The uppermost plot presents the height or positional value on the Z^W-axis, Z_e. The following two plots give the position in polar coordinates, r, β, while the next plot represents the end-effector's orientation in Euler angles, ψ, θ, ϕ. The last one is a step plot, showing the transitions between the states in the FSM that occurred during the test.

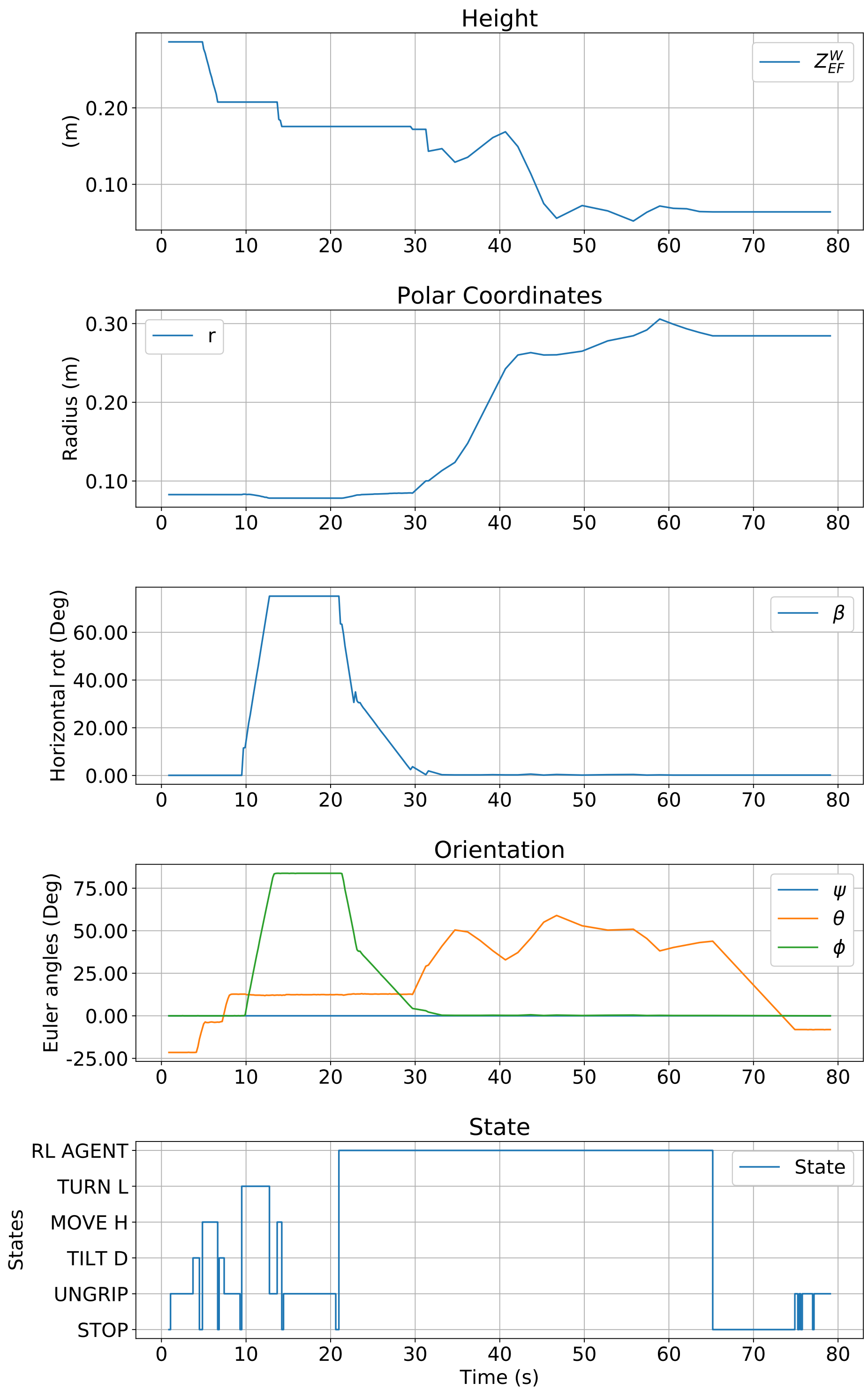


Figure 5.7: End-effector pose and active FSM state over time.

Finally, Figure 5.8 exhibit plots related to the lever and RL agent. In the first plot, the visually estimated angle, ω_e, is plotted next to the potentiometer read angle, ω_m. For the next three plots, the estimated and measured lever position on X^W-axis, $l_{x,e}$, $l_{x,m}$, Y^W-axis, $l_{y,e}$, $l_{y,m}$, and Z^W-axis, $l_{z,e}$, $l_{z,m}$, are given. Next, the cumulative reward is plotted, followed by the binary Success plot.

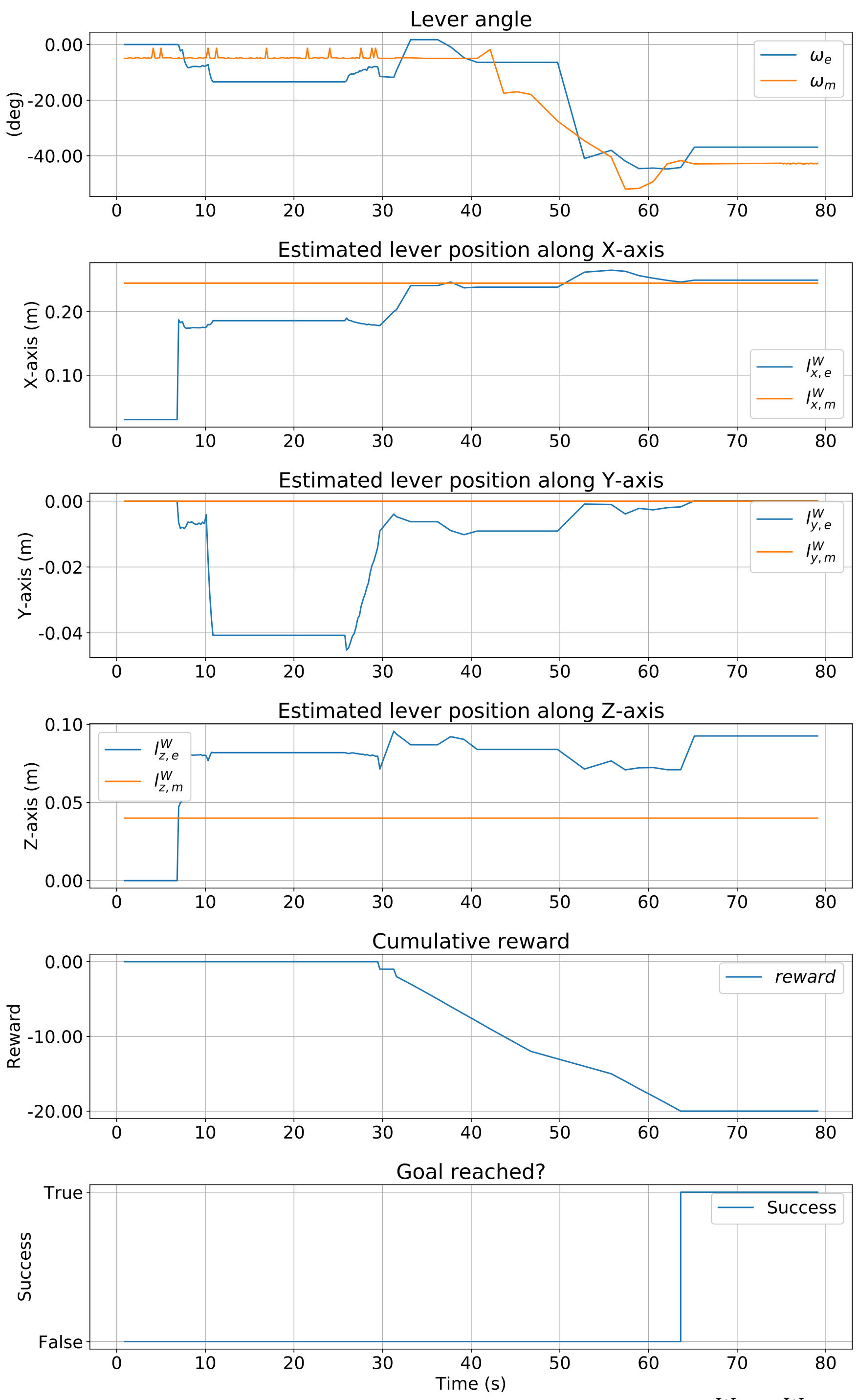


Figure 5.8: From top: Estimated and measured lever angle, X^W-, Y^W-, and Z^W-positions measured and estimated, cumulative reward, and success.

5.2.2 Visually Estimated Lever

In this scenario, a visually estimated lever pose is used as input to the RL agent. The angle and position are estimated based on readings from the detected Aruco corners and generated homogeneous transformation matrix, as described in Section 3.6.2.

Again, the reader is advised to study Figure 5.11 before reading further. The test started with the human operator in charge of the manipulator. First, the end-effector was tilted and moved downwards until all three Aruco indices appeared in the camera view, and the lever pose could be estimated. Next, the manipulator was turned left by approximately $55°$ and moved forward by roughly $18cm$. Afterward, the operator moved the manipulator down by $8cm$ and reduced its radius by around $8cm$. After this demonstration of manual control, the operator typed in a goal angle of $-25°$, activated the RL agent, and observed the actions of the AI.

The results from the test can be seen in the figures: Figure 5.9, Figure 5.10, Figure 5.11, Figure 5.12, which are the same type of plots described in the previous section.

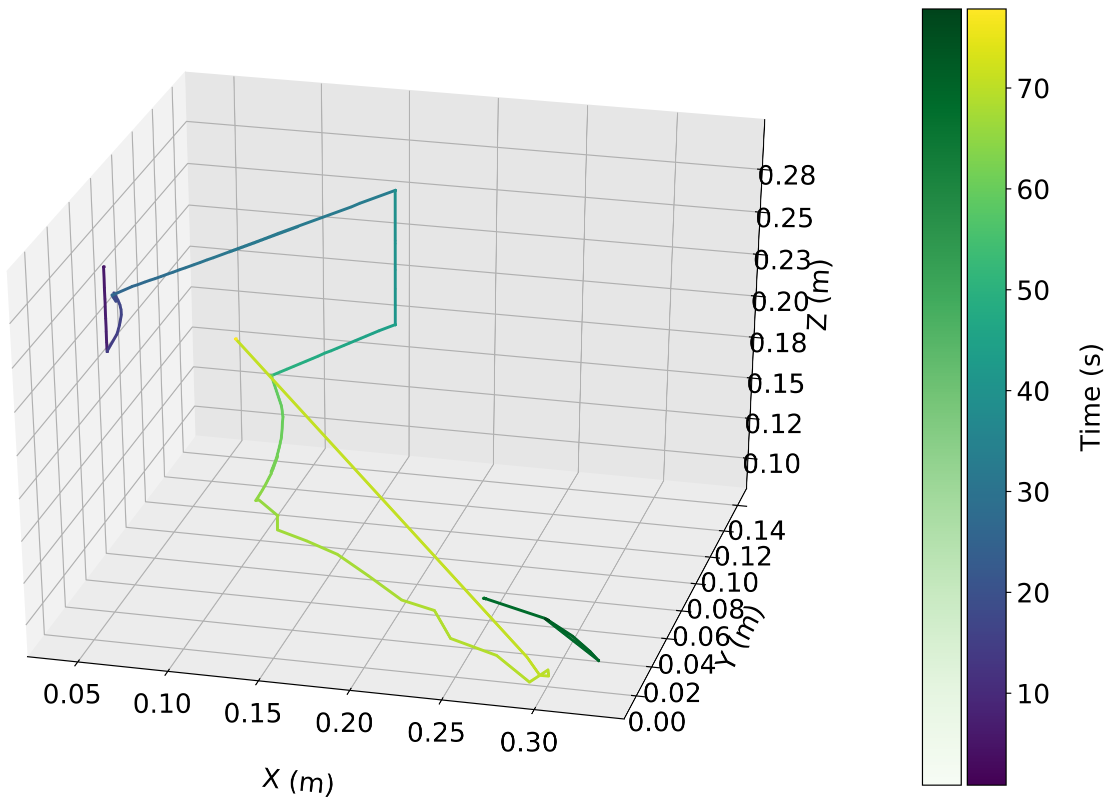
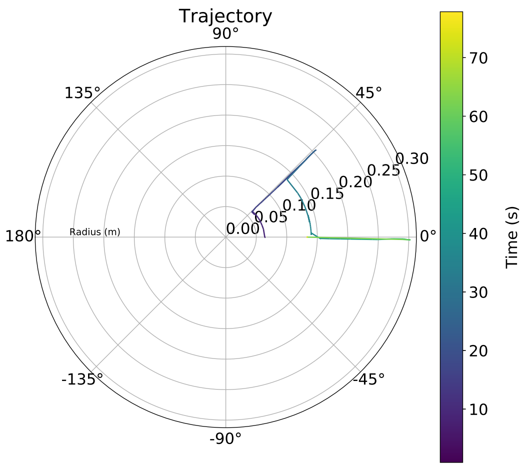


Figure 5.9: The path of the manipulator's end-effector plotted together with the path of the uppermost point of the lever (green). The paths are represented as gradients with respect to time.



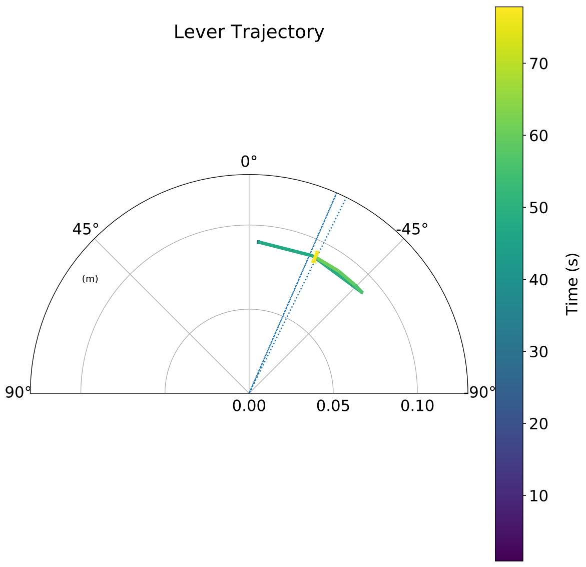


Figure 5.10: Left: Path of the end-effector in the $X^W Y^W$-plane, represented with polar coordinates. Right: Path of the lever in the $X^W Z^W$-plane, represented with polar coordinates. The threshold is plotted around the goal angle as two dotted lines.

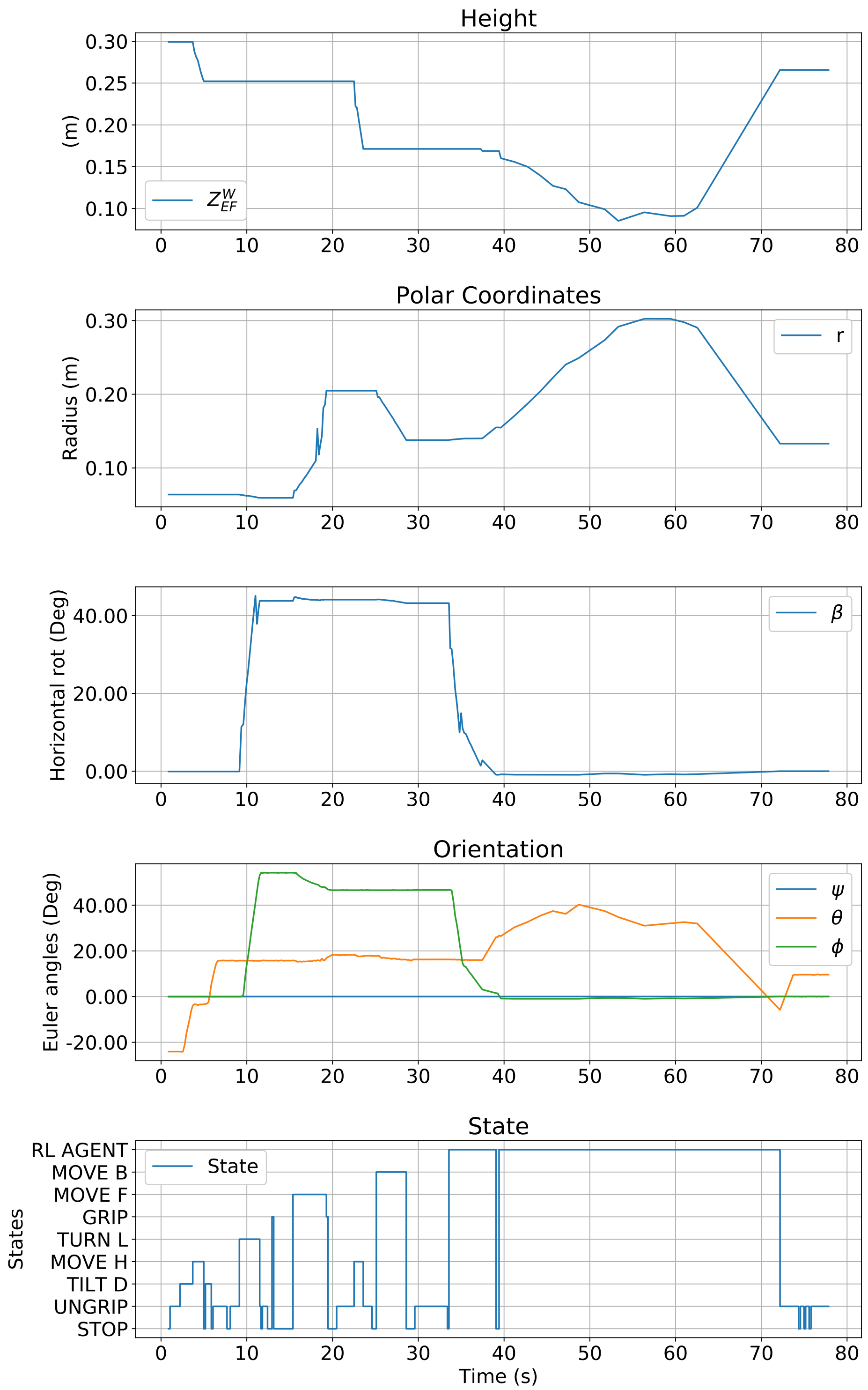


Figure 5.11: End-effector pose and active FSM state over time.

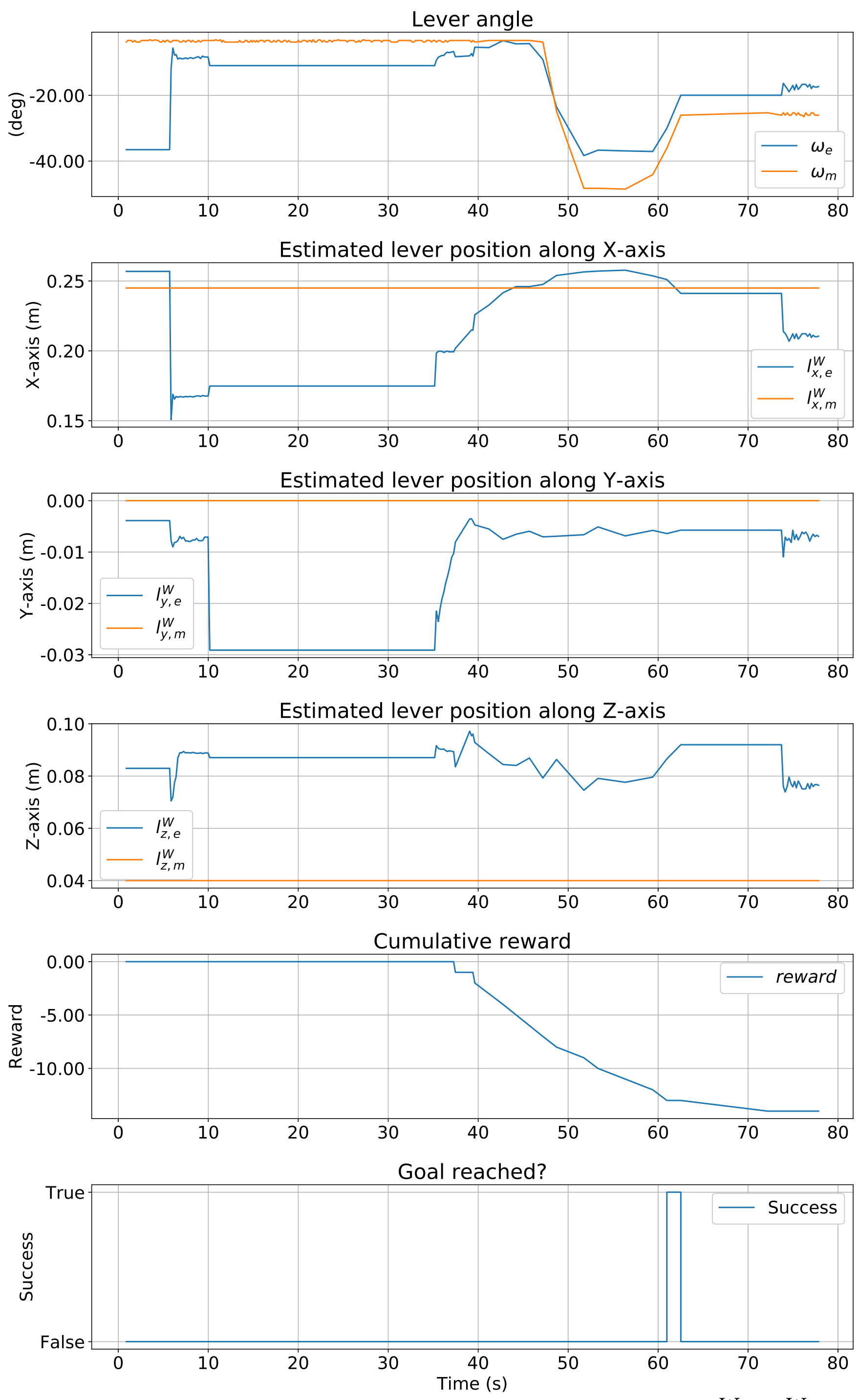


Figure 5.12: From top: Estimated and measured lever angle, X^W-, Y^W-, and Z^W-positions measured and estimated, cumulative reward, and success.

5.3 Discussion

5.3.1 RL Agent

Training the agent with a dense reward function did not yield adequate results. The best success ratio of around 80% was achieved after 85 epochs, as can be seen in Figure 5.4. The success ratios varied greatly during training, showing little signs of stabilizing at a high percentage. The considerable variation required the implementation of early stopping, as training for too long often resulted in a low success rate. The early stopping threshold parameter was set at 80%.

The sparse reward function fared far better. The success ratio steadily climbed to a high success ratio of 96.67%, which is 29 out of 30 successful validations, see Figure 5.2 and Figure 5.3. There was no need for early stopping when training with the sparse reward function. The noticeable drop in reward and success rate at epoch number 51 in Figure 5.3 is because the replay buffer is not saved after one training session. This might seem strange since the actor-network does not need a replay buffer to perform good actions when not training. Still, an empty replay buffer during the first epochs of a new training session can lead to unwanted behavior, as the transitions the network is trained on are drawn from this buffer. The sparse reward works better than the dense since HER is implemented. HER creates synthetic goals during the learning phase, which works well for sparse rewards. In the experiments carried out in [42], HER performed better on sparse than dense rewards.

During the collaboration challenge, it was noted that the retrained agent in Figure 5.3 performed worse than the one from Figure 5.2 that had only been trained for 50 epochs, even though the success rate was closer to 100% in the simulated environment. The agent's behavior was inspected in the MuJoCo environment, and an explanation was found. It had learned that it did not matter for the reward signal if it pushed the lever in a smooth and safe motion or dragged the end-effector along the ground towards the lever before pulling it. This is a flaw in the training procedure since the agent is allowed to learn behavior not tolerated in the real world. Solutions to this problem could be adding a larger negative reward every time the end-effector collides with anything other than the lever itself. Yet, by doing such a thing, the reward function will become denser, possibly affecting the training as seen in Figure 5.4. Another option is to implement Model Predictive Control (MPC) instead of the traditional PID controller on the manipulator. MPC allows for the control of a process while simultaneously satisfying a set of constraints [65]. With such a controller, the physical obstacles could be represented as constraints in the MPC, and we avoid the problem of reward shaping.

5.3.2 Measured Lever

In Figure 5.8, the measured lever position is constant, while the measured angle signal is stable at around $-5°$ for the first 40 seconds. Some spikes are present, which are likely caused by inaccuracies from the moving parts of the potentiometer. After 40 seconds, the lever angle starts gradually falling to around $-52°$. Meanwhile, looking at Figure 5.7, we note that the agent is slowly increasing the manipulator's radius while simultaneously reducing its height. Figure 5.6 seems to agree with this description, even though the exact timescale is not possible to read from that plot. Just before the 60 seconds mark, the lever angle is steadily increased until it comes within the threshold of the $-45°$ goal. Simultaneously, we can see in Figure 5.7 that the manipulator's tilt grows while the radius decreases. The height of the manipulator is also slightly reduced. Since the lever angle is negative when it points away from the manipulator, these actions seem sensible. As soon as the measured lever angle hits its target angle of $45 + / - 1.4°$, the Success plot steps from 'False' to 'True'. It is impressive that even though the agent initially pushed the lever too far, it corrected itself by picking it up again and slowly increasing the angle. Note that it is possible for the agent to push the lever too far as it does not evaluate the lever angle when a step is being performed in the environment.

5.3.3 Visually Estimated Lever

The difficulty increased when switching from measured to estimated lever values. The operator had to intervene several times to cancel the operation since the RL agent often positioned the manipulator too close to the lever base and risked dealing damage to the equipment.

The estimated lever angle, seen in the first plot of Figure 5.12, is reasonably close to the measured value, with the most significant difference of around 10°. The estimated value depends on the camera recording the Aruco indices attached to the lever. When these indices are not detected, the previous set of detections are used in the estimation.

The estimated X^W- and Z^W-position are noticeably affected by errors. The error in X^W-position becomes smaller as the end-effector approaches the manipulator at around 45 seconds, yet the difference in Z^W-position is only barely improved. This suggests that the X^W-position is more critical information to the agent, as it managed to manipulate the lever even though the Z^W-position was off by around $5cm$. The Y^W estimate was fairly accurate during the entire test, but suffered minor errors between 10 and 35 seconds. Inspecting Figure 5.11 revealed that the manipulator was turning horizontally such that the Aruco indices moved out of the camera frame during this time.

The PnP algorithm relies only on four detected points in the worst case and 12 points in the best case. With such a small set of points, inaccuracies in the camera, or a bad camera calibration distorting the image, will impact the transformation matrix. Further, the author did not calibrate the camera, since the supplied camera matrix was assumed to be correct. Distortion was not taken into account as well. Additionally, the translations of the transformation matrix from camera space to end-effector space were measured using a ruler and are prone to inaccuracies. These are likely the main contributing factors to the X^W- and Z^W-positional errors.

The agent is somewhat robust towards unreliable estimates. The agent reports success at approximately 61 seconds into the test. However, as the end-effector retracts, the estimate is updated to $-20°$, and the success signal falls to False. Meanwhile, the measured lever angle still stands at $-25°$. During other tests, it was noticed that when the estimates stopped updating, the agent still managed to put the lever in its correct position and was seemingly confused when it failed to receive a success signal, moving the lever carefully up and down from the goal angle.

Once the success signal becomes active, the agent resets the manipulator's position by reducing the radius, opening the gripper, and increasing the height, as seen in Figure 5.11.

The RL agent was trained on measured lever poses with no simulated noise or offsets. Distorting the lever pose during training would better mimic the visually estimated values, and could have contributed to better stability while performing the cooperation challenge. Nevertheless, it is impressive that the agent still managed to complete the challenge.

Chapter 6

Conclusions and Future Work

This thesis introduces a Human-Machine Cooperation Interface, demonstrated on a lever pull task, using a robotic manipulator and a simulation-trained RL agent. The cooperation challenge was completed using two different sets of lever poses. The first challenge used potentiometer readings and manually measured distances as input parameters for the agent. The second method used an RGB camera for visual detection and estimation of the lever pose. Both challenges were completed, but placing the lever at its wanted angle on estimated values was more challenging. Estimated values are preferred as one cannot assume that the object to be manipulated is continuously measured. To give the agent information extracted from a camera through robotic vision mimics the human ability to perceive objects and distances and is a valuable tool for Human-Machine Cooperation.

The next paragraphs are dedicated to answering the research questions from Section 1.2.

Can a human operator cooperate with an RL agent through an HMCI to solve complex challenges with a robotic manipulator?
From the results, we can clearly state that a human operator, in fact, can cooperate on an intuitive level with an RL agent through an HMCI to solve complex challenges with a robotic manipulator. The agent is overseen by a human while performing a difficult task, that is, placing a lever in a pre-determined position, and is stopped by the operator when it performs strange actions that might cause damage to the setup. The human operator is not able to operate the manipulator accurately enough, with the HMI from the author's project thesis [23], to push the lever into its goals position. Yet the human can perceive and calculate risks far better than the RL agent. Thus a symbiosis of mutual benefit is created, illustrating the value of Human-Machine Cooperation.

How transferable is an RL agent trained in a simulated environment to a physical one?
The RL agent was trained in the MuJoCo environment and was applied to the physical setup without any further training. To the author's surprise, the agent performed well. The largest hiccups were rooted in learned policies that worked well enough in the simulation but risked causing damage in the real world. Hitting the floor with the end-effector is perfectly acceptable behavior in the simulation, but increasing the physical actuator in a direction in which the manipulator cannot move, causes unnecessary

and undesirable stress and wear.

Can a noisy visually estimate of the lever pose replace reliable direct measurements as input signals for an RL agent?
Running the RL agent on estimated values did indeed increase the difficulty for the agent, which resulted in more work for the operator, who was required to interfere at a higher rate than before. However, considering that the estimates, and especially the Z^W-position of the lever, were affected by large offsets, it is still impressive that the agent successfully managed to manipulate the lever into its goal position.

The manipulator employed PID controllers in both the simulation and real life. By exchanging these controllers with MPC, one could include the physical constraints directly in the controller, thereby avoiding many of the dangerous actions taken by the RL agent without the need to design a dense reward function. Such a controller would alleviate much of the risk assessment required by the human operator and probably increase the success rate of the RL agent in the real-world setup due to the assumed decrease in dangerous situations.

To increase the robustness of the RL agent, work could be put into making the digital simulation closer to the physical one. Noise could be added to the lever pose measurements mimicking inaccuracies, but also, a time lag simulating the Aruco indices disappearing from the camera frame would increase the realism.

In Chapter 1, it was claimed that the HMCI developed here would classify as a level 4 system on the levels of autonomy by Lloyd's Register, Appendix A.3. The author still believes this to be true when the RL agent receives measured lever pose values on its inputs. The system is stable most of the time and completes its task without too much intervention from the human operator. However, when the system uses estimated lever pose values, it is somewhere between level 3 and level 4. The RL agent's rate of failure, or rate of performing a nonsensical action that might damage the equipment, is increased, resulting in a higher workload on the human operator, who must be ready to intervene at any moment.

The work presented in this thesis can be considered relevant for AUVs (Autonomous Underwater Vehicles), which human operators might oversee while they are performing tasks at an offshore platform or seabed installation. Disturbances from waves and currents are active in such scenarios and make the problem even more complex. Further, the AUV is moving relative to the platform as well. These challenges could be simulated by improving the physical setup. By placing the lever on a different platform and applying wave motions to the lever, the manipulator, or both, one could simulate such an environment and demonstrate the system's usefulness in a setup more true to a real-life scenario.

NASA plans to send humans back to the Moon within the next three years, and the humans will most definitely be accompanied by robots that can perform routine scientific and maintenance-related tasks. Going on spacewalks are risky endeavors that can cause irreparable radiation damage to the astronauts and is a risk that should be mitigated. Research within Human-Machine Cooperation could contribute to keeping the astronauts safe by facilitating cooperation between the humans from within the Lunar outpost and autonomous robots out on the surface.

Bibliography

[1] K. Donners, M. Waelkens, and J. Deckers. “Water mills in the area of Sagalassos: a disappearing ancient technology”. In: *Anatolian Studies* 52 (2002), pp. 1–17. DOI: 10.2307/3643076.

[2] Marilyn Palmer. “The Portsmouth Block Mills: Bentham, Brunel and the Start of the Royal Navy’s Industrial Revolution. By Jonathan Coad”. In: *Archaeological Journal* 163.1 (2006), pp. 289–290. DOI: 10.1080/00665983.2006.11020695.

[3] Maite Aparicio Latorre. “Cockpit assembly line for the model 3, Tesla”. MA thesis. Public University of Navarra, 2017.

[4] Brandi House, Jonathan Malkin, and Jeff Bilmes. “The VoiceBot: a voice controlled robot arm”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Boston, Massachusetts, Apr. 2009, pp. 183–192.

[5] Siddharth Narayanan and C Ramesh Reddy. “Bomb defusing robotic arm using gesture control”. In: *International Journal of Engineering Research and Technology* 4.02 (2015), pp. 89–93.

[6] Peter Vamplew and Anthony Adams. “Recognition of sign language gestures using neural networks”. In: *Australian Journal of Intelligent Information Processing Systems* 5.2 (1998), pp. 94–102.

[7] Hong Zeng et al. “Semi-autonomous robotic arm reaching with hybrid gaze-brain machine interface”. In: *Frontiers in neurorobotics* 13 (2020), p. 111.

[8] Vincent Van Roy, Daniel Vertesy, and Giacomo Damioli. “AI and robotics innovation”. In: *Handbook of Labor, Human Resources and Population Economics* (2020), pp. 1–35.

[9] Dmitry Kalashnikov et al. “Scalable deep reinforcement learning for vision-based robotic manipulation”. In: *Conference on Robot Learning*. Zürich, Switzerland, 2018, pp. 651–673.

[10] Hai Nguyen and Hung La. “Review of deep reinforcement learning for robot manipulation”. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. Naples, Italy, Feb. 2019, pp. 590–595.

[11] Athanasios S Polydoros and Lazaros Nalpantidis. “Survey of model-based reinforcement learning: Applications on robotics”. In: *Journal of Intelligent & Robotic Systems* 86.2 (2017), pp. 153–173.

[12] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation". Version 2. In: 1 (2016). arXiv: `1610.00633`.

[13] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. "Sim-to-real transfer in deep reinforcement learning for robotics: a survey". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. Canberra, Australia, Nov. 2020, pp. 737–744.

[14] Jean-Michel Hoc. "From human–machine interaction to human–machine cooperation". In: *Ergonomics* 43.7 (2000), pp. 833–843.

[15] Lloyd's Register. *Cyber-Enabled Ships Shipright Procedure—Autonomous Ships*. 2016. URL: `http://info.lr.org/l/12702/2016-07-07/32rrbk` (visited on May 29, 2022).

[16] Ian Sample. *Computer says no: why making AIs fair, accountable and transparent is crucial*. URL: `https://www.theguardian.com/science/2017/nov/05/computer-says-no-why-making-ais-fair-accountable-and-transparent-is-crucial` (visited on Nov. 10, 2021).

[17] Thilo Hagendorff. "The ethics of AI ethics: An evaluation of guidelines". In: *Minds and Machines* 30.1 (2020), pp. 99–120.

[18] Oxford Dictionary. *Definition: Accountability*. URL: `https://www.oxfordlearnersdictionaries.com/definition/english/accountability` (visited on June 16, 2022).

[19] Oxford Dictionary. *Definition: Robustness.* URL: `https://www.oxfordlearnersd com/definition/english/robustness?q=robustness` (visited on June 16, 2022).

[20] Nicolas Padoy and Gregory D Hager. “Human-machine collaborative surgery using learned models”. In: *2011 IEEE International Conference on Robotics and Automation.* Shanghai, China, May 2011, pp. 5285–5292.

[21] Suncheol Kwon and Jung Kim. “Real-Time Upper Limb Motion Estimation From Surface Electromyography and Joint Angular Velocities Using an Artificial Neural Network for Human–Machine Cooperation”. In: *IEEE Transactions on Information Technology in Biomedicine* 15.4 (2011), pp. 522–530. DOI: `10.1109/TITB.2011.2151869`.

[22] Sindre Benjamin Remman. “Robotic manipulation using deep reinforcement learning”. MA thesis. NTNU, 2020.

[23] Simon Julian Nagelsaker Lexau. *Hand Tracking for Robotic Control through Deep Convolutional Neural Networks.* Project Thesis. Nov. 2021.

[24] Leonid B. Freidovich. *Lecture Notes from Modelling in Robotics and Control Methods for Robotic Applications.* Umeå: Umeå University, 2021.

[25] Robotis. *OpenManipulator-X e-Manual.* URL: https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/ (visited on May 9, 2022).

[26] Wikipedia. *Denavit-Hartenberg parameters.* Mar. 17, 2022. URL: https://en.wikipedia.org/wiki/Denavit-Hartenberg_parameters (visited on Mar. 22, 2022).

[27] Charles W. Wampler. "Manipulator Inverse Kinematic Solutions Based on Vector Formulations and Damped Least-Squares Methods". In: *IEEE Transactions on Systems, Man, and Cybernetics* 16.1 (1986), pp. 93–101. DOI: 10.1109/TSMC.1986.289285.

[28] Yoshihiko Nakamura and Hideo Hanafusa. "Inverse Kinematic Solutions With Singularity Robustness for Robot Manipulator Control". In: *Journal of Dynamic Systems, Measurement, and Control* 108.3 (1986), pp. 163–171. DOI: 10.1115/1.3143764.

[29] Samuel R Buss and Jin-Su Kim. "Selectively damped least squares for inverse kinematics". In: *Journal of Graphics tools* 10.3 (2005), pp. 37–49. DOI: 10.1080/2151237X.2005.10129202.

[30] Michael JD Powell. "An efficient method for finding the minimum of a function of several variables without calculating derivatives". In: *The computer journal* 7.2 (1964), pp. 155–162. DOI: 10.1093/comjnl/7.2.155.

[31] Charles G Broyden. “A class of methods for solving nonlinear simultaneous equations”. In: *Mathematics of computation* 19.92 (1965), pp. 577–593.

[32] Roger Fletcher. *Practical Methods of Optimization*. Second Edition. Chichester: John Wiley & Sons, 1987.

[33] Andreas Aristidou and Joan Lasenby. “FABRIK: A fast, iterative solver for the Inverse Kinematics problem”. In: *Graphical Models* 73.5 (2011), pp. 243–260.

[34] Jiacun Wang. *Formal Methods in Computer Science*. New York: CRC Press, 2019.

[35] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed Bandit Allocation Indices*. Second Edition. Chichester: John Wiley & Sons, 2011.

[36] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Version 1. 2013. arXiv: `1312.5602`.

[37] Elena Pashenkova, Irina Rish, and Rina Dechter. “Value iteration and policy iteration algorithms for Markov decision problem”. In: *AAAI’96: Workshop on Structural Issues in Planning and Temporal Reasoning*. Portland, Oregon, Aug. 1996.

[38] Svajone Bekesiene, Rasa Smaliukiene, and Ramute Vaicaitiene. “Using artificial neural networks in predicting the level of stress among military conscripts”. In: *Mathematics* 9.6 (2021), p. 626.

[39] Maria V Valueva et al. "Application of the residue number system to reduce hardware costs of the convolutional neural network implementation". In: *Mathematics and Computers in Simulation* 177 (2020), pp. 232–243.

[40] Timothy P Lillicrap et al. *Continuous control with deep reinforcement learning*. Version 6. 2015. arXiv: `1509.02971 [cs.LG]`.

[41] Boris T Polyak and Anatoli B Juditsky. "Acceleration of stochastic approximation by averaging". In: *SIAM journal on control and optimization* 30.4 (1992), pp. 838–855.

[42] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems*. Long Beach Convention & Entertainment Center, California: Curran Associates, Inc., Nov. 2017.

[43] Sintef. *Definition of the term: Robotic Vision*. URL: `https://www.sintef.no/en/expertise/digital/optical-measurement-systems/robot-vision/` (visited on May 5, 2022).

[44] Martin A Fischler and Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[45] OpenCV. *Perspective-n-Point (PnP) pose computation.* URL: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html (visited on May 6, 2022).

[46] Zhengyou Zhang. “A flexible new technique for camera calibration”. In: *IEEE Transactions on pattern analysis and machine intelligence* 22.11 (2000), pp. 1330–1334.

[47] Wikipedia. *Pinhole Camera Model.* June 16, 2021. URL: https://en.wikipedia.org/wiki/Pinhole_camera_model (visited on May 5, 2022).

[48] Marian Körber et al. *Comparing popular simulation environments in the scope of robotics and reinforcement learning.* Version 1. 2021. arXiv: 2103.04616.

[49] Roboti LLC. *MuJoCo Documentation.* URL: https://mujoco.readthedocs.io/en/latest/overview.html (visited on Mar. 17, 2022).

[50] OpenAI. *GitHub repository of the DDPG + HER implementation.* 2022. URL: https://github.com/openai/baselines/tree/master/baselines/her (visited on June 2, 2022).

[51] Microchip. *MCP3001, Single Channel ADC.* URL: https://www.microchip.com/en-us/product/MCP3001 (visited on May 2, 2022).

[52] Raspberry Pi. *Raspberry Pi 3 Model B Schematic.* 2022. URL: `https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-reduced-schematics.pdf` (visited on June 2, 2022).

[53] Wikipedia. *Serial Peripheral Interface.* Apr. 20, 2022. URL: `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface` (visited on May 2, 2022).

[54] Wikipedia. *User Datagram Protocol.* Apr. 14, 2022. URL: `https://en.wikipedia.org/wiki/User_Datagram_Protocol` (visited on May 2, 2022).

[55] OpenCV. *OpenCV Aruco Markers Documentation.* URL: `https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html` (visited on May 5, 2022).

[56] Google. *MediaPipe GitHub repository.* URL: `https://google.github.io/mediapipe/` (visited on Dec. 16, 2021).

[57] Google. *Model Card: MediaPipe Hands.* URL: `https://drive.google.com/file/d/1-rmIgTfuCbBPW_IFHkh3f0-U_lnGrWpg/preview` (visited on Nov. 4, 2021).

[58] Hongyi Xu et al. “Ghum & ghuml: Generative 3d human shape and articulated pose models”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* Virtual, June 2020, pp. 6184–6193.

[59] The Qt Company. *Qt application framework.* URL: `https://www.qt.io/` (visited on Dec. 11, 2021).

[60] Brian Gerkey. *Why ROS 2.* May 2022. URL: `https://design.ros2.org/articles/why_ros2.html` (visited on May 14, 2022).

[61] Dirk Thomas. *Changes between ROS 1 and ROS 2.* June 2017. URL: `http://design.ros2.org/articles/changes.html` (visited on May 12, 2022).

[62] Open Robotics. *Python migration guide from ROS 1.* URL: `https://docs.ros.org/en/foxy/Contributing/Migration-Guide-Python.html` (visited on May 12, 2022).

[63] Visual Computing Laboratory. *MeshLab Software.* 2022. URL: `https://www.meshlab.net/` (visited on June 2, 2022).

[64] Simon Julian Nagelsaker Lexau. *GitHub repository of the Human-Machine Cooperation Interface.* 2022. URL: `https://github.com/Nagelsaker/mimir` (visited on June 2, 2022).

[65] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control.* Second Edition. London: Springer-Verlag, 2007, pp. 13–26.

[66] PERO Vidan et al. "Autonomous Systems & Ships-Training and Education on Maritime Faculties". In: *8th International Maritime Science Conference.* Budva, Montenegro, Apr. 2019, pp. 91–101.

Appendix A

A.1 OpenManipulator-X Dimensions

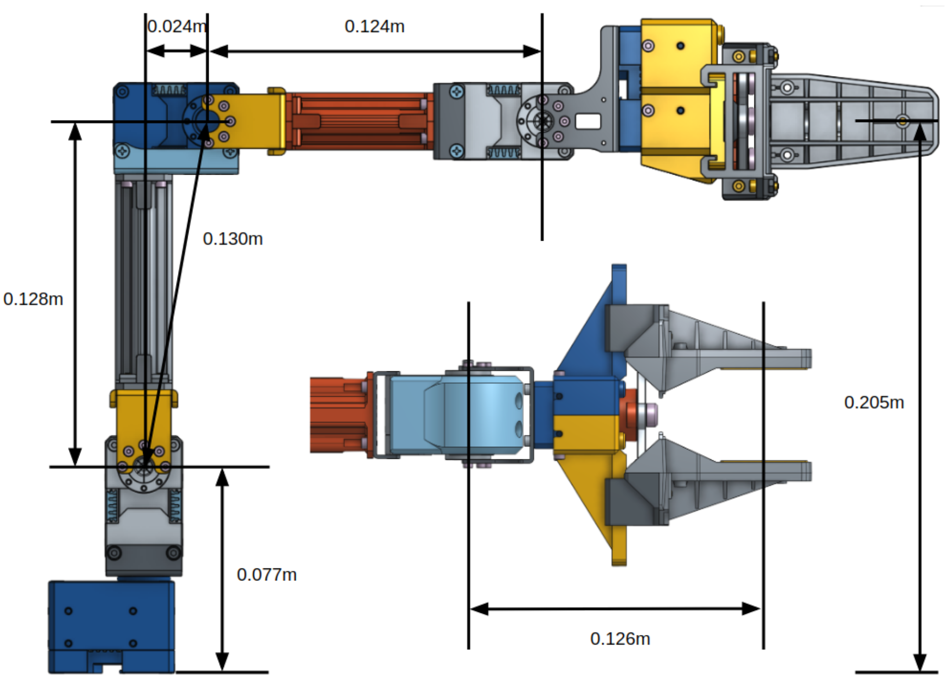


Figure A.1: Dimensions of the OpenManipulator-X. From Robotis e-Manual [25]

A.2 Class Diagrams

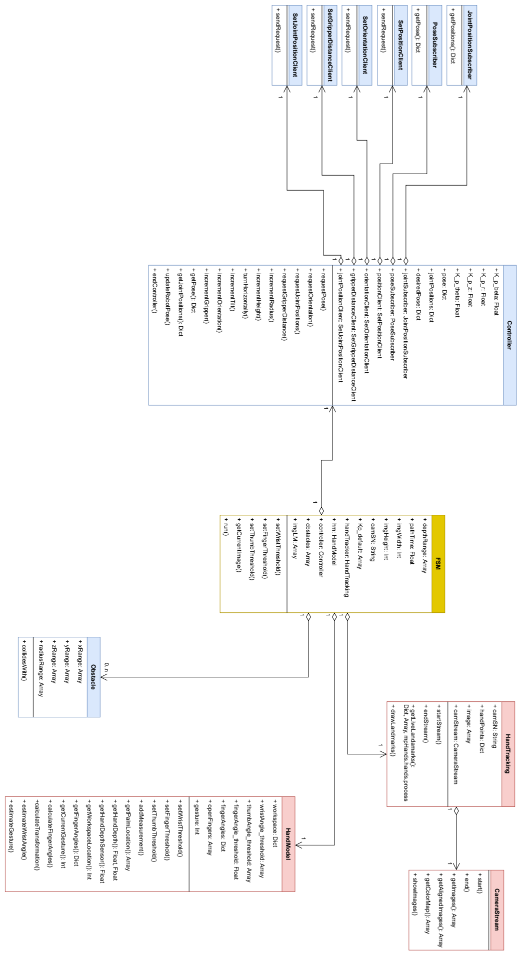


Figure A.2: Class diagram depicting the most important parts of the system.

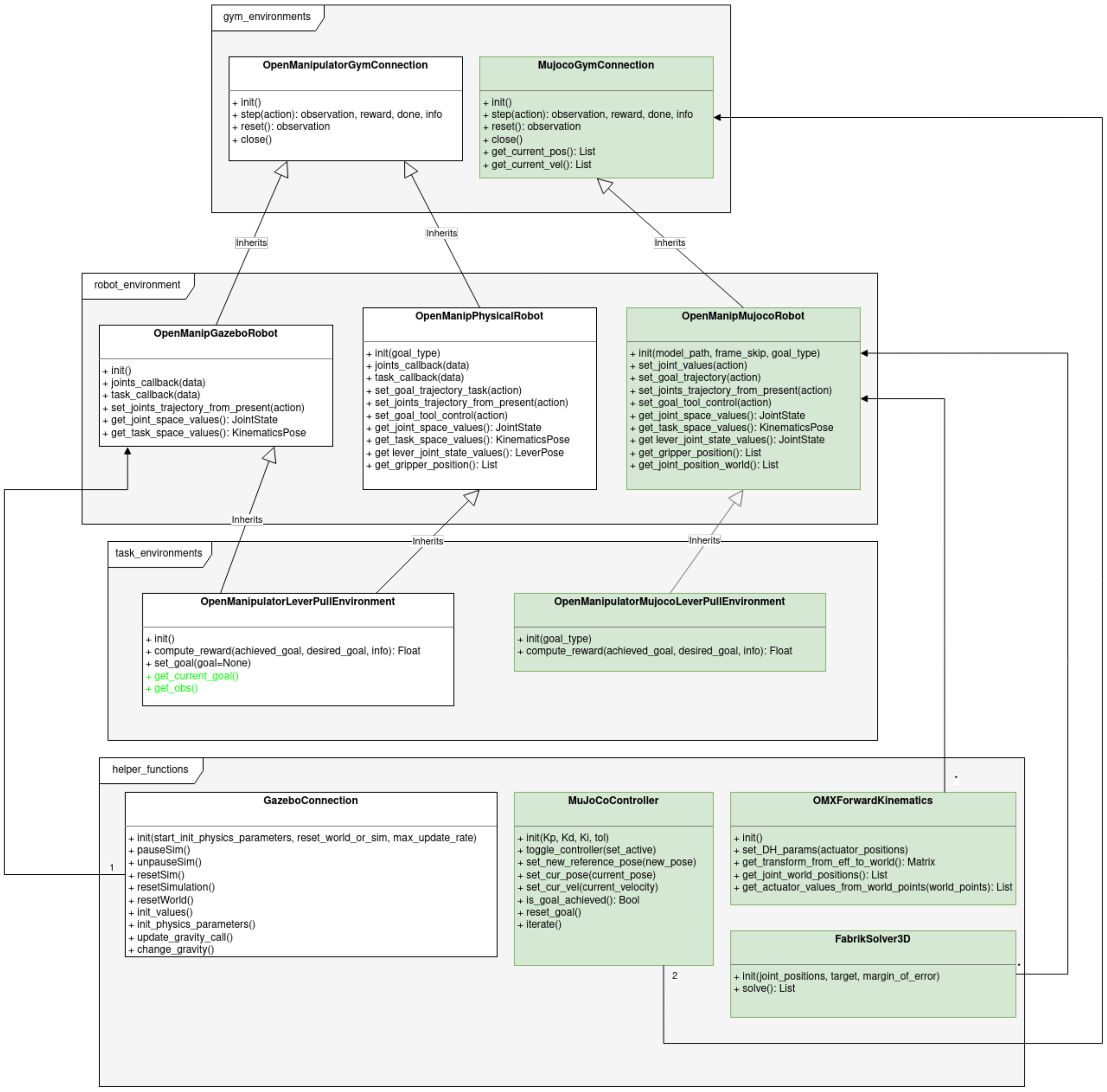

gym_environments
OpenManipulatorGymConnection
+ init()
+ step(action): observation, reward, done, info
+ reset(): observation
+ close()
MujocoGymConnection
+ init()
+ step(action): observation, reward, done, info
+ reset(): observation
+ close()
+ get_current_pos(): List
+ get_current_vel(): List
Inherits
Inherits
Inherits
robot_environment
OpenManipGazeboRobot
+ init()
+ joints_callback(data)
+ task_callback(data)
+ set_joints_trajectory_from_present(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
OpenManipPhysicalRobot
+ init(goal_type)
+ joints_callback(data)
+ task_callback(data)
+ set_goal_trajectory_task(action)
+ set_joints_trajectory_from_present(action)
+ set_goal_tool_control(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
+ get lever_joint_state_values(): LeverPose
+ get_gripper_position(): List
OpenManipMujocoRobot
+ init(model_path, frame_skip, goal_type)
+ set_joint_values(action)
+ set_goal_trajectory(action)
+ set_joints_trajectory_from_present(action)
+ set_goal_tool_control(action)
+ get_joint_space_values(): JointState
+ get_task_space_values(): KinematicsPose
+ get lever_joint_state_values(): JointState
+ get_gripper_position(): List
+ get_joint_position_world(): List
Inherits
Inherits
Inherits
task_environments
OpenManipulatorLeverPullEnvironment
+ init()
+ compute_reward(achieved_goal, desired_goal, info): Float
+ set_goal(goal=None)
+ get_current_goal()
+ get_obs()
OpenManipulatorMujocoLeverPullEnvironment
+ init(goal_type)
+ compute_reward(achieved_goal, desired_goal, info): Float
helper_functions
GazeboConnection
+ init(start_init_physics_parameters, reset_world_or_sim, max_update_rate)
+ pauseSim()
+ unpauseSim()
+ resetSim()
+ resetSimulation()
+ resetWorld()
+ init_values()
+ init_physics_parameters()
+ update_gravity_call()
+ change_gravity()
1
MuJoCoController
+ init(Kp, Kd, Ki, tol)
+ toggle_controller(set_active)
+ set_new_reference_pose(new_pose)
+ set_cur_pose(current_pose)
+ set_cur_vel(current_velocity)
+ is_goal_achieved(): Bool
+ reset_goal()
+ iterate()
2
OMXForwardKinematics
+ init()
+ set_DH_params(actuator_positions)
+ get_transform_from_eff_to_world(): Matrix
+ get_joint_world_positions(): List
+ get_actuator_values_from_world_points(world_points): List
FabrikSolver3D
+ init(joint_positions, target, margin_of_error)
+ solve(): List

Figure A.3: Extended class diagram of the *Open Manipulator RL Environments* package. Green parts were written as part of this thesis

A.3 Levels of Autonomy

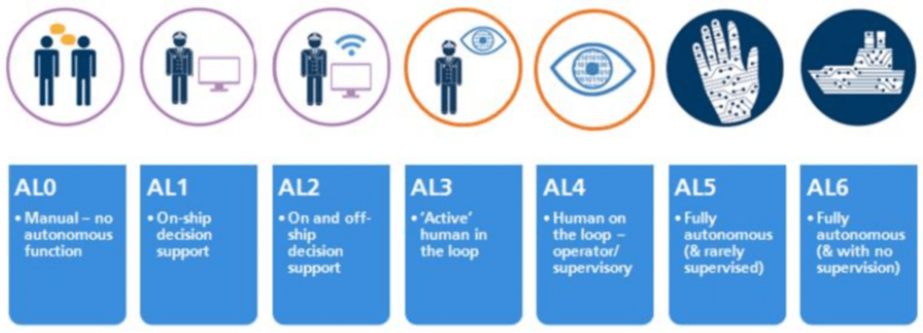

AL0
• Manual – no autonomous function
AL1
• On-ship decision support
AL2
• On and off-ship decision support
AL3
• 'Active' human in the loop
AL4
• Human on the loop – operator/ supervisory
AL5
• Fully autonomous (& rarely supervised)
AL6
• Fully autonomous (& with no supervision)

Figure A.4: Levels of Autonomy according to LLoyd's Register. Image borrowed from [66]

A.4 Code

Listing A.1: Python code for reading digital angle voltage, converting it to lever angle and publishing the result over UDP

```python
import numpy as np
import logging
log_format = "%(levelname)s | %(asctime)-15s | %(message)s"
logging.basicConfig(format=log_format, level=logging.DEBUG)
import RPi.GPIO as GPIO
import time
import socket
import spidev

def transmit(value, sock, ip, port, freq):
    '''
    Transmit value over UDP
    '''
    message = bytes(f"{value}", "utf-8")
    try:
        sock.sendto(message, (ip,port))
    except:
        pass
    time.sleep(1/freq)

def main():
    # Setup UDP connection
    DST_IP = "10.42.0.1"
```

```
DST_PORT = 20000
sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
freq = 10 # Hz

# SPI
min_angle = -96
max_angle = 98
min_val = 124 #50
max_val = 917 #450
bus = 0
device = 0
n = 10
spi = spidev.SpiDev()
spi.open(bus, device)

# SPI settings
spi.max_speed_hz = int(2.8e6)
spi.mode = 0b11
vref = 5.0

try:
    while True:
        out = spi.readbytes(4)
        b1 = int(bin(out[0] << 5),2)
        b2 = int(bin(out[1])[:7],2)
        b3 = int(bin(out[2]),2)
        val = int(bin(b1 | b2),2)
```

```
                angle = np.deg2rad((val - min_val)/
                    (max_val-min_val) * (max_angle-min_angle) +
                    min_angle)
                transmit(angle, sock, DST_IP, DST_PORT, freq)
                print(f"bits:\t{bin(b1 | b2)}\tangle:
                    {angle}\tval:\t{int(bin(b1 | b2),2)}")
        except KeyboardInterrupt:
            spi.close()
            sock.close()

    if __name__ == "__main__":
        main()
```