UPPSALA
UNIVERSITET

Juni 2025

Optimized Motion Control of the ViperX 300 S: Navigating Dynamic Environments

Alexander Åberg & Silas Ekman

Högskoleingenjörsprogrammet i elektroteknik

We would like to begin by expressing our sincere gratitude to **Christos K. Vergines**, **Olle Svensson**, **Kjell Staffas**, and **Pekka Alakulju** from various departments at Uppsala University for their dedication and support. We also extend our thanks to Alexander's line manager **Henrik Wennebeck** at SAAB.

Abstract

The following research delved into optimizing the usage of a robotic arm known as ViperX 300 S with the aim of accomplishing a “pick-and-place-machine”. Furthermore, different varieties of regulator techniques and algorithms were implemented and evaluated against each other to examine which was the most suitable for the specific task, which is moving a fragile object. There are two algorithms developed in Python, such as the current based algorithm where the robotic arm learns to reach different position with negligible error, as well effort based where the robotic arm uses effort to reach specific positions.
There is a camera on the robotic arm that was discarded, and object markers were not set or used. Regulators were implemented to ensure it reached its target points. The main goal was to create a model-based control and a PID regulator that was sufficient for a pick and place algorithm, which is regulating an object's movement automatically whilst preventing harsh changes in velocity. However, due to fragile hardware and limited prerequisite skills, this was never completed.
The test area was established indoors on the plane floor with moving and stationary cardboard boxes in different shapes, creating a maze that the robotic arm had to pass through. Simulations and measurements were carried out to determine which were the most suitable for the task. This project contributes to injury prevention for individuals working with robotic arms, while also advancing the implementation of algorithms that offer effective solutions with enhanced performance.

Teknisk-naturvetenskapliga fakulteten
Uppsala universitet, Utgivningsort Uppsala
Handledare: Christos Verginis Ämnesgranskare: Kjell Staffas
Examinator: Andrej Savin

Populärvetenskaplig Sammanfattning

Denna rapport fördjupade sig i användningen av en robotarm vid namn ViperX 300 S med målet att skapa en "pick and place" algoritm. Vidare implementera PID regulator med justerbara parameterar som styr dämpning, stabilitet och så att det stationära reglerfelet blir så litet som möjligt. Huvudsakligen utvecklades två algoritmer, en som styrs av vridmoment och en annan som styrs av ström. Den som styrs av vridmoment var använd i simuleringar medans den strömbaserade varianten användes på den fysiska robotarmen. Tanken var att skapa en modellprediktiv reglering (Model Predictive Control), där robotarmen förutser framtida scenarion och därmed undviker potentiella kollisioner men gjordes aldrig.
För att undvika kollisioner helt så uteslöts även objektprogrammering och därmed Vision-kameran och detta med samråd från handledaren. Målet blev i slutändan att implementera en strömbaserad kontroller med position som output. Den ursprungliga kontrollen var positionsbaserad men inte strömstyrd vilket orakade att förändringar behövdes göras i källkoden. Baserat på källkoden så modifierades olika launch-filer, XML samt YAML filer för att göra kontrollen aktiv.
Rapporten kommer fördjupa sig i regler och styrsystem samt den tekniska delen att kunna applicera detta i verkligt system. Vidare kommer mycket programmering användas med olika programspråk. Programspråken kommer ha sin utgångspunkt i ROS som är bibliotek eller ramverk för detta robotsystem. Vidare så diskuteras möjligheter till förbättring samt hur felsökning av robotarmen gick till.

Table of contents

Chapter I

Introduction

“Do you not see that I am like one of those machines?”
- Mary Shelley, Frankenstein (1818)

1.1 Background

The word robot derives from a slavic word called “rabota” which identifies with forced labor and is mentioned in a science fiction play from the 1920’s written by Karel Čapek which was named R.U.R. This stands for Rossumovi Univerzální Roboti. In this play, a company called R.U.R used biotechnology in order to manufacture artificial workers. Even though the play was popular, the first robotic arm wasn’t made untill 1959 by George Devol and Joseph Engleberger titled Unimate#001. Further on, during 1961 George received a patent for his invention which eventually led to their establishment of the world's first robotic company called Unimation. This company developed robots for welding, die casting and other implementations for the automotive industry (Čapek, 2001). A robotic arm can be identified by some important key features. Namley structure and joints, also known as Degrees of Freedom (DOF). These rigid segments are connected by joints which can be linear or rotary. In addition, these joints determine the flexibility of the arm, where more joints equals more freedom of motion (Roberts, n.d.).

End effector, also called Tooling indicates the operating part of the arm or more simplified the hand of the arm. These may vary from a simple claw to a welding torch, suction cup or a very functional human-like hand. In addition, these claws are called grippers which can be divided into more subcategories, such as mechanical, often two or three claws that open and close to grab an object or pneumatic grippers which use air pressure to control the claws. This allows for a stronger force towards the object. Moreover there are magnetic grippers which apply magnetism for lifting objects which are ideal for metal parts. On top of that, vacuum grippers use suction to handle light weight or fragile objects. An actuator in the robotic arm is a machine itself that controls the joints and end effectors by converting energy into physical movement. Without this the robot wouldn’t be able to perform tasks. This component is often powered by electric motors, which run on direct current or alternating current. These are often the most common types since they are precise, easy to manipulate but also effective with integration with several types of sensors and feedback systems. Besides electrical motors there are also hydraulic and pneumatic actuators. Furthermore it’s important to know that actuators work with control systems in order to ensure that the robot moves as intended. With the help of sensors which can measure force, torque and distance. The actuator will be given real-time feedback and process it in order to proceed with its task (Roberts, n.d.).

The control system orders the arm which task to perform. It could also be recognized as the “brain” of the robot. This is where all the decisions are made, real time calculations, commands and other instructions. In order to handle all this information most robotic arms use a Central Processing Unit (CPU) or a microcontroller. Additionally, on the controller there is software which defines what tasks and operations the robot should execute. This software is usually developed with some sort of programming language, such as C++, Java or Python. Robotic arms use control loops which ensure smooth, precise and repeatable movement. There are two most common types, which are open-loop or closed loop. The open-loop system doesn’t have any feedback information, but rather only follows pre-coded instructions. This usually revolves around simpler tasks, i.e moving fast towards a desired location or a swift movement during a small amount of time. Closed-loop control

continousley sends back information in real time while checking its own movement using feedback sensors. This allows the robotic arm to adjust its own position if it were to move in the wrong direction (Roberts, n.d.).

It's important to note that there are different types of robotic arms that are not defined by their given task. As an example, the Cartesian Robotic Arm uses linear movement in three different directions, denoted "x", "y" and "z". These directions give beneficial strength such as high precision and are especially good for a simple pick-and-place task as well for 3D-printing. There is also Selective Compliance Articulated Robot Arm (SCARA) which has rotational movement in a horizontal plane combined with vertical movement. This gives high velocity, precise movement and makes it ideal for horizontal tasks. Not to mention the Spherical Robotic Arm, which moves with telescoping motion whilst rotating. This arm is very efficient upon reaching through small spaces. Some applications are welding and injection molding (Robotics Academy, n.d.).

1.2 Background robotic arm

1.2.1 Rigid-body systems

In physical terms a rigid-body is defined by a solid body where deformation (a change in its shape caused by an applied force) is equal to zero or even negligible. That is, a certain distance between two given points on its body remains constant in the time domain even if any external forces or momentum are applied on it. Moreover a rigid body is often recognized as a continuous distribution of mass. Whereas if you were to combine several types of these bodies, you would achieve a rigid-body system. That's why a robotic arm with mulitple rigid segments connected by rotating joints can also be called a rigid-body system. However, in reality a perfectly rigid body would be impossible to achieve, there will always be deformation (in very small scales) which turns it into an approximation. Rigid bodies often experience two types of motion. One would be whilst the entire body moves along a linear path with no rotation, this is known as translational motion. Whereas if the body were to rotate about an axis it would go under the name of rotational motion. Moreover, both of these momentums could occur together (Bolmsjö, 2006).

In 3D space (3 dimensions) a rigid body has 6 degrees of freedom (DOF, number of ways the body can move), this would be along the x-axis, y-axis and so on. In this case, it has 3 degrees for translational movement and 3 for rotational movement (Batlle & Barjau Condomines, 2022).

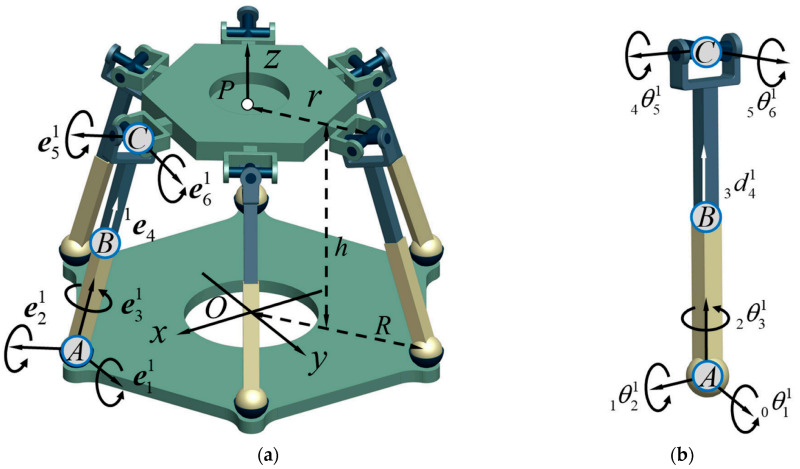

Figure 1: Two different rigid-body systems, (a) and (b), showing varying rotational angles, distinct points labeled A, B, and C, as well as length notations and coordinates (Zhao et al., 2023).

1.2.2 Robot operating system

Robot Operating System (ROS) is an open-source framework that runs on top of Ubuntu (Linux distribution), especially designed for robotic software development. The main languages for this program is C++ and Python, whereas C++ is recommended due to better performance and application. Further on, ROS contains several useful packages. There are many common robotic basic and very advanced functions as well as algorithms. This helps users with not having to reinvent the wheel. To put it briefly, ROS helps developers working with robotic arms with a good launch for their projects by providing pre-developed software (Batlle & Barjau Condomines, 2022).

1.2.3 Linux

Linux is a well known operating system (such as Windows) which often runs on a computer. This program acts as an interface between the user and the computer's hardware, it's the core software of the computer. It allows other programs to run properly on a computer, constructing a Graphical User Interface (GUI) and a file system which other programs can interact with. It's also important to note that Linux has several distributions, this is due to Linux being open source which allows users themselves to develop an own version and release it for usage (Linux Kernel Organization, n.d.).

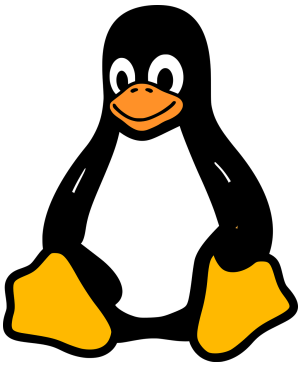

Figure 2: The mascot of Linux, Tux (Linux Foundation, n.d.).

1.3 Purpose

Several tests and simulations of a ViperX 300 S robotic arm test rig will be performed. With the help of the test rig, the project's purpose will be fulfilled which is to provide an automatic "pick-and-place" robotic arm. Further on it includes an interactive software that gives instructions to the arm, while displaying simulations and important data.

1.4 Main Research Questions

- Which regulator will be the most effective?
- Which algorithm will be the most beneficial?

1.5 Delimitations

The focus of this project is to achieve a pick and place methodology without any human interaction, purely automatic. It covers regulator technology, deep software development and signal processing of data measured by sensors. Any other topics that could be included or related to these types of projects are not of relevance for this paper.

Chapter II

Theory

"The more I learn, the more I realize how much I don't know."
- Albert Einstein

2.1 Control theory

The most important key components of a control system consist of input signals, control system components and the output signals. The inputs are usually referred to as actuating signals, denoted "u", whereas the output signals can be called controlled variables, denoted "y". Generally speaking, the whole purpose of the control system is to control the output signals in some predefined way which are generated by the input signals passing through the control system components (Thomas, 2016).

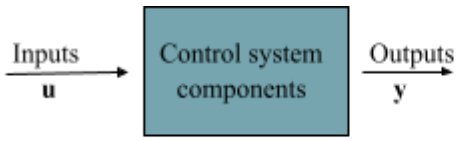

Figure 3: Key components of a control system, where the inputs are denoted u and the outputs denoted y. Figure created by the authors.

There are two types of control systems as mentioned earlier, open-loop and closed-loop. These are also called "non feedback systems" and "feedback control systems". They can usually be defined by looking if the system has a link or not between its output towards its input on its block diagram. This link provides more accurate control. This is achieved by sending back the controlled signal "y" towards the input "u", and then comparing them, taking the difference between their values which gives an actuating signal proportional to the difference which then is sent through the system to correct an error. This error is the difference between the desired output signal, often called "setpoint" and the actual(real time) output(the measured value) (Thomas, 2016). Mathematically expressed as:

$$e(t) = r(t) - y(t)$$

Equation 1: Calculation of the error, e(t).

Where: $e(t)$ ~ Error signal/setpoint
$r(t)$ ~ Reference input
$y(t)$ ~ Measured value

The error signal is then used by the control system to adjust the system's behavior and minimize any deviations from the desired output. An example would be a robotic arm with the desired mechanism to reduce speed when encountering objects. Sensors detect an object in front of the arm's travel path, sending an input signal with that information, fed into the control system. The arm starts to reduce its velocity. At this point the developer might have a desired velocity reduction depending on the distance between the arm and the object. This can be achieved with a closed loop system, sending back real time data which allows the system to correct the error. The arm might have a certain velocity reduction but not equal to the desired one. In this case the error signal adjusts the speed and minimizes the deviation, leading to a desired output signal (Thomas, 2016).

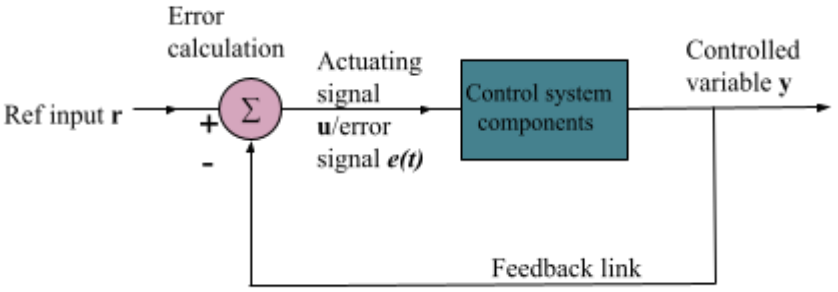

Figure 4: Closed-loop control system with a reference input denoted r, an error calculation where the error signal e(t) is calculated, a control system that performs a task and the output of the system, controlled variable denoted y. Figure created by the authors.

The non feedback system doesn't have a feedback link, the output doesn't have any influence on the input. This system's output is a direct result of the input signals, with no continuous alterations based on environmental changes. A simple car seat heating system would be a good example of an open loop system, where the heat is set to a specific level, produced and runs continuously but also doesn't change the temperature regardless of the seat's temperature. It makes no correction or adjustment (Thomas, 2016).

2.1.1 Terminologies

When designing a control system there are different ways to analyze the result of the system. The first step would be to apply a unit step function, *u(t)*, also called heavy-side function, towards the control's system input. This input signal reflects a sudden change in the input, e.g. turning on a system at the begining($t = 0$). Afterwards, you look at the output signal/the response of the system and the characteristics of the waveform (Thomas, 2016).

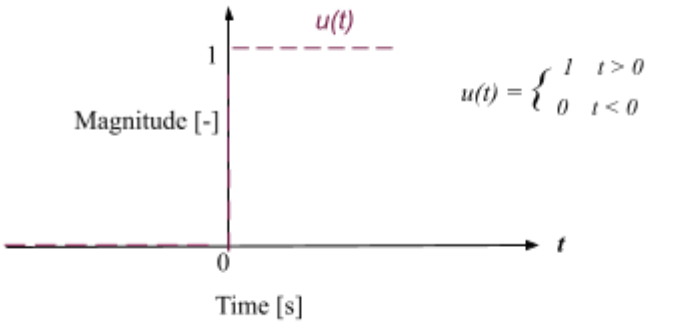

Figure 5: Unit step function, u(t), which is 1 when t is above 0 whilst 0 when t is below 0 in the time domain. Figure created by the authors.

Rise Time is defined by the amount of time it takes for the system to go from 10% towards 90% of the steady-state, also called final-value. The steady-state value of the control system is the final-value the system's output reaches after all transients of the waveform have passed. It can be described as the long-term behavior of the control system. Sometimes the amplitude of the Rise Time can exceed the steady-state value over time. This gives the system a certain Overshoot, which is measured in percent with reference to the final-value (Thomas, 2016). Mathematically expressed as:

$$\%OS = \left(\frac{y_{max} - y_{ss}}{y_{ss}}\right) \times 100$$

Equation 2: Calculation of the overshoot percentage.

Where:

$\%OS$ ~ Overshoot percentage [%]

y_{ss} ~ steady-state value

y_{max} ~ Maximum output value

A control system's transfer function have a natural frequency: ω_n, and a certain damping ratio: ζ. These indicate how fast the system oscillates by nature, measured in rad/s. Whereas the damping ratio describes how fast these oscillations decay but also determines overshoot and settling time. For instance, if the transfer function of a system is known, in Laplace domain, with ***s*** being being the Laplace variable (Thomas, 2016):

$$H(s) = \frac{\omega_n}{S^2 + 4\zeta\omega_n s + \omega_n}$$

It is possible to calculate the peak time, t_p, which is the time at which the control system's output reaches its maximum overshoot, also called Peak Overshoot, measured in precentage, *PO*, both mathematically expressed as:

$$t_p = \frac{\pi}{\omega_n\sqrt{1-\zeta}}$$

Equation 3: Calculation of the peak time, measured in seconds.

Where:

t_p ~ peak time [s]

ω_n ~ Natural frequency component [rad/s]

ζ ~ Damping ratio [-]

Furthermore,

$$\%PO = e^{-\frac{\zeta\pi}{\sqrt{1-\zeta^2}}} \times 100$$

Equation 4: Calculation of Peak Overshoot.

Where:

$\%PO$ ~ Peak Overshoot [%]

ζ ~ Damping ratio [-]

In addition, for:

$\zeta > 1 \Rightarrow$ indication of overdamped, no overshoot which gives a slow response time.

$\zeta = 1 \Rightarrow$, indication of critical damp, no overshoot which gives the fastest possible settling time.

$0 < \zeta < 1 \Rightarrow$, indication of underdamped, here overshoot occurs.

$\zeta = 0 \Rightarrow$, no damping, gives continuous oscillations hence no settling at all.

It is also possible to calculate the damping ratio with the following mathematical equation:

$$\zeta = \frac{c}{2\sqrt{mk}}$$

Equation 5: Calculation of Damping ratio.

Where:

c ~ Damping coefficient [Ns/m]
ζ ~ Damping ratio [-]
m ~ total mass [kg]
k ~ stiffness [N/m]

With the given information a fine example of when overshoot is not of desire, and rather something that is extremely forbidden to occur would be for a pick-and-place robotic arm. If an overshoot occurs, the automatic movement might move too fast and beyond the desired location. This would then damage structures and even workers surrounding it. Hence it's very important to avoid overshoot in an automatic system (Thomas, 2016).

The amount of time it takes for the processed signal to settle within a certain percentage of the final value is called the Settling time. Commonly it's chosen to be 5%. And lastly, there is a certain error, denoted Steady-State Error, which is the final difference between the processed signal and the desired output value (Thomas, 2016).

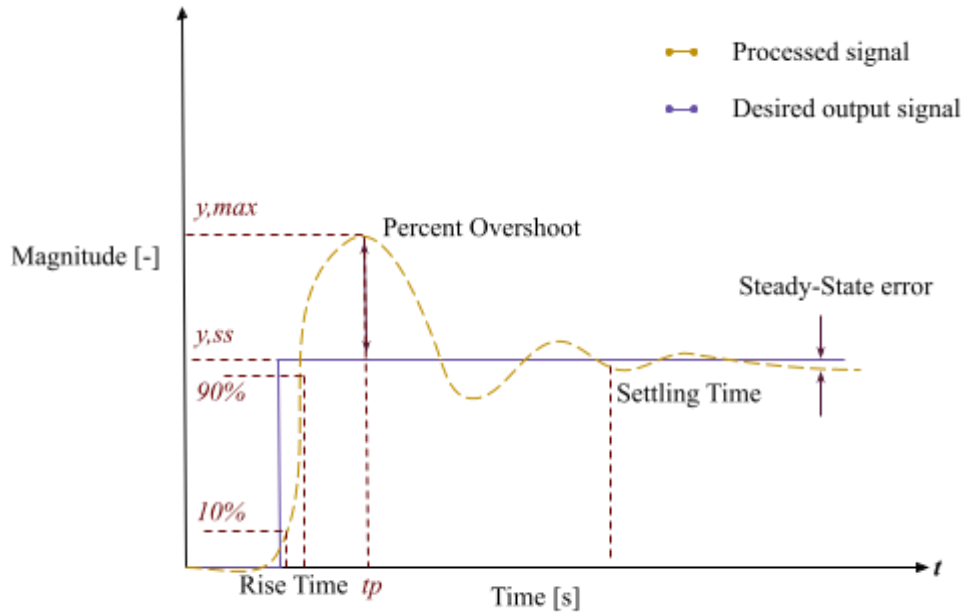

Figure 6: Usual response of a closed-loop control system with an implemented PID regulator. In addition, including known variables such as Rise Time, Peak time, Percent Overshoot, Settling Time, Steady-State error, Processed signal, Desired output signal and lastly, ymax and yss. Figure created by the authors.

2.1.2 PID Regulator

PID represents the most common control algorithm in today's manufacturing industry. Consisting of three segments; proportional, integral and derivative. All these 3 components are modified in order to achieve an efficient automatic control system. In addition, they're always used in a closed-loop control system. This is due to the PID controller relies on feedback information to compute corrections with the help of its three components (Thomas, 2016).

A block diagram of a PID implementation within a control system may appear as follows:

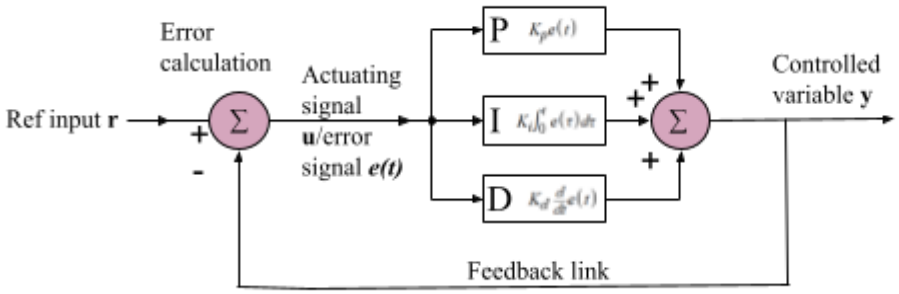

Figure 7: Block diagram of a control system including PID algorithm. Figure created by the authors.

The PID controller recieves input signals from sensors, further on calculating the difference between the real value and the desired signal. Subsequently adjusting its output signal(i.e voltage,current flow, temperature) with the help of its three components (Thomas, 2016).

The proportional component(P) solely relies on the calculated difference between the desired output signal and the signal being processed. The proportional constant gain K_p dictates how responsive the system is to the error signal. That is, it controls how rapid the response is to the reference signal. Since the increase in K_p leads to faster response, it also results in lesser stability margins because the gain crossover frequency shifts higher. The integral component(I) takes the difference in time between the start and when the error gets decreased to a low level and sums all the small points. This creates a combined controller with the P component. The derivative component(D) mimics the integral component, it takes the time difference but instead of summing it up it takes the slope of the curve and divides that by time difference. This part is mainly used to counter oscillations (Thomas, 2016).

2.1.3 Model analysis of the controller

There are 7 different movable joints on the robotic arm. Whereas each joint can be manuverd by either torque or current adjustment, this is called model-based control. Here, the current can be used to influence the proportional gain in order to make the system more precise, which makes the controller adapt to changes such as load or motion.

Below is an example of a single joint arm modelled based on torque and current.

The main equations that will model the controller are for a single joint.

$$J\frac{d\theta^2}{dt^2} + b\frac{d\theta}{dt} = T$$

These parameters model the torque with an added friction term which is proportional to speed.

$$T = Kr * I_a$$

The torque is proportional to the current hence the equation above.

$$L_m * \frac{dI_a}{dt} + R_m * I_a = u$$

)These model the losses of the motor one inductive part and one resistance part.

However, expanding this idea to the real would include many more parameters to be incorporated, below is an image of a simplistic arm with gripper detached (not a joint that is rotating) so 6 joints are present with 5 segments.

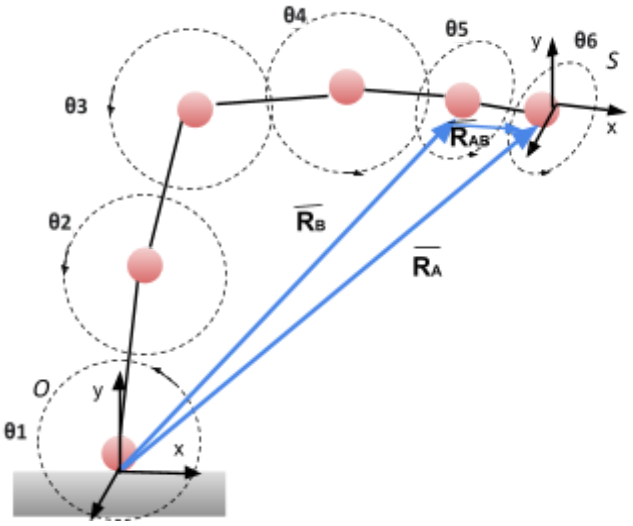

Figure 8 A rigid a 6 joint arm with base KS and end effector KS and a relative vector between Rb and Ra - used in applications such reference frames- and different angular notation for each rotation, figure made by authors.

The forward kinematics properties of arm determines the end effector position relative to the base frame which is at the very start of the first joint. The end-effector is the last joint of the sketch and is supposed to be the gripper but since this is detached in the sketch one could visualize it as at the very end of the joint 6. The forward kinematics of the end effector consists of all matrices of every single joint up to that joint after getting multiplied together incorporating the length between them. After some manipulations one gets a state space vectors of w [rad/s] and v [m /s] which can be used in a **Jacobian matrix**.

The main idea about the Jacobian matrix is to **transform a point in 3d space to a rotation** and be able to go **back from the rotation to translation** by transposing it. This is highly useful since this function can be mapped to other things, such as find a relationship between torque (force ∟ with length rotating around a point), moment of interia and angular acceleration (slope of angular velocity) by Eulers formula. This a less tedious way to determine the torque of the end effector torque because it needs additional parameters such rotation direction

and ∟ force applied for each joint which are otherwise difficult to measure using sensors. The Jacobian matrix is most likely used in **all** the inbuilt **controllers** in **interbotix source code**l.

2.2 Signal processing

2.2.1 Filtering

Filters allow signals with certain frequencies to pass whilst attenuating others. This is usually of desire when one would like to remove unwanted noise. Which refers to unwanted interference that can distort a signal or reduce the system performance. For instance, in audio systems, noise causes hissing sounds which prevents the user from hearing the audio. Furthermore it can cause inaccurate measurements from a sensor whilst trying to calculate a certain distance which can cause an overshoot or undershoot for a control system (Adams, 2020). There are 4 ideal filter types that are mostly used, these are: Low-pass, high-pass, bandpass and bandstop. This means that they aren't correct in reality but are sufficient. For instance, an ideal filter requires infinite impulse responses such that they satisfy:

$$h(t) \geq 1 \quad \text{for all t between} - \infty \; and \; \infty.$$

Equation 6: Ideal filter requirement.

This means that these filters would be non causal. Which means that its output depends on future inputs. This makes it impossible to implement in real-time systems. However, non-ideal filters help us achieve these ideal filter types characteristics. Which are real-world approximations of ideal filters, with some limitations. For instance, non-ideal filters have a finite roll-off which may allow some unwanted frequencies to still pass. They may also introduce certain phase shifts that can distort signals. In addition, they are dependent on electrical components, such as resistors, capacitors and inductors. These components have a certain tolerance and losses which affects the performance. Lastly, Ideal filters completely block unwanted frequencies but the non-ideal filters only attenuate them to a certain level (Adams, 2020).

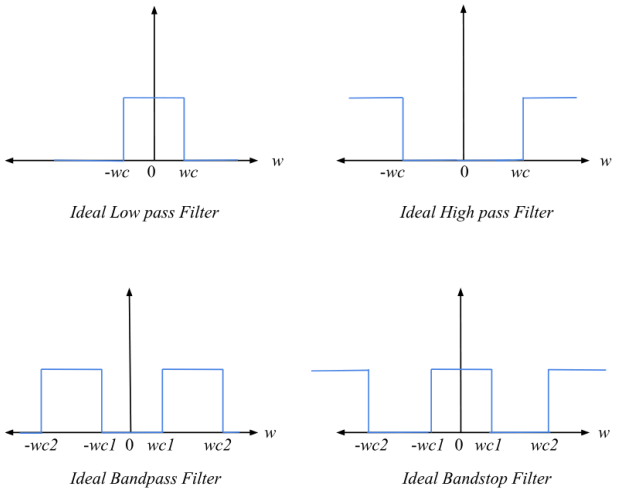

Figure 8: 4 graphs consisting of Ideal Low pass Filter, Ideal High pass Filter, Ideal Bandpass Filter and an Ideal Bandstop Filter, where the horizontal axis is measured in angular frequency. Figure created by the authors.

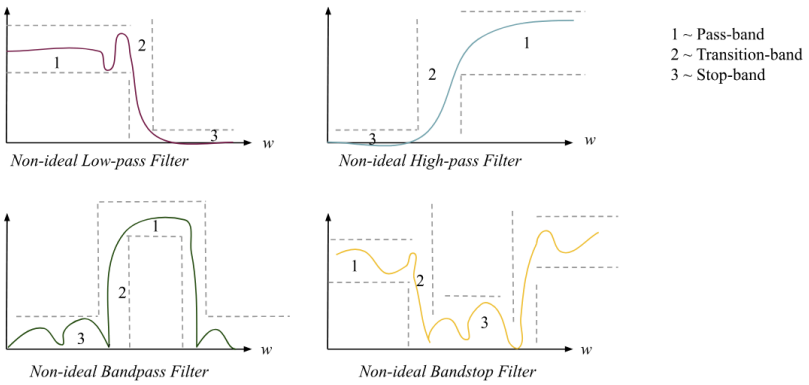

Figure 9: 4 graphs consisting of Non-Ideal Low pass Filter, Non-Ideal High pass Filter, Non-Ideal Bandpass Filter and a Non-Ideal Bandstop Filter, where the horizontal axis is measured in angular frequency. In addition, the Pass-band-Transition-band and the Stop-band have been marked. Figure created by the authors.

2.2.2 PWM

Pulse-width modulation(PWM) contributes to obtaining analog results with digital methodologies. That is, using a digital control to create a square wave by having a signal switched between on and off(1 or 0 in amplitude). This “on” or “off” pattern will simulate quantities, such as voltage or current, in between its full magnitude towards almost none, by adjusting the amount of the time the signal is “on” versus the time that the signal is “off”. Whereas the amount of "on time" is called the pulse width. That is, in order to achieve varying analog signals, you modulate the pulse width, hence the name (Adams, 2020).

2.3 Software

2.3.1 Python

Python is a beginner friendly programming language known for its simplicity, compared with other languages such as Assembly, C++ and so forth. It is widely used in many fields, such as web development, data science, artificial intelligence, robotics, control systems, etc (Python Software Foundation, n.d.)

2.3.2 PyRobot

PyRobot is an open-source Python library which provides a pre-made package. This library includes different classes, functions, and interfaces for robotic control, making it easier to begin working with different robotic tasks without the need of re-inventing basic concepts (Python Software Foundation, n.d.).

2.3.3 YamL

YAML(Yet Another Markup Language), unlike Python, is a data serialization language, and it is used mostly for writing configuration files (Ben-Kiki et al, n.d.)

2.3.4 Xacro

Xacro(XML Macros) is an XML macro language, it helps one to construct more readable XML files by using macros that expand to larger XML expressions (Open Source Robotics Foundation, n.d.).

2.3.5 XML

XML(Extensible Markup Language) is a programming language that stores and transmits data. It's characterized by its data being encapsulated(<build_depend>rviz</build_depend>), its hierarchical structure upwards going downwards, where the data is denoted parents and child elements. However, an XML file doesn't contain any data that performs any actions, it only contains data. Hence, in ROS, XML is used for configuration files, such as launch files(these define how ROS nodes and parameters initiates and how they're linked together) or URDF files(these describe the robot's kinematic characteristics, joints, links, sensors etc…). In conclusion, files in ROS that end with .launch, .urdf and .xacro are different shapes of XML (Microsoft, n.d.).

2.3.6 URDF

As mentioned earlier, URDF(Unified Robot Description Format) is a specification that uses XML syntax to describe a robot. That is, it's an XML file but it follows rules, attributes and elements that are set by URDF specifications in order to describe the robots joints,links, geometry and physical attributes (Microsoft, n.d.).

2.3.7 Rviz

Rviz(ROS visualization) is a useful three dimensional tool included within ROS that helps the user to visualize the robot digitally, as it would be in reality. These can be the robot's sensors, its surrounding environment, obstacles around it and the data the robot is working with. Therefore using Rviz helps the user to simulate or debug software to prevent damaging the arm in the real world (Open Source Robotics Foundation, n.d.). However, it is limited since there is no physics engine applied, unlike in Gazebo.

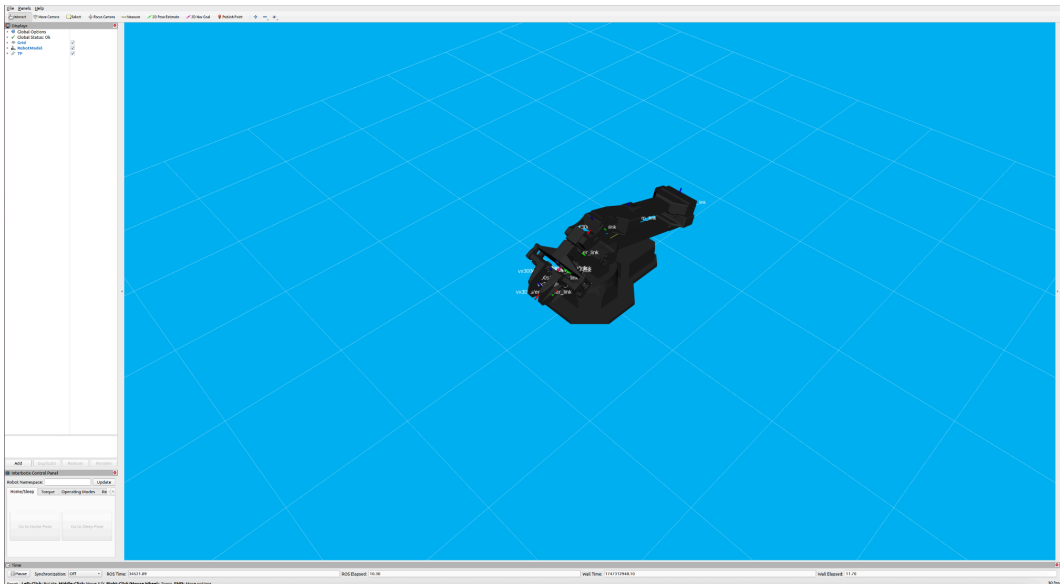

Figure 10: Simulation of the ViperX 300S in Rviz, including its joints, surrounding area and different interaction functions with the arm. Figure created by the authors.

2.3.8 Gazebo

Similar to Rviz, Gazebo is a three dimensional simulator for robots in ROS but with a built in physical engine. This engine doesn't neglect gravitation, inertia, friction, collisions or dynamics whilst simulating a robot. Which contributes to a more realistic behaviour in the simulation. To illustrate, ROS nodes can be used to maneuver the robot in the simulation, namely sending torque, velocity, current or position commands, whereas these ROS nodes can read that data from the simulated sensors as if they were real sensors, which are called JointState messages (Open Source Robotics Foundation, n.d.).

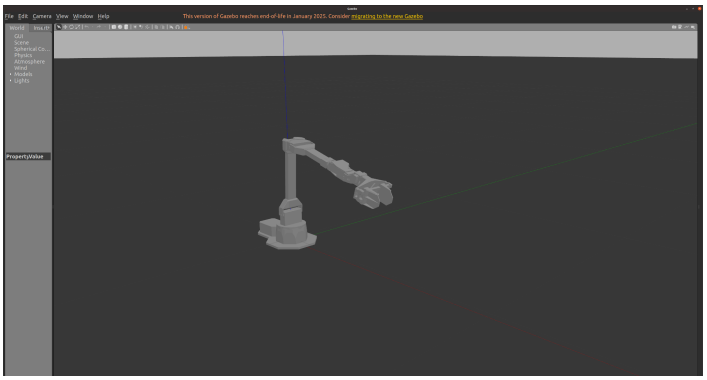

Figure 11: Simulation of the ViperX 300S in Gazebo, including its joints, physical forces and certain built-in functions to maneuver the arm. Figure created by the authors.

2.3.9 Matlab

Matlab is another programming language but also a large proportion of a computing environment. Here you can simulate a control system, plot corresponding graphs and analyze the results. It is widely used for numerical computing in engineering fields (MathWorks, n.d.)

2.3.10 Reinforcement Learning Algorithm

The Reinforcement Learning algorithm teaches the robotic arm how to make decisions by interacting with an environment. The arm performs actions, receives feedback in the form of something positive or negative, and improves its strategy over time, that is, relying on trial and error to discover optimal behavior. This algorithm has several components, at first, an agent, which makes decisions and performs actions. Further on we have the environment, the surrounding area the agent interacts with. This environment helps the agent to improve by providing obstacles it can learn from. There are also states, these represent the current situation of the agent's location within the environment (Sutton & Barto, 2018).

Action, a choice the agent makes to influence its environment. This action will be given a Reward, a signal that tells the agent if its decision was something positive or negative. There are certain Policies, which describe several strategies that link states to actions. That is, the agent follows a policy to decide which actions to perform in a given state. Value Functions estimates how good it is to be in a particular state by predicting the expected total reward from that state onward. The Q-Function estimates the expected reward for performing a specific action in a given state (Sutton & Barto, 2018).

With all this in mind, implementing the algorithm it may come across as: Agent starts in an initial state($t=0$) and selects an action based on a policy. The environment then responds by relocating to a new state and provides a

reward. This makes the agent update its policy and this may repeat until the agent has learnt an effective strategy (Sutton & Barto, 2018).

2.3.11 Model Predictive Control Algorithm

The Model Predictive Control algorithm mimics the Reinforcement Learning algorithm, but it uses a model of the system to predict future actions whilst having to fulfill system constraints. This algorithm will teach the robotic arm the most efficient way to pick up an object without deforming it. Which is achieved by using a mathematical model of the system to predict future states (MathWorks, n.d.)

2.3.12 ROS Noetic Ninjemys -framework & conventions

ROS stands for **R**obot **O**perating **S**ystem and contains a set of conventions, frameworks and libraries that allow a direct communication between the robotic arm and the computer controlling it. Algorithms developed within this framework can interact with each other, which create a complex structure. **Nodes, topics, message, roscore, publisher** and **subscriber** are some components that make up the ROS framework.

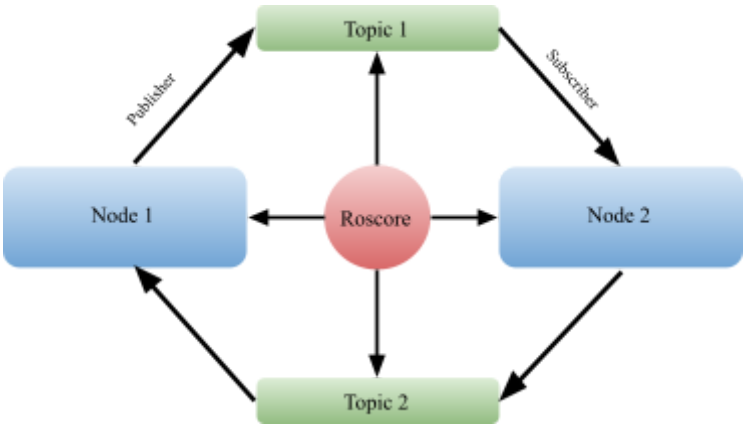

Figure 12: A Simple communication path between two nodes. Figure created by the authors.

..

Table 1: Basic terminology of ROS

Terminology	**Description**
Node	Performs a task e.g. reading sensor value
Topic	Listening to a task or value
Message	An information set
Roscore	Underlying registration
Publisher	Posting messages
Subscriber	Reading messages

1. A node as a single executable file that performs some kind of computation or task. It can for example read a sensor value which sends the information further.
2. This information could be sent to a subscriber which listens to the data that can further transmit the information to a topic.
3. Topics can be seen as a bus and that has no memory of the process, the only way it communicates is to pass information in one direction.
4. A topic can pass a message further to a node (see figure 12). However, if there is a need to communicate back the information, it must be sent through another topic. This creates two "gates" that are open in one direction but closed in the other.
5. The underlying registration of all the parts is the roscore.
6. The ROS publisher is responsible for sending the messages to a topic, so every data that is sent forth must go through it and every sensor value that requires readings must go through the subscriber.

2.3.13 Joint locations and names

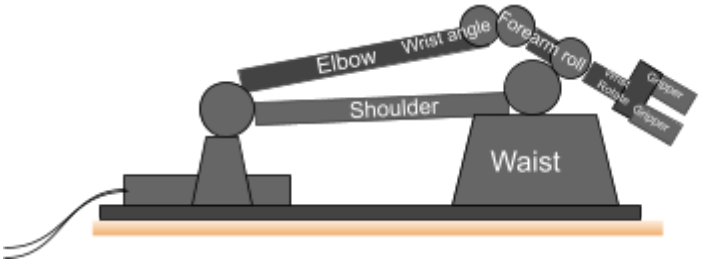

Figure 13: Illustration of the robotic arm Viperx 300s, including its joint locations and their notation. *Figure created by the authors.*

Table 2: Containing ID's, names and servomotor in the order from the base to the end-effector

Servomotor	**ID**	**Name**
xm540-270	1	waist
xm540-270	2	shoulder
xm540-270	3	shoulder_shadow
xm540-270	4	elbow
xm540-270	5	elbow_shawdow
xm540-270	6	forarm_roll
xm540-270	7	wrist_angle
xm430-w350	**8**	wrist_rotate
xm430-w350	**9**	end-effector or gripper

Note: *Shadow names are not present on the actual arm, so there are only 7 joints in total. The shadow ID's gets "skipped" so the last joint becomes 7. Also, the gripper will be interchangeably called left-finger and right-finger, the reader should be aware of this, so no confusion occurs in the implementation and results part.*

Chapter III

Implementation

"A thing constructed can only be loved after it is constructed; but a thing created is loved before it exists."
- Charles Dickens, Our Mutual Friend

3.1 Start of the project

At the beginning of the project, initial research was conducted. Firstly, core principles of C++, Python, XML, signal processing and regulator theory had to be studied. This was done due to this knowledge being required to complete the project. Furthermore, there was a delay in delivery for the robotic arm which gave the group an opportunity to explore ROS. Here it was quickly grasped that the operating system Windows couldn't satisfy the requirements to run ROS which forced the group to change to Ubuntu 20.04(a certain Linux distribution).

3.2 Installing Linux

There are many ways to change the operating system on a computer. For this project, VirtualBox was used which is a free and open source virtualization software developed by Oracle. Which allows the user to run one or more operating systems within the current operating system on a computer (Go, 2022).

3.3 Installing Ros Noetic Ninjemys

With Ubuntu 20.04 on VirtualBox the next step was to load Ros Noetic Ninjemys. Firstly, a terminal would be opened by pressing `Ctrl+Alt+T`. This text-based interface allowed the user to interact with the operating system, mainly to perform certain tasks, in our case, downloading ROS. Hence, to download ROS, you would have to run certain commands subsequently as listed below. But, before jumping into the navigation, it's very important to clarify that one may consider Ubuntu computer's software manager (denoted apt) like a customer. This customer transits to specific stores (called repositories) to download software packages (Go, 2022).

As a starting point, you would have to run the following command, which does the crucial first step of letting your system know where to look for ROS: `sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'` ~ "Sudo" runs the following command with administrator priviliges, "sh -c '...'" runs a whole command string as the administrator, "echo" prints text on the terminal whereas the "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" is a specific line of code that's being printed(the address of the ROS software for our version of Ubuntu). In addition, the "$(lsb_release -sc)" solves the codename for our Ubuntu version(in our case Ros Noetic) and puts it into the address. Furthermore we have the ">" symbol which can be translated into "redirect the output to", hence, instead of printing the address to our terminal, it sends it into a file. Lastly, the "/etc/apt/sources.list.d/ros-latest.list" is a specific file where Ubuntu's Apt looks for any extra repositories (Go, 2022).

2: `sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654` ~ This command gives the user a digital security stamp (referred to as a key) from an authorized place (a "keyserver"). The "apt-key adv ... --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654" tells the software to download a certain key with the identification of the long text at the end. In addition, the "--keyserver 'hkp://keyserver.ubuntu.com:80'" tells the command its online location to get the key. In conclusion, repositories authenticate their software packages. Which forces your software to acquire the corresponding key to verify that the security stamp is authentic and that the software package hasn't been modified since the latest release. That is, after telling your software where the ROS store is, the command aquires the key needed to trust the ROS items that's meant to be downloaded, essentialy a security step to ensure its genuine ROS software (Go, 2022).

3: `sudo apt update` ~ This command orders your software manager to target all known repositories in your system, download the latest list of available packages and their versions.

4: `sudo apt install -y ros-noetic-desktop-full` ~ The "apt install" command tells the software manager to install a specified package, in this case, "ros-noetic-desktop-full". This package itself doesn't contain much but pulls in all the other packages needed for a complete ROS Noetic installation. Whereas "-y" orders the apt to answer "yes" making the installation execute without having to type 'y' and press Enter. This is due to the fact that the Apt asks the user to confirm if they want to download and install the packages, including their dependencies (Go, 2022).

5: `echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc`
`source ~/.bashrc` ~ Here, “"/opt/ros/noetic/setup.bash"” is a certain file that came with the ROS installation. It contains instructions that tell the software where the ROS files are located. Running this file in the terminal initiates that terminal to handle ROS commands. In addition, the “source” command executes a file's commands within the current shell which activates the ROS environment whereas the “>>” simply appends the output to the end of the file. In conclusion, line number 5 adds the command source “/opt/ros/noetic/setup.bash” to the end of the .bashrc file. This allows the ROS environment to automatically load whenever a new terminal is opened, and in that terminal, ROS commands are then automatically implemented (Go, 2022).

6: `sudo apt install -y python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential` ~ This command creates a development environment for **ROS**. It installs several Python 3 tools(rosdep, rosinstall, rosinstall-generator and wstool) (Go, 2022).

7: `sudo rosdep init`
`rosdep update` ~ After installing python3-rosdep, “sudo rosdep init” initializes a configuration file that allows rosdep to retrieve information of all dependency files to be installed. Rosdep update refreshes and updates the installation.

8: `mkdir ~/catkin_ws/src”` using the command mkdir creates a new folder and “`~/catkin_ws/src` “ specifies which directory the folder should be in. `cd ~/catkin_ws/` The “cd” command stands for “change directory and can be used to navigate to a specific path. It moves up a directory “`~/catkin_ws/`” and by using `catkin_make` ~ it starts building the program.
9: `echo "source $(pwd)/devel/setup.bash" >> ~/.bashrc`
`source ~/.bashrc` ~ This source command activates the active terminal as the primary window (Go, 2022).

3.4 Assembling the robotic arm

Most of the robotic arm came assembled in its package, aside from a few components: grippers, cushions for the grippers and an arm tag which was meant to be mounted on top of the arm. In addition, power supply cables were included to establish connection between the hardware and the software. Lastly, a wireless playstation controller was included but the group had no intention of its usage. In the package a certain kit also came, which Trossen robotics calls their “vision kit”. This kit included a camera, a certain tag which the camera is meant to track, cables to power the camera, a camera stand and some cubes for the arm to pick and place.

Figure 14: Picture of the vision kit included in the purchase order from Trossen robotics, 6 colored cubes, two power cables, an emoji tag for the camera to track, 5 screws and a camera stand to help the camera being placed above the robotic arm (Trossen Robotics, n.d.).

A wooden base plate for the waist was mounted under the arm to prevent gravitational force from pulling down the arm during operation. Further on, screw-holes were made in an allowed work environment provided by Pekka Alakulju in order to mount the arm on the wooden base plate by using 6 x M3 screws holding the arm down.

Figure 15: Visual representation of the wooden base plate combined with the robotic arm. Figure created by the authors.

3.5. Position based PID with trajectory control using effort in Gazebo

Although the controller exists, it is not natively integrated in Gazebo so some launch and yaml files needed to be modified in order to use it. In the yaml file the effort trajectory controller had to be added with specific PID gains. Also the ros_controller had to be updated in order to load the controllers using the controller manager interface. These changed things are not enough to make it work properly, inside the file system there exist a file called robot_model.urdf.xarco. This could be described as a description for the robot like arm position, link connections and dimensions just to name a few. The XARCO file had to be changed in order to make it include <transmission> tags to make the EffortJointInterface activated. This interface allowed for the controller to communicate and respond to commands during a simulated process. Additionally, a python code had to be created in order to test and validate the changed filesystem. Below is a code snippet of the change in the xarco file.

```
  <xacro:if value="${urdf_loc != ''}">
    <xacro:include filename="${urdf_loc}"/>
  </xacro:if>

  <xacro:property name="base_link_full" value="$(arg
robot_name)/$(arg base_link_frame)"/>
  <gazebo reference="${base_link_full}">
    <plugin name="gazebo_ros_control"
filename="libgazebo_ros_control.so">
      <robotNamespace>/$(arg robot_name)</robotNamespace>
    </plugin>
  </gazebo>

</robot>
```

Figure 16 : Figure created by the authors.

3.7 Developing the filesystem

3.7.1 Description file

```
interbotix_xsarm_descriptions)/rviz/xsarm_description.rviz" />

  <arg name="controllers_config" default="$(find
interbotix_xsarm_descriptions)/config/vx300s_controllers.yaml"/>




  <param name="robot_description" command="$(find xacro)/xacro '$(find
interbotix_xsarm_descriptions)/urdf/$(arg robot_model).urdf.xacro'
robot_name:=$(arg robot_name) base_link_frame:=$(arg base_link_frame)
show_ar_tag:=$(arg show_ar_tag) show_gripper_bar:=$(arg show_gripper_bar)
show_gripper_fingers:=$(arg show_gripper_fingers) use_world_frame:=$(arg
use_world_frame) external_urdf_loc:=$(arg external_urdf_loc) use_gazebo:=$(arg
use_gazebo) load_gazebo_configs:=$(arg load_gazebo_configs)"/>
```

Figure 17: Figure created by the authors.

The <param> name was modified to include all arguments defined in the Xacro file. The launch file uses these parameters and reads the Xacro file with the specified arguments.The param command calls the description launcher which loads the robot URDF model, name and base_link frame.

3.7.2 Ros_control launch file

```
<node
name="xs_hardware_interface"
pkg="interbotix_xs_ros_control"
type="xs_hardware_interface"
output="screen"
ns="$(arg robot_name)">
</node>

<node name="controller_spawner"
pkg="controller_manager"
type="spawner"
respawn="false"
output="screen"
args="joint_state_controller joint_trajectory_controller"/>

</group>
```

Figure 18: Figure created by the authors.

The two node was added in the ros_control launcher this was because it need to incorporate the hardware interface. The hardware interface is a library that declares every sensor value and it can either read or write meaning it can bridge between the controller and the robot The node needs to contains specific parameters like name, pkg, type and ns. Note that a node needs to start with **<node**.

3.7.3 Vx300s_controllers.yaml

```yaml
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50


joint_trajectory_controller:
  type: position_controllers/JointTrajectoryController
  joints:
       - waist
       - shoulder
       - elbow
       - forearm_roll
       - wrist_angle
       - wrist_rotate
       - left_finger
       - right_finger


  command_timeout: 0.5
  state_publish_rate: 50
  action_monitor_rate: 30


    gains:
       waist:         {p: 5.0, i: 0.0, d: 0.1}
       shoulder:      {p: 5.0, i: 0.0, d: 0.1}
       elbow:         {p: 5.0, i: 0.0, d: 0.1}
       forearm_roll:  {p: 5.0, i: 0.0, d: 0.1}
       wrist_angle:   {p: 5.0, i: 0.0, d: 0.1}
       wrist_rotate:  {p: 5.0, i: 0.0, d: 0.1}
       left_finger:   {p: 5.0, i: 0.0, d: 0.1}
       right_finger:  {p: 5.0, i: 0.0, d: 0.1}
```

Figure 19: Figure created by the authors.

This was an unnecessary modification beside for the joint_state_publisher which enables the joint_state controller, however since the code was based on the Gazebo directory and was kept for further developments. The `joint_trajectory_controller` is necessary in the simulation, and it sets the gains for PID, refresh rates and publish rate (meaning how much per second it can publish).

3.7.4 PID regulator in Gazebo

```
def _trajectory(start, goal, move_t, hold_t, names):
   postion0 = JointTrajectoryPoint(
       positions=[start.get(j, 0.0) for j in joint_names],
       time_from_start=rospy.Duration(0.0))
   position1 = JointTrajectoryPoint(
       positions=[goal.get(j, 0.0) for j in joint_names],
       time_from_start=rospy.Duration(move_t))
   position3 = JointTrajectoryPoint(positions=[goal.get(j, 0.0) for j in names],
       time_from_start=rospy.Duration(move_t + hold_t))
   return JointTrajectory(joint_names=names, points=[position0, position1, position2],
                          header=rospy.Header(stamp=rospy.Time.now()))

def convert_pose(pose):
   converted = {}
   for k in pose:
       if k == "gripper":
           val = pose[k]
           converted["left_finger"] = 0.03 if val > 1.0 else 0.0
           converted["right_finger"] = -converted["left_finger"]
       else:
           converted[k] = math.radians(pose[k])
   return converted
```

Figure 20: Figure created by the authors.

The trajectory controller gets its parameters from the `start` position the `goal` position which are preset see table 3, move time which was set to 3s and then the hold time of 1s after it has reached its position. Then `postion0` sets the reference frame for the arm, to act as the base coordinate system. Position 1 is when it hits it's target position, `position1`.Then it holds that `position2` for 1s as mentioned earlier. Then it returns the trajectory path with time stamps that syncs with the time in gazebo.

```
def main():
   rospy.init_node("vx300s_move_to_points")
   joint_state_topic  = "/joint_states"
   action_toward_target_topic = "/joint_trajectory_controller/follow_joint_trajectory"
   rospy.Subscriber(joint_state_topic, JointState, _js_cb)
   arm_cli = actionlib.SimpleActionClient(act_topic, FollowJointTrajectoryAction)
   rospy.loginfo(f"Väntar på rörelse {act_topic} ...")
   client.wait_for_server()

   rospy.loginfo("rörelse på gång ...")
   while not rospy.is_shutdown() and _last_jointstate is None:
       rospy.sleep(0.1)
```

Figure 21: Figure created by the authors.

Because the controller uses a class it needs to have main() function which initalizes a node `vx300s_move_to_points` then uses the topics to publish the `topics` like effort and position. The `rospy.Subscriber` reads the values from the `topics` which then builds the `client` which can can communicate with the controller. The client uses the information to form a trajectory path based on effort. The last part `rospy.loginfo("rörelse på gång ...")` just logs the data and the last line acts as safety for the movement.

3.8 Simulation

3.8.1 Objective and simulation environment

The primary objective was to design and implement an effort- or torque- based PID controller with adjustable parameters for each joint. The controller implemented an algorithm by using effort to follow a trajectory to reach its target position. In order to safely verify the different algorithms, the program Gazebo was used.

3.8.2 Position based PID using current on the actual arm in the real-world

Gazebo had a separate launch file and was not exactly structured the way that the control_launch file (not the ros_control launch file). So translating a launch from gazebo to control launch file was not obvious and much trail and error was made to make this file launch correctly. Since the changed XARCO file only worked with gazebo, this created a cascade of changes in both the description launch file and the mode yaml file. Attempts were made to establish a connection between the hardwareinterface allowing direct communication between the robot arm and the user. Due to the hardwareinterface not being able to support the trajectory effort controller directly (but could be bypassed i gazebo) it wasn't ideal solution. So instead a current based PID was constructed based on the inbuilt mode.yaml configurations. By changing the state of motors states to PWM and also replace position PID with current to position.

There was never python code made for this controller but underlying modes and configurations were set so it works when publishing positions. Also, a vx300s_controller.yaml was added, so one could change gain parameters based on the desired behavior of the arm.

Chapter IV

Results

“All roads lead to Rome”
- Alain de Lille, “mille vie ducunt hominem per secula Romam”

4.1 Limitations

As mentioned earlier, halfway through the project it was decided to scratch the main idea, a pick and place methodology whilst avoiding moving obstacles. Including two different algorithms competing against each other. This was due to the realization that achieving such tasks required more time and knowledge. In addition, during the project, cables were damaged due to rapid movement of the arm. This happened due to faulty software which made the group realise that the arm was very fragile, which also contributed to the idea of scratching the main idea. Hence the camera above the robotic arm was disassembled, the whole development of the pick and place software was canceled, the hindrance zone never began construction nor did the algorithms start to get developed.

4.2 Was the robotic arm able to arrive at its destination without collisions?

When launching the simulation programs and initiating the software scripts the robotic arm managed to smootlhy follow the trajectory path from starting point to destination of desire in the real-world and in the simulations. Since only the “`forearm_roll`” was tested in the real world there was little data to display. But it proved the system was functional, such as commands being published to the joint nodes, which caused movement. Therefore, it can be said that both controllers worked, effort-based in Gazebo and the real world controller.

4.3 Which algorithm had the highest quality?

The algorithm initially used with the tuning script caused oscillations, instability, high steady-state error and high relative errors. Figure 22 illustrates how the old tuning algorithm performed. Furthermore, Comparing Figure 23 with Figure 22, a difference can be seen in both stability and the relative error. It is however important that one should keep in mind that these are not exactly the same position. Lastly, Figure 23 has a smoother slope which tells it had a better optimization, also, one could see oscillations at the beginning but also at the end of the travel path. With this in mind, it was concluded by the group that the 2nd algorithm, used in Figure 23, performed better than the first when comparing the two different graphs.

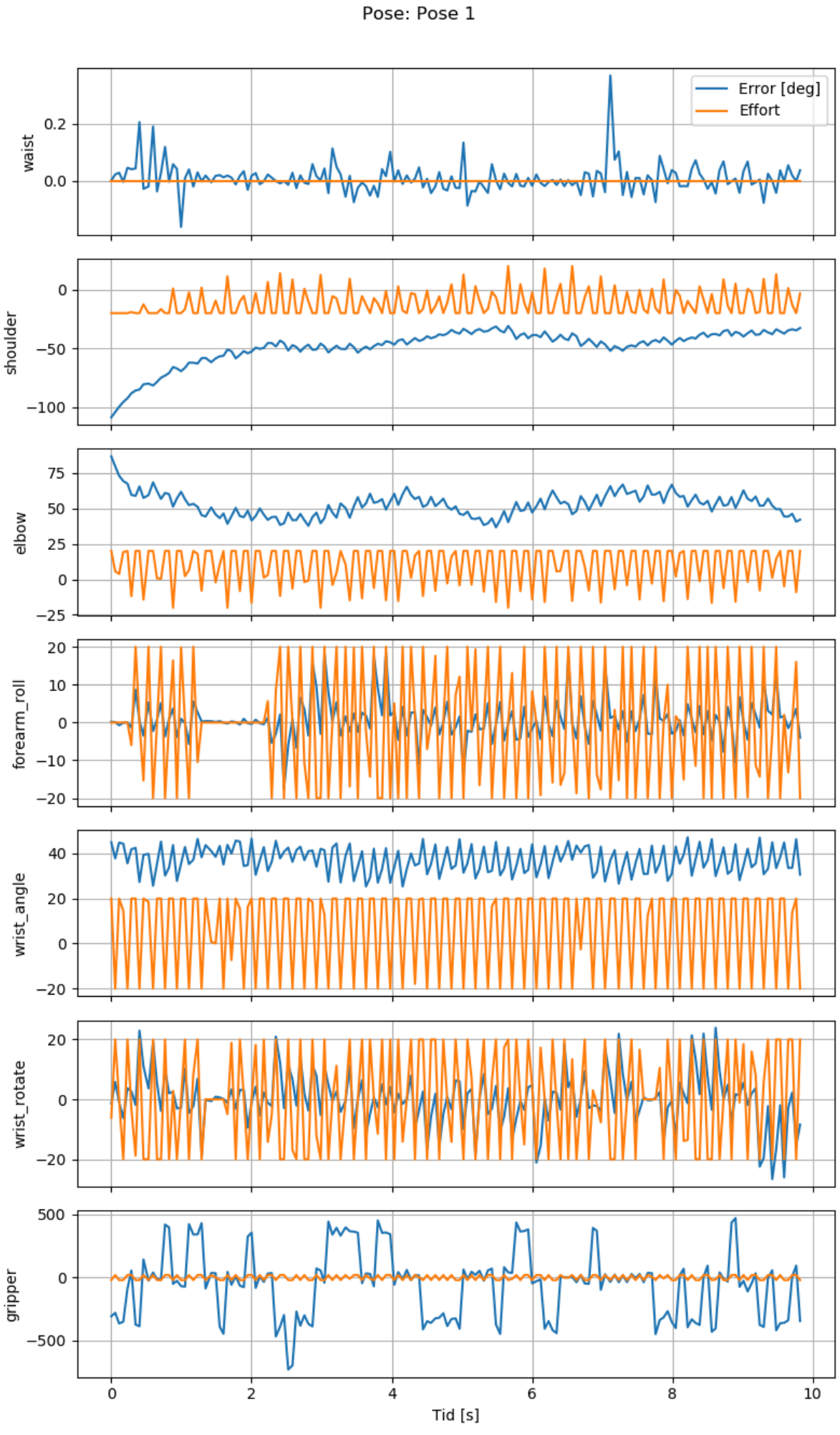
Pose: Pose 1
Error [deg]
Effort
waist
shoulder
elbow
forearm_roll
wrist_angle
wrist_rotate
gripper
Tid [s]

Figure 22:Position, error and effort in the gazebo simulation for the first tuning algorithm script, whereas the effort is displayed in an orange line, the error is displayed in a blue line. Here there are many oscillations at every point. Figure created by the authors.

4.4 Which regulator was most suitable?

Upon comparing both regulators, the model predictive control regulator proved to be the most suitable. This decision was made by analyzing the performance of each regulator, particularly in terms of smoothness, best stability, less overshoot, and suitable settling time, as illustrated in Figure 6, further on comparing each of the figures below and their data. The model predictive control regulator demonstrated better control in maintaining a smooth and stable trajectory, with minimal overshoot and steady-state error (Figure 29) compared to the PID regulator. While our PID controller showed faster response time, its overshoot (Figure 22), which caused oscillations, made it less ideal for tasks requiring precision, and certainly something not of desire since having an overshoot might cause collisions in the real world, damaging the robotic arm and even the object its supposed to transfer.

4.5 Simulation data for the Gazebo effort based controller

In order for the robotic arm to have an effort-based controller, ROS topics and nodes provided essential parameters that had to be used in the script. These parameters included different degrees and positions for each of the seven joints, which are crucial for defining the arm's configuration and achieving desired poses in the simulated environment. As seen in Table 3, this table displays the specific angular values in radians for each of the seven joints across five different set poses within the Gazebo simulation. These joint positions serve as targets for the effort-based controller, informing the necessary torques to achieve the desired arms position, later making the controller continuously adjust the efforts to minimize the error between the current position and the position of desire, thereby enabling motion control of the ViperX 300 S in the simulated environment. Lastly, Table 5 contains each PID parameter for each joint , used in the software controller, which all affected how the PID regulator performed, as mentioned in the theory section 2.1.2.

Pose	Waist	Shoulder	Elbow	Foreamr_roll	Wrist_angle	Wrist_rotate	Gripper
1	0	-106	88,81	0	45,84	0	0
2	-85,94	-85,94	74,48	0	57,30	0	0
3	-85,94	-85,94	74,48	0	0	0	114,59
4	-85,94	-85,94	74,48	0	10	0	0
5	0	-106	88,81	0	45,84	0	0

Table 3: Parameters from 5 different poses for all 7 joints returned from ros nodes in degrees.

Time between each pose	Hold time for each pose
3 s	1 s

Table 4: Display of time between each pose and the amount of time the robotic arm stops for each pose, measured unit in seconds.

Joints	Kp	Ki	Kd
waist	100	0,01	10
shoulder	100	0,01	10
forearm_roll	100	0,01	10
wrist_angle	100	0,01	10
left_finger	100	0,01	10
right_finger	100	0,01	10

Table 5: Different PID parameters for each joint, 7 in total.

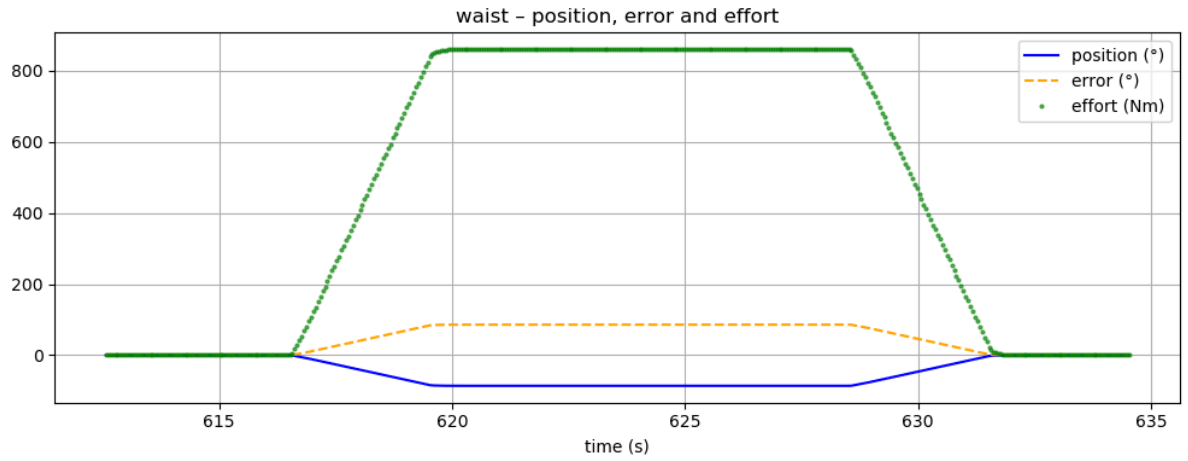

Figure 23 :waist's position, error and effort in the gazebo simulation for the 2nd tuning algorithm script, whereas the effort is displayed in a green dotted line, the error is displayed in an orange dotted line and the desired position is displayed in a flat non dotted line in blue. Figure created by the authors.

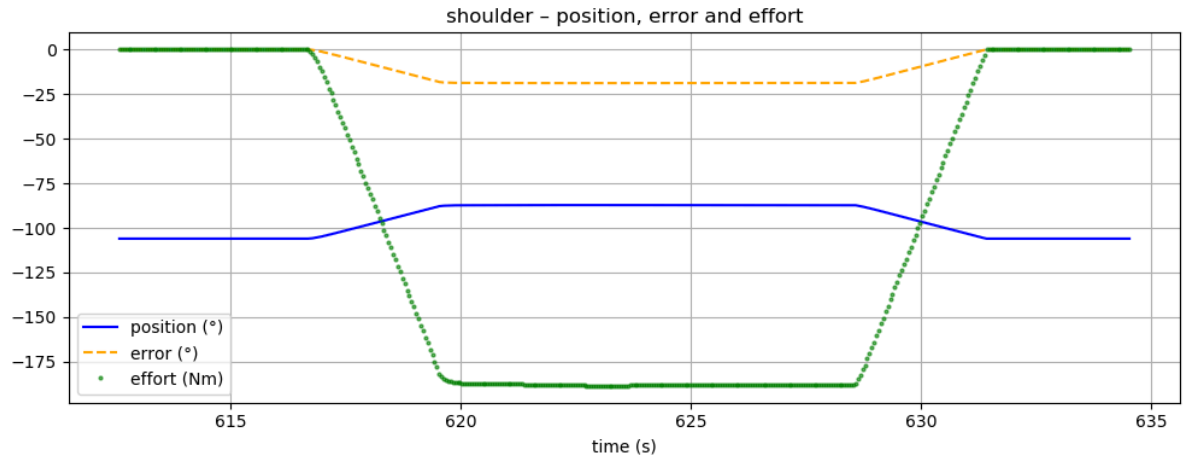

Figure 24: Shoulders position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Here there are no oscillations during the entire travel path. Figure created by the authors.

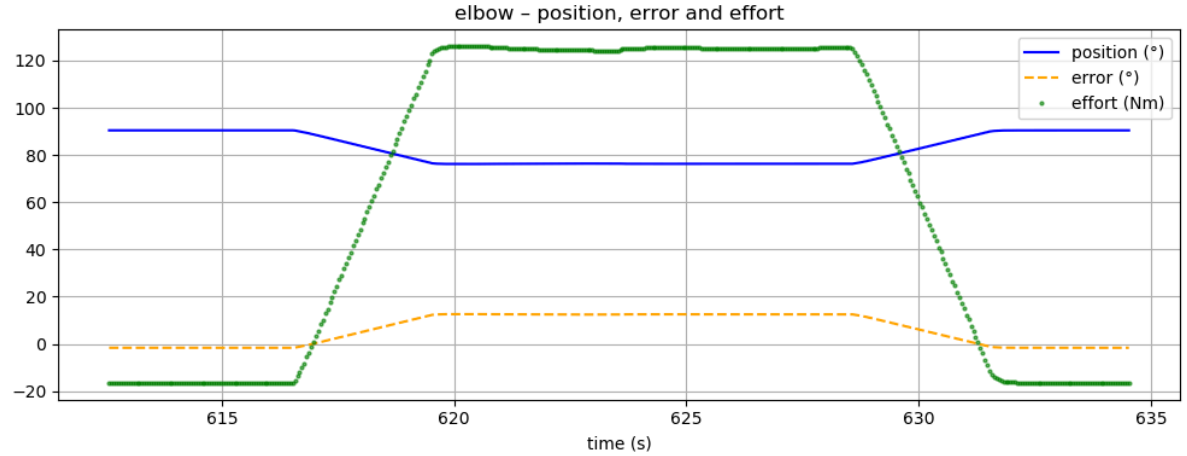

Figure 25: Elbow's position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Figure created by the authors.

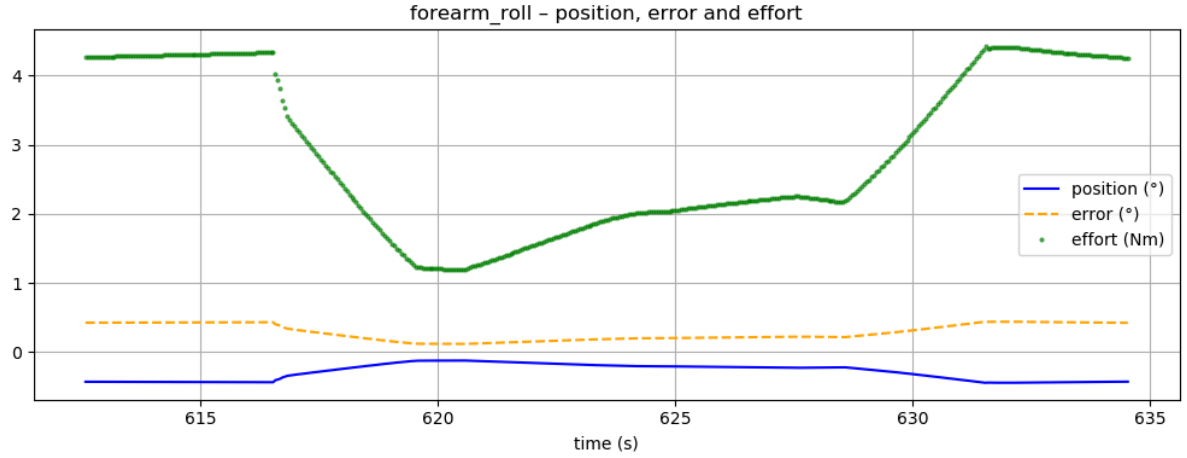

Figure 26: Forearm rolls position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Figure created by the authors.

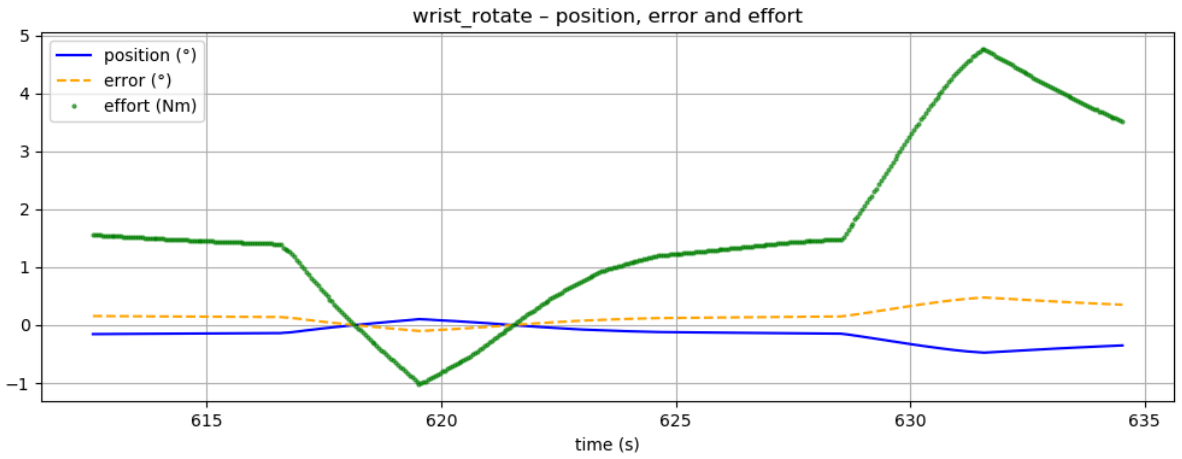

Figure 27: wrist_rotate position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Figure created by the authors.

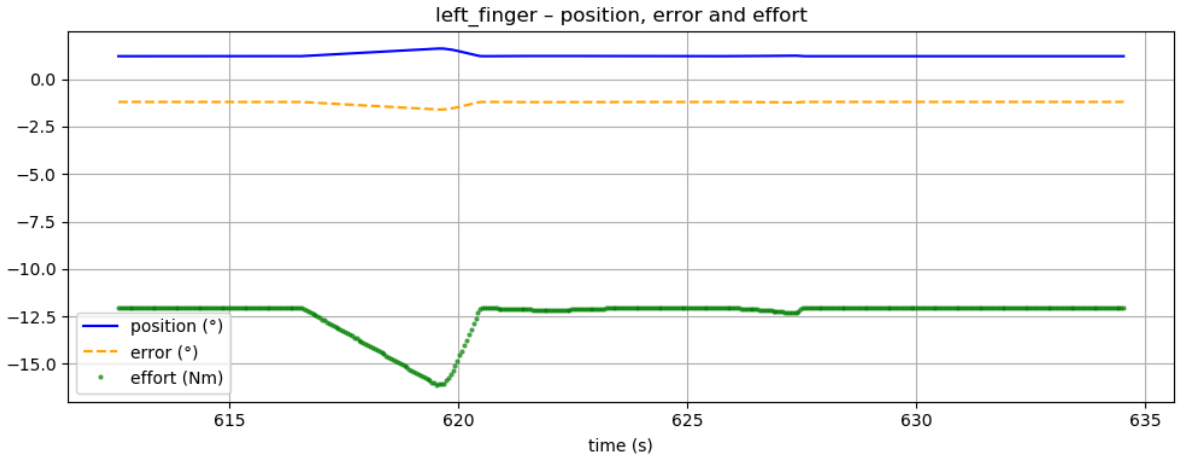

Figure 28: left_finger position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Figure created by the authors.

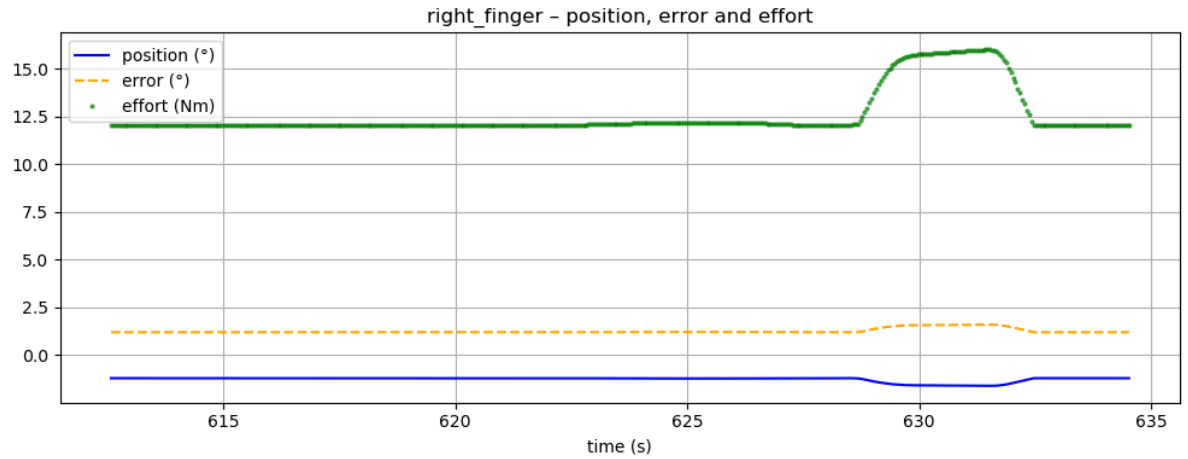

Figure 29: right_finger (part of the gripper) position, error and effort in the gazebo simulation. Where the error is displayed in an orange dotted line, effort is displayed in a green dotted line whereas the desired position is displayed in a blue non dotted line. Figure created by the authors.

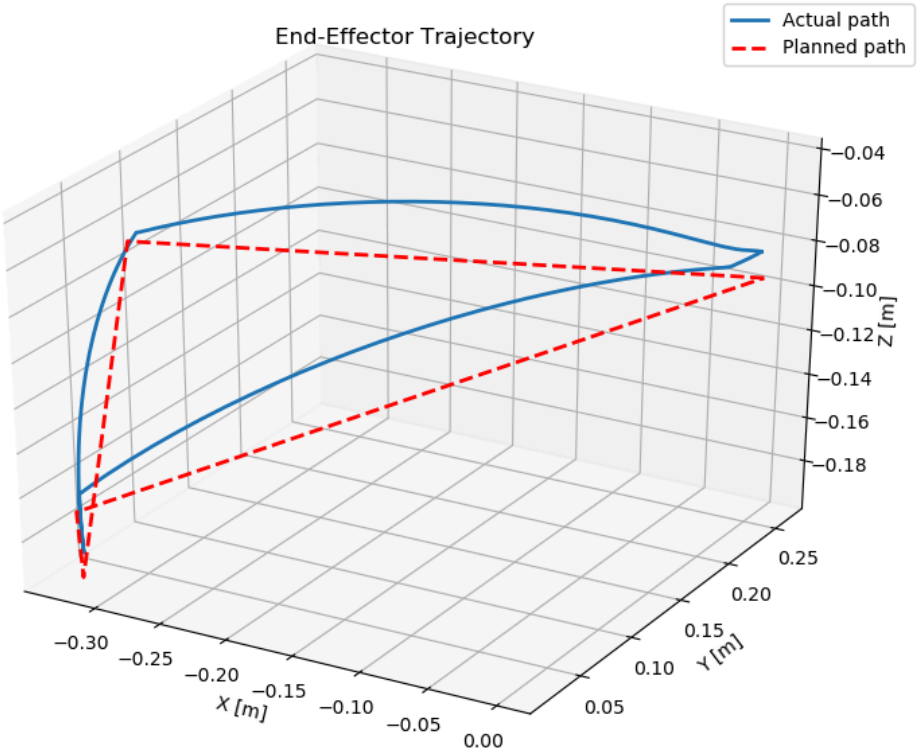

Figure 30: 3D view of the trajectory, both the planned and the actual travel path, whereas the planned path is displayed in a dotted red color, the actual path being displayed in a blue straight line. Figure created by the authors.

4.6 Visual representation of complete setup

After completing the development of the entire software and combining it with the assembled robotic arm the entire complete setup was placed on a flat surface to establish a full test area. This gave the group the complete setup. Since the moving hindrance zone never was completed it is excluded from this setup. Lastly, cables empowering the robotic arm and those linked to the computer going towards an electrical outlet of 230 Volt AC were placed such that they couldn't possibly interfere whilst the robotic arm performed any movement.

Figure 31: Visual representation of complete setup with the robotic arm being in operating mode. Here the light from the diodes display's that the breadboard is fully operating as current passes through them. Figure 17: Figure created by the authors.

Chapter V

Discussion

"When you give advice, you should first discern whether or not the other person is willing to accept it."
-Bushidō, 武士道

5.1 Control theory

Using the ViperX 300S became more challenging when the realisation of not just implementing 1 PID regulator for the gripper but rather 7 more for each joint. Making them working simultaneously made it even riskier for hardware malfunction. Furthermore using a camera above the gripper didn't feel as reliable as mounting a sensor, or a camera at the gripper which in return covers more dimensions compared to a camera just looking from above. This also made it quite challenging to make the arm have the possibility to navigate through a hindrance zone since some objects may not be a hundred percent visualized.

5.2 Signal processing

For this project there was no development of a digital filter. Instead the robotic arm used the inbuilt current controller with pre-set filters which was sufficient. This inbuilt filter digitally filtered out noise that was sent through the feedback link in the PID system, later on forwarding the signal through ros topics and nodes which helpt the PID controller to calculate its next action. Using this inbuilt filter saved a lot of time since it was already functional with the arm and the ROS software. If the group would have used their own sensors mounted on the robotic arm, a certain attenuator and receiver circuit would have had to be developed. And hence it would be required for the group to have constructed their own types of filter(analog or digital). In addition, making this new hardware to function with a computer and the arm would be very challenging and not something that there was enough time for.

5.3 Software

Linux was a new approach for both students participating in this project. There was no prior knowledge studied before the project which made it a challenging process to apply it. Even more, when applied, working in that operating system became more challenging due to its heavy knowledge required to make it work effectively.

Implementing ROS on Linux was an underrated process. Prior installtion the students had to learn and study how ROS works and its environment, what shall be included and how they're implemented. When this was achieved, several steps had to be executed which wasn't something as simple as downloading an exe.file. Instead, students had to create everything from scratch. In addition, working in the ROS enviorment seemed like learning a new programming language combined with specific hardware.

Developing the software was extremely challenging, especially due to working with several launch files in languages we hadn't taken any educational course prior to this project. Further on developing software that has

to be functional with a physical robotic arm. This was something students never had done before. Earlier software development mostly was simulations or towards simple breadboards.

5.4 Future work

Students will later continue experimenting on the robotic arm to resolve the missing parts of the main idea. These will include assembling a hindrance zone with a DC motor, continuing the software development of 2 algorithms meant to compete against each other but also using some sort of sensors or a camera to detect objects which will essentially solve the pick and place methodology.

The hindrance zone would have consisted of one Dc motor, 12 gear wheels, 6 cardboard boxes and strings, which can be borrowed from **Pekka Alakulju.** The Dc motor would have a cylindrical rod attached to its rotational axis, whereas at the other edge a gear wheel would be placed. As the Dc motor rotates the first attached gear wheel makes the other connected gear wheels rotate as well, moving the strings and the cardboards attached. Hence creating a hindrance zone with moving obstacles, powered by direct current.

To implement a torque algorithm that would properly work, one would need to consider these problems: the different moment of inertia, $I_{cylinder} = \frac{1}{2}mr^2$ for each joint, a way to calculate center of mass that is changed by movement and model for disturbance such as friction and heat etc. Deep understanding of rigid bodies and moment of inertia and a longer time-span for the project would be required to develop this algorithm, so in the initial stage of the code only the (K_P, K_I, K_D) parameters would be considered for each joint. In addition to that, a tuning script was created to test which the best parameters were for the controller. The tuning script consisted of different functions, one of which moved the arm to different random positions and another one tested parameters with the lowest relative error. Also a clipping function was used in this process to limit the forces acting on the arm: $\tau(t) = clipping(\tau(t) - \tau_{max}, \tau_{max})$. Although the tuning script produced parameters that improved after each iteration, it didn't quite reach the results that would be considered acceptable for further testing. For instance, it started with the lowest joints and got very close to the actual value of the position, but the positions for each joint deviated more and more from reference. An example of this would be to introduce a coordinate system $P_i \in (x_i, y_i, z_i)$ and an error $\Delta e = \theta_i^{ref} - \theta_i(t)$ and let it move to another position in the 3d where all coordinates are changed. Then using the standard eq. for the translation and rotation transformation to convert it to torque in the multi-joint system and (can be reversed transformed aswell). See PoE formula below.

$$\tau(\theta) = e^{\xi_1\theta_1 +} e^{\xi_2\theta_2} \dots e^{\xi_n\theta_m} M$$

Equation 6: Ideal filter requirement.

where:

θ_i is the joint angle

ξ_i is the twist angle for the joint

M is the end-effector or gripper position.

Then the next joint coordinates would then be slightly shifted or tilted from the first because the first was near but not exactly on the goal target. Which resulted in that the joint's actual positions to progressively be worse as it moved up the in the joint hierarchy. This exponentially accumulated drift in position can be described in eq. below and shown in fig X.

$$= e^{\xi_1\theta_1 + \Delta e_{joint\,1}} + e^{\xi_2\theta_2 + \Delta e_{joint\,2}} \dots e^{\xi_n\theta_m + \Delta e_{joint\,n}} M$$

Also if the positions relative to one another may cancel out its still bounded by the triangle inequality below since it cannot be shorter than the “perfect” way to that point.

$$\left|\left| \sum_{j=1}^{N} \Delta e_{gripper} \right|\right| \leq \sum_{j=1}^{N} \left|\left| \Delta e_{gripper} \right|\right|$$

All of these discoveries led to the final algorithm which utilized the trossens robotics PID controller, trajectory joint effort controller, to find the target position making the robot arm act like a simple agent.

5.5 Final thoughts

The ViperX300S robotic arm had a lot of **hardware issues** with loose cables disabling several joints in a chain reaction. For instance, one data transfer cable was torn apart at the waist joint, disabling several motor servos upwards, which occurred twice due to rapid movement of the arm which caused collision with its environment. This happened due to poor tuning of the pid parameters in the software code. In addition, debugging the error messages in the control panel from the robotic arm was extremely challenging and would often take several days to find the error and fix it.

Operating a robotic arm with better hardware than the ViperX 300S would have saved **a lot of time**. As an example, the robotic arm has a retail price of at least 5 thousands usd which is a hefty price as a hobby arm. Especially since much of the arm is already pre-developed. Such as the breadboard not being constructed in-house nor the motors. It is however important to include that the motors retail price was the main part bumping up the arms retail price. In addition, a lot of the parts(screws especially) were 3D printed of cheap plastic materials which easily deformed under pressure. Lastly, the arm had clamped and short cables(especially around the gripper) which could easily be broken if the arm moved in a certain way.

5.6 Conclusion

During our years at Uppsala University we have learnt a lot about electrical engineering. In our opinion, mostly the theory behind how electrical hardware functions. This knowledge was later implemented on different projects which were very giving. Whereas the final project was the most challenging since it required the students to leave their comfort zone, approach new topics that were difficult but could be solved with earlier possessed knowledge. In addition, from this project it was discovered that after completing your Bachelors degree it is possible to take your Masters degree in Robotics, not something that Uppsala University offers compared to University of Bristol, KTH Royal Institute of Technology, Princeton, Michigan and many more universities. With this in mind, our group thinks that Uppsala University should implement a masters degree possibility in this subject due to how widely it is used in the current market.

References

Adams, M. D. (2020). *Signals and systems* (3rd ed.). Department of Electrical and Computer Engineering, University of Victoria.

Batlle, J. A., & Barjau Condomines, A. (2022). *Rigid body dynamics*. Cambridge University Press.

Ben-Kiki, O., Evans, C., & Ingerson, B. (n.d.). *YAML Ain't Markup Language (YAML™) revision 1.2*. YAML.org. Retrieved February 25, 2025, from https://yaml.org/

Bolmsjö, G. S. (2006). *Industriell robotteknik*. Studentlitteratur.

Čapek, K. (2001). *R.U.R. (Rossum's Universal Robots)*. Dover Publications. (Original work published 1920)

Go, I. (2022). *Installing ROS Noetic on Ubuntu 20.04 using VirtualBox (on Windows)*. GitHub. Retrieved March 21, 2025, from https://github.com/sutd-robotics/virtualbox-ubuntu-ros

Linux Foundation. (n.d.). *Tux the penguin [Mascot of Linux]*. https://www.linuxfoundation.org

Linux Kernel Organization. (n.d.). *The Linux Kernel Archives*. Retrieved March 21, 2025, from https://www.kernel.org/category/about.html

MathWorks. (n.d.). *MATLAB*. Retrieved March 2, 2025, from https://www.mathworks.com/products/matlab.html

MathWorks. (n.d.). *What is model predictive control?* Retrieved March 28, 2025, from https://www.mathworks.com/help/mpc/gs/what-is-mpc.html

Microsoft. (n.d.). *XML för nybörjare*. Microsoft Support. Retrieved March 2, 2025, from https://support.microsoft.com/sv-se/office/xml-f%C3%B6r-nyb%C3%B6rjare-a87d234d-4c2e-4409-9cbc-45e4eb857d44

Molin, B. (2020). *Analog elektronik* (3:e uppl.). Studentlitteratur.

Open Source Robotics Foundation. (2020). *ROS Noetic Ninjemys*. ROS Wiki. Retrieved March 28, 2025, from https://wiki.ros.org/noetic

Open Source Robotics Foundation. (n.d.). *gazebo*. ROS Wiki. Retrieved March 2, 2025, from http://wiki.ros.org/gazebo

Open Source Robotics Foundation. (n.d.). *rviz*. ROS Wiki. Retrieved March 2, 2025, from http://wiki.ros.org/rviz

Open Source Robotics Foundation. (n.d.). *Xacro*. ROS Wiki. Retrieved March 21, 2025, from https://wiki.ros.org/xacro

Python Software Foundation. (n.d.). *Beginner's guide for programmers*. Python Wiki. Retrieved February 25, 2025, from https://wiki.python.org/moin/BeginnersGuide/Programmers

Roberts, E. (n.d.). *History of robotics*. Stanford University. Retrieved February 20, 2025, from https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/history.html

Robotics Academy. (n.d.). *History of robots*. Retrieved February 20, 2025, from https://www.roboticsacademy.com.au/history-of-robots/

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

Thomas, B. (2016). *Modern reglerteknik*. Liber.

Trossen Robotics. (n.d.). *Interbotix ROS arm vision kit*. RoboSavvy. Retrieved March 15, 2025, from https://robosavvy.co.uk/interbotix-ros-arm-vision-kit.html

Zhao, J.-S., Wei, S.-T., & Sun, X.-C. (2023). Computational dynamics of multi-rigid-body system in screw coordinate. *Applied Sciences, 13*(10), 6341. https://doi.org/10.3390/app13106341