universität innsbruck

Fakultät für Technische Wissenschaften

Development of a virtual reality interface for multibody systems and serial robots

Aaron Bacher, BSc

Innsbruck, August 16, 2023

Masterarbeit

verfasst im Rahmen eines gemeinsamen Masterstudienprogramms von LFUI und UMIT – Joint Degree Programme

eingereicht an der Leopold-Franzens-Universität Innsbruck, Fakultät für Technische Wissenschaften zur Erlangung des akademischen Grades

Diplomingenieur

Beurteiler:

Univ.-Prof. Dipl.-Ing. Dr. Johannes Gerstmayr

Institut für Mechatronik
Maschinenelemente und Konstruktionstechnik

Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Johannes Gerstmayr, Universität Innsbruck, Institut für Mechatronik, Maschinenelemente und Konstruktionstechnik

Acknowledgment

First of all, I would like to thank Peter Manzl and Martin Siegfried Sereinig, who over the course of the thesis have spent countless hours with me in the laboratory and helped me with advice and support on a wide variety of problems.
I would like to thank the institute and especially Prof. Johannes Gerstmayr for providing the space and financial resources. Special thanks also go to him as my supervisor for the inspiring meetings, the time often made available at very short notice, and his help with the implementation of OpenVR in Exudyn.
Also Wolfgang Mark deserves my gratitude for helping me with some minor technical problems and with the development and the manufacturing of the 3D printed parts used in this project.
Finally, my biggest thanks go to my family, who made my studies possible in the first place and supported me throughout. A special thanks goes to my mother, who helped me with the linguistic design of this work with words and deeds.

Kurzfassung

In dieser Arbeit werden kollaborative Robotik und virtuelle Realität (VR) verknüpft, um eine Schnittstelle zu erstellen, welche einem Nutzer die physische Interaktion mit einer Mehrkörpersimulation in Echtzeit ermöglichen soll. Dabei soll ein Nutzer Kräfte auf einen Roboter aufbringen können, welche auf die Simulation übertragen werden und der Nutzer dadurch entsprechendes haptisches Feedback direkt am Roboter und visuelles Feedback über eine VR-Brille erhält. Als Simulationsprogramm dient Exudyn, ein effizientes Mehrkörpersimulationsprogramm, welches an der Universität Innsbruck entwickelt wird. Die Kommunikation der einzelnen Komponenten soll über das Robot Operating System (ROS) realisiert werden.
Nach einem kurzen Überblick über das Zusammenspiel der einzelnen Komponenten und über ähnliche Projekte werden fundamentale Inhalte für die Arbeit eingeführt und erörtert. Im Anschluss widmet sich diese Arbeit zunächst der Robotersteuerung. Wie in der ursprünglichen Aufgabenstellung vorgeschlagen, wird zuerst ein Impedanzregler betrachtet. An einem 7-Achs Roboter vom Modell Franka Emika Panda werden erste praktische Versuche durchgeführt, wobei der Roboter einer Trajektorie, welche ihm, wie auch später bei der Echtzeitsimulation, erst nach und nach mitgeteilt wird, möglichst genau und mit kleiner Latenz folgen soll. Da dieser Reglertyp aber auch nach empirischer Anpassung der Parameter noch hohe Verzögerungen aufweist und sich die Nachgiebigkeiten des Reglers und des Simulationsmodells überlagern, wird entschieden, stattdessen einen PID-Regler auf Geschwindigkeitsebene zu verwenden. Dieser ist grundsätzlich in der Lage, der Trajektorie präziser und mit kleinerer Verzögerung zu folgen, ist aber deutlich anfälliger für Verzögerungen in Verbindung mit dem ROS-Netzwerk. Deshalb wird zum einen eine FIFO-Warteschlange eingeführt, die Befehle kurz zwischenspeichern und sie der Steuerung anschließend in zeitlich regelmäßigen Abständen übergeben kann, wodurch Verzögerungen kompensiert werden können. Zweitens wird die Sendefrequenz der Befehle von der Simulation zur Steuerung verkleinert, wodurch das Netzwerk entlastet wird und die mittlere Sendedauer und deren Standardabweichung verkleinert werden können. Zwischen diesen Befehlen wird polynomial interpoliert, damit dem Regler neue Referenz-Werte mit der geforderten 1 kHz Frequenz gesendet werden können.
Die Schnittstelle für das Rendering auf die VR-Brille wird direkt in C++ in Exudyn implementiert und Methoden und Abläufe werden beschrieben, mit welchen die Positionen verschiedener Objekte verfolgt werden, um sie in VR korrekt darstellen zu können. Anschließend werden verschiedene Aspekte beleuchtet, welche die Echtzeitfähigkeit des Systems garantieren sollen. Dazu zählt unter anderem die Verwendung eines Echtzeit-Patches für das Betriebssystem und die Auslagerung der Visualisierung auf einen zweiten PC. Nach Beschreibung des im Zuge der Arbeit erstellten Python Interfaces schließt die Arbeit mit der Vorstellung der Ergebnisse anhand eines konkreten Beispiels.

Abstract

In this work, collaborative robotics and virtual reality (VR) are combined to create an interface that will allow a user to physically interact with a multibody simulation in real-time. The idea is for a user to be able to apply forces to a robot, which are transmitted to the simulation, which again gives the user appropriate haptic feedback directly on the robot and visual feedback on a head mounted display. The simulation program used is Exudyn, which is being developed at the University of Innsbruck. The simulation itself needs to be highly efficient, as the multibody system needs to be simulated faster than real-time to enable the described control loop. The communication of the individual components is supposed to be realized via the Robot Operating System (ROS).
After a brief overview of the interaction of the individual components and of similar projects, fundamental content for the work is introduced and discussed. Subsequently, the first item to be discussed is robot control. An impedance controller is considered first since such controller type was proposed in the original assignment. First practical experiments are carried out on the 7-axis Franka Emika Panda robot. As in the ensuing real-time simulation, a given trajectory, communicated only gradually, is to be followed as accurately as possible and with low latency. However, high delays even after empirical adjustment of the parameters and controller compliance interfering with the one of the simulation models suggest the use of a PID controller at velocity level instead. The PID controller can follow the trajectory more precisely, is more reactive to applied efforts due to a smaller delay and more sensitive to delays in the connection via the ROS network. Therefore, a FIFO queue is introduced to buffer commands and pass them to the controller at regular time intervals, thus compensating delays. Secondly, the transmission frequency of the commands from the simulation to the controller is reduced, relieving the network by reducing the average transmission time and its standard deviation. Polynomial interpolation is performed as fine interpolation between commands in order for the controller to be supplied with the required 1 kHz frequency.
The interface for rendering to the head mounted display is implemented directly in C++ in Exudyn and methods and procedures are described to track the positions of various objects in order to display them correctly in VR. Subsequently, various aspects intended to guarantee the real-time capability of the system are focused on. They encompass the use of a real-time patch for the operating system and the associated outsourcing of the rendering process to a second PC. After describing the Python API created in the course of this work, the thesis concludes with the presentation of the results by way of a specific example.

Contents

List of Figures

Glossary

The present thesis introduces and uses multiple abbreviations for various terms. Frequently used abbreviations and their meanings are listed in the table below. Abbreviations which are used only locally within one or two consecutive sections are not listed here.

Abbreviation	Meaning
EE	end-effector
FCI	franka control interface
HHC	hand held controller
HMD	head mounted display
HRC	human robot collaboration
MBS	multibody system
OS	operating system
RCU	robot control unit
ROS	robot operating system
RSP	reference sampling point
RT	real-time
UIP	user interaction point
VR	virtual reality

Furthermore, it is necessary to define some more terms that are used frequently:

- In mechanical engineering, the term "wrench" describes a vector that includes force and torque. In the context of ROS, it is common to use the term "effort" instead. It is also used when referring to forces and torques in general and not specifically to the vector itself.
- The term "pose" describes the position and orientation of an object in Cartesian space.

1 Introduction

Understanding how multibody systems (MBS) behave when certain efforts are applied is crucial to every development process in machine dynamics. In many cases appropriate simulation software can help to get an understanding of that. In many cases appropriate simulation software supports better conception by not only providing an array of numbers as simulation results but by also visualising the simulated system. However, while analyzing the result of such simulations and looking at their visualization on a computer screen or via VR goggles can be helpful, getting a haptic feedback increases the degree of reality and recognition value significantly[1]. If that is the case with an MBS, one way to achieve this is to create a physical model. Depending on the complexity of the model this, however, requires appropriate tools, materials and know-how. And even if all the prerequisites are in place, as soon as even a small change is needed in the system, complex changes or even a restart may be necessary. In comparison, applying the same change to a simulation may require only changing a number.

1.1 Objectives

The main objective of this thesis is to create a human-robot interface which allows the user to physically interact with a multibody system simulation. In order to do so, the movements of a predefined point in an MBS simulation is mapped on the end-effector (EE) of a robotic arm, while forces and torques applied to the manipulator are transmitted back to the MBS simulation, where they are applied to the same point. This point will be called user interaction point, short UIP, in further proceedings. For the MBS simulation Exudyn[2][1] is used, a simulation program for flexible multibody dynamics, which is developed and maintained at the Institute for Mechatronics - Machine Elements and Design at the University of Innsbruck (AT), where the present thesis has been written. For the manipulator a Franka Emika Panda robot is used. For communication, the robot operating system (ROS) is used, which should allow for the robot to be easily replaced if necessary. For good user experience, the robot should follow the reference trajectory as closely as possible with minimal latencies between MBS simulation and robot movements.

In addition to the haptic interface, a virtual reality (VR) system is set up to give the user the sensation of being fully immersed in the MBS, interacting with what the MBS consists of. Therefore the user has to wear a head mounted display (HMD), in which the MBS is rendered in such a way that the UIP is at the very position where the EE of the manipulator is located in the real world.

[1] `https://github.com/jgerstmayr/EXUDYN`, last accessed 09.05.2023

Due to hardware limitations of the robot[2] and high latencies in the communication via ROS high-frequency oscillations with the robot cannot be run. Therefore simulations are limited to low-frequency MBSs.

1.2 Related work

VR-systems and collaborative robots have continuously increased in popularity over the last few years and the present thesis is not the first attempt to combine them. Moniri et al. [3] have developed a system with a human giving commands to a robot using a VR system. Since human and robot are in two non-overlapping work spaces, there is no physical contact between them. Huy et al. [4] have created a system where a human operator controls a mobile robot using of a see-through HMD and a custom handheld device. Although the human can enter the robot's workspace, there is still no physical human-robot collaboration (HRC). Evrard et al. [5] have developed a framework with the user interacting with a virtual avatar or other virtual environment by using a device equipped with sensors and actuators in all six degrees of freedom, thus being able to perceive user input while simultaneously giving haptic feedback.

While those examples all combine HRC within virtual environments, they are not very similar to the objective of the present thesis. Closer, however, comes Gatterer [6], who uses a robot manipulator to give real haptic feedback to the user, who navigates in a virtual environment using an HMD, a VR treadmill and other sensors for full body tracking. For example, the manipulator may bring a wall nearer to the user when the user "walks" towards it. This project uses the Unity-engine[3] to render the virtual environment and to handle physics and collisions of virtual objects with each other, tasks which should be covered by Exudyn in the present thesis.

Eberhard [4] created a framework where the user can interact with an MBS in VR, thus making it very similar to the thesis project. It does, however, not provide any haptic input or feedback, which is the main focus of the present thesis.

1.3 Thesis structure

This thesis project encompasses two main components, namely the haptic and visual interfaces. For ease of understanding and navigation, a brief summary of each chapter is provided in this section.
Chapter 2 covers several different topics, all relevant to the realization of this project without being new findings. It may be also interpreted as state of the art, which this thesis is based on. After a basic overview of all the components involved in this project, the utilized robot operating system (ROS), the concept of collaborative robots, MBSs and a simple MBS example, on which the present project will be tested throughout this thesis, are introduced. Chapter 3 covers the development process of the robot controller. After

[2] `https://frankaemika.github.io/docs/control_parameters.html`, last accessed 12.04.2023

[3] `https://unity.com`, last accessed 24.03.2023

[4] see `https://www.itm.uni-stuttgart.de/en/research/echtzeitsimulation-fuer-mehrkoerpersysteme-in-einer-virtuelle-realitaet-anlage/` "Real-Time Simulation of Multibody Systems in a Virtual Reality Environment" by Eberhard, Peter. Institute of Engineering and Computational Mechanics. University of Stuttgart. Last accessed 24.03.2023

an outline of the used robot's capabilities with standard functionalities and their limitations to the present application, two different advanced control / trajectory generation strategies for real-time application are described. Finally, tests with the MBS simulation built in Exudyn are performed. In chapter 4 an overview of the VR-hardware is given, important aspects of the rendering process are examined and the use of different hardware components of the VR-system to locate physical objects is described. Chapter 5 focuses on the actions taken to minimize latencies between the single system components in order to ensure functionality and improve user experience. Chapter 6 describes the different user interfaces for the present project: the programming interface used to add robot- and virtual reality functionalities to an MBS built in Exudyn and the physical object mounted on the robot EE as physical UIP. In chapter 7 a simulation with working visualization is performed and discussed. Also, some restrictions applicable to the current implementation are mentioned. Chapter 8 gives a summary of what has been accomplished in the course of the present thesis and an outlook on which components could still be improved. The appendix holds a description on how to set up the VR-system and put the whole system into operation.

2 Fundamentals

This chapter gives a general overview of which components are required to meet the objectives listed in section 1.1 and explains how they interact with each other. The reader is introduced to several topics essential to a proper understanding of the explanations in the following chapters.

2.1 System overview

The present thesis project is divided into several components. Figure 2.1 shows them and how they are connected to each other.

The main objective of the present thesis is a human-robot interface to enable the user to physically interact with an MBS simulation. This interaction is visualized at the top of Figure 2.1. The user may apply forces to the robot EE but due to Newton's third law and the dynamics of the MBS the robot also applies forces onto the user. The term wrench describes a vector containing all spatial forces and torques. The wrench registered by the robot's sensors are sent to the MBS simulation and applied at the UIP. In the op-

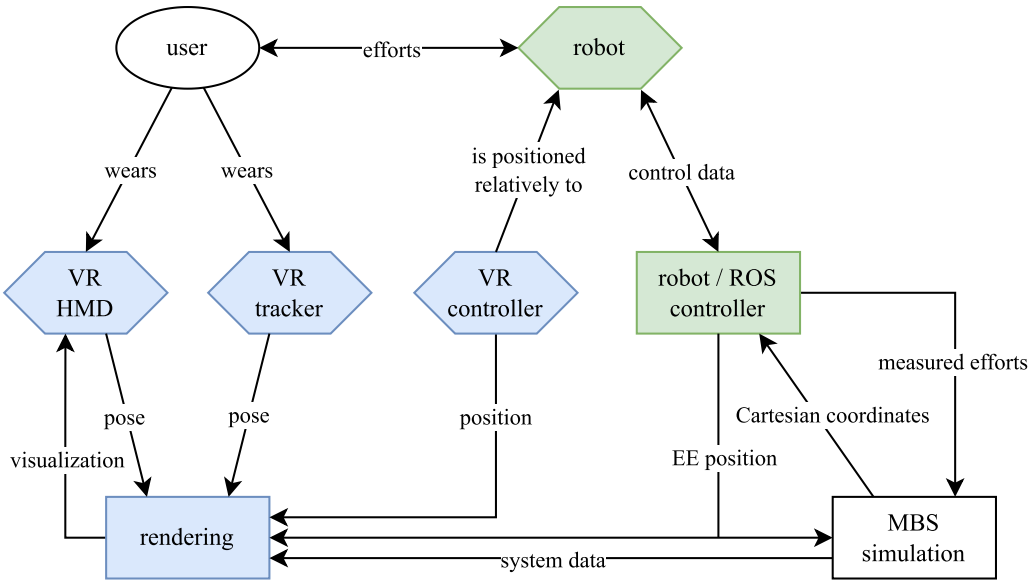


Figure 2.1: Conceptual overview of system components and their relationships. Physical objects are marked as hexagons, software modules as rectangles. Furthermore, the interactive part is colored in green and system components needed for visualization in blue.

posite direction, the simulation sends commands to the robot to control its movements. Depending on the implementation of the robot controller, the commands may vary in format, but are always Cartesian coordinates (e.g. positions or velocities). The robot controller transforms them to currents for each single joint before sending them to the robot itself.

The second part of the present project encompasses the virtual reality system, which corresponds to the bottom left part of Figure 2.1. There the MBS is constantly rendered to a head mounted display (HMD) worn by the user. To define the position of the rendering in VR a VR tracking device, a hand-held controller, is positioned next to the robot. At this point it must be noted that unlike the above-mentioned robot-controller, which actively controls the robot, the here discussed VR hand-held controller does not control the VR system. It may be interpreted as a sort of remote control, used as a mouse pointer or tracking device to interact with the system, however, the VR system works also without. Using such a device, the robot and its EE can be located in VR space and the MBS can be rendered at the correct position. Additionally, some other objects (e.g. tables), present in the real world in the user's workspace and in known positions relative to the hand-held controller, are also rendered to prevent the user from running into them. Finally, the user's hand is rendered to allow for the user to assess the distance between the hand and the robot EE or any other rendered object. To locate the hand, the user is wearing a VR tracker at their wrist.

Figure 2.1 also shows the EE position (in Cartesian coordinates) sent by the robot controller to the MBS simulation as well as to the rendering module. This is needed once for calibration at both destinations at the start of the application. All components will be described in more detail later on in this thesis.

2.2 ROS

The robot operating system [7] (ROS) is not an operating system in a conventional way but provides a simple communication layer for inter-process communication for distributed systems in a network. It is open-source, widely used in robotics and can be used within several programming languages. Every process (software module) accessing the ROS-network is called a node and communicates with other nodes by sending messages to each other. This happens on a peer-to-peer basis, no physical server is required. The only server-like component needed is a so-called master, which provides registration and naming services for processes to be able to find each other at runtime [8].

The messages can be composed of different fields, i.e. they can hold an arbitrary amount of different data fields and types. The ROS framework provides several standard message types (e.g. the `std_msgs` or `geometry_msgs`), but also custom messages can be created. For communication two general possibilities are available:

- By publishing on a certain topic, which is represented by a string, the content of the published message is made available to every node in the ROS network. To receive the message another node needs to subscribe to the same topic. As many nodes can subscribe simultaneously, this form of communication can be interpreted as broadcasting and is also known as the observer-principle [9].
- Another way of communication is making use of so-called services. A node requests information or a general server functionality from another node by sending

a previously defined type of request-message, receiving the answer through an also previously defined type of response-message. In contrast to publishing messages on a topic, the information interchanged between the two nodes is not made available to other nodes. Furthermore, the information is only sent on request and not permanently.

Regardless of the type of transmission, ROS-messages per default are transmitted via TCP/IP. The C++ implementation of ROS allows for UDP to be used, which is, however, not maintained in the Python implementation[1] used for this project. This shows one of the major disadvantages of ROS: it lacks real-time capability. ROS2 claims to be real-time capable[2], but as at the time of the start of this work ROS2 had not been widely supported yet, ROS was not replaced with ROS2 for this project.

Since applications can become large and unwieldy, ROS offers the possibility to collect scripts, message types, etc. in packages. This allows for them to be organized by subject matter and facilitates distribution, as dependencies on each other or on 3rd party or default packages can be clearly defined.

In order to use packages with all their contents they need to be built first. This is done with catkin, the official build system for ROS, based on CMake. In addition to CMake's build capabilities, catkin contains further utilities for package-navigation and cross-compilation[3].

2.3 Collaborative robots

As the main objective is to create a human-robot interface, it seems coherent to use a collaborative robot. This term, also known as cobot, was first used by Colgate et al. [10] and describes a robot that, in contrast to classic industrial robots, is generally designed for robot tasks in an undefined workspace and specifically for HRC. It is equipped with various position and force/torque sensors, which enable it to detect collisions and stop its movements if needed, which is particularly important when humans are within its workspace and to prevent damage to itself or it's surroundings. Furthermore, modern cobots are often equipped with an easy-to-use graphical user interface, which allows simple robot programming and teaching the robot new movements by putting it into appropriate poses by hand, storing them and making the robot move through them in the correct order.

2.4 Multibody system simulation

Every mechanical system containing more than one individual part can be described as a multibody system. The single parts interact with each other via different constraints [11]. They range from small systems, such as a simple slider cranks or a pendulums, to complex systems, such as cars or trains. For an efficient development of complex systems it is necessary to test new systems theoretically before assembly, thus making

[1] `http://wiki.ros.org/Topics#TopicTransports`, last accessed 29.03.2023

[2] `https://docs.ros.org/en/dashing/Tutorials/Real-Time-Programming.html`, last accessed 29.03.2023

[3] `https://wiki.ros.org/catkin/conceptual_overview`, last accessed 20.04.2023

the development process more efficient [11]. Programs capable of simulating multibody system dynamics [12] are therefore called multibody system simulations.

2.4.1 Exudyn

Exudyn [2] is a multibody simulation program developed and maintained at the University of Innsbruck. It is a C++ library with a Python-wrapper, which allows for a specific system to be built together conveniently in Python while the complex and computationally intensive calculations are executed efficiently in C++. It is open-source[4] and can therefore be used and modified by everybody. For these reasons it is an excellent choice for use in the present thesis.

The Exudyn documentation[5] provides explanations and examples of all implemented functionalities. Not all of them are needed in the present thesis, so for ease of understanding the basic building blocks (items) for such models and other utilities are described in the following.

Items

Conventionally, an MBS consists of multiple bodies connected to each other in different ways. In Exudyn all bodies, connections etc. are called items and for implementation they are divided into different categories.

The items that represent a physical body or a connection between such bodies are called (computational) objects. Depending on their functionality they can be further subdivided: A body represents a physical body and a connector represents a joint/connection between two bodies (e.g. joint, spring-damper).

While these objects provide mathematical equations for the MBS, the resulting system is not solved for unknown quantities of the same. Those are provided by so-called nodes (not to be confused with ROS nodes, described in section 2.2), which do not have any physical quantities, but only provide (unknown) coordinates for the system. Accordingly, they are attached to objects, with object types being defined by different amount of nodes: One node suffices for a rigid body, several nodes are required for finite-element bodies.

With nodes and objects an MBS in Exudyn can now be defined unambiguously mathematically, but it takes further items for the practical implementation of the system. Markers can be placed on objects and serve as interfaces between them, i.e. a connector cannot be placed directly between two objects but only between two markers, which in turn are placed on the two objects. Also, markers are needed to apply loads, which are themselves a separate type of item, to objects.

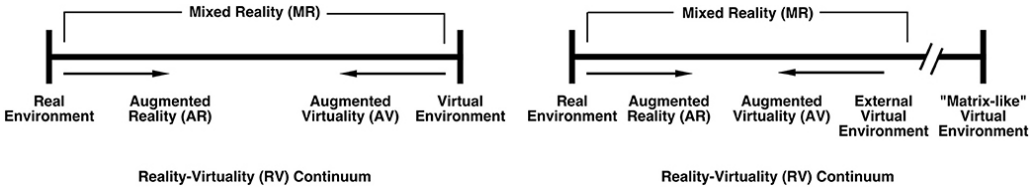
To read physical quantities (e.g. displacements, orientations) of different objects, sensors can be attached.

SystemData

All the items needed for a specific MBS and the simulation coordinates (e.g. the ODE2 coordinates) are stored in the systemData data structure. This data structure can be ac-

[4] source code available at `https://github.com/jgerstmayr/EXUDYN`, last accessed on 07.03.2023

[5] documentation available at `https://github.com/jgerstmayr/EXUDYN/blob/master/docs/theDoc/theDoc.pdf`, last accessed 07.03.2023



(a) (b)

Figure 2.2: Reality-Virtuality (RV) Continuum by Milgram and Kishino [15] (a) and Skarbez et al. [16] (b)

cessed and modified in Python but must be used with utmost caution to prevent any system/simulation failures. The systemData structure is used to synchronize the simulation on multiple computers, discussed in chapter 5.

PreStepUserFunction

When setting up an MBS in Exudyn, all items and their connections between each other need to be defined before starting the simulation. To allow for interaction with the simulation while it is running, e.g. to update loads, Exudyn provides the possibility to define a so-called `PreStepUserFunction`, which is called once before every computational step. In this project the `PreStepUserFunction` is used for the robot-simulation interface.

2.4.2 Example simulation

In order to test this thesis project during the development process, a simple Exudyn example MBS is created[6]. A single pendulum is chosen which can swing in two mutually perpendicular planes around a pivot at the top of the rod, thus providing for two degrees of freedom. The origin of the MBS coordinate system $\mathscr{K}_{\mathrm{MBS}}$ is also located at the pivot. To prevent endless swinging, a slight rotational damping of $1\,\frac{\mathrm{Nms}}{\mathrm{rad}}$ is installed in each axis at the pivot point. While the pendulum rod is massless, the tip of the pendulum has a mass of 6 kg and is thus affected by gravity ($9.81\,\frac{\mathrm{m}}{\mathrm{s}^2}$). The tip will also serve as UIP. To further simplify the example during testing, one direction has been locked by cancelling any forces applied to it, which makes the pendulum swing in only one plane.

2.5 Virtual Reality

Although virtual reality has been around since the 1960s[13], its popularity has fluctuated greatly since then[14]. In spite of having been declared dead by some, in the last few years it has gained acceptance and is now thriving in many fields of application, such as military, medicine, home entertainment and many others[13]. Its benefits are manifold: it may provide quality of life/work improvements or just give the user the sensation of being fully immersed in a purely virtually created world to have visual experiences, which would be hard to reproduce in reality with a comparable amount of effort.

[6] source code of final version of pendulum available at `https://github.com/bacaar/exuCobot/blob/main/exudyn_interface/scripts/example4_1.py`, last accessed 26.04.2023. This code already includes the interfaces developed throughout this thesis.

The relation between reality and virtuality has been described by Milgram and Kishino [15] in their Reality Virtuality Continuum, depicted in Figure 2.2a. As the name states, they describe the transition as continuous. On the left side, there is the real world we live in, which follows the laws of physics. Users sees things as they are, a worn HMD would be fully see-through without any artificial changes (i.e. a transparent glass or even without any display). On the right side, there is the virtual world, which is purely synthetic, and no aspects of the real world are visible. In between there is the mixed reality, which can be divided into two classes: Augmented reality represents the real world augmented with additional information. An example would be Google Glass[7], which is a see-through HMD with a mini-projector which projects images through a semi-transparent prism directly onto the user's retina. Another example would be head-up displays (HUD) used e.g. in modern aeronautics but also more and more in automotive industry. Further to the right there is augmented virtuality, where real objects are added to a virtual environment.

Skarbes et al. [16] note, however, that Milgram and Kishino[15] refer to different types of environments created by the use of visual displays only, while all other human senses are neglected. Thus, even an entirely virtual environment in their RV-continuum is not really virtual, since the user still perceives external stimuli via other senses and thus the realistic effect of the virtual image is diminished. An example would be an environment where the law of gravity is neglected. While the visual display might give the sensation of floating around in zero-g, the user would still feel that they are in fact affected by gravity in the real world [16]. Thus Skarbes et al. propose to extend the original RV-continuum with another section even further to the right, the matrix[8]-like virtual environment [16] (see Figure 2.2b) where the user's own senses are disconnected from the user's brain and fully replaced by technology. They also change the denomination of the RV-continuum from virtual environment to external virtual environment, due to the fact that even if technology creates a perfect virtual environment that is confusingly similar to reality, it is still only external as the user can still tell the difference from a real environment using all senses.

To sum up, it can be stated that an application can be classified differently depending on the number of stimuli derived from the real world respectively created synthetically, and which senses are considered in the first place. According to Milgram and Kishino[15], the created (visual) environment in this project is considered purely virtual, since all the see-through capabilities of the HMD are not used, it therefore is completely opaque and all the rendered objects are created only virtually. According to Skarbez et al., who also allow for other senses to be considered in this classification, however, the entire application created in this thesis is part of augmented reality, since the haptic feedback allows stimuli from the real world. For sake of simplicity, though, the system will always be denominated as virtual reality[9] (VR) throughout this thesis, as it is used by most people as the general umbrella term for all systems of this kind.

[7] `https://www.google.com/glass/start/`, last accessed 07.03.2023

[8] `https://en.wikipedia.org/wiki/The_Matrix`, last accessed 09.05.2023

[9] This term is actually a contradiction in terms, since "virtuality" and "reality" are the two extremes of both described continua.

3 Robot control

This chapter focuses on the development of a controller for the interactive haptic part, implemented via a robot following a specified trajectory. The utilized robot is the Franka Emika Panda serial manipulator. Various control approaches, their implementations, and comparisons are presented for comprehensive understanding.

3.1 Franka Emika Panda

The Franka Emika Panda robot is a seven-axis collaborative robot from the German manufacturer Franka Emika. It is mostly used for teaching and research[1], although also a version for industrial applications is available now[2]. It has torque sensors in all seven joints[3] and with the seventh - redundant - axis the robot provides additional flexibility for movements, as there are only six degrees of freedoms in the Cartesian space (3x translation, 3x rotation). Further advantages, which make the robot an optimal choice for use in the present thesis are its open source control code[4] and already built-in compatibility with ROS.

Figure 3.1 shows the robot in its default configuration, i.e. with all joint angles in zero-position and Figure 3.2 shows its accessible workspace. The robot has a maximum range of 855 mm or a little less if border singularities should be avoided to preserve all degrees of freedom. There are no singularities within the workspace, which is beneficial for path planning, as there are no precautions to be taken.

Besides the robotic arm itself, also the robot control unit (RCU) belongs to the system, which can be accessed directly over Ethernet[5] via the manufacturer-specific API libfranka (see subsection 3.1.1). It accepts various types of commands in both Cartesian or joint space, and controls the robot's joint currents to carry out the desired movement.

3.1.1 Robot API

In addition to the sensors and the redundant degree of freedom, also its programming user interface is a good reason to use this specific robot. The so-called Franka Control Interface[6]

[1] `https://petercorke.com/robotics/franka-emika-panda-kinematics-and-singularities/`, last accessed 18.02.2023

[2] `https://www.franka.de/de/production`, last accessed 28.05.2023

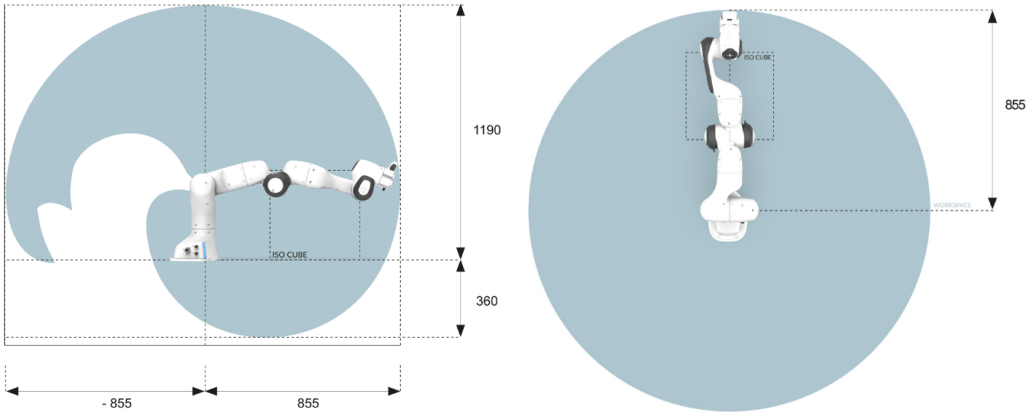
[3] `https://wiredworkers.io/wp-content/uploads/2019/12/Panda_FrankaEmika_ENG.pdf`, last accessed 28.10.2022

[4] available at `https://github.com/frankaemika`, last accessed on 02.03.2022

[5] `https://frankaemika.github.io/docs/libfranka.html`, last accessed 22.12.2022

[6] `https://frankaemika.github.io/docs/control_parameters.html`, last accessed 28.10.2022

Figure 3.1: The Franka Emika Panda robot in its default configuration with $\theta_i = 0$ for $i \in \{1, ..., 7\}$.[6]



(a) side view (b) top view

Figure 3.2: Workspace of the Franka Emika Panda robot[3]

(FCI) runs on the RCU and facilitates low-level bidirectional communication between workstation and manipulator[7]. Before utilization, it needs to be enabled in Franka Desk, the graphical user interface, accessible in arbitrary browsers via the robot's local IP-address. Once FCI is enabled, it can be accessed in several ways[8]. One way is the open source[9] API called libfranka[10] for writing robot controllers for realtime and non-realtime

[7] `https://pkj-robotics.dk/wp-content/uploads/2020/09/Franka-Emika_Brochure_EN_April20_PKJ.pdf`, last accessed 03.04.2023

[8] `https://frankaemika.github.io/docs/`, last accessed 05.04.2023

[9] available at `https://github.com/frankaemika/libfranka`, last accessed on 20.02.2023

[10] see footnote 5, last accessed 22.12.2022

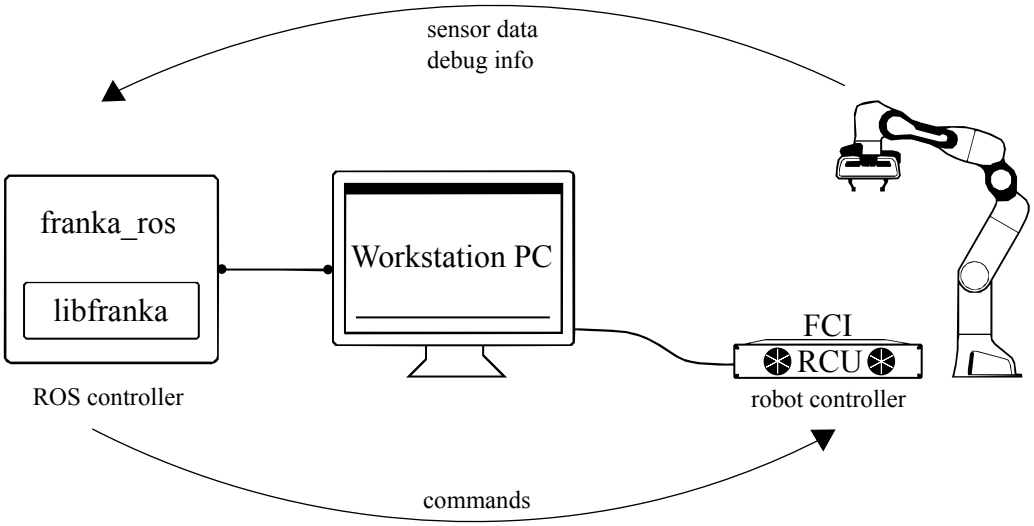


Figure 3.3: Interaction between different components when working with the Franka Emika Panda robot.

applications in C++. Furthermore, it comes with several example controllers which make robot programming also accessible for intermediates. Additionally, Franka Emika gives users also open-source[11] access to franka_ros[12], which, as the name already suggests, adds ROS-support to libfranka by integrating it into ROS-Control[13]. This includes basic ROS-functionalities, e.g. topics and messages, but also more advanced utilities like franka_gazebo, a package which allows for simulating a robot, or access to MoveIt (see section 3.2) to control the robot in a simple way.

Figure 3.3 visualizes the interaction between the above-mentioned components. The FCI runs on the RCU, the separate processing unit which connects the workstation with the robot. On the workstation, a C++ application can include libfranka and also ROS to communicate with other nodes. At this point it should be noted that in the present chapter the term "controller" is used for two different things: when speaking of the "robot controller" the controller on the RCU is referred to, while "ROS controller" indicates the controller which runs on the workstation PC, the left side in the above figure.

3.2 MoveIt

The motion planning framework MoveIt [17] provides various features for robot programming with ROS. It is easy to use and besides inverse kinematics and collision avoidance algorithms it also provides basic robot control such as point to point movement, where the user only specifies the trajectory's start and endpoint. The subordinated software plans the motion according to characteristics such as shortest length or fastest execution time. MoveIt can carry out these tasks while avoiding collisions according to its current collision model.

[11] available at `https://github.com/frankaemika/franka_ros`, last accessed on 20.02.2023
[12] `https://frankaemika.github.io/docs/franka_ros.html`, last accessed 24.08.2022
[13] `https://frankaemika.github.io/docs/overview.html`, last accessed 04.11.2022

Since in this work the robot movements should have minimal latencies to the movements of the simulation, elaborated motion planning far ahead in time is neither possible nor needed. What is needed instead are small point-to-point segments, which have to be chained together smoothly. This is where clear drawbacks of MoveIt show up. On the one hand ROS is not real-time capable which would, however, be absolutely necessary to be able to update the desired path in short and regular intervals. On the other hand, point-to-point movements with MoveIt have proven to stick to the given start and end point, but not to follow a straight line between those two points (even with no obstacle or singularity between them). Instead, the robot EE follows an unnecessarily curvy path, which is obviously unsuitable for the present application. Even if both of these problems could be fixed, a third one would remain: The robot must not stop after each given endpoint, but rather continue the movement to the next one smoothly.

So a continuous path controller is needed, which can be updated with new positions in real-time and hence follows that exact path and does neither more nor less. This does not fit the purpose of MoveIt and accordingly, at the time of this work, nothing has been found in the MoveIt documentation providing this functionality.

3.3 Impedance control

A second approach to achieve reliable path tracing is using *impedance control* [18]. Nowadays impedance control is one of the most successful forms of force control [19] and is therefore often used in interaction control. The robot is instructed to create a variable compliance and in return it creates a certain force onto the environment (the contact force [20]) if it is deflected from its reference pose, also called equilibrium pose by Hogan [21].

Hogan [18] stated that "[...] a manipulator may impress a force on its environment or impose a displacement or velocity on it, but not both". Generally spoken this means that every single physical system (in the case of robotics: the manipulator and the environment) can be classified either as an admittance or an impedance, with an admittance receiving a force (and/or torque) as input and having a corresponding motion (translation or rotation) as output. As its counterpart, every impedance accepts motion as input and applies a force on its surrounding. Note that those two types of systems complement each other: they are able to interact, which two systems of the same type cannot.

Applying this concept to the interplay between a manipulator and the environment it is in, one of them must be an impedance, while the other one is an admittance. As each object has an inertia and its movement may be restricted by kinematic constraints, the EE of a robot colliding with it or grasping it might not be able to perform the movement it is asked to make. Instead, while trying to do so, it applies a force that acts on the surrounding. Hence, from the manipulator's view, its environment is an admittance and must behave like an impedance to be able to interact [18].

So, opposed to conventional robot control, where only the robot's motion is controlled, or force control, where, as the name already states, only the force is controlled along a degree of freedom, Hogan[18] proposed a control strategy which combines those two principles thus creating a relation between pose and force. By adding a changeable impedance as a "disturbance response", interaction between the two systems may be controlled.

The simplest form of an impedance is a static relation between force f and displacement Δx: a stiffness k. Apart from the robot's hardware (torque and joint limits, link lengths), there is no restriction for Δx and thus there is no need for it to be small [21].

$$\mathrm{f} = k \cdot \Delta x \tag{3.1}$$

Basically, this is only Hooke's law for a linear spring. But as a system, e.g. a robot manipulator, might have not only one but multiple degrees of freedom, the scalar Δx is replaced by the vector $\mathbf{x}$ and, accordingly, the scalar constant k is replaced with the matrix $\mathbf{K}$. For example, $\mathbf{x}$ may have 6 entries, 3 translations and 3 rotations. Furthermore, when describing a dynamic system, also the derivatives of the positions should be considered. Thus damper constants $\mathbf{B}$ and an inertia matrix $\mathbf{M}$ are introduced as well. By doing so, the position vector needs to be time-dependent and is thus denominated $\mathbf{x}(t)$ in the following and describes the distance between robot EE and the desired equilibrium pose[21]. Therby we receive the oscillation equation for a spring damper system with inertia[19]:

$$\mathbf{f}(t) = \mathbf{M} \cdot \ddot{\mathbf{x}}(t) + \mathbf{B} \cdot \dot{\mathbf{x}}(t) + \mathbf{K} \cdot \mathbf{x}(t) \tag{3.2}$$

The inertia matrix $\mathbf{M}$ describes the masses of the single robot links and depends on the robot the controller is operating. Therefore it cannot be modified to change the behaviour of the impedance controller. To achieve that, only $\mathbf{B}$ and $\mathbf{K}$ may be edited. Those are often diagonal and thus decoupled matrices [22], as e.g. an applied force in one direction should not cause a movement in another direction, and are therefore easy to adjust manually. With Equation 3.2, the robot's Jacobian, and some coupling terms for acceleration and inertia the single torque values for the robot joints could be calculated [21], but as will be shown in the next chapter, for the Franka Emika Panda this has already been implemented by libfranka and will therefore not be further discussed here. Moreover, in the course of this calculation the gravitational force of the manipulator is compensated, which makes it stay in any possible configuration it has been brought.

In summary, it can be said that the principle of impedance control does not only allow smooth point-to-point control by continuously adjusting its equilibrium pose, as the compliance acts as a sort of filter for noisy reference data[23], but also allows for the displacement of the EE from its equilibrium pose by an external force: The further the EE is deflected, the bigger the control force, which the robot tries to return to its equilibrium pose with, becomes, which makes it behave like a real spring-damper system. Depending on the set stiffness, deflecting the manipulator becomes harder or easier. Therefore, impedance control is suitable in particular for human interaction with the robot, as the operating user can displace the robot simply with their hands. The question is, however, if this last characteristic is really desired for the project's use case. This will be looked at in the following sections.

3.3.1 Implementation

There are different methods of implementing impedance control. Lange [19] lists and explains two of them. The FCI provides a Cartesian impedance controller as part of the standard controllers, so it does not have to be implemented from scratch for this thesis.

The `CartesianImpedanceExampleController` [14] can be used almost as it is, with minor changes described below: A ROS-listener listens to a specific ROS-topic and sets every new incoming pose as the new EE equilibrium pose. Thus poses can simply be published to this topic. Depending on the set impedance, the robust behaviour of the impedance principle (Δx can be big on multiple axes simultaneously) allows for every valid pose to be reached by the robot EE within the next few seconds. If a new pose arrives before the previous one has been reached, the previous reference will be overwritten and thus ignored.

Latter fact allows fairly simple communication between Exudyn and the controller via ROS: For the robot to follow a given trajectory it is sufficient to publish the current Exudyn-pose as often as possible and the controller will always have the most recent one as its reference.

Coordinate transformation

In order for the robot EE to follow the path of the UIP of the MBS simulation and for the applied efforts to be applied to the simulation correctly, the UIP coordinates within the MBS coordinate system need to be mapped to the robot's base coordinate system (for the Franka Emika Panda robot: frame X_0, Y_0, Z_0 in Figure 3.1). Thus the transformation between them has to be found. The straightforward solution is to just find the transformation between the two frames, which are generally not aligned with each other. As this, however, has to be done manually for every single application, it is easier to define the coordinate system in Exudyn, which can be done at will, already in the first place with the same orientation as its real world counterpart. The ensuing rotation matrix between the two frames is the identity matrix and only the translation between them has to be determined. If the MBS has already been set up and the present thesis's work is implemented only afterwards, orientation can be adjusted with a separate rotation matrix afterwards, see subsection 6.1.2.

The orientations of the two systems now correspond with each other. The next step involves transforming relative translations of the Exudyn object to absolute robot world coordinates (i.e. in the robot base $\mathscr{K}_{\mathrm{RB}}$ frame). The UIP in Exudyn is defined initially relative to the MBS origin, so e.g. in the case of the example described in subsection 2.4.2 it is located at ${}^{MBS}[0,0,-l]^T$, with l being the length of the pendulum rod. The robot EE, however, may start at any position, in our tests it is often around ${}^{RB}[0.4,0,0.1]^T$, units in m. This can be seen in Figure 3.4. While the physical UIP on the robot and the virtual UIP of the MBS are overlapping, their coordinate origins are in totally different places.

So the initial pose of the robot EE must be acquired and the initial pose of the simulation model mapped to it. When the model moves, every other pose is then mapped to $\mathscr{K}_{\mathrm{RB}}$ coordinates as well. These coordinates are then be passed to the robot as a new command.

This transformation/mapping can be done either in the Exudyn client (Python) or in the ROS controller (C++). As the whole process only consists of a vector subtraction and is not very complex nor computationally intensive, for convenience it is decided to do it in Python.

[14] available at `https://github.com/frankaemika/franka_ros/blob/develop/franka_example_controllers/include/franka_example_controllers/cartesian_impedance_example_controller.h`, last accessed 12.01.2023

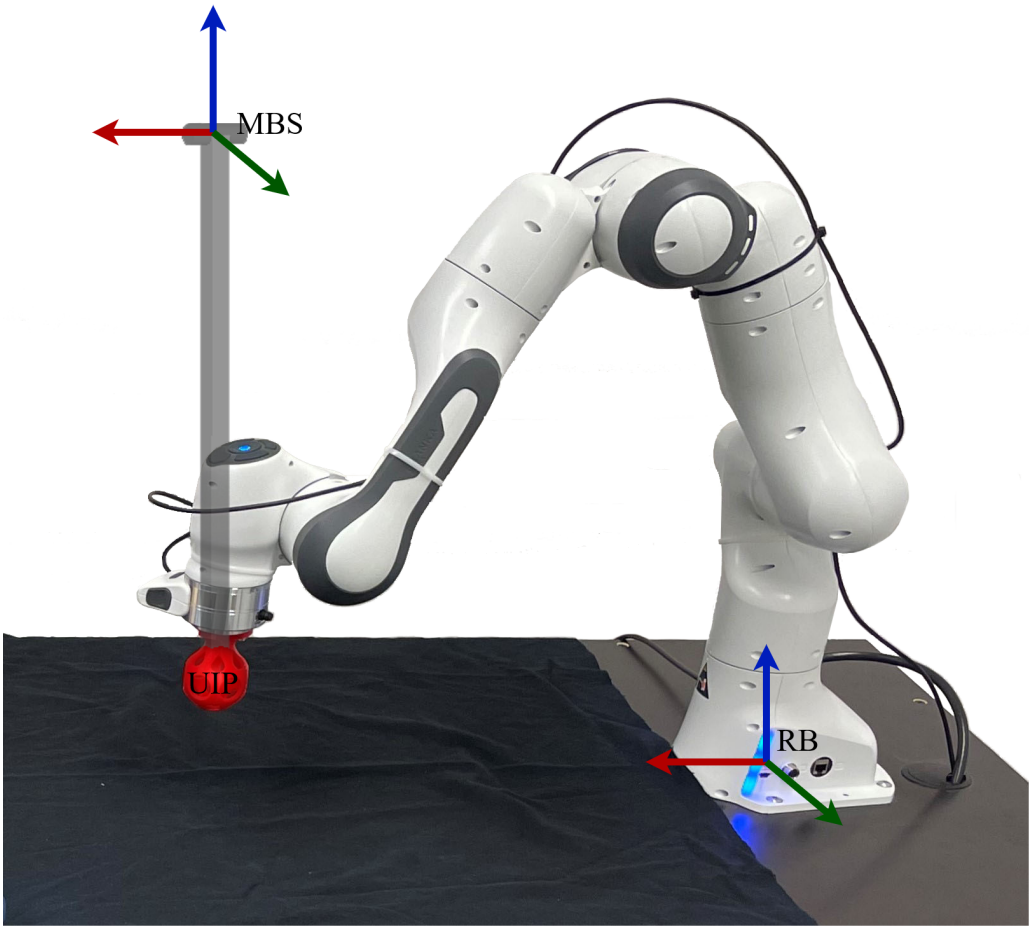


Figure 3.4: Robot in real world overlaid with simulated pendulum and origins of $\mathcal{K}_{\text{RB}}$ and $\mathcal{K}_{\text{MBS}}$. The UIP has different coordinates in the two coordinate systems.

Communication

To allow interaction between the robot and the Exudyn simulation, several communication channels are necessary. As stated above, before starting the motion, the Exudyn-client must store the Cartesian robot end-effector start position. Per default, the ROS controller does not provide that information on a ROS-topic but only the position of the current joint angles. With them, the EE position can be calculated with help of the forward kinematics. Although this information is not provided by ROS itself, it can, however, be extracted with the libfranka on the robot controller in the first place and then published to a custom ROS-topic. Since the ROS example controllers are configured in such way that the name of the respective controller is added at the front of the actual topic name, the topic is named `cartesian_impedance_example_controller/currentPose`. The message type used for this purpose is the `PoseStamped`-message, provided by the `geometry_msgs`-ROS-package. It holds a field for a pose (three floats for position and four floats for orientation in form of a quaternion with the scalar part at the end of the vector) and a field for a header, where, among others, a timestamp can be stored. This timestamp is not necessary for functionality, but rather for recording and analyzing the travelled

trajectory afterwards. The Exudyn-client unsubscribes from the named topic once it got the EE starting position at the beginning of the process. For other nodes the topic is still available.

In addition to the current position, the controller must also pass on the current force and torque acting on it to the Exudyn-client. As this is already provided by franka_ros within the `/franka_state_controller/F_ext`-ROS-topic, the efforts can be taken from there. Unlike the previously described custom `current-pose`-message, which is sent once per controller iteration and thus with a frequency of 1 kHz[15], the efforts are published only with a frequency of about 30 Hz.

The efforts are registered and published in the so called k-frame[16], which here coincides with the one of the EE. Since the absolute EE orientation is irrelevant for the application of the present thesis, the efforts must be transformed to the robot base frame.

Furthermore, it should be pointed out that the published efforts are the ones the robot needs to apply to compensate external efforts. They are therefore always opposed to the acting efforts and their signs must be inverted when they are mapped onto the simulation model.

As described above, while the simulation model receives efforts, the controller must receive the new reference position from the Exudyn MBS simulation in the opposite direction. The functionality to accept such messages is already given by the `CartesianImpedanceExampleController`: The controllers listen to the `/cartesian_impedance_example_controller/equilibrium_pose`-ROS-topic and if a new message arrives, the corresponding callback method (`equilibriumPoseCallback()`) updates the respective class attributes

```
Eigen::Vector3d position_d_reference_;
Eigen::Quaterniond orientation_d_reference_;
```

This concept is adapted to create the new customized impedance controller named `ExuCobotImpedanceController`[17], which includes some minor changes to the provided example controller:

- Some class methods and attributes are renamed. For example, the *equilibrium pose* is renamed to *reference pose*, as the main purpose of the controller in this context is to follow a trajectory, not to provide soft user interaction.
- Logging functionalities are implemented to be able to analyze the commands afterwards. This happens with log files in CSV-format.

Those changes, however, do not affect the behaviour of the controller.

Parameters & results

The last missing part before being able to use the controller is to set the parameters. As can be seen in Equation 3.2, several different components are required to define the

[15] `https://frankaemika.github.io/docs/overview.html`, last accessed 04.04.2023

[16] `https://frankaemika.github.io/libfranka/classfranka_1_1Robot.html#k-frame`, last accessed 04.04.2023

[17] source code available at `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_example_controllers/src/exucobot_cartesian_impedance_controller.cpp` and corresponding header file, last accessed 26.04.2023

behaviour of the controller properly: the mass of the system (the manipulator) and the damper and spring constants. The masses of the single robot links are already known to the robot and thus only the mass of the mounted EE (section 6.2) must be set in Franka Desk, the local web interface of the robot, as this can be changed by the user at any point. This mass should not be adapted to influence the robot's behaviour and therefore the only way to change it is to modify the damper and spring constants. They are defined in a local parameter file[18] and amount per default to

$$\mathbf{K}_{trans} = \begin{pmatrix} 200 & 0 & 0 \\ 0 & 200 & 0 \\ 0 & 0 & 200 \end{pmatrix} \frac{\mathrm{N}}{\mathrm{m}}, \tag{3.3}$$

$$\mathbf{K}_{rot} = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{pmatrix} \frac{\mathrm{N\,m}}{\mathrm{rad}}, \tag{3.4}$$

$$\mathbf{B}_{trans} = \begin{pmatrix} 28.2843 & 0 & 0 \\ 0 & 28.2843 & 0 \\ 0 & 0 & 28.2843 \end{pmatrix} \frac{\mathrm{N\,s}}{\mathrm{m}}, \tag{3.5}$$

$$\mathbf{B}_{rot} = \begin{pmatrix} 6.32456 & 0 & 0 \\ 0 & 6.32456 & 0 \\ 0 & 0 & 6.32456 \end{pmatrix} \frac{\mathrm{N\,m\,s}}{\mathrm{rad}}. \tag{3.6}$$

As mentioned in section 3.3, the single axes are often decoupled and hence represented in diagonal matrices, as there should be no direct coupling between applied forces on one axis and movements on another. This is also the case here, whereby the translational part is clearly higher for both the stiffness and the damping.

A first test with these default parameters is the analysis of the step response. For that purpose, the controller is tasked to change the robot's EE reference pose every 5 s. To this effect, two positions are defined between which the robot should oscillate. They are located 10 cm apart from each other with constant orientation.

Figure 3.5 shows the step response on the x-axis, the positions on the other two axes remain the same for both poses. Two major issues can be detected at first glance: Neither does the robot come close to its reference position, nor does it reach the movement end position in a reasonable amount of time. As soon as the reference position changes, the robot immediately starts moving towards it, but the closer it gets, the smaller its effort becomes to continue moving. This leads to the robot slowing down and coming to a rest already 0.02 m before the reference and about 1.4 s after starting the movement. This deviation is probably due to joint friction and measurements inaccuracies of the robot. It must also be noted that the impedance controller with its spring and damper constants corresponds to a PD controller, which has no integral component that could compensate for a permanent control deviation. In general this controller corresponds to a first order lag (PT_1) element. Its step response is shaped like the inverse of an exponential function and reaches 63% of its step distance after a certain amount of time T_1. This time constant characterizes the system and for the default compliance parameters it amounts to $T_{1d} = 0.546\,\mathrm{s}$.

[18] `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_example_controllers/cfg/compliance_param.cfg`, last accessed 16.02.2023

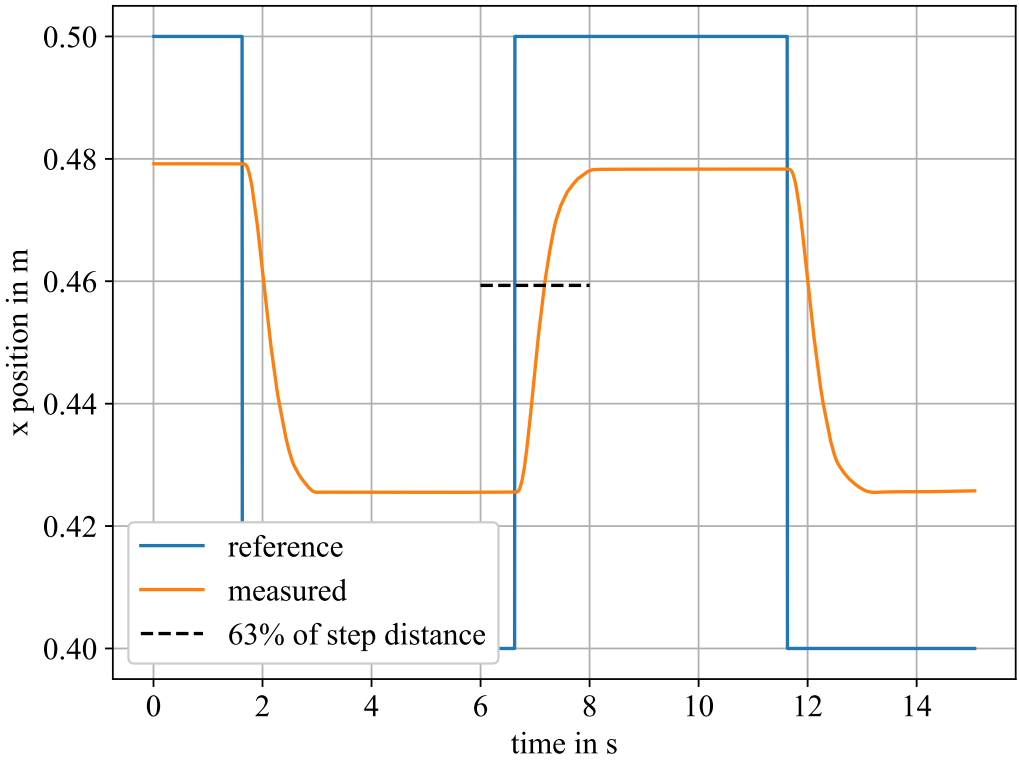
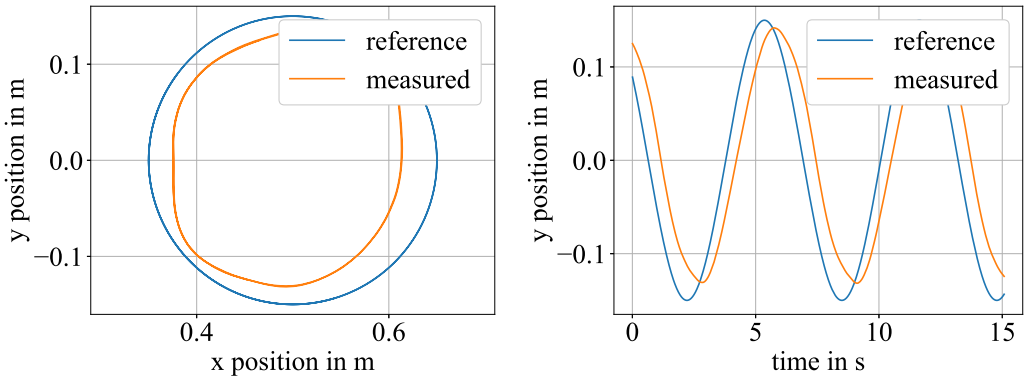


Figure 3.5: Step response of impedance controller with default parameters.

The difference between the positioning errors at the two different reference poses are probably due to the initial joint configuration of the robot, friction within the joints or inaccuracies at the mass compensation of the robotic arm (with a custom end effector build-up and a cable which is mounted along the arm for another application). Besides the noticeable differences in positioning accuracy at the two positions one can observe that these inaccuracies are almost the same again when the reference trajectory is repeated.

To look at this phenomenon in more detail, another test was made with the same parameters, but this time a circle trajectory ($z = const$, $r = 0.15\,\text{m}$) is sent to the controller and again reference and measured trajectories are recorded. The results are shown in Figure 3.6. Figure 3.6a shows that the travelled trajectory is nowhere near its reference: Neither does the figure have the smaller distance to the circle center point than the reference trajectory, nor does it have the basic shape of a circle. This is due to the positioning inaccuracies already noticed at step response. Comparing the two graphs, both based on the same measurement, it can be seen in the second graph that not only one, but more than two circumnavigations have been recorded, although in the left graph only one is visible. This shows anew that the inaccuracies are repeatable and indicates that they depend on the current robot joint configuration.

The temporal delay between reference and measured trajectory, which is visible in Figure 3.6b and is about 0.48 s, measured at the zero crossings, also has an effect on the inaccuracies. Assuming that at one point in time the robot is exactly in its reference position, the robot's effort to change its pose changes only slowly when the reference



(a) y-x-graph (b) y-t-graph

Figure 3.6: Reference and measured circle trajectories of robot with impedance controller with default parameters. The data used for both graphs is the same.

changes continuously and not abruptly, as it is the case with the step response. This is because at the beginning the distance between reference and current pose is only small and additionally the damping characteristics prevent sudden changes. Hence, while the reference trajectory is moving on, the robot might still be standing still. When the distance between current and reference pose is finally sufficient for the effort to be high enough to move the robot, the movement is significantly behind its reference. The same happens when there is a change in direction. This can be observed in the y-t-graph. Once the reference trajectory reaches its lower or upper limit and begins moving in the opposite direction, the robot slows down, as the distance to its reference decreases, but does not accelerate in the new direction until the reference is far enough away for the effort to be high again.

Given the fact that the default parameters are unsuitable for precise trajectory movements, they are increased simultaneously step by step: The stiffness parameters to achieve faster reaching of the reference pose, the damping parameters to prevent overshoots due to higher stiffness. Due to its physical characteristics, damping prevents rapid changes in velocity, which is necessary as the trajectory needs to stay within the robot's hardware limits. However, e.g. when looking at the step response, the damping parameter should be kept small to facilitate reaching the reference fast. These values should therefore be as small as possible, but as high as necessary. Such configuration is found empirically at

$$\mathbf{K}_{trans} = \begin{pmatrix} 3600 & 0 & 0 \\ 0 & 3600 & 0 \\ 0 & 0 & 3600 \end{pmatrix} \frac{\mathrm{N}}{\mathrm{m}}, \tag{3.7}$$

$$\mathbf{K}_{rot} = \begin{pmatrix} 80 & 0 & 0 \\ 0 & 80 & 0 \\ 0 & 0 & 80 \end{pmatrix} \frac{\mathrm{N\,m}}{\mathrm{rad}}, \tag{3.8}$$

$$\mathbf{B}_{trans} = \begin{pmatrix} 80 & 0 & 0 \\ 0 & 80 & 0 \\ 0 & 0 & 80 \end{pmatrix} \frac{\mathrm{N\,s}}{\mathrm{m}}, \tag{3.9}$$

$$\mathbf{B}_{rot} = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{pmatrix} \frac{\mathrm{N\,m\,s}}{\mathrm{rad}}. \tag{3.10}$$

whereby the aim is to increase the translational stiffness and use the damping only to compensate jerky movements. Higher values for stiffness do not result in faster reaching of the reference. On the contrary it is observed that higher, i.e. more aggressive, control parameters, shortens the distance that can be covered by means of a single change of the reference position, as the robot controller crashes on larger distances, presumably because given commands would force the robot to exceed hardware limits. The parameters still work for the previous experiment, which is repeated and the result shown in Figure 3.7. The remaining deviation from the reference position is significantly smaller, it is about 1.4 mm for both reference poses, which is considered acceptable since high accuracy is only of low priority because users cannot perceive such small deviations in this use case. But still, the time it takes the controller to get into the equilibrium pose is approximately 1.7 s and hence even 0.3 s longer than with the default parameters. This is due to the longer way it has to cover. However, the time the system with custom parameters takes to reach 63% of its step distance amounts only to $T_{1c} = 0.224\,\mathrm{s}$, which corresponds to about 41% of T_{1d} from the default compliance parameters. The small jerk before reaching the end position is due to the robot's joint configuration. The joints are supposed to move linearly and synchronously from their start positions to their reference positions. In practice, however, linearity and synchronicity may not be fully given, which is why the movement is not completely smooth.

Also the circle trajectory, given beforehand to the controller with default parameters, is sent to the controller with customized parameters. As is shown in Figure 3.8, the increased parameters have a huge impact on temporal delay, which, measured at the zero crossings on the y-t-chart, is now at about 0.23 s, thus halved with respect to the default parameters, as well as on the robot's ability to reach extreme poses. The two features are closely related to each other: when the delay is small, the robot controller is more likely to reach its reference poses. Also the circle on the y-x-graph looks more roundish and even though not being perfect, it meets the requirements for this work.

Force filtering

In the next step the robot and the simulation are coupled via ROS. The force registered by the robot at its EE should influence the simulation and accordingly the motion of the robot. But when looking at the force autonomously published by the ROS controller, it can be noted that even without an external force being applied to any part of the manipulator, the published force on the topic `/franka_state_controller/F_ext`, so the wrench vector, called effort in the ROS context, is not zero. According to the messages on the topic, on all axes forces and torques are registered. This might be due to imprecise measurements and wrong parameters, e.g. wrong weight of the EE mount. To prove that those deviations are not specific to a certain EE pose, the effort is recorded in different robot configurations for 3 s each and the mean values are listed in Table 3.1. The standard deviation values are between 0.001 N m and 0.005 N m for the torque measurements and between 0.007 N and 0.03 N for the force measurements.

The used EE poses differ in both position and orientation from each other and the resulting recorded efforts vary widely between them. If the controller and the simulation

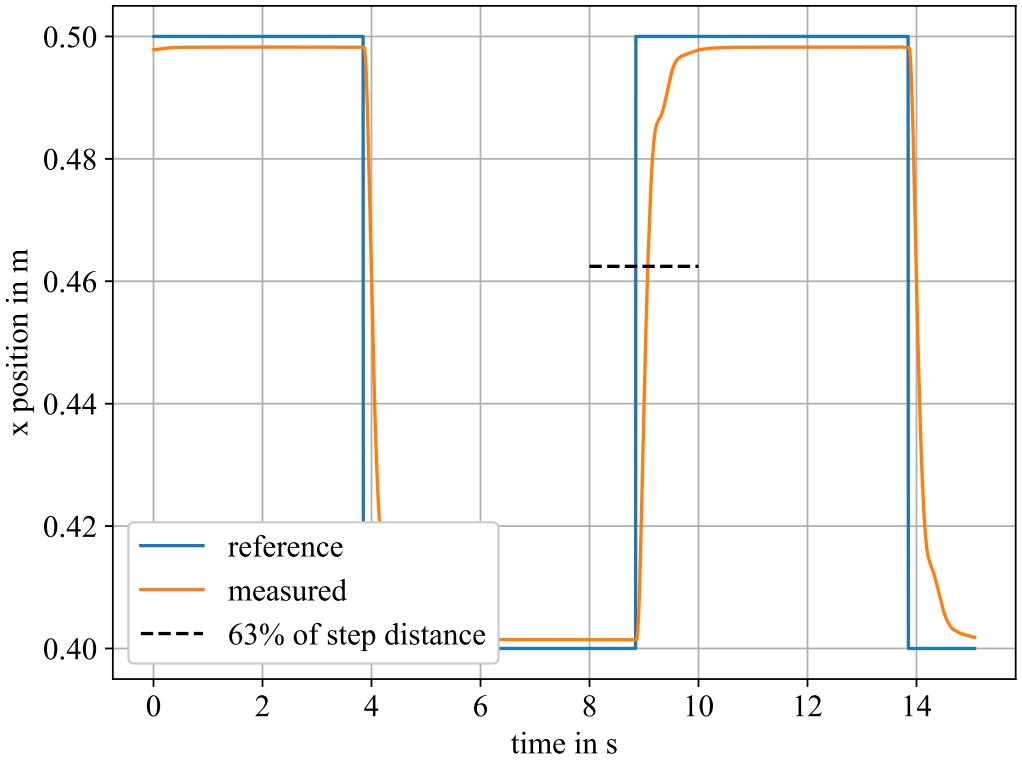
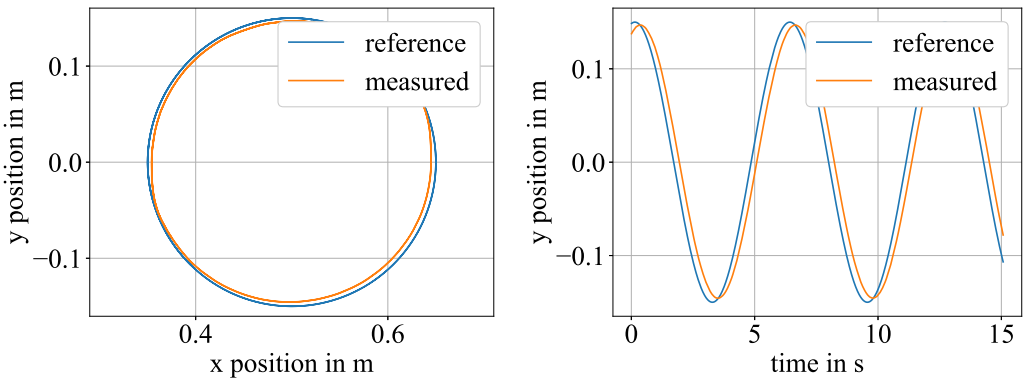


Figure 3.7: Step response of impedance controller with custom parameters.



(a) y-x-graph (b) y-t-graph

Figure 3.8: Reference and measured circle trajectories of robot with impedance controller with custom parameters. The data used for both graphs is the same.

model were coupled without filtering these efforts, the simulation would interpret those nonzero values as user inputs. A potential outcome is depicted in Figure 3.9, where the robot measures an initial static force of about $f_y = -0.6\,\text{N}$. When the simulation is started at around 2.5 s into the measurement, this static force offset causes a slight deflection of the pendulum. This deflection increases the measured force, which leads to a swing up of the pendulum and an increased movement of the real robot system.

The static effort offsets are different for each robot EE pose, so simply adding f =

	f_x	f_y	f_z	τ_x	τ_y	τ_z
pose 1	0.888	-0.788	0.960	-0.159	0.181	0.179
pose 2	1.415	-0.022	0.775	0.136	-0.227	-0.036
pose 3	0.482	1.616	-1.646	0.628	-0.166	-0.011

Table 3.1: Measured force (in N) and torque (in N m) offsets at different EE poses (listed in Table 3.2).

	j_0	j_1	j_2	j_3	j_4	j_5	j_6
pose 1	-0.079	0.696	0.132	-1.692	-0.161	2.345	0.084
pose 2	0.120	0.505	-0.099	-1.916	0.038	2.462	-0.041
pose 3	-0.234	-1.417	-0.363	-1.753	-0.948	1.233	-1.031

Table 3.2: Joint configurations in rad for poses in Table 3.1.

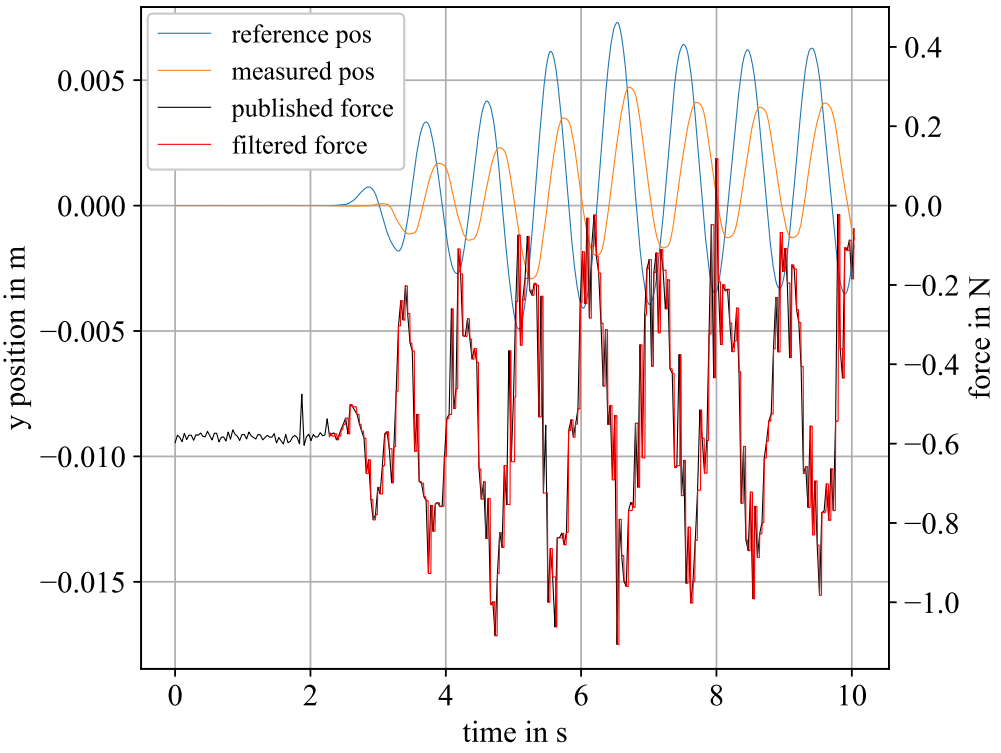


Figure 3.9: Swing-up of simulation without force filtering.

0.6 N to the measured force does not solve the problem. Instead, another way must be found to remove these variable offsets.

To this purpose, on the one hand the static offset must be eliminated and on the other hand the noise needs to be filtered. Both objectives are reached by applying a calibration strategy. Before starting the simulation the measured efforts are recorded for a certain period of time (e.g. 5 s) and from those values the mean is calculated. The mean, maximal, and minimal values of the recordings are then stored for each degree of freedom separately(six in total: 3x force for translation along x, y, z; 3x torque for

rotation around x, y, z). Since these efforts are measured and published to the ROS-topic already by default with no involvement of the impedance (or any other) controller, this calculation is done in Python on the simulation side and not on the ROS controller in C++. When the simulation starts, the respective mean value is subtracted from every incoming effort to get rid of the static offset. Furthermore, to eliminate the noise, a threshold $c_{th,i}$ is precalculated for every degree of freedom $i \in \{1,...,6\}$:

$$c_{th,i} = (\max(\text{data}_i) - \min(\text{data}_i)) \cdot S \tag{3.11}$$

with S being a safety factor empirically set to $S = 10$. This is required due to the fact that self-induced motion creates even bigger measured forces. Therefore, if the magnitude of the offset-corrected effort is inferior to this threshold, it is set to 0.

As mentioned before, at this point the efforts are negated to correspond to the actual acting effort.

3.3.2 Results

The force input being under control, the combined system can be tested with the two dimensional pendulum simulation (subsection 2.4.2) for the first time. Due to limited computation time, new poses are not sent in each iteration but only about every 6 ms, thus with a frequency of $f \approx 167$ Hz unlike the frequency of $f = 1$ kHz as applied for the circle trajectories before. The reference poses thus have bigger distances between them, but because of the nature of the impedance controller, this does not perceptibly affect the outcome. Figure 3.10 shows the timeline of input forces and EE positions on the y-axis during the experiment, which represents a decay test of the oscillator. There is no movement in the x-axis, as it is locked by the simulation settings, and only small movements in z direction, which are similar to those in y direction, but much smaller in magnitude. The orientation of the EE (rotation around x, y, z) is also ignored.

Comparing this chart to Figure 3.9, it can be noted that the force is now filtered, which results in absolutely no force at the beginning of the test when the manipulator is standing still and experiences no external forces. Also the forces, self-induced by the robot while oscillating, are neglected. It becomes evident why the safety factor has to be set so high, as the induced forces might be relatively big ($|f| > 2$ N in this example) for actually no force applied to the robot by the user. However, in absolute terms, 2 N are significantly less than the efforts registered when the user is indeed applying them, thus clipping the force with a threshold is thought to be acceptable.

Looking at the robot positions, it is found that reference and measured positions are not synchronized very well. The measured robot position seems to have a larger delay than at the circle-test with the customized impedance parameters (Figure 3.8) and hence does not reach the maximal and minimal reference trajectory positions. The delay, however, is actually only slightly larger, being at 0.27 s compared to 0.23 s before. What has changed in contrast to the circle trajectory is the oscillation period. While it was $T_{\text{circle}} = 6.28$ s then, the quasi-period is at $T_{\text{pendulum}} = 1.4$ s now. This shows that the controller had significantly more time to reach the reference positions at the circle example.

Figure 3.11 shows a detailed view of Figure 3.10 when the robot starts moving for the first time. It is clearly visible that the measured EE position is deflected before the

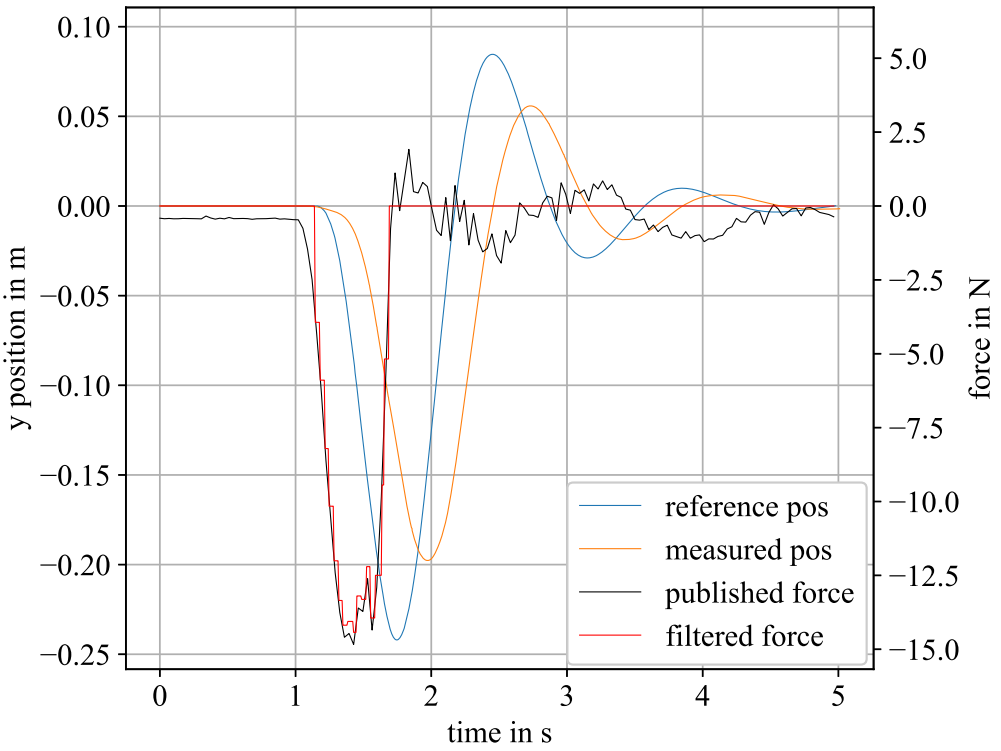


Figure 3.10: 2D-pendulum with impedance controller.

reference position changes, even before the applied force is large enough to be registered by the simulation. In fact, the robot starts moving as soon as an external force is applied to it. This is due to the characteristics of an impedance controller which allows user interaction and takes its reference as an equilibrium pose. This means that the compliances of the two systems, the robot controller on the one hand and the simulation model on the other, are superimposed. Once the applied force is big enough to exceed the threshold, which happens at around $t = 1.13\,\text{s}$, the simulation reacts, and commands the robot to move. Due to the slow reaction time of the impedance, which has already been observed in previous charts, the measured position is surpassed by the reference position. It is only once the distance between them is big enough that the manipulator begins to accelerate. This happens at $t = 1.3\,\text{s}$.

Due to above-mentioned high latency another problem occurs when the user tries to interact with the robot, i.e. with the simulation, while it is oscillating. The following assumption may help clarify the problem. The user, not wearing an HMD at the moment because the application does not make sense with two asynchronous systems, is not aware of the simulation itself, only of the robot's movement. The user interacts with the robot (= with the pendulum) after the oscillation of the simulation has already been started. With reference to Figure 3.10, the user might want to stop the oscillation right after starting it and therefore applies a force against the robot's movement shortly before $t = 2\,\text{s}$. At this point the robot is about to reach its maximum position in negative y-direction. So, the force applied to slow the movement down is applied in positive y-direction. However, at exaclty the same moment in time the simulation is already moving in positive y-direction,

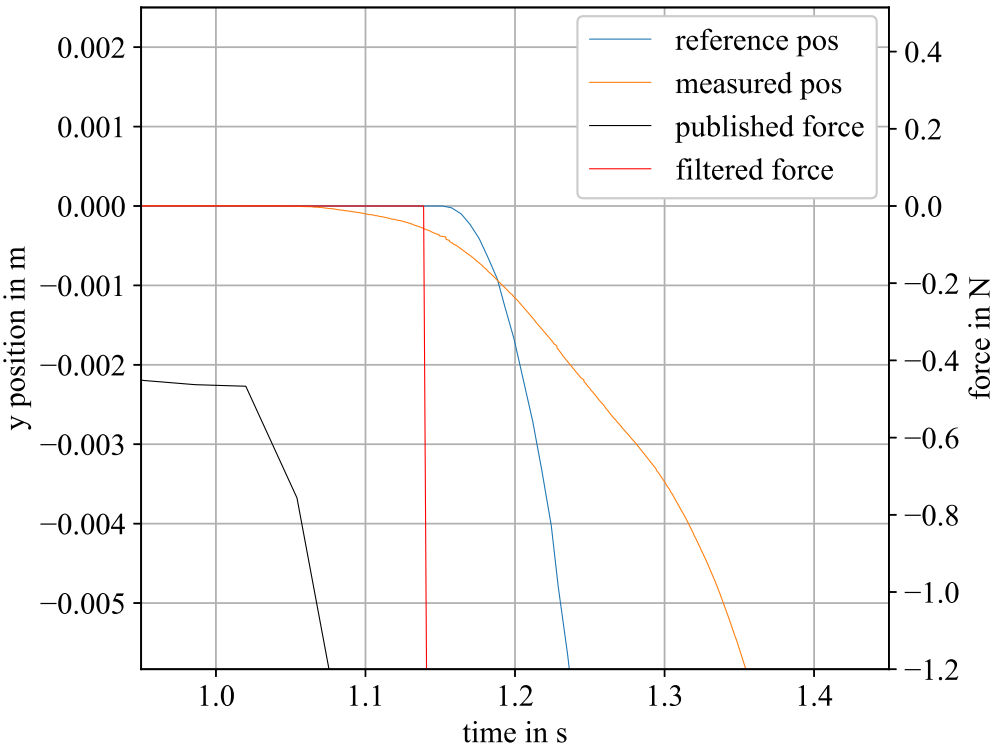


Figure 3.11: Detailed view of Figure 3.10.

the additional force thus amplifying the oscillation instead of slowing it down.

3.3.3 Conclusion

According to the above observations it can be said that using an impedance controller has advantages and disadvantages. First of all, it is suitable for user-robot interactions, as the built-in compliance reduces the risk of injuries to the user due to collisions, although it is clearly no substitute for other safety measurements. However, as in the final version the user will be wearing an HMD and therefore will not be able to see the robot directly, it generally increases the safety of the application. What is more, the controller is easy to program since only a number of parameters and an arbitrary reference pose are required. Even if that pose is far away from the current robot manipulator position, with the correct parameters the robot will manage to get there. Furthermore, the impedance controller is very robust to latencies in the ROS network, which will be addressed in subsection 3.4.2. So for example when a new reference position does not arrive on time, it keeps the previous one as current one. High latency makes it unlikely for the robot to have already arrived at the reference position. The robot keeps moving.

On the other hand there are also three major disadvantages. While in some cases of collaborative robotics the soft characteristics of the impedance controller might be suitable to decrease the robot's potential of harming the user or its surroundings in general, in the case of the present thesis it is not desired, as it overlaps and interferes with the compliance of the simulated model. When a force is applied on the EE, the robot should move only according to the constraints of the simulation software and not according

advantages	disadvantages
suitable for user interaction	inaccurate, unusable for small movements
easy to program	slow response time, big delay
robust to network latencies	compliance of the robot controller overlaps with compliance of the simulation model

Table 3.3: Advantages and disadvantages of impedance control in this application.

to the stiffness of the controller. Furthermore, temporal delay and a certain inaccuracy are implied by this soft characteristic which diminish the feeling of really being in the simulation and being able to influence it.

Table 3.3 lists the advantages and disadvantages in shortened form. As slow response time and compliance overlap lead to a considerably reduced user experience, it is decided to test a position/velocity controller for this application.

3.4 Velocity control

Two of the main disadvantages of the impedance controller are inaccuracies and compliance. In contrast to an impedance controller, a classic position controller sends angular positions for all joints to the robot, where a PID-controller brings the robot into the desired configuration by regulating its currents [24]. The described principle is quite old and even though more advanced controllers exist, a PID-controller can often still be applied when the primary goal is to follow a path [25]. This is also the case with the Franka Emika Panda Robot[19]. A controller of this type would therefore solve two of the three disadvantages in Table 3.3, with only slow response time / big delay still being an issue.

However, for human-robot interaction Zelenak et al. [24] suggest using another controller to improve user experience. They state that using a velocity controller instead of a position controller shows decreased contact forces, which results in the robot movement feeling more natural to the user. The transition from position controller to velocity controller is simple since the FCI offers the possibility to send the desired joint (or Cartesian) velocities instead of the joint positions (or Cartesian poses) as commands to the robot and interpret them as such. Once a trajectory is given it can be realised by sending the path's first derivative.

3.4.1 Trajectory interpolation

As it was already the case with the impedance controller, also the velocity controller is realized as an open loop controller. A current velocity/position feedback could be introduced, which, however, is not mandatory according to the implementation in libfranka. The single commands only need to be continuous to each other. Also, while the PID controller itself is already implemented on the RCU, it is the main task for the application programmer to provide it with new data. And it is here that the main difference between the impedance and the velocity controller becomes evident. While it is acceptable for the first to receive a new pose every few milliseconds, with these poses potentially being

[19] `https://wiki.ros.org/ros_control` last accessed 04.11.2022

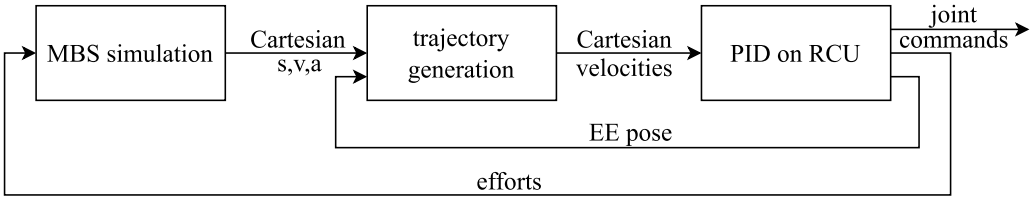


Figure 3.12: Block diagram for controlling a robot from an MBS with velocity control.

significantly separated from each other, this is not acceptable for the latter one. For every control cycle (1 ms) exactly one new command is expected and starting from the previous velocity, it must be physically feasible, i.e. within the velocity, acceleration, and jerk limits of the robot joints, to reach the new velocity within one control cycle[20].

Just replacing the previously used impedance controller with the velocity controller is therefore not enough since the updates the robot would receive would be sent at the correct frequency. However, as the sending frequency cannot be increased (see section section 5.3), another solution to the problem has to be found. A principle that is able to cope with this problem is fine interpolation [26]. It originally comes from CNC-manufacturing but is also applicable to other use cases in robotics. The principle consists in splitting the interpolation process into rough and fine time steps. An external controller might not be able to interpolate within the required time period, either because of computational hardware limits or because it lacks the information necessary to compensate small errors in the effective trajectory and to provide high dynamic responsiveness to them. In that case it is only tasked with rough interpolation, while fine interpolation is performed afterwards by another control unit on machine-side during runtime.

Control algorithms, which are executed during runtime, are called *online*. Thus a trajectory generation performed during runtime is called online-trajectory-generation (OTG) and is generally used to react to unpredictable events by calculating a new valid trajectory[27] starting from the current position, velocity, and acceleration. In order to be able to operate in real-time, they must therefore be kept simple and well implemented.

In the context of this thesis, the MBS simulation software provides an approximate trajectory by publishing a new position and the corresponding derivatives every few milliseconds, thus representing rough interpolation. The ROS controller ensures that the robot receives a new valid command every millisecond. Its concrete task is therefore to interpolate / generate a trajectory between them for the robot to be able to get from one pose to the other within the specified time.

This control principle can be seen in Figure 3.12. The simulation program simulates an MBS and sends the Cartesian position, velocity, and acceleration (3x3 `double` values) of the UIP to the trajectory generation implemented on the ROS controller, which interpolates between them and sends the EE's (Cartesian) velocities (6 `double` values) with a frequency of 1 kHz to the PID on the RCU, which transforms the 6 Cartesian velocities to the corresponding 7 joint commands for the Franka Emika Panda robot. In return, the RCU passes the current EE pose back to the trajectory generation to enable the correction of positioning errors that might occur. It also sends the measured efforts to the MBS simulation to enable MBS reaction accordingly.

[20] `https://frankaemika.github.io/docs/libfranka.html`, last accessed 22.12.2022

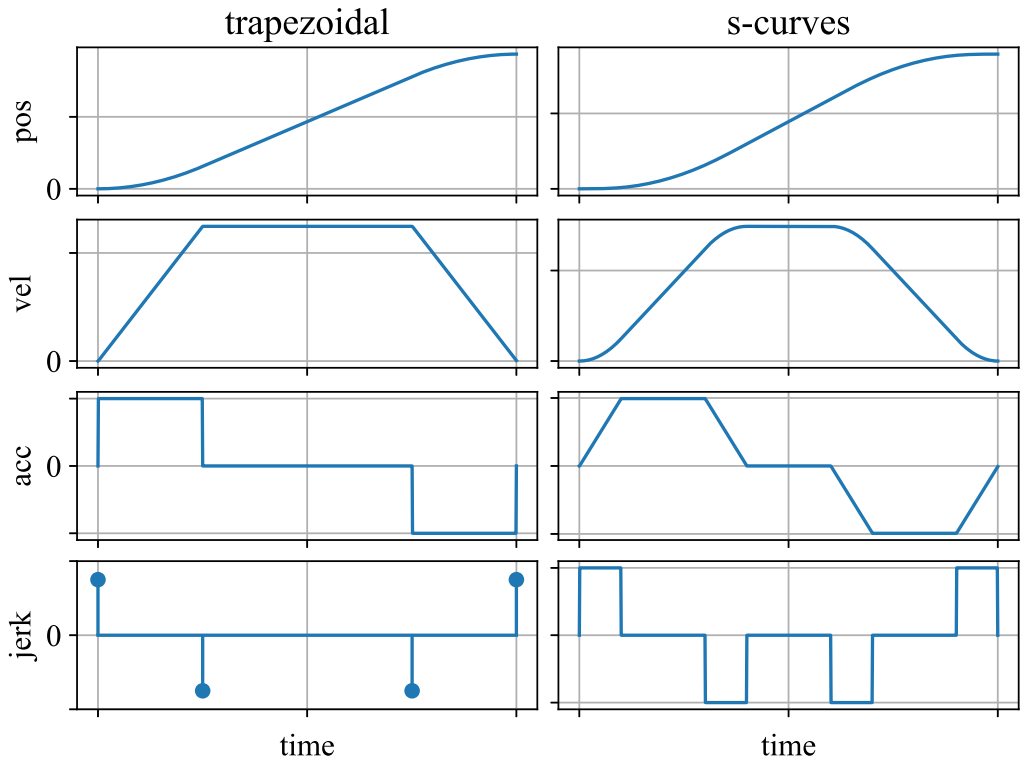


Figure 3.13: Qualitative difference between trapezoidal and s-curves motion profiles. Units and absolute numerical values are neglected.

S-curves motion profiles

An established method to create a trajectory for a manipulator between two given poses is to use so called *s-curves*. Whereas s-curves are similar to trapezoidal motion profiles, here a constant jerk is applied to change acceleration, velocity and position rather than constant acceleration only changing velocity and position [28]. In other words: the trajectory must be differentiable one more time.

Figure 3.13 shows the difference between trapezoidal and s-curves motion profiles. As can be understood from the figure, *trapezoidal* refers to the velocity profile and thus gives this concept its name. Also the s-curves have trapezoidal profiles, but because of the additional integration the trapezoids represent acceleration. However, the main purpose is to get rid of the pulse-shaped jerk profile to prevent the acceleration value from jumping, which is necessary e.g. for operating the Franka Emika Panda robot. Nguyen et al. [29] take it even one step further by proposing to repeat this process of integrating the trajectory multiple times and thus smooth it out even more. This approach is not necessary for this thesis, as smoother trajectories also come with a disadvantage: the time needed to cover a certain distance increases if maximal acceleration and velocity stay the same.

S-curves are used by authors for online trajectory generation, e.g. Huber and Wollherr [30], who have developed an OTG-algorithm especially for HRC. However, this algorithm prioritizes human comfort and safety and although specifically the latter is extremely important, in the case of this thesis this is unsuitable, as it would prevent the

robot from following the simulation's path precisely. A more general approach to trajectory generation with s-curves is taken by Berscheid and Kröger [31], who do not apply their algorithm to collaborative robotics specifically but to trajectory generation in general. Their algorithm takes a robot with multiple degrees of freedom from an arbitrary kinematic start state[21] to an arbitrary kinematic target state. As for most s-curves motion profile algorithms, also this one calculates time optimal trajectories, i.e. where the target state is reached in a minimal amount of time [32]. Therefore this algorithm is neither suitable for usage in this thesis. What is instead required here, is to pass from one pose to another within an exactly defined period of time. To realize this, the maximally applicable jerk[22] would need to be scaled down in order to decrease acceleration and velocity and so to slow down the movement and increase the required time. One way to achieve this is to extend the algorithm of Berscheid and Kröger [31] to include variable maximum jerks, accelerations, and velocities. But since even without this extension the algorithm is fairly complicated, it was decided to implement a less complex method, described in the next section.

Polynomial Interpolation

An alternative method to interpolate between two points is polynomial interpolation. Depending on the application, different polynomials can be used, usually they are of third (cubic) or fifth (quintic) degree [33]. Depending on their degree they have a specific amount of coefficients b_i ($i \in \{0,...,3\}$ for third degree, $i \in \{0,...,5\}$ for fifth degree) and can therefore be applied in different use cases. Since the time derivations of a quintic polynomial are of higher degree than those of a cubic polynomial, a quintic polynomial with its derivatives can be assumed generally smoother than a cubic one and is thus used in the present project.

$$f(x) = b_5 \cdot x^5 + b_4 \cdot x^4 + b_3 \cdot x^3 + b_2 \cdot x^2 + b_1 \cdot x + b_0 \tag{3.12}$$

Equation 3.12 shows the basic structure of a polynomial of fifth degree. With the six unknown parameters b_i six independent conditions, which can be chosen as desired, are necessary to define the polynomial unambiguously. If the above equation is interpreted as the trajectory $s(t)$ itself and thus the variable x is replaced with the time t, the trajectory can be written as shown in Equation 3.13. The corresponding velocity $v(t)$ and acceleration $a(t)$ along the trajectory are obtained by deriving the equation by respect of t once and twice, respectively.

$$s(t) = b_5 \cdot t^5 + b_4 \cdot t^4 + b_3 \cdot t^3 + b_2 \cdot t^2 + b_1 \cdot t + b_0 \tag{3.13}$$

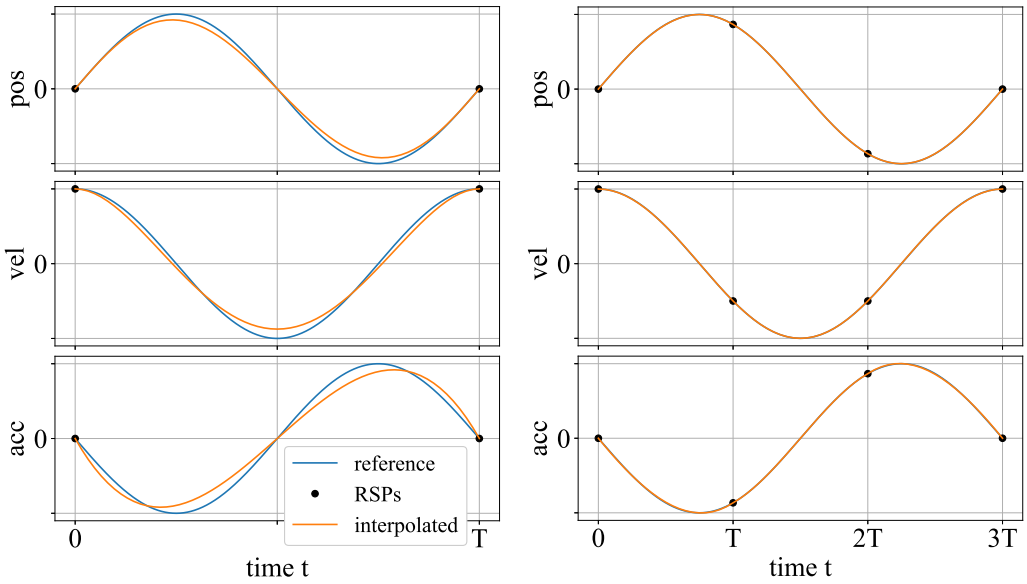
$$v(t) = 5 \cdot b_5 \cdot t^4 + 4 \cdot b_4 \cdot t^3 + 3 \cdot b_3 \cdot t^2 + 2 \cdot b_2 \cdot t + b_1 \tag{3.14}$$

$$a(t) = 20 \cdot b_5 \cdot t^3 + 12 \cdot b_4 \cdot t^2 + 6 \cdot b_3 \cdot t + 2 \cdot b_2 \tag{3.15}$$

With the three equations and the two given states

[21] A (kinematic) state is defined by the position and its time derivatives up to third order [31], i.e. velocity, acceleration and jerk

[22] for Franka Emika Panda available at `https://frankaemika.github.io/docs/control_parameters.html`, last accessed 14.03.2023



(a) 1 sampling point per period. (b) 3 sampling points per period.

Figure 3.14: Polynomial interpolation of sinusoidal curve with different amount of sampling points. Units and absolute numerical values are neglected.

$$\begin{aligned} s(0) &= s_0, & s(T) &= s_T, \\ v(0) &= v_0, & v(T) &= v_T, \\ a(0) &= a_0, & a(T) &= a_T \end{aligned} \tag{3.16}$$

at $t = 0$ and $t = T$ respectively, each consisting of a position, a velocity and an acceleration, six equations can be formed and thus the parameters b_i be determined unambiguously [34]. This is done by solving the resulting linear equation system given in Equation 3.17. The time of the first state being equal to 0 considerably simplifies the upper half of the matrix. This solving process must be done for every single degree of freedom separately.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & T & T^2 & T^3 & T^4 & T^5 \\ 0 & 1 & 2T & 3T^2 & 4T^3 & 5T^4 \\ 0 & 0 & 2 & 6T & 12T^2 & 20T^3 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} = \begin{pmatrix} s_0 \\ v_0 \\ a_0 \\ s_T \\ v_T \\ a_T \end{pmatrix} \tag{3.17}$$

Once the polynomial is defined, it can be evaluated for each point in time within $[0,T]$, with the result being sent as a command to the robot. $s(t)$ is evaluated for a position controller, $v(t)$ for a velocity controller. In contrast to conventional point to point control where both velocity and acceleration equal 0 at start and end point, they do not have to be 0 in this case and so fluent movements can be created by several segments lined up one after the other.

A possible outcome for such an interpolation can be seen in Figure 3.14. In this case the recreation of a sinusoidal curve is attempted by use of polynomial interpolation. In classic signal theory the sampling frequency f_s must be $f_s > 2 \cdot f_{max}$ in order to recreate an signal correctly, whereby f_{max} is the highest occurring frequency in the signal. This is not the case here, because in addition to the sampled value of the original (reference) signal, also its first and second derivatives are available. Instead, as can be seen in Figure 3.14a, already one reference state per period is enough to approximately recreate the original signal. As the reference states are obtained by sampling the reference trajectory, they will be called reference sampling point (RSP) from now on. The second RSP visible at $t = T$ theoretically belongs already to the next period. Although the interpolated trajectory has the same frequency as the reference, there are still some deviations in the trajectories' amplitudes visible. By adding just adding two additional RSPs per period those deviations can be reduced to make them invisible at this zoom level. In the final project the reference trajectory is sampled several times per second, so that applied efforts can be converted to corresponding robot movements as soon as possible. This way minor deviations between reference and interpolated trajectory are no longer noticeable.

3.4.2 Delay handling

In the previous section it has been assumed that by the time t equals T, a new RSP is available and a new polynomial can be calculated. However, with the RSPs being transferred via ROS, a network with potential latencies and irregularities, this cannot be guaranteed. The above-mentioned latencies are discussed in more detail in section 5.3. For the time being it is important to know that they occur, that they are not negligible and that it must therefore be ensured that the robot controller does not stop if no new RSP is available and can resume the planned movement when a new RSP finally arrives.

Command buffer

Using ROS was an initial design choice to support a general interface and ensure interchangeability and reusability. As previously mentioned it was decided to not upgrade to ROS2, so the communication method is considered a fixed boundary condition and cannot be changed. Thus another way has to be found to cope with the occurring network delays. A first solution is found by introducing a buffer at the ROS controller, which stores incoming RSPs and the respective time the robot is given to reach them. Together they form a command, which is provided to the controller as soon as the previous trajectory segment is finished.

The disadvantage of a buffer is, however, that in order to be able to provide new RSPs, a reserve must be created. To achieve this, the first values that the controller receives are written to the buffer and held back for the time being. The movement of the robot is only started when the buffer contains a predefined number of entries. This results in additional latency between the MBS and the robot's movements.

The buffer should therefore be kept small, and to determine its minimally required length, the maximal occurring latency has to be found. To do so, a sampled curve is sent via ROS and the times of sending and receiving are recorded for 200 s. For this example, a sinusoidal curve is chosen, as it was already implemented for the circle trajectories in e.g. Figure 3.8. However, the shape of the trajectory itself is of no importance for

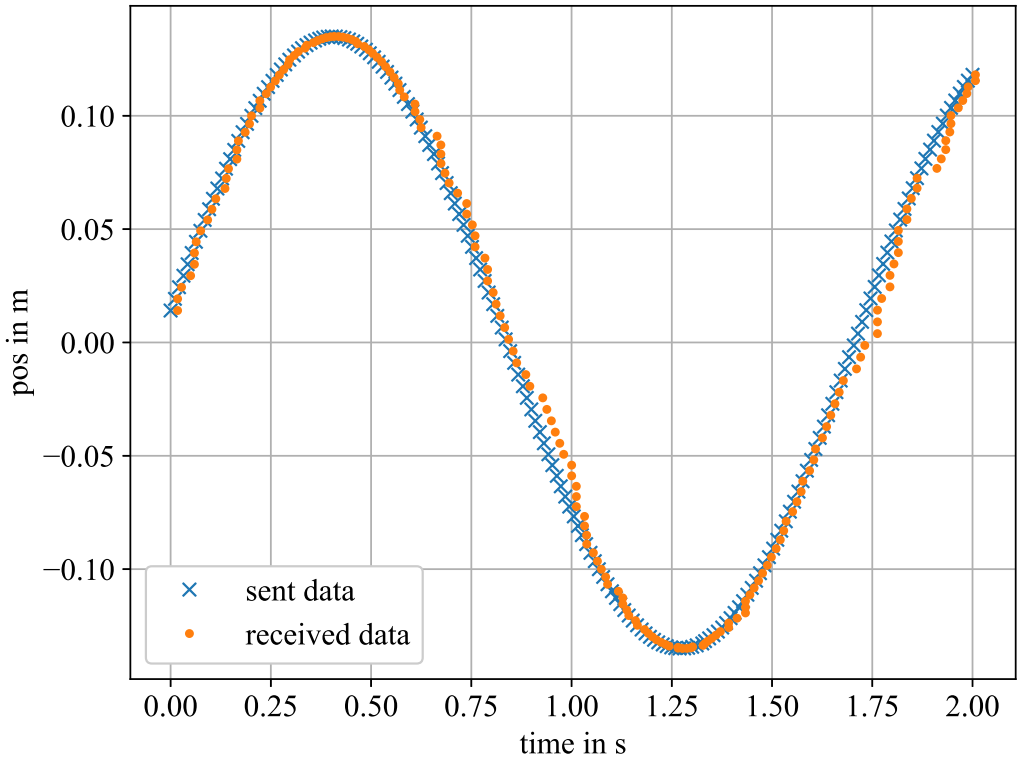


Figure 3.15: Timing irregularities when sending reference poses from the MBS to the ROS controller.

this test. With a sending frequency of 100 Hz the maximal occurring latency is 0.071 s and therefore the buffer would need a length of $\lceil \frac{0.071\,\mathrm{s}}{1/100\,\mathrm{Hz}} \rceil = 8$. Figure 3.15 shows a 2 s excerpt of the measurement. It is understood that while the data is sent on a regular basis, the delay at which it is received at the controller fluctuates.

Further delay handling

An initial buffer length of 8 entries and a sending frequency of 100 Hz equals to a programmed delay of 80 ms. This is a considerable delay, especially given the fact that while the maximum delay of above's measurement is 0.071 s, the mean value is only 0.01 s with a standard deviation of 0.008 s. Furthermore, this maximum delay was only the maximum delay within this specific measurement, so an even higher latency might occur at some point. So, whereas most of the time the size of this buffer's length might be too large, in some rare cases it might even not be sufficient.

Therefore, in addition to the buffer, whose size is to be decreased, another procedure is required, which can cope with RSPs that do not arrive in time. In other words: what to do if there is in fact no new RSP available when a previous trajectory segment ends. A position controller that passes a new (joint or Cartesian) position to the robot controller at each iteration, would, unless implemented otherwise, continue to send the last position of the last trajectory, resulting in an abrupt stop of the robot's movement, which in turn would most likely lead to a software error of the robot's controller as hardware acceleration and/or jerk limits would be exceeded. A velocity controller avoids this problem as,

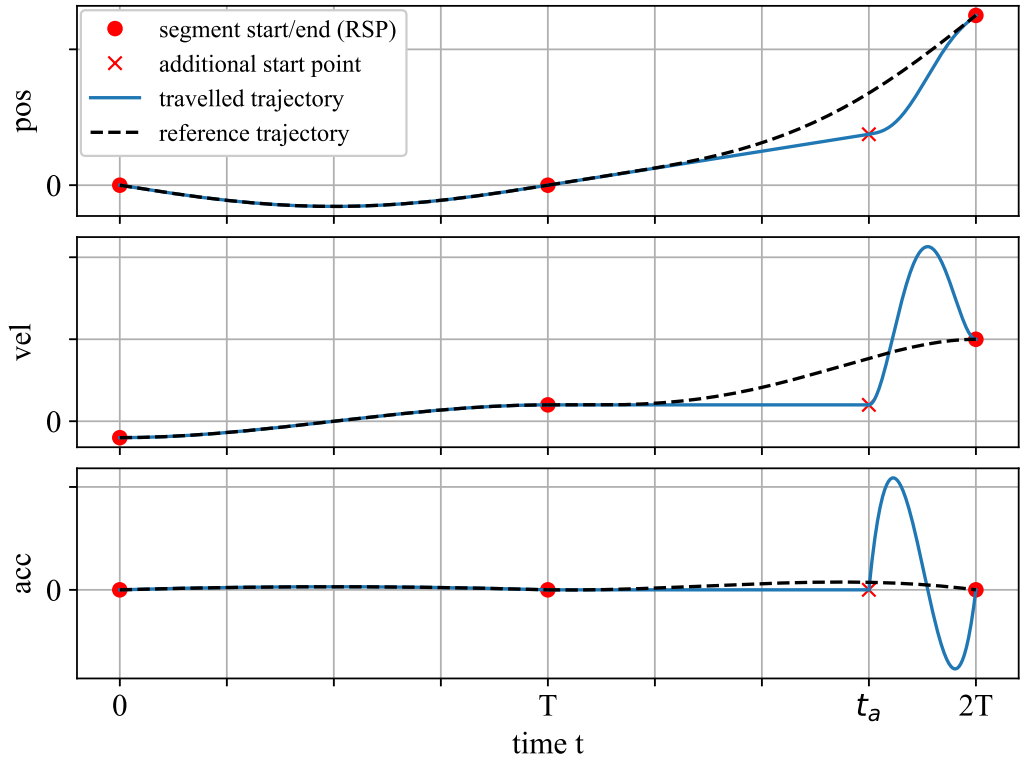


Figure 3.16: Delay handling principle 1: when the next RSP arrives at $t = t_a$, the trajectory is generated only for the remaining part of the segment duration T. Units and absolute numerical values are neglected.

instead of the last position, the last velocity is published and the robot does thus not have to change movement direction or speed. This characteristic is also mentioned by Zeleneak et al. [24], who designate velocity controllers as "highly robust to slow or randomly delayed control signals". Nevertheless, the question how to continue the movement when a new RSP is finally available again is yet not answered.

A first approach involves trying to get to the next RSP from the current position, velocity and acceleration, so from the trajectory state at the time t_a when the next RSP arrives. The current values (pos, vel, acc) can be obtained using the FCI. The time remaining within this trajectory segment to reach the reference can be calculated by subtracting the original segment duration from the time that has already elapsed since the last segment was completed. The sooner the new RSP is available, the more time remains to complete the segment in time and the less the trajectory needs to be changed, compared to the one which would be used if the entire segment duration were available. However, if the availability of the next RSP is given only shortly before the end of the planned segment, there is only little time left and the trajectory (and/or its derivatives) might have to take sharp turns to fulfill the end conditions. Such scenario is shown schematically in Figure 3.16. The end RSP of the second segment (from $t = T$ to $t = 2T$) is not available when the first one ends ($t = T$) and arrives only when about 75 % of the second segment's duration have already passed. Although the robot moves in roughly the right direction in the meantime, as soon as the RSP arrives, it has to accelerate and then slow down dras-

tically to reach the reference with the desired velocity. Especially the acceleration curve clearly exceeds the maximum values of the original (reference) trajectory, which might lead to problems due to hardware limitations of the robot. Those are listed in the FCI-documentation[23]. Whereas the limits in joint space are irrelevant since all commands in this thesis are given in Cartesian space, it is not defined in detail what the limits in Cartesian space refer to. They could be the limits for each axis individually or a norm, for example the Euclidean, of all three axes. Even if looking at it conservatively assuming that they represent a norm, the question remains to what extent they are applicable. They could only be the theoretical maximal limits in the center of the robot's workspace, far from any singularities. As this question could not be answered, it cannot be checked if the new trajectory with its derivatives is within any limits.

Although the trajectories generated by the RSPs using the entire segment duration cannot be controlled either, the probability that high velocities, accelerations or jerks could occur is manageable, as the used MBSs do not exhibit high frequency oscillations (see section 1.1). However, this does not apply to the trajectories that attempt to make up for lost time due to communication-related delays. This can also be observed in Figure 3.16, where the pseudo-frequency and amplitude of the final oscillation of the travelled trajectory's acceleration is clearly higher than the one of the reference.

For testing purposes this principle was implemented in the ROS controller, however, with little success. The robot controller often crashed due of exceeded hardware limits, i.e. the robot stopped. A more sophisticated principle would be not to make up the entire lost distance to the reference within the end of the segment, but only what is feasible when complying with the hardware limits. The remaining deviation could be made up step by step in the next segments. Without precise knowledge of the exact limits, however, this algorithm is only based on speculation and this approach is therefore not productive.

Instead, a simpler principle is tested. If no new RSP is available when a segment ends, the robot keeps moving at its current velocity. However, when a new RSP arrives, the new trajectory is calculated as if the RSP had already been available from the beginning, i.e. over the entire segment duration T. Subsequently, the velocity is read out at the current time and passed to the RCU as a new command. A possible outcome of this algorithm is depicted in Figure 3.17. Resuming the reference trajectory regardless of the robot's effective previous velocity or acceleration (which is 0) leads to a jump in velocity and thus to an impulse in acceleration. However, the RCU is apparently able to handle the two discontinuities without any problems most of the times. The movement of the robot arm is even smoother with this principle than with the previous approach, which manifests itself in quieter and more uniform motor noises. However, the drawback of this principle is the drift in position created this way, as the EE is not at the correct position when the reference trajectory is resumed. This can be seen in Figure 3.17, where at $t = 2T$ the EE is not at the reference position. So the question is how to proceed after such a drift has occurred.

One possibility consists in taking the current position of the EE (at $t = 2T$) as the new start value for the new trajectory. As the end RSP of a theoretical third segment remains unchanged, this would bring the EE back to its reference trajectory within the

[23] `https://frankaemika.github.io/docs/control_parameters.html#limits-for-panda`, accessed 21.03.2023

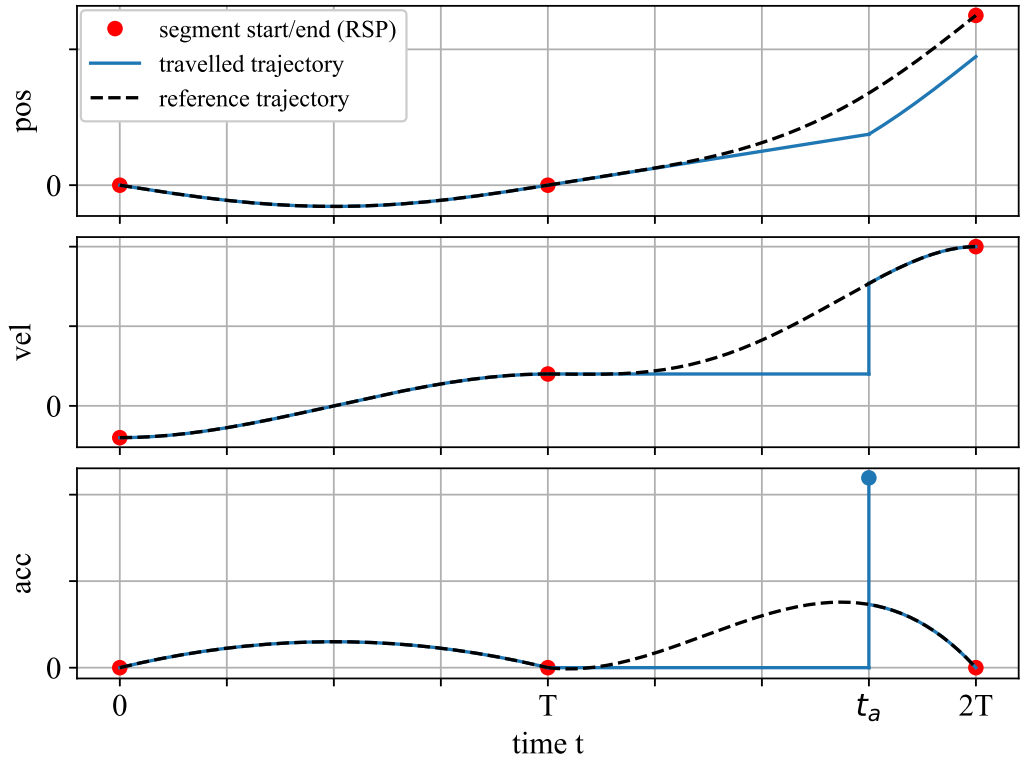


Figure 3.17: Delay handling principle 2: the trajectory is generated for the full segment duration T. Units and absolute numerical values are neglected.

next segment. However, this brings up a problem similar to the one encountered in principle 1: While there the time needed to cover a certain distance is too short, now the distance which must be traversed within one segment's duration might be too big, thus potentially forcing the robot out of its hardware limits[24].

A second approach is to ignore the drift and start the next segment's trajectory where the previous reference trajectory has ended. Doing so, the drift is not compensated, but due to rather short segment duration, the drifts are only small and as they might occur while the robot is moving in an arbitrary direction, they might compensate each other.

With those principles the buffer length for now is decreased from 8 to 4 entries, which corresponds to a programmed delay of 40 ms.

3.4.3 Implementation

Having acquired the theoretical knowledge about algorithms and principles, the next step is to implement them in C++. The FCI documentation gives detailed information on how to write a controller using franka_ros from scratch[25]. However, Franka Emika already provides a velocity controller within its `franka_ros/franka_example_controllers-`

[24]This is again subject to the assumption that the MBS itself does not command any movement that would require exceeding the robot's hardware limits.

[25]`https://frankaemika.github.io/docs/franka_ros.html#writing-your-own-controller`, last accessed 20.03.2023

package[26], named CartesianVelocityExampleController. This example has hardcoded an EE path and is therefore not suitable for this thesis's purposes. Thus it is taken as a basis and considerably adapted to fulfill the requirements. The entire controller code can now be found on GitHub[27]. In the following, selected sections of interest from the code are described.

The base class, which the controller is derived from, has four virtual methods, which need to be overwritten by the derived class. Those four methods are

- `bool init(hardware_interface::RobotHW*, ros::NodeHandle &);`
- `void starting(const ros::Time &);`
- `void update(const ros::Time &, const ros::Duration &);`
- `void stopping(const ros::Time &);`

While overriding the `init()` and `update()` methods is crucial for the controller to work hence compulsory, the other two are optional.

The `init()` and the `starting()` methods both initialize variables. Whereas the `init()` serves as a sort of constructor, which can be called anytime between declaration of the controller-object and starting of the movement, the `starting()` method is called directly before starting the movement. Thus the error-checking, which has been adopted from the example controller, adjusting the controller's name and the console output, is done in the `init()`, because the information required for it is already available. The same applies to opening and initializing (writing header line of CSV files) the log files. Whether the controller writes to log files or not can be toggled via the macro `ENABLE_LOGGING` in the controller's header file. Regardless of that, however, the log files are always opened and initialized, thus deleting any data from previous runs and preventing them from being misinterpreted as valid for any other run.

To enhance efficiency a ring buffer is used to store the RSPs. Entries do not have to be shifted each time a reference state is removed from the front of the first-in-first-out queue. This buffer is also initialized here, i.e. the reading and writing indices are set to 0 and 1 respectively. Thus the only tasks left to do in the `starting()` method are to set the timers to zero and to retrieve the current EE pose.

The `update()` method is called once per control iteration, thus being the one where the above-mentioned algorithms etc. are implemented in. Its only parameter is the the time provided by ROS, which allows for the method to determine whether a new trajectory has to be calculated or only the existing one should be evaluated. It also roughly checks whether new velocity commands are permissible from a hardware point of view.

The `stopping()` method is called when the robot controller stops. It therefore serves as a sort of destructor and in this particular code it is only responsible for closing log files properly. If the robot is still in motion, the joint movements are stopped automatically but abruptly by the RCU. This method could also perform a smooth and controlled stopping process, however, this has not yet been implemented.

Apart from these four overwriting methods there are also some custom methods. `updateReferencePoseCallback(const util::segmentCommand &)` is, as the name

[26] `https://github.com/frankaemika/franka_ros/tree/develop/franka_example_controllers`, last accessed 21.03.2023

[27] `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_example_controllers/include/franka_example_controllers/exucobot_cartesian_velocity_controller.h` and corresponding source file, last accessed 25.04.2023

suggests, the ROS callback method for receiving new RSPs from the MBS simulation built in Exudyn. The message is passed as parameter to the method, where its content is transformed into the corresponding data structure (struct `SamplingPoint3`, which represents a sampling point and thus contains one `double` value each for position, velocity, acceleration, and jerk for each axis x, y, and z respectively[28]) and put at the end of the circular buffer together with the passed requested segment duration.

The method `updateTrajectory()` is called by the `update()` method when a new trajectory is needed. It retrieves the required information from the RSP buffer, calculates a new trajectory (= polynomial) and stores the new parameters b_i in the corresponding class attribute.

`evaluatePolynomial(std::vector<double> &coef, double t)` is finally called to evaluate the trajectory, specified by the coefficients `coef` at a given time `t`. As the purpose of this function is purely mathematical and could also be used with other polynomials not representing a trajectory, it is not implemented as a class method, but as an independent function.

As the two possible trajectories from previous sections differ only in their starting position, both approaches are implemented in the code and switching between them is enabled by changing the value of

```
bool allowDrift_ = true;
```

in the corresponding header file[29], which, as stated, is set to `true` by default. Thus the second approach is used by default.

3.4.4 Results

The measurement of Figure 3.10, which shows the user applying force to the robot EE simulating the pendulum-MBS described in subsection 2.4.2, is repeated with the velocity controller in order to evaluate the developed principles and algorithms.

Figure 3.18 shows the result of the measurement. As the force was applied manually, the exact force values differ between the experiments (see Figure 3.10). Since the force is transmitted via ROS to the MBS, it would theoretically be possible to record these messages and replay them to have the same input again. This could be realized by creating a so-called bag-file, which essentially stores all messages published to a specific topic while recording, and can be replayed multiple times later on, i.e. all those recorded messages are published again in the same order and with the same time offset to each other. The procedure described above has also put into practice but with moderate success, as the robot joints have some tolerances which make them move freely. When force is applied by the user, registered by the robot and passed to the MBS, the robot is told in return to move accordingly. Since the whole process runs in near real-time and the user usually does not only apply a short force impulse, the robot still experiences an external force effect from the user when it starts to move according to its trajectory. An ideal robot

[28] implementation available at `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_example_controllers/include/franka_example_controllers/samplingPoint.h`, last accessed 18.04.2023

[29] `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_example_controllers/include/franka_example_controllers/exucobot_cartesian_velocity_controller.h`, last accesed 15.04.2023

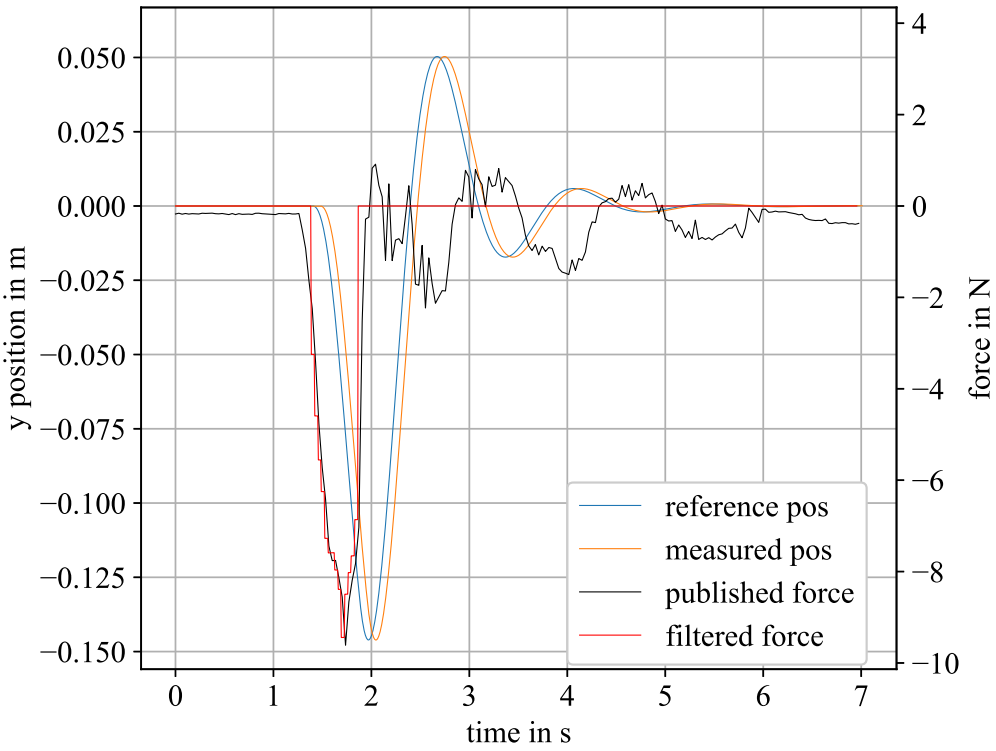


Figure 3.18: 2D-pendulum with velocity controller.

with no tolerance in the joints would not be affected by this applied force. However, collaborative robots as the one used in this project, which already provide a built-in compliance by default, behave differently. Accordingly, the applied force is missing when the bag-files are played later, so the result is not the same as when the file was recorded.

Although the applied forces are similar but not exactly the same in both experiments, there is a clear difference between them, which is not due to force application: The measured trajectory is delayed compared to the reference trajectory by a time offset of 76 ms, which is significantly shorter than it was with the impedance controller (270 ms). The delay of 76 ms is composed by the programmed delay created by the buffer ($4 \cdot 10\,\text{ms} = 40\,\text{ms}$), the sending time over the ROS network, and the time needed for computation. The delay created by the ROS sending time, however, is not defined by the average sending time of all messages, but by the time it takes to fill the buffer to its minimum size. So, if the, in this case four, messages take above average time to arrive at the ROS controller, the whole simulation is additionally delayed by this amount of time.

An additional advantage of the velocity controller is the fact that the robot follows a well-defined trajectory. In contrast to the impedance controller it does not try to reach the current reference position following a straight path. It therefore enables the EE of the robot to reach the aspired extreme values of the trajectory.

The last issue discussed in subsection 3.4.2 concerned the new starting point for the next trajectory. The options debated referred to either the actual EE pose or the theoretical endpoint of the previous trajectory. The trajectory visible in Figure 3.18 has been created with the second method, i.e. the algorithm does not try to compensate the drift in

any active way. While there is no perceptible drift within the 7 s of recording, in further experiments simulations lasting multiple minutes were performed without giving evidence of any significant and perceptible drift. In contrast, the method using the effective robot EE pose in order to compensate any drift can be run for several minutes. However, this method seems less robust, as it leads to software errors more often.

4 Visualization

As described in section 2.5, depending on the definition, different systems can be referred to as VR. This chapter presents required hardware for visual VR systems, introduces necessary software, and gives a short general overview of its implementation. Further on it elaborates more crucial content in more detail and finally describes how objects are located to properly render them in the VR view. It is therefore limited to visual VR systems, as they have been described by Milgram and Kishino [15].

4.1 Hardware

Over the past decade, several different HMD have entered the consumer market. One of them is the VIVE Pro Eye[1] of the HTC manufacturer, which is used for the present project. Like other comparable products from other manufacturers, it is mainly used as consumer electronics.

Figure 4.1[2] shows the devices used in this thesis. The centrepiece of such VR technology certainly is the head mounted display (HMD). In fact, this consists of actually not just one, but two displays, one for each eye. To give the user the feeling of 3D viewing - which is the main purpose of such HMDs - the images rendered on the two screens differ slightly from each other: the viewpoints of the images on the two displays are next to each other, at a distance that corresponds to distance between the eyes of the user. This

[1] `https://www.vive.com/us/product/vive-pro-eye/overview/`, last accessed on 09.03.2023
[2] single images taken from `https://www.vive.com/us/`, last accessed 07.04.2023

Figure 4.1: Used VR devices from HTC: a VIVE Pro Eye on the left, a (hand held) controller on the bottom right, a tracker on top center and a base station on the top right.

must be taken into account when setting up the rendering process. This process does not run on the HMD itself but on a computer that must be powerful enough for this very purpose. Above all, a suitable graphics card is required for this purpose[3].

However, the HMD and a computationally powerful computer are not enough to make the VR system work. To locate the HMD's (and thus also the user's) position within the VR coordinate system $\mathscr{K}_{\text{VR}}$, two so-called SteamVR Base Stations are used. They are each mounted on a tripod at two opposite corners of the workspace, facing inwards, to track not only the HMD, but also additional hand-held controllers (HHCs) or trackers. HHCs have digital and analog buttons and allow for the user to interact with the virtual environment. Compared to them, trackers are smaller and have no input options. They have a thread on their back instead, which allows for easy mounting in various places, for example on the user's body with straps provided for this purpose. A variety of other products are commercially available and could be potentially used in this or similar applications, but since they are not relevant for the present thesis they are not listed here. A guide on how to setup a VR system can be found in appendix A.2.

4.2 Rendering

To render images onto a conventional computer screen, different programming libraries and APIs can be used. Exudyn uses GLFW[4] and OpenGL[5] for rendering its MBSs. These two libraries enable the creation of a simple desktop window and efficient rendering of a variety of different objects. However, for rendering images onto a HMD, the above-mentioned libraries are not sufficient. As an HMD is an external device, hardware drivers are required as well as an additional API to access hardware-specific functionalities from within the programming code. Those drivers are provided by a software called SteamVR[6] by Valve. SteamVR is a runtime environment that runs on a computer and establishes and manages the connection to the VR system (e.g. HTC Vive, but not only). On the other hand, OpenVR[7], also developed by Valve, is a Software Development Kit (SDK) and C++ API, which grands access to VR systems from code, without any further knowledge about the actual VR system itself.

Together they form a rendering pipeline, which can be seen in Figure 4.2. While the pipeline for the PC screen is simple, the one for the HMD is more complex, as the data has to be passed through multiple modules. However, it is not enough for an HMD to receive data to be rendered on the two screens. To work properly, its pose needs to be returned to the software application for the next frame to be rendered accordingly. The connection between the HMD and the program, therefore, is not a one-way but a two-way connection. Furthermore, the poses of the tracking devices need to be passed to the program as well. All poses are given in $\mathscr{K}_{\text{VR}}$-coordinates. The complete resulting VR system is depicted in Figure 4.3.

[3] All system specifications and hardware requirements can be found at `https://www.vive.com/us/product/vive-pro-eye/specs/`, last accessed 09.03.2023

[4] `https://www.glfw.org/`, last accessed 09.03.2023

[5] `https://www.opengl.org/`, last accessed 09.03.2023

[6] `https://www.steamvr.com/en/`, last accessed 03.04.2023

[7] source code available at `https://github.com/ValveSoftware/openvr`, last accessed 09.03.2023

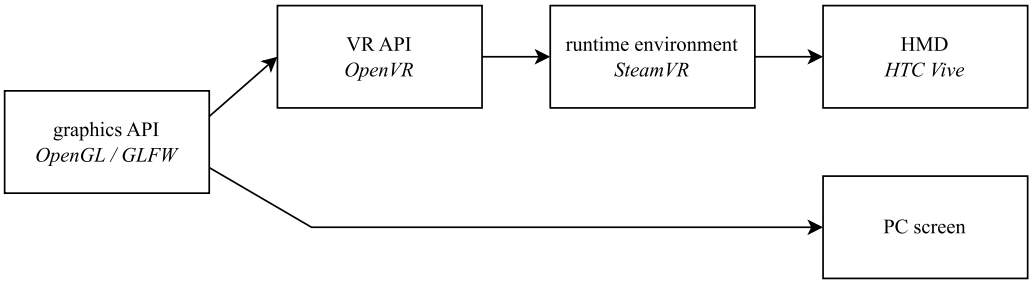


Figure 4.2: VR render pipeline. The respective software/hardware used for single modules are indicated in italics.

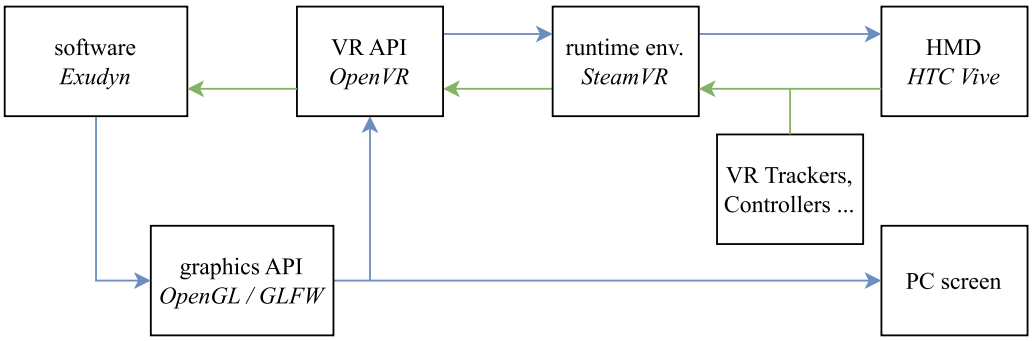


Figure 4.3: Overview of the VR system. Blue arrows represent image data being sent to the screens, green arrows Cartesian poses being sent back to the software application.

OpenVR implementation

Exudyn already includes rendering functionalities, which allow for the MBS to be rendered onto a computer screen using OpenGL and GLFW. The corresponding code can therefore be used which makes further discussion at this point obsolete. It is, however, of major concern to this thesis, how to include the OpenVR functionalities in Exudyn.

In contrast to conventional rendering onto a computer screen, where only one point of view is rendered to a window, working with an HMD implies that two points of view need to be rendered to two different displays respectively: one for each eye. The C++ library OpenVR provides the programmer with tools to get the poses of the two viewpoints from the HMD and to return the image buffers created with OpenGL to the HMD. Furthermore, it allows for interaction with additional VR tracking devices.

The MBS needs to be rendered from Exudyn onto a computer screen as well as onto an HMD. For this very purpose, OpenVR functionalities need to be included in the Exudyn C++-source code. OpenVR provides several sample programs on its GitHub repository, including one for using the SDK with OpenGL[8]. This example code uses multiple third party libraries for OpenGL which are not used in Exudyn. In order to not unncessarily extend the Exudyn dependency list unnecessarily, this sample code is rewritten using only GLFW, which is already in use in Exudyn, as additional dependency. Furthermore, unnecessary code pieces, e.g. scene creation, which in this thesis is handled by other Exudyn modules, and graphic shaders compilation, Exudyn does not

[8] `https://github.com/ValveSoftware/openvr/blob/master/samples/hellovr_opengl/hellovr_opengl_main.cpp`, last accessed 03.04.2023

use explicit shaders at all, are removed. As a result, the remaining features are basic rendering routines and the interaction interfaces with different VR tracking devices. The resulting code is implemented directly in Exudyn[9].

Once these dependencies are replaced, the projections rendered onto the HMD are no longer identical to the ones before the exchange nor do they behave according to the rotation and translation of the HMD. To analyze and solve this problem, the equations which transform 3D room coordinates to 2D screen coordinates need to be looked at in more detail.

4.2.1 Coordinate transformation

Rendering objects onto a screen entails the transformation of local object coordinates to screen coordinates. This process can be divided into multiple steps and includes scaling, rotation, and translation of the objects themselves, rotation and translation of the camera (viewport) and applying the desired projective transformation to the scene, in order for the image on the screen to look natural. All these transformations can be represented as matrices compatible between them, so the entire process can be performed within one single transformation

$$\begin{pmatrix} {}^{c}x \\ {}^{c}y \\ {}^{c}w \end{pmatrix} = {}^{cv}\mathbf{P} \cdot {}^{vs}\mathbf{V} \cdot {}^{sl}\mathbf{M} \cdot \begin{pmatrix} {}^{l}x \\ {}^{l}y \\ {}^{l}z \\ {}^{l}w \end{pmatrix}. \tag{4.1}$$

The symbols in this equation are defined as follows:

- The local coordinates ${}^{l}\mathbf{x}$ describe the object geometry within a local (object bound) coordinate system.
- The clip coordinates ${}^{c}\mathbf{x}$ represent the normalized coordinates that are used to determine whether and, if so, which part of an object is within the camera's field of view[10]. They will be transformed to screen coordinates by OpenGL autonomously before rendering.
- The model matrix ${}^{sl}\mathbf{M}$ transforms local (object bound) coordinates to scene coordinates, with the latter describing the object's position and orientation within the scene.
- The view matrix ${}^{vs}\mathbf{V}$ transforms scene coordinates to the viewport coordinate system, i.e. how they are viewed from the camera.
- The projection matrix ${}^{cv}\mathbf{P}$ transforms viewport coordinates to clip coordinates.

In case of virtual reality, where the user wears a HMD and the scene must therefore be rendered to two displays simultaneously (one for each eye), the view matrix is divided into a matrix ${}^{hs}\mathbf{H}$, which transforms the scene coordinates relatively to a coordinate system h in the middle of the HMD, and a second matrix ${}^{eh}\mathbf{E}$, which translates the coordinates by half the offset between the user's eye positions and system h. This matrix is separately defined for each eye, but as the structure of the equation is the same for both

[9] `https://github.com/jgerstmayr/EXUDYN/blob/master/main/src/Graphics/OpenVRInterface.cpp` and corresponding header file, last accessed 03.04.2023

[10] `https://learnopengl.com/Getting-started/Coordinate-Systems`, last accessed on 03.04.2023

of them, no distinction between them is made here. Furthermore, since the HMD, e.g. the HTC VIVE, can only be adjusted to the position of the user's eyes by moving the two displays horizontally and, what is more, this can only be done for both eyes equally, i.e. mirrored, the two matrices differ only in the sign of one entry.

As the first transformation, i.e. from local to scene coordinates ${}^{s}\mathbf{x}$, is calculated by Exudyn in the background, this transformation is omitted in the following in order to simplify the equation. Thus, the adapted equation is

$$\begin{pmatrix} {}^{c}x \\ {}^{c}y \\ {}^{c}w \end{pmatrix} = {}^{ce}\mathbf{P} \cdot {}^{eh}\mathbf{E} \cdot {}^{hs}\mathbf{H} \cdot \begin{pmatrix} {}^{s}x \\ {}^{s}y \\ {}^{s}z \\ {}^{s}w \end{pmatrix}. \tag{4.2}$$

Theoretically this would be the equation necessary to project an arbitrary 3-dimensional scene onto an HMD and it is also the one used to combine the projection matrix in the official OpenVR-OpenGL sample code[11]. However, as the rendering process includes multiple C++-libraries (OpenVR for VR-handling, Exudyn for matrix operations and OpenGL for rendering), some adaptations are required to make the system work. When looking at the library implementations, it can be noticed that all of them use matrices, but not all of them handle them the same way. There is a clear difference in how matrices are stored internally. Some of them store them in column major notation, so a matrix is read column by column, whereas others store them as row major notation, with a matrix

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \tag{4.3}$$

being read row by row. Equation 4.3 shows an arbitrary matrix whose elements have two indices each. The first index represents the row of the element, the second the column. The order of the indices, with the row preceding the column, already suggests a prioritization of a dimension, in this case of the row. However, depending on the convention used, this order could also be changed.

The same applies to programming libraries where matrices, even if they are two-dimensional constructs, are often stored as one-dimensional arrays. Thus the entries of the matrices need to be put into a one-dimensional order, which is done by row or column dominance respectively. For Equation 4.3 the storing order would be

$$\underbrace{m_{11}, m_{12}, m_{13}, m_{14}}_{\text{first row}}, m_{21}, \ldots \tag{4.4}$$

for row major and

$$\underbrace{m_{11}, m_{21}, m_{31}, m_{41}}_{\text{first column}}, m_{12}, \ldots \tag{4.5}$$

for column major matrices. If a stored column major matrix is read as row major, the result is the transposed matrix of the stored one and vice versa.

[11] see footnote 8

For this reason, working with several different programming libraries at the same time implies considering the conventions used in each of them. Thus the difference between the sample program and the implementation in Exudyn leading to incorrect rendering can be explained by the fact that the matrix implementation of the first uses column-major notation[12] whereas Exudyn uses row-major notation[13].

Therefore, in the OpenVR sample program[14] the matrices coming from OpenVR as row-major are immediately transposed. Then they are combined according to Equation 4.2 and passed to OpenGL. In the Exudyn implementation, the matrices can be accepted and multiplied as they are, but the result must be transposed in order to meet OpenGL's requirements. So the difference is the point in time when the transposition is performed: either before or after the multiplication. As this process happens twice in every frame (once per each eye), it is recommendable to improve computational efficiency as much as possible. The matrices cannot be passed from OpenVR to Exudyn with a simple assignment operator because the two libraries use different data types for them. The new matrices in Exudyn must therefore be constructed from scratch and the individual values are transferred one by one. With that in mind, all three matrices can be composed already as their transposed ones and in order for the result to be the same, the multiplication order must be inverted.

$$\mathbf{x}_c = \left({}^{ce}\mathbf{P} \cdot {}^{eh}\mathbf{E} \cdot {}^{hs}\mathbf{H}\right)^T \cdot \mathbf{x}_s \tag{4.6}$$

$$= {}^{hs}\mathbf{H}^T \cdot {}^{eh}\mathbf{E}^T \cdot {}^{ce}\mathbf{P}^T \cdot \mathbf{x}_s \tag{4.7}$$

By doing so, two matrix transpositions per visualization frame can be saved, as the transposition of a single matrix during construction does not add any additional computational overhead.

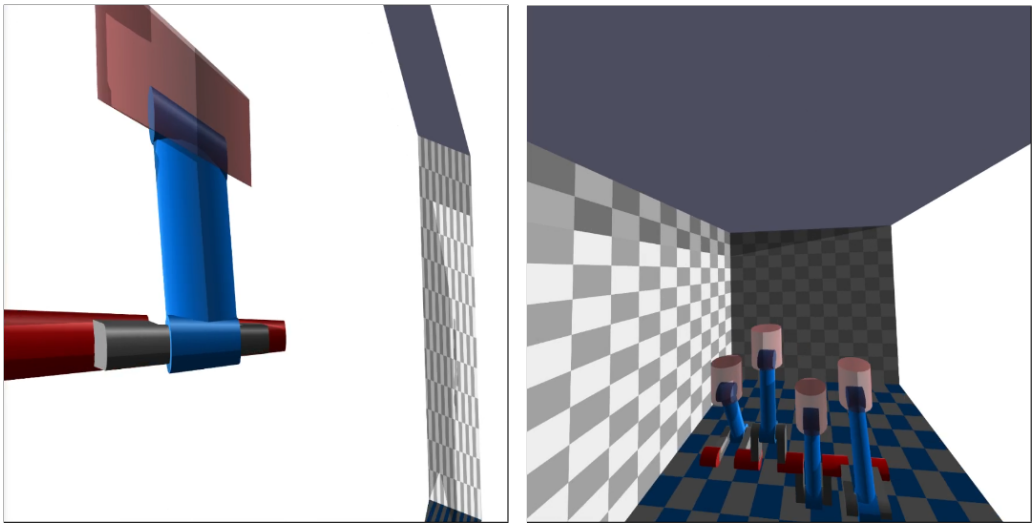
Figure 4.4 shows the effect different multiplication orders or matrix notations have on the image displayed on the HMD (or any other screen). In contrast to every other test with Exudyn performed in the present thesis, for this one the standard MBS of the pendulum has not been used, since it is too simple to allow concrete conclusions about the perspective. Instead a crankshaft with four cylinders in a cuboid box, which is open on one side, is used[15]. Although the depicted scene and also the camera's point of view are identical for both images, the renderings are completely different: While the right image, whose projection has been calculated with Equation 4.7, shows the scene as one would expect it to look like, the left image shows the same scene but with the projection resulting from Equation 4.2 (a row-major matrix is interpreted as column-major) and results therefore in a totally disfigured image.

[12] `https://github.com/ValveSoftware/openvr/blob/master/samples/shared/Matrices.h`, last accessed 27.04.2023

[13] `https://github.com/jgerstmayr/EXUDYN/blob/master/docs/theDoc/theDoc.pdf`, last accessed 27.04.2023

[14] `https://github.com/ValveSoftware/openvr/blob/master/samples/hellovr_opengl/hellovr_opengl_main.cpp`, last accessed 27.04.2023

[15] source code for MBS taken from `https://github.com/jgerstmayr/EXUDYN/blob/master/main/pythonDev/Examples/openVRengine.py`, last accessed 20.04.2023



(a) $\mathbf{P} \cdot \mathbf{E} \cdot \mathbf{H}$ (b) $\mathbf{H} \cdot \mathbf{E} \cdot \mathbf{P}$

Figure 4.4: Comparison between output of different matrix multiplication orders. Both show the same scene, viewed from the same position and view-angle. In (a) the multiplication order $\mathbf{P} \cdot \mathbf{E} \cdot \mathbf{H}$ is used. In (b) the different column and row-notations from OpenGL and Exudyn are considered correctly, leading to the desired Image.

4.3 Object localization

In order to be able to render different objects at the correct position and with the correct orientation in VR, their poses need to be tracked. For this project, this applies to two objects: the manipulator, for the MBS to be rendered in relation to it, and the user's hand, for the user to be able to better estimate distances in VR, as the HTC VIVE is completely opaque. For both objects a separate tracking device is necessary.

4.3.1 Hand tracking

When purchasing a HTC VIVE, two HHCs[16] are included in the package per default. The HHCs are tracked by the VR system and can be used to interact with the VR environment in various ways. However, they are meant to be held in hand. This turns out to be inconvenient for this project since the user needs to use their hand to interact with the robot. Holding a gadget for mere tracking purposes of the hand is counterproductive. As an alternative a VIVE Tracker[17] can be used, another tracking device for the HTC VR system. In contrary to an HHC it provides any possibility for user input, which is, however, irrelevant for the present project. VIVE trackers are meant to be fixed at whatever object is needed to be tracked. To this end they are equipped with a thread on their backside. To be mounted somewhere on the user's body, special straps[18] designed for this purpose may be used.

[16] `https://www.vive.com/us/accessory/controller/`, last accessed 31.03.2023

[17] `https://www.vive.com/us/accessory/tracker3/`, last accessed 31.03.2023

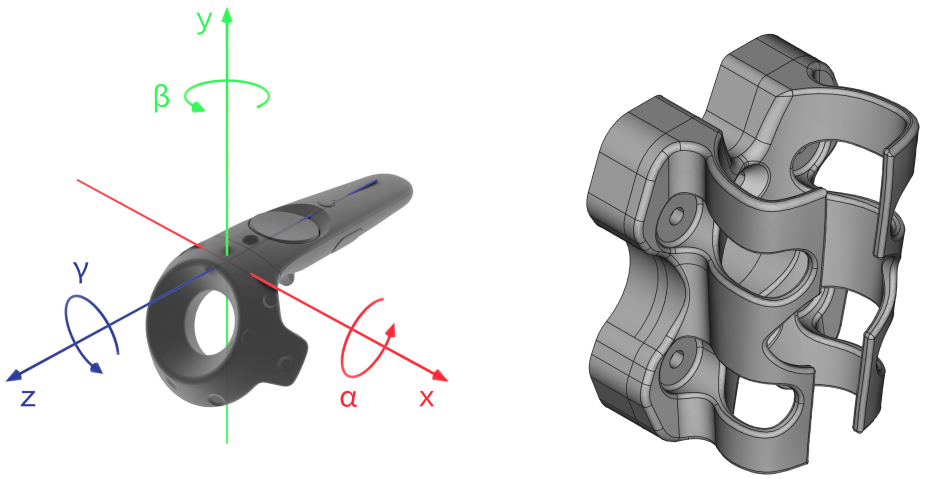
[18] e.g. `https://www.vive.com/us/accessory/rebuff-strap/`, last accessed 31.03.2023, but there are also other 3rd party suppliers

Figure 4.5: VIVE Tracker mounted on the back of a user's hand. The axis of the middle finger should line up with the markings on top of the tracker.

Figure 4.5 shows how a tracker can be mounted on the user's hand with the above-mentioned straps. The strap used during the work on this thesis has a hole on the one end and a Velcro fastener on the other end. So the user puts their thumb in the hole, makes the belt go along the back of the hand to the wrist, wraps it around the wrist and finally fastens the strap with the Velcro. In order to track not only the hand's position, but also its orientation correctly, it is decided that the tracker should always be rotated as shown in Figure 4.5, with an imaginary line through the markings (status LED, power button/triangle, deepenings) on the tracker in line with the long axis of the middle finger. With the tracker on the back of the hand, the user is still able to use the hand freely.

In spite of providing a good mounting method of the tracker, this arrangement is quite imprecise in terms of hand location The first issue is the strap itself. The part of the strap, where the tracker is mounted with the thread, is made of soft tissue. The distance to the back of the hand therefore depends on how tightly the tracker is screwed on. The second problem consists in the fact that the distance between thumb-hole and mounting screw cannot be changed. Because of that, the tracker's position on the user's hand may therefore differ depending on the hand anatomies of the various users. Furthermore, the hand displayed in the VR view[19] is of the same size for every user. It does therefore not necessarily correspond with the current user's actual hand size. Finally, the position of the tracker also changes slightly due to thumb or wrist movement. Thus the described tracking device and mounting method are inappropriate to provide for accurate representation of the hand in VR. For more accuracy another mounting method or another tracking device (e.g. a glove) would be required. However, the described method still gives the user a rough idea of the distance to other displayed objects and is therefore considered sufficient.

[19] used model: `https://grabcad.com/library/articulated-dummy-1`, last accessed 25.02.2023



(a) Coordinate system $\mathcal{K}_{\text{HHC}}$ of a HHC. (b) CAD-model of mounting station for a HHC.

Figure 4.6: HHC mounting station.

4.3.2 Robot tracking

The second object to be tracked is the manipulator, or more precisely its EE, which represents the physical UIP. The most intuitive approach to realize this would be to mount another VIVE Tracker on the EE itself. This, however, comes with two disadvantages: First, an additional weight on the EE would have to be registered in Franka Desk to enable the robot controller to compensate it and send the correct force to the MBS simulation. As this would have to be done only once, it should not be of great concern. The main disadvantage, however, is the additional tracker itself. As there is no tracker included in the default packaging of the HTC VIVE, it would need to be purchased separately. The two HHCs received alongside the HMD are still available and it is decided to make use of one of them for this purpose. An HHC, however, is much more unwieldy than a tracker and does not even dispose of a mechanism to be mounted somewhere. Not only would mounting it on the EE be complicated, but also the collision box of the Franka Emika Panda robot would have to be increased. A more convenient way to locate the EE is to mount the HHC at the manipulator's base. From there the relative EE-pose and thus its coordinate frame $\mathcal{K}_{\text{EE}}$ can be determined via forward kinematics by use of the (known) robot joint angles. As in the case of this thesis the robot itself is mounted on top of a table whose dimensions can be determined easily, the HHC does not have to be mounted directly at the robot's base, but can be mounted somewhere at the table.

For this purpose a mounting station is designed and 3D printed. It can can be mounted at the side of the table at the standardized aluminum 80x80 mm profile 8[20] and an HHC can be inserted in it. To simplify subsequent coordinate transformations, it is beneficial to align the coordinate axes of the HHC, shown in Figure 4.6a[21], once positioned in the mounting station, with the table edges, i.e. the x and y-axes are parallel to the table edges and the z-axis points vertically upwards accordingly. Figure 4.6b shows the CAD-model

[20] e.g. `https://www.item24.com/en-de/profile-8-80x80-natural-2627/`, last accessed 09.05.2023

[21] image taken from `https://yiyuan.space/digital3dscanner`, last accessed 08.04.2023

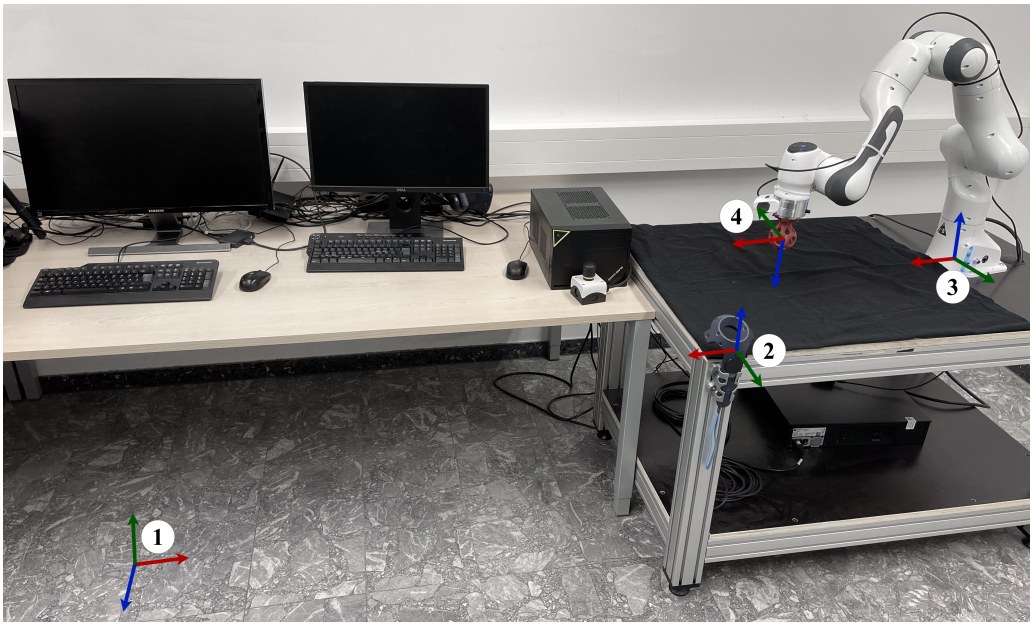


Figure 4.7: Overview of different coordinate systems needed for localizing the robot EE for rendering. VR base coordinate system $\mathscr{K}_{\text{VR}}$ (1), HHC coordinate system $\mathscr{K}_{\text{HHC}}$ (2), robot base coordinate system $\mathscr{K}_{\text{RB}}$ (3), robot EE coordinate system $\mathscr{K}_{\text{EE}}$ (4).

of the described mounting station. Figure 4.7 finally shows all the coordinate systems needed to localize the robot EE in the VR view.

For this purpose, the program first needs to know the position of the HHC, which can be obtained via OpenVR. The offset between the HHC position and the robot base coordinate system can be estimated based on the position of the robot on the table and, if the HHC is mounted as pictured in Figure 4.7, on the offset between table corner and the origin of the HHC coordinate system $\mathscr{K}_{\text{HHC}}$. As previously described, the user's hand cannot be located very accurately. This accuracy is estimated to be approximately ± 1 cm in each direction and it is assumed that this error is crucial for the total error the robot can be located in VR with. Therefore the position of the robot or, more specifically, the distance between the table corner and the origin of $\mathscr{K}_{\text{HHC}}$ does not need to be extremely accurate. With the robot base coordinates in the VR frame $\mathscr{K}_{\text{VR}}$ being known, the EE position in the same frame can be determined by combining the robot base coordinates and the EE coordinates within the robot base coordinate system, which are continuously published onto the ROS topic `/exucobot_cartesian_velocity_controller/currentPose`.

The offset between the HHC and the robot base is measured separately by hand and therefore the user is responsible for obtaining it. However, as mentioned before, the position of the HHC can be retrieved by OpenVR. For this purpose, a program[22] has been written which records the positions of all active VR devices for a few seconds. It then outputs the average position for every device respectively, which can be updated in Python afterwards, see subsection 6.1.3.

Finally, both the translations from the VR base coordinate system to the HHC and

[22] source code available at `https://github.com/bacaar/exuCobot/tree/main/util/locateVrDevices`, last accessed 14.04.2023

from the HHC to the robot base must be entered manually in RobotVRInterface.py, a procedure covered in more detail in section 6.1. All offsets must be specified in VR base coordinates. It is therefore advisable to align the VR system with the robot base coordinate system or with an angular offset of $k \cdot \frac{\pi}{2}, k \in \{0,1,2,3\}$. This facilitates the determination process as no trigonometric functions have to be used.

5 Real-time assurance

In order to minimize complexity and to provide the user with all the content at once, most measures for guaranteeing the real-time (RT) capability of the system have been disregarded in the previous chapters and are now covered in the following. Note that the term *real-time* does not apply to this system without restrictions, since some latencies will remain in the communication between the individual components. Strictly speaking, one should therefore speak of *near*-real-time, but to optimize readability, the term *real-time* is used.

In this work, two principles of *real-time* are distinguished. On the one hand, there is the RT of the simulation. Since the simulation of the MBS is just a simulation, in order to save time it could theoretically run faster than the MBS would behave in the real world, or more slowly in order to make details more visible. This would, however, undermine the principle of the present project. In order for the principle to work, the processing of e.g. 1 s of the simulation must also take exactly 1 s in reality, which takes us to the second principle: the computational processes must be able to run in RT. This would not be absolutely necessary if it was only for the simulation because the user would not notice a delay of just a few milliseconds, given a hardware fast enough to be able to cope with the complexity of the MBS. The robot controller, however, expects a new command at intervals of exactly 1 ms and does not tolerate any delays.

5.1 Real-time kernel

Popular general purpose operating systems (GPOS) such as Windows, MacOS or most Linux-distributions are not RT-capable and denominated time-sharing. That means that the OS process manager / scheduler can allocate computing time to individual processes. Although the user can roughly set the importance of processes, if the process manager considers another process to be more important, it is placed in front and others are delayed. This is not the case with RT-capable systems. Generally, they can be characterized by three major components/properties: time, reliability, and the operating environment [35]. First of all, precise timing is what gives a real-time operating system (RTOS) its name. In RTOS time-crucial processes do not only have a logical task, as they do in time-sharing OS, but they also have a deadline by which they must be completed. As there is usually a reason why a RTOS is used, these constraints must be strictly respected, i.e. the system must be reliable. Furthermore, also the operating environment of a computer is essential for characterizing a specific RT-system. For example, for a robot controller it is pointless to consider only the software implementation itself without looking at the manipulator with its mechanical and electrical restrictions. To ensure that the entire system

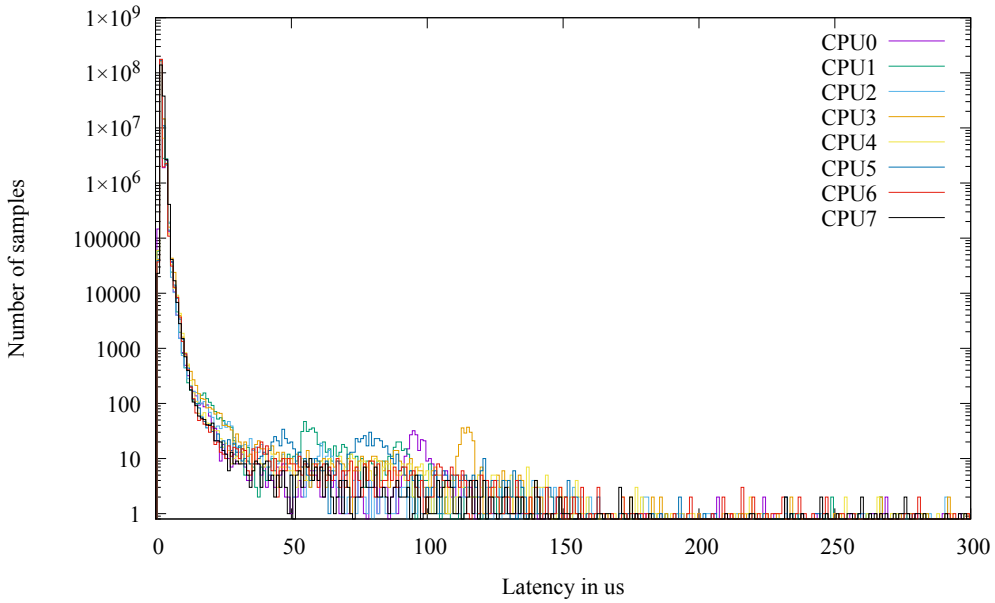


Figure 5.1: Cyclictest latency histogram of a normal OS. Absolute peak latency is 14.91 ms.

is able to meet the requirements, it must be predictable [35]. The meaning of predictability may vary from case to case [35] but shall not be discussed in more detail at this point.

In the present thesis the robot and its controller are a closed system provided by an external manufacturer and expected to work without any further changes. To be controlled, RT capability on the computer controlling the robot is required[1]. The Franka documentation suggests the PREEMPT_RT[2]-patch for Linux OS. Thus not a proprietary kernel, fast and very reliable, but only providing essential support to achieve RT-capability, or a research oriented RTOS, mainly used in the development of new OS, are needed, but only a commercial OS with RT capabilities [35]. They are generally slower and less predictable than the first two types, but as an extension to commercial OS they still provide most GPOS functionalities. This advantage outweighs the fact that this type of RTOS sometimes fails to meet a deadline. Since the worst case scenario is the controller and the robot being stopped, this is considered acceptable. Missing a deadline diminishes the performance of the system but does not lead to any damage, thus it can be called a firm real-time system[36].

This patch needs to be installed before operating the robot. When writing a controller with libfranka, real-time operating is handled by this library, so installing the patch is the only prerequisite for the operation of the robot. To check proper functioning of the patch, a test is run. Cyclictest[3] measures the difference between the intended and actual wake-up time of a thread to analyze the latencies of the system. The latencies can be due to hardware, firmware or the OS itself.

[1] `https://frankaemika.github.io/docs/installation_linux.html`, last accessed 14.03.2023

[2] `https://wiki.linuxfoundation.org/realtime/start`, last accessed 14.03.2023

[3] `https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start`, last accessed 14.03.2023

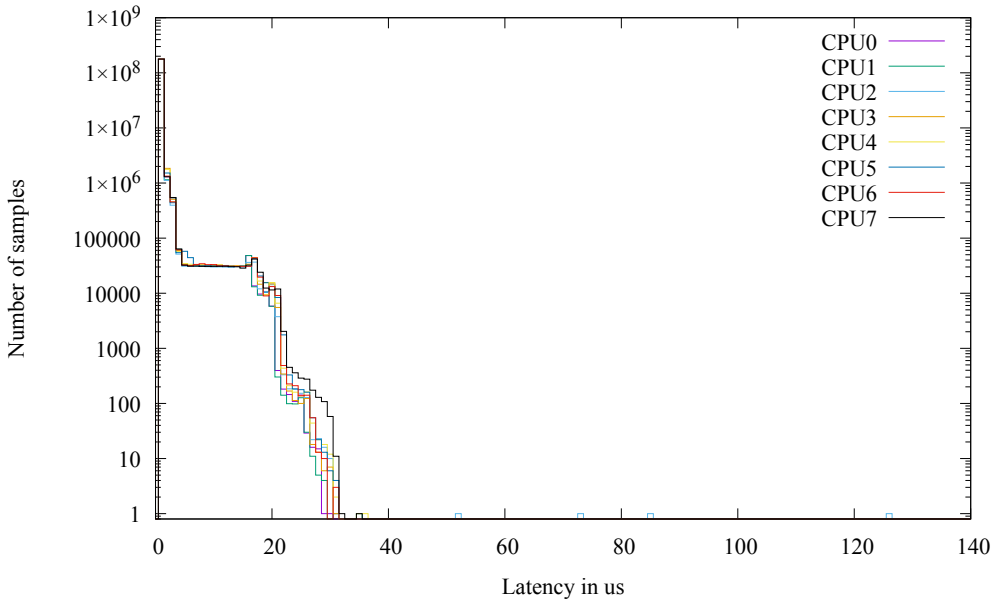


Figure 5.2: Cyclictest latency histogram of a RTOS. Absolute peak latency is 126 µs.

Figure 5.1 shows the result of cyclictest v1.00 on a GPOS, i.e. without any RT capabilities. For the test 180 million iterations at a distance of 200 µs each are executed. Since the PREEMT_RT-patch was already installed on the computer when the present work was started and it was not removed for the measurement, the present test is run on a computer with comparable hardware (Ubuntu 20.04.2 LTS 64 bit, AMD Ryzen 7 3700X 8-Core Processor 3.6 GHz, 16 GB RAM)[4]. While most response times are less than 50 µs, individual ones last up to several milliseconds, which is clearly not acceptable when RT capabilities are required.

Figure 5.2, on the contrary, shows the result of cyclictest v1.50, run with the same parameters as the previous one, but on the computer which controls the robot, i.e. with the PREEMT_RT-patch version 5.10.73-rt54 installed. This computer runs Ubuntu 20.04.4 LTS 64bit on a Intel Core i7-9700 3 GHz with 16 GB RAM. Almost all measured response times are below 30 µs and only few are longer. The all time longest measurement is 126 µs, which is still almost 120 times faster than the the longest measurement at the GPOS.

However, while the RT-patch is necessary for operating the robot, it has a major disadvantage. Even as noted in the Franka documentation[5], the kernel is incompatible with any NVIDIA graphics drivers needed to access the graphics card, which again is indispensable for rendering onto the HMD. Although some custom patches can be found online[6], which claim to solve the problem and to allow the user to install required drivers also on a RTOS, none of them have worked in this case, possibly because they were developed

[4]CPU comparison: `https://cpu.userbenchmark.com/Compare/AMD-Ryzen-7-3700X-vs-Intel-Core-i7-9700/4043vsm816180`, last accessed 07.04.2023

[5]see footnote 1

[6]`https://gist.github.com/pantor/9786c41c03a97bca7a52aa0a72fa9387`, last accessed on 15.03.2023

for older operating systems. In addition, workarounds of the like might often lead to problems as they avoid hurdles by simply removing restrictions with no guarantee that the original system is still operational afterwards. Referred to this context, the question would be if the RTOS was still RT-capable if it had to run a graphics card.

This being said, the decision is taken to outsource the VR-handling to another computer to ensure the functionality of both, the RTOS and the graphics drivers. Thus the first computer is only responsible for simulation and robot control. Since the simulation (with the corresponding robot movement) and the visualization must be synchronous in order to guarantee usability, the two computers must communicate with each other.

For this purpose, both computers need to be connected to the same local network and a ROS-master needs to be running on only one of them. Consequently, the other computer is given the master's computer IP-address[7]. In concrete terms, this means that on both computers the `ROS_MASTER_URI` (URI = unified resource identifier) must be set to the IPv4-address of the one computer which roscore runs on. Also the port must be specified, which by default is set to 11311. The `ROS_IP` must be set to each computer's own IP address. Since small latencies are less important for visualization than for robot control, it is decided to run the ROS-master on the computer controlling the robot to minimize latencies there. Now ROS-nodes on both computers are able to communicate as if they were on one computer.

Once this connection is established, the MBS simulation data (systemData) can be sent from the robot controlling PC to the PC responsible for visualization. To avoid the creation of a new ROS message type, which would require a catkin-workspace to be set up also on the VR-PC, the systemData, which consists of multiple one-dimensional vectors of `floats`, is packed into a one-dimensional `Float64MultiArray`-message of the ROS `std_msgs` package. To gain knowledge about where a vector ends and the next one starts, its length is inserted beforehand. For example, if the two vectors $\{8,5,7\}$ and $\{9,6\}$ were to be combined, the result would be $\{3,8,5,7,2,9,6\}$. Additionally, the current simulation time is sent to the VR-controlling computer, thus put at the front of the array, without previous length-entry.

On the VR-controlling computer this data structure is received, the single arrays of the systemData are rebuilt, and the visualization can be updated accordingly.

5.2 Exudyn

The entire concept of imposing trajectories to the robot to be traversed and the robot sending applied forces back to the simulation, would be obsolete if the simulation did not run in RT. Exudyn provides this possibility per default. It should, however, be noted that this possibility has nothing to do with conventional RT processing. If it is activated and the complexity of the MBS is low enough to enable the MBS to run faster than RT, the simulation is artificially slowed down for it to run in RT. If the complexity is too high to run in RT, the activation of this option does not have any impact. It can be activated by defining the following attribute of the TimeIntegrationSettings as `true`:

```
simulateInRealtime = True
```

This is essential since its default value is `False`. The attributes

[7] `http://wiki.ros.org/ROS/Tutorials/MultipleMachines`, last accessed 15.03.2023

```
realtimeFactor = 1
realtimeWaitMicroseconds = 1000
```

could be modified to change the RT behaviour of the simulation, but for this thesis they keep their default values, which are the ones given above. By changing the `realtimeFactor` the simulation could be slowed down (<1) or sped up (>1). By changing the `realtimeWaitMicroseconds` the update interval used by the simulation to check if it is still running in RT is modified. Lower values mean increased RT accuracy at the cost of higher CPU usage, while higher values mean lower CPU usage and decreased RT accuracy[8].

5.3 ROS

Apart from the simulation, the OS, and the robot, which must all be RT-capable, there is a fourth component crucial to timing: communication. As described above, inter-process communication is handled entirely through the ROS-network. As in ROS the not RT-capable TCP/IP protocol is employed, the ROS-network itself is not RT-capable[9], thus there not being any guarantee that messages arrive in time. Therefore a buffer for RSPs was introduced in subsection 3.4.2. Thereby the problem caused by the RT-incapability of ROS is bypassed at the cost of an increased delay between commanded and effective velocity change. The aim is to keep the size of this buffer small and the single segment lengths short. However, as measurements show, those two objectives cannot be optimized at the same time. For further analysis, two ROS nodes are initialized, one representing the sender, which publishes messages at a constant frequency f, and the other one as a listener, which awaits the messages. Both nodes run on the same PC as the ROS-master. The message type used to send RSPs from Exudyn to the ROS-controller is `SegmentCommand` with an additional `Header` field[10]. In the `Header`-field the current ROS-time is written just before the message is sent. When it arrives at the listener, the time sent along is compared to the current time and the difference/latency is logged.

Figure 5.3 shows the time it takes messages to arrive at a listener when sent with different frequencies. It is clearly visible that the increase in sending time the standard deviation σ are related to frequency. For low frequencies ($f = 30\,\text{Hz}$ and $f = 60\,\text{Hz}$) the sending time is constant at 5 ms with a standard deviation of $\sigma < 1\,\text{ms}$. For $f > 100\,\text{Hz}$ both, mean sending time and standard deviation, increase significantly and reach mean values of about 0.2 s at the end of the considered frequency range taken into consideration. Standard deviations fluctuate between 1 ms and 1.9 ms.

Although the messages are sent using TCP/IP, increasing frequency results in higher package loss. It may be that some messages may still be in a queue when the sending process is completed, but even after an additional waiting time of several seconds has elapsed, not all messages have arrived at the listener. At 50 Hz the loss percentage is

[8] `https://github.com/jgerstmayr/EXUDYN/blob/master/docs/theDoc/theDoc.pdf`, last accessed on 07.03.2023

[9] `https://docs.ros.org/en/foxy/Tutorials/Demos/Real-Time-Programming.html`, accessed 15.03.2023

[10] `https://github.com/bacaar/exuCobot/blob/main/util/msg/segmentCommandStamped.msg`, last accessed 15.04.2023

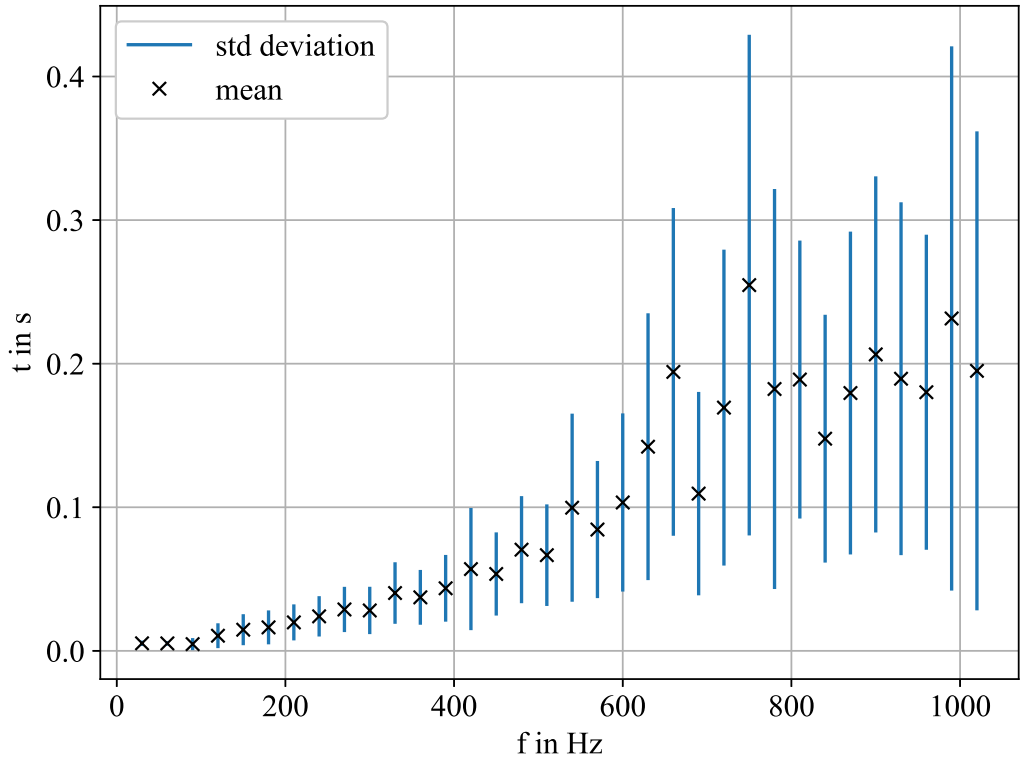


Figure 5.3: Sending times in the ROS communication at different sending frequencies. Each frequency has been recorded for 10 s.

about 0.2% and it increases by approximate 0.3% / 100 Hz until 850 Hz. Beyond that frequency, the loss rises dramatically to over 10% at 1000 Hz. Since the present application expects successive segments, such data loss is unacceptable.

Sending time depends not only on frequency, but also on the size of the data sent. A smaller package takes less time to arrive at the subscriber. Reducing the data by sending only the position while estimating velocity and acceleration is therefore taken into consideration. Data would so be reduced to a third of its original size, thus allowing for shorter sending times at higher frequency. It is, however, unnecessary to apply a higher frequency in the opposite direction, since the efforts applied to the robot are published with only 30 Hz and the trajectory remains the same until a new effort is received. This is because the Exudyn robot-client and the ROS controller are not synchronous: both run independently from each other and Exudyn considers received efforts as current and valid as long as no new ones are received.

Therefore, the proposal to not transmit velocity and acceleration is discarded and instead it is decided instead to apply low frequency: the trajectory segments are sent with 30 Hz, i.e. have an approximate length of 0.033 s. For this segment length a nominal command buffer size of 3 is chosen empirically.

6 User-interface

In the present chapter the various user-interfaces elaborated in this thesis are described. These include the Python programming interface and how to use it to make an Exudyn MBS simulation VR-capable, as well as the physical interface which the user can grab and interact with the robot. Finally, safety precautions which have been taken are discussed.

6.1 Programming interface

Even if throughout this thesis the interfaces are tested with the single pendulum described in subsection 2.4.2, one of the main goals of this project is to make the interface usable for not just one but a variety of MBSs. Thus a programming interface is created to further facilitate the integration of the present project into new MBS Exudyn-simulations. The underlying principle is to specify the UIP in the simulation, position an Exudyn-marker at that location and pass it to the interface. Everything else is dealt with in the background. The above-mentioned interface is called `RobotVrInterface`[1] and described in the following.

6.1.1 Overview

To use the project of the present thesis to the full extent, both robot control and VR rendering must be handled simultaneously. However the interfaces to those two external devices must be implemented on two separate computers, as elaborated in section 5.1. For proper functioning the simulated and visualized MBSs need to be the same for both computers. In order to avoid implementing the same MBS twice with different interfaces, the `RobotVrInterface`, as the name suggests, is designed to address both interfaces. Depending on a parameter at its initialization it only uses one of them, though. In the following, the two scripts running on separate PCs are therefore referred to as robot-client or VR-client.

A simple way to determine which client to initialize, i.e. which interface to use, is using program arguments. This way the same Python Exudyn script can be used on both computers, but depending on the chosen parameter, only the corresponding interface is used. To determine whether a robot- or VR-client should be initialized, the `RobotVrInterface` does not only contain the interface, but also the function

```
def parseArgv(argv):
```

[1] available at `https://github.com/bacaar/exuCobot/blob/main/exudyn_interface/scripts/RobotVrInterface.py`, last accessed 13.04.2023

to parse the program arguments. It takes the list of strings, received via `sys.argv`, as parameter and searches for a string "r" or "robot" for the robot client or "v" or "vr" for the VR-client. Within the code the two clients are identified by the numbers `1` or `2`, respectively. Additionally, the function searches for a second, optional argument which must be either "-i" or "-v", to decide whether the impedance (`true`) or the velocity controller (`false`) should be addressed. By default the velocity controller is addressed. Note that the function only decides which one to address, not which one to use. The corresponding controller must be started separately[2]. It then returns a tuple, composed by an integer for the client and a boolean variable for the controller.

Once the program arguments are parsed, the interface can be initialized. The interface is divided into three classes: the `RobotVrInterface`, the `RobotInterface` and the `VrInterface`. While the latter two implement the actual interfaces, the first one only keeps track of which interface to use and then calls the respective method of the corresponding class.

Polymorphism, which would provide for the latter two classes to derive from the `RobotVrInterface`, thus enabling them to overwrite individual methods, is not used because a different class would have to be created in the MBS script according to the respective client. With the present implementation a `RobotVrInterface` is created with the appropriate parameters and all case distinctions are handled in the background.

6.1.2 Class methods

This subsection gives the reader an overview of the different class methods implemented in the interfaces and how they can be added to an existing Exudyn MBS. As outlined above, only the `RobotVrInterface` must be initialized and used, all interactions with `RobotInterface` and `VrInterface` are handled in the background. In the following the various class methods are described.

```
def __init__(self, clientType, useImpedanceController=
    False, rotMatrix=np.eye(3)):
```

According to the client type (1 = robot-client, 2 = VR-client) an instance of the respective class is created by this method. The `__init__()`-methods of the robot- and the VR-client initialize hard-coded class attributes. Additionally, the robot-client reads the current EE pose from the corresponding ROS-topic, which is necessary to map poses from the Exudyn coordinate system to the robot-base coordinate system (described in section 3.3.1). The parameter `useImpedanceController` tells the interface which topics it needs to listen to in order to gather relevant robot information and. accordingly, which topic should be sent new commands. The last parameter `rotMatrix` represents the rotation at which the model is seen by the VR-client initially. It can be used e.g. when the MBS coordinate system does not correspond to the VR coordinate system.

```
def determineRobotStartPosition(self,
    interactionPointOffset=np.array([0,0,0])):
```

[2] by using the command `roslaunch franka_example_controllers exucobot_cartesian_velocity_controller.launch robot_ip:=<fci-ip>`. The `franka_example_controllers` names the package the controller is located in.

reads the current robot EE pose from the corresponding ROS topic and combined with the base position of the robot it determines where the EE is located in the VR coordinate system. The parameter `interactionPointOffset` defines the offset between the position of the UIP and the origin in the simulation model. It returns the coordinates to be used as `referencePosition` when creating new Exudyn objects.

```
def createEnvironment(self, mbs):
```

only affects the `VrInterface`. There it creates the environment, which the user will see when wearing the HMD. It is currently implemented for use in the laboratory of the Institute of Mechatronics at the University of Innsbruck. Besides the floor also walls are created in the corresponding positions and with the additional function `createTable(...)`, which allows the user to place tables in arbitrary positions, although currently only parallel to the axes of the VR coordinate system, tables are placed in the same position as they are also placed in the real world. Since the HMD is completely opaque, these objects help the user to orientate themselves within the room. However, those objects are implemented only as Exudyn "ObjectGround"s, so they have no impact on the simulation.

```
def setUIP(self, marker, mbs):
```

adds a `loadVectorUserFunction` to the MBS: the passed marker must be at the position of the UIP in the simulation model. The user-function continuously maps the force current applied to the robot EE to the UIP within the simulation. The robot EE is published to ROS by the robot-controller and recorded by a ROS-callback function within the VR-client.

```
def update(self, mbs, SC, t):
```

is called once for every simulation iteration on the robot-client or once for every visualization frame on the VR-client. On the robot-client, the current position, velocity, and acceleration of the UIP are transformed to robot-base coordinates and published on the respective ROS topic, as these values represent a new RSP for the ROS controller. For the time being the work is limited to the translational part of the UIP's movement. As mentioned in section 5.3, in order to not overload the ROS network, this does not happen in every iteration. The same applies to the current Exudyn systemData, which is published together with the current simulation time on a separate ROS topic. So the VR-client can update its visualization. The VR-client then reads the published system state and updates the visualization accordingly. It also reads the current tracker pose from OpenVR via Exudyn and updates the position of the hand in the visualization.
The `update()`-method must be called from within Exudyn's `PreStepUserFunction`.

```
def simulate(self, mbs, SC, simulationSettings):
```

handles the simulation itself. Before the Exudyn render process is started, the method first adapts some settings in the sytemContainer (SC) and the simulationSettings, which are relevant for visualization and simulation respectively. The robot-client starts the Exudyn simulation where the MBS is simulated and the `update()` method is called once in each iteration by the `PreStepUserFunction`. The VR-client calls the `update()`

method directly from within a `while`-loop and commands the Exudyn renderer to update the graphics. It quits if the user sends a Exudyn stop signal.

6.1.3 Parameters

The `RobotVrInterface` can almost be used as it is except for five parameters defined at the beginning of the file. Those might have to be modified by the user:

- `HHC_POS_IN_VR_FRAME`: It defines the position of the HCC in VR coordinates and is the first part to locate the robot. To obtain it, the user compiles and runs a specific program[3], while the controller is positioned in its mounting station at the robot table. It must consist of three values [x, y, z], e.g. put into a one-dimensional numpy[4] array.
- `HHC_TO_ROBOT_BASE`: It represents the second part of locating the robot and describes the translation between HCC and robot base coordinate system. It must also consist of three values which define the mentioned translation in VR-coordinates.
- `VR_POS_CORRECTION`: Although the robot EE should be locatable with the above two constants and the robot joint configuration, during testing the placement of the MBS appeared to be too high in the VR view. With this constant (three coordinates [x, y, z] in the VR frame) the UIP-position can be translated in the VR view, making it more intuitive to the user. It is to be determined empirically.
- `HAND_MODEL_OFFSET`: This constant describes the translation between the origin of the tracker mounted on the hand of the user and the origin of the hand model seen in the VR view. Depending on the hand size of the user, this constant needs to be changed empirically. It should be noted that due to the mounting method the correct display of the hand in every possible wrist position is impossible. Determining this constant with millimeter precision is therefore not important, as it will be wrong as soon as the wrist moves.
- `VR_FPS`: Defines the frame rate at which the visualization is rendered to the HMD.

6.1.4 Tracker bug

There seems to be a bug in OpenVR / SteamVR when it comes to identifying VR-devices, especially trackers. When the program is started and the tracker is powered on only afterwards, the tracker is identified correctly as a tracker. However, if the tracker has been powered on before starting the program, the tracker is identified as an HHC. So in the current implementation the program searches for a tracker or an HHC and renders the hand to the position of the first device found. For this reason it is necessary to switch off all other HHCs and trackers when using the system, as the hand could be rendered at one of their locations instead.

6.2 User-robot interface

To enable physical interactions between the user and the robot, a suitable EE-extension needs to be mounted on the EE of the robot. Theoretically, the user could also grab the

[3] source code available at `https://github.com/bacaar/exuCobot/tree/main/util/locateVrDevices`, last accessed 25.04.2023

[4] `https://numpy.org/`, last accessed 25.04.2023

robot itself and apply efforts anywhere. However, the default EE of the robot, a gripper, is not shaped and meant for user interaction. What is more, applying efforts to arbitrary places on the robot would not correspond to the place where the transmitted effort is applied within the MBS simulation. A well-defined user interaction point (UIP) in the real world, which corresponds to the UIP in the simulation, is therefore required.

The Franka Emika Panda robot has a standardized EE flange (DIN ISO 9409-1-A50 "Manipulating industrial robots - Mechanical interfaces - Part 1: Plates")[5], which provides the possibility to mount various objects on it. As the mounting mechanism consists of only four bolts arranged in a square, such object cannot only be an attachment from Franka Emika itself, but also one from any third party, i.e. also self-made. An object is to be constructed and fulfill three firm requirements.

1. The object must be mountable on the robot's flange according to the previously mentioned standard.
2. The object must be able to be grabbed by the user easily and safely.
3. The object must be stiff and not deform or break when efforts are applied.

Additionally, three further aspects (weak requirements) should be considered:

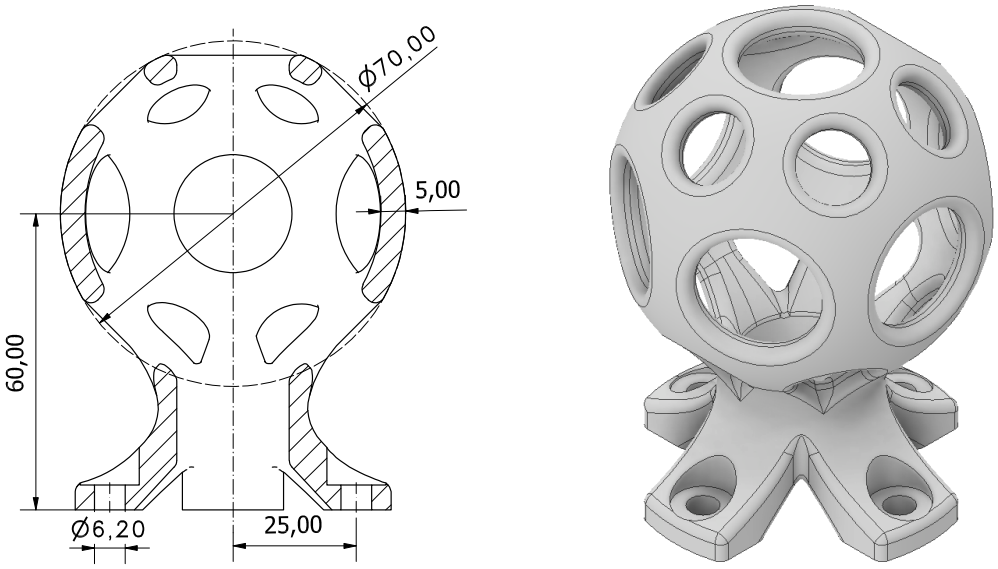
1. Rotational symmetry around the axis perpendicular to the robot's flange allows for the object to be grabbed from either side, eliminating the need for reassembly when the user changes position.
2. Light weight helps to save material and prevents unnecessary efforts being applied to the robot.
3. A striking color helps to make it clear that this object is meant to be grabbed by the user.

Since there are no contradictions in those six requirements, a object can be constructed which fulfills them all. The combination of firm requirement 2 and weak requirement 1 suggests that the basic form is based on a spherical shape. It can be grabbed from either side, even from below. To fit into the hand of the user, the diameter is empirically chosen to be $d_s = 70\,\text{mm}$. This is also a commonly used size for other round objects to grab, e.g. door knobs[6]. To meet firm requirement 1, for the flange connection four uniformly circularly arranged holes for M6 bolts are designed in the cross-shaped object-flange at a distance of $r = 25\,\text{mm}$ to the central and flange-perpendicular axis. The distance between sphere and mounting point is of no great importance, it just has to be known (60 mm) and inserted in Franka Desk afterwards. Both structures, object-flange and the sphere, are connected with a freeform curve. The result is depicted in Figure 6.1. Since it has the shape of a knob, it is referred to as such below.

To meet weak requirement 2, the sphere is hollowed. Only an outer shell with a thickness of $t_s = 5\,\text{mm}$ remains and additional holes are left, too, to further reduce weight and material. Since the knob has to transfer only small efforts to the EE flange (e.g. $|\text{f}|_{max} < 10\,\text{N}$ in Figure 3.18), these structural changes should not affect its rate of deformation with the applied forces being so low. Therefore, no further investigations are made in this regard.

[5] `https://www.wiredworkers.io/wp-content/uploads/2019/12/Panda_FrankaEmika_ENG.pdf`, last accessed 12.04.2023

[6] see `https://www.baunetzwissen.de/beschlaege/fachwissen/tuerbeschlaege/tuerdruecker-150232` (which refers to DIN 18255), last accessed 14.04.2023



(a) Section of CAD-model with selection of relevant dimensions. Units in mm.

(b) 3D-view of CAD-model.

Figure 6.1: Knob-object used as UIP.

When the design is finished[7], the knob is manufactured. Because of its geometry it is decided to be constructed using additive manufacturing (fused deposition modeling), also known as 3D-printing. As filament red PLA is used.

6.3 Safety

With a human and a robot working closely together and even interacting with each other, safety measures cannot be neglected. Already by default, the used robot, which actually is a cobot, has a built-in force threshold[8], which stops any robot's movement when exceeded. This threshold can be modified in Cartesian or joint space. However, as the principle of of this project only works if efforts can be transferred from the user to the robot and vice versa, this threshold must not be set too low, as then the robot stops when efforts are applied. The thresholds are specified in the franka_control_node.yaml[9] and are set to 20 N per axis per default. In order to give the user a little more leeway, it has been increased to 40 N. Although even this doubled threshold does not pose any serious danger to the user, it is strongly recommended to approach the robot with hands/arms only and keep the rest of the body, i.e. head and thorax, at a safe distance. Being hit by the robot at high speed can still hurt, since both the manipulator and the knob are rigid objects.

For further safety, the emergency stop button should always be within the user's range and there should be no obstacles in the room except for those rendered in VR.

[7] CAD-model available at `https://github.com/bacaar/exuCobot/tree/main/3d_models/knob`, last accessed 14.04.2023

[8] `https://frankaemika.github.io/docs/franka_ros.html`, last accessed 13.04.2023

[9] available at `https://github.com/bacaar/exuCobot/blob/main/franka_ros/franka_control/config/franka_control_node.yaml`, last accessed 13.04.2023

7 Results

With robot control, VR rendering, and the user interface in place, the user should now be able to interact with the robot and investigate how this affects the MBS both haptically and visually. This chapter shows such process from multiple angles, evaluates it and lists some restrictions for simulations with the current implementation.

7.1 Robot control

Again the MBS described in subsection 2.4.2 is simulated. The trajectory in y-direction can be seen in Figure 7.1. This time the pendulum does not experience a short force application at the beginning, as it was the case in Figure 3.18, but is deflected statically.

In contrast to Figure 3.18, where the delay between reference and measured trajectory is 76 ms, the delay in this example amounts to 115 ms. This is due to the fact that the sending frequency has been decreased in section 5.3 in order for the whole system to be more robust to latencies in the ROS communication. This delay is, however, still less than half the delay measured when using the impedance controller. The biggest part of the delay is caused by the buffer. As the buffer per default holds 3 entries and each entry represents a command for a segment length of 33 ms, it is already responsible for a delay of 99 ms. The remaining 16 ms are due to the time needed to fill the buffer. As ROS is not RT-capable there might also be some inconsistencies or inaccuracies in its clock.

Because of this non-negligible delay it is difficult for the user to hold the system on a fixed point outside its rest position. An oscillation at around $t = 1.5\,\mathrm{s}$ is visible in Figure 7.1, where the pendulum tries to return to its rest position. The user immediately increases the force to counter this movement, which makes the y-position increase. This effect is not intended by the user, who now reduces the applied force, this time to zero. This puts the pendulum in free oscillation. Its amplitude diminishes because of the rotational damper at its pivot.

impedance controller	velocity controller v1	velocity controller v2
	$4 \cdot 10\,\mathrm{ms}$ (buffer)	$3 \cdot 33\,\mathrm{ms}$ (buffer)
	$+36\,\mathrm{ms}$ (buffer filling)	$+16\,\mathrm{ms}$ (buffer filling)
270 ms (total)	$= 76\,\mathrm{ms}$ (total)	$= 115\,\mathrm{ms}$ (total)

Table 7.1: Comparison between latencies of different controller types and implementations. The velocity controller v1 represents the implementation used in chapter 3, the velocity controller v2 the one developed in chapter 5 and used in present chapter.

Table 7.1 gives an overview of the delays occurring with different controllers and

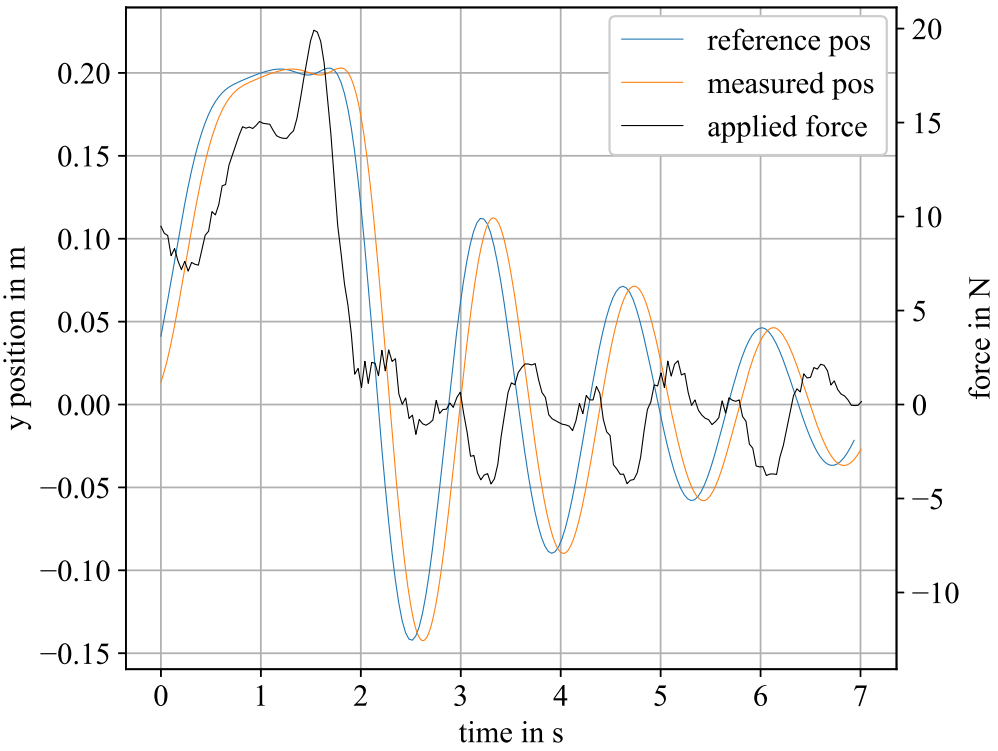


Figure 7.1: Trajectory of pendulum in y direction with initial static deflection followed by damped swinging.

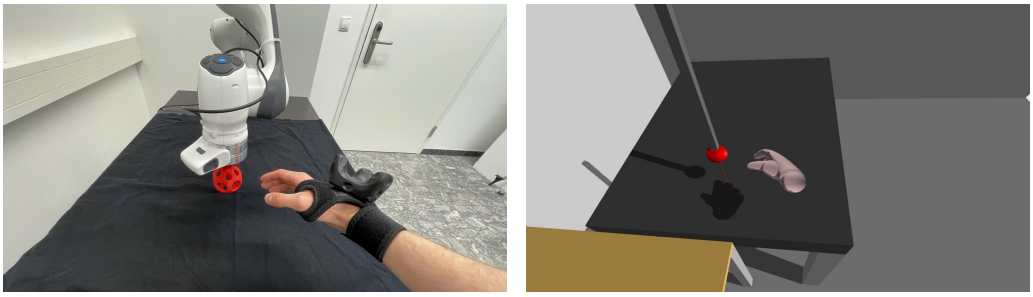
their implementations. The impedance controller used at the beginning of the present thesis, i.e. at the beginning of chapter 3, shows a large latency between the simulation and the output of the corresponding movement on the robot. This latency is predominantly due to the characteristics of the impedance controller as there is no buffer implemented. The ROS sending time has a minor effect, too, but even with a sending frequency of 100 Hz it would take the network less than 10 ms to send a message on average.

The first tested implementation of the velocity controller has a built-in buffer of 4 segments at 10 ms each, i.e. a latency of at least 40 ms occurs. In the measurement another 36 ms are lost while filling the buffer. The second version of the implementation of the velocity controller provides a buffer of 3 segments at 33 ms each, resulting in a total buffer time of 99 ms. The time employed to fill this buffer is shorter, because the average ROS sending time at 33 Hz is significantly shorter than at 100 Hz and has a smaller standard deviation, too.

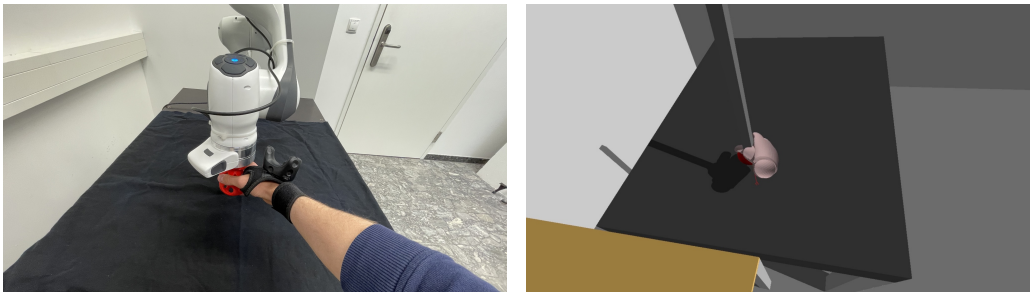
7.2 Visualization

A video showing the user interacting with the robot is available on the GitHub repository[1]. The reader of present thesis in paper-format can see a sequence of images, representing the key moments of the video, in Figure 7.2.

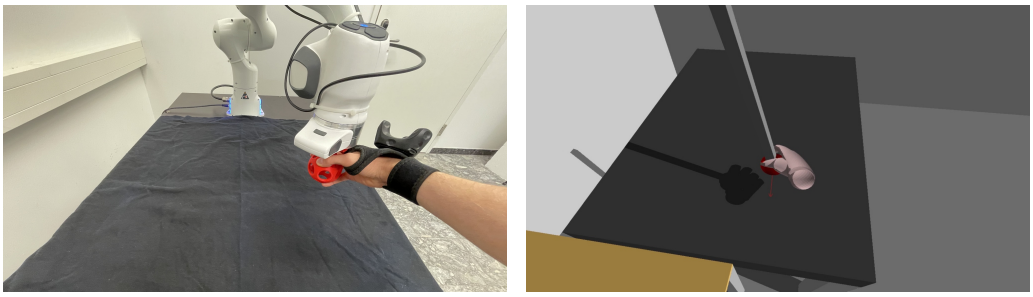
[1] `https://github.com/bacaar/exuCobot/blob/main/demo.mp4`, last accessed 16.05.2023. This video does not show the exact same measurement as depicted in Figure 7.1.



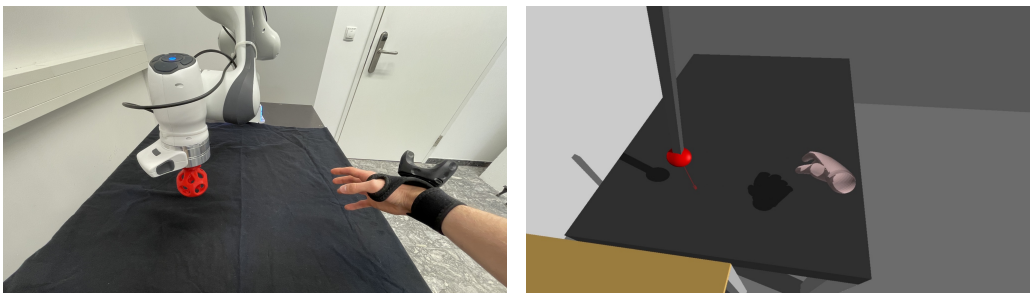
(a) The user approaches the UIP.



(b) The user grasps the UIP.



(c) The user deflects the pendulum.



(d) The pendulum has swung to the other side.

Figure 7.2: Image sequence of interaction procedure. The photos on the left have been taken from a different point of view than the rendering on the right.

As described in subsection 6.1.2, two tables and the walls of the real room are rendered at the correct positions in VR. Also the user's hand is updated and rendered at the correct position. These conditions help the user to navigate in the real room safely and to find the robot while wearing the HMD. The localization of the user's hand and the robot is done via the VR system and though not being very accurate, it still gives the user a rough idea

of the distance to other objects, which is enough for navigating and avoiding unexpected collisions when wearing the HMD.

7.3 Restrictions

While the visualization works fine for any kind of simulation, the current implementation for controlling the robot and thus the simulation itself, unfortunately, is still limited by a few restrictions.

7.3.1 Force and acceleration limits

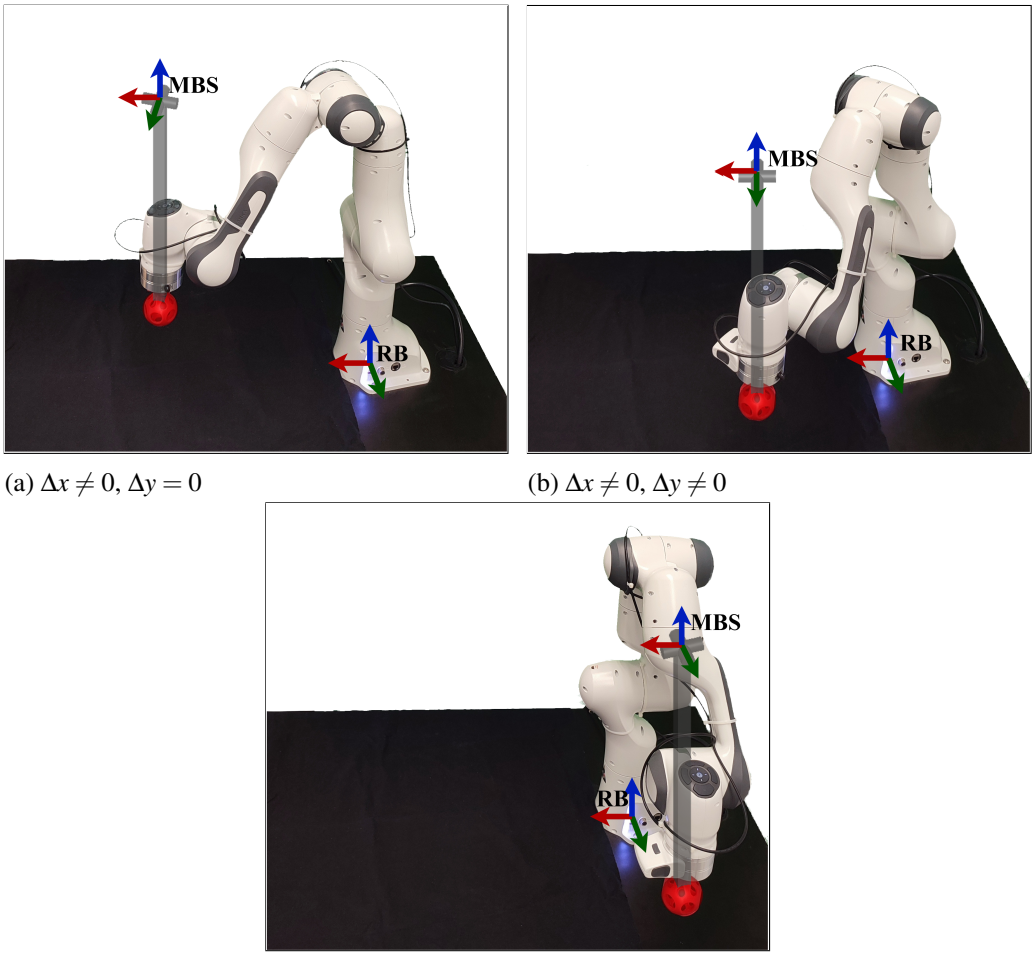
Due to the latencies the system in its current implementation is not perfectly suitable to deflect systems statically. This is observed in Figure 7.1. Also, because of acceleration, jerk, and force restrictions, punching the UIP results in the controller crashing and thus the robot stopping. Instead, the force applied to the UIP should be increased gently. When doing so, the simulation works fine as long as the ROS latencies do not exceed limits, as the command buffer is not able to compensate them anymore.

Furthermore, high frequencies, e.g. at an MBS with high stiffness, cannot be followed by the robot because of acceleration limits. The actual frequency limit is not known.

7.3.2 DOF limits

The attentive reader may have noticed that all measurements of the pendulum are always analyzed along the y-axis of the robot, i.e. with the robot moving around the x-axis of $\mathscr{K}_{\text{MBS}}$ and within the yz-plane. Only when trying the different compliance parameters at the impedance controller oscillations in x-direction are applied. The used MBS of the pendulum would perfectly allow for the pendulum to swing also within the xz plane. However, when allowing it to do so by passing it the applied forces, it ends up in an uncontrolled and undamped oscillation in that direction, although the model is damped equally in all directions. So far, it has not been possible to completely clarify the reasons for it. The oscillation appears when simulating the same MBS with the impedance controller as well as when another model is simulated with both of them. So neither the MBS nor the controller seem to be responsible. Furthermore, the oscillation in x-direction also appears if forces, i.e. movements in the y-direction are locked.

At this point it should be noted that all measurements and tests throughout this thesis are performed with the EE's initial position at around ${}^{RB}\mathbf{p} = [0.4, 0, 0.1]^T$, units in m. This can be seen in Figure 7.3a. Seen from the top (negative z-direction), the x-axes (red) of $\mathscr{K}_{\text{RB}}$ and $\mathscr{K}_{\text{MBS}}$ are collinear to each other, thus the difference in y-direction Δy between the two coordinate systems is 0. If the manipulator starts in the configuration of Figure 7.3c, i.e. Δx between the two coordinate systems is 0, the robot is able to simulate an oscillation around the y-axis (green) correctly (EE movement in xz-plane). However, an undamped oscillation in y-direction occurs. When changing the EE's initial position to somewhere in between those two extremes, the unwanted oscillation is not limited anymore to only one axis, but is visible on both. Its direction still points to the base of the robot. An example configuration for such an EE pose is depicted in Figure 7.3b.



(a) $\Delta x \neq 0$, $\Delta y = 0$

(b) $\Delta x \neq 0$, $\Delta y \neq 0$

(c) $\Delta x = 0$, $\Delta y \neq 0$

Figure 7.3: Different EE starting positions for the MBS simulation. Depending on the position unstoppable oscillations in different directions occur. Axes coloring: x red, y green, z blue.

However, it should be noted that just a small deflection from $\Delta y = 0$ or $\Delta x = 0$ (some cm) does not immediately lead to the oscillations.
The only parameter that could be found to have an influence on the oscillation is the simulated mass. Per default the pendulum tip has a mass of 6 kg. While reducing it had no effect, increasing it (e.g. doubling) reduced the amplitude of the oscillation. Hence the problem might be connected to the registered efforts but is not further investigated at this point. Therefore the simulated MBS is currently limited to 2 translational degrees of freedom. In the case of the pendulum, the MBS itself has only one degree of freedom, however, its tip can move within a two-dimensional plane.

8 Conclusion

The present thesis provides the possibility to combine MBS simulations, haptic feedback and VR. For that purpose two interfaces for the MBS simulation program Exudyn have been developed. The first allows Exudyn to control a robot manipulator and to receive efforts applied to the robot, so that physical interaction with the simulated MBS is possible. Because of the modular design, a huge amount of communication between the single components is required, which accounts for non-negligible latencies. These are due to ROS and the network in general and make it impossible to reliably control the robot with the required frequency of 1 kHz. Instead, the simulated data is sent with reduced frequency, and fine interpolation is used for robot control. This is done with polynomial interpolation, which is able to recreate the trajectory accurately. Problems only occur if the arrivals of the reference sampling points at the controller are delayed.

The second interface renders Exudyn graphics to an HMD of a VR-system and makes it thus possible to immerse oneself into the world of virtual multibody systems. With the help of two VR-devices different objects can be located in the real world and their virtual counterparts are rendered at the correct location in VR.

With an easy-to-use programming interface those extensions to Exudyn can be added to existing MBS simulations. They can thus be used to get a first impression of how different systems behave when certain efforts are applied.

The described system consists of several different components. Although they have not been developed from scratch, communication and coordination turned out to be very complex. Therefore, and due to time constraints, the ultimate goal of the thesis has been limited to a proof of concept. This proof, however, has been fully completed and the feasibility could be demonstrated successfully.

As elaborated in section 7.3, the parts of the current implementation most liable to failure are the robot/ROS controller and communication. The velocity control principle which has been used already comes with some characteristics designed to help compensate for delays. These features, however, only work when acceleration is low and so no fast changes of direction and velocity occur. Catching up with the reference trajectory once the robot moves away from it would probably be the first issue to address. It can be assumed that ROS2 would be better suitable for communication as it is designed for real-time and industrial applications. It would also be possible to not use ROS at all and design a different, customized interface for the controller in the Exudyn C++ source code to minimize latencies.

A Appendix

A.1 How to start the system

The following section describes how the developed system can be started. The way how the computer can be connected with the robot is described in subsection 3.1.1. The installation of the VR devices is described in section A.2 and the connection between the two PCs, one for controlling the robot and running the simulation, the other for VR handling, is described in section 5.1.

A.1.1 Basic requirements

To start off, the two computers must be powered on and connected in a local network. It is recommended to use a small local network in order not to be influenced by other data flows and to keep any latencies low. Additionally, the robot must be powered on, connected to the corresponding computer and FCI must be enabled in Franka Desk, which can be accessed via the robot's local IP address in any modern browser. When also the robot's joints are unlocked, the status lights at the robot's base should light blue. As described in section 5.1, a ROS master node must be running on the PC connected to the robot and the ROS Network needs to be configured accordingly.

To be able to use Exudyn's VR capabilities, Exudyn must be built accordingly, as the standard version, which is received when Exudyn is installed via `pip`, is not sufficient. To use those capabilities, Exudyn must be compiled separately. To do so, download the source code from GitHub[1] and build it with

```
python3 setup.py install --parallel --openvr
```

The setup.py-file is located in the main/ directory of the repository. The parameter `-parallel` makes the program compile faster by enabling parallel processing, the parameter `-openvr` activates the VR capabilities. Having done this, Exudyn can use the OpenVR library.

In order for the simulation to be rendered on the correct location in VR, the positionn of the robot position must be determined and set accordingly in the `RobotVrInterface`. This is described in subsection 4.3.2 and subsection 6.1.3.

Once all those requirements are met and also the ROS packages, available on GitHub[2], have been built within a catkin workspace, the specific programs of the thesis can be started.

[1] `https://github.com/jgerstmayr/EXUDYN`, last accessed 15.05.2023

[2] `https://github.com/bacaar/exuCobot/tree/main`, last accessed 15.05.2023

A.1.2 Robot/simulation setup

First of all, the ROS controller must be started, as it provides the EE pose of the robot on a ROS-topic, which is needed by the `RobotVrInterface`. This can be achieved by calling

```
roslaunch franka_example_controllers exucobot_cartesian
_velocity_controller.launch robot_ip:=<fci-ip>.
```

from the computer's console. Instead of `<fci-ip>` the robot's IP address must be inserted. Then the MBS simulation can be started. This is done by calling

```
python3 <exudynProgram> r -v
```

from the console. Instead of `<exudynProgram>` the filename of the MBS simulation must be used, e.g. `example4_1.py` on the GitHub repository. The first parameter `r` tells the `RobotVrInterface` that it is responsible for controlling the robot. The second parameter `-v` tells the interface that it should address the velocity controller. The alternative would be `-i` for impedance controller, but then also the corresponding controller would have to be started via `roslaunch`. Once the program is started, the interface needs some seconds to calibrate the effort input from the robot. Therefore, the robot should not be touched in the meantime. This is also stated in the console output of the program. When the Exudyn rendering window opens, everything is ready and the robot can be used to physically interact with the simulation.

A.1.3 VR setup

While it is possible for the simulation to run with the robot only, it is not possible to use the VR system without the simulation, i.e. subsection A.1.2 must be done before starting the VR part, described in the following.

On the computer which is connected to the VR hardware, SteamVR (or the VR Monitor from Steam) must be started first. Once SteamVR is running and the HMD, the two base stations and the tracker on the user's hand are detected by the VR system, a copy of the `exudynProgram` used in the above section to start the simulation with the robot can be used to start the corresponding VR rendering:

```
python3 <exudynProgram> vr -v
```

The parameter `vr` tells the program to be responsible for VR rendering and not for commanding the robot. It needs to know which controller is used in order to listen to the correct ROS-topic to gather information about the EE pose of the robot. This is specified by the parameter `-v`. Furthermore, it should be noted that the current implementation has hard-coded the path to the hand's .stl-file. The directory containing that file must be therefore located in the same directory the `<exudynProgram>` is in.

Finally, it should be mentioned that with SteamVR it is possible to display the rendering on the HMD also on the desktop screen of the computer. To do so, the "Display VR View" option from the SteamVR menu is selected. This may be useful as in the Exudyn render window the inserted room walls in the VR-client might block the view of the MBS itself, depending on the orientation of the HMD.

A.2 VR installation

The following section describes the installation process of a VR system and offers a less cost-intense alternative, which does not contain all the features of the original though.

A complete VR setup consists of several components, which have to be set up on both the hardware and the software side. On the hardware side, a HMD is needed along with two base stations that track the position and orientation of the HMD and all controllers/-trackers in the room. Controllers or similar devices are optional. On the software side, SteamVR, a runtime environment, and OpenVR, a C++ SDK, are required.

Head mounted display

The hardware only requires some cabling, the software setup is described in the next section.

- The base stations should be mounted at two opposite corners of the work area on a tripod each at a minimum height of 2 m. They should face the center of the work area and be tilted 30-45° downwards. Both the user's subsequent sitting position and the other base station should be clearly visible. Both base stations require active power connection[3].
- The base stations communicate with the computer via radio. For this it is necessary to connect the radio receiver to the computer via USB and place it visibly for both base stations.
- The HMD is connected to the link box via cable. Three cables leave the linke box on the opposite side:
  - 1x displayport → to the GPU of the computer
  - 1x USB → to the computer
  - 1x power → power socket

  Once the setup is complete, pressing the blue button on the link box turns on the HMD.

HMD alternative

As an cheap alternative to expensive HMDs, e.g. the HTC VIVE, but also products from other manufacturers, and the rest of the setup described above, also a smartphone together with an appropriate software called Riftcat[4] can be used to emulate an HMD. Here, no hardware setup, e.g. base stations, is necessary, but the software side is slightly more complex. However, it should be noted that this inexpensive alternative brings about three major drawbacks:

- Although the smartphone is used as an HMD, it cannot be used in practice without a suitable mount (e.g. Google Cardboard).

[3] A more detailed explanation for installing the base stations can be found at `https://www.vive.com/hk/support/vive/category_howto/installing-the-base-stations.html`, last accessed 11.1.2023.

[4] `https://riftcat.com/vridge`, last accessed 20.04.2023

- Unlike a real HMD, where position and orientation in space are tracked, only the orientation of the smartphone is passed on to the computer here. Hence, it is only suitable for development purposes to test basic functionalities.
- The system cannot be used longer than 5 min or 10 min if a free account is created. For unlimited use a premium version must be purchased. For development purposes the free version is sufficient since after time expiration the system can be restarted quickly.

For usage, it is necessary to install the software on the PC (only tested with Windows 10) as well as the associated app VRidge on the smartphone (available for Android and iOS, only tested with Android), available in the respective app store. Afterwards, the smartphone can be connected to the computer via WIFI or USB with the help of the two applications. The transfer is started via the big "Play" button in the desktop application. SteamVR opens automatically if installed (see next section).

SteamVR

SteamVR is a runtime environment that makes the entire VR system available to the computer/user. The installation is done via Steam, which in turn is a distribution platform for computer games and other PC software from Valve Corporation. After installing the Steam client[5] and creating a corresponding free account, SteamVR can be easily obtained and installed for free via the client's internal store. The VRMonitor can be used without creating an account. A PC restart may be required to run SteamVR correctly.

If the program is executed and a HMD, either a "real" HMD or the smartphone alternative with RiftCat running in the background on the same computer, is connected to the system and powered on, the *room setup* should start automatically when it is executed for the first time, thus calibrating the entire setup. If this is not the case, it can also be started manually via the menu in the upper left corner of the window.

OpenVR

OpenVR is an SDK for developing VR applications with C++. It is provided by Valve via GitHub as open source and under the BSD-3 - license [6]. While the code that can be created with the SDK is basically independent of the operating system, a different version of the SDK must be installed depending on the operating system and compiler. The files available on GitHub are suitable for Microsoft Visual C++, but not for g++. For the latter e.g. MSYS2[7] provides a suitable version on Windows. On Ubuntu Linux a suitable version can be acquired via the OS' internal package manager.

For compiling with g++ a special linker option is required: `-lopenvr_api` (tested with g++ 11.2.0 on Windows 10 and Ubuntu 20.04). Microsoft Visual C++ probably also requires something equivalent. On GitHub example programs can be found[8] for which the Visual Studio project files have already been configured for.

[5] `https://store.steampowered.com/about/`, last accessed 11.01.023

[6] `https://github.com/ValveSoftware/openvr`, last accessed 20.04.2023

[7] `https://www.msys2.org/`, last accessed 20.04.2023

[8] e.g. `https://github.com/ValveSoftware/openvr/tree/master/samples/hellovr_opengl`, last accessed 20.04.2023

Bibliography

[1] M. Stamer, J. Michaels, and J. Tümler, "Investigating the benefits of haptic feedback during in-car interactions in virtual reality," in *HCI in Mobility, Transport, and Automotive Systems. Automated Driving and In-Vehicle Experience Design: Second International Conference, MobiTAS 2020, Held as Part of the 22nd HCI International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I 22*, pp. 404–416, Springer, 2020.

[2] J. Gerstmayr, "Exudyn – A C++ based Python package for flexible multibody systems," Research Square, 2023. doi: 10.21203/rs.3.rs-2693700/v1.

[3] M. M. Moniri, F. A. E. Valcarcel, D. Merkel, and D. Sonntag, "Human gaze and focus-of-attention in dual reality human-robot collaboration," in *2016 12th International Conference on Intelligent Environments (IE)*, pp. 238–241, IEEE, 2016.

[4] D. Q. Huy, I. Vietcheslav, and G. S. G. Lee, "See-through and spatial augmented reality-a novel framework for human-robot interaction," in *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*, pp. 719–726, IEEE, 2017.

[5] P. Evrard, F. Keith, J.-R. Chardonnet, and A. Kheddar, "Framework for haptic interaction with virtual avatars," in *RO-MAN 2008-The 17th IEEE International Symposium on Robot and Human Interactive Communication*, pp. 15–20, IEEE, 2008.

[6] C. Gatterer, "Virtual Reality zum Anfassen: VR-System mit robotergestütztem haptischem Feedback: VR system with robotic haptic feedback," Master's thesis, Wien, 2017.

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[8] A. Mahtani, L. Sanchez, E. Fernandez, and A. Martinez, *Effective robotics programming with ROS*. Packt Publishing Ltd, 2016.

[9] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[10] J. E. Colgate, W. Wannasuphoprasit, and M. A. Peshkin, "Cobots: Robots for collaboration with human operators," in *Proceedings of the 1996 ASME international mechanical engineering congress and exposition*, 1996.

[11] T. Larsson, *Multibody dynamic simulation in product development*. PhD thesis, Luleå University of Technology, 2001.

[12] E. J. Haug, *Computer aided kinematics and dynamics of mechanical systems*, vol. 1. Allyn and Bacon Boston, 1989.

[13] M. A. Muhanna, "Virtual reality and the cave: Taxonomy, interaction challenges and research directions," *Journal of King Saud University-Computer and Information Sciences*, vol. 27, no. 3, pp. 344–361, 2015.

[14] I. Wohlgenannt, A. Simons, and S. Stieglitz, "Virtual reality," *Business & Information Systems Engineering*, vol. 62, pp. 455–461, 2020.

[15] P. Milgram and F. Kishino, "A taxonomy of mixed reality visual displays," *IEICE TRANSACTIONS on Information and Systems*, vol. 77, no. 12, pp. 1321–1329, 1994.

[16] R. Skarbez, M. Smith, and M. C. Whitton, "Revisiting milgram and kishino's reality-virtuality continuum," *Frontiers in Virtual Reality*, vol. 2, p. 647997, 2021.

[17] S. Chitta, I. Sucan, and S. Cousins, "Moveit!," *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 18–19, 2012.

[18] N. Hogan, "Impedance control: An approach to manipulation: Part I—Theory," *J. Dyn. Sys., Meas., Control.*, vol. 107, March 1985.

[19] G. J. Lahr, J. V. Soares, H. B. Garcia, A. A. Siqueira, and G. A. Caurin, "Understanding the implementation of impedance control in industrial robots," in *2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)*, pp. 269–274, IEEE, 2016.

[20] L. Sciavicco and B. Siciliano, *Interaction Control*, pp. 271–294. London: Springer London, 2000.

[21] N. Hogan, "Impedance control: An approach to manipulation: Part II—Implementation," *J. Dyn. Sys., Meas., Control.*, vol. 107, March 1985.

[22] A. D. Luca, "Robotics 2 - impedance control." Department of Computer, Control and Management Engineering, Sapienza University of Rome, 2020.

[23] F. Lange, M. Frommberger, and G. Hirzinger, "Is impedance-based control suitable for trajectory smoothing?," *IFAC Proceedings Volumes*, vol. 39, no. 15, pp. 31–36, 2006.

[24] A. Zelenak, C. Peterson, J. Thompson, and M. Pryor, "The advantages of velocity control for reactive robot motion," in *Dynamic Systems and Control Conference*, vol. 57267, p. V003T43A003, American Society of Mechanical Engineers, 2015.

[25] P. K. Padhy, T. Sasaki, S. Nakamura, and H. Hashimoto, "Modeling and position control of mobile robot," in *2010 11th IEEE International Workshop on Advanced Motion Control (AMC)*, pp. 100–105, IEEE, 2010.

[26] H. Shen, J. Fu, Y. He, and X. Yao, "On-line asynchronous compensation methods for static/quasi-static error implemented on cnc machine tools," *International Journal of Machine Tools and Manufacture*, vol. 60, pp. 14–26, 2012.

[27] T. Kröger and F. M. Wahl, "Online trajectory generation: Basic concepts for instantaneous reactions to unforeseen events," *IEEE Transactions on Robotics*, vol. 26, no. 1, pp. 94–111, 2009.

[28] K. M. Lynch and F. C. Park, *Modern robotics*. Cambridge University Press, dec 2019.

[29] K. D. Nguyen, T.-C. Ng, and I.-M. Chen, "On algorithms for planning s-curve motion profiles," *International Journal of Advanced Robotic Systems*, vol. 5, no. 1, p. 11, 2008.

[30] G. Huber and D. Wollherr, "An online trajectory generator on se (3) for human–robot collaboration," *Robotica*, vol. 38, no. 10, pp. 1756–1777, 2020.

[31] L. Berscheid and T. Kröger, "Jerk-limited real-time trajectory generation with arbitrary target states," *arXiv preprint arXiv:2105.04830*, 2021.

[32] J. LaSalle, "Time optimal control systems," *Proceedings of the National Academy of Sciences*, vol. 45, no. 4, pp. 573–577, 1959.

[33] J. J. Craig, *Introduction to Robtotics - Mechanics and Control*. Pearson Education International, third ed., 2005. chapter 7.3.

[34] S. Macfarlane and E. A. Croft, "Jerk-bounded manipulator trajectory planning: design for real-time applications," *IEEE Transactions on robotics and automation*, vol. 19, no. 1, pp. 42–52, 2003.

[35] K. G. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[36] H. Kopetz and W. Steiner, *Real-time systems: design principles for distributed embedded applications*. Springer Nature, third ed., 2022.

Verpflichtungs- und Einverständniserklärung

Ich erkläre, dass ich meine Masterarbeit selbständig verfasst und alle in ihr verwendeten Unterlagen, Hilfsmittel und die zugrunde gelegte Literatur genannt habe.

Ich nehme zur Kenntnis, dass auch bei auszugsweiser Veröffentlichung meiner Masterarbeit die Universität, das/die Institut/e und der/die Arbeitsbereich/e, an dem/denen die Masterarbeit ausgearbeitet wurde, und die Betreuerin/nen bzw. der/die Betreuer zu nennen sind.

Ich nehme zur Kenntnis, dass meine Masterarbeit zur internen Dokumentation und Archivierung sowie zur Abgleichung mit der Plagiatssoftware elektronisch im Dateiformat pdf ohne Kennwortschutz bei der/dem Betreuer/in einzureichen ist, wobei auf die elektronisch archivierte Masterarbeit nur die/der Betreuerin/Betreuer der Masterarbeit und das studienrechtliche Organ Zugriff haben.

Innsbruck am . . . .

Aaron Bacher, BSc