Konstruktion einer Sandbox für die Analyse von Kontrollalgorithmen und das Training von Robotern

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Klara Brandstätter, BSc
Matrikelnummer 01326465

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Eduard Gröller, TU Wien
Mitwirkung: Ass.Prof. Dr. Bernd Bickel, IST Austria

Wien, 26. November 2020

Klara Brandstätter	Eduard Gröller

Informatics

Building a Sandbox Towards Investigating the Behavior of Control Algorithms and Training of Real-World Robots

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Klara Brandstätter, BSc
Registration Number 01326465

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Eduard Gröller, TU Wien
Assistance: Ass.Prof. Dr. Bernd Bickel, IST Austria

Vienna, 26th November, 2020

Klara Brandstätter Eduard Gröller

Erklärung zur Verfassung der Arbeit

Klara Brandstätter, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. November 2020

Klara Brandstätter

Danksagung

Zuallererst möchte ich Bernd danken, für seine Geduld, und weil er mein Interesse für Robotik geweckt hat, und mir diese tolle Gelegenheit gegeben hat, an diesem neuen herausfordernden Projet zu arbeiten.

Danke auch an Edi, für seine weisen Ratschläge, dass man irgendwann akzeptieren muss, dass Dinge nicht immer so funktionieren wie man will, und dass man lernen muss, das auch rechtzeitig zu erkennen.

Danke an meine Mama, dass sie so ist, wie sie ist, die warmherzigste und geduldigste Person die ich kenne, auch dann noch, wenn meine Geduld sich dem Ende zuneigt.

Danke an Bubus (der Allgemeinheit bekannt als Andrea) und Judith, zwei meiner liebsten Freundinnen, mit denen ich über alles reden kann.

Zusammen mit meinen Freundinnen und Studienkolleginnen Michaela und Silvana haben wir unser Studium gemeistert, haben uns ermutigt und getröstet, besonders jetzt in den letzten Monaten. Mit ihnen zu studieren hat so viel mehr Spaß gemacht, und ich bin dankbar, für die wunderbare Freundschaft, die zwischen uns entstanden ist.

Danke auch an Gabi, die Freundin meiner Mama, der ich mathematische Probleme schildern konnte, und die mich mit Schokolade versorgt hat.

Danke an meine Katzen, besonders Minzi, die in stiller Gesellschaft neben meinem Computer schlief.

Danke auch an meine exotischen Zimmerpflanzen, die mich während der Arbeit mit Sauerstoff versorgten.

Acknowledgements

First of all, I want to thank Bernd, for awakening my interest in robotics, and giving me the opportunity to start this challenging new project, and for his patience.

Thank you to Edi, for his wise words that at some point, one has to accept that things do not always work as expected, and that one has to learn when it is time to move on.

Thank you to my mom for being who she is, she is the most warm-hearted and patient person I know. Even when I am not so patient anymore.

Thank you to Bubus (aka Andrea) and Judith, two of my dearest friends. I can talk about everything with them.

Together with my friends and fellow students Michaela and Silvana we mastered our studies, we encouraged and comforted each other, especially now during the final sprint to our master's degree. With them, studying was so much more fun, and I am thankful for the friendship.

Thank you to my mom's friend Gabi, who understood my mathematical problems and who supplied me with chocolate.

Thank you to my cats, particularly Minzi, who often slept silently next to my computer.

Thank you to my exotic house plants who provided oxygen during my work.

Kurzfassung

Die Kontrolle von humanoiden und tierähnlichen Robotern ist nach wie vor eine große Herausforderung. Methoden aus dem Bereich des maschinellen Lernens funktionieren bereits gut in Simulationen. Doch die Diskrepanz zwischen Simulation und Realität erschwert es manchmal, ebenso gute Resultate auf dem echten Roboter zu erreichen. Des Weiteren brauchen Lernalgorithmen eine enorme Menge an Trainingsdaten. Das Ziel dieser Arbeit ist die Konstruktion einer Sandbox, die es ermöglichen soll, simulierte und echte Roboter miteinander zu vergleichen, und die ein kontrolliertes und kontinuierliches Sammeln von simulierten und echten Daten unterstützt.

Die Sandbox besteht aus einer Motion-Capture-Komponente und einer Simulationskomponente. Die Motion-Capture-Komponente ist verantwortlich für die Datensammlung. Dafür wird ein System von OptiTrack mit sechs hoch präzisen Infrarotkameras verwendet. Die Simulationskomponente wird mit Simulink und der Simscape-Multibody-Library realisiert und ist verantwortlich für die Exploration und den Vergleich von Simulationsdaten mit realen Daten.

Für diese Arbeit wird ein vierbeiniger Roboter von ROBOTIS verwendet, der von 15 Dynamixel-Servomotoren gesteuert wird. Um den Roboter in die Sandbox zu integrieren, muss sein Controller neu programmiert werden. Das vereinfacht den Transfer von Bewegungsdaten auf den Roboter und ermöglicht es, den Roboter von der Ferne zu steuern. Der Roboter wird dann mit reflektierenden Markern versehen und seine Bewegungen werden aufgezeichnet. Mit der CAD-Software SolidWorks, wird ein 3D Modell des Roboters nachgebaut, welches für die Simulation in Simulink verwendet wird. Das Ergebnis ist ein System, das präzise die Bewegungsdaten eines kleinen Roboters sammeln kann. Diese Daten werden dann zur Simulation weitergeleitet und können dort mit Simulationsdaten verglichen werden. Simulationsdaten können auch einfach auf dem echten Roboter getestet und erneut aufgezeichnet werden. Das System ist also ein geschlossener Kreislauf, der iterative Roboterforschung ermöglicht.

Zwei Datensätze werden mithilfe der resultierenden Sandbox verglichen: Ein Datensatz von ROBOTIS, der ideale Gelenkswinkel für den Roboter beinhaltet, und ein Datensatz, der mit dem Motion-Capture-System gewonnen wird. Beide Datensätze werden für Simulationen verwendet. Die Position und Orientierung des simulierten Roboters werden mit den Motion-Capture-Daten verglichen. Trotz starker Variationen in den Simulationsergebnissen, behält der simulierte Roboter sein ungefähres Gangbild und weicht nur wenige Zentimeter vom echten Roboter ab.

Abstract

The control of legged robots and teaching robotic hands to grasp are still challenging tasks. Machine learning approaches already work well in simulation. However, the discrepancy between simulation and reality sometimes causes difficulties when applying simulation results to the real robot. Learning algorithms also require a huge amount of training data. The goal of this work is to build a sandbox that provides a detailed comparison between simulated and real-world robots and offers a way of controlled and continuous data collection and exploration.

The sandbox consists of a motion capture and a simulation component. The motion capture component is responsible for the continuous data collection and is realized with a system from OptiTrack with six high-precision infrared cameras. The simulation component is realized with Simulink and the Simscape Multibody Library. This component is responsible for the exploration and comparison of simulated data with real-world data. The robot that is selected for this work is a small four-legged puppy robot from ROBOTIS that is actuated with 15 Dynamixel servomotors. To integrate the robot into the sandbox, the robot's controller is reprogrammed to make a transfer from motion data to the robot easier and to control the robot remotely. The robot is programmed with a straight walking gait and equipped with reflective markers to track its movements.

With the computer-aided design (CAD) software SolidWorks, a 3D model of the puppy robot is constructed that is used for simulation in Simulink.

The result is a system that accurately gathers 6 degrees of freedom (DOF) data of a small robot. This data is transferred to the simulation and can be compared to simulated data. Data from the simulation can also be tested easily on the real robot and tracked again. This way, a closed-loop system is provided for iterative robot exploration.

Two datasets are compared with the help of the resulting sandbox: A dataset from ROBOTIS containing ideal joint angles for the robot, and a dataset that is obtained with the motion capture system, containing tracked joint angles. The datasets are simulated and the position and orientation of the robot are compared to the data from the motion capture. Despite strong variations in the simulated results, the simulated robot kept a similar direction and was only a few centimetres off from the real robot.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

This thesis was realized at the research group of Computer Graphics and Digital Fabrication at the Institute of Science and Technology (IST).

Nowadays, control strategies for robots are largely developed with the help of powerful physics simulations and a variety of machine learning algorithms. Especially challenging is the control of legged robots, but also the training of robotic hands to grasp any kinds of physical objects. Physics simulations are convenient because they do not require much hardware and are cheaper than experiments on real robots. A failed simulation has no severe consequences, whereas a failed walking gait, in reality, can damage the robot or even harm researchers by executing unexpected motion. Due to advances in machine learning, it is already possible to automatically design locomotion controllers that are robust and agile in simulation. However, when the control strategy is deployed on the robot the results are sometimes not satisfying and what has worked perfectly in simulation might not work as well in reality anymore. This discrepancy between simulation and reality is called the *reality gap*. Quantifying the reality gap is often challenging. It is difficult to predict real-world performance solely based on simulation outcomes because it is not known where and how big the differences between simulation and reality are. Also, simulations always have to face a trade-off between accuracy and speed. Thereby, it is often not clear which aspects of the simulation require precision. Further, for robotic grasping, learning hand-eye coordination or predicting object motion requires the acquisition of an enormous amount of training data [sima, icr, aig].
To enable a flawless control of real-life robots from simulations, it is important to have an environment at one's disposal that offers the possibility of continuous data collection and exploration. In the future, this could facilitate the creation of data-efficient learning algorithms and experimental evaluation.

1.2 Aim of the Work

The goal of this thesis is to build a sandbox towards investigating the behaviour of control algorithms and training of real-world robots. This sandbox should help researchers to get started experimenting with a robot performing various tasks (e.g. grasping an object or simple locomotion) and provide a basis for the development of data-driven models for the control of robots. It should allow researchers to accurately observe the motion of robots and collect relevant parameters such as the position of the robot and joint angles. The collected data should be visualized and compared to simulation data with the goal to provide deeper insight into the behaviour of simulation models and real-world robots. This thesis provides the reader with an exemplary sandbox design and explains important steps that can be helpful for designing one's own sandbox.

In the following, an overview of the design concept for the sandbox is given. The main components of the sandbox are introduced. The interaction between each of the components is explained and necessary hard- and software requirements are presented. In the following chapters, these components are discussed in detail.
The sandbox consists of two components that interact with each other. To use the sandbox a third component is needed, which interacts with the sandbox, namely a robot of interest. For this work, a four-legged robot from ROBOTIS™ is selected to analyse its walking motion with the help of the sandbox. ROBOTIS provides robot solutions and manufactures robotic hardware for study and industry, but also for educational purposes [robb]. The robot is actuated by 15 Dynamixel servomotors. The motors are connected to a controller that provides them with programmed motion patterns.
The first component of the sandbox is an optical motion capture system that is responsible for collecting information about the behaviour of the robot and how it performs programmed motion. With the motion capture system, position and orientation data from the robot's body parts are gathered. This is done with reflective markers that are attached to the robot. A system from OptiTrack™ is used that provides real-time and precise 6-DOF tracking using OptiTrack Prime 41 infrared cameras. An OptiTrack system is chosen because of its high accuracy tracking in comparison to other tracking methods, and because it is less expensive than systems from other manufacturers (e.g. Vicon).
The second component of the sandbox is a simulation tool that is meant to generate artificial motion data for a virtual robot model. With this tool, it is also possible to analyse motion from different simulations. The robot model can be animated with the tracking data from the motion capture that can be considered as a ground truth. The tracking data can also be used for simulation, which enables a direct comparison between real-life data and simulated data. This comparison could later be useful for tweaking simulation parameters. This work only approaches the comparison of different simulation results with real-life data. Optimizing simulations is not part of this thesis. Also, machine learning algorithms and training of robots are not yet supported. The sandbox is designed to be a closed-loop, which makes it possible to load simulation results onto the robot, such that these results can again be tracked and then compared to the

simulation. The current simulations cannot yet generate suitable data for the real robot, but the functionality is there in case new algorithms are added. Simulation and analysis is done with MATLAB® Simulink®. Simulink is chosen because it is widely used for the development of physical simulations in robotics. It is well documented and provides many examples to get familiar with the software. Further, the MATLAB programming environment is already familiar, which makes development easier.
In Figure 1.1, the sandbox design is depicted graphically, showing the three components, the robot, and the sandbox with the motion capture system, and the simulation and analysis tool. The black arrows indicate the direction of interaction between the components. The real robot provides input for the motion capture system, which computes data that can be used by the simulation. In the future, the simulation should also generate data that can be used to control the real robot. This sandbox could help to iteratively improve the design of robot motion control.

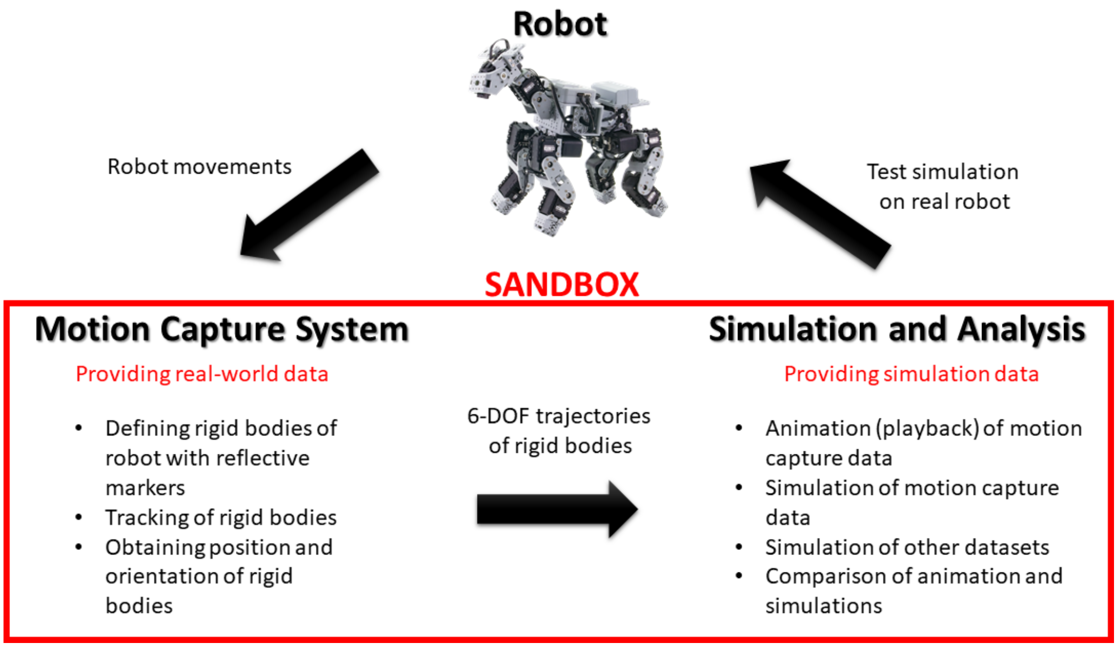

Robot
Robot movements
Test simulation on real robot
SANDBOX
Motion Capture System
Providing real-world data
• Defining rigid bodies of robot with reflective markers
• Tracking of rigid bodies
• Obtaining position and orientation of rigid bodies
6-DOF trajectories of rigid bodies
Simulation and Analysis
Providing simulation data
• Animation (playback) of motion capture data
• Simulation of motion capture data
• Simulation of other datasets
• Comparison of animation and simulations

Figure 1.1: Schematic sandbox design. The arrows indicate the interaction between the components [robc].

1.3 Structure of the Work

In Chapter 2, related work connected to the two sandbox components, the motion capture system, and the simulation tool is discussed. Robotic research that uses optical motion capture systems is addressed and simulation tools that are commonly used in robotic research.
The robot that is used with the sandbox is described in Chapter 3, before the sandbox

components are detailed. This is mainly because, the following chapters occasionally refer to the robot's structure, so it is helpful if this information is already present.
In Chapter 4, various motion capture systems in general are introduced. Requirements to the motion capture system are described and the setup of the OptiTrack system in particular is explained.
After the setup, the tracking of the robot is discussed in Chapter 5, providing information on how OptiTrack performs the 3D reconstruction from markers and mentioning important facts to consider, when placing markers on an object, and particularly on the robot used in this work.
Chapter 6 deals with the implementation of the simulation tool with MATLAB and Simulink. The concepts of kinematics and dynamics are introduced in the context of robotics. The creation of the 3D simulation model for the robot is described, and the processing of simulation data is discussed.
In Chapter 7, the results of this work are summarized. The sandbox as a whole is presented once more in a compact manner, pointing out its capabilities and limits. Further, a motion pattern of the robot is analysed with the help of the sandbox and compared to simulated data.
Chapter 8 provides ideas that could be implemented in the future to improve the sandbox and make it more versatile. A final reflection of this work and the challenges it posed is given too. This chapter concludes the thesis with a summary of the achieved work.

CHAPTER 2

Related Work

In this chapter, relevant literature, projects, and applications related to this work are discussed. Since the sandbox consists of the motion capture component and the simulation component, related research is split into two sections. One section focuses on robotic projects that use optical motion capture systems, and the other section deals with physics simulation tools and software designed for robot control. Only papers that mention the use of optical motion capture systems explicitly have been taken into account. Search terms used were 'OptiTrack', 'Vicon', 'motion capture', in combination with 'robot' or 'robotic'. The description of the papers focuses on the parts that are realized with the motion capture systems. Of course, optical motion capture systems are also widespread in sports sciences, biomechanics, and entertainment, but this is not discussed in this chapter, since no robots are involved [NK18].

2.1 Optical Motion Capture Systems in Robotics Research

Optical motion capture systems provide a way of validating the movements of robots against simulations.
A team of researchers around Jean-Baptiste Mouret work in the context of the *Resibots* project on algorithms for low-cost robots that are able to recover from unexpected damage automatically within a few minutes [res]. They state that robots are fragile and are far from being as resilient as the simplest animal when it comes to functioning in difficult conditions. Animals manage to adapt to a multitude of injuries.
The focus of this project lies on trial-and-error learning algorithms that enable robots to develop compensatory behaviours when they are damaged [CCTM15]. These algorithms aim to bridge the reality gap. Usually, closing the reality gap is tried by optimizing simulations and making policies more robust, e.g. by using domain randomization. With the trial-and-error approach, the high-dimensional policy parameter space is searched for

thousands of varying, high-performing solutions for a task such as walking. They are stored in a low-dimensional behaviour map. On the real robot, the map is searched for the most adapted policy, using Bayesian optimization that takes the simulation results as prior. The real robot can be damaged, e.g. having (multiple) blocked legs. It is assumed that among those behaviours, some bridge the reality gap better than others.
The algorithms are tested on four- or six-legged robots. The four-legged robot is pictured in Figure 2.1. Experiments on the real robots are conducted in a 5.5 m $\times$ 6 m area, using an OptiTrack motion capture system with eight Prime 13 cameras [res]. The performance of the trial-and-error algorithms on the real robots is measured according to the distance the robot covers within ten seconds. For this, markers are attached to the robot's main body, and their walking gait is tracked. The covered distance between the damaged robot in simulation and the damaged real robot is compared. For a quadruped robot, in reality, 20 trials are necessary to learn and adapt a gait to its injuries. In simulation only 10 trials are needed. The median performance of the simulation is 1 m, and 0.9 m for the real robot [DDM19].

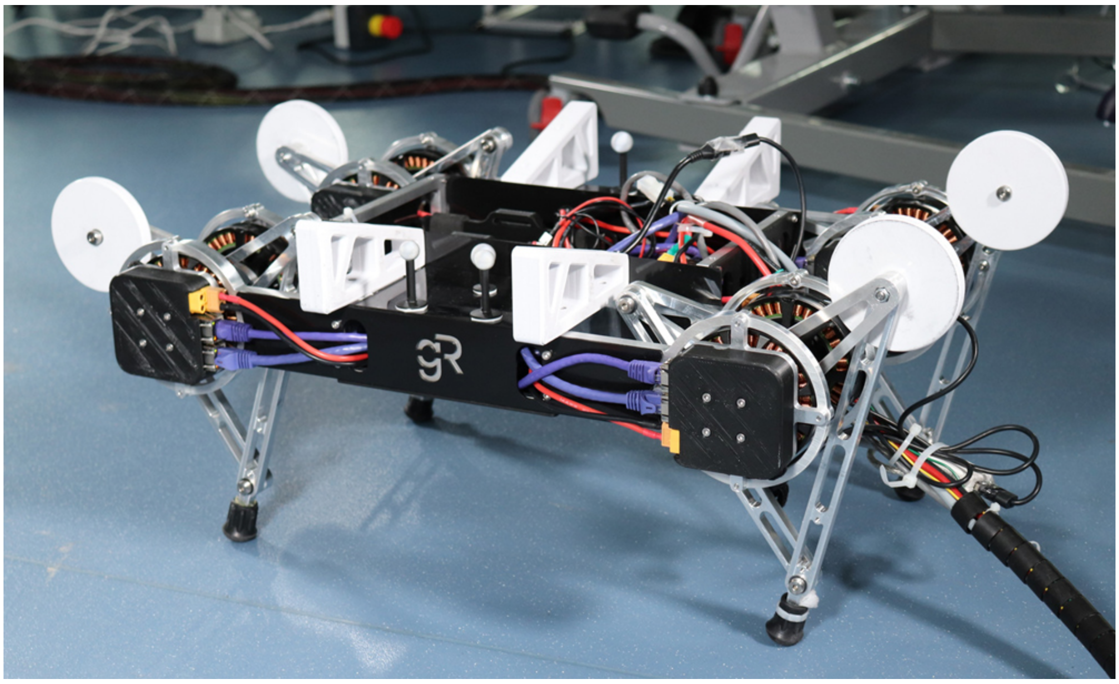


Figure 2.1: Resibot's quadruped robot with markers [res].

Kolvenbach et al. [KHB+19] work on jumping locomotion for four-legged robots on low-gravity celestial bodies such as the moon. Jumping is a useful way of locomotion and gives robots the possibility to surmount obstacles. In their work, they present the design of a 22 kg quadruped robot, with 2-DOF for each leg that performs energy-efficient jumps by exploiting conditions of lunar gravity. To mimic the moon's gravitational force, their test setup consists of a 9.51° inclined rail where the robot is mounted sideways on a sled with a ball thrust bearing. This way, the robot can translate along the rail with low friction. To observe the jumping movements the setup is placed in a 45 m^2 area, where 14 Vicon tracking cameras capture ground truth orientation data for the experiment. The tracking setup is shown in Figure 2.2. A leg of the robot consists of a four-bar parallel motion linkage and has two built-in tension springs. For flight stabilization during jumps, the robot uses a reaction wheel that corrects the robot's orientation to ensure a safe landing. The robot is controlled using the Robot Operating System (ROS). The robot is able to repeatedly execute vertical jumps of over 0.9 m. It can also perform single forward leaps of up to 1.3 m [KHB+19].

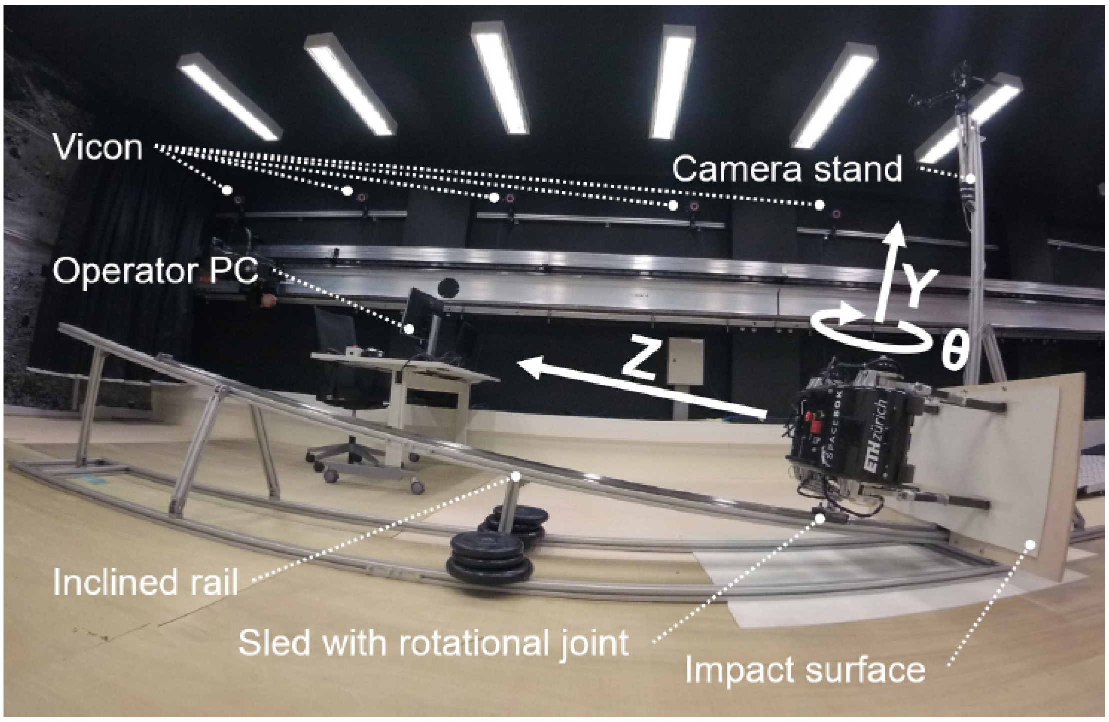


Figure 2.2: Tracking setup for the jumping experiment with lunar gravity [KHB+19].

Weinmeister et al. [WEWI15] integrate a 1-DOF spine-like structure into a small (10 cm hight, 10.5 cm width) four-legged robot. The robot and the spine-like structure are shown in Figure 2.3. With this spine-like structure, maneuverability is increased compared to robots that have rigid spines while keeping a low controlling effort. Their work is inspired by rescue dogs deployed in areas of disaster. In situations where the life of a dog is endangered, robots could jump in. But for such tasks, robots need to be versatile and easy to control. The movements of the robot, especially the turning using the spine-like structure, are observed in a test area of 4.5 m $\times$ 2.8 m with an OptiTrack motion capture system. 14 infrared cameras capture the motion at 240 Hz. The spine consists of a compliant joint that bends vertically. The joint is located between two trunk modules that have two legs attached. A leg consists of two joints. With the integrated spine, the robot manages to turn within a 0.51 m radius at 0.31 m/s and can be steered around objects within the test area [WEWI15].

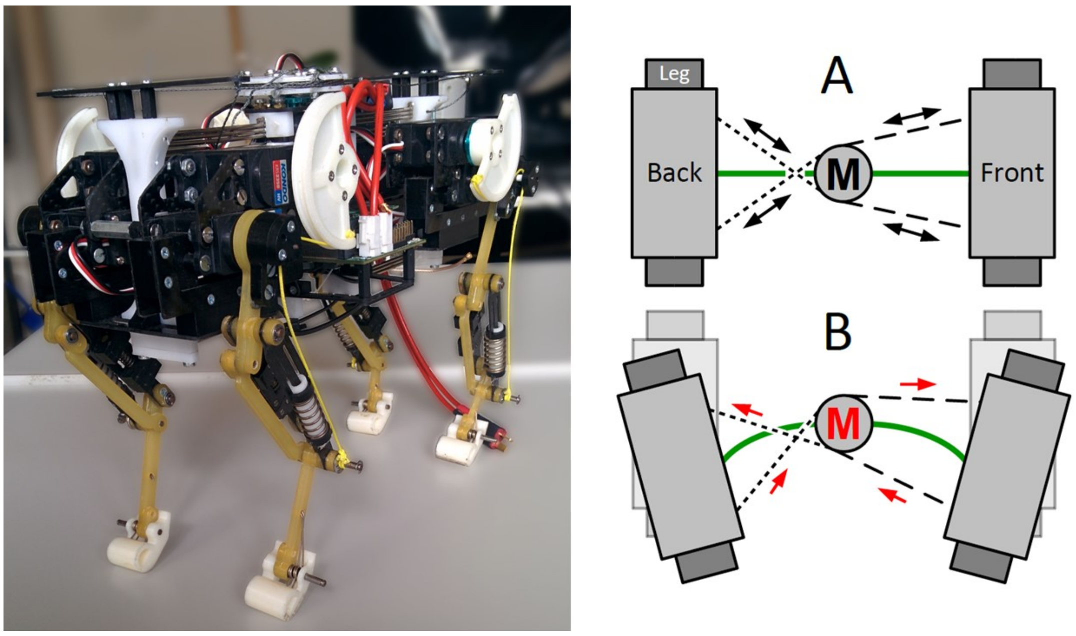


Figure 2.3: Quadruped robot and spine structure with a motor in the middle [che, WEWI15].

Shachaf et al. [SIZ19] use an OptiTrack motion capture system with 12 Prime 13 cameras for the analysis of a reconfigurable single actuator wave robot (RSAW). The robot consists of two waves that are connected to each other with a universal joint (2-DOF, yaw and pitch), to enable the robot to change direction in the plane and to lift one wave to overcome obstacles. It has a length of 73.4 cm, a width of 15 cm and a height of 8 cm and is shown in Figure 2.4. The design of the robot aims for maneuverability and the ability to crawl over different terrains and to surmount obstacles. The robot is also capable of carrying a load of 1.25 kg on its center. These properties can be useful for search and rescue operations where equipment needs to be transported. With the motion capture system, straight paths and rotation paths of the robot are tested for deviations. On average the robot deviates 3 cm per 1 m when moving straight ahead (yaw set to zero). The rotation paths are tested on floor tiles, plastic and polypropylene surfaces, with the joint's yaw set to -60 degrees. The robot is able to climb over 10 cm high obstacles, and when operated remotely manages to run over grass and dirt [SIZ19].

Chitta et al. [CVGL07] address localisation of quadruped robots in known environments, by using only proprioceptive sensors, such as accelerometers, angular rate gyros, and joint encoders. Those sensors provide a local pose of the robot and joint angles. This information, in combination with a known rough terrain, can be used to locate the robot. The method is not very accurate but it could be used to offer robustness to failure of

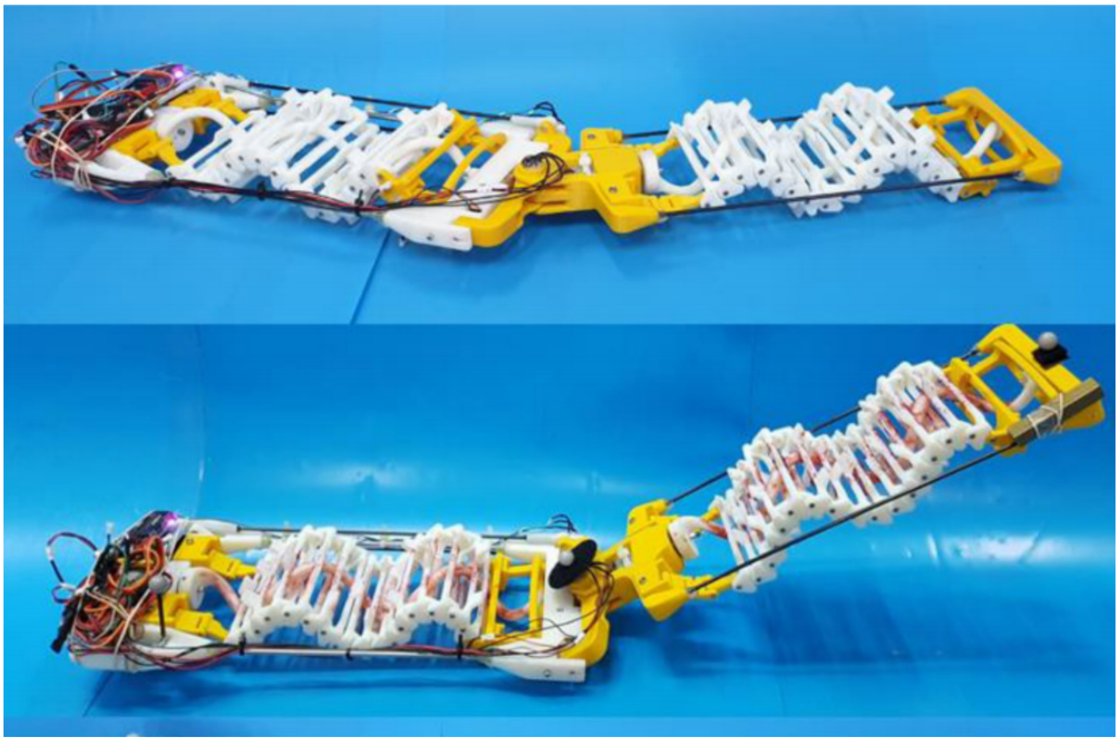


Figure 2.4: Wave robot [SIZ19].

other prior sensors like cameras or GPS. The robot they use is called *LittleDog* and is from Boston Dynamics Inc (see Figure 2.5). It is equipped with an accelerometer, a gyroscope and has single axis force sensors at the bottom of the feet. Each of the four legs has three joints. The experimental setup consists of a Vicon motion capture system with six cameras operating at 100 Hz. Several reflective markers on the robot's body are tracked by the cameras and provide position and orientation of the robot. They use 60 cm × 60 cm terrain boards to test the robot. The boards are scanned to obtain elevation maps and are located by the motion capture system with markers. When the robot is sent over the terrain boards, data from the motion capture system and the proprioceptive sensors is logged. The motion capture data is used as a reference for the true position of the robot. For the localisation estimation, particle filtering, which is a common method in robotics, is used. The localisation performance with respect to dead reckoning is improved by 45 % with the proposed approach. On flat terrain, this approach does not work, since there are no ground features that can be associated with a pose. Also, for spatially and temporally similar terrain features, trajectory estimates can differ from the actual robot location [CVGL07].

A work by Camurri et al. [CBCS15] also deals with robot localisation. They propose a real-time simultaneous localisation and mapping (SLAM) system for a quadruped robot, which combines visual 3D data and inertial data. The localisation method uses

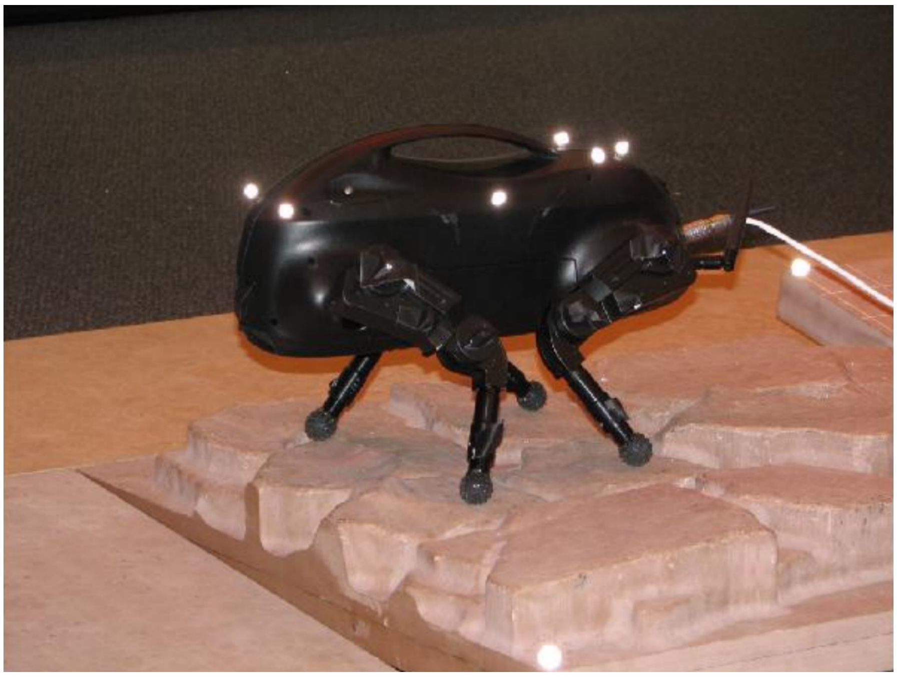


Figure 2.5: LittleDog with markers on terrain board [CVGL07].

Iterative Closest Point (ICP) registration enhanced with prior background subtraction for processing the depth images. They use an 80 kg, 1 m tall and 1 m long, hydraulic legged robot that is equipped with an RGB-D and a stereo camera (for calibration) on its head. The legs of the robot have 3-DOF each, two joints for hip and knee flexion and extension and one for hip abduction and adduction. It also has an inertial sensor attached to the body. The SLAM system is implemented onboard the robot. Within the local map surrounding the robot, it manages to keep the localisation error under 5 %. The robot is teleoperated on a flat surface of approximately 3 m^2 around different obstacles arranged on the ground. A Vicon motion capture system is used for obtaining ground truth data to compare localisation performance. The camera setup is not mentioned [CBCS15].

Abondance et al. [ATW20] present the design of a soft robotic hand prototype with four soft fingers that can manipulate objects while maintaining the grasp. The soft fingers consist of two side by side bellow actuators made of silicone with air chambers inside. They can control air pressure on both sides of the finger to actuate the fingers. The soft robotic hand is shown in Figure 2.6, opening a jar. They define three motion primitives for the held objects: translation along the x-axis and the y-axis, and rotation about the z-axis. These primitives are useful for a variety of tasks in daily life, such as moving fragile dishes or storing delicate products such as pastries in a refrigerator. They also examine a finger gait that makes continuous rotation possible for different object sizes and shapes. They demonstrate the usefulness of in-hand manipulation in real-world applications, by performing unscrewing of a jar, and orienting food (muffins and broccoli) for packaging. They use a Vicon motion capture system to track the workspace of the fingers, to get information about how fingers can be used for grasping and to understand the motion range, strength and operating pressures [ATW20].

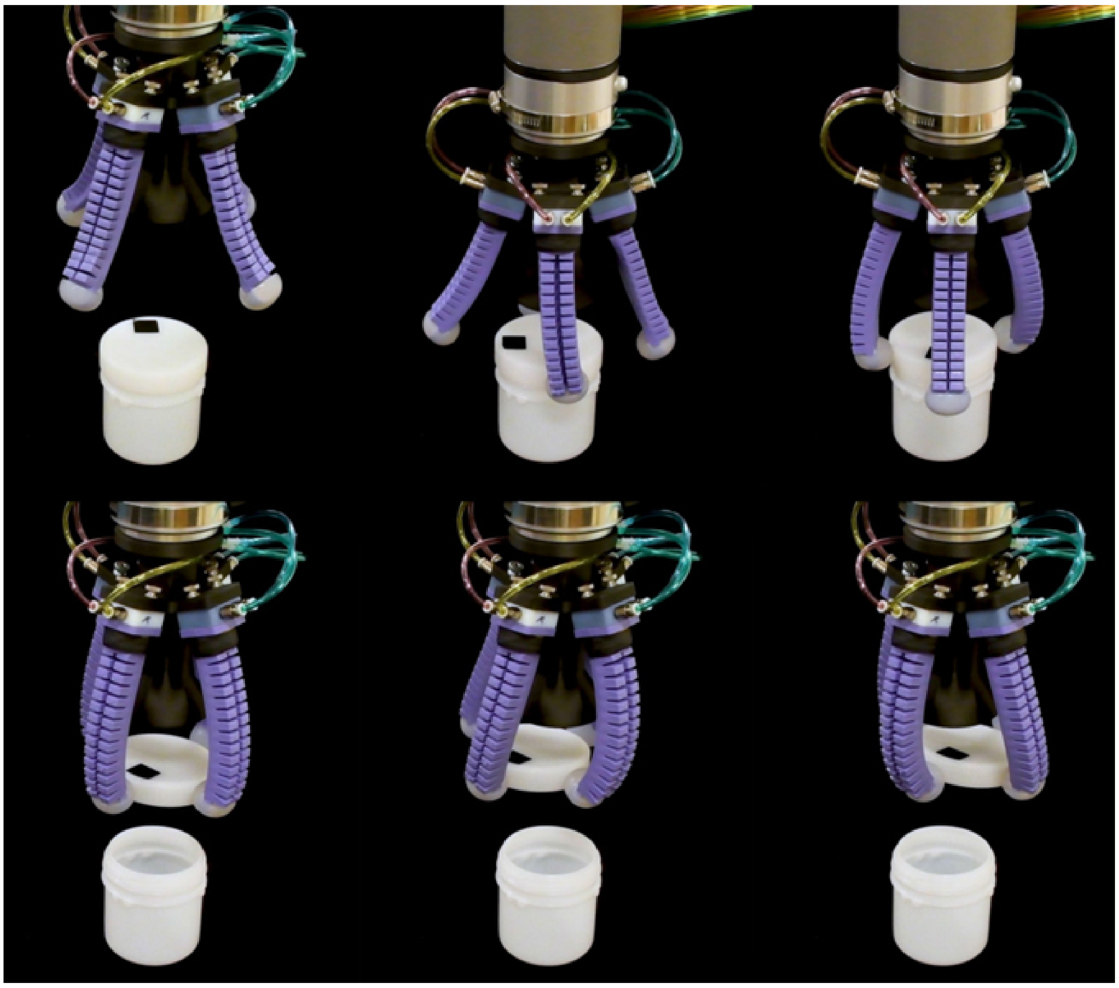


Figure 2.6: Soft robotic hand opening a jar [ATW20].

Apart from validating the movements of robots against simulations, motion capture systems are also used for steering robots. In 2013, Lexus filmed a story called 'Swarm', featuring small aircraft from KMel Robotics, also known as quadrotors [kme]. OptiTrack motion capture technology was used with 35 Prime 41 cameras. In a 60-second spot, a swarm of quadrotors flies through different locations in Vancouver at night. With the OptiTrack motion capture system used as simulated GPS, the location of the quadrotors was tracked and adjusted if they were slightly off course. This made it possible to fly complex maneuvers with many quadrotors at once [kme, vid].

Panerati et al. [PMG+19] address in their work the improvement of robustness and connectivity for multi-robot systems. In swarm robotics, communication (or swarm connectivity) is the key to effective collaboration. Failure of one robot does not need to lead to a failure of the complete system. They implement a control strategy for eight two-wheeled robots (Khepera IV by K-team). The robots have a cylindrical shape (14 cm diameter, 6 cm height). They are equipped with an onboard Linux operating system that executes the controller software. The connection between robots is modelled with an undirected graph. Every robot has a defined communication radius (coverage) of 65 cm. A robot knows its relative position to other robots in this radius. They can also access the positional information of those other robots and their direction. Large ground coverage is beneficial if such systems were used for search, rescue, or exploration missions. Panerati et al. discuss the system's performance in cases of communication or hardware failures. For the experimental setup, OptiTrack Prime 13 cameras are placed 2.3 m above the ground. The tracking area is $2\,\mathrm{m} \times 2\,\mathrm{m}$. The x- and y-position and the yaw of the robots are captured in global coordinates. This information is transformed into local coordinates and fed to the robots. Since each robot only has information of itself and its neighbours, which can be error-prone if failures occur, the OptiTrack data is used to provide correct reference information. They analyse the algebraic connectivity of the robotic network, the ground coverage (in m^2) of the robots and the robustness of the system. To increase the robustness of the system, a robot that exhibits vulnerability moves closer to its neighbours to increase the connectivity. This automatically reduces the ground coverage of the whole system, but it keeps the system connected. They show that connectivity alone is not sufficient for the design of resilient multi-robot systems, especially if a communication failure occurs [PMG+19].

Orsag et al. [KOO14] propose the design of a mobile manipulating unmanned aerial system (MM-UAS). This quadrotor aerial vehicle is equipped with 2-DOF manipulators (see Figure 2.7). Their theoretical mission plan for such robots includes the save retrieval of sensitive equipment in industrial disaster areas. Aerial systems have the advantage of flying over debris and obstacles. To show the dexterity of their robot they let it perform three tasks, an insertion task, where the robot has to plug in an emergency light, a pick-and-place task, and a valve turning task. They implement human-machine interaction to control the aerial system, using voice recognition, joystick controllers, and motion detection. They perform their experiments indoors and emulate GPS data with an OptiTrack system, consisting of 12 Flex 13 cameras that track the global position of the robot and the objects the robot has to grasp [OKBO17, KOO14].

The following works use motion capture technology for robotics in the healthcare sector. Lin et al. [LKL19] analyse in their research the physical fatigue in teleoperation of assistive robots. Teleoperation via motion mapping is an intuitive option when it comes to teaching whole-body coordination and controlling of humanoid robots. In the scope of their work, they use the Tele-robotic Intelligent Nursing Assistant (TRINA) system. In healthcare, tele-nursing robots should be able to perform patient-caring tasks, such as cleaning and food delivery, in a cluttered environment and interact with various objects, e.g. carrying blankets. They should also protect the safety of healthcare workers and

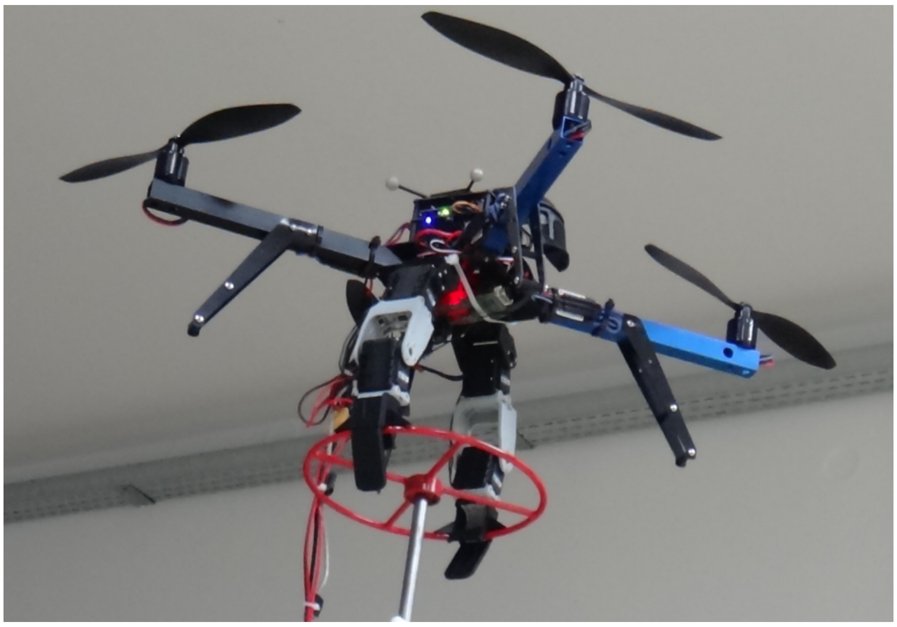


Figure 2.7: Aerial vehicle with 2-DOF manipulators turning a valve [KOO14].

patients. The robot consists of a humanoid torso with two arms and two three-fingered grippers and is positioned on an omnidirectional mobile base. On the head, it has a 180° fisheye lens, on the chest, a Microsoft Kinect 2 and two depth cameras on the wrists. The robot is depicted in Figure 2.8. A Vicon motion capture system with 10 Vero cameras is used to record the movements of the teleoperator. It captures reflective markers that are fixed to the teleoperator's torso, arms and legs. This way, locomotion, reaching and grasping of the robot, and the movement of the robot's cameras can be controlled. Physical fatigue occurs when tasks afford precise manipulation and when a posture has to be maintained for a longer period of time. Lin et al. [LKL19, LKL20] introduce an autonomous grasping function that can be initiated manually by the teleoperator. They define a Teleoperation Assistance Zone (TAZ), which consists of a bounding box around a target object. Once the teleoperator reaches into the TAZ, the robot manages to grasp the object automatically. This efficiently reduces physical workload.

Li et al. [LPB+20] present in their work the design of a fully actuated robotic assistant that is used for magnetic resonance imaging (MRI)-guided precision conformal ablation of brain tumors. For this, the robot uses a needle-based ultrasound thermal ablator probe. The robot has 8-DOF to move the ablator inside the brain through an opening in the skull (burr hole). During this procedure, the patient is lying inside the MRI scanner. 3-DOF translational motion is used to position the probe at the skull lesion. 2-DOF rotary motion is used to orient the probe and 3-DOF are used to place and orient the probe at the focus of treatment. The ablator probe has tracking coils attached to localize the probe in the MRI images. To register the robot with respect to the patient's head, a removable fiducial frame consisting of tubes of MRI-visible contrast fluids is used that

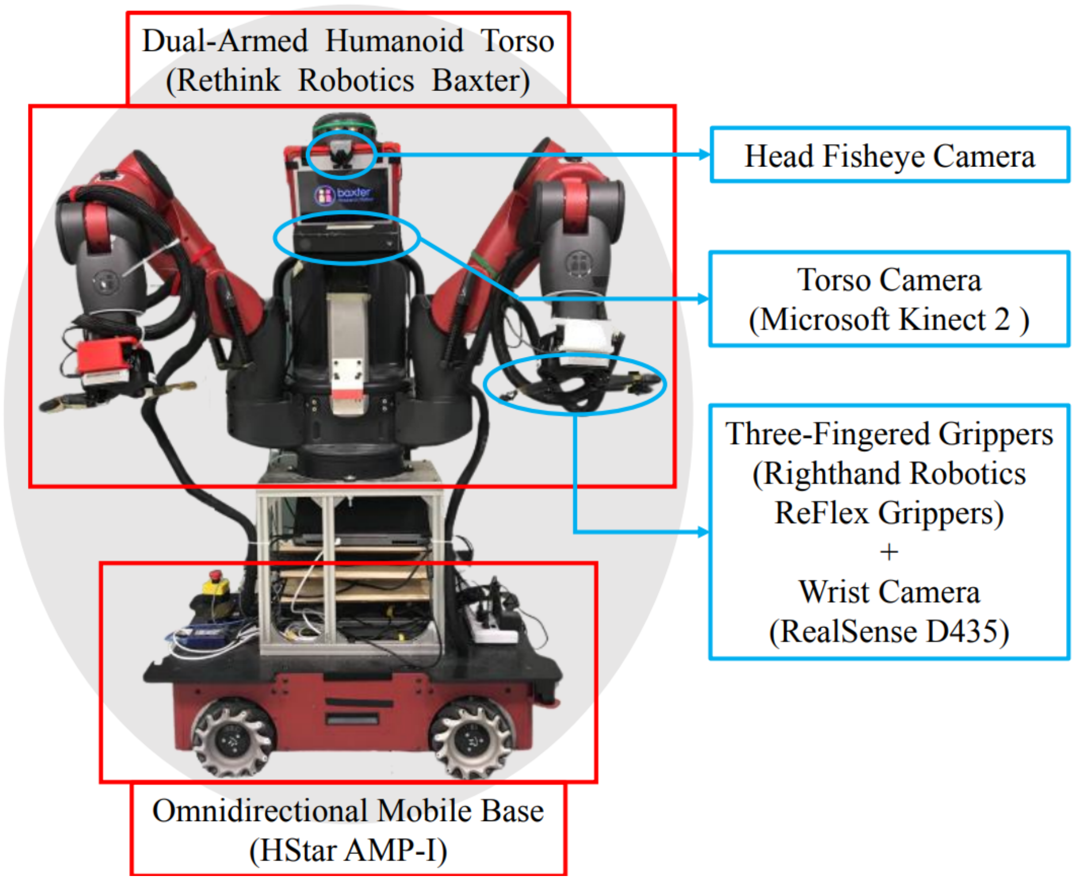


Figure 2.8: Nursing robot TRINA [LKL20].

has a known offset from the robot. Once the patient is positioned, the probe is aligned and inserted into the brain. With the help of the MRI images, the correct position of the probe is confirmed and the treatment starts. With an OptiTrack system with six Flex 13 cameras (accuracy 0.2 mm), they test the probe positioning accuracy of the robot. This is done in free space outside the MRI scanner. A tracking frame with markers is attached to the probe to measure the precision of the needle tip. An image of the setup with the robot is shown in Figure 2.9. The positioning error of the tip (distance from actual tip position to desired position) is 1.11 ± 0.42 mm, the insertion angle error (difference between actual needle insertion angle and planned insertion angle) is 1.19 ± 0.52 degrees. So far, they have tested their system with MRI phantom studies and with cadaver studies [LPB+20].

Strydom et al. [SBW+19] present the design of a leg manipulator robot for robot-assisted orthopaedic surgery, such as knee arthroscopy. For knee replacements without robotic assistance, surgeons manually align femur and tibia with varying accuracy depending on experience. The manipulator is meant to improve spatial accuracy and helps to

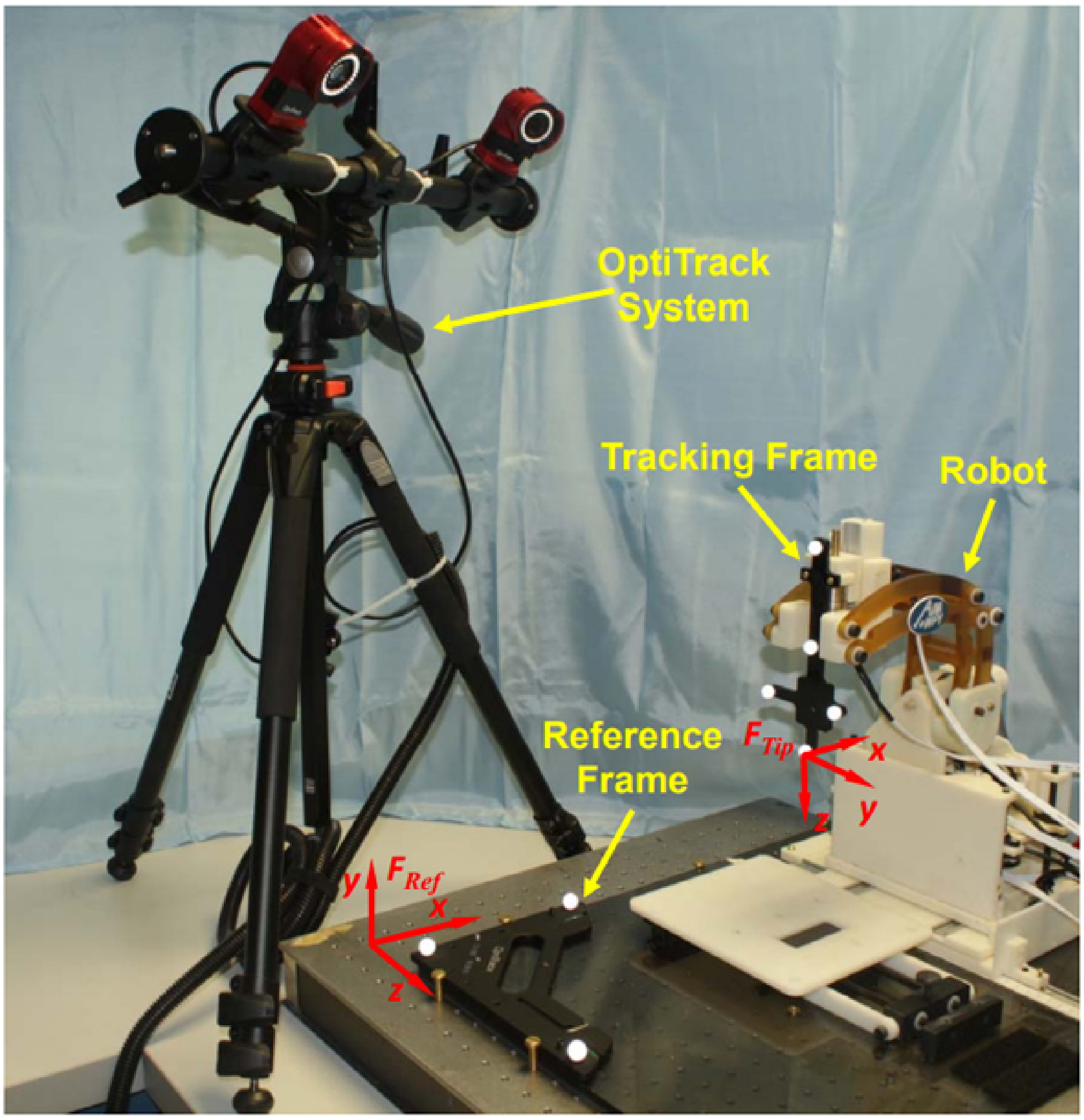


Figure 2.9: Motion capture setup with robot for brain tumor ablation [LPB+20].

reduce iatrogenic damage. They use ten OptiTrack cameras that are placed on a 3 m × 3 m × 2.5 m structure around the operating theatre to track positions on the patient's leg, the manipulator robot, and surgical instruments. They also discuss a novel design of rigid bodies with reflective markers such that marker visibility is maximised and they are not damaged during an arthroscopy. They attach rigid bodies on femur and tibia and with additional computed tomography (CT) scans they calculate relative positions inside the leg. This way, they can compute the pose and joint angles of the leg. The patient's leg is attached to the manipulator robot with a foot interface to adjust the knee and ankle angles automatically (see Figure 2.10). For knee arthroscopy, it is required to know exactly where the knee joint gap is to pass through the 4 mm arthroscope, otherwise, unintended damage can happen, when this gap is overestimated. The leg manipulator in combination with CT scans and the tracking data form a system that provides real-time anatomical measurements during surgery. The positional accuracy of a point in the leg is dependent on the accuracy of the CT scan, on average the accuracy is 0.75 mm. This is negligible for computing joint angles [SBW+19].

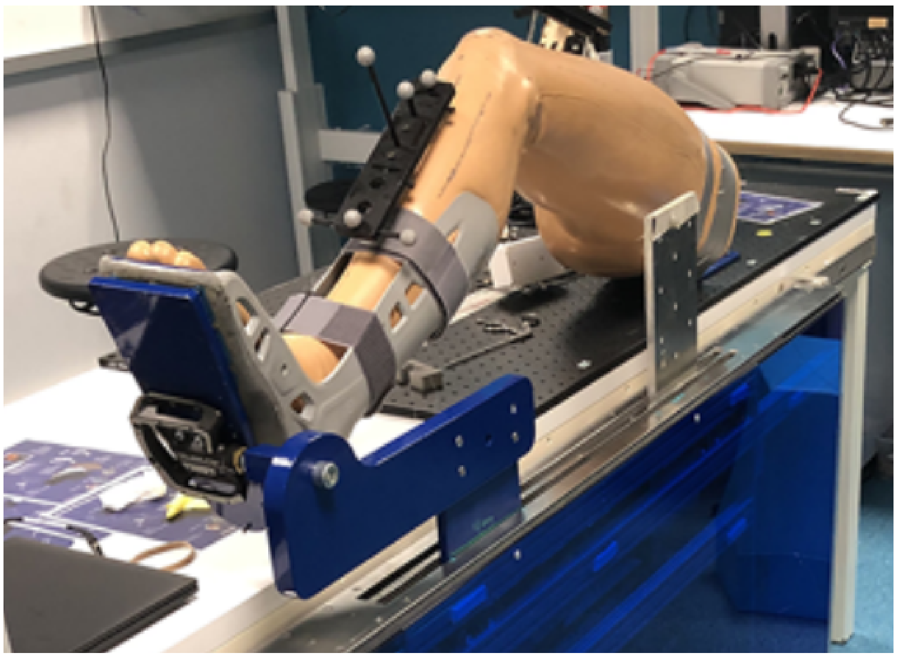


Figure 2.10: Leg manipulator robot [SBW+19].

The various works presented above give an insight into how optical motion capture systems are used in robotics research. Systems from OptiTrack and Vicon are commonly used, and both manufacturers achieve comparable performance regarding their cameras, but OptiTrack tends to be less expensive. From the presented papers, eight use an OptiTrack and five a Vicon system. The data gathered by the motion capture systems are used to validate the movements of robots against simulations, or as ground truth for robot localisation with proprioceptive sensors. Motion capture supports the steering of multi-robot systems and helps to track the workspace of robot manipulators. In the healthcare sector, motion capture measures the precision of robot-assisted surgery, and also enables tele-nursing.
In this work, the OptiTrack motion capture system is used to gather position and joint angles from a robot to compare them with simulated data. In the works discussed above, usually, only the robot's body position is tracked because the focus lies rather on how far a robot runs, or how high it jumps. For this work, tracking the joint angles of the robot's legs could help to better understand the impact that physical forces have on the

movement of legs.
The motion capture setups and the number of cameras depend on available space, purpose of tracking, and size of the robot. The majority of setups in the related work uses more than ten cameras and most tracking areas are larger than 4 m^2. For this work, a tracking area of approximately 1 m^2 is available, which is sufficient for the small robot that is used. Six cameras are used for capturing, which is enough to equally cover the tracking area from all sides and acceptable in terms of price.

2.2 Simulation in Robotics Research

Simulations are an essential part when working with robots. They allow researchers to test their ideas more easily instead of using real robots [MC17]. In this section simulation tools are listed that are used in robotics.

Simulink Simscape Multibody is the simulation environment used for this work. It is used with MATLAB and provides multibody simulation for 3D mechanical systems like robots among others. Simulink is a graphical programming language that uses blocks to represent robot bodies, joints, forces, and constraints. Robot models can be imported from computer-aided design (CAD) software. A simple 3D animation provides visualisation of the system dynamics [simc].

MATLAB Robotics System Toolbox offers algorithms and tools for simulation design and testing of manipulators, mobile and humanoid robots. It provides forward and inverse kinematics and dynamics, collision checking and trajectory generation. It also supports path planning and following, localization and motion control and a library of robots that can be used. The toolbox can be connected to the Gazebo simulator [tob].

Gazebo is an open-source robotics simulator by the Open-Source Robotics Foundation, Inc (OSRF). It integrates the Robot Operating System (ROS) and provides various robot, object, and simulation models. With the graphics rendering engine OGRE it offers high-quality textures, shadows, and lighting. Further, physics engines such as ODE, Simbody, Bullet, and DART are supported. Supported model file formats are the native file format SDF (Simulation Description format) and the URDF (Unified Robot Description Format) used by ROS [gaz].

CoppeliaSim, former V-REP, is a robot simulator with an integrated development environment. It uses a distributed control architecture where each robot is individually controlled by ROS, a plugin or embedded script, a remote client or a custom program. This makes it particularly useful for multi-robot systems. Controller programming is supported for C/C++, Python, MATLAB, Octave, Java, and Lua [cop].

ARGoS stands for Autonomous Robots Go Swarming and is an open-source parallel multi-engine simulator specifically designed for swarm robotics. In ARGoS a simulation can be partitioned and different physics engines can be used in parallel. The architecture is multi-threaded and modular, making it easy to add custom features and allocate computational resources to get accuracy where necessary without impacting speed [PTO+12].

MuJoCo is short for Multi-Joint dynamics with Contact. It is a physics engine that has been developed for research in robotics, but also biomechanics, graphics, and animation. MuJoCo's purpose is model-based optimisation, especially optimisation via contacts. It features, among others, simulation in generalized coordinates, inverse dynamics, cloth, particle and soft objects simulations, various solvers and integrators and a GUI with 3D visualization in OpenGL. Supported model file formats are the native MJCF file format and URDF [muja].

Webots is an open-source robot simulator, with a development environment for modelling, programming, and simulating robots. It comes with a large asset library containing robots, actuators, sensors, objects, and materials. For simulation, the physics engine ODE is used [web].

Open Dynamics Engine (ODE) is a high-performance open-source library with a C/C++ API offering simulation of rigid body dynamics. It provides advanced joint types and has integrated collision detection with friction. ODE is suitable for the simulation of vehicles and objects and creatures in virtual reality environments. Its applications are in computer games and simulation tools like Gazebo [ode].

Bullet Physics is an open-source library for rigid body dynamics and collision detection. It is written in C++ and provides Python bindings as well. It supports model file formats from MuJoCo (MJCF), URDF, and SDF. It is used for physics simulations in robotics and machine learning, for games, virtual reality, and visual effects [bul].

DART stands for Dynamic Animation and Robotics Toolkit and is a C++ library. It provides kinematic and dynamic applications for robotics and animation. Collision detection from FCL (Flexible Collision Library), Bullet, and ODE is supported. The simulator is not viewed as a black box, but users can modify internal kinematic and dynamic properties. It is integrated into Gazebo and supports URDF and SDF model formats [LGH+18].

Simbody is an open-source multibody physics API and was designed for biomedical engineering. It is suitable for coarse-grained molecule and internal coordinates modelling, but also for the creation of mechanical models like skeletons, neuromuscular models or any other model that consists of rigid bodies and joints, where forces and constraints are needed. So, it can also be used for robotics or animation. It is written in C++. Rigid body mechanics are provided by an advanced Featherstone-style formulation and results for any set of n generalized coordinates are computed in O(n) time [simb].

SD/FAST is a physically-based simulation for mechanical systems. It takes the description of an articulated rigid body system and derives its nonlinear equations of motion. Contact dynamics are not computed. C or Fortran source code is generated from the equations, which can be used in any simulation environment. SD/FAST provides extremely fast simulations and real time execution is possible for many systems [sdf].

PhysX SDK is an open-source multithreaded physics solution by NVIDIA. Despite being a game engine, it is also used for robotics and artificial intelligence. Since version 4.0, PhysX has a new Temporal Gauss-Seidel Solver (TGS) that makes jointed bodies more robust. With TGS, constraints are recomputed dynamically each iteration, based on the relative motion of bodies [phy].

In 2015, Erez et al., who are the developers of MuJoCo, compared in a paper [ETT15], and also on the MuJoCo website [mujb], their physics engine with Bullet, Havoc (game engine), ODE and PhysX. With their performance tests, they focused on numerically challenging tasks typical for robotics in contrast to multi-body dynamics and gaming. For testing, they created four systems with different challenges. On a 35-DOF robotic arm that is grasping a capsule they test contacts and dynamic simulation. Then, they let a 25-DOF humanoid model fall on the floor and let it wiggle due to sinusoidal torques applied to the joints. There they measured speed and accuracy. With a planar kinematic chain with 5-DOF and frictionless hinge joints, they tested the energy and momentum conservation. In the last experiment, they let 27 randomly oriented capsules fall on the floor. With this 162-DOF system, they wanted to measure performance.
They evaluated the speed-accuracy trade-off for the systems for different timesteps. They concluded that there was no engine that performed uniformly better than the others. For constrained systems important in robotics, MuJoCo was the fastest and most accurate one. It could perform stable grasping at larger timesteps. When dealing with many disconnected bodies, as in the capsules experiment, MuJoCo was the slowest with respect to raw CPU time and ODE the fastest. PhysX and Havoc made the most compromises regarding physical accuracy. This was mainly because those engines are optimized for gaming and focus more on stability than accuracy [ETT15, mujb].

In 2018, Pitonakova et al. [PGPW18] provided a feature and performance comparison of the three robot simulators V-REP(now CoppeliaSim), Gazebo, and ARGoS. With each simulator, they performed two types of benchmark tests: For the GUI benchmark test, they ran a simulation of robots that move in a straight line and avoid obstacles in real-time. Simulation time was one minute, and the user interfaces were visible during the simulation. For the headless benchmark test, they ran the same simulation for five minutes from the command line without user interface. They had two simulation environments, a small scene with robots on a large 2D plane, and a large scene with an industrial building model that consisted of 41,6000 vertices. ARGoS could not import that model, so 5,200 boxes corresponding to the number of vertices of the building model were placed randomly in the scene. They used 1, 5, 10, and 50 simple wheeled or flying robots in both experiments.
They concluded that V-REP was the most resource-hungry and complex simulator. It was suitable for high-precision robot modelling with only a few robots and offers a variety of features, such as physics engines, a model library, mesh manipulation, and optimisation. ARGoS was better for swarm robotic tasks, but compared to V-REP, ARGoS traded-off environment, robots, and physics complexity for better performance. ARGoS lacked the ability to import 3D meshes. Gazebo was between V-REP and ARGoS. Regarding

features, it was closer to V-REP, but robot models were similarly simple as in ARGoS. Gazebo performed better than ARGoS in the larger simulation scene, but usability in Gazebo was poor and 3D meshes could not be edited or optimized.

In 2019, de Melo et al. [PGL+19] compared V-REP and Gazebo with Unity in terms of usability for robotic experiments and not performance. With all three simulators, a group of users had to install the software, create a simple scene, insert a robot, and control the robot. They rated the software with the System Usability Scale (SUS) questionnaire, where effectiveness, efficiency and satisfaction are covered. Results showed that V-REP and Unity provided a lot of freedom for scene editing and had a good user interface, whereas Gazebo, as was also mentioned by Pitonakova et al. [PGPW18], had a poor interface and was more limited. For robotics projects, V-REP and Gazebo stood out, which is not surprising because Unity is made for gaming, which is reflected in the high graphics quality and the less accurate physics, using the PhysX engine. V-REP offers more native features than Gazebo. In Gazebo, they need to be integrated via plugins. However, Gazebo is open-source in contrast to V-REP and provides a large documentation [PGL+19].

No simulator nor physics engine is perfect, and it always depends what compromises one is willing to make, if one prefers a ready to use GUI, or codes its simulation from scratch, if one values accuracy more than robustness or speed or vice versa, or if the familiarity of a system makes one choose a system over another one. For this work, Simulink was chosen because the MATLAB environment was already familiar, and many simulation techniques are ready to use, which made a faster start possible.

CHAPTER 3

Robot Assembly and Programming

In this chapter, the robot is described that is used in the scope of this work. The robot's structure and relevant components such as the motors and the robot controller are introduced. The controller's firmware is programmed to function in combination with a robot software that is used to create task programs for the robot. Later, it is required that the robot can execute motion that has been computed by a simulation. This simulation data needs to be loaded onto the controller. With the initial controller firmware, this is not possible. So, in this chapter, also the reprogramming of the controller firmware is described such that the controller can accept simulation data. Further, it is explained how the simulation data is preprocessed to bring it in a convenient format for the controller. Most of the information in the following chapter, regarding the robot's specifications, is taken from the ROBOTIS e-manual [fir, cm5, dyn, zig].

3.1 Robot Overview

The robot is one of several robots that can be built with the ROBOTIS *Premium* kit [robc], namely, wheeled robots, two-legged, four-legged, and six-legged robots with 1-DOF up to 18-DOF. This robot kit was chosen because it offers an instructed assembly of the robots, and preprogrammed robot behaviour that can be used straight away. The associated software *RoboPlus* [roba] includes applications for designing robot motions and creating robot behaviour that can be controlled remotely. Further, the servomotors driving the robot can be examined with respect to their operating temperature, position, input voltage, and other motor specific parameters.
The robot of interest is a four-legged robot in the shape of a little dog or puppy that has 15-DOF. It is shown in Figure 3.1. The puppy consists of 16 rigid parts (or rigid bodies) that are connected with 15 AX-12A Dynamixel [dyn] servomotors, which serve as joints.

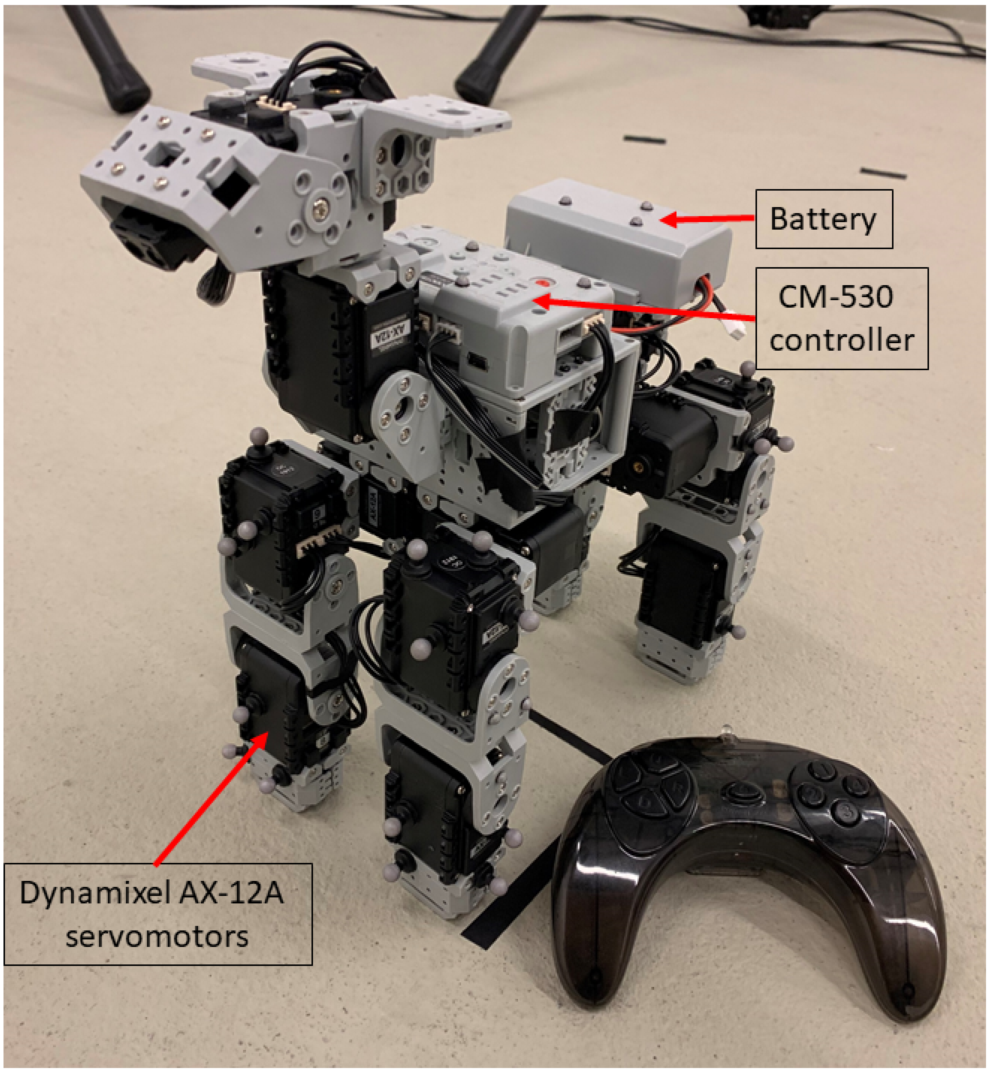


Figure 3.1: Assembled *puppy* robot with 15-DOF and remote control for wireless communication.

The robot's torso consists of two rigid bodies. Each leg has three rigid bodies. Neck and head are also two rigid bodies that can be moved separately. The rigid bodies and corresponding data are named (and abbreviated) in alphabetical order. When referring to rigid bodies, the abbreviation with capital letters is used. Rigid body data (e.g. joint angles) are abbreviated with three letters: body (bdy), back-left-bottom (blb or BLbtm), back-left-middle (blm or BLmid), back-left-top (blt or BLtop), back-right-bottom (brb or BRbtm), back-right-middle (brm or BRmid), back-right-top (brt or BRtop), front-left-bottom (flb or FLbtm), front-left-middle (flm or FLmid), front-left-top (flt or FLtop), front-right-bottom (frb or FRbtm), front-right-middle (frm or FRmid), front-right-top (frt or FRtop), head, neck, tail. These abbreviations are used in the following chapters

when referring to the robot's rigid bodies or joint angle data.
In Figure 3.1, the Dynamixel servomotors are indicated as well as the controller fixed to the body. The Dynamixel motors are connected to the controller and among each other. The controller is used for switching the robot on and off, and for executing the robot's task program. The controller is powered by a battery, which is practical when tracking the robot, since no cables interfere with the robot's motions. One of two Zig-110A [zig] ZigBee modules is connected to the controller, the other module is built in a remote control. They provide wireless serial communication between the robot controller and the remote control. A person controlling the robot can do this from outside the tracking area and does not interfere with the tracking by potentially restricting the cameras' views with their body.

3.2 Controller Specifications and RoboPlus

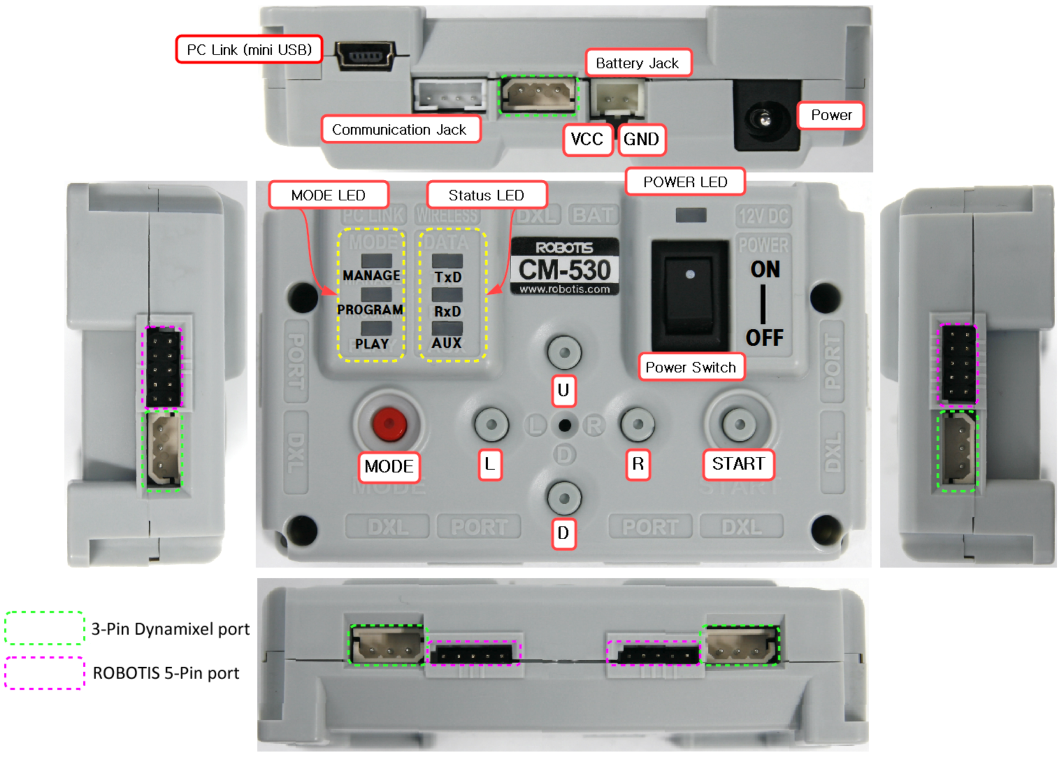


Figure 3.2: Robot controller CM-530 [cm5].

The Dynamixel servomotors are controlled with the CM-530 controller. The CM-530 includes an ARM Cortex STM32F103RE microprocessor and can be programmed with the RoboPlus software via USB. The voltage supply ranges from 6 V - 15 V. 11.1 V is recommended for the use with the Dynamixels. In Figure 3.2 the CM-530 is shown from the top and the four sides. The five ports on the sides, outlined in green, are three-pin Dynamixel ports. With these five ports, Dynamixels are connected in series (daisy chain) to the controller. The ports outlined in violet are used to connect other peripheral devices, such as IR or distance measurement sensors. The communication jack is needed for wireless communication using the ZigBee module. When looking at the top side of the CM-530, apart from the power switch, there are buttons and LEDs that are used in combination with the RoboPlus software to program additional user input when the robot is operated or to inform the user when data is sent or received. They are not described any further, since their functionality is erased, once the controller firmware is reprogrammed [cm5].

The RoboPlus software is designed for educational purposes, which makes usage intuitive. A robot is controlled by a *task* program (.tsk), which defines what the robot does when certain buttons are pressed for example. The task program references a *motion* file (.mtn), which consists of positions and speed data for Dynamixels that make up a movement for the robot. In RoboPlus, writing a task program mainly consists of selecting predefined statements and variables per mouse click in a graphical user interface and inserting them in a selected line in the program code. For experienced programmers, coding in this environment is impractical and slow. The process of creating motion is similarly cumbersome. Each robot pose has to be defined manually, adjusting the angles for the servomotors separately. This is time-consuming, and it is also not productive to load simulated data that potentially consists of hundreds of poses, on the robot in this way. A more practical solution is to directly provide position and speed data for the Dynamixels without having to use RoboPlus. The convenience of RoboPlus is that the robot can be programmed at a higher level, hardware-specific information about the Dynamixels or the controller is not required. To control the robot without RoboPlus, it is important to know how Dynamixels work. This is explained in the following section.

3.3 Dynamixel Specifications

The 15 AX-12A Dynamixel servomotors are responsible for the actuation of the robot. A Dynamixel has two modes of operation, joint mode and wheel mode. In wheel model, the Dynamixel can turn endlessly, clockwise and counter-clockwise. In joint mode, only rotation between 0 and 300 degrees is possible. For the puppy robot, the Dynamixels are in joint mode. The gear ratio of the servomotor is 254:1 (driven gear:driving gear). The stall torque is 1.5 Nm at 12 V. Stall torque is the maximum static torque produced by a motor that has zero rotational acceleration. Input voltage is between 9.0 V and 12.0 V [dyn].

The Dynamixels use half-duplex asynchronous serial communication with 8 bit, 1 stop bit, and no parity. The baud rate is between 7843 bits per second (bps) and 1 Mbps. The Dynamixels and the controller send and receive data from each other in form of packets. There are two kinds of packets: instruction packets that are sent by the controller and status packets that are sent by the Dynamixels to the controller in response. Every

Dynamixel has a unique ID through which it is addressed by the controller. Dynamixels are connected to the controller with 3-Pin cables. Pin 1 provides the ground (GND), Pin 2 the positive supply voltage (VDD), and Pin 3 the data packets. The Dynamixels can be connected in series (daisy chain). They use a half-duplex Universal Asynchronous Receiver Transmitter (UART) interface. This means they only have one pin that can either receive (RxD) or transmit (TxD) data.
An instruction packet that is sent by the controller consist of the following information:

Header 1	Header 2	Packet ID	Length	Instruction	Param 1 - N	Checksum

Both headers have a value of 0xFF and indicate the start of a packet. The packet ID defines the ID of the Dynamixel that should receive the packet. IDs range from 0 - 253 (0x00 - 0xFD) ID 254 is the broadcast ID that makes all Dynamixels execute the packet. The byte length of the packet is composed of instruction, number of parameters and the checksum (length = number of parameters + 2). The instruction field defines the type of instruction that the Dynamixel should execute. Depending on the instruction, additional data might be required, which is added in the parameter fields (Param 1 - N). An instruction relevant in this context is the *Sync Write* instruction (0x83). A packet with this instruction writes data on the same address with the same length for multiple Dynamixels at once. The checksum field checks if the packet was corrupted during communication. The checksum is computed as follows: $checksum = \sim (ID + length + instruction + param1 + \ldots paramN)$, where $\sim$ is the ones complement operator. The lower bytes are used if the checksum exceeds 0xFF.
The Dynamixels receive the instruction packet and send back a status packet. The status packet has the same structure as the instruction packet, but instead of the instruction field there is an error field. Errors occur, for instance, if undefined instructions are sent, the checksum is incorrect, the operating temperature of the Dynamixel is too high, or the position of the Dynamixel is out of range in joint mode [dxl].

The Dynamixel has an internal control table. The control table is a data structure with multiple fields. The size of a field is either one or two bytes. Each field has a unique address. When sending *Read* instruction packets with a valid address as a parameter, the Dynamixel sends back a status packet with the requested data. Likewise, the Dynamixel can be controlled by sending a *Write* instruction packet with a desired goal position as a parameter to the address where the goal position field is located in the control table. Some fields can be read-only (R), i.e. they are used for monitoring purposes. Such fields contain, for instance, the present temperature, the voltage, or the current position of the Dynamixel. Other fields also have write access (RW), i.e. they are used for controlling the device. Two of those fields are described in more detail because they are essential for controlling the Dynamixels directly without using RoboPlus.

Goal Position: The goal position defines the angle destination of the Dynamixel motor. Values between 0 and 1023 are available. 0 corresponds to 0 degrees and 1023 to 300 degrees. The remaining 60 degrees are only possible if the Dynamixel

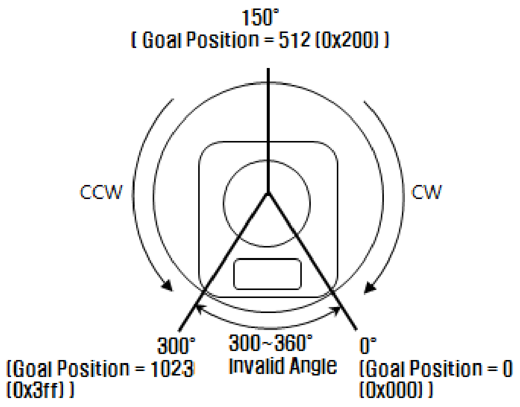


Figure 3.3: Dynamixel AX-12A servomotor and values of goal positions in joint mode (front view of Dynamixel). If the robot is in a neutral pose, all Dynamixels are at 512 [dyn, axi].

is in wheel mode, not in joint mode. The angle unit is 0.29 degrees. In Figure 3.3, the goal positions are outlined. 150 degrees (512) is the default (neutral) position of the Dynamixel. The goal position value is located at address 30 and 31 in the control table [dyn] and has a size of two bytes.

Moving Speed: The moving speed defines the velocity to a given goal position. Again values between 0 and 1023 are available. The moving speed is given in revolutions per minute (rpm). A value of 0 means the motor uses the maximum rpm it can generate, without controlling the speed. With a value of 1023, the motor moves at 114 rpm. The moving speed value is situated at address 32 and 33, also requiring two bytes [dyn].

The Dynamixel AX-12A can only be actuated by providing a goal position and the corresponding moving speed. It is not possible to provide torque for actuation.

3.4 Programming of Controller Firmware

Knowing the basics of how to actuate the controller on a more embedded level, the firmware of the controller can be rewritten. Once this is done, RoboPlus cannot be used anymore, but it is possible to restore the old firmware using an application called RoboPlus Manager that is used for managing controller and Dynamixels.
A new firmware file is created using Embedded C. Embedded C extends the C language for programming embedded systems such as microprocessors. The Eclipse IDE for C/C++ Developers Version 2019-12(4.14.0) is used as recommended by ROBOTIS. ROBOTIS also provides example projects for Eclipse to facilitate the firmware programming. Those example projects showcase simple functionalities, e.g. how to press a button, turn LEDs on and off, establish a connection with the remote control, and how to control Dynamixels

[fir]. The projects use the STM32F10x Standard Peripherals Library V2.0.3. The library offers access to all functions of the STM32F10x microcontroller series. It provides modules for external communication (periphery) through input/output ports, interfaces, LEDs, or switches [stm]. The projects already define all the necessary input and output ports and allocate the required memory on the microprocessor. They define the baud rates for the devices and set the system clock. Also, the sending and receiving of packets to and from Dynamixels are already abstracted into functions, which makes the reprogramming of the controller manageable, even when no previous knowledge of microcontroller programming is available [fir].
The example projects are used as a template to reprogram the firmware. With the new firmware, it is possible to control the robot remotely with data, e.g. computed by a simulation. For this, the data needs to be brought into a format that can be used by the Dynamixels.

3.4.1 Providing Actuation Data for the Dynamixel

Actuation data for the Dynamixels, coming from a simulation, consist of joint angles (degrees) at a given time point (seconds). This is also referred to as a *pose*. This data is contained in a matrix of size (#time points × #joints). The goal position of the Dynamixel requires input values between zero and 1023. If the robot is standing in a neutral pose, the Dynamixel's goal position is at 512, or 150 degrees. In the simulation, the neutral pose is defined for every joint to be zero degrees. The conversion from simulation angles to Dynamixel goal positions is as follows:

$$goalPos = 512 + round(simData \cdot (1023/300)) \tag{3.1}$$

goalPos and *simData* are (#time points × #joints)-matrices, comprising robot poses at a given time step. Using (1023/300) as a conversion factor and 512 as offset brings the angles into the range of $[0, 1023]$. The simulation assures that angles are always in that range and do not have values that are physically unreachable by the robot. Since the goal position only consists of integer values, the converted angles are rounded to the nearest integer.
Apart from the goal position, the Dynamixels also need a moving speed to that goal position. From the angle data and the time points, the degrees per second (deg/s) between poses are computed. They are then converted to rpm and brought in the range of [0,1023] as required by the Dynamixels.

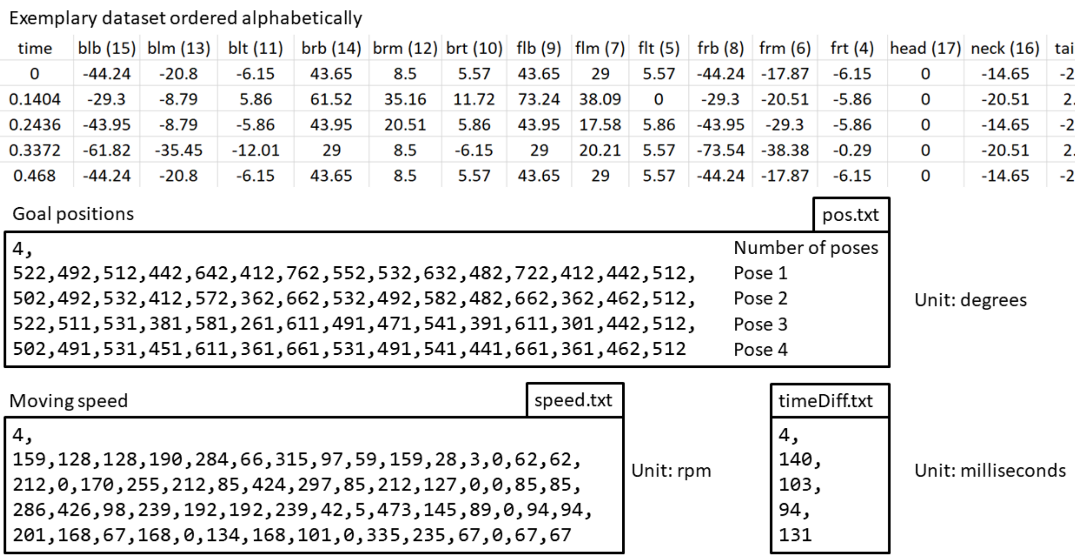
$$dps = diff(simData)/diff(time) \tag{3.2}$$

$$rpm = abs(dps \cdot (1/6)) \tag{3.3}$$

$$movingSpeed = rpm \cdot (1023/114) \tag{3.4}$$

dps denotes the deg/s, *rpm* the revolutions per minute and $diff()$ computes the difference between the time points and the angles. *time* is a vector of time points. The conversion factor between deg/s and rpm is 1/6. The absolute value, $abs()$, is used to avoid negative

velocities. The conversion factor to obtain the appropriate Dynamixel values is 1023/114. Further, for the microcontroller, the difference between the time points is required, so it knows, how much time it has to give the Dynamixels to execute their motion. The time difference is converted from seconds to milliseconds.
Computed goal position, moving speed, and time difference are then saved to text files (pos.txt, speed.txt, timeDiff.txt) that are later written into arrays so they can be processed by the microcontroller.
In Figure 3.4, an exemplary dataset is shown that is converted for use with the Dynamixels. This dataset is not the result of a simulation, but actually from ROBOTIS. It is a forward walking motion specified for the puppy robot. The dataset is used throughout this work for tracking and as input for simulations. The exemplary dataset, as well as all datasets that are used by a simulation, are sorted alphabetically according to the rigid body names. The number in parentheses next to the abbreviated rigid body name defines the ID of the Dynamixel that is actuating the rigid body. Below the ROBOTIS dataset is the converted goal position and moving speed data, as well as the time difference between the poses. The converted data has only four poses while the original data has five. This is because the first and last pose are the same. So, when computing the differences between the angles to get the moving speed, the moving speed between the last and the first pose is computed too. This makes it possible to repeat the four poses continuously. The first value in pos.txt, speed.txt and timeDiff.txt indicates the number of poses.


Exemplary dataset ordered alphabetically

time	blb (15)	blm (13)	blt (11)	brb (14)	brm (12)	brt (10)	flb (9)	flm (7)	flt (5)	frb (8)	frm (6)	frt (4)	head (17)	neck (16)	tail (1)
0	-44.24	-20.8	-6.15	43.65	8.5	5.57	43.65	29	5.57	-44.24	-17.87	-6.15	0	-14.65	-2.93
0.1404	-29.3	-8.79	5.86	61.52	35.16	11.72	73.24	38.09	0	-29.3	-20.51	-5.86	0	-20.51	2.93
0.2436	-43.95	-8.79	-5.86	43.95	20.51	5.86	43.95	17.58	5.86	-43.95	-29.3	-5.86	0	-14.65	-2.93
0.3372	-61.82	-35.45	-12.01	29	8.5	-6.15	29	20.21	5.57	-73.54	-38.38	-0.29	0	-20.51	2.93
0.468	-44.24	-20.8	-6.15	43.65	8.5	5.57	43.65	29	5.57	-44.24	-17.87	-6.15	0	-14.65	-2.93

Goal positions pos.txt

```
4,                                                            Number of poses
522,492,512,442,642,412,762,552,532,632,482,722,412,442,512,  Pose 1
502,492,532,412,572,362,662,532,492,582,482,662,362,462,512,  Pose 2
522,511,531,381,581,261,611,491,471,541,391,611,301,442,512,  Pose 3
502,491,531,451,611,361,661,531,491,541,441,661,361,462,512   Pose 4
```

Unit: degrees

Moving speed speed.txt

```
4,
159,128,128,190,284,66,315,97,59,159,28,3,0,62,62,
212,0,170,255,212,85,424,297,85,212,127,0,0,85,85,
286,426,98,239,192,192,239,42,5,473,145,89,0,94,94,
201,168,67,168,0,134,168,101,0,335,235,67,0,67,67
```

Unit: rpm

timeDiff.txt

```
4,
140,
103,
94,
131
```

Unit: milliseconds

Figure 3.4: Exemplary walking motion dataset from ROBOTIS for the puppy robot, and the converted data required for the Dynamixels.

3.4.2 Robot Control With New Firmware

This section shows how the robot is controlled, once the firmware is reprogrammed. The program is simple and its only purpose is, to actuate the robot with simulation data. The remote control is used to make the robot execute the motion. The robot always starts from a neutral pose, with all Dynamixels at position 512. Later, this neutral starting pose is required for conveniently tracking the rigid bodies. Synchronous write instruction packets are sent to the Dynamixels, providing all Dynamixels at once with goal position and moving speed of the current pose. Then, the program waits for the instruction packet to be executed and continues to the next one. Initially, it was intended to make the robot move forward as long as a remote control button was pressed, and stop, once the button was released. The problem with this was that the signal transmission from the remote control to the robot was not perfectly stable, which lead to unwanted sudden stops during the movements if the signal from the remote control was not received in time. Because of that, the remote control is only used to start the robot movements and waits until the end. Then, the movements can be executed anew. Interference from the remote control during the movements is thereby omitted.
Algorithm 3.1 shows a simplified pseudocode version of the main() function of the program. The main() function comprises the code that is responsible for the behaviour of the robot. The main() function is part of a main.c file that defines port numbers, addresses and other microcontroller-specific data that is needed for the program. The main.c file is the only file that needs to be adapted in order to program the robot. Apart from the main.c file, there are the STM32F10x library files and header and source files for Dynamixel and remote control related functions for sending and receiving packets. They provide an abstraction of the hardware level. This functionality is already available in the Eclipse project and ready to be used in the main.c file.

Once the coding is done, the Eclipse project is built. A `cm530.hex` firmware file is created that needs to be transferred to the controller. This is done with the controller's boot loader. The boot loader cannot be erased and is used to install new firmware on the controller. RoboPlus provides a terminal through which the boot loader can be accessed when the controller is connected to the PC via USB. A new firmware can only be loaded once the old firmware has been erased. With the terminal, the new firmware file is then transmitted to the controller. The new program can be used right away with the robot. If new motion data is to be tested on the robot, the Eclipse project has to be built anew with the updated text files containing the Dynamixel data. Then the newly generated firmware has to be transmitted to the controller again. It is conceivable that this process could be automated in the future, to reduce the intermediate steps of building, erasing and installing firmware [fir].
Now that the robot is supplied with a motion, this motion can be further examined within the sandbox.

Algorithm 3.1: Robot Control Dynamixel Sync Write

```
void main()
{
    turn on power LED;
    set all Dynamixels to 512 (neutral pose);
    p = first pose;
    robotIsMoving = 0;
    while (1)
    {
        if (receiving data from ZigBee & robotIsMoving == 0)
        {
            if (button 'U' is pressed)
            {
                robotIsMoving == 1;
            }
            else if (button '1' is pressed)
            {
                set all Dynamixels back to 512 (neutral pose);
            }
        }
        else if (robotIsMoving == 1)
        {
            make synchronous write instruction packet;
            for (all Dynamixels)
            {
                set ID;
                set goal position of p;
                set moving speed of p;
            }
            set packet length;
            send packet;
            wait for Dynamixels to reach goal position;
            p = next pose;
            if (p == last pose)
            {
                robotIsMoving == 0;
                p = first pose;
            }
        }
    }
}
```

CHAPTER 4

The Motion Capture System

The motion capture system is the essential part of the sandbox that provides a connection between the real-world and the simulation. It enables the gathering of data that can be further processed and evaluated in the simulation. In this chapter, the requirements that the sandbox imposes on a motion capture system are pointed out, and an overview of motion capture techniques and their advantages and disadvantages is given. The considerations that went into the selection of a suitable system are discussed and its final setup is described.

4.1 Overview of Motion Capture Systems

This section gives a short overview of motion capture techniques to show what possibilities exist for tracking robots. Most of the systems rely on sensors or markers for tracking. The following classification of motion capture systems is taken from Field et al. [FSNP09] and partially accompanied by additional sources [MHQ16, SC02, xse, BR14, prib, FAQ, RMM18].

4.1.1 Acoustic tracking

Acoustic pulses are emitted from transmitters (e.g. loudspeakers) positioned around the desired tracking area. The moving object to be tracked is equipped with receivers (e.g. microphones). Depending on the distance to the transmitters, the microphones receive acoustic pulses of varying intensity. These pulses can be used to estimate the position of the object. If the object blocks the path between receiver and transmitter, reduced intensity does not correspond to distance and measuring errors occur. Background noise can be an issue depending on the frequencies used by the system. Outdoor use is possible, but humidity, temperature, and wind can also affect the accuracy. Tracking can be done in real-time, it is wireless and portable. However, there is no reference position. The tracking error of acoustic systems can be in the millimetre range, but also errors in the centimetre range are common [MHQ16, FSNP09].

4.1.2 Mechanical tracking

In a mechanical tracking system, either a reference position is connected with physical links and joints to an object, or an exoskeleton is used. There is no reference position using exoskeletons, therefore global orientation and position are not detected. The movement of the object is detected by measuring the orientations and displacements of the joints. On the one hand, mechanical tracking is fast and is not affected by occlusions or environmental influences and it is portable. On the other hand, only poses that are possible with the exoskeleton or the physical links can be tracked [FSNP09].

4.1.3 Magnetic tracking

Electromagnetic tracking systems have a transmitter that generates three orthogonal magnetic fields from three coils. The transmitter is fixed and position and orientation are known. The tracked object has a receiver, also with coils, attached. The transmitter generates current in the receiver coils. The signal in the coils is measured and the position and orientation relative to the transmitter can be calculated. Several receivers can be tracked. The farther away the receivers are from the transmitter the weaker the signal. Metal objects in the tracking environment can distort the signal. Further, the magnetic field has a limited range. Reasonable accuracy can only be guaranteed within approximately 1.0-2.5 metres of the transmitter. Wireless systems are available and there are no problems with occlusion [FSNP09, SC02].

4.1.4 Inertial tracking

Inertial tracking makes use of gyroscopic forces and measures acceleration and inclination with the help of accelerometers and inclinometers. Knowing the initial position of an object, the new position and orientation can be determined with accelerometers. Inclinometers measure angles or the tilt of an object. The advantage is that there is no limitation in range. This tracking method is used in large-scale systems, e.g. for flight navigation. Since the position is computed relative to the previous position, errors accumulate over time, making the tracking inaccurate. In large-scale systems, a global navigation satellite system can provide a reference to avoid the drift in position. VR or AR applications often just use orientation measurements [FSNP09, SC02, xse].

4.1.5 Optical marker-less tracking

Marker-less optical motion capture systems use image processing and segmentation techniques to find relevant features in images provided by cameras. Advantages are the low-cost equipment and that no markers or sensors are necessary. Cameras with depth sensors can capture 3D geometries of objects. Occlusions of the object to be tracked are a problem as well as the sensitivity regarding lighting conditions and high post-processing costs. It is suitable for applications, where accuracy is negligible [FSNP09, BR14].

4.1.6 Optical marker-based tracking

Optical marker-based motion capture systems use cameras to capture images of markers that are attached to an object to be tracked. With infrared-emitting cameras, reflective markers, also called passive markers, can be detected. It is not possible to distinguish markers from each other, but post-processing software reconstructs the correct marker path. In contrast to passive markers, active markers emit infrared light that is captured by the camera. It is possible to identify individual markers due to their illumination patterns.
From captured marker images, two-dimensional (2D) positions are computed and overlapping positions are triangulated to obtain a three-dimensional (3D) position [FAQ]. A minimum of two cameras is needed to perform the triangulation. This means a marker must always be visible by at least two cameras. Otherwise, occlusion errors happen. To counteract this, redundant markers can be used or the number of cameras is increased. Also, tracking space is restricted to the views of the cameras, but active markers have a higher tracking range than passive markers. Active markers need a power supply, therefore, some sort of charging device has to be attached to the object that is tracked. Passive markers do not require this and are light in weight. Depending on the setup, the system is portable, but it needs recalibration as soon as the cameras are moved. Such systems can reach accuracies of $+/-0.10\,\mathrm{mm}$ [prib, FSNP09].

Optical marker-based tracking can also be done with *ArUco* markers [RMM18]. These squared planar fiducial markers consist of a black border and an internal black and white binary pattern that uniquely identifies them. Tracking compared to the above system is cheap. The size of the markers is known and for the tracking one camera can be enough. The camera can also be fixed onto the object to be tracked, and the markers are positioned around the tracking area. Then the camera's pose is estimated via the fixed markers. This method is less performant and less accurate than tracking with the above system. Markers need to be placed on planar surfaces. This is not always possible for tracked objects [RMM18].

4.2 Requirements and Selection of System

The acquisition of powerful motion capture hardware is costly, therefore, it is necessary to check in advance if a motion capture system meets the requirements. The selection of a suitable motion capture system depended on the following considerations:

1. The system is used indoors.
2. The capture area should be between 1 m^2 and 2 m^2.
3. The system should be as accurate as possible (error $< 1\,\mathrm{mm}$).

4. The system should track in real-time.
5. The system should be able to track an approximately 25 cm tall robot.
6. The system should accurately provide 6-DOF data for all rigid body parts of the robot.
7. The system should be portable if necessary.

Given the above requirements, it was decided to use an optical motion capture system. It can be used indoors and for small capture areas. The accuracy of optical motion capture systems is very high and they can track in real-time.
Due to the small size of the robot, and the necessity of tracking every rigid body, sensors used by other motion capture systems might weigh the robot down and impair its movements. Passive markers can be very small and light-weight, which is ideal for the robot.
Possibilities were systems from Vicon [vic] or OptiTrack [opt]. Regarding performance and precision, they are both comparable and widely used in Virtual Reality, movement sciences, animation, and robotics. The choice fell on OptiTrack because it was cheaper than Vicon. Also, OptiTrack systems seemed more easily extendable and adaptable than Vicon systems.

4.3 Finding a Suitable Capture Area

In this section, important aspects when looking for a capture area are explained. To obtain the best accuracy while tracking, the environment of the system has to be taken into account. The best system will not deliver satisfactory results if the environmental conditions influence the tracking results.

It has to be considered that the actual capture volume that is covered by the cameras is significantly smaller than the area that is needed to set up the cameras. If it is possible to mount the cameras on walls, capture volume can be increased [cama]. Also, a truss system can provide a stable option for fixing cameras. Tripods are another option for camera mounting, but they are space-consuming and reduce the available capture volume. The advantage of wall mounts is that the cameras are not exposed to vibrations on the floor due to foot traffic. Cameras are fixed and cannot be moved accidentally. Tripods can be bumped into, and frequent foot traffic can affect camera calibration accuracy. But the camera setup can be rearranged easily, when cameras are added or removed. Tripods can be installed quickly and they are portable [cama].
Because of the advantages of tripods, it was decided to use them as mounting structure. This made it necessary to find a room that would not be frequented regularly to reduce the foot traffic and thus the need to recalibrate the cameras too often.
The robot is tracked on the ground, therefore the flooring should not reflect the infrared lights from the camera. Reflections can interfere with the tracking. Rubber mats can be used to cover reflective flooring. Flexible, deformable flooring, such as carpets, result in

unstable positioning of the tripods and should be avoided.
The markers are detected by reflecting the infrared light emitted by the cameras. Any other infrared sources can impact the tracking. Since sunlight contains infrared radiation, it is recommended to block windows [set]. Also, any other infrared light sources such as incandescent or halogen lights should be cleared from the capture volume or covered up. This also counts for inconspicuous materials that gleam bright white within the infrared spectrum [set].
Objects in the capture volume that are not needed should be removed because they can lead to occlusions between the cameras and the markers [set].

For the installation of the motion capture system, IST provided an empty room in the basement. The available space for the setup is 3.9 m $\times$ 3.9 m. There are no windows, the concrete floor does not reflect infrared lighting and foot traffic is rare. This makes it a suitable room for tracking.

4.4 Camera Specifications

The cameras are the most expensive part of a motion capture system. How many cameras are needed for tracking the robot depends on the way the robot is tracked. It has to be clarified if only the three-dimensional position of the robot is of importance, or also the positions of its limbs.
In general, two cameras are sufficient for reconstructing a three-dimensional point from markers. However, if a marker gets occluded during tracking and one of the cameras loses direct view to the marker, reconstruction is not possible. It is recommended to use more than the minimum number of cameras necessary in case one or more cameras do not have an unobstructed view to a marker. This increases the probability that there are still cameras left that see the marker.
As mentioned before, the number of cameras also depends on what needs to be tracked. If only the position of the robot's body is required, without its limbs, then marker placements on top of the robot are sufficient, tracking can be done with at least three or four cameras.
For this work, also tracking of the limbs is required, more precisely, the tracking of the rigid bodies of the robot. The robot consists of 16 rigid bodies that are connected with motors. There are three rigid bodies per leg, the torso consists of two, and the neck and the head also consist of one rigid body each. The head and the neck are not tracked, since they do not contribute to the locomotion. More about the robot tracking is explained in Chapter 5. The remaining 14 rigid bodies occlude each other one way or another. A single camera can never capture all rigid bodies at once. To ensure that all rigid bodies are visible from at least two or three cameras, it was decided to use six cameras.

The motion capture system uses six OptiTrack Prime 41 cameras [pria]. In Figure 4.1, the Prime 41 is shown from different sides. The camera can track passive and active markers and combinations of both.
The 12 mm lens has a wide band anti-reflective coating, which benefits light transmission

and provides large volumes with low distortion [pria].
Like all OptiTrack cameras, the Prime 41 has integrated image processing, which detects markers and computes marker centers. It can also be used for capturing reference video material with a 3D overlay of markers [pria].
The cameras deliver 2D object data in form of segmented marker images, as well as reconstructed 3D points. Further, multiple markers can be combined to define a rigid body, whose position and orientation can be tracked [pria].
With an additional Camera SDK, it is possible to access the raw camera frames and use them for custom computer vision tasks. However, this is not used in this work [pria].
In Table 4.1 the specifications of the Prime 41 are listed [prib].

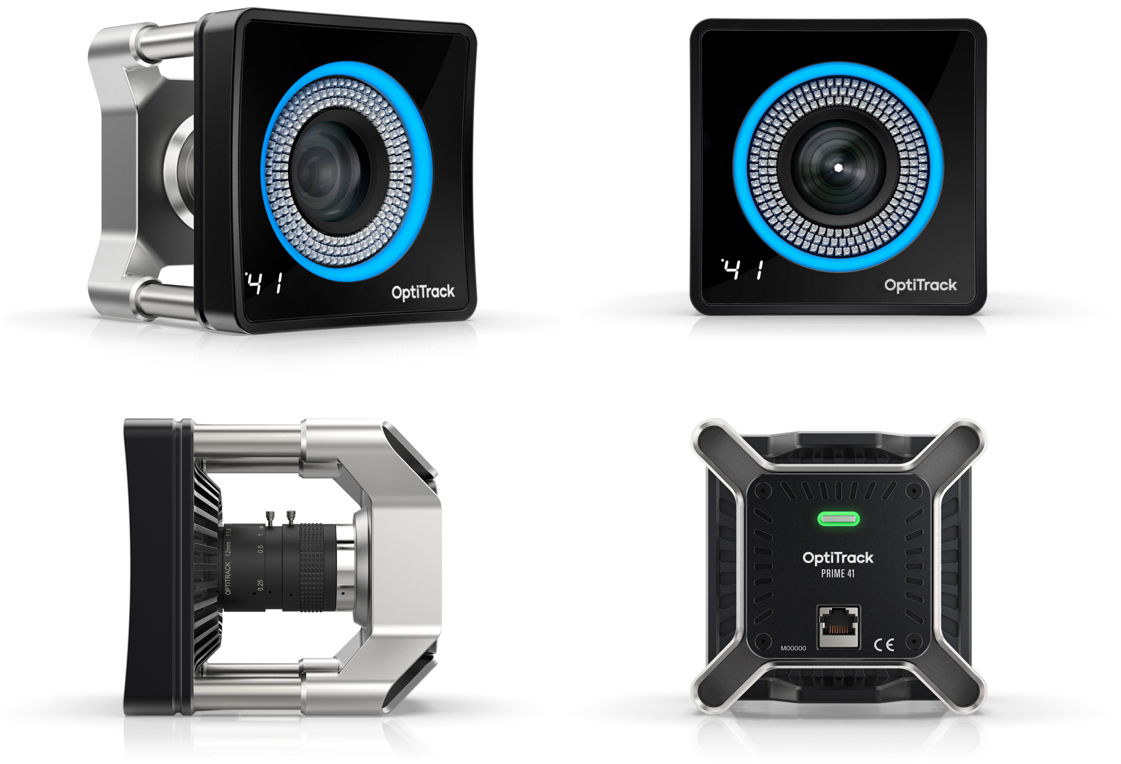


Figure 4.1: OptiTrack's Prime 41 tracking camera [pria].

4.5 Camera Placement

The placement of the cameras in an optical motion capture system is important for getting the most information out of each captured image. If the cameras are well-arranged, the tracking accuracy can be improved significantly. The 3D coordinates of a marker are reconstructed from the 2D view of each camera. Different vantages on the tracking volume provide wide angles that improve triangulation results and thus tracking quality.

Image Sensor	Resolution: Frame Rate: Latency:	2048×2048 30 - 180 Hz 5.5 ms
Tracking Performance/Range	3D Accuracy: Passive Markers:	+/- 0.10 mm 30 m
Shutter Speed	Default: Minimum: Maximum:	0.5 ms 0.01 ms 5.3 ms at 180 fps
Image Processing	Object Segmentation Raw Grayscale MJPEG Grayscale	 512×512
LED Ring	170 LEDs 850 nm infrared adjustable brightness Illumination:	 strobe, continuous
Lens & Filter	12 mm F/1.8 51°horizontal FOV 51°vertical FOV adjustable focus and F-stop 850 nm band-pass filter	
Connections	GigE (Gigabit Ethernet) data port PoE+ (Power over Ethernet) Ethernet camera sync	

Table 4.1: Technical camera specifications of OptiTrack's Prime 41 [prib].

Because of that, cameras should be distributed evenly around the tracking volume. This also reduces the possibility of marker occlusions [camb].
The camera placement varies depending on the capture application. If high accuracy is important, cameras should be placed close to the object to be tracked. This ensures that a marker covers more pixel in the captured image. However, this also reduces the capture volume. If only one side of an object needs to be tracked, it can also be sufficient to place cameras only on the specific side [camb].
It has to be noted that the actual tracking volume is a lot smaller than the space available for the camera placement.

For this work, it is necessary to track the 3D position and rotation of the robot's rigid bodies. It should be possible to track different movement patterns of the robot. Currently, the robot's movement is restricted to a straight-ahead walking motion, but other motions such as turning around are conceivable. Therefore, the robot needs enough space to

perform these walking cycles without leaving the capture volume.
In Figure 4.2, the final camera placement can be seen. The cameras have been arranged

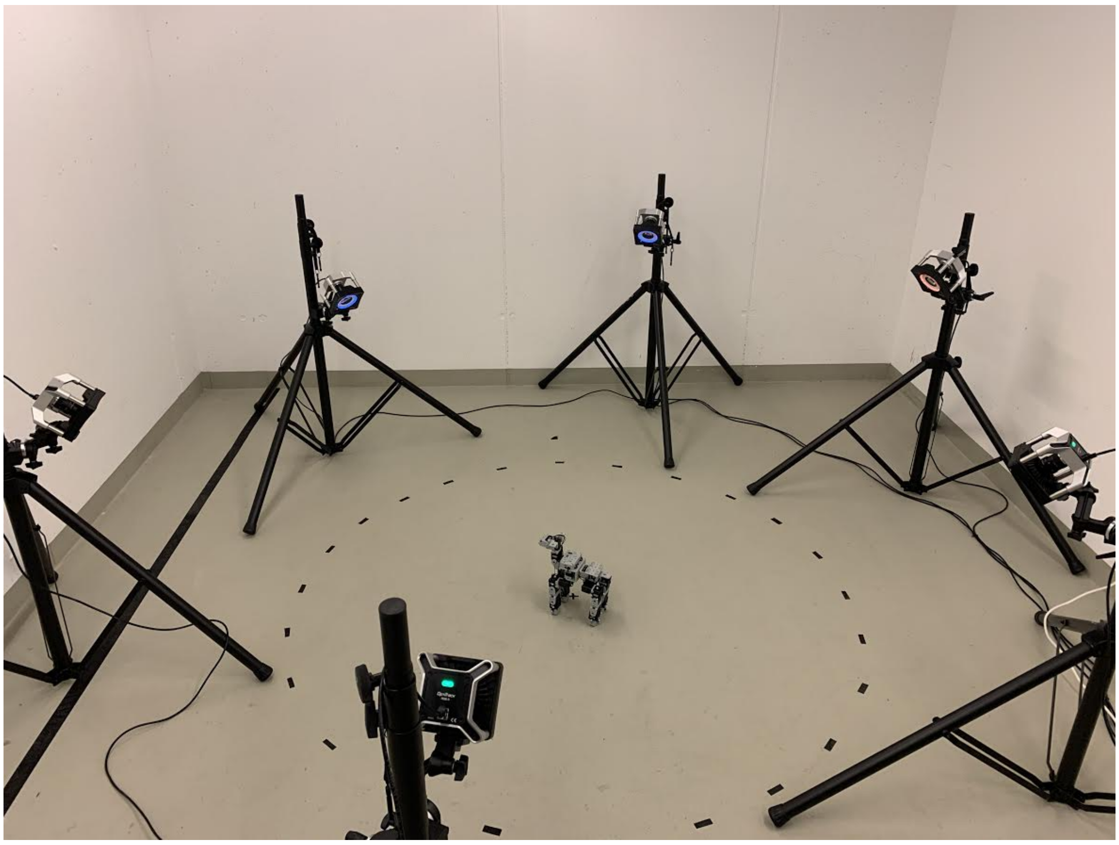


Figure 4.2: The motion capture system setup with six cameras circularly placed around the capture volume. The black circle on the floor approximately indicates the border of the capture area.

in a circle approximately equidistantly around the capture volume. The border of the capture volume is marked with black tape on the floor. Markers close to the taped circle are not seen by all cameras. Placing the cameras circularly and at a constant distance from each other results in an evenly covered volume captured from all sides. Due to the room size, the cameras can not be placed farther away from each other while keeping their circular arrangement, but another way to increase the capture area is to position the cameras higher up. Since the robot is small and moves on the ground, tracking might get less accurate with an elevated camera position. Markers on the lower limbs of the robot can be harder to track if the cameras are looking down on the robot.
However, if cameras are positioned at a low height, cameras across from one another detect each other's infrared lights. The lights would need to be masked in the captured images and tracking data would be lost in that area, so this placement is avoided [camb].
It can be useful to place cameras at different elevations to provide more varying views of

the capture volume. Then again cameras at lower elevations provide a smaller capture area and the overall capture area that is visible by all cameras is reduced [camb].

An alternative to the camera placement depicted in Figure 4.2 could be a setup, where three cameras are positioned on the left and right side of the robot respectively. Since the robot only walks straight-ahead in one direction, ideally, the markers on both sides of the robot are captured by the three cameras on each side. But once the robot is capable of other motions, this setup would have to be altered.

4.6 Camera Network

The Prime 41 is an Ethernet-based camera with a Gigabit Ethernet port (1000 Mb/sec). Ethernet cables have fast data transfer rates and simultaneously power the cameras. No additional power cables are needed, which facilitates the setup. It is important that the camera network is separated from other local area networks (LANs) to prevent interference with other data traffic, which could lead to frame drops during capturing [wir].
Besides the cameras, a PoE+ Gigabit Ethernet switch is needed. Its main functions are the data transfer to the host PC and the consistent power supply to the cameras via the Ethernet cables. The difference between PoE and PoE+ is that PoE provides 15.4 watts to the cameras, whereas PoE+ provides 30 watts. The Prime 41 has stronger infrared strobes than other OptiTrack cameras and therefore requires a PoE+ switch. This motion capture system uses a Netgear ProSAFE GS728TPP switch [swi].
Additionally, all cameras need to be synchronized. OptiTrack's eSync 2 [esy] is used for synchronisation. It is also connected to the switch and aligns the camera input signals with a precision of approximately +/- 5 µs.
In Figure 4.3, the schematic camera network is shown. The six cameras are connected with Ethernet cables to the switch, which also connects to the eSync 2 that provides the synchronisation for the whole system. Another Ethernet cable connects the switch to the host PC.

4.7 Aiming, Focusing, and Calibration

Once the camera network is connected to the host PC, the motion capture setup can be optimized. The camera's focus and aiming are adjusted to the capture area and the cameras are calibrated. Then, markers are combined to rigid bodies (see Chapter 5) and the tracking is performed. All of this is done in *Motive:Tracker* Version 2.2, OptiTrack's software platform that provides control over the motion capture system [motb].
Motive has two main viewports, the perspective view and the camera preview, which are used to observe 3D and 2D tracking data. The perspective viewport provides reconstructed 3D representations of the captured data, the markers, and corresponding rigid bodies. Also, the cameras are visualized. In the camera preview marker images from each camera are shown. Depending on the camera settings, also full resolution raw-grayscale and compressed MJPEG-grayscale images are provided [mota].

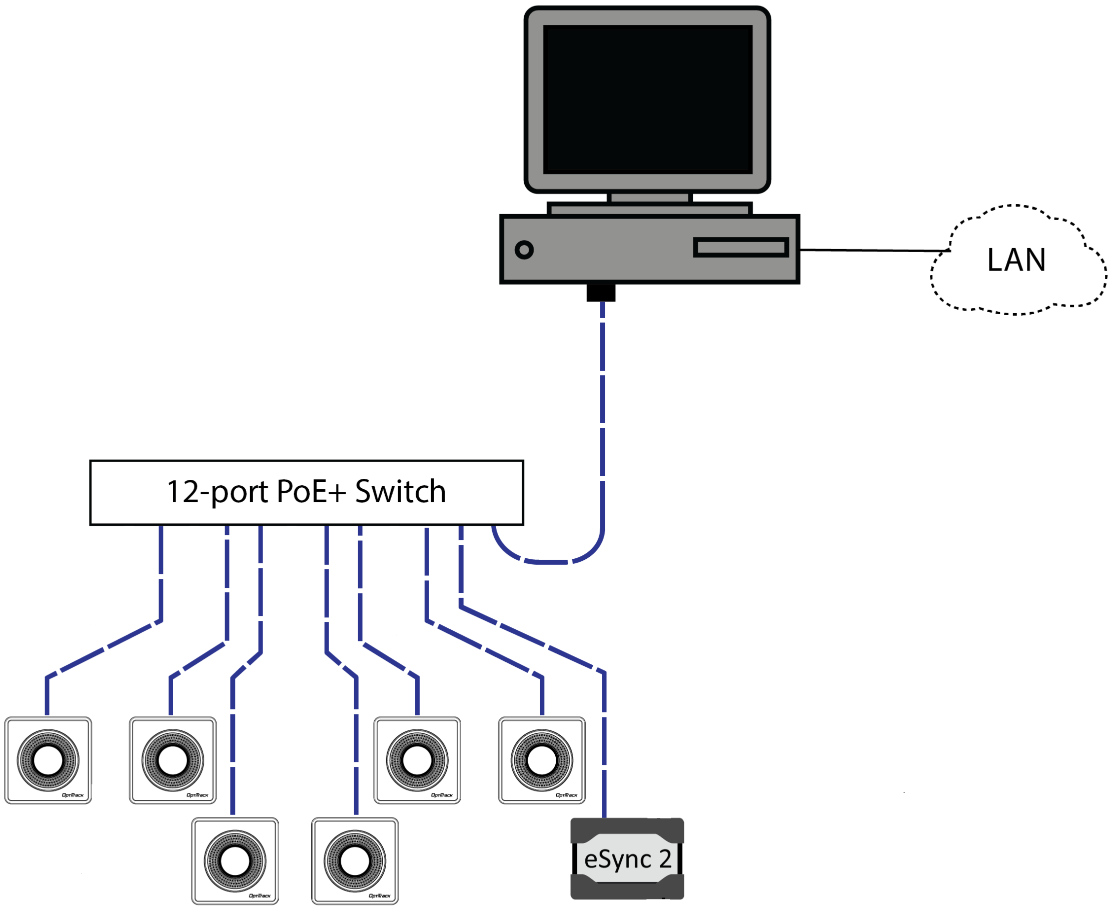


Figure 4.3: Wiring of the motion capture system. The cameras are connected via Ethernet cables to a switch that provides power and transfers the data to the host PC. An eSync 2 provides the synchronisation for the cameras. The camera network has to be separated from the LAN to prevent frame drops [esy, wir].

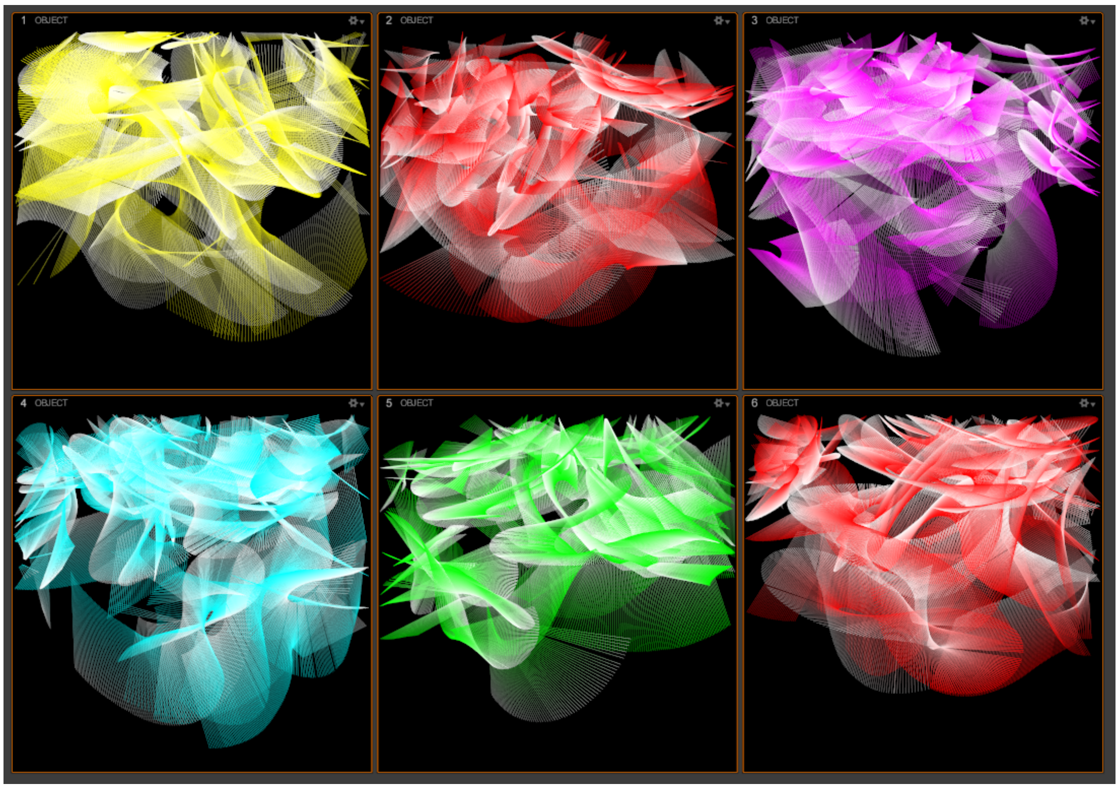
The raw-grayscale images help aim and focus the cameras onto the capture volume. If all cameras are aimed at the desired capture region a marker is placed inside the capture area. In the camera preview, the marker is zoomed in and on the cameras, the focus is adjusted until the marker image is resolved [aim]. A low f-stop (large aperture) is beneficial for capturing bright images. All cameras have an exposure of 80 µs.
Before calibrating the cameras, it has to be made sure that all cameras are positioned correctly and are focused on the capture volume. Any changes made after calibration require a recalibration of the system. Calibration is required to calculate each camera's relative position and orientation as well as the distortions in the camera images [cal]. For this, a calibration wand is moved through the capture volume and markers on the wand are captured by the cameras. The markers are fixed at known distances alongside the wand. The wand used for calibration has three 12.7 mm markers with a spacing of

250 mm and a standard deviation of two microns. The accuracy of the wand is essential for the accuracy of the motion capture system. From the marker positions in the images, the cameras triangulate their relative positions and orientations to each other. The whole capture volume should be evenly scanned with the wand, and all cameras should capture a similar number of wand samples. More samples can be useful in regions where most of the tracking happens. This increases accuracy in that specific area. 2000 - 5000 samples are sufficient [cal]. Too many samples negatively influence the accuracy of the calibration. The Prime 41 indicates with colored LEDs when enough samples have been gathered, then Motive computes the calibration and lens distortion automatically [cal]. In Figure 4.4a, the path of the calibration wand is shown from the perspective of each camera. The calibrated capture volume with the relative camera positions and orientations can be seen in Figure 4.4b. The path that has been traced with the calibration wand is visualized in blue. In Table 4.2, the calibration results can be seen in terms of numbers. Mean 2D and 3D reprojection errors, triangulation and wand errors are given. A maximum residual offset between rays belonging to one 3D point is recommended for reconstruction, as well as a maximum marker-to-camera distance in which reconstruction from markers is considered [cal].
Usually, camera calibration results degrade over time due to temperature changes or slight camera movements. Motive offers continuous calibration, which automatically recalibrates the system after the first calibration [cal].

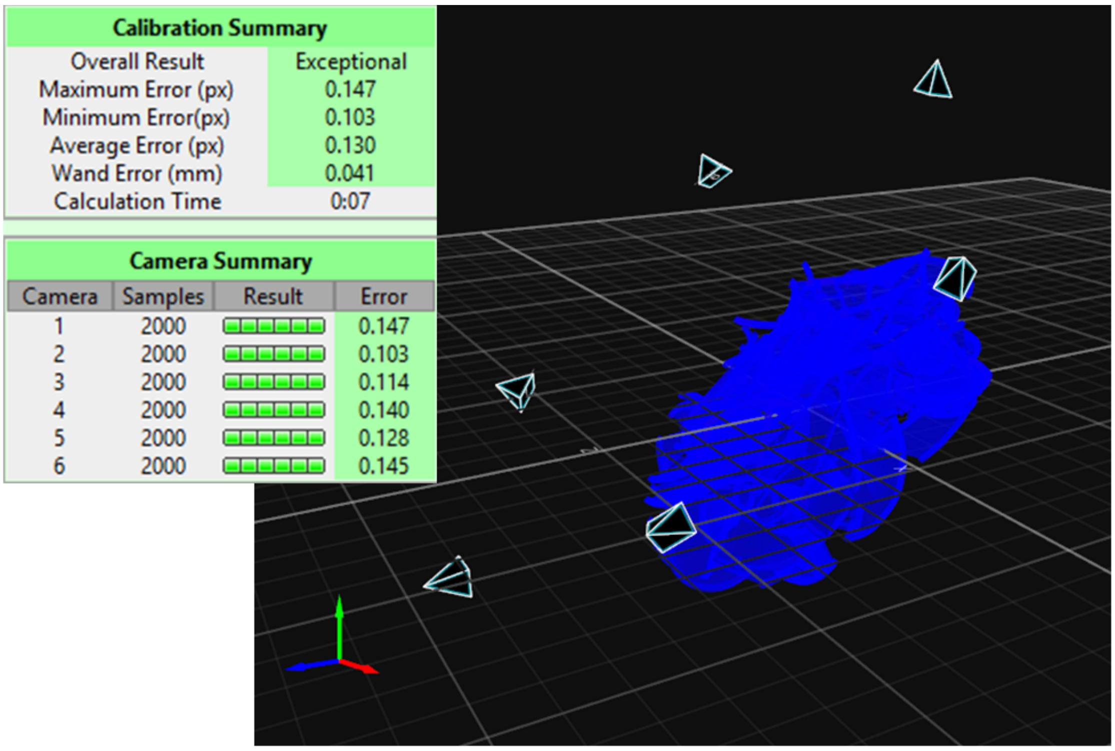
After calibration, the ground plane and the origin of the capture volume have to be defined. This is done with a calibration square consisting of a longer and a shorter leg and three markers, one at the corner and two at the legs. The marker at the corner indicates the origin, the marker at the shorter leg the positive x-axis and the marker at the longer leg the positive z-axis [cal].

Overall Reprojection	Mean 3D Error:	0.123 mm
	Mean 2D Error:	0.130 mm
Worst Camera	Mean 3D Error:	0.124 mm
	Mean 2D Error:	0.143 mm
Triangulation	Recommended residual offset:	2.600 mm
	Residual Mean Error:	0.100 mm
Overall Wand Error	Mean Error:	0.041 mm
Ray length	Suggested Max:	3.100 m

Table 4.2: Calibration results with mean errors. The residual offset is the maximum offset distance between rays belonging to one 3D point. The wand error displays the error of the captured wand length compared to the provided wand length before calibration. The ray length suggests a maximum marker-to-camera distance in which the reconstruction from markers is considered [cal].



(a) Path of the calibration wand from the perspective of all six cameras.



Calibration Summary

Overall Result	Exceptional
Maximum Error (px)	0.147
Minimum Error(px)	0.103
Average Error (px)	0.130
Wand Error (mm)	0.041
Calculation Time	0:07

Camera Summary

Camera	Samples	Result	Error
1	2000	🟩🟩🟩🟩🟩🟩	0.147
2	2000	🟩🟩🟩🟩🟩🟩	0.103
3	2000	🟩🟩🟩🟩🟩🟩	0.114
4	2000	🟩🟩🟩🟩🟩🟩	0.140
5	2000	🟩🟩🟩🟩🟩🟩	0.128
6	2000	🟩🟩🟩🟩🟩🟩	0.145

(b) Calibration results, calibrated capture volume, relative camera positions and camera orientations shown in the perspective viewport. The tracked wand samples are shown in blue. The ground plane has not been defined yet, so, the volume is not aligned with the coordinate system.

Figure 4.4: Wanding samples and calibration results.

CHAPTER 5

Tracking the Robot

Following the hardware setup and the camera calibration, the system is ready for tracking the robot. In this chapter, important aspects regarding marker placement on the robot and creating rigid bodies for tracking the robot's movements are explained. The tracking results are discussed in the context with data post-processing.

5.1 Rigid Body Marker Placement

The motion capture system uses passive reflective markers. The marker placement on the robot is crucial for accurate tracking results. Apart from the neck and the head of the robot, all other body parts are tracked.
Markers are available in different sizes and shapes. Marker sizes are selected depending on the desired tracking accuracy and the size of the tracked object. Smaller markers lead to more precise tracking but are more difficult to track if the cameras are farther away [mar]. Due to the robot's small size, it is necessary to use small markers.
In Figure 5.1, the robot is shown from the front, back, top, left, and right side with the markers. 6.4 mm spherical markers were used for the robot in combination with 4 mm hemispheric markers. The spherical markers are attached to 9 mm × 4 mm marker bases and adhered onto the robot. The hemispheric markers can be glued directly onto the robot without marker bases. For every rigid body, either four spherical or three hemispheric markers were used.
The markers have a reflective surface that is recognized by the cameras. If the surface is scratched or otherwise damaged, the tracking can be limited [mar].

The following points are important for marker placement:

- The markers have to be placed in a way, such that they do not interfere with the robot's motion or are occluded by a bent leg. One marker on its own only provides a position, but multiple markers in relation to each other can provide a position and orientation (pitch, roll and yaw) [rbt]. For the tracking, it is therefore important to combine multiple markers and tell Motive that they belong to the same rigid body. In Motive, those markers are then connected with each other to represent a schematic virtual rigid body. Motive automatically labels the markers, when creating a rigid body [rbt].

- To create the rigid body in Motive, at least three markers are necessary to define a plane in 3D space. Additional markers can serve as a backup in case a marker is occluded during tracking, and they add additional position data to the rigid body, which makes the tracking more stable [rbt].

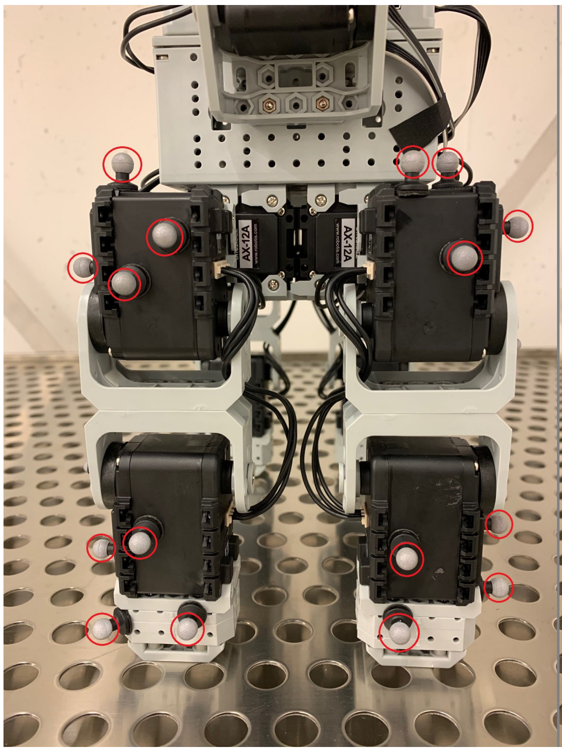
- When placing too many markers on a rigid body in close vicinity, they can overlap in the camera image and Motive cannot distinguish the marker reflections. Consequently, during capture, Motive may swap the marker labels. OptiTrack recommends using at least four markers per rigid body [rbt].

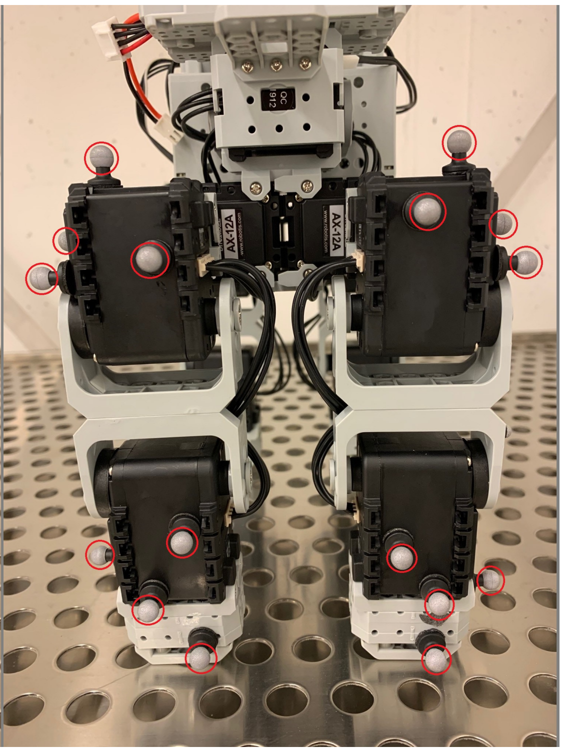
- Markers belonging to a rigid body should be spread as far as possible to obtain more accurate orientation data. In doing so, even small positional changes lead to slight changes in orientation [rbt].

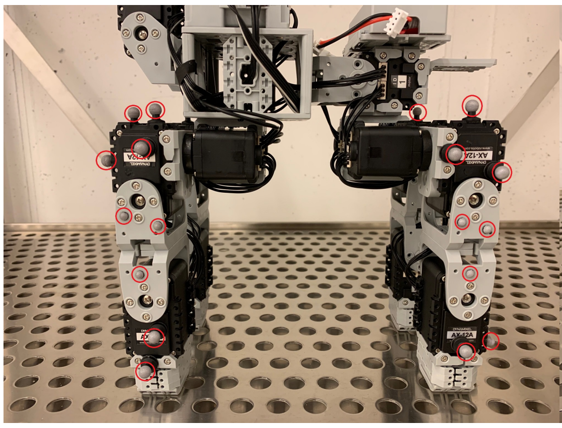
- It is also important to place the markers asymmetrically within a rigid body. Shapes like squares or equilateral triangles should be avoided. During capture, it can make the rigid body flip, which leads to wrong orientation data [rbt].

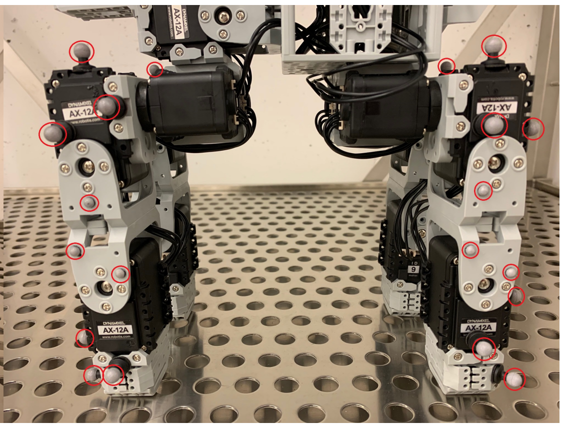
- Further, the marker placement for multiple rigid bodies should be unique. This helps to stabilize tracking and prevents labelling errors. Uniqueness can also be achieved by varying the distance between markers and the number of markers per rigid body [rbt].

The rigid bodies of the robot, especially the three rigid bodies that make up a leg, are small and there is little space for uniqueness regarding marker arrangements and marker-to-marker distances. It has to be considered that markers between two adjacent rigid bodies should not be too close together or form arrangements that are similar to the rigid bodies' arrangements themselves. As can be seen in Figure 5.1c and Figure 5.1d, there are only three markers for the middle rigid body of the legs. This was done to avoid having too many markers too close to each other, and to make the middle rigid body unique in marker count compared to its neighbours.
In Figure 5.2a, all 50 markers on the robot are visualized in Motive. The first three markers, belonging to a rigid body, have the same color, the fourth marker is colored differently. Figure 5.2b shows the corresponding rigid bodies. Markers belonging to a rigid body are connected with lines. The large spheres indicate the pivots of the rigid bodies. The rigid bodies have a coordinate system attached and provide 6-DOF data.


(a) front


(b) back


(c) left


(d) right

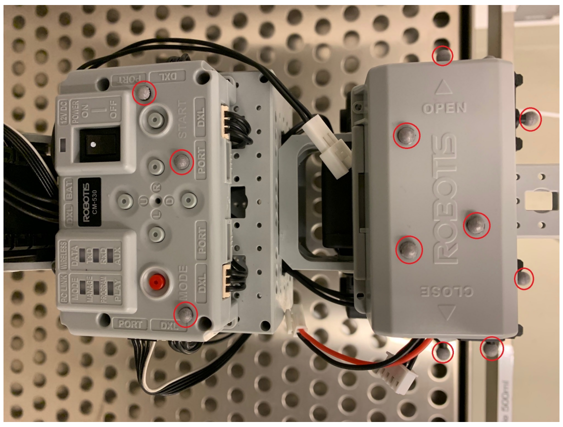
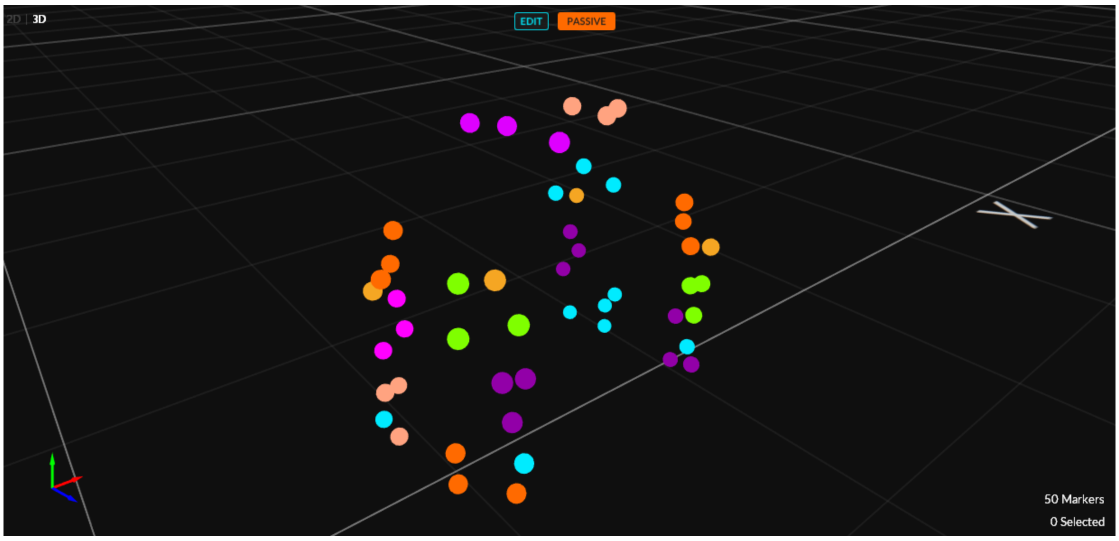
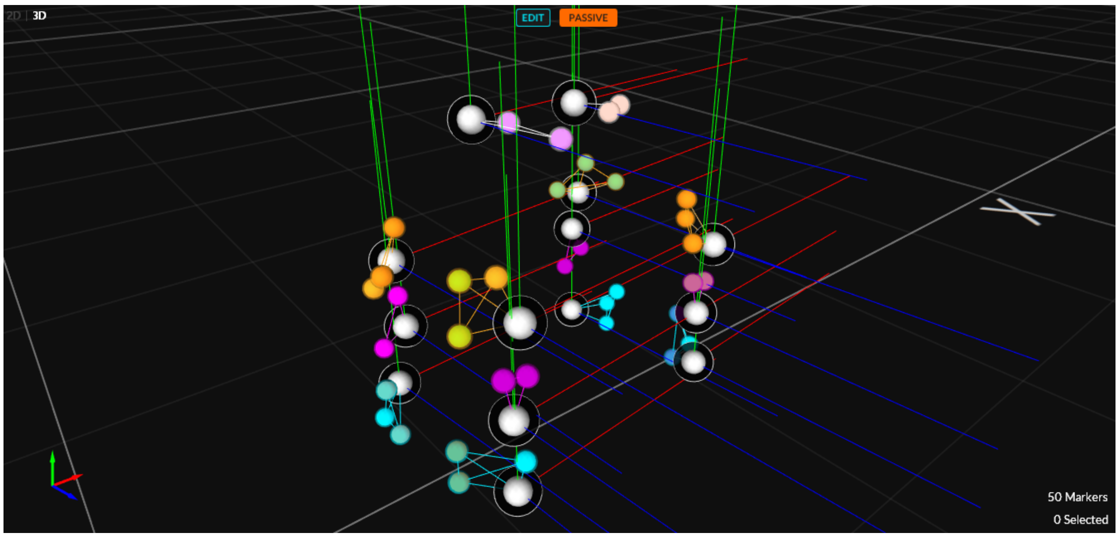

(e) top

Figure 5.1: Marker placement on the robot. Per rigid body, either four 6.4 mm reflective precision spheres (markers) with 9 mm × 4 mm marker bases and rubber adhesive or three 4 mm hemispheric markers are used.



(a) Markers of the robot shown in Motive. A rigid body has three or four markers. Markers belonging to the same rigid body have the same color. If a rigid body has four markers, the fourth one is colored differently.



(b) Definition of the rigid bodies in Motive. The larger spheres indicate the pivots of the rigid bodies. The pivots correspond to marker positions. Every rigid body has its own coordinate system. Markers belonging to a rigid body are connected with lines.

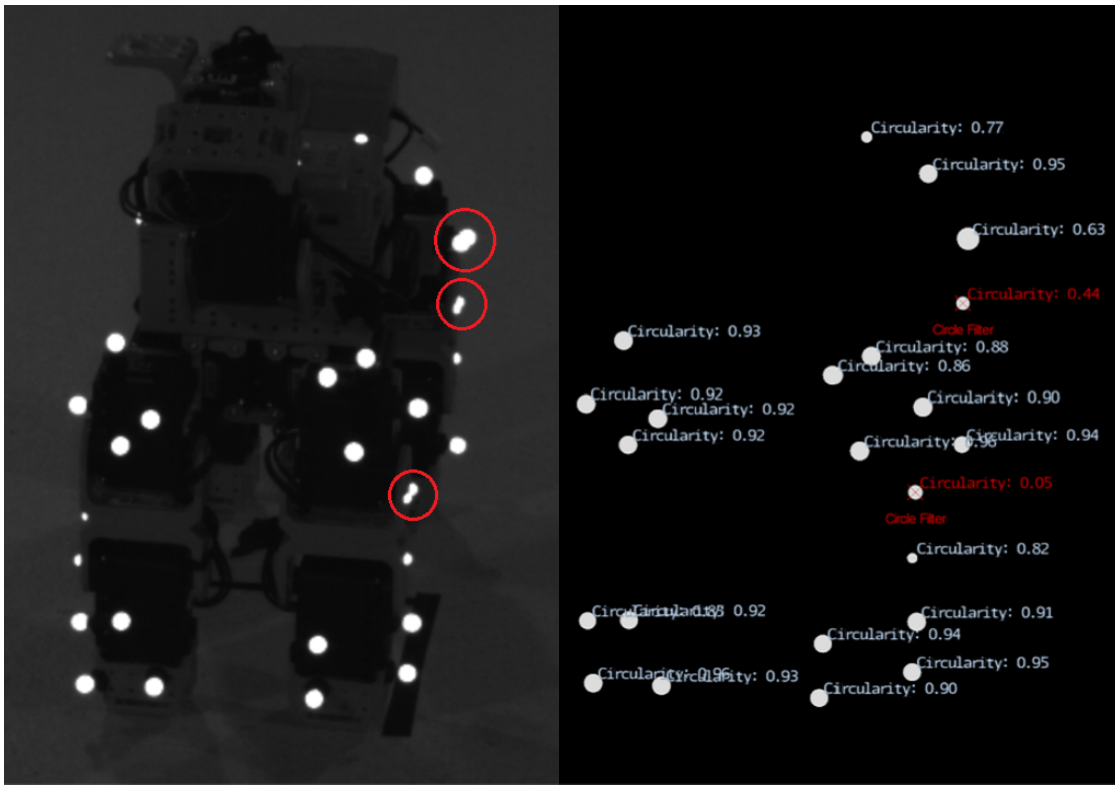
Figure 5.2: Markers and rigid bodies defined in Motive. The robot is shown from the top-front-left. Altogether, 50 markers are used for the robot.

5.2 Optimizing Reconstruction

In the reconstruction process, 3D points are derived from 2D coordinates that are captured by the cameras. In real-time, the 2D marker centroid locations are triangulated frame-by-frame from synchronized camera images. It is also possible to reconstruct 3D data after capture in post-processing [rec].
The cameras have a threshold setting that specifies what minimum brightness value a pixel must have to be considered for processing. Pixels below the threshold are filtered out. A lower threshold also considers dim or small markers that are far away from the camera, a higher threshold helps to filter unwanted reflections. The threshold for the cameras is set to 200. 255 would be the maximum value. 200 is the default setting and works well for this application [rec]. To optimize the marker detection, the tracking is performed in the dark. Only the cameras' LEDs provide light. This makes the reflective markers stand out more in the darkness, increases the contrast between the background and the markers and reduces reflections from outside the capture volume [rec].

The thresholded camera images are then further processed through two filter stages. The first filter is the *2D object filter* and is applied on the cameras' hardware level, the second filter is on the software level. They are essential for deciding which 2D reflections belong to a marker and are reconstructed into 3D points.
The 2D object filter checks the thresholded reflections and determines based on their size and shape if they are marker reflections or not [rec]. The size of a reflection is specified by its number of pixels. The minimum and maximum number of pixels per reflection depend on the capture application. Close-up captures have bigger marker reflections. The minimum threshold has been set to ten pixels (default: 4 pixels) to remove small flickering noise that occurs sometimes. The maximum threshold has been kept at default (2000 pixels) because it does not negatively affect the filtering. Most marker reflections in this application have sizes between 20 and 100 pixels [rec].
The shape of the reflections is specified by their roundness. It is assumed that marker reflections are circular. Reflections that are not circular are filtered out. The circularity threshold can be set between zero and one. Zero means no circularity and one means perfect circularity. The circularity threshold is set to 0.6 (default). Most marker reflections have circularities between 0.80 and 0.95. From the reflections that fulfil the size and roundness criteria, their 2D centroid is determined. This data is then sent to the host PC [rec].
Figure 5.3 shows the results of the 2D object filter stage on a zoomed-in camera image. Marker reflections that do not meet the requirements are crossed out. It happens that from a certain camera view, markers partially occlude each other. Such reflections are considered as a single marker reflection and are mostly filtered out by the circularity filter.

In the second filter stage, it is decided which marker centroid contributes to a reconstructed 3D point. The 2D centroid shoots a marker ray perpendicular to the camera image into the capture volume. If at least two rays converge within an acceptable offset distance (residual distance), reconstruction is performed [rec]. The accuracy of the ray convergence depends



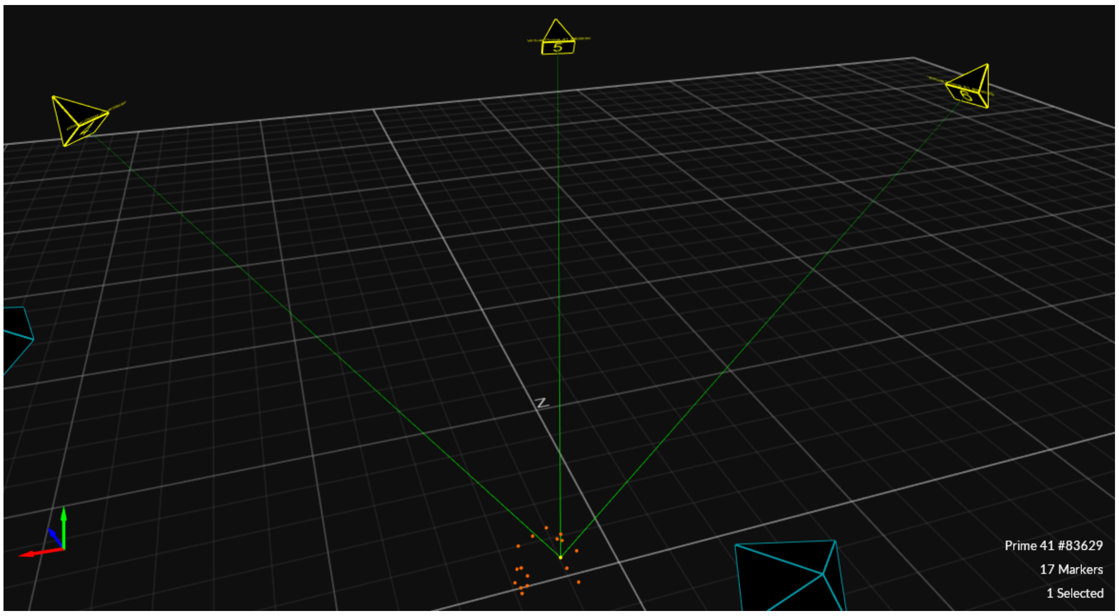
(a) (b)

Figure 5.3: Zoomed-in on camera image for a better view on markers. 2D object filter selecting only the marker reflections for reconstruction that meet the size and roundness criteria. Partially occluded markers (highlighted red on the robot's left) (a) are filtered out if they do not satisfy the roundness criterion of ≤ 0.6 (b).

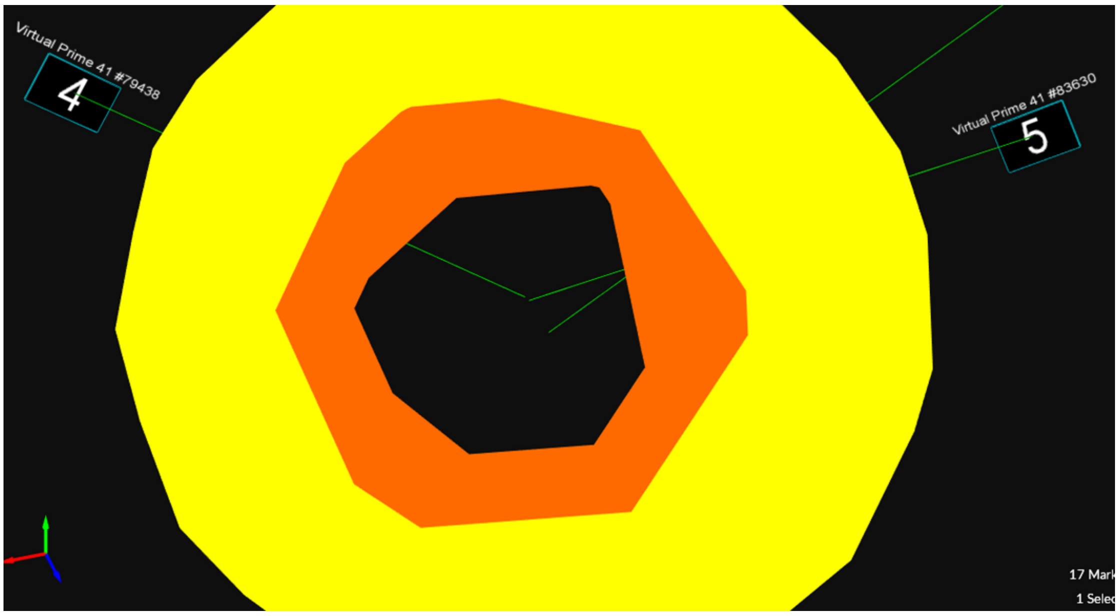
on the calibration results. The calibration results suggest what maximum offset distance between rays is allowable for contributing to the reconstruction. A ray that is beyond the maximum offset distance to other rays is not considered. With degrading calibration, the maximum offset distance is increased automatically. For precise reconstruction, the distance should be lowered. The maximum offset distance in the system is set to 2.57 mm. Further, if two markers have a distance smaller than the minimum offset, they are merged and reconstructed as one marker [rec].

Per default, the minimum ray count that is required for reconstruction is two. The ray count can be increased to prevent irrelevant reconstructions or decreased to counteract marker occlusions. This system only has six cameras, and most markers, especially those on the legs, are usually not seen by more than three cameras at once because the view of the other cameras is blocked by the robot's body. Therefore, the minimum ray count has been kept at two [rec].

In Figure 5.4, marker rays from three cameras to a selected marker are shown. The



(a) Three cameras track the selected marker. The marker rays are shown in green.



(b) The offset of the converging rays can be seen inside the marker.

Figure 5.4: Marker rays from cameras to selected marker (a) and residual distance (b).

residual distance between marker rays can be seen when zooming inside the marker.

The reconstruction approach described above is used for the standard marker-based tracking. Marker-based tracking is the preferred option for systems with more than five cameras. Ray based tracking is another possibility, which is suitable for a setup with fewer cameras. Ray based tracking also takes rays that would not have been considered for reconstruction because the minimum ray count was not fulfilled. Additionally, it uses rigid body definitions to enable more stable tracking [rec].

5.3 Tracking and Post-Processing

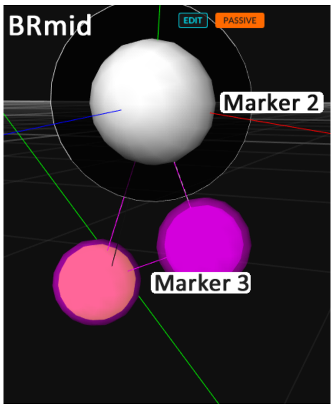
Once the rigid bodies are defined in Motive, the robot is ready to be tracked. The robot's joints, which are actuated by the Dynamixel motors have one rotational DOF. The robot was positioned along the x-axis, such that the z-axis would be the axis of rotation for the robot's joints. Apart from the top joints of the legs that rotate around the x-axis, all other joints rotate around the z-axis.
When rigid bodies are created, their coordinate system is aligned with the global coordinate system. Motive uses right-handed coordinate systems. It is beneficial to position the robot parallel to either the x- or z-axis, otherwise, the robot's rotational axes are not aligned. This would require to compute the rotational axes from the x-, y- and z-rotations of the rigid bodies. Another way is to align the rigid body coordinate system with the robot's rotational axes, in case the robot is not positioned parallel to the global axes. It is important that the robot starts in a neutral pose, such that the angles of all joints are zero. For the Dynamixels, this means their goal positions are at 512. The rigid bodies are per default aligned with the global coordinate system. A random starting pose would make it harder to later compute the correct joint angles from the rigid body rotations since the joint angles were not zero.

In Motive, a single motion capture recording (or take) is saved in a *Take* (.TAK) file. This file contains all the necessary data that is needed to recreate the capture. This includes the camera calibration, the 2D data from the cameras, the 3D reconstructions, the labelled markers and rigid bodies, and edits from the post-processing.
Motive has a live and an edit mode. In the live mode, the cameras are active and streaming data that is reconstructed in real-time. In live mode, motion capture recordings are performed. In edit mode, the captured data can be played back from Take files. The data can be edited if the tracking produced label swaps. Gaps in marker trajectories can be interpolated [mota].
Motive distinguishes between three data types: 2D data from the camera images, 3D data that is reconstructed from the 2D data, and *solved data*. Solved data is the 6-DOF data of rigid bodies. This data is computed during tracking from the 3D data. Solved data can be deleted and recomputed in post-processing. This is necessary if labels are swapped during capture or occluded, such that solved data is created incorrectly [motb]. An example of label swaps and gaps in tracking data is depicted in Figure 5.5. Because of the swap, the rigid body's orientation is flipped and the expected marker positions of

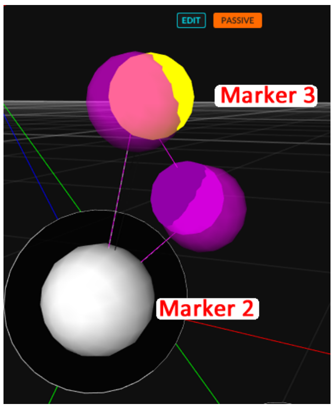
the rigid body do not correspond to the actual marker positions anymore (Figure 5.5b). Gaps in the tracking data keep the rigid body at its expected position, but without marker data (Figure 5.5c). Taking a look at the trajectories in the Graph View shows the swapped marker paths and the missing tracking data (Figure 5.5d). This is corrected manually during post-processing and the trajectories are smoothed (Figure 5.5e).
Figure 5.6 shows tracking results of four markers belonging to the front-left-bottom (FLbtm) rigid body of the robot and the corresponding 6-DOF solved data. Motive's Graph View shows the position and orientation data from frame 80 to frame 880. This data has already been post-processed.

5.4 Data Export

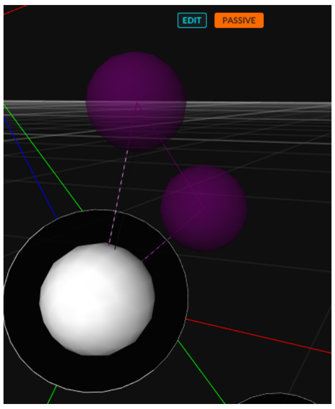
After post-processing, rigid body positions and orientations are exported as comma separated values (CSV).
The position data consists of x, y and z values in centimeters. For the rotation data, either Quaternions or Euler angles can be used. In the motion capture system, the robot is positioned such that its revolute joints either have x-axis or z-axis rotations, so the representation in Euler angles (rotation order xyz) was chosen since Euler angles already have separated x-, y- and z-rotations. For Euler rotations, a right handed-coordinate system is used. Gimbal lock is a disadvantage of Euler angles compared to Quaternions, however, the robot does not perform such extreme motions, so gimbal lock is unlikely to occur.
The exported data is in local rigid body coordinate space. The rigid bodies have to be in alphabetical order as follows: *body, BLbtm, BLmid, BLtop, BRbtm, BRmid, BRtop, FLbtm, FLmid, FLtop, FRbtm, FRmid, FRtop, tail.* This facilitates the loading of the data in the simulation afterwards.
It has to be noted that the rotations of the rigid bodies accumulate from the *body* down to the *btm* joints. For instance, rotations captured at the *FLbtm* rigid body already consist of rotations happening at *FLmid* and so on. This has to be corrected when working with the rotations later in the simulation.
Only frame ranges of interest are exported, capture data at the beginning and at the end, where the robot does not yet move is excluded. The exported frame rate is 120 Hz.



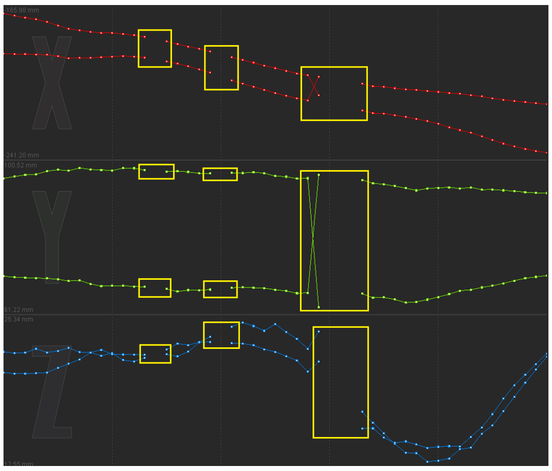
(a) Frame 698 (correct).



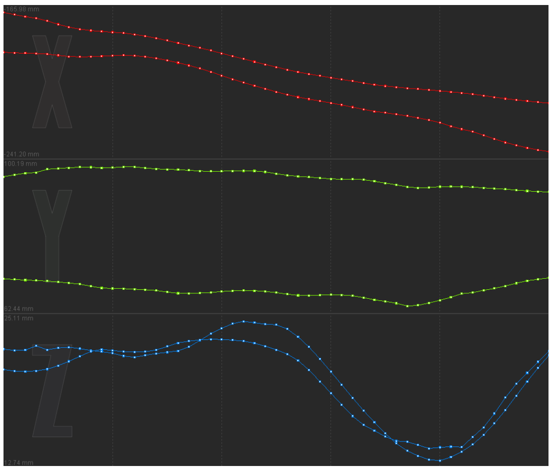
(b) Frame 699 (label swap).



(c) Frame 700 (gap).

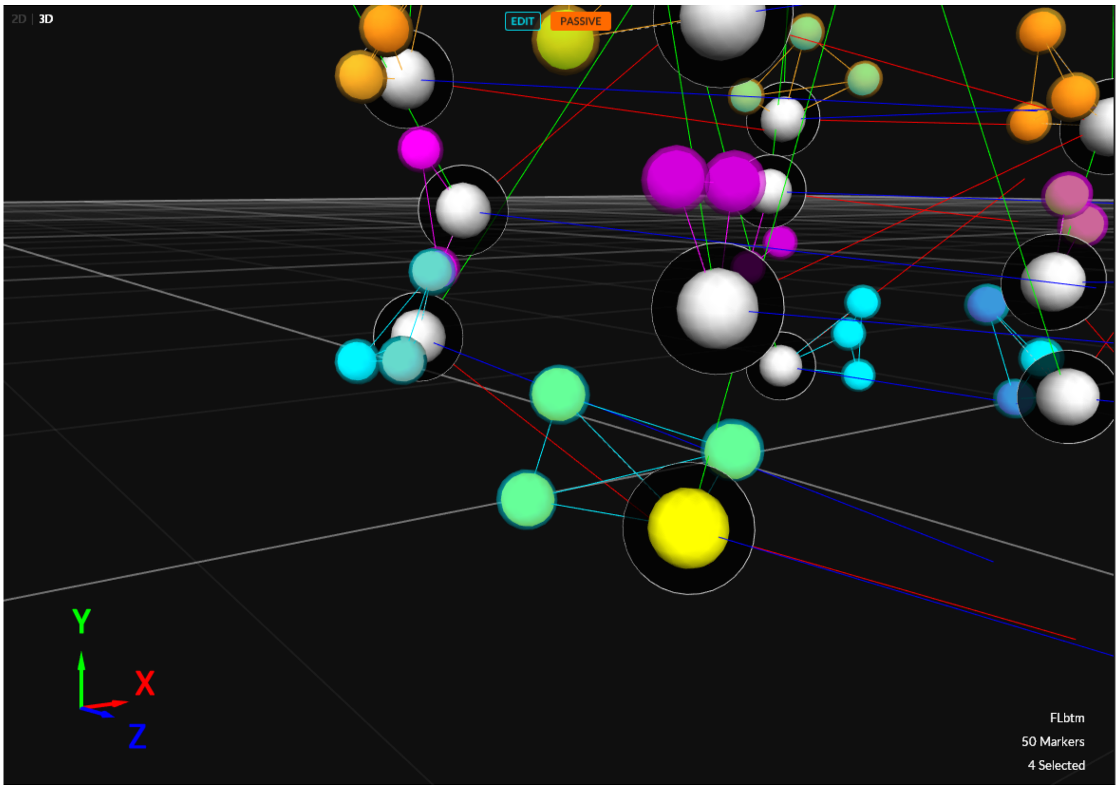


(d) Before post-processing.

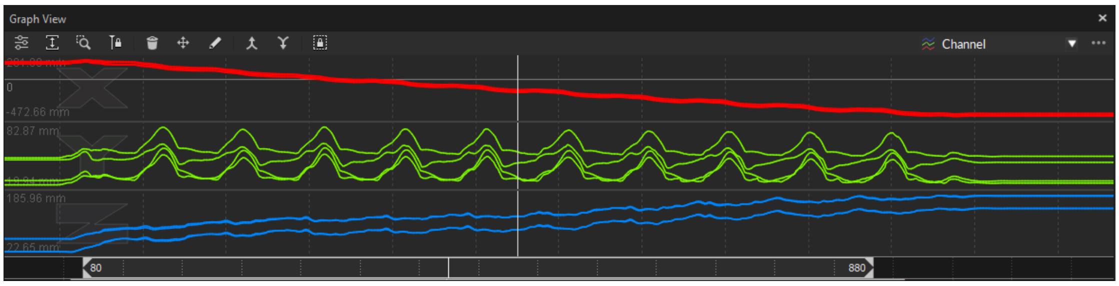


(e) After post-processing.

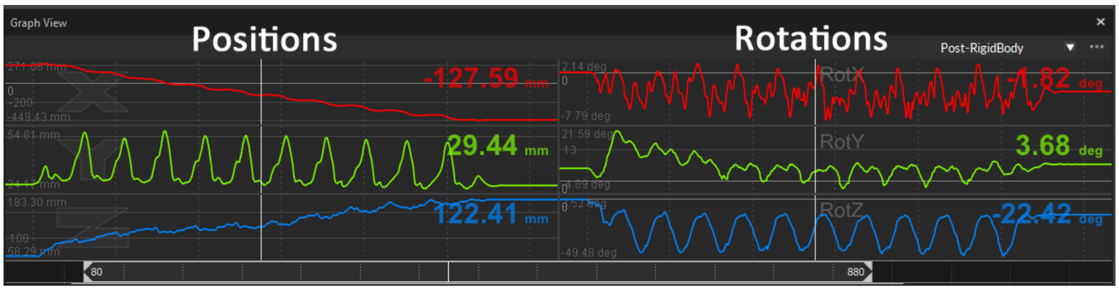
Figure 5.5: Correctly labelled markers in frame 698 (a). Swap of label 2 and 3 of the back-right-middle (BRmid) rigid body in frame 699 (b) and missing tracking data in frame 700 (c). The white sphere indicates the pivot of the rigid body that is set to marker 2. In frame 699, the expected marker position (transparent violet sphere) is not aligned with the actual marker position (yellow), due to to the label swap. Frame 700 lacks tracking data. Only the rigid body is shown at its expected position. The incorrect trajectories (d) are responsible for the swaps and gaps and are corrected during post-processing (e).



(a) Front-left-bottom (FLbtm) rigid body is selected in edit mode (large yellow sphere).



(b) x- (red), y- (green) and z-positions (blue) of the four FLbtm markers shown in Motive's Graph View.



(c) Position and rotation data of FLbtm rigid body. Solved data has been recomputed after post-processing.

Figure 5.6: Tracking results of markers and corresponding front-left-bottom (FLbtm) rigid body of the robot.

CHAPTER 6

Simulation

After the assembly of the robot and the acquisition of tracking data, it is time do develop a tool that is able to analyse the captured data and compare it with simulated data. This chapter deals with the implementation of the simulation, which is the second important part of the sandbox. It is essential for understanding the relationships between real-world and simulated data. Robot simulation is a challenging task, applying simulation results successfully onto the real robot even more so. The *reality gap* is the discrepancy between simulation and real-world outcome. Sometimes, satisfying simulations perform poorly in reality. Gaining an understanding of what makes simulations fail in real-world could help improve robot control.
In Section 6.1, the requirements of the simulation are discussed. Section 6.2 gives a short introduction to robot kinematics and Section 6.3 describes the concepts of forward and inverse dynamics, because they are fundamental for simulation applications. Since robot kinematics and dynamics are complex topics that fill whole books, only the overall idea behind these concepts is mentioned.
The final simulation is realized with MATLAB and Simulink® R2020a [simd]. Simulink offers model-based design and simulation for robotics and many other applications in electronics, signal processing, or computer vision. Simulated models are realized using graphical and textual programming. Simulink provides 3D visualisation of the robot model and also a 2D Data Inspector for simulated data. Before the Simulink simulation, there existed a different approach, which had to be abandoned because it did not deliver satisfying results in the designated time. Due to the effort that was also put into this approach, it is described briefly in Section 6.4. In Section 6.5, the creation of a 3D robot model for the Simulink simulation is described and in Section 6.6 and the following, the implementation of the simulation model in Simulink is explained including input data processing.

6.1 Requirements

The following requirements are imposed onto the sandbox simulation:

1. It should be possible to simulate different walking patterns for the robot.
2. The walking patterns should be compared to find clues why simulations behave differently.
3. Animation from the motion capture data should be possible.
4. Motion capture data and simulation results should be compared to analyse the differences between simulation and reality.
3. Simulation results should be visualized in 2D and 3D to facilitate comparison.
4. To close the sandbox cycle from robot $\rightarrow$ motion capture system $\rightarrow$ simulation $\rightarrow$ robot, simulated data should be exported and used on the real robot.

6.2 Robot Kinematics

Kinematics deals with the motion of rigid bodies in a robotic system, without considering internal or external torques and forces that are responsible for the motion [SK08]. It is essential for robot design and simulations, since it describes position and orientation of rigid bodies in space. Besides position and orientation, which is referred to as *pose*, also the derivatives of the pose, including velocity and acceleration, are considered [SK08]. Depending on the robot's topology, a robot can be defined as a kinematic chain that connects a rigid body to two other rigid bodies, except for the first and last rigid bodies in the chain that only have one neighbour. If a kinematic chain is open and branched, it is called a kinematic tree. Rigid bodies in a kinematic chain/tree are also referred to as *links*. The rigid bodies at the end of a kinematic chain/tree are called *end-effectors*. The connection between two rigid bodies is called a kinematic *joint*. The kinematic chain/tree consists of *coordinate reference frames* for each joint. A robot pose is therefore expressed by displacements of frames relative to some other frame [SK08].

6.2.1 Joints

A kinematic joint constrains the relative motion between two rigid bodies [SK08]. Common joint forms are prismatic, revolute, spherical and 6-DOF joints. A prismatic joint or sliding joint permits sliding motion of one rigid body relative to the other rigid body along a defined axis of the coordinate reference frame, usually the z-axis. This joint has therefore 1-DOF. A revolute joint also referred to as hinge joint allows rotation of one rigid body relative to another rigid body. The conventional axis of rotation is the z-axis, so the revolute joint also has only 1-DOF. Revolute joints belong to the most frequently used joints in real-world robots because they are easily realized with rotating

motors. A spherical joint can rotate one rigid body relative to another about axes in three different directions, which makes it a joint with 3-DOF. A spherical joint can be composed of three revolute joints, often arranged orthogonally to each other. However, this requires the addition of three virtual massless links with zero length, which can lead to computational difficulties. Also, if the axes of the revolute joints become coplanar, a singularity occurs, which does not happen with a spherical joint. The 6-DOF joint has no constraints, rotation and translation are possible in every direction. Rigid bodies connected with a 6-DOF joint are actually not jointed at all. 6-DOF joints are used for mobile robots to move freely in the world coordinate frame. For this, the robot's base frame, from which the other frames start out, is connected with a 6-DOF joint to the world coordinate frame. The base frame is also called a *floating base*, in contrast to a *fixed base*, where the robot cannot move from its initial location [SK08].
The puppy robot is a branched kinematic tree with a floating base. The tree structure is as follows:

- `body` (6-DOF)
  - `front-left-top` (1-DOF)
    - `front-left-middle` (1-DOF)
      - `front-left-bottom` (1-DOF)
  - `front-right-top` (1-DOF)
    - `front-right-middle` (1-DOF)
      - `front-right-bottom` (1-DOF)
  - tail (1-DOF)
    - `back-left-top` (1-DOF)
      - `back-left-middle` (1-DOF)
        - `back-left-bottom` (1-DOF)
    - `back-right-top` (1-DOF)
      - `back-right-middle` (1-DOF)
        - `back-right-bottom` (1-DOF)

6.2.2 Geometric Joint Representation

The links and joints with their respective reference frames define the geometry of a robot [SK08]. The reference frames could be located and oriented arbitrarily, however, due to consistency and computational efficiency, conventions have been established for fixing reference frames on links. The foundation of the conventions was laid by Denavit and Hartenberg [DH55] and there exist several adaptations. They all have in common that they use four parameters to position one frame relative to another one: the link length (a_i), the link twist (α_i), the joint offset (d_i) and the joint angle (θ_i). To make this work, the x-axis of one reference frame has to be perpendicular to and intersect the z-axis of the previous frame. This convention can be used for prismatic and revolute joints, or combinations of the latter, for modelling joints with higher DOF [SK08]. The original Denavit and Hartenberg convention places joint i between link i and $i+1$. The joint offset and angle (d_i and θ_i) are measured with respect to the $(i-1)$th joint axis. Thus, the joint axis and parameter subscripts do not match. Standard Denavit-Hartenberg parameters cannot be generalized and do not sufficiently describe the geometry of branched kinematic trees or multi-DOF joints. Adaptations of this convention counteract these problems and can be application dependent [SK08].

6.2.3 Forward and Inverse Kinematics

Forward kinematics defines the problem of finding relative positions and orientations of any two rigid bodies. The joint positions and link geometries are known. This problem is solved by computing the transformations between the reference frames and concatenating them. Using conventions facilitates the computations [SK08].
Inverse kinematics solves the opposite problem. The relative positions and orientations of any two rigid bodies are known and the joint positions have to be determined [SK08]. This involves the solving of a set of nonlinear equations that can have multiple or no solutions. Often, numerical methods, such as symbolic elimination or iterative methods are used. Closed-form solutions only exist for six DOF robots with special kinematic structures.
Important for robot control are the forward and inverse instantaneous kinematics or velocity kinematics. Forward instantaneous kinematics computes the angular and linear velocities of the rigid bodies, given the joint rates of motion, and inverse instantaneous kinematics computes the joint rates given the rigid body velocities. They can only be computed once the forward or inverse position kinematics problem has been solved [SK08].
In this context, the Jacobian matrix should be mentioned shortly because it is an important quantity in robot motion control and analysis. It is needed for the derivation of the dynamic equations and its analysis is used for determining singularities in robot configurations. The Jacobian represents an instantaneous transformation between the joint rates and the velocities of the rigid bodies. Its a $6 \times n$ matrix, n being the number of DOF (joints), where the upper $3 \times n$ matrix consists of the linear velocities of a given body and the lower matrix of the angular velocities [SHV06]. The Jacobian is used for the computation of the joint-space inertia matrix that is mentioned in the following section.

6.3 Forward and Inverse Dynamics

For simulation, control, and mechanical design, dynamics is an important concept. For position control, it is necessary to know the robot's dynamic properties, e.g. how much force needs to be applied to make the robot exert the desired motion [SHV06]. With the dynamic equations of motion, relationships are made between actuation and contact forces that act on a robot mechanism and the resulting acceleration and motion trajectories. Deriving these equations is difficult due to nonlinearities and the high number of DOF in a robotic system [SHV06]. The puppy robot, for instance, has 15-DOF. Forward dynamics computes joint accelerations from given joint actuator torques or forces. Inverse dynamics deals with the computation of joint actuator torques or forces from robot trajectories such as position, velocity, and acceleration. Inverse dynamics is needed for feedforward control to compute torques that are necessary for a motor to reach a defined position. Forward dynamics is used for simulation, to compute trajectories from torque input [SK08]. A basic canonical formulation of the equations of motion is given in the following set of differential equations [SHV06]:

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \tau_{\mathbf{g}}(q) = \tau \tag{6.1}$$

In this formulation, the coefficients are functions of $\mathbf{q}$ and $\dot{\mathbf{q}}$. $\mathbf{q}$ is an n-dimensional vector of joint positions. For example, for revolute joints, q_i is the angle of joint i. $\dot{\mathbf{q}}$ is the joint velocity and $\ddot{\mathbf{q}}$ is the joint acceleration. τ are force variables and n is the number of DOF of the robot. τ_g is the gravity vector. $\mathbf{H}$ is the $n \times n$ joint-space inertia matrix. The inertia matrix maps joint accelerations to joint torques/forces. For its computation, centers of mass, masses, and rotational inertias of the rigid bodies are needed. $\mathbf{C}\dot{\mathbf{q}}$ denotes the Coriolis and centrifugal forces, with $\mathbf{C}$ also being an $n \times n$ matrix.
To derive the terms in Equation 6.1, common methods are the Lagrange formulation and the Newton-Euler formulation. Both lead to the same final result. The Lagrange formulation treats the robot as a whole and uses a Lagrangian function, which is the difference between kinetic energy and potential energy. The Newton-Euler formulation takes each robot link and describes its linear and angular motion. These equations contain coupled forces and torques since links are connected to other links. After performing a so-called forward-backward recursion, all of the equations are eventually gathered to define the whole robot [SHV06].
Equation 6.1 is a representation of the dynamics of interconnected ideal rigid bodies. Even for simple robotic systems, this equation is already complex. Still, it is an idealization, which does not consider numerous other dynamic effects, e.g. friction at joints, contact forces, or the fact that a physical body is not completely rigid. Additional terms can be added to this equation to include these dynamical effects. Extending Equation 6.1 by adding contact forces for one exemplary contact point on the right side of the equation leads to [TSL]:

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \tau_{\mathbf{g}}(q) = \tau + \mathbf{J}^T f_n \mathbf{n} + \mathbf{J}^T \mathbf{D} \mathbf{f_d} \tag{6.2}$$

The new terms are the Jacobian matrix $\mathbf{J}$, which is evaluated at the contact point, the normal force $\mathbf{n}$, the magnitude of the normal force f_n, the matrix $\mathbf{D}$ with discretized Coulomb friction directions, and tangent forces along the friction directions $\mathbf{f_d}$ [SHV06, TSL].

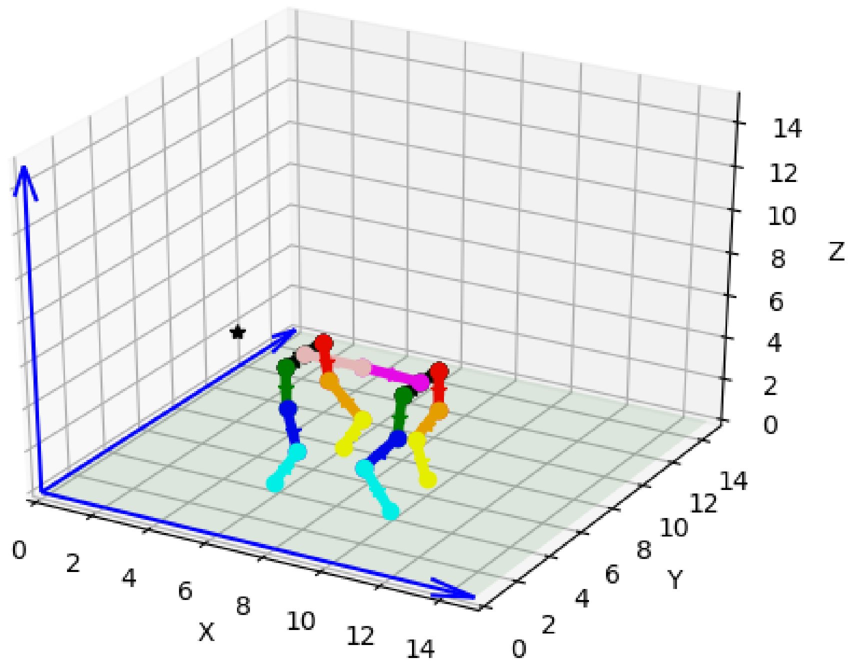
6.4 Initial (Discarded) Approach

Originally, it was planned to use a simulation, provided by a student from IST [Bha]. This simulation was programmed in Python 3 and computed forward and inverse dynamics and handled contact forces for simplified models made of rods, with simple inertias and masses. The models were defined according to the Denavit-Hartenberg convention, using prismatic and revolute joints. For the equations of motion, the Lagrange formulation was used, including contact forces as in Equation 6.2. To solve the forward and inverse dynamics problem, the equations were formulated as a linear complementarity problem (LCP), subjected to contact force constraints [TSL]. Complementarity problems are frequently used for modelling contacts. Lemke's algorithm was used to solve the LCP [ZD15].

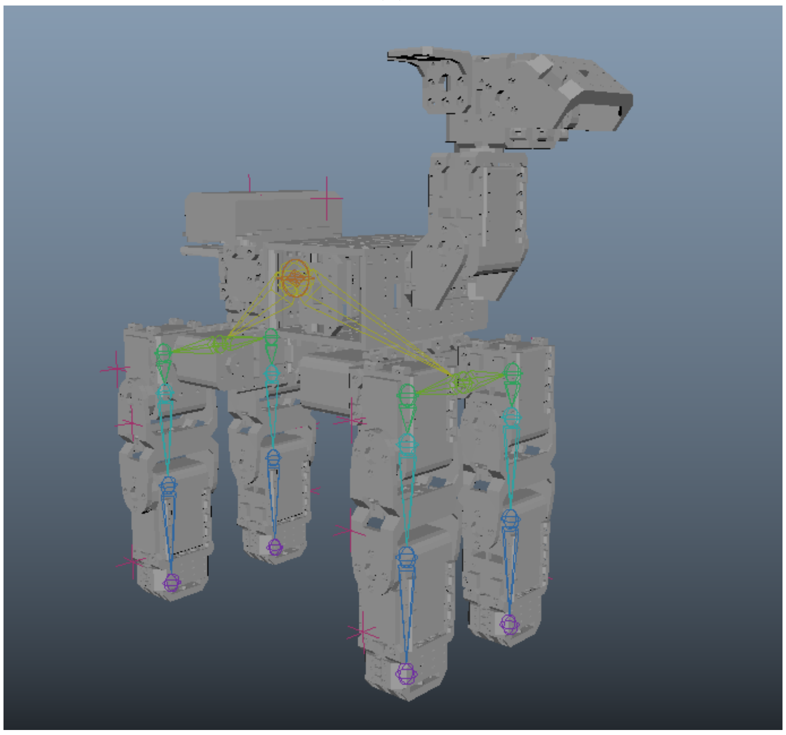
The rigid bodies of the puppy robot were modelled with rods and according to the Denavit-Hartenberg convention. It was mentioned already in Section 6.2 that this convention is not suited for branched kinematic trees like this four-legged robot. Massless dummy links with zero length had to be inserted between some links to guarantee that all revolute joints rotate around their local z-axis. The simulation did not consider the use of 6-DOF joints and was never tested with such complex robot geometry. When a makeshift 6-DOF joint was composed of three prismatic and three revolute joints, matrix singularities occurred. Debugging did not turn out fruitful, and without good knowledge of robotics, it can only be speculated if there were errors in the mathematical computations, or if the simulation simply was not designed for such complex robots. The implementation was based on concepts more suited for fixed base 6-DOF robot arms and not for 15-DOF legged robots with a floating base.

The simulation included a plain 3D visualisation that was not real-time and not suited for comparing and analysing data. So, it was envisaged to design a visualisation separated from the simulation that would make the comparison of different simulations easier. This visualisation would have included a 3D view for showing different robot motions, and a 2D graph view, for analysing motion trajectories. The JavaScript libraries Three.js, for 3D rendering, and d3.js, for the visualisation of graphs, would have been used together to realize this. Therefore, a rigged robot model was already created in Maya that could be animated and used in the visualisation. The software that came along with the ROBOTIS robot uses 3D models for visualizing robot motions. All the models were contained in an .assets file. From this software, the single robot parts could be extracted using a Unity Assets Bundle Extractor [uab]. The object files (.obj) from each robot part were obtained from the .assets file. With the parts, the robot could be rebuilt in Maya.
Figure 6.1a depicts this simple visualisation created in Python and Figure 6.1b shows the rigged robot model. Rigid body motion capture data was imported as animation into Maya. The model's rig was parented to the rigid body data and could be animated in this way.

Since the simulation did not work properly, this whole approach was discarded and it was decided to use MATLAB and Simulink [simd]. Simulink provides tools for physical



(a)



(b)

Figure 6.1: Simple visual output from the student's simulation, with model geometry, approximated with rods (a). Puppy model created with Autodesk Maya 2019 (b). The model has been rigged and can be animated with joint data from motion capture or simulation. The pink markers represent the rigid body pivot locations from the motion capture. The markers contain exported animation data. The rig has been parented onto the markers to move the model according to the marker animation.

modelling like the Simscape Multibody™ Library [simc], which offers a simulation environment for mechanical systems. Robots can be defined with joints, constraints, and forces. A 3D animation enables direct visualisation of the robot and simulated data can be inspected directly in 2D graphs. With this approach simulation, visualisation, and analysis would be combined in a single application.

6.5 3D Robot Model Creation

In this section, the construction of the 3D robot model is explained. The aim of the 3D model is to represent the real robot in the simulation and to provide similar physical properties. Such physical properties are, for instance, the geometry of the robot's rigid bodies, its masses, and its inertias.
Initially, as described in the previous section, it was tried to approximate the virtual robot model and use rods with simple inertias and centers of mass. The major drawback of this approach is that the simulation results coming from such a simplified model would likely perform poorly on the real robot. Comparing simulation with reality would be difficult since the simulation would be based on completely different physical properties. A more realistic model would be the model created in Maya. But also this model does not provide inertias and centers of mass.
The Simscape Multibody Library offers a plugin for computer-aided design (CAD) software that manages the import of assemblies into Simulink [plu]. The advantage of CAD software is that it automatically computes masses and inertias of parts and permits the definition of joints and constraints between parts. All this data can be used directly in Simulink.

Knowing this, it was decided to build a 3D robot model with the CAD software SolidWorks® [sol]. ROBOTIS provides 3D data for CAD software, including .step files of various robot components required for the puppy [stp]. The .step files with the robot geometry are imported into SolidWorks as *parts*. Individual parts can then be combined to an *assembly*, and assemblies can be further combined to even larger assemblies. Parts in an assembly are joined by using *mates*. The mates create geometric relationships between parts, such as parallel, coincident, or concentric, and define how many DOF one part has relative to its joined part. Mates can be established between features, like points, edges, faces, or circular shapes.
The provided .step geometries exhibit mostly round features. For the definition of revolute joints between two rigid bodies, concentric and coincident mates are established between two circular geometries. The only movement left for the geometries is rotation around the axis going through their common center. Figure 6.2a shows the 3D robot model designed in SolidWorks using concentric and coincident mates for modelling revolute joint constraints. The joined features are highlighted in orange and violet.

For the simulation, actually, two models have to be created. One model with revolute joints used for simulation and an additional model that can be animated with the motion capture data, such that a direct comparison between simulated movements and real-world movements can be made. The motion capture data includes xyz rotations for each rigid body. To replicate the exact movements, it is necessary that also the model's rigid bodies can rotate around all axes, needing spherical joint constraints. This is achieved by finding the centers of the rotation axes of two joined rigid bodies, and use coincident mates on those centers. Figure 6.2b shows the second model, with the rotation axes and the coincident point mates at the centers.

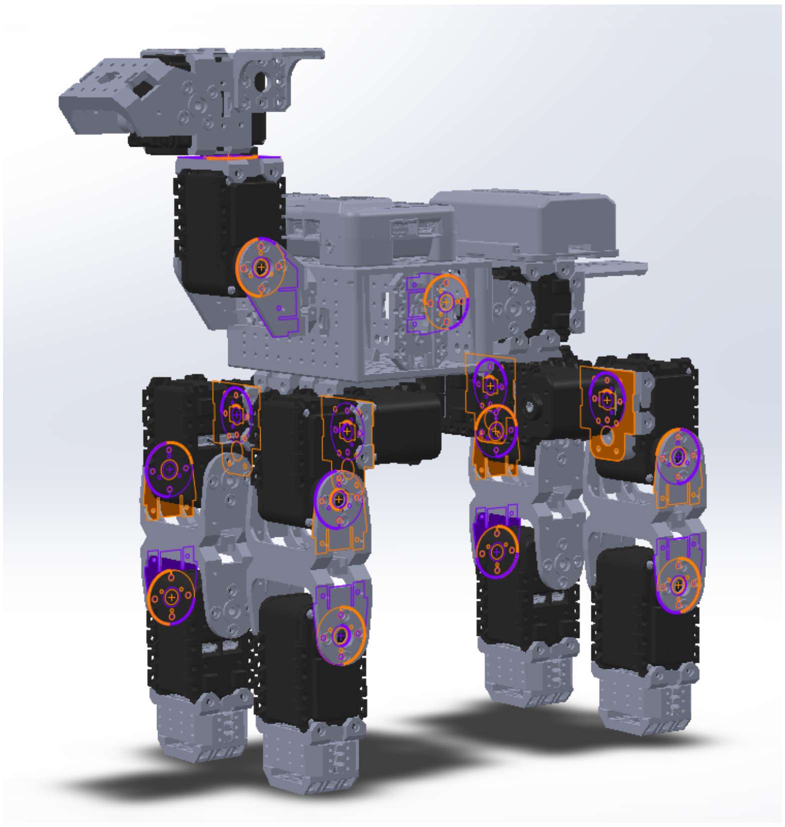
The masses, centers of mass, and inertias that SolidWorks automatically computes for each part are also not exact since the material properties of the real robot parts are not specified. Nevertheless, it is a decent approximation and much better than the initial one.

6.6 Simulink Simulation Model Overview

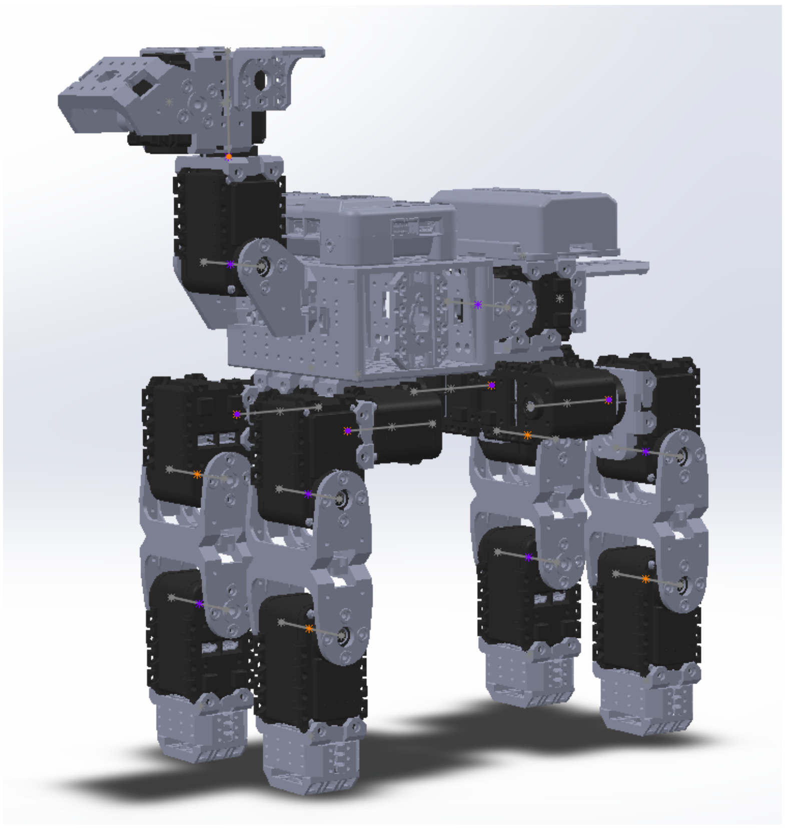
This section gives an overview of the Simulink simulation model before the following sections describe the model in more detail, starting with the preparation of appropriate input data, explaining the model design and finishing with a comment about solver settings.
Simulink is a graphical programming environment. Dynamic systems are realized with block diagrams. Blocks can consist of physical components, like geometry, or of functions or whole systems. A block can have input ports, output ports, or both, and has parameters that can be edited [simd]. Ports of individual blocks can be connected, which enables signal propagation through the system. Simulink offers a variety of block libraries for different functionalities. In this context, the Simscape Multibody [simc] and the Simscape Multibody Contact Forces [Mil] Library are of relevance, since they include all the blocks necessary for simulating mechanical systems like robots. They contain blocks for joints, solid geometries, or forces, among others [simc].

Simscape Multibody has a plugin for SolidWorks that enables the export of an assembly into an .xml file with detailed structure and attributes of the assembly. Geometry files (.step/.stl) of the individual parts are generated too. Those files can be imported with the MATLAB command `smimport('robot.xml')` and Simscape Multibody creates an equivalent model in Simulink (.slx), consisting of the necessary blocks [cad]. In an additional MATLAB script (.m), the attributes of the CAD model, like transformations, masses, or inertias are saved. Figure 6.3 shows the steps of the model export and import.

The basic blocks for the puppy robot model in Simulink are `Solid`, `Joint`, and `Rigid Transform` blocks. In the Solid blocks, the body geometry, the inertias, and masses of the parts are provided. Rigid Transform blocks provide frames with correct poses of the Solids and connect Solids with each other. The mates that were defined in the previous section translate into `Revolute Joint` and `Spherical Joint` blocks.
The Simulink model that was generated from the .xml file just provided a representation of the CAD model, which could not be simulated straight away because joint input was not specified and there were no contact forces involved yet.



(a) Concentric and coincident mates for modelling revolute joint constraints.



(b) Coincident point mates for modelling spherical joint constraints.

Figure 6.2: 3D robot model designed in SolidWorks, modelled with revolute joints, for simulation (a), and with spherical joints, for animation with motion capture data (b).

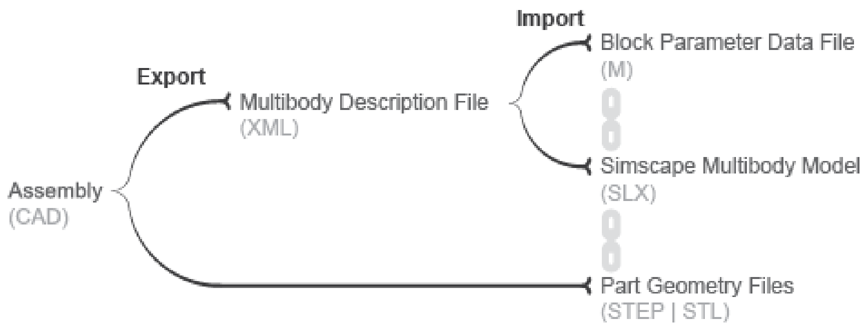


Figure 6.3: Steps of the CAD assembly export and import [cad].

In Section 6.5, a robot model with revolute joints and one with spherical joints was described. These models are identical apart from their joint definitions, so both CAD models are combined to one Simulink model that can switch between joint definitions. Thus, the Simulink model can be used for simulation as well as for motion capture animation. The difference is that for the animation, no contact forces are involved and the model basically only plays back the positional and rotational data from the motion capture.
In Figure 6.4, the top-level hierarchy of the Simulink model is shown. Different robot parts are packed into subsystems to maintain clarity. The `q_input` subsystem block provides the angle input for the joints. This is described in more detail in Section 6.7. The `robot_transform` block positions the robot in the world frame that is defined by the `World` block. The `ground_plane` and `gp_transform` blocks define a plane on which the robot is supposed to walk. Below the `World` block is the `Mechanism Configuration block` that specifies overall parameters like gravity and the `Solver Configuration` block that specifies solver settings for the simulation. The last two blocks and the `World` block are necessary for every Simulink model and have not been altered. The lines connecting the blocks are the signals. Thicker lines indicate bundled signals from or to a bus. A bus is useful for organizing models with many signals.
The `body`, `tail`, `front_left_leg`, `front_right_leg`, `back_right_leg`, `back-_left_leg`, `head_neck` specify the subsystems for the respective body parts of the robot, including Joint blocks for actuation and Rigid Transform and Solid blocks for the geometries. In the leg subsystems, also the contact force control is handled. That is why the leg subsystems in Figure 6.4 are connected with the `ground_plane` block. The handling of the contact forces is described in Section 6.9. The subsystems of the robot parts have an output port that propagates the simulation output data to the top-level hierarchy and collects it in a bus. The data that is gathered in the bus is logged and different simulation runs can be compared later. Logged are robot positions, joint angles, joint velocities, torque, and normal and friction contact forces. More about the simulation output is outlined in Chapter 7, where the results of the simulation are discussed.

Apart from the Simulink model file, `robot.slx`, there are a few MATLAB scripts that

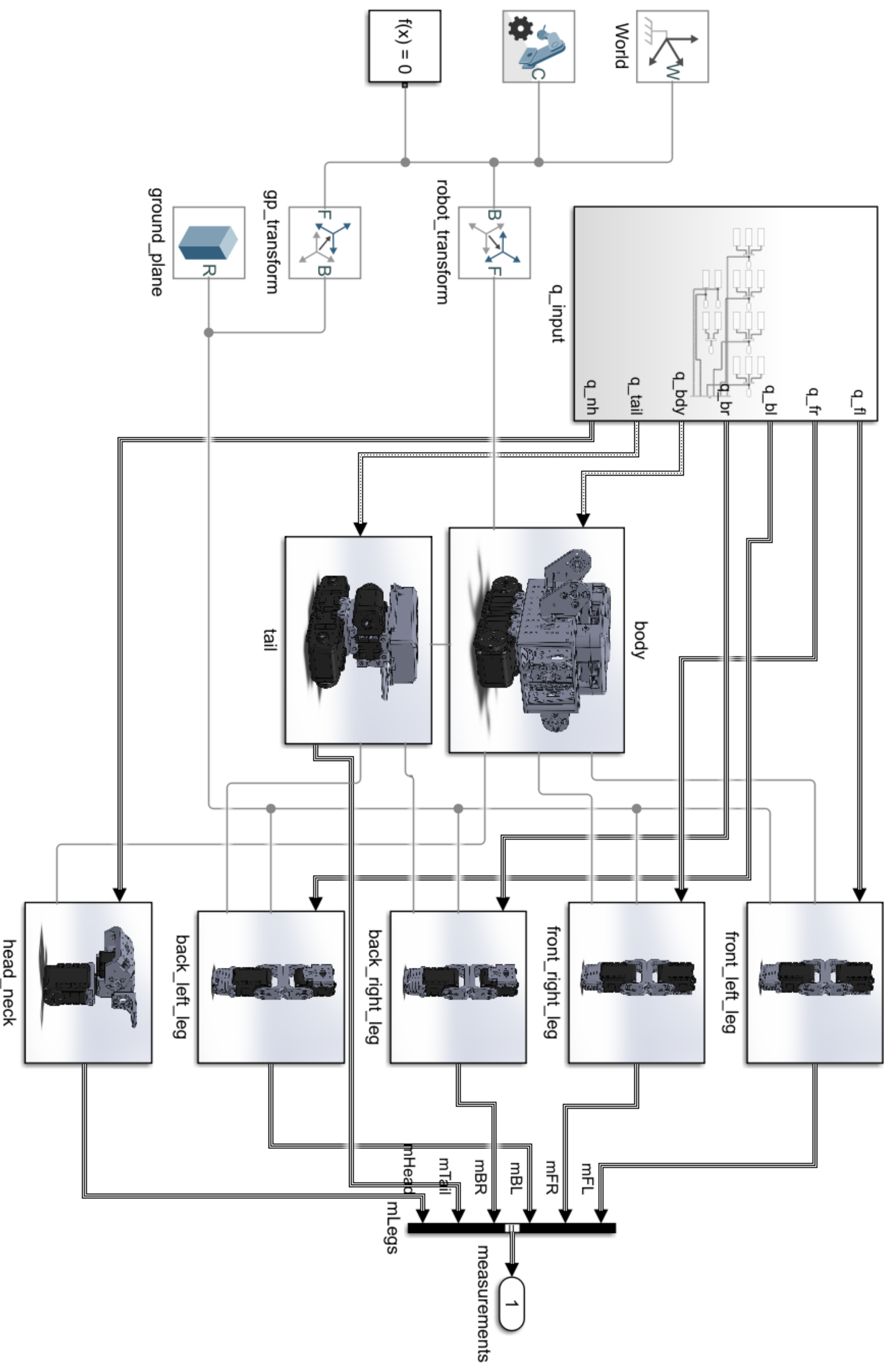


Figure 6.4: Simulink model at the top-level hierarchy. Components for different robot parts and data input are hidden in subsystems to reduce visual clutter.

are used for organizing simulation runs and parameters and prepare the data for the simulation:

- `simulation.m` is used for loading the robot model and the input data. Various simulation runs with different parameter settings can be performed.
- `robotParameters.m` specifies model parameters for different blocks, e.g. what type of actuation a joint block accepts (motion or torque).
- `setRobotParameters.m` updates the model if parameter names change or if new parameters are added.
- `importData.m` imports the motion capture data from the .csv file and saves it in MATLAB workspace.
- `loadData.m` formats the motion capture data that has been imported into the workspace, such that it can be processed by the model.
- `loadRobotisData.m` does the same as `loadData.m` but with a different dataset, obtained from the ROBOTIS software, which requires different processing.
- `interpolateData.m` is used by `loadData.m` and `loadRobotisData.m` and interpolates and filters data to obtain smooth trajectories for the model.

6.7 Input Data

Depending on whether the Simulink model is used for simulation or animation, input data is used differently. To keep it simple, a consistent data structure is created that contains all the information for both model variants. A MATLAB 1×16 `struct` (structure) array is used for this, named `dataSim`. `dataSim` contains a `joint` field with 16 entries for all robot joints. Each `joint` entry is again a `struct`, containing 18 fields with `timeseries` data. A `timeseries` object represents data values and corresponding time values. `dataSim(1).joint` accesses the 18 fields of the first joint entry. The 18 fields are described in Table 6.1. They contain position and rotation data for all axes and first and second derivatives. Derivatives provide velocity and acceleration and are needed for the solver. Depending on whether the model is used for simulation or motion capture animation, different data is needed. Only z-rotation derivatives and x-rotation derivatives (the top leg joints rotate around the x-axis) are currently required for simulation, but the other derivatives are included for the sake of completeness, and in case the model is extended in the future.

The alphabetical order of the joints in the `dataSim` struct is important for automatically loading the struct into the model. The order with corresponding abbreviations is: bdy (body), blb (back-left-bottom), blm (back-left-middle), blt (back-left-top), brb (back-right-bottom), brm (back-right-middle), brt (back-right-top), flb (front-left-bottom), flm (front-left-middle), flt (front-left-top), frb (front-right-bottom), frm (front-right-middle), frt (front-right-top), head, neck, and tail. In Figure 6.5, the `q_input` subsystem is shown, with the blocks containing the joint data from `dataSim`. Output ports, numbered from 1-7, propagate the signals out of the subsystem back to the top-level hierarchy, and from there to the respective subsystems.

`px`	pos. x-axis (6-DOF joint, anim)	`rx`	rot. x-axis (anim, sim top joint)
`py`	pos. y-axis (6-DOF joint, anim)	`ry`	rot. y-axis (anim)
`pz`	pos. z-axis (6-DOF joint, anim)	`rz`	rot. z-axis (sim, anim)
`d1px`	1st derivative x pos.	`d1x`	1st derivative x rot. (sim top joint)
`d2px`	2nd derivative x pos.	`d2x`	2nd derivative x rot. (sim top joint)
`d1py`	1st derivative y pos.	`d1y`	1st derivative y rot.
`d2py`	2nd derivative y pos.	`d2y`	2nd derivative y rot.
`d1pz`	1st derivative z pos.	`d1z`	1st derivative z rot. (sim)
`d2pz`	2nd derivative z pos.	`d2z`	2nd derivative z rot. (sim)

Table 6.1: Description of the 18 timeseries fields for each joint, required for simulation (sim) or animation (anim).

Before the data is loaded into `dataSim`, it has to be preprocessed. There are two different datasets that are used for simulation, the motion capture .csv file, containing the positions and rotations of the rigid bodies, and a .mat file, containing angles from the ROBOTIS software. These are the angles the real robot has been programmed with. The data from the ROBOTIS software contains rotation data only.

The motion capture data is imported from the .csv file into the MATLAB workspace. There, it is represented as a matrix of size `nFrames` × 85, `nFrames` being the number of exported frames from Motive. The first column contains time steps, the other 84 columns contain the position and rotation data for 14 rigid bodies (excluding the neck and head that are not tracked). The data is already ordered as required by `dataSim`. Position and rotation data are separated. Position data is only required for the animation with the motion capture data and only for the 6-DOF body joint, which is considered the floating base of the robot and positions the robot in the world frame.
Rotations in the rigid bodies accumulate from the top to the bottom of the kinematic tree because the motion capture system has no notion of the kinematic relationships. Thus, from every rigid body rotation, the rotation of the previous rigid body in the kinematic tree has to be taken out.

It has to be taken care of that the coordinate frames of the rigid bodies from the motion capture and the coordinate frames of the joints from the simulation are oriented the same. It is beneficial to consider this already during tracking, and orient the rigid bodies accordingly in Motive.
The position data for the 6-DOF body joint is offset, such that the robot always starts from the same position when simulating, and not from the position the robot was at during tracking.
The captured rotation data was sampled at 120 frames per second, so there is a data

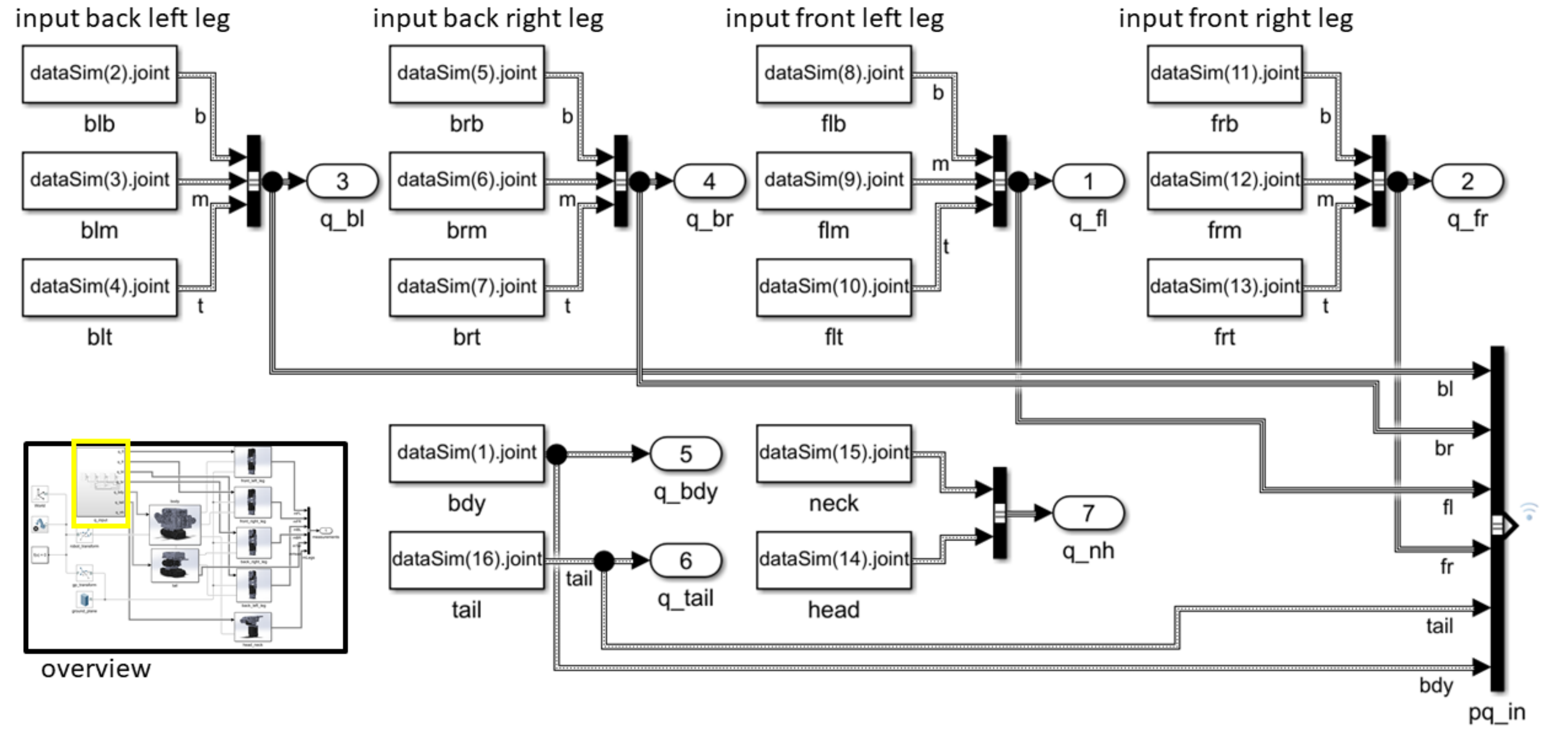


Figure 6.5: The `q_input` subsystem provides the data from `dataSim` for the model. Every joint has its own block that contains position, rotation, and derivative data. Output ports are numbered from 1-7. They propagate the signals out of the subsystem. `pq_in` is a Bus Selector block that is used for logging the collected input signals.

point every 0.0083 seconds. The solver needs at least a time step ≤ 0.001 and additionally requires time derivatives up to the second order. Using MATLAB's `cubicpolytraj` [cub] function, cubic interpolation is performed with a step size of 0.0001 on the (smoothed) rotation trajectories. The function also computes the velocities and the accelerations of the input data. Due to the large number of data points and the small step size, the derivatives are extremely noisy and exhibit sharp changes, which negatively affect the simulation results. Hence, the derivatives are filtered to reduce the noise and to create smoother trajectories. MATLAB's Savitzky-Golay filter is used for this [sgo]. A comparison between filtered and unfiltered data can be seen in Figure 6.6.

The ROBOTIS dataset only consists of one motion cycle with five time points and the corresponding rotations. The data matrix has a size of 5×16, where the first column contains the time and the other 15 columns contain the joint rotations. The data is looped to get a longer walking pattern. It is also interpolated and smoothed before it is loaded into the `dataSim` struct. The ROBOTIS data is considered as a simulation reference dataset that has optimal angle trajectories that can be compared with motion capture data.

6.8 Joint Blocks and Variant Subsystems

After the input signals have been loaded into the model, they are propagated to the joint blocks in the model's subsystems. In Figure 6.7a, the `front_left_leg` subsystem is shown, representatively for all other leg subsystems because they are identical in structure. Also, the `tail` and `head_neck` subsystems contain the same functionality albeit adapted accordingly. Therefore, they are not specifically mentioned here.
The `front_left_leg` subsystem has three input ports: Port `j` connects the geometry from the `body` subsystem to the joint of the current subsystem. Port `q` provides the rotation data for the joints and `ground` provides the ground-plane geometry for the force control between foot and ground that happens in the `FLbtm` subsystem (see the following section). The measurement subsystem collects the output data from the joints, which is computed during the simulation. The joints' output consists of the rotation input signal, velocities, and computed torques. One output port propagates the measured signals back to the top-level. Normal and friction forces are measured too.
Before the rotation data can be used by the joints, it has to be converted from a unitless Simulink signal to a physical signal. The unit of the physical signal is specified in degrees. This conversion is done in the `SPS` subsystems. `SPS` stands for Simulink Physical Signal. The `SPS` subsystems are variant subsystems, this means that different model configurations can be selected that are used for the simulation. At the top of Figure 6.7b, the inside of the variant subsystem that is highlighted in yellow in Figure 6.7a is shown. Ports in a variant subsystem contain further subsystems that are not connected by signals. The subsystem that is active in the variant subsystem is highlighted, while the inactive one is greyed out. The subsystems inside the variant subsystems provide different implementations for the same purpose. The blue colored blocks in Figure 6.7b are `Simulink Physical Signal Converter` blocks. One variant supplies the

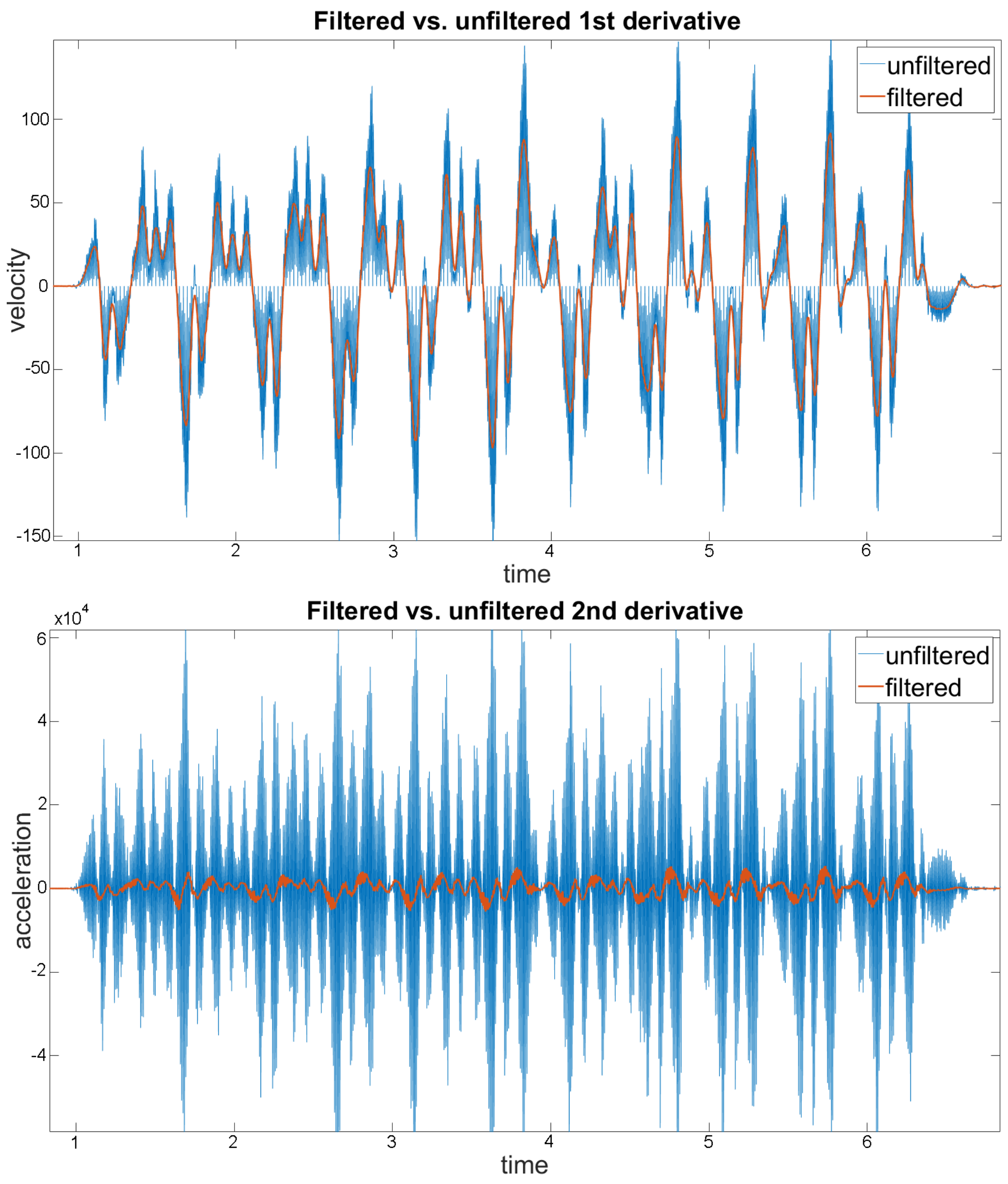
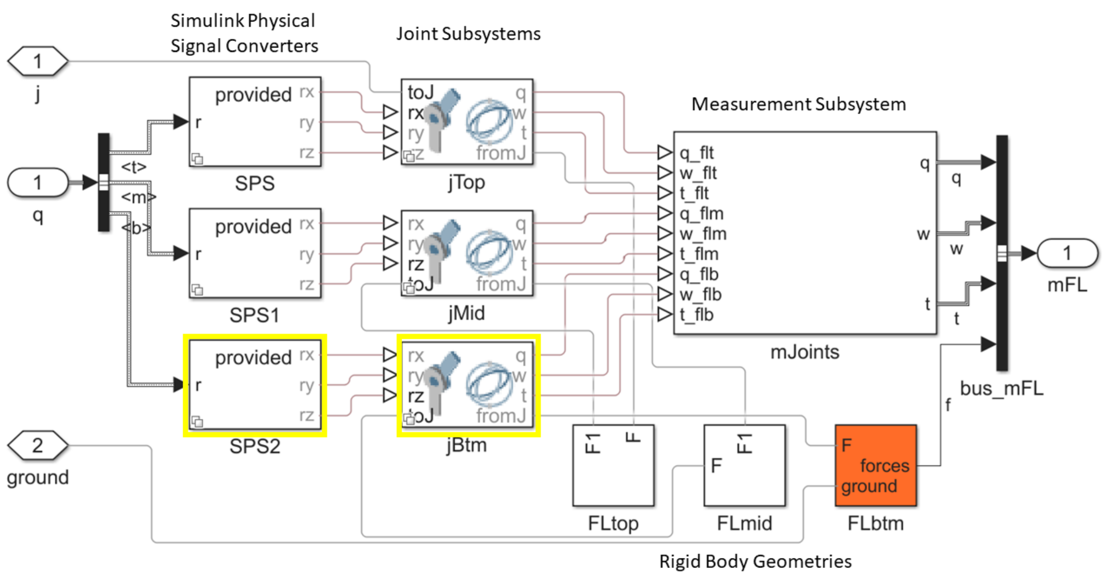
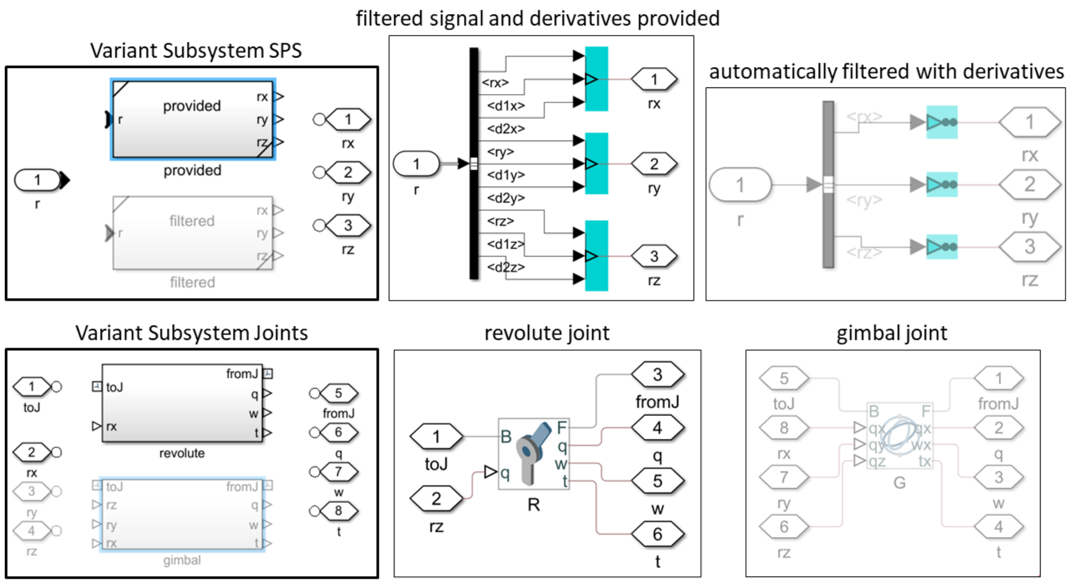


Figure 6.6: Velocity and acceleration trajectories filtered and unfiltered.



(a) `front_left_leg` subsystem with signal converters, joint, geometry and output measurement subsystems.



(b) Variant joint and SPS subsystems (left) provide selection between two different model configurations (middle and right).

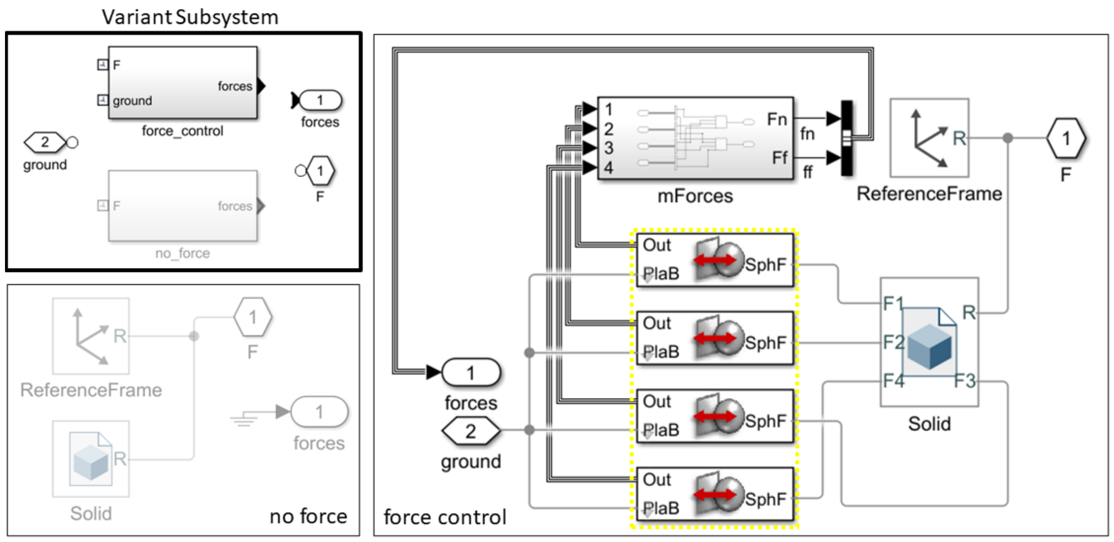
Figure 6.7: `front_left_leg` subsystem with highlighted SPS and joint variant subsystems (a). Detailed view of variant subsystems (b).

blocks with the interpolated and smoothed rotation and derivative data. The other variant only needs the rotation data and filters the data automatically with a low-pass filter. It also computes derivatives. The drawback is that the automatically filtered data has a slight time offset that does not occur with the self-computed filtered data.
At the bottom of Figure 6.7b, another variant subsystem for the joint configurations is shown. As has been mentioned before, the model can be used for simulation or simple playback of the motion capture data. If the simulation variant is active, the model uses revolute joints that represent the realistic range of motion for the robot. If the animation variant is active, the focus is on the exact reproduction of the tracked motion. Therefore, a 3-DOF gimbal joint is used that is provided with the complete xyz rotation data from the motion capture. The difference between a gimbal and a spherical joint is that the gimbal joint needs Euler angles as input and the spherical joint needs quaternions.
The `body` subsystem features the same variant subsystems, with the difference that a so-called *bushing joint* is used for the variant joint controls. A bushing joint is similar to a 6-DOF joint, but the rotations are again represented by Euler angles. For the simulation variant, the position and rotation of the robot are computed by the simulation. If using the animation variant, the bushing joint is completely supplied with position and rotation data.

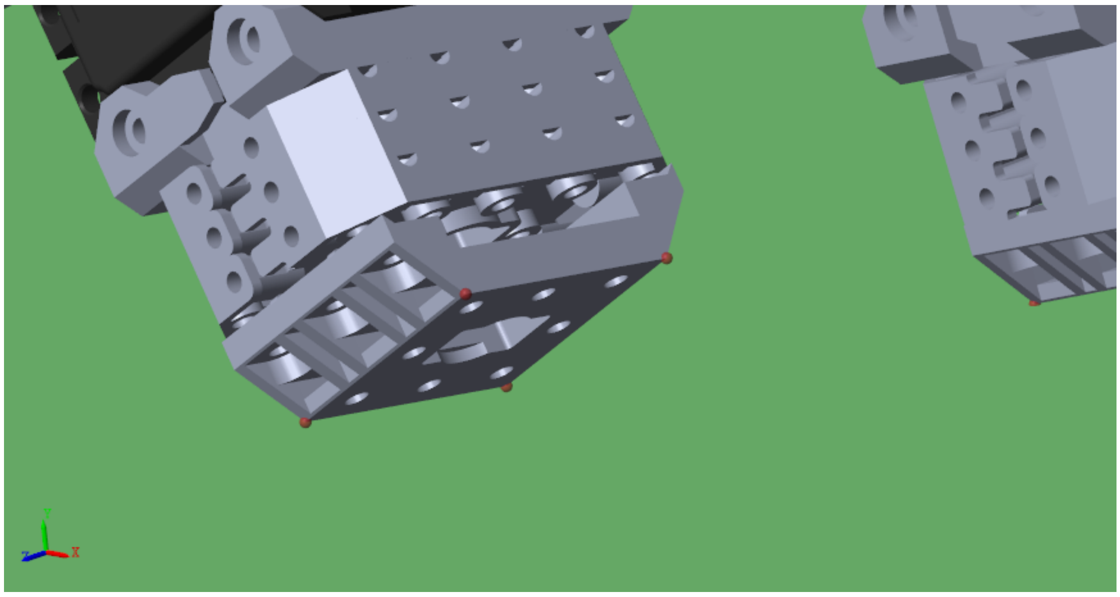
Currently, the model can only be simulated with motion actuation [act]. This means motion is provided in form of joint angles (rotations) and the corresponding force/torque is computed automatically. Thus, the simulation calculates inverse dynamics.

6.9 Handling Contact Forces

Contact forces occur if the bottom geometry of the robot's legs touches the ground plane. Simscape Multibody Contact Forces Library [Mil] is used to model simplified contact forces between plane and sphere geometries. Again, a variant subsystem has been used to switch between simulation with contact forces and animation without contact forces. The structure of the variant subsystem is depicted in Figure 6.8a. The `Solid` block contains the robot geometry that has to deal with the contact forces. The contact forces are realized with `Sphere to Plane Force` blocks. Instead of computing the contact forces at every point where the robot touches the ground, they are only evaluated at the four corners of the bottom leg geometry. This is shown in Figure 6.8b. At the corners, spheres have been mounted. Between those spheres and the ground plane, contact forces are computed. The output of each contact force block is computed friction and normal forces, which are gathered for logging. The subsystem has an output port that propagates the logged forces to the top-level and two input ports. One is for the ground plane geometry that is provided for the contact force blocks. The other one (denoted `F`) is just the connection of the bottom geometry to the rest of the robot's leg. In the `Sphere to Plane Force` blocks, the following linear contact force law is used that implements a



(a) Variant subsystem for contact force control.



(b) Contact forces are implemented between the spheres and the ground plane. The spheres are mounted at the bottom of the robot's legs.

Figure 6.8: Handling of contact forces in the Simulink model.

spring-damper to resist penetration [for]:

$$F_N = \begin{cases} K \cdot \delta_{pen} + D \cdot v_{pen} & \delta_{pen} > 0, v_{pen} > 0 \\ K \cdot \delta_{pen} & \delta_{pen} > 0, v_{pen} < 0 \\ 0 & \delta_{pen} \leq 0 \end{cases} \tag{6.3}$$

F_N is the normal force that is exerted along the direction of penetration. K and D are the spring stiffness and damping coefficient of the force law. δ_{pen} is the relative penetration and v_{pen} is the contact velocity. With decreasing penetration, the damping force is zero [FL16, for]. Spring stiffness K is set to 10^4 N/m (default) and damping coefficient D to 10^2 Ns/m (default). With the normal force and an additional friction coefficient μ, the friction force law F_f is composited [fri]:

$$F_f = F_N \cdot \mu \qquad\qquad \mu = f(v_{poc}) \tag{6.4}$$

v_{poc} is the velocity at the contact point between sphere and plane. μ is a function of this velocity. The friction coefficient is defined as follows [fri]:

$$\mu = \begin{cases} v_{poc} \cdot \frac{\mu_s}{v_{th}} & v_{poc} < v_{th} \\ \mu_s - v_{poc} \cdot \frac{(\mu_s - \mu_k)}{0.5 \cdot v_{th}} & v_{th} \leq v_{poc} \leq 1.5 \cdot v_{th} \\ \mu_k & v_{poc} > 1.5 \cdot v_{th} \end{cases} \tag{6.5}$$

μ_s and μ_k are the static and kinetic friction coefficients, respectively. v_{th} is the velocity threshold in m/s that is needed to overcome static friction and make the sphere slide along the plane under the influence of kinetic friction. The static friction coefficient is set to 0.7, and the kinetic friction coefficient to 0.5. The velocity threshold is set to 0.001 m/s. These are the default values for the `Sphere to Plane Force` blocks and they have not been changed.

6.10 Model Solver

By calling `sim('robot')` in the MATLAB command window or in a script, the model simulation starts. At successive time steps, the states of the dynamic system are computed over a defined time span. Information for the computation is provided by the parameters in the model blocks. This information is gathered in a set of ordinary differential equations (ODEs) that are solved by Simulink *solvers*. When creating new models, a solver is automatically suggested by Simulink based on the model characteristics. For physical simulations, Simulink offers the `ode15s` solver. `ode` stands for ordinary differential equation, `15` indicates that the maximum order of numerical differentiation formulas (NDFs) can be varied between first- and fifth-order formulas. It is recommended to start with a maximum order of two for the NDFs. `s` means that the solver can solve stiff differential equations [cho, var]. Stiffness means that solutions vary slowly, but there are also close solutions that vary rapidly. Because of that, time steps have to be very small to get satisfactory results [Mol04]. The Simulink model is considered a stiff system, due

to its dynamics (contact forces). `ode15s` is a variable-step continuous implicit solver. It reduces step size when model states change rapidly and increases step size when model states change slowly. This is useful for the detection of zero crossings that are caused by discontinuities of the model dynamics, e.g. when contact forces act on the legs. The solver then reduces the step size to capture these dynamic changes. This increases accuracy but also slows down the simulation [how]. An implicit continuous solver offers more stability for oscillatory behaviour than an explicit solver. This is beneficial for stiff problems. But it also is computationally more expensive [com].
There was not enough time to dive deeper into the solver selection and the parameter tuning of the solver, so mainly the default settings for the `ode15s` solver have been used since they delivered satisfying results. The comparison of different solvers and solver settings is left for future work (Chapter 8).

CHAPTER 7

Results

In this chapter, the final results of this work are presented. As a primary result, the ready-to-use sandbox at its current development stage is summarized. In the previous three chapters, the components of the sandbox have been explained separately in detail. Now, the sandbox as a whole is discussed once more, providing a compact finishing overview of the components, what data is required and produced by the different sandbox components and how the interaction between the components is managed. It is outlined which of the initial goals that were set in Chapter 1 have been reached and what is left for further development (Chapter 8).
A motion pattern of the puppy robot is analysed and compared to simulated data, to provide an example of what is currently possible with the sandbox.

7.1 The Resulting Sandbox

7.1.1 Software and Hardware

For the motion capture system, six OptiTrack Prime 41 [pria] infrared cameras are chosen due to their high tracking accuracy. An OptiTrack system is preferred to a Vicon system because it is less expensive and easier adaptable and extendable while delivering equally good performance. Motive:Tracker Version 2.2.0 [motb] is used for managing the tracking of the robot.
The 3D robot model used for simulation is generated with the CAD software SolidWorks 2019 [sol]. The simulation is realized with MATLAB and Simulink Version R2020a [simd], using Simscape Multibody [simc] for representing robot joints and constraints. With the Simscape Multibody Link Plugin [plu] the CAD model is imported into the Simscape Multibody environment. The Simscape Multibody Contact Forces Library Version 20.1.5.1 is used for approximating robot-to-ground contact forces [Mil].
The robot that is examined with the sandbox is from ROBOTIS [robc] and is actuated

with 15 Dynamixel servomotors [dyn]. The robot controller is reprogrammed with Embedded C in the Eclipse IDE for C/C++ Developers Version 2019-12 (4.14.0) using the STM32F10x Standard Peripherals Library Version 1.0 that is included in code examples provided by ROBOTIS [fir, stm].

7.1.2 Capabilities of the Sandbox

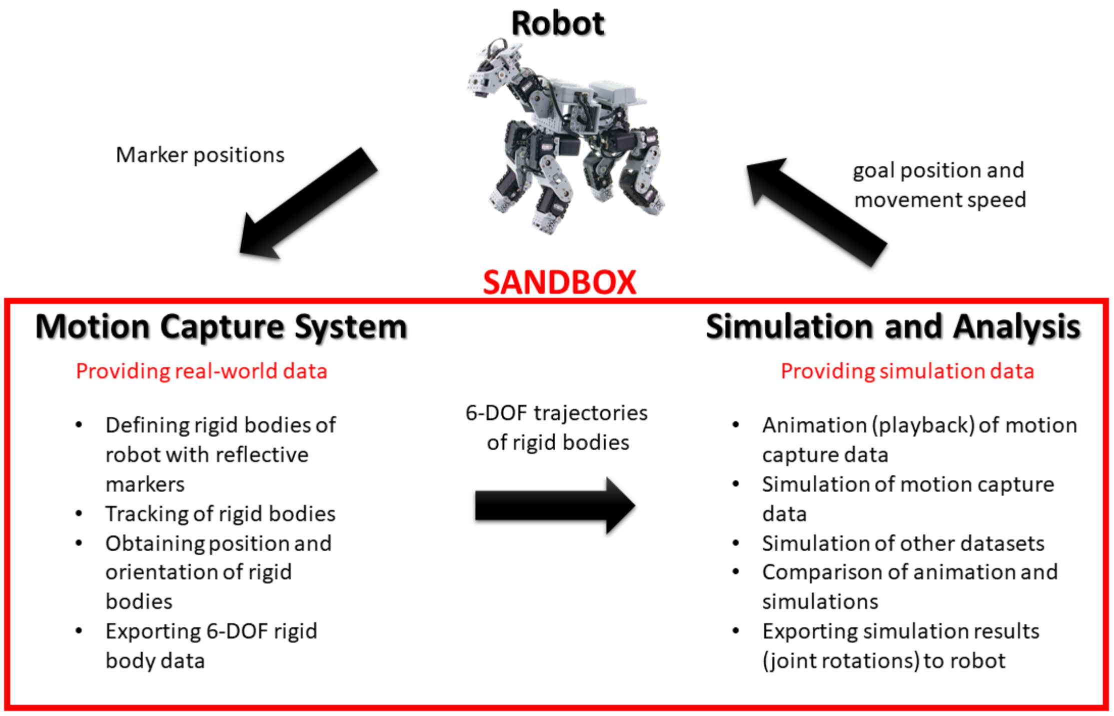

Robot
Marker positions
goal position and movement speed
SANDBOX
Motion Capture System
Providing real-world data
• Defining rigid bodies of robot with reflective markers
• Tracking of rigid bodies
• Obtaining position and orientation of rigid bodies
• Exporting 6-DOF rigid body data
6-DOF trajectories of rigid bodies
Simulation and Analysis
Providing simulation data
• Animation (playback) of motion capture data
• Simulation of motion capture data
• Simulation of other datasets
• Comparison of animation and simulations
• Exporting simulation results (joint rotations) to robot

Figure 7.1: Resulting sandbox design [robc].

.

It is possible to track a small-sized robot and gather its rigid body positions (x, y, z) and orientations (roll, pitch, yaw) at a frame rate of 120 Hz. This data is exported to a .csv file that is read into a matrix in MATLAB. This matrix is processed to actuate a 3D robot model in Simulink. The data can be used to either replicate the motion of the real robot or to compute simulated motion for the 3D model. The replicated and the simulated motion can then be compared to each other or to simulations from other datasets. Interesting in this context is the deferring robot's body position at a certain point in time between replicated and simulated motion, or between two simulated motions. Simulations are currently generated by letting MATLAB compute inverse dynamics from known joint angles taking contact forces into consideration. The computation of inverse dynamics results in torque values for the joints, which could be applied to the motors of a real robot. However, the servomotors of the puppy robot do not support torque actuation, which is why the computed torques are currently neglected and only the

resulting simulation is observed and used for comparison.
Further, it is possible to convert robot poses to a format that is understood by the Dynamixels of the real robot. This conversion is necessary when loading simulated motion to the real robot, to see how it performs in reality.
In Figure 7.1 the sandbox is depicted once again, showing the interaction between the components and their usage.

7.1.3 Interaction Between the Components

Providing data in a convenient way such that interaction between each component stays simple and clear is a crucial part of the sandbox design. Data transmitted between components consists of robot poses at given time points. The following conventions have been established to keep datasets consistent inside the sandbox:

- Datasets are treated as matrices, where the rows represent the number of poses and the columns represent the rigid bodies.
- Rigid body data are sorted alphabetically by their abbreviated names: body (bdy), back-left-bottom (blb), back-left-middle (blm), back-left-top (blt), back-right-bottom (brb), back-right-middle (brm), back-right-top (blt), front-left-bottom (flb), front-left-middle (flm), front-left-top (flt), front-right-bottom (frb), front-right-middle (frm), front-right-top (frt), head, neck, tail.
- If time points for each pose are included in the dataset, they occupy the first column of the matrix, followed by the rigid bodies starting with the second column.
- **Representation 1:** Poses for a rigid body can consist of 6-DOF data, where each rigid body occupies six columns. The first three columns contain the x-, y-, and z-component of the position, and the last three columns contain the x-, y- and z-components of the rotation. This representation is relevant when replicating the captured motion in Simulink using 3-DOF or 6-DOF joints. The first column contains time points.
- **Representation 2:** Poses can also consist of just the z-axis rotation component. This is sufficient for actuating the 1-DOF revolute joints connecting the rigid bodies, as the rigid body positions are inherently defined by the robot's kinematic structure. This representation is suitable for simulating the robot in Simulink. The first column also contains time points.
- The unit for time points is seconds, for positions, it is centimeters, and for rotations, it is degrees.

The interaction between the motion capture system and the simulation tool is unidirectional. The tracked data is transmitted to the simulation tool as explained in Representation 1 above. Datasets that do not come from the motion capture system

are preferably depicted according to Representation 2. Such a dataset is, for instance, provided by ROBOTIS and used as a reference that can be compared to the dataset coming from the motion capture.
Interaction between the robot and the sandbox is bidirectional. The robot provides 6-DOF data through attached markers and receives motion data from the simulation tool. For simulation-to-robot communication, the motion data is separated into three arrays that are processed by the microcontroller. The first array contains the goal positions (angles) of the Dynamixel motors for all poses. The first array entry contains the number of poses, the following $n \times 15$ entries the goal positions, where n is the number of poses. The second array contains the moving speed for each Dynamixel motor that is needed to reach a goal position. The speed array is composed exactly like the goal position array. A third array comprises the time differences between poses. This is necessary so the controller knows how much time it has to give the Dynamixels to reach their positions. Time differences are given in milliseconds, the speed in revolutions per minute, and the positions in degrees. Speed and position values have to be mapped to the interval $[0, 1023]$ in order to be used by the controller.

7.1.4 Limitations of the Sandbox

Currently, the sandbox does not provide machine learning algorithms for training the 3D robot model. So, it is not yet possible, to automatically generate new robot motion that could be tested on the real robot. The functionality to load simulated data to the robot is present since it is also used to load already existing robot motion to the robot, e.g. the provided motion data from ROBOTIS that has been used for tracking.

7.2 Comparison of Real-World and Simulated Data

With the help of the sandbox, a motion pattern of the puppy robot is analysed and discussed.

7.2.1 Input Datasets

ROBOTIS provides a walking motion for the puppy robot, which has been used for the comparison of real-world with simulated data. In the following, the ROBOTIS walking motion it is referred to as ROBOTIS dataset. The ROBOTIS dataset represents ideal robot poses. The puppy robot is actuated with the ROBOTIS dataset and its movements are tracked. From this motion capture, another dataset is obtained, which represents the actual poses the robot manages to take in the real-world, where physical constraints affect its motion. Such constraints are, for instance, the friction between ground and feet or friction inside the Dynamixel motors. The dataset generated from tracking is referred to as mocap (short for motion capture) dataset.

In Figure 7.2, the ROBOTIS and the mocap datasets are depicted. They show rotations over time of the robot's rigid bodies at their corresponding joints. It can be seen that

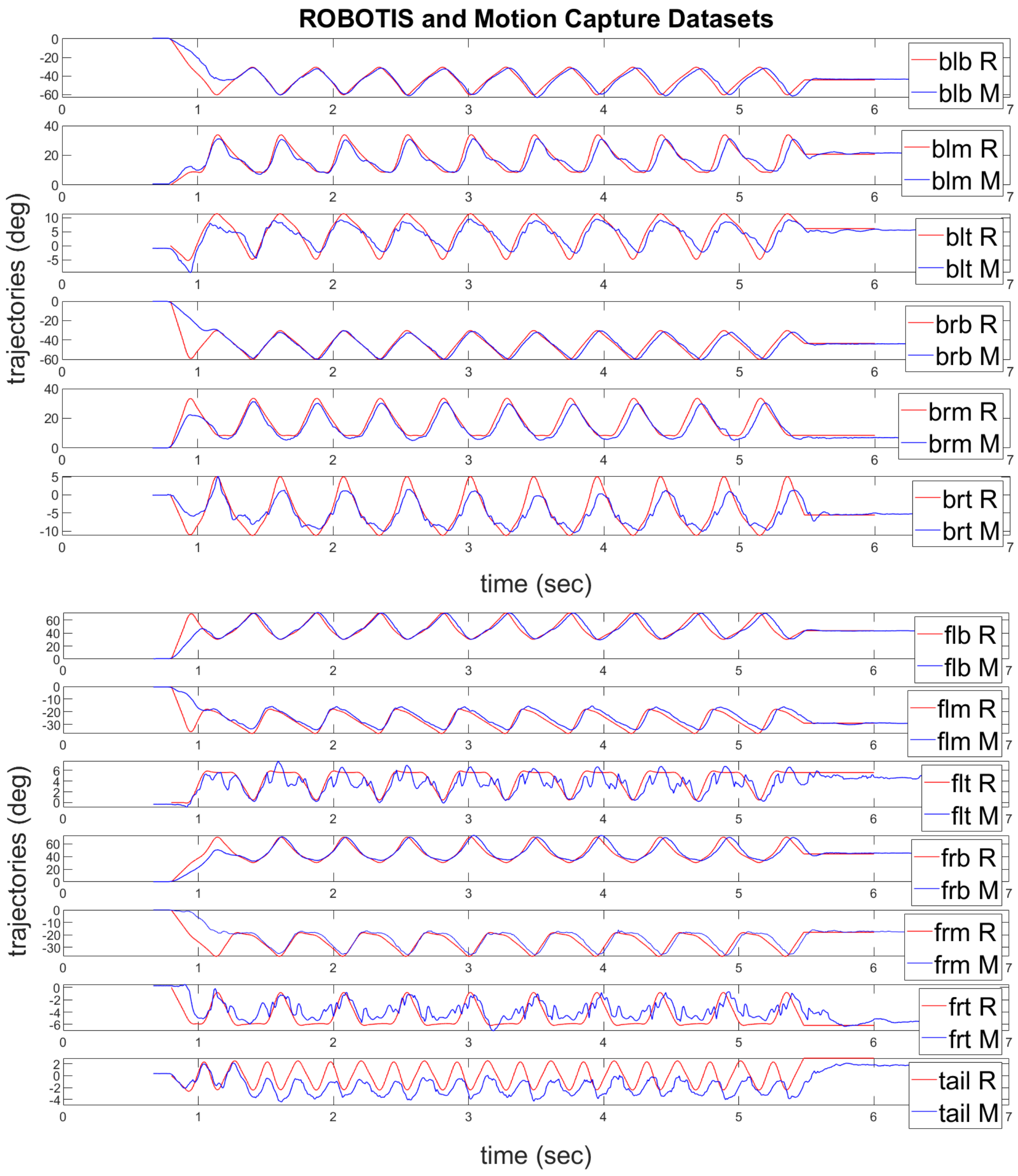


Figure 7.2: Angle trajectories from ROBOTIS (red, R) compared to trajectories tracked with the motion capture system (blue, M). Step size: 0.0001 (10^{-4}).

the mocap dataset is much noisier. This is clearly visible at the top rigid bodies of the legs, especially at the front-top (flt, frt) rigid bodies. Assumptions could be made that the impact of the feet on the ground are responsible for the jitter at the top rigid bodies. This impact is stronger for the front legs, as they contribute more to moving the robot forward, which is why the jitter for the front-top rigid bodies is more present. Also, the rotations of the tail are significantly smaller in comparison to the ROBOTIS dataset. This could be because the tail joint holds together the front and the back part of the robot. So, the forces that act upon this motor might prevent it to fully execute the rotations. This probably also applies to the top rigid bodies.
Taking a look at the start of the trajectories, it is noticeable that the mocap dataset does not always reach the peaks defined by the ROBOTIS dataset. This is because the ROBOTIS poses are looped. The timespan the motors need to go from their neutral pose to their first pose is longer for some motors than the timespan they need to go from their last pose to their first pose. The timespan to reach the first pose has been calculated from the last pose. Therefore, some motors do not fully execute the rotation to their first pose, when starting from the neutral pose.
It is also visible that the mocap dataset lags slightly behind as time goes on. This is probably because the milliseconds that pass between poses for the motors have to be rounded to integer values, and these inaccuracies accumulate with time.

In the previous chapter (Section 6.10), the MATLAB solver ode15s for ordinary differential equations, which is used for the simulations, has been mentioned already. The order of the normal differentiation formulas is set to 2. ode15s is a variable step solver for stiff problems. The average step size for the solver is around 10^{-4}, so the ROBOTIS and the mocap datasets are interpolated to have this precision. Higher precision would increase computation time without significantly improving the simulation, lower precision would reduce the quality of the simulation results.
Apart from the angle trajectories for the joints, the solver requires first and second derivatives for the physics simulation. MATLAB can compute these derivatives automatically when converting the input signals to physical signals for the joint blocks (see Section 6.8) and filters the signals with a low-pass filter. Filtering is essential to reduce the noise in the derivatives. The filtering that is applied by MATLAB offsets the data by a small time interval. In general, this is not problematic, but to avoid this time offset, another own input filtering is implemented, which does not exhibit a time offset (see Section 6.7). The advantage of the self-implemented filtering is that parameters such as filter order and window size can be adjusted.

7.2.2 Simulation Results

Simscape Multibody provides a Mechanics Explorer [mec] to visualize the robot model and its simulations in 3D. It also offers a tree view to explore the hierarchy of the robot model. The viewport of the Mechanics Explorer is shown in Figure 7.3. For this work, it is mainly used to check if simulations deliver visually acceptable results before simulation results are examined in more detail.

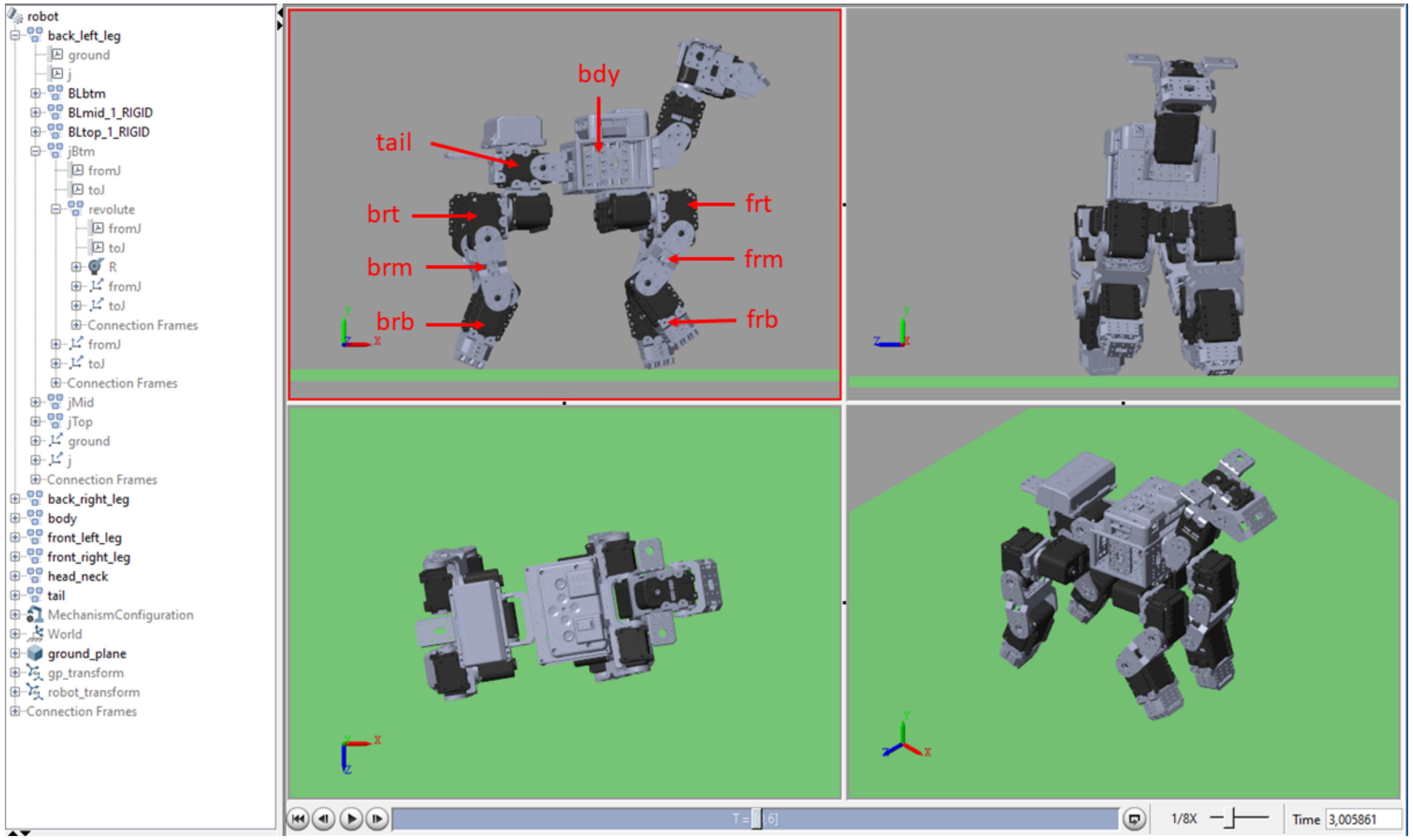


Figure 7.3: In the Simscape Multibody Mechanics Explorer simulation results can be visualized for the puppy robot from different viewpoints. Some rigid bodies have been named exemplarily. On the left is a tree view for exploration of the robot model hierarchy.

From the ROBOTIS and the mocap dataset, four simulations and one playback animation that represents the real-world motion are computed. For the playback animation, the complete mocap dataset with all the rotation data (for the gimbal joints) and the position data for the body of the robot is used. Two of the four simulations are created using only the z-rotations (x-rotations for top leg joints) of the mocap dataset for actuating the revolute joints. The other two simulations are created with the ROBOTIS dataset. For both datasets, the automatic derivative computation and the filtering of MATLAB is used as well as the self-implemented version. The effects of this *input variation* on the simulation results are examined too.
The animation and the simulations are compared with respect to position and orientation of the robot's body (bdy) in the time interval $[0, 6]$. Time is measured in seconds.
The simulation (or animation) results are named according to their input datasets as follows:

- animation or anim (playback, complete rotation data and body positions)
- robotis M (MATLAB filtering and derivatives)
- robotis (own filtering and derivatives)
- mocap M (MATLAB filtering and derivatives, x- and z-rotations only)
- mocap (own filtering and derivatives, x- and z-rotations only)

In Figures 7.4, 7.5, and 7.6, simulation results from the ROBOTIS and the mocap datasets are shown together with the motion capture animation, which represents the real robot's trajectories. The position and rotation components of the robot's main body (bdy) are computed during the simulation. All other robot parts are actuated with the angle trajectories and computed derivatives. In all figures, the thicker blue line represents the trajectories from the motion capture animation. The slight time offset that occurs due to MATLAB's filtering method, is well visible in Figure 7.4. For the ROBOTIS dataset, MATLAB's filtering and the own filtering method deliver very similar results. Because the ROBOTIS trajectories are ideal and noiseless, also the simulation output is more regular than the simulation output from the mocap dataset.
On closer inspection, it can be noticed that the beginning of the simulated y-position and the y-rotation trajectories differ from the animation trajectories. This is because the robot is positioned slightly above the ground plane and is set down by gravity once simulation starts. The small positive z-rotation is because the robot's back legs are a bit shorter than the front legs, so when the robot touches the ground with all legs, its body is tilted a little backwards.
The simulated robot walks in the direction of the positive x-axis in a right-handed coordinate system. Taking a look at all the x-position trajectories in Figure 7.6, it can be seen that the simulated robot walks on average 4.45 cm less far than the real robot. The final y-positions exhibit a average difference of about -0.22 cm, and the final z-positions vary between 1.89 cm and -4 cm. The final x-rotations in Figure 7.6 vary between 0.74 and

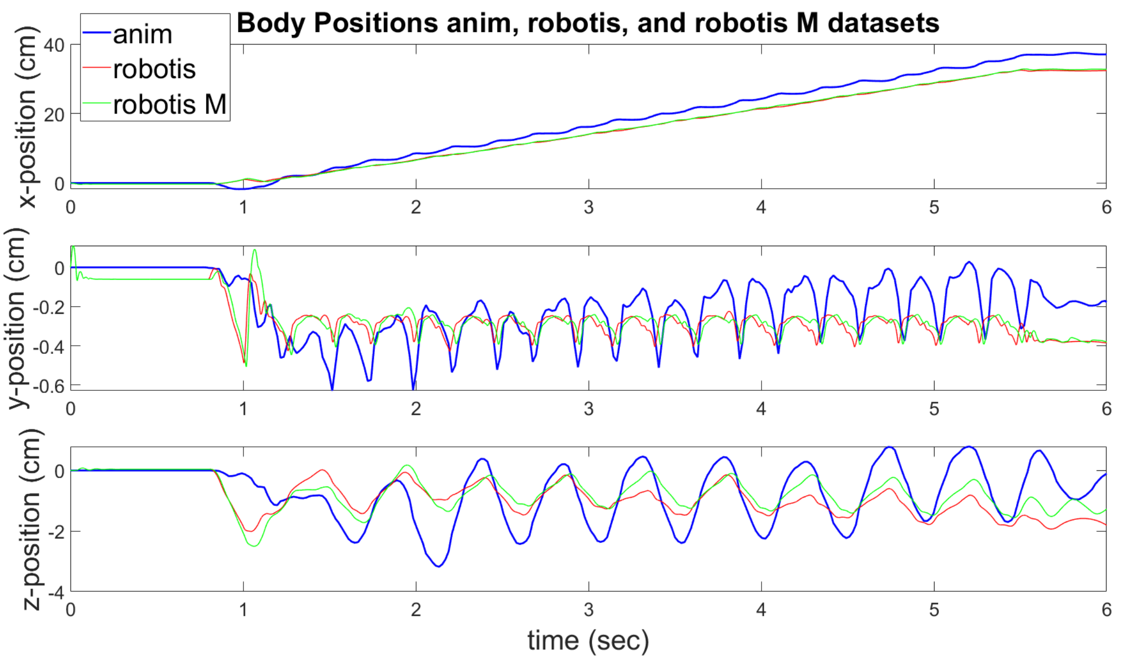
-1 degrees, and the final z-rotations between 0.83 and -0.16 degrees. Examining the final y-rotations, it can be seen that the simulation, which uses the automatic MATLAB filtering (mocap M, pink trajectory), has a right twist of -10.88 degrees, unlike the other simulations that show an average left twist of 4.82 degrees.

In general, the simulation results are very different compared to the trajectories of the real robot. Similarities in the trajectories of the real robot and the mocap-simulated robot are identifiable, especially at the beginning, however, they deviate quickly.
A reason for this variety is, among others, the amount of filtering that is applied to the derivatives. In Figure 7.7, the first derivatives (velocities) filtered with varying strength are shown for the front-left-bottom joint. Because of the large number of datapoints, the derivatives are extremely noisy and filter window sizes need to be large, to remove the higher frequencies and provide an over-all smooth signal. The size of a filter window has to be odd. Even a window size of 91 still results in noisy peaks. Much larger window sizes (e.g. 1001, 2001) manage to smooth out the peaks significantly. This amount of smoothing is comparable with the automatic MATLAB filtering. In Figure 7.8, simulated body positions and rotations are depicted resulting from derivatives filtered with varying strength. Based on these results, it cannot be said that more filtering improves the simulations. It can be observed that for window sizes 1001 and 2001 the filtered velocities are similar, the resulting position and rotation trajectories are not. It seems that the simulation is prone to reacting to such small changes.
Apart from the derivatives, there are many parameters in the simulation that could be tuned to see if they deliver simulation results that are closer to the real robot's movements. Such parameters include kinetic and static friction coefficients or contact stiffness and contact damping for the force control between robot and ground. Also, the internal mechanics of the joints have been neglected so far, e.g. how much torque is necessary to move the joint by a unit angle, or to maintain a constant velocity. For this, the real robot's motor specifications would have to be taken into account too. Improving the simulation is not part of this thesis and left for future work.

The comparison of the joint trajectories from the ROBOTIS and the mocap datasets (Figure 7.2) shows that reality is far from ideal. However, due to the ROBOTIS and mocap datasets, there exists a way to compare simulation with reality and close the sandbox loop (simulation $\rightarrow$ reality $\rightarrow$ simulation). With the ROBOTIS dataset, a simulation from ideal joint trajectories is created. These ideal joint trajectories are used to actuate the real robot. The resulting mocap dataset provides information on how the ideal trajectories perform in reality, providing a direct comparison with the simulation. The mocap trajectories are also used to create another simulation, which is not based on ideality but on reality instead. This makes it possible to also examine the changes that occur when real data is simulated.

After having analysed the robot's motion pattern and the simulation results, the question of suitable performance metrics arises that could be used. Absolute robot body positions between the real robot and the simulated robot could be compared. However, small errors at the beginning of the simulation add up over time and eventually result in large

distances. As has been seen in the comparison above, positions already vary largely when changing filtering methods. Therefore, comparing absolute positions between the simulated robot and the real robot is not useful. In practice, it is often much more important that the real robot maintains a certain direction, that it covers a certain distance without falling, or manages to walk safely over rough terrain. Regarding the simulation results, it makes sense to measure covered distance of simulation and real robot and use this as a performance metric.
Taking once again a look at the body positions in Figure 7.6, where all simulation results are shown together with the real robot's trajectories, it can be seen that the simulations generated by the ROBOTIS dataset cover more distance and are closer to the real trajectories than the simulations from the mocap dataset. Among the simulations from the ROBOTIS dataset, the dataset with the MATLAB filtering (robotis M) performs slightly better than the dataset with the own filtering (robotis M).
The robot's position (in cm) at the beginning of each simulation run is $[0, 0, 0]$. The robot walks along the x-axis. Deviations from a straightforward course show up in the z values. The real robot's position at the end of the run is at $[37.01, -0.18, -0.10]$. The robot simulated with the robotis M dataset finishes at $[32.70, -0.38, -1.28]$, and the robot simulated with the robotis dataset reaches $[32.35, -0.38, -1.81]$.
This shows that the robot manages to walk in simulation almost as far as in reality. This is already a successful outcome, given the fact, that the simulation parameters have not yet been optimized. Also, the motion of the real robot is stable, and despite not looking like the simulated motion, it is a satisfying result.



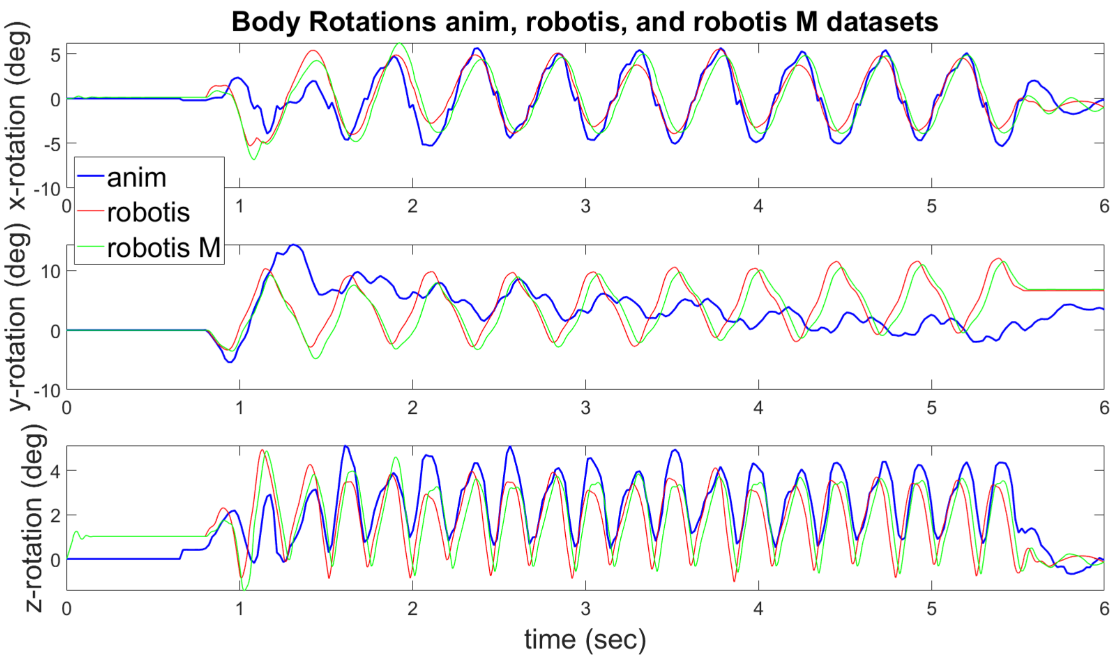
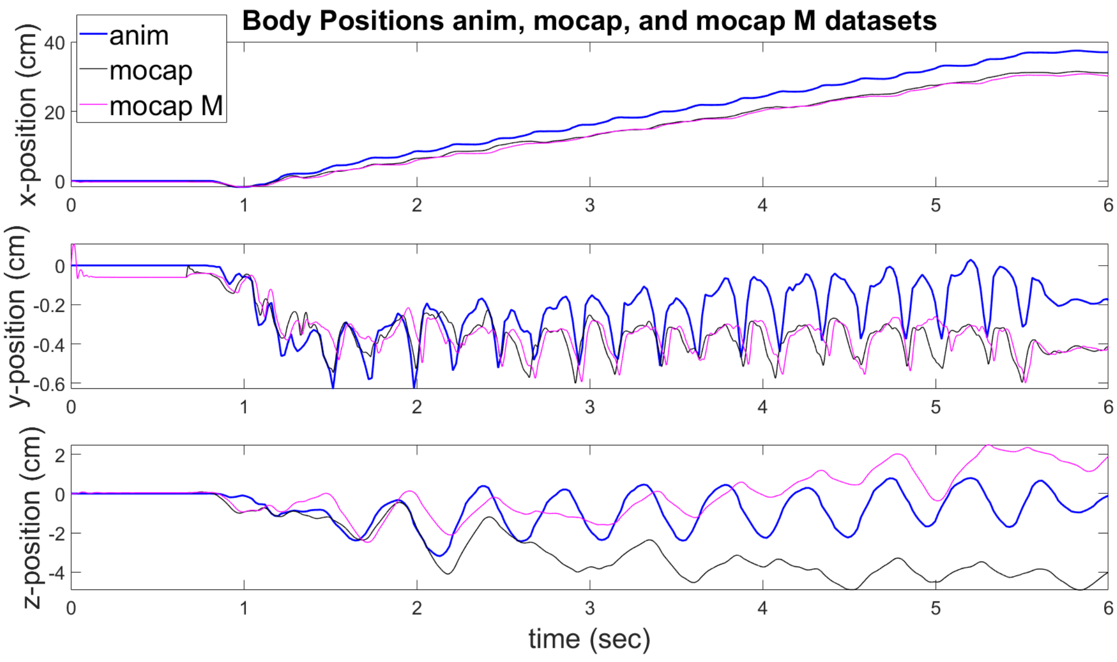


Figure 7.4: Simulation results from ROBOTIS dataset with MATLAB (robotis M) and own filtering (robotis) and derivatives compared to the real robot's trajectories.



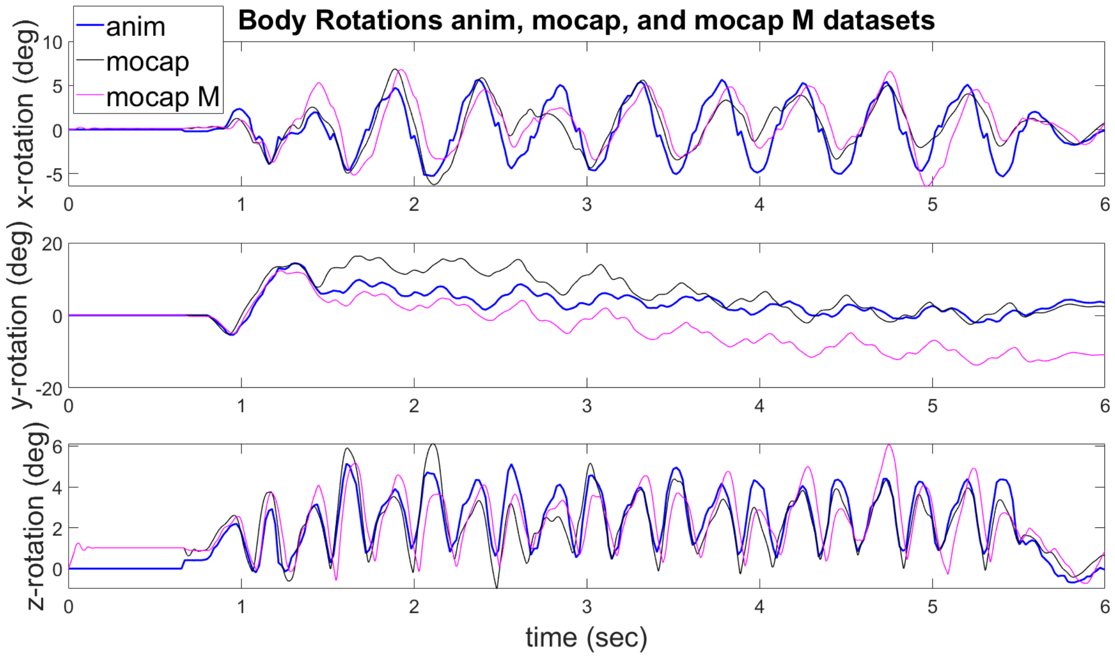
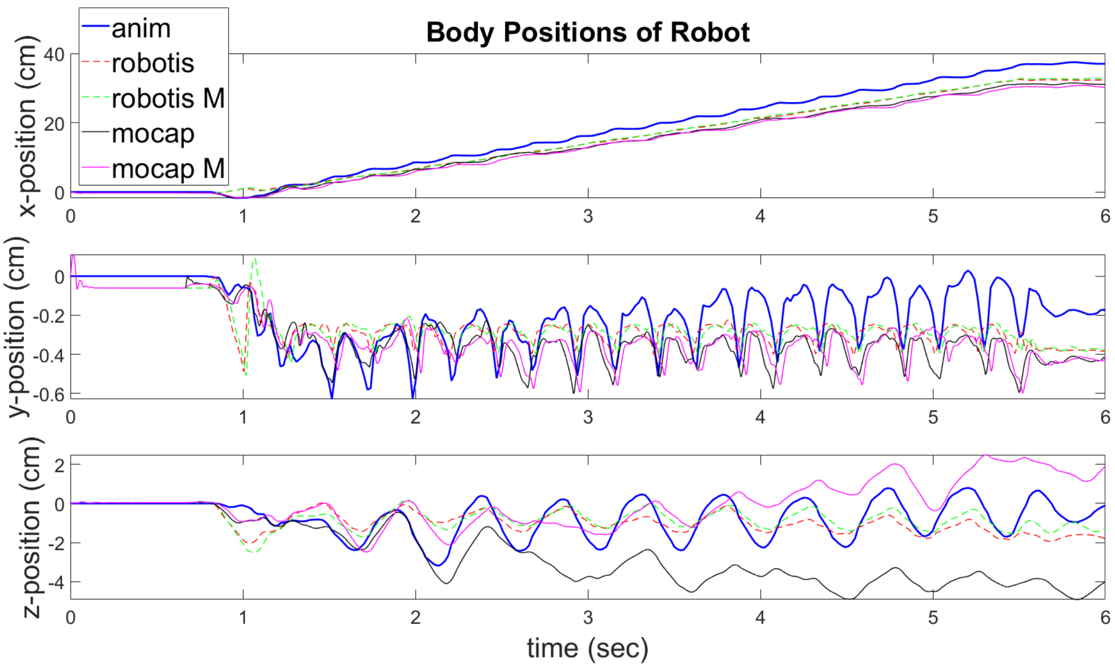


Figure 7.5: Simulation results from mocap dataset with MATLAB (mocap M) and own filtering (mocap) and derivatives compared to the real robot's trajectories.



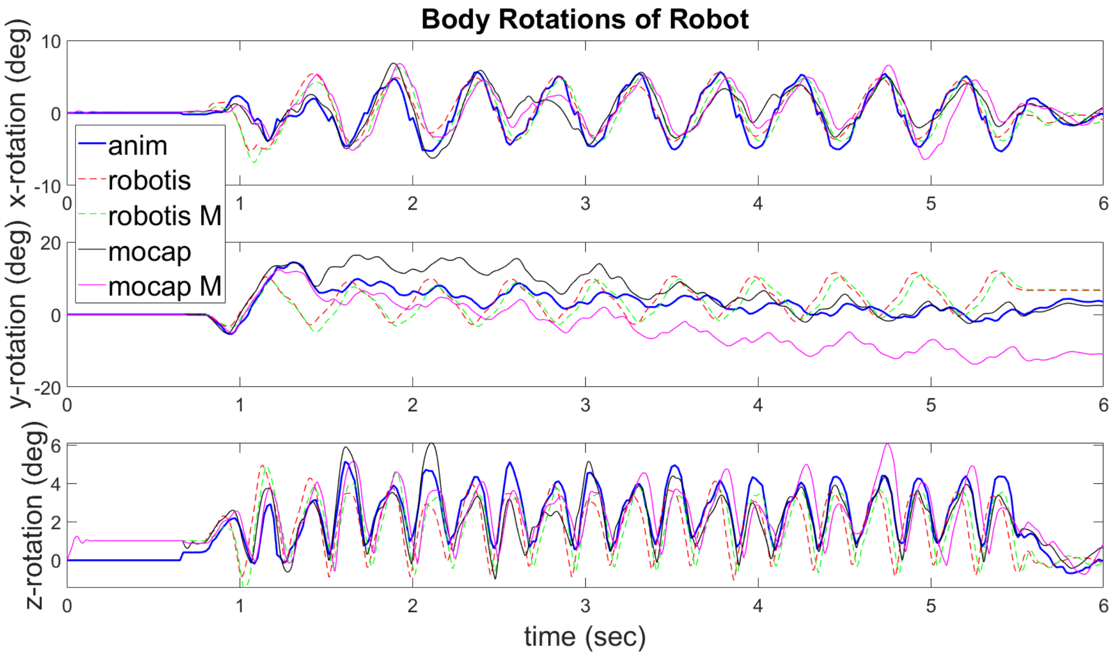
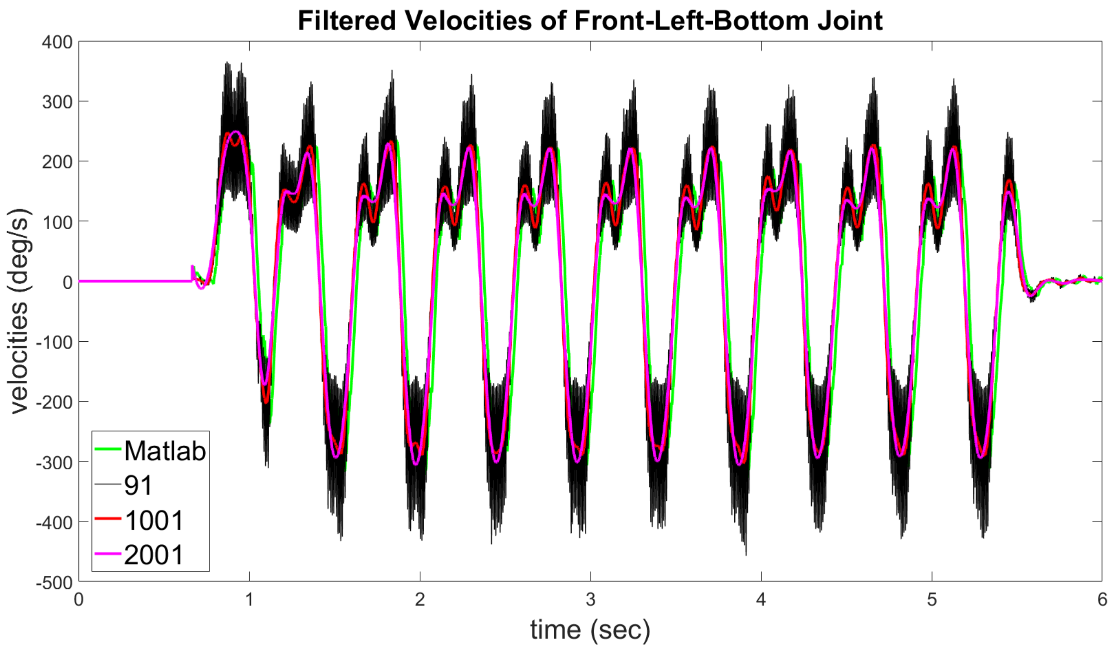
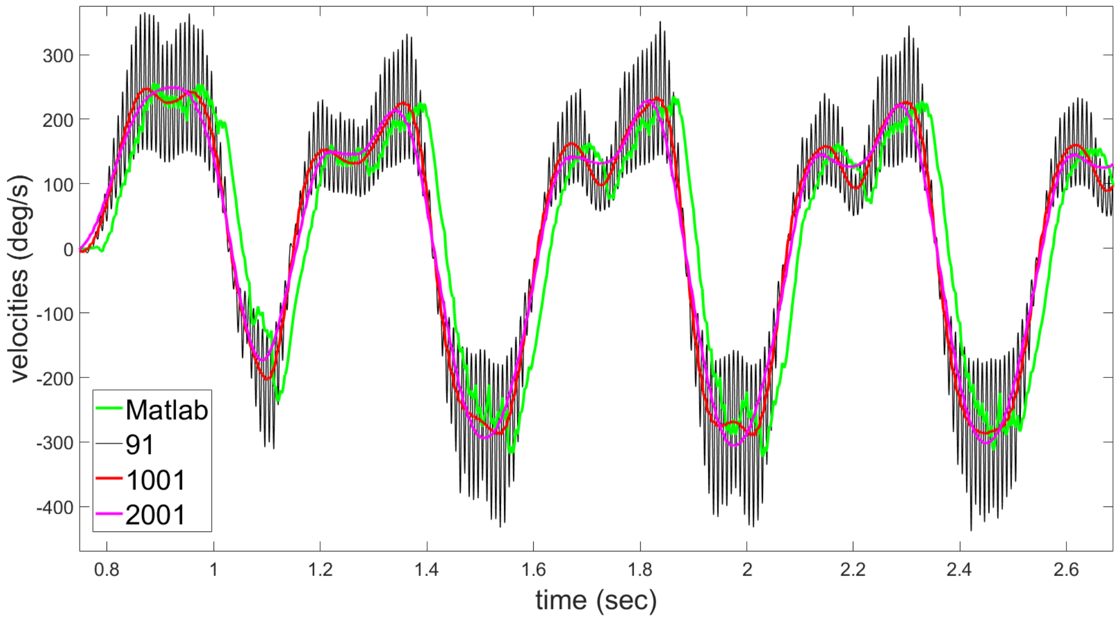


Figure 7.6: All simulation results compared to the real robot's trajectories.



(a)



(b)

Figure 7.7: Velocities (1st derivative) of angle trajectories for front-left-bottom joint of mocap dataset. MATLAB filtering and own filtering with different filter window sizes (91, 1001, 2001) (a). Zoomed-in view of velocities (b).

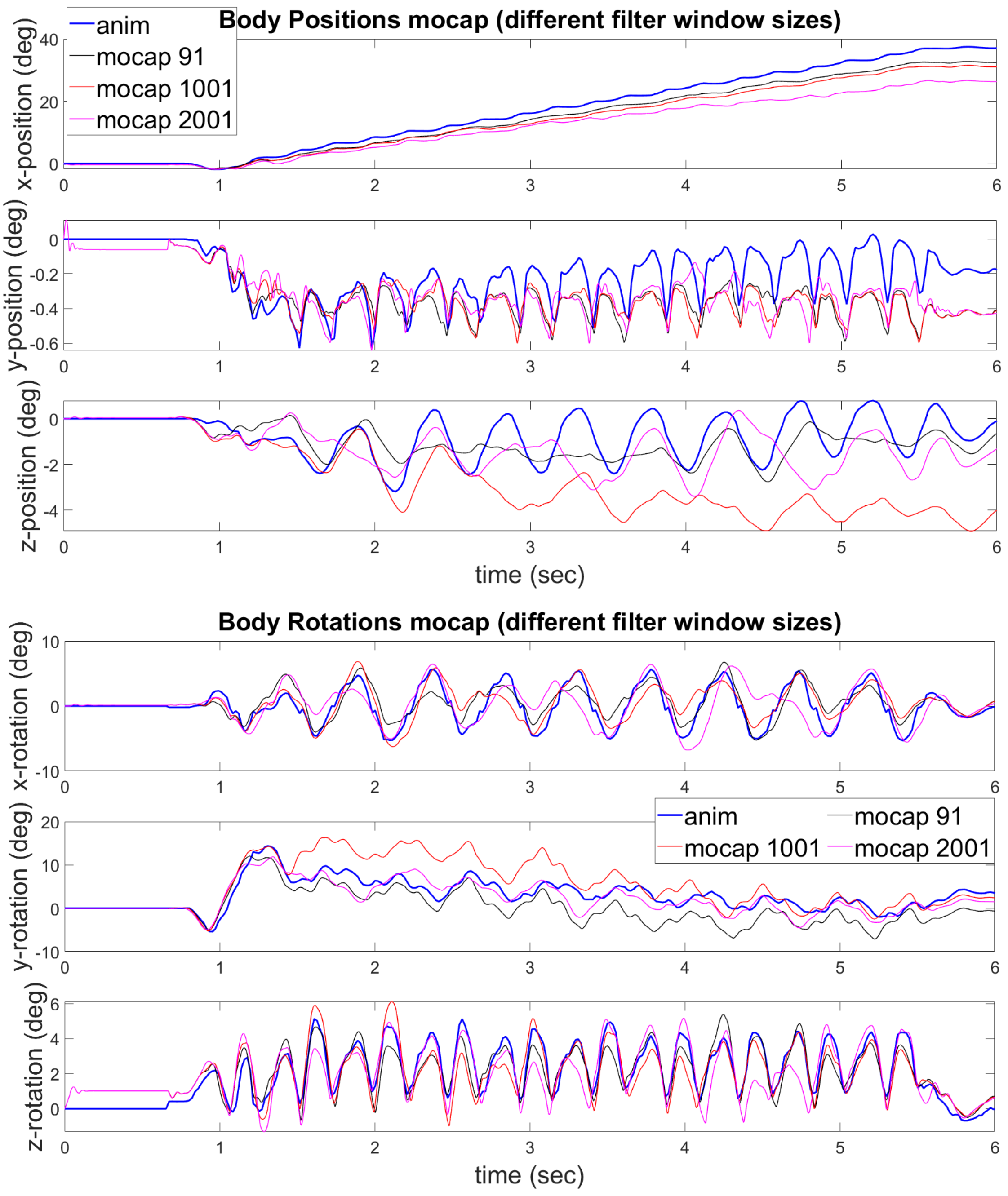


Figure 7.8: Mocap simulation results for different filter window sizes.

CHAPTER 8

Conclusion

8.1 Discussion

The diversity of the components posed a major challenge in this work. Each of the components required knowledge of different hard- and software and domain-specific terminologies. This made it difficult to study every aspect in depth. Designing the sandbox was started from scratch, a suitable robot had to be found, the accuracy of motion capture systems had to be determined to select the most appropriate one, and also simulation software had to be chosen before setup and implementation could start. The interaction between the components was also challenging, especially finding suitable formats for the data and aiming for consistency across components. Using existing simulation frameworks, such as Simulink, helped to get familiar with robotics and the physical complexity behind it. This sandbox laid the foundation for the exploration of control algorithms and the future training of robots. It offers a good starting point to dive deeper into robotics.

In the previous chapter, the robot's simulated and real walking pattern have been analysed and the performance of the simulated results has been compared in terms of covered distance. But, the simulation also generates other data, that can be examined with the sandbox and used for performance analysis. In addition to the joint angles and the position and orientation data of the robot's body, also velocities, torque, and contact forces are logged. It has been shown, that the robot's body trajectories vary greatly for differently filtered velocities. The impact of velocities on different performance goals could be investigated.
Apart from the performance metric discussed above, it would also be possible to define other metrics with the collected data. Performance could be measured based on how fast the robot can walk. This would require the analysis of torque in the simulation and a check if the simulated torque is achievable by the real robot. Another option could be to measure how smooth the gait of the robot is. For this performance goal, the joint

angles could be compared based on their smoothness. Different ground conditions could be examined by changing contact force parameters, and it could be simulated, which surface is best suited for the robot to execute a certain gait over a defined distance.
The sandbox also turned out to be a helpful debugging tool for examining tracked joint trajectories. At one point, comparing the real robot's trajectories with simulated ROBOTIS data showed that the real robot exhibited a drift to the left, whereas the simulation only had a slight drift to the right. After inspecting the real robot, loose screws were detected in the robot's front left leg. This had led to imprecise movements, which in turn had caused the drift to the left.

8.2 Future Work

The sandbox is meant to be a tool that helps to understand the reality gap and, ideally, to overcome it. Currently, the sandbox can be used to examine this reality gap, by determining the differences between reality and simulation. But, it is not yet possible to generate motion that can be tested directly on the robot. This is because the robot motors can only be actuated by angle input and not torque input. All data that is available for the simulation is angle data. With angle data, and without any other additional (machine learning) algorithms, the Simulink robot model can only be controlled using inverse dynamics. Inverse dynamics would compute torque, but the torque cannot be used by the robot. So, at the moment, the simulation results are just used for comparison with the real-world data.
In the future, control algorithms and machine learning approaches should be incorporated that can create angle input for the real robot. Deep reinforcement learning or supervised learning methods are promising for training legged locomotion. Further, hybrid learning strategies could be investigated that learn from simulated data and real-world data. Simulated data is cheaply available in large amounts, but also not as accurate as real-world data. Real-world data is limited and hard to capture, but more precise. Hybrid learning strategies incorporate the advantages of simulated and real-world data.

The focus of this work was on creating a usable and working sandbox design, on bringing the components together and on getting first simulation results. The components and the interaction between the components could still be improved. For now, motion capture data has to be exported from Motive and imported into the simulation manually. It would be convenient to use captured data directly in the simulation, without requiring the export/import steps. OptiTrack offers a client/server SDK, called NatNet [nat], for live streaming motion tracking data to other applications. NatNet provides an interface to MATLAB for streaming marker and rigid body data. However, if the tracking data is streamed directly, the post-processing steps are omitted. Until now, post-processing has always been necessary, mostly for interpolating gaps in marker trajectories and correcting rigid body orientations. It would be possible, to add post-processing steps in MATLAB too, or to just take the parts of the tracking data that are correct.
The simulation component is the one that can be refined the most. For the simulations, a solver recommended by MATLAB has been used with default settings. Other MATLAB

solvers could be tested to see if they deliver better results. A more detailed examination of how to compute and filter derivatives for the simulation input could be made to see if there is an improvement in simulation results depending on the applied filtering. It could also be tested if there are better simulation step sizes that provide a reasonable trade-off between simulation accuracy and computation time.
The Simulink robot model could be improved further. The mechanics' specifications of the real robot's motors could be used to adjust the model's joint parameters, such as spring stiffness and damping coefficient for the joint torque. The parameters for the contact force control could be tuned, including kinetic and static friction coefficients or contact stiffness and contact damping. The real robot walks on a concrete floor, so existing friction coefficients for plastic-on-concrete could be examined.
The firmware creation for the puppy robot could also be automated. Currently, the robot poses are converted to a format that is readable by the microcontroller, and then the robot program is built with the updated robot poses and loaded to the controller. It would be practical, if the conversion, the build, and the loading to the controller could be done in one step. Ideally, once a simulation result is available, and the robot is connected to the computer, the results are automatically transferred to the controller and can be tested on the real robot right away.

It has to be noted that the sandbox has been adapted for the puppy robot. This means the motion capture setup works best with robots similar to the puppy robot regarding size and constitution. Every robot is different, is constructed for a specific purpose, has different numbers of joints and different methods of actuation. For larger robots or robots with a different constitution, e.g. made for grasping objects, a modification of the motion capture setup has to be considered. Also, the 3D robot model used for simulation is specifically designed for the puppy robot. Other robots require their own models. Nevertheless, the construction of the sandbox is applicable to different kinds of robots too. And the information that is given in the previous chapters, can be helpful for others when building their own sandbox.

8.3 Summary

In this thesis, the design approach for a sandbox was described that facilitates the analysis of robots and their control algorithms. The sandbox consists of a motion capture component and a simulation component. Prior to the detailed description and realisation of the components, robotic applications using motion capture systems were introduced, and widely used robot simulators, as well as physics engines, were presented to give an overview of existing research and software. In robotics, motion capture systems are used to obtain movement data of robots to compare them to data computed by mathematical models or simulations. They are also used for steering or exploring the motion range of robots.
The motion capture component of the sandbox was realized with an OptiTrack system, consisting of six high-precision Prime 41 infrared tracking cameras. In this context, it was important to find a suitable environment for the camera setup and an ideal camera

placement, such that good tracking accuracy could be reached. The robot that was selected for this work was a small four-legged puppy robot from ROBOTIS actuated with 15 Dynamixel servomotors. To integrate the puppy robot into the sandbox, the robot controller was reprogrammed to make the transfer of motion data to the robot easier and to control the robot remotely. The robot was programmed with a straight walking gait and equipped with reflective markers to track its joint movements. Optimal marker placement and marker reconstruction were discussed as well as post-processing of tracking data.
The simulation component of the sandbox was realized with the graphical programming environment Simulink and the Simscape Multibody Library. In the CAD software SolidWorks, a detailed 3D model of the puppy robot was created with mass and inertia matrices and correct joint constraints. The model was imported into Simulink and extended with functionality, such that both tracking data and simulation data could be used to simulate the model. A data structure was created to store motion data in a consistent form for the simulation. In general, consistency for transferring data between the sandbox components was important to avoid confusion and mistakes. Another necessary step was the preprocessing (interpolation and filtering) of input data for the simulation. The robot model was simulated using inverse dynamics with angle data from ROBOTIS and tracking data from the motion capture system as inputs. Due to the lack of torque control of the real robot, it was not possible to test these simulation results in reality.
Nevertheless, a good comparison between simulation and reality could be made, making the notorious reality gap clearly visible. Comparison of the ROBOTIS dataset and the tracking dataset already showed that the ideal trajectories with which the robot was programmed changed significantly in reality, due to physical influences. For the comparison of the tracking data and the simulated data, the position and orientation trajectories of the robot's body were taken. Overall, the simulated robot did not manage to walk as far as the real robot, but kept a similar direction and was only a few centimetres off.
In the future, improving the simulation and aiming for more simulation accuracy is considered. There was not enough time to investigate learning algorithms and the training of robots. This is another major topic to be addressed to fully exploit the capabilities the sandbox has to offer.

List of Figures

Bibliography

[act] MATLAB Specifying Joint Actuation Inputs. https://nl.mathworks.com/help/physmod/sm/ug/joint-actuation.html. Accessed: 22.8.2020.

[aig] Closing the Simulation-to-Reality Gap for Deep Robotic Learning. https://ai.googleblog.com/2017/10/closing-simulation-to-reality-gap-for.html. Accessed: 6.10.2020.

[aim] OptiTrack - Aiming and Focusing. https://v22.wiki.optitrack.com/index.php?title=Aiming_and_Focusing. Accessed: 10.8.2020.

[ATW20] S. Abondance, C. B. Teeple, and R. J. Wood. A Dexterous Soft Robotic Hand for Delicate In-Hand Manipulation. *IEEE Robotics and Automation Letters*, 5(4):5502–5509, 2020.

[axi] Dynamixel AX-12A Servomotor Image. https://www.reichelt.com/ch/de/servomotor-robotik-9-0-12-v-dc-ax-12a-p249909.html. Accessed: 26.11.2020.

[Bha] M. Bhargava. Rigid Body Simulation. https://github.com/manas-avi/Rigid_Body_Simulation. Accessed: 26.11.2020.

[BR14] D. A. Bravo M. and C. F. Rengifo R. Motion Capture System for Applications in Robotics. In *2014 III International Congress of Engineering Mechatronics and Automation (CIIMA)*, pages 1–5, 2014.

[bul] Bullet Real-Time Physics Simulation. https://pybullet.org/wordpress/. Accessed: 4.10.2020.

[cad] CAD Translation. https://nl.mathworks.com/help/physmod/sm/ug/cad-translation.html. Accessed: 20.8.2020.

[cal] OptiTrack - Calibration. https://v22.wiki.optitrack.com/index.php?title=Calibration. Accessed: 10.8.2020.

[cama] OptiTrack - Camera Mount Structures. `https://v22.wiki.optitrack.com/index.php?title=Camera_Mount_Structures`. Accessed: 6.8.2020.

[camb] OptiTrack - Camera Placement. `https://v22.wiki.optitrack.com/index.php?title=Camera_Placement`. Accessed: 6.8.2020.

[CBCS15] M. Camurri, S. Bazeille, D. G. Caldwell, and C. Semini. Real-Time Depth and Inertial Fusion for Local SLAM on Dynamic Legged Robots. In *2015 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, MFI 2015, San Diego, CA, USA, September 14-16, 2015*, pages 259–264. IEEE, 2015.

[CCTM15] A. Cully, J. Clune, D. Tarapore, and J. Mouret. Robots That Can Adapt Like Animals. *Nat.*, 521(7553):503–507, 2015.

[che] Cheetah-Cub-S - Biorobotics Laboratory, EPFL. `https://www.epfl.ch/labs/biorob/research/quadruped/quadruped-past/cheetah-cub-s/`. Accessed: 4.10.2020.

[cho] MATLAB Choose a Solver. `https://nl.mathworks.com/help/simulink/ug/choose-a-solver.html`. Accessed: 23.8.2020.

[cm5] ROBOTIS e-Manual Controller CM-530. `https://emanual.robotis.com/docs/en/parts/controller/cm-530/`. Accessed: 24.8.2020.

[com] MATLAB Compare Solvers. `https://nl.mathworks.com/help/simulink/ug/compare-solvers.html`. Accessed: 23.8.2020.

[cop] CoppeliaSim Robot Simulator. `https://www.coppeliarobotics.com/`. Accessed: 4.10.2020.

[cub] MATLAB Function `cubicpolytraj`. `https://nl.mathworks.com/help/robotics/ref/cubicpolytraj.html`. Accessed: 21.8.2020.

[CVGL07] S. Chitta, P. Vemaza, R. Geykhman, and D. D. Lee. Proprioceptive Localization for a Quadrupedal Robot on Known Terrain. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4582–4587, 2007.

[DDM19] E. Dalin, P. Desreumaux, and J. Mouret. Learning and Adapting Quadruped Gaits with the "Intelligent Trial & Error" Algorithm. In *IEEE ICRA Workshop on "Learning legged locomotion"*, Montreal, Canada, 2019.

[DH55] J. Denavit and R. S. Hartenberg. A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *Journal of Applied Mechanics*, 22:215–221, 1955.

[dxl] ROBOTIS Dynamixel Communication. https://emanual.robotis.com/docs/en/dxl/protocol1/. Accessed: 31.8.2020.

[dyn] ROBOTIS e-Manual Dynamixel AX12-A. https://emanual.robotis.com/docs/en/dxl/ax/ax-12a/. Accessed: 24.8.2020.

[esy] OptiTrack - eSync2. https://optitrack.com/products/esync-2/indepth.html. Accessed: 9.8.2020.

[ETT15] T. Erez, Y. Tassa, and E. Todorov. Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 4397–4404. IEEE, 2015.

[FAQ] OptiTrack General FAQs. https://optitrack.com/support/faq/general.html. Accessed: 3.8.2020.

[fir] ROBOTIS e-Manual Firmware Development. https://emanual.robotis.com/docs/en/software/embedded_sdk/embedded_c_cm530/. Accessed: 24.8.2020.

[FL16] P. Flores and H. Lankarani. *Contact Force Models for Multibody Dynamics*. Springer International Publishing, 01 2016.

[for] Simscape Multibody Contact Forces Library - Contact Force Laws. https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/64648/versions/2/previews/CFL_Libs/Libraries/Help/html/Force_Laws.html. Accessed: 18.11.2020.

[fri] Simscape Multibody Contact Forces Library - Friction Force Laws. https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/64648/versions/2/previews/CFL_Libs/Libraries/Help/html/Friction_Laws.html. Accessed: 18.11.2020.

[FSNP09] M. Field, D. Stirling, F. Naghdy, and Z. Pan. Motion Capture in Robotics Review. In *2009 IEEE International Conference on Control and Automation*, pages 1697–1702, 2009.

[gaz] GAZEBO Physics Engine. http://gazebosim.org/. Accessed: 4.10.2020.

[how] MATLAB How Simscape Models Represent Physical Systems. https://nl.mathworks.com/help/physmod/simscape/ug/how-simscape-models-represent-physical-systems.html. Accessed: 23.8.2020.

[icr] Learning Legged Locomotion Workshop @ ICRA 2019. https://sites.google.com/view/learning-legged-locomotion. Accessed: 6.10.2020.

[KHB$^+$19] H. Kolvenbach, E. Hampp, P. Barton, R. Zenkl, and M. Hutter. Towards Jumping Locomotion for Quadruped Robots on the Moon. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*, pages 5459–5466. IEEE, 2019.

[kme] Learn how KMel Robotics brought quadrotors to life for Lexus' Amazing in Motion series. https://optitrack.com/about/customers/kmelRobotics.html. Accessed: 30.9.2020.

[KOO14] C. M. Korpela, M. Orsag, and P. Y. Oh. Towards Valve Turning Using a Dual-Arm Aerial Manipulator. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 3411–3416. IEEE, 2014.

[LGH$^+$18] J. Lee, M. X. Grey, S. Ha, T. Kunz, S. Jain, Y. Ye, S. S. Srinivasa, M. Stilman, and C. K. Liu. DART: Dynamic Animation and Robotics Toolkit. *J. Open Source Softw.*, 3(22):500, 2018.

[LKL19] T. Lin, A. U. Krishnan, and Z. Li. Physical Fatigue Analysis of Assistive Robot Teleoperation via Whole-body Motion Mapping. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2240–2245, 2019.

[LKL20] T. Lin, A. U. Krishnan, and Z. Li. Shared Autonomous Interface for Reducing Physical Effort in Robot Teleoperation via Human Motion Mapping. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pages 9157–9163. IEEE, 2020.

[LPB$^+$20] G. Li, N. Patel, C. Burdette, J. G. Pilitsis, H. Su, and G. S. Fischer. A Fully Actuated Robotic Assistant for MRI-Guided Precision Conformal Ablation of Brain Tumors. *IEEE/ASME Transactions on Mechatronics*, 2020.

[mar] OptiTrack - Markers. https://v22.wiki.optitrack.com/index.php?title=Markers. Accessed: 9.8.2020.

[MC17] J. Mouret and K. I. Chatzilygeroudis. 20 Years of Reality Gap: A Few Thoughts about Simulators in Evolutionary Robotics. In Peter A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1121–1124. ACM, 2017.

[mec] Simscape Multibody Mechanics Explorer. https://nl.mathworks.com/help/physmod/sm/ref/mechanicsexplorer-app.html. Accessed: 17.9.2020.

[MHQ16] M. Mao, J. He, and L. Qiu. CAT: High-Precision Acoustic Motion Tracking. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, page 69–81, New York, NY, USA, 2016. Association for Computing Machinery.

[Mil] S. Miller. Simscape Multibody Contact Forces Library. `https://github.com/mathworks/Simscape-Multibody-Contact-Forces-Library/releases/tag/20.1.5.1`. Accessed: 20.8.2020.

[Mol04] C. B. Moler. *Numerical Computing with MATLAB*, chapter 7. Society for Industrial and Applied Mathmatics, 2004.

[mota] OptiTrack - Motive Basics. `https://v22.wiki.optitrack.com/index.php?title=Motive_Basics`. Accessed: 10.8.2020.

[motb] OptiTrack - Motive Documentation. `https://v22.wiki.optitrack.com/index.php?title=Motive_Documentation`. Accessed: 9.8.2020.

[muja] MuJoCo advanced physics simulation. `http://www.mujoco.org/`. Accessed: 4.10.2020.

[mujb] MuJoCo Performance. `http://www.mujoco.org/performance.html`. Accessed: 4.10.2020.

[nat] OptiTrack - NatNet SDK. `https://optitrack.com/products/natnet-sdk/`. Accessed: 23.9.2020.

[NK18] G. Nagymate and R. Kiss. Application of OptiTrack Motion Capture Systems in Human Movement Analysis - A systematic Literature Review. *Recent Innovations in Mechatronics*, 5, 07 2018.

[ode] Open Dynamics Engine. `https://www.ode.org/`. Accessed: 4.10.2020.

[OKBO17] M. Orsag, C. M. Korpela, S. Bogdan, and P. Y. Oh. Dexterous Aerial Robots - Mobile Manipulation Using Unmanned Aerial Systems. *IEEE Trans. Robotics*, 33(6):1453–1466, 2017.

[opt] OptiTrack - Motion Capture. `https://optitrack.com/`. Accessed 4.8.2020.

[PGL$^+$19] M. Pessoa de Melo, J. Gomes da Silva Neto, P. Lima da Silva, J. Teixeira, and V. Teichrieb. Analysis and Comparison of Robotics 3D Simulators. In *21st Symposium on Virtual and Augmented Reality, SVR 2019, Rio de Janeiro, Brazil, October 28-31, 2019*, pages 242–251. IEEE, 2019.

[PGPW18] L. Pitonakova, M. Giuliani, A. G. Pipe, and A. F. T. Winfield. Feature and Performance Comparison of the V-REP, Gazebo and ARGoS Robot Simulators. In Manuel Giuliani, Tareq Assaf, and Maria Elena Giannaccini, editors, *Towards Autonomous Robotic Systems - 19th Annual Conference, TAROS 2018, Bristol, UK July 25-27, 2018, Proceedings*, volume 10965 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2018.

[phy] NVIDIA PhysX SDK. `https://developer.nvidia.com/physx-sdk`. Accessed: 4.10.2020.

[plu] Simscape Multibody Link Plugin. `https://nl.mathworks.com/help/physmod/smlink/index.html`. Accessed: 20.8.2020.

[PMG+19] J. Panerati, M. Minelli, C. Ghedini, L. Meyer, M. Kaufmann, L. Sabattini, and G. Beltrame. Robust Connectivity Maintenance for Fallible Robots. *Auton. Robots*, 43(3):769–787, 2019.

[pria] OptiTrack - Prime 41. `https://optitrack.com/products/primex-41/`. Accessed: 6.8.2020.

[prib] OptiTrack Prime 41 Specs. `https://www.optitrack.com/products/prime-41/specs.html`. Accessed: 3.8.2020.

[PTO+12] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo. ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[rbt] OptiTrack - Rigid Body Tracking. `https://v22.wiki.optitrack.com/index.php?title=Rigid_Body_Tracking`. Accessed: 10.8.2020.

[rec] OptiTrack - Reconstruction and 2D Mode. `https://v22.wiki.optitrack.com/index.php?title=Reconstruction_and_2D_Mode`. Accessed: 12.8.2020.

[res] The Resibots Project. `https://www.resibots.eu/`. Accessed: 29.9.2020.

[RMM18] F. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer. Speeded Up Detection of Squared Fiducial Markers. *Image and Vision Computing*, 76, 06 2018.

[roba] ROBOTIS RoboPlus (R+ Educational Software). `http://www.robotis.us/roboplus-r-educational-software-apps/`. Accessed: 24.8.2020.

[robb] ROBOTIS - About Us. `http://www.robotis.us/about-us/`. Accessed: 23.8.2020.

[robc] ROBOTIS - ROBOTIS Premium Product Page. `http://www.robotis.us/robotis-premium/`. Accessed:23.8.2020.

[SBW$^+$19] M. Strydom, A. Banach, L. Wu, R. Crawford, J. Roberts, and A. Jaiprakash. Real-time Joint Motion Analysis and Instrument Tracking for Robot-Assisted Orthopaedic Surgery. *CoRR*, abs/1909.02721, 2019.

[SC02] W. R. Sherman and A. B. Craig. *Understanding Virtual Reality: Interface, Application, and Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[sdf] SD/FAST physically-based simulation. `https://support.ptc.com/support/sdfast/index.html`. Accessed: 4.10.2020.

[set] OptiTrack - Setup Area. `https://v22.wiki.optitrack.com/index.php?title=Prepare_Setup_Area`. Accessed: 4.8.2020.

[sgo] MATLAB Savitzky-Golay Filter. `https://nl.mathworks.com/help/signal/ref/sgolayfilt.html`. Accessed: 20.9.2020.

[SHV06] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control.* John Wiley & Sons, 2006.

[sima] 2nd Workshop on Closing the Reality Gap in Sim2Real Transfer for Robotics. `https://sim2real.github.io/`. Accessed: 2.9.2020.

[simb] Simbody: Multibody Physics API. `https://simtk.org/projects/simbody/`. Accessed: 4.10.2020.

[simc] MATLAB Simscape Multibody Documentation. `https://www.mathworks.com/help/physmod/sm/`. Accessed: 15.8.2020.

[simd] MATLAB Simulink. `https://nl.mathworks.com/help/simulink/index.html`. Accessed: 15.8.2020.

[SIZ19] D. Shachaf, O. Inbar, and D. Zarrouk. RSAW, A Highly Reconfigurable Wave Robot: Analysis, Design, and Experiments. *IEEE Robotics Autom. Lett.*, 4(4):4475–4482, 2019.

[SK08] B. Siciliano and O. Khatib. *Springer Handbook of Robotics.* Springer-Verlag Berlin Heidelberg, 2008.

[sol] SolidWorks. `https://www.solidworks.com/`. Accessed: 19.8.2020.

[stm] STM32F10x Standard Peripherals Library. `https://www.mikrocontroller.net/articles/STM32F10x_Standard_Peripherals_Library`. Accessed: 1.9.2020.

[stp] ROBOTIS-Download Robot Parts. `http://en.robotis.com/service/downloadpage.php?ca_id=70`. Accessed: 30.8.2020.

[swi] Netgear Switch ProSAFE GS728TPP. https://www.netgear.at/business/products/switches/smart/GS728TPP.aspx#tab-technischespezifikationen. Accessed: 9.8.2020.

[tob] MATLAB Robtics System Toolbox. https://nl.mathworks.com/products/robotics.html. Accessed: 4.10.2020.

[TSL] J. Tan, K. Siu, and C. K. Liu. Contact Handling for Articulated Rigid Bodies Using LCP. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.296.5327. Accessed: 26.11.2020.

[uab] Unity Assets Bundle Extractor. https://github.com/DerPopo/UABE. Accessed: 15.8.2020.

[var] MATLAB Variable Step Solvers in Simulink. https://nl.mathworks.com/help/simulink/ug/variable-step-solvers-in-simulink-1.html. Accessed: 23.8.2020.

[vic] Vicon - Motion Capture. https://www.vicon.com/. Accessed: 4.8.2020.

[vid] YouTube Video Amazing in Motion - 'Swarm'. https://www.youtube.com/watch?v=5ofbd05pKuA&ab_channel=LexusQatar. Accessed: 2.10.2020.

[web] Webots Simulator. https://cyberbotics.com/#features. Accessed: 5.10.2020.

[WEWI15] K. Weinmeister, P. Eckert, H. Witte, and A. J. Ijspeert. Cheetah-cub-S: Steering of a Quadruped Robot Using Trunk Motion. In *2015 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2015, West Lafayette, IN, USA, October 18-20, 2015*, pages 1–6. IEEE, 2015.

[wir] OptiTrack - Cabling and Wiring. https://v22.wiki.optitrack.com/index.php?title=Cabling_and_Wiring. Accessed: 9.8.2020.

[xse] Xsens, Inertial Sensor Modules. https://www.xsens.com/inertial-sensor-modules. Accessed: 4.8.2020.

[ZD15] S. Zapolsky and E. M. Drumwright. Inverse Dynamics with Rigid Contact and Friction. *CoRR*, abs/1509.03355, 2015.

[zig] ROBOTIS e-Manual ZigBee Communication. https://emanual.robotis.com/docs/en/parts/communication/zig-110/. Accessed: 24.8.2020.