Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Henrik M. Arnesen, Kristian Grinde, Vegard Hovland and Even Vestland

Development of Biomimetic Robot Leg with ROS Implementation

Utvikling av Biomimetisk Robotfot med ROS Implementasjon

Bachelor's project in Electrical Engineering - Automation (E2103)
Supervisor: Torleif Anstensrud

May 2021

NTNU
Kunnskap for en bedre verden

Henrik M. Arnesen, Kristian Grinde, Vegard Hovland and Even Vestland

Development of Biomimetic Robot Leg with ROS Implementation

Utvikling av Biomimetisk Robotfot med ROS Implementasjon

Bachelor's project in Electrical Engineering - Automation (E2103)
Supervisor: Torleif Anstensrud
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Kunnskap for en bedre verden

Abstract

This thesis documents the development of an over-actuated robot leg with four degrees of freedom. The thesis covers design, fabrication, assembly, mathematical framework, control, and simulation. The thesis aims to enable future research on biomimetic robot movement and create a physical model for educational purposes in robotics.

The robot leg consists of four actuators where three are placed in the same plane to emulate the hip, knee, and ankle of a domestic cat's hind leg. This makes the robot over-actuated when viewed as a planar robot. 3D-printed parts make up the majority of parts connecting the actuators together. A simple anatomical analysis was undertaken to find correct proportions for each link. The robot leg is fastened to a stand with caster wheels to emulate locomotion.

A mathematical model for the kinematics of the robot leg was created and implemented in Matlab. This enabled planning trajectories between waypoints found in a gait analysis on cats. The Matlab model for the robot was never implemented on the physical model but is included as a foundation for integrating Matlab with ROS in future work. A more complete modeling scheme and more optimal controllers for the robot are also discussed.

The embedded system consists of four brushed DC motors controlled by an Arduino Mega running independent joint PID control for each actuator. Dual motor drivers are used for translating the PID control signals into actuator speeds. The angular position of the joints is recorded using incremental encoders connected to an Arduino Nano operating as an incremental encoder interface. The communication between the units is done using the I2C protocol and between the Mega and ROS using serial USB. In the end, the physical model worked as intended, but the actuators were undersized and could not handle the gravitational forces acting on the upper joints.

A robot model for use in ROS was created by exporting design files. The robot model enables simulating movements and gaits before implementing it on the physical model. Trajectories were generated by setting a start and stop pose for the robot leg. ROS transmits setpoints to the embedded motor controllers that track the planned trajectory. Finally, a model including four legs was created and simulated with the use of ROS.

In the end, most aspects of the robot worked as intended. The actuators were too weak, but in the short time before failure ROS was able to send setpoints to the controllers, and the correct poses were achieved. Given more time or a larger budget, the group is confident that the robot would be completely operational. To make future work on the robot easier, a lot of the discussions found in this thesis are focused on future work and possible improvements.

Sammendrag

Denne bacheloroppgaven dokumenterer utviklingen av et overaktuert robotben med fire frihetsgrader. Dette omfatter design, konstruksjon, det matematiske rammeverket, regulering og simulering. Målet med prosjektet er å muliggjøre framtidig forskning på biomimetisk robotbevegelse og skape en fysisk modell for bruk i undervisning innen robotikk.

Robotbenet består av fire aktuatorer hvor tre er plassert i samme plan for å etterligne hofte, kne og ankel til en vanlig huskatt. Dette gjør roboten overaktuert hvis den blir sett på i to dimensjoner. 3D-printede deler utgjør de fleste delene som kobler aktuatorene sammen. En enkel anatomisk analyse ble utført for å finne korrekte proporsjoner for hver lenke. Robotbenet er festet til et stativ med kulehjul for å muliggjøre bevegelse.

En matematisk modell for kinematikken til roboten ble implementert i Matlab. Dette muliggjorde planlegging av baner mellom viapunkter funnet i en gangeanalyse av katter. Matlab modellen for roboten ble aldri implementert på den fysiske modellen, men er inkludert som grunnlaget for integrasjon mellom Matlab og ROS i framtidig arbeid. En mer komplett matematisk modell og optimal regulerings løsning er også presentert.

Det integrerte elektroniske systemet består av fire likestrøms børstemotorer kontrollert av en Arduino Mega som kjører individuell PID-regulator for hvert ledd. Doble motordrivere brukes for å konvertere PID-regulatorens pådragssignal til aktuatorhastighet. Vinkelposisjonen til leddene er målt med inkrementelle enkodere, hvor signalene blir lest og posisjonen lagret av en Arduino Nano for hver aktuator. Kommunikasjonen mellom enhetene er gjort ved hjelp av I2C protokollen, mens mellom Megaen og ROS benyttes seriell USB. Til slutt fungerte det integrerte elektroniske systemet som planlagt med unntak av motorene som viste seg å være for svake for roboten under bruk.

Ved å eksportere informasjon fra designfilene ble det laget en robotmodell som kunne brukes i blant annet ROS. Denne muliggjør simulering av bevegelse og ganglag før testing på den fysiske modellen. Baner ble generert ved å sette start og stopp posisjoner for robotbenet. ROS er satt opp til å sende settpunkter til motor regulatorene slik at den fysiske roboten vil følge den planlagte banen. En modell med fire ben er også laget og simulert i ROS.

De fleste aspektene ved roboten fungerte til slutt som planlagt, med unntak av aktuatorene som var for svake. Før de sviktet ble det vist at ROS sendte settpunkter til regulatorene og at ønskede posisjoner ble oppnådd. Gitt mer tid eller større budsjett er gruppemedlemmene sikre på at full funksjonalitet ville blitt oppnådd. For å gjøre framtidig arbeid enklere er mye av diskusjonen funnet i denne oppgaven fokusert på framtidig arbeid og mulige forbedringer.

Acknowledgments

With the completion of this project, the authors of this thesis end our 3-year bachelor program in electrical engineering with a specialization in automation. We would like to thank our lecturers during these years for providing us with the knowledge needed to complete this assignment and confidently call ourselves engineers.

For help and assistance during this thesis, we would like to thank the department of engineering cybernetics and especially the technical and mechanical workshops for lending us equipment and workspaces. Another thanks must be given to MakeNTNU for the 200+ hours of printing time needed to print the parts for our robot. Furthermore, we would like to thank Florian Fischer for his critical help with the creation of the URDF for our robot and Mathias Hauan Arbo for his much-appreciated guidance in ROS.

Finally, we would like to thank our supervisor Torleif Anstensrud for his sound advice, help, and guidance throughout this project.

Signatures

Henrik M. Arnesen
<henrimar@stud.ntnu.no>

Kristian Grinde
<krigrin@stud.ntnu.no>

Vegard Hovland
<vegarbho@stud.ntnu.no>

Even Vestland
<evenves@stud.ntnu.no>

The content of this thesis is freely available, but publication (with reference) may only be pursued due to agreement with the authors.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Project Assignment . . . 2
1.3 Thesis Statement . . . 3
1.4 Problem statements . . . 3
1.5 Objectives . . . 4
1.6 Structure of the report . . . 4
1.7 Definitions . . . 6

2 Method **7**
2.1 Development method . . . 7
2.2 Research method . . . 8
2.3 HSE and Risk assessment . . . 8

3 Design **9**
3.1 Introduction . . . 9
3.2 Method and Equipment . . . 10
3.2.1 Parametric modeling principles . . . 10
3.2.2 3D-printing . . . 10
3.2.3 Captured nuts . . . 10
3.2.4 Software . . . 11
3.2.5 Equipment . . . 11
3.3 Results and Empirical Findings . . . 11
3.3.1 Anatomical Analysis . . . 11
3.3.2 Gait Analysis . . . 12
3.3.3 Robot Configuration . . . 13
3.3.4 Leg Design . . . 15
3.3.5 Stand Design . . . 19
3.3.6 Complete Assembly . . . 22
3.3.7 Order of operations . . . 23
3.3.8 URDF Export . . . 23
3.4 Analysis and Discussion . . . 24
3.4.1 Anatomical and Gait Analysis . . . 24

3.4.2 Method of Mounting 25
3.4.3 Leg Design 25
3.4.4 Stand Design 28
3.4.5 3D-Print and Material 28
3.4.6 Interdisciplinary Project 30
3.4.7 URDF-Export 30
3.4.8 Spring Dampening Addition 31
3.5 Chapter Conclusion 32

4 Mathematical Model **33**
4.1 Introduction 33
4.2 Method and Software 34
4.2.1 Method 34
4.2.2 Software 34
4.3 Theoretical Framework 34
4.3.1 Rigid Transformation 34
4.3.2 Denavit-Hartenberg Convention 35
4.3.3 World vs base frame 36
4.3.4 Forward Kinematics 36
4.3.5 Inverse Kinematics 38
4.3.6 Velocity Kinematics 38
4.3.7 Waypoints and path 39
4.3.8 Trajectory 40
4.3.9 Lagrangian mechanics 40
4.4 Results and Empirical Findings 41
4.4.1 Denavit–Hartenberg Parameters 41
4.4.2 Forward Kinematics 42
4.4.3 Inverse Kinematics 46
4.4.4 Velocity Kinematics 47
4.4.5 Path planning 48
4.4.6 Plotting Functions 51
4.4.7 System dynamics 51
4.5 Analysis and Discussion 54
4.5.1 Denavit–Hartenberg Parameters 54
4.5.2 Velocity Kinematics 54
4.5.3 Trajectory Optimization 55
4.5.4 Code optimization 56
4.5.5 Hybrid systems 56
4.5.6 Experimental modeling 57
4.5.7 Physical simulation in Matlab 57
4.6 Chapter Conclusion 58

5 Control **60**

5.1 Introduction . . . 60
5.2 Theoretical Framework . . . 61
5.2.1 Feedback Loop . . . 61
5.2.2 PID . . . 61
5.2.3 Pole Placement . . . 62
5.2.4 Linear-Quadratic Regulator . . . 64
5.2.5 Digital Control Systems . . . 65
5.2.6 MIMO Systems . . . 66
5.2.7 Closed Loop Response . . . 67
5.3 Discussion and Results . . . 67
5.3.1 Choice of Controller . . . 67
5.3.2 Discrete PID . . . 68
5.3.3 Controller Tuning . . . 69

6 Embedded Systems **70**
6.1 Introduction . . . 70
6.2 Research Method and Equipment . . . 71
6.2.1 Method . . . 71
6.2.2 Equipment . . . 71
6.2.3 Software . . . 71
6.2.4 Arduino libraries . . . 72
6.3 Theoretical Framework . . . 72
6.3.1 Actuator Types . . . 72
6.3.2 Back Driven actuators . . . 73
6.3.3 Braking . . . 73
6.3.4 Gearing . . . 74
6.3.5 Inter-Integrated Circuit . . . 76
6.3.6 Serial Peripheral Interface . . . 76
6.3.7 Encoders . . . 77
6.3.8 Arduino Platform . . . 78
6.3.9 EEPROM . . . 78
6.4 Results and Empirical Findings . . . 79
6.4.1 Embedded circuit schematic . . . 79
6.4.2 Actuators . . . 82
6.4.3 Embedded Controller . . . 82
6.4.4 Incremental encoder Interface . . . 83
6.5 Analysis and Discussion . . . 85
6.5.1 Actuators, Gearing and Belt Drive . . . 85
6.5.2 Mounting Hub . . . 86
6.5.3 Encoders . . . 87
6.5.4 Circuit Design . . . 87
6.5.5 Choice of Microcontrollers . . . 88
6.5.6 Current Sense . . . 90

6.5.7 Software Design 91
6.5.8 Future Expansions 92
6.6 Chapter Conclusion 93

7 Robot Operating System 94
7.1 Introduction 94
7.2 Research Method and Equipment 96
7.2.1 Method 96
7.2.2 ROS packages 96
7.3 Theoretical Framework 96
7.3.1 Communication Infrastructure 96
7.3.2 Filesystem 99
7.3.3 Project-essential applications 101
7.4 Results and Empirical Findings 104
7.4.1 Source directory 104
7.4.2 Simulation 105
7.4.3 Motion planning and trajectory execution 107
7.4.4 Joint command interface 110
7.4.5 Hardware interface 110
7.4.6 GUI 112
7.4.7 Launching physical robot 112
7.5 Analysis and Discussion 112
7.5.1 ROS version and distribution 112
7.5.2 Motion Planning 113
7.5.3 Controller implementation 114
7.5.4 Hardware interface 115
7.5.5 Serial comunication 115
7.5.6 Features for further work 116
7.6 Chapter Conclusion 117

8 Results and Empirical Findings 118
8.1 Physical Model 118
8.2 Gait Execution 120

9 Analysis and Discussion 121
9.1 Future Work 121

10 Conclusions 123

A Budget and Record 131

B Gantt 132

C Parts List 133

D Arduino Code **135**
D.1 Master 135
D.2 Slave 148

E Python Code **150**

F Matlab Code **157**

G Inverse Kinematics **165**

H User Manual **169**
H.1 Introduction 169
H.2 Equipment 169
H.3 Create Catkin workspace 170
H.4 Running simulation 170
H.5 Executing on real robot 171
H.5.1 Plot data from topics 172
H.5.2 Executing trajectory 173

I Poster **174**

List of Figures

2.1 Kanban board for Arduino Software 7

3.1 Captured Nut Design 11
3.2 Gait analysis - Finding angles and positions from slideshow of cat gait . 13
3.3 Robot-leg Configuration 14
3.4 Link Design - Motor encoder cap 15
3.5 Link Design - Upper link 16
3.6 Link Design - Lower link 17
3.7 Link design - Femur and tibia 18
3.8 Link design - Hip 18
3.9 Link design - End effector 19
3.10 Stand Design - Top Plate Top 20
3.11 Stand Design - Top Plate Bottom 21
3.12 Stand Design - Pipe Connector 21
3.13 Stand Design - Castor wheel connector 22
3.14 Complete Assembly - Leg and Stand Design 23
3.15 Part configuration-tree in Fusion 360 24
3.16 Quadruped robot visualization 25
3.17 Link Design - Upper link, old design 26
3.18 Link Design - Lower link, new design 27
3.19 Asymmetric link design 27
3.20 Design on its side and tilted $45°$ on printer build plate 29
3.21 Possible spring dampening system 31

4.1 Trajectory equations expressed with matrices 40
4.2 A-Matrices for robot configuration with symbolic calculations 43
4.3 T-Matrices for robot configuration with symbolic calculations 44
4.4 Forward Kinematics - xz-projection 45
4.5 Forward Kinematics - yr-projection 45
4.6 Linear and angular movement 48
4.7 Robot plotted for each waypoint generated 49
4.8 Actuator angles for complete gait cycle 50
4.9 Actuator velocities for complete gait cycle 50

4.10 Actuator accelerations for complete gait cycle . . . 51
4.11 End effector coordinates for complete gait . . . 55
4.12 Imported URDF in Matlab . . . 58

5.1 Negative feedback loop . . . 61
5.2 PID illustration . . . 62
5.3 State feedback controller . . . 63
5.4 State Estimator, (Anstensrud 2020c) . . . 64
5.5 MIMO system . . . 66
5.6 Block diagram for independent joint PID . . . 67
5.7 Windup . . . 69

6.1 Different gear designs . . . 75
6.2 Different gear drives . . . 76
6.3 Different gear drives . . . 77
6.5 Embedded circuit schematic . . . 79
6.6 On-board electronics . . . 80
6.7 Arduino Nano circuit board . . . 81
6.8 Power supply and emergency stop . . . 82
6.9 Power buffer . . . 85
6.11 Raspberry Pi 4 (The Raspberry Pi Foundation 2021) . . . 90

7.1 *MoveIt* pipeline overview (Ioan A. Sucan and Sachin Chitta 2021a) . . . 104
7.2 Source directory . . . 105
7.3 *Gazebo* simulation window on startup . . . 106
7.4 *Gazebo* simulation with position controllers . . . 107
7.5 Planned trajectory generated by *MoveIt* . . . 108
7.6 Planned trajectory with Cartesian path enabled . . . 109
7.7 A distance traveled by only running loop_gait.py . . . 110
7.8 Latency test . . . 111
7.9 Displaying information using *RViz* and *plotjuggler* . . . 112
7.10 Example of the end effectors orientation being dependent on the hip joint 114
7.11 Quadruped robot simulation in *Gazebo* and move groups in *MoveIt* . . . 117

8.1 Physical model - Joint . . . 118
8.2 Physical model - complete . . . 119
8.3 Joint 3 and 4 tracking trajectory . . . 120

G.1 Forward Kinematics - xz-projection . . . 165
G.2 Forward Kinematics - yr-projection . . . 166

H.1 *rqt_graph* . . . 172
H.2 Suggested start positions . . . 172
H.3 *Plotjuggler* layout . . . 173

List of Tables

3.1 Link and animal comparison 12
3.2 Proportions in percent of total leg length for cats and dogs 12
3.3 Gait analysis - Waypoints 13
3.4 Actual link lengths 15

4.1 Denavit-Hartenberg Table for an arbitrary system 36
4.2 Symbolic Denavit Hartenberg table for this specific configuration 42

A.1 Bachelor thesis budget in NOK 131

Code Listings

6.1 Creating a object list used to control the four actuators 83
6.2 Receiving integer from slave 84
6.3 Transmitting integer from slave 84
6.4 Saving to EEPROM 85
7.1 Low latency mode 111
D.1 actuator.h 135
D.2 actuator.cpp 137
D.3 variables.h 141
D.4 main.ino 142
D.5 slave1.ino 148
E.1 command interface 150
E.2 loop_gait.py 153
F.1 calculation of $\boldsymbol{A}$-matrix 157
F.2 Symbolic calculation of all $\boldsymbol{A}$-matrices 157
F.3 Symbolic calculation of all $\boldsymbol{T}$-matrices 158
F.4 Symbolic calculation of $\boldsymbol{J}$-matrix 158
F.5 Inverse kinematics for the configuration using geometry 159
F.6 Using the inverse kinematics function to return a matrix containing all angles for all via points 159
F.7 Symbolic calculation of quintic polynomial between two angles 160
F.8 Symbolic calculation of all polynomials for the configuration 161
F.9 Converting the symbolic functions into arrays of angles for plotting 161
F.10 Ploting function for the robot in its current configuration 162
F.11 Animated plotting of robot 163
F.12 Determining the velocity direction if any for waypoints 164
H.1 Create catkin workspace 170
H.2 Source setup file 170
H.3 Launch Gazebo simulation 170
H.4 Launch MoveIt 170
H.5 Loop gait 171
H.6 Launch robotleg.launch 171
H.7 *rqt_graph* 171
H.8 Run *plotjuggler* 172

H.9 Run gait loop 173

Chapter 1

Introduction

1.1 Background

In modern times, automation is becoming an increasingly important topic, and technology is constantly developing new ways to eliminate menial tasks from everyday life. One of the more exciting fields in the world of automation is the development of robots. Robots have many possibilities for different forms of locomotion. With the ability to influence their environment, they can be a great way to spare humans from dull and sometimes potentially dangerous physical tasks.

Robots come in many different shapes and forms depending on what tasks they are meant to do. They are most commonly found in industrial settings where they are used in the manufacturing of products, typically in the form of a manipulator arm fixed to a stationary frame along a manufacturing belt. However, as the development of robots has progressed, the focus of many research institutions has turned slightly over to mobile robots, with the existence of a mobile robotics lab in most renowned technological universities. The ability to move around in a larger environment opens the possibility for making robots with more flexible areas of use.

Mobile robots can travel by land, air, and sea. Land-based robots are the most universal, as they have the best possibilities to perform heavy physical labor with the technology that exists today. Air and water-based robots are more suited for observational purposes such as exploration and inspection, as well as transportation of lightweight cargo. When it comes to land-based mobile robots, the case has been made that while wheeled and tracked robots seem to have the most effective way of locomotion, legged robots have many environments in where they are superior (Silva and Tenreiro Machado 2007). Legged robots can maneuver a larger variety of terrain, even rough terrain that can prove difficult for some humans, while wheeled and tracked robots are more dependent on working environments with minimal verticality and paved roads.

Legged robots can be categorized by how many legs they have. Bipedal robots, for

example, are robots with two legs, typically made in an attempt to mimic humans. Bipedal robots generally have a higher center of gravity than multi-legged robots and, combined with fewer points of contact with the ground to distribute weight, they have a harder time keeping their balance. They are therefore in need of more demanding control systems and do not necessarily have fewer actuators than other multi-legged robots. Four-legged robots, known as quadrupedal robots, have a lot more natural stability and are therefore more suitable for practical use. This is also true for robots with even more legs, like hexapedals and octopedals. However, in their case, the number of actuators will indeed increase and amount to more sources of power consumption.

When it comes to the design of modern mobile robots, many of them are inspired by living creatures. This is a practice known as biomimicry, meaning life imitation, and is founded on the belief that one should look to nature for inspiration on how to solve design problems. This is the case for most legged robots, and so, four-legged robots are often inspired by naturally four-legged animals such as dogs and cats. Modern examples of this are the robot Spot from Boston Dynamics, imitating a big dog (Boston Dynamics 2021), and Mini Cheetah from MIT, imitating an average-sized cat (Katz, Carlo and Kim 2019). Something worth noting is that their anatomies are not made to be exact copies of these respectful animals. The designers have purposefully simplified the robot's legs, having one joint and link less than the animal they are inspired by. The reasons for this are that the additional DOF increases the mechanical complexity and cost considerably. Besides, there have been made studies of how many actuators robots of different leg categories can be reduced to without significantly impacting their capabilities for locomotion (Yoneda and Ota 2003). This poses the interesting question of the design of four-legged creatures, which is polished over millennia by evolution, can eventually bring a noteworthy edge in agility over simplified robots, or if their extra joints and links truly are redundant and not worth implementing in future robots (Arnesen, Grinde, Hovland and Vestland 2021a).

1.2 Project Assignment

Developing test bench for robot leg with four degrees of freedom

Four-legged robots have in recent years become popular as a platform for future use among humans. Common for many of them is a configuration with two degrees of freedom in walking direction and one degree of freedom perpendicular to the walking direction to make the robot more agile. The hypothesis that inspires this project is if adding another degree of freedom, and making the robot more anatomically correct compared to quadrupedal animals, can give the robot increased agility and mobility. To be able to test this hypothesis, the group that undertakes this project will be tasked to build a test bench for a robotic leg with four degrees of freedom, three in a two-dimensional walking direction and one perpendicular to this walking direction for

three-dimensional movement. Each joint could be controlled by a position regulator where the result of inverse kinematics makes up the reference angels. The test rig should include the possibility for horizontal and vertical movement where the robot can move under its own power. Mounting the robot on a movable stand or track enables the possibility of controlling the position of the robot by counting steps and measuring the step length. The test bench should be constructed in a way that enables it to be used for future projects and in a teaching setting. To test the hypothesis, different gaits can be calculated and tested. Comparisons in two dimensions can be made in simulations to see possible improvements between two and three degrees of freedom.

Possible topics for the execution of this bachelor thesis:

- Prepare concept drawings and schematics for future usage
- Calculation of direct, inverse, and velocity kinematics for the model
- Simulation of the kinematics in e.g., Matlab
- Choice of actuators and sensors based on specifications
- Design and construct the physical model
- Choice of microcontroller for communication with ROS (e.g., Arduino)
- Using ROS to set up communication between actuators and sensors
- Explore the possibility to measure impact in the foot to regulate angles /positions
- Designing a demonstration for a physical and mathematical model
- Setting up the project with future use in mind

1.3 Thesis Statement

Develop and build a 4dof robot leg, using mathematical modeling and ROS-applications, for use in education and research.

1.4 Problem statements

Due to the wide breadth of the thesis, several problems would have to be overcome. This non-comprehensive list aims to summarize the most pertinent to the project.

- The design of a robot leg on a tight budget demands careful planning when it comes to design and material choices.
- Explore possibilities for model-based controllers and the required mathematics to support this.
- Creating a simulated model would enable visualization of the robot leg and even expand upon the physical model.
- Moving the robot leg between different poses requires choosing and tuning controllers.
- Choosing strong and fast enough actuators is crucial for achieving satisfactory movement when implementing the controllers.

- Using one or more microcontrollers in unison with ROS requires reliable communication to be set up between the units.
- Documentation for all aspects of the project is crucial for future work and expansions.

1.5 Objectives

Some objectives were defined to solve the problems previously stated. This will enforce a structure upon which the work on this thesis can be built.

- By designing as many parts as possible and using 3D-printing technology, the budget can be held low.
- Using matrix calculations and trigonometry, a precise model for the kinematics of the robot leg can be calculated. Implementing these in Matlab will simplify the mathematics and allow visualization of the results.
- Implementation of the design in ROS will enable simulating different gaits and poses with *Gazebo* and *Rviz*. Exploring the possibilities for connecting Matlab to ROS could expand the simulation possibilities.
- As a start, PID-based controllers will be implemented while exploring alternative solutions using model-based controllers like LQR.
- Budgeting constraints will severely limit the available motors. But by sourcing motors with a high gear ratio, strong enough motors will hopefully be found.
- The communication method between ROS and the robot would depend on the microcontrollers selected. The speed and reliability of the method will be taken into account before choosing between wired or wireless communication. If more than one microcontroller were to be used, a choice between I2C and SPI would have to be made.
- Using Git, all code is tracked and version history stored. This thesis paper will serve as documentation, and the discussion within will focus heavily on future work and expansions. All design elements like 3D models and electronic schematics will be uploaded to the git repository.

1.6 Structure of the report

The report was divided into chapters based on the subject within. This structure was enforced to make it easier to read and reference for future work. Each chapter will contain its own sections on the method, theory results, discussion, and conclusions. Due to the nature of the chapters some of them will contain all of the before-mentioned sections.

The first chapter includes the background for the thesis as well as the project assignment and thesis statement. Problem statements will be proposed, and objectives for how to solve them will be presented. It will also contain a list of definitions for terms

and abbreviations that will appear in the thesis to counteract any confusion.

The second chapter will include an overview of the development and research method used to realize this project.

The third chapter will contain all aspects of the design process. It will present a simple analysis of anatomical proportion and natural gait for selected animals. Thoughts behind all parts that make out the physical model and the workflow to make the 3D model work in ROS is also presented.

In the fourth chapter, the kinematics of the robot will be explained. Methods for implementing the waypoints previously found in the gait analysis will be described using kinematics. Finally, further modeling of the robot to allow for more optimal controllers will be explored.

The fifth chapter will introduce different principles in control theory, followed up by a discussion on what and why the chosen control strategy is implemented in this thesis. This will mainly serve as a theoretical chapter, and the discussion and result sections are thus joined.

The sixth chapter will describe everything related to the embedded system. The motors, gearing, microcontrollers, and communication methods will be presented and discussed.

In the seventh chapter, the implementation of ROS in this thesis is explained. This includes a theoretical framework for the basic principles of the ROS features used. How trajectories are executed using *MoveIt*, and further work on robot locomotion using ROS will be discussed.

The final three chapters will include discussion, results, and conclusions for the project as a whole. The possible future work for the project will also be presented.

1.7 Definitions

Term	Definition
ADC	Analog-to-digital converter
API	Application Programming Interface
Biomimetics	Emulation of natural configurations or processes in man-made products
CAD	Computer-aided design
CoM	Center of Mass
CPR	Counts per Revolution (Encoders)
DAC	Digital-to-analog converter
DH	Denavit-Hartenberg
DOF	Degrees of freedom
EEPROM	Electrically Erasable Programmable Read-Only Memory
Gait	Position and angles for all joints and end effector during a complete walk cycle
HSE	Health, Safety, and Environment
I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
LQR	Linear-Quadratic Regulator
Lidar	Light Detection and Ranging
MIMO	Multiple-input and multiple-output
Odometry	Estimated change in position and orientation from start position
Pose	Position and orientation of all joints in a robot
PWM	Pulse-Width Modulation
ROS	Robot operating system
SCL	Serial Clock Line
SDA	Serial Data Line
SLAM	Simultaneous localization and mapping
SPI	Serial Peripheral Interface
STL	Filetype representing design using only triangles
URDF	Unified Robot Description Format
ZN	Ziegler-Nichols

Chapter 2

Method

2.1 Development method

During the preliminary phase of the project, a Gantt chart was made to plan the development process. However, the need for a more agile development method was also discussed (Arnesen, Grinde, Hovland and Vestland 2021a). The project consists of different components like hardware and software, which can be worked upon independently. A general workflow representing the waterfall method is presented in the Gantt chart shown in *Appendix B*. Within these different components, the agile development method Kanban was used. To support the use of Kanban, Trello was used as a Kanban board. A Kanban board allowed easier management of the development process as new features were needed or physical parts needed redesign. Figure *Figure 2.1* shows how the Kanban board for Arduino software was set up to track features and bugs. Limiting the number of WiP and tasks needed to be tested is a key structure of Kanban. More information about Kanban and agile development is readily available in software engineering textbooks Sommerville 2015.

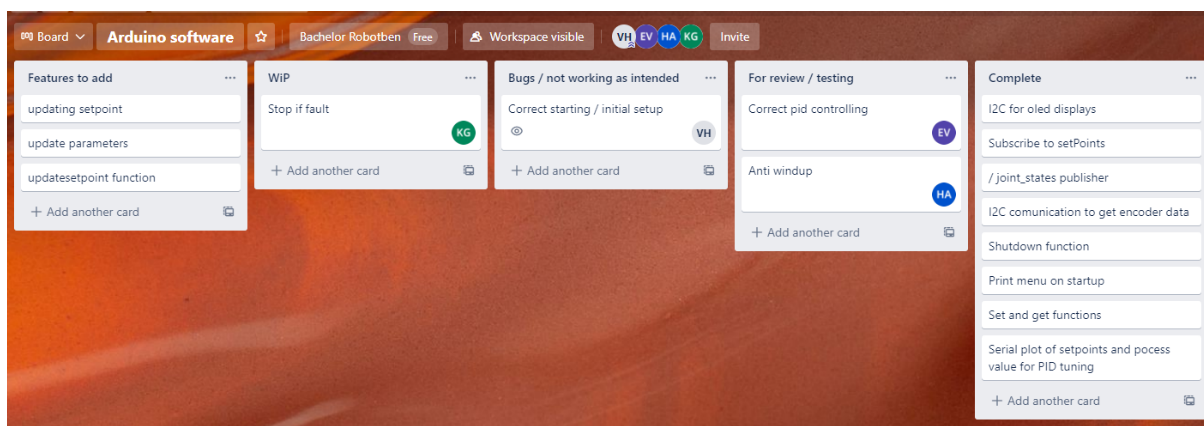


Figure 2.1: Kanban board for Arduino Software

This project contains a lot of written software for its different parts. Multiple languages

are used, and some of the software was either worked on or used by multiple team members simultaneously. A git repository was added to support the agile development method further. Git is used for version control and collaboration. When features were added or the bugs listed on the Kanban board were fixed, the changes were committed to git. A public git repository also provides an easy code distribution for further work and collaboration. The final commit to the Git repository is provided as a downloadable ZIP file (Arnesen, Grinde, Hovland and Vestland 2021b). The software written in this thesis is either shown in the chapters, provided in the appendix, or referred to the Git repository, depending on size and importance.

Microsoft teams were used for communicating within the group for sharing files connected to OneDrive. The combination between Git and MS teams allows for completely remote work if the Covid situation demands it. It also ensures that each member is always up to date with the latest code or figures. Bi-weekly meetings and communication with the thesis supervisor were also done using teams. For writing this thesis, LaTeX was used as the main word processing program. Like MS Teams and Git, this enables the group to work on the same file simultaneously. LaTeX also comes with Git implementation for version control.

2.2 Research method

Developing a robot leg is a multidisciplinary task and requires many different parts to work together. An effort to divide the different parts into separate functions was made. This allowed the team to focus on individual tasks and narrow their research, and thus the research method varies between the different tasks. In general, an effort to use primary or other high-quality sources was made. Some of the tasks were entirely new for the team, and thus researching the basics was necessary. Other tasks were based upon the knowledge already acquired from the Electrical engineering program at NTNU.

2.3 HSE and Risk assessment

In the preliminary phase of the project, risk assessments concerning HSE and equipment damage were assessed (Arnesen, Grinde, Hovland and Vestland 2021a). Having a good grasp of the potential risks that may disrupt the project is essential to be able to manage them as well as possible. Therefore, these risks were described and assessed by probability and consequence. The electrical and mechanical risks addressed are soldering, motor testing, 3D printing, and column drilling. The assessment of these risks and risks concerning injury or regarding the COVID situation is included in the preliminary project. In the end, the risk assessment done in the preliminary phase was sufficient and well managed as there were no accidents, and COVID did not negatively influence the project.

Chapter 3

Design

3.1 Introduction

The design, fabrication and assembly of a physical robot leg involve planning for proportions, material, assembly, and functionality. All these aspects must be taken into consideration for the leg to be fully functional. This chapter will give an overview of the design methods and choices made to meet the goals stated in the thesis statement. An analysis of the biomimetic properties this thesis seeks to explore is also included. The design of the individual parts will be explored, and the complete physical model will be presented. Finally, an extensive discussion on the choices made during the duration of the project will be conducted.

3.2 Method and Equipment

3.2.1 Parametric modeling principles

The goal of parametric modeling is to enable editing important parameters without having to do a complete redesign. In this project, the parametric model was implemented to design every aspect of the robot's leg and stand without having all the precise measurements for every part and to ease changing dimensions later on. An example of this is creating the slots for the captured nuts without having the width or height of the nut. In Fusion 360, using a plugin to import and export user parameters (Autodesk 2021), the entire project was set up using one CSV database for all essential and recurring parameters. This enables adjustments to be made in this one database for use in all designs. The drawback or challenge with using this method is that all different designs must be opened and updated if a parameter is changed. This also forces all group members to always import the database upon opening a new or existing design to ensure the parameters are updated.

3.2.2 3D-printing

Most of the parts for the robot leg and stand were manufactured using additive manufacturing in the form of 3D printing. This method involves adding thin layers of molten plastics. In this project, polylactic acid (PLA) and polyethylene terephthalate glycol (PETG) was used. PLA is an easy-to-use material with no dangerous vapors during printing. This makes PLA a popular material for 3D printing. The downside is that it is a brittle material that can easily break. The durability of PLA is also low, making it less suitable for long-term use. The upside is that if parts break or degrade beyond usability, they can easily be replaced since it is so readily available. PETG is another plastic material that is perfectly safe for use. Compared to PLA, it is both stronger and more durable for long-term use. However, when printing, it can be a more difficult product to handle since it is hygroscopic and will draw moisture from the air. To avoid this, PETG should be stored in a dry box with some method of desiccating the air. Before use, PETG should also be dried to increase the quality of the print. Both of these precautions might not always be available at every printing location, so the usage of PETG should be weighed against the risk if multiple tries is not an option (Dwamena 2021).

3.2.3 Captured nuts

For both the leg and stand design, captured nuts were chosen to hold the parts together. This method enables the robot to be disassembled without wearing out the plastic. The captured nut concept used in the links and other parts can be seen in *Figure 3.1*. The first method involves having slots that the nut can be pushed into sideways. This method is best for large parts where long screws would otherwise be needed. The other method uses a recessed hole, with the same size as the nut, at

the opposite side of the screw head. The screw then passes through both parts. This method works best for thinner parts.

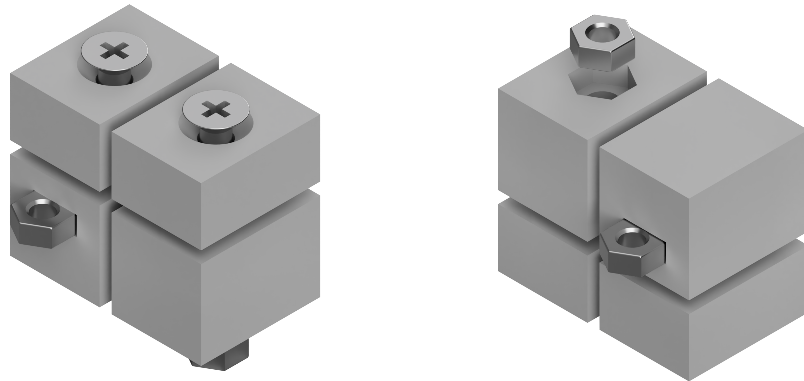


Figure 3.1: Captured Nut Design

3.2.4 Software

Name	**Description**	**Documentation**
Affinity Designer	Vector design program	Affinity 2021
Autodesk Fusion 360	3D modelling software	AutoDesk 2021
Ultimaker Cura	3D-print slicer	Ultimaker 2021b

3.2.5 Equipment

Name	**Description**	**Documentation**
Raise 3D Pro2 Plus	Big format 3D-printer	Raise3D 2021
Ultimaker 2+	3D-printer	Ultimaker 2021a
Ultimaker 2+ Extended	3D-printer with extended build-height	Ultimaker 2021a

3.3 Results and Empirical Findings

3.3.1 Anatomical Analysis

As stated, one of the goals of this thesis is to design a setup that can test a four-dof robot leg gait. An analysis of some quadrupedal animals was undertaken to determine a set of possible link proportions, a. In our configuration, the different links corresponds to animal bones like stated in *Table* 3.1.

Link number	Animal counterpart
Link 1	Hip displacement
Link 2	Femur - Hip
Link 3	Tibia - Calf
Link 4	Metatarsal - Foot
Link 5	Phalanges - Toes

Table 3.1: Link and animal comparison

Two animals, feline (cat) and canine (wild dog), were chosen to have some options for different proportions. As seen in *Table 3.2*, the cat data (Gospodarek 2019) differs little from the wild dog data (Hildebrand 1952). The two data sets use a different metric for the total length of the animal limb. The feline data includes phalanges, while the canine source measures femur + tibia + metatarsal as the total length. With the canine source missing data for phalanges, the cat proportions will be used going forward with this thesis.

Animal	Femur [%]	Tibia [%]	Tarsal and metatarsal [%]	Phalanges [%]
Cat	32.9	36.43	20.09	10.59
Wild dog	39.6	41.9	18.8	-

Table 3.2: Proportions in percent of total leg length for cats and dogs

In this thesis, the flexibility of the phalanges is discounted. In cats, the phalanges can bend over $180°$. During a normal gait, the phalanges are pointed forward when in contact with the ground. Therefore, it might be a good idea to add an angled part to the end effector. This could enable the test bench to achieve a more natural gait.

Measurements on several images of domestic cats were done to find the proportions for the base height. Using the ratios found earlier, the leg length to height proportion was found to be 0.65.

3.3.2 Gait Analysis

It was decided to find some waypoints using motion studies for cats to get a good starting point for the path and trajectory planning in Matlab. Using the same research by Gospodarek used in the anatomical analysis (Gospodarek 2019), some waypoints could be determined. Taking a picture from Gospodarek's study (*Figure 3.2*), the relative lengths of the links were first measured. These measurements were then compared to the proportions found in the anatomical analysis to find a proportional value for y- and x-values that match the actual length of the robot. By measuring the three longest links, finding the proportional value for each link, and taking the average, the proportional value was found to be 2.14.

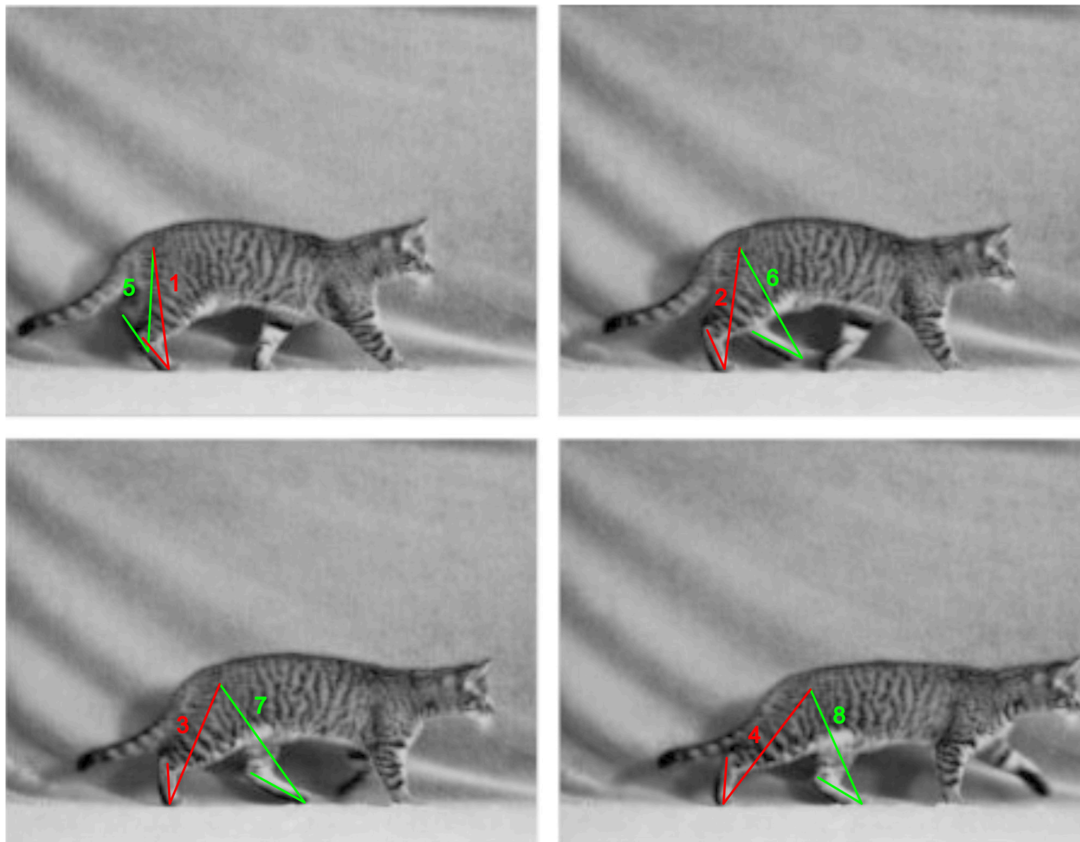


Figure 3.2: Gait analysis - Finding angles and positions from slideshow of cat gait

In *Figure 3.2*, the position of the hip was approximated. Lines from the hip to the paw and along the tarsal bones were added for each picture. Then trigonometry and the Pythagorean theorem were used in combination with the proportional value was used to find values for y,z, and φ in mm. No values for x can be found using this method, as the analysis is in 2D. The result can be found in *Table 3.3*.

Waypoint	1	2	3	4	5	6	7	8
x [mm]	0	0	0	0	0	0	0	0
y [mm]	60	-60	-198	-344	-21	245	335	202
z [mm]	-520	-520	-520	-520	-451	-477	-511	-520
φ [deg]	48.9°	65.8°	86.9°	94.1°	55.1°	28.1°	28.8°	31.0°

Table 3.3: Gait analysis - Waypoints

The chosen angle for the phalanges would have to be taken into account to use this set of coordinates and angles.

3.3.3 Robot Configuration

As shown in the section on the mathematical model, the configuration of the robot leg is as shown in *Figure* 3.3. Here we can see the four actuators and the links and axis

needed to model the leg correctly.

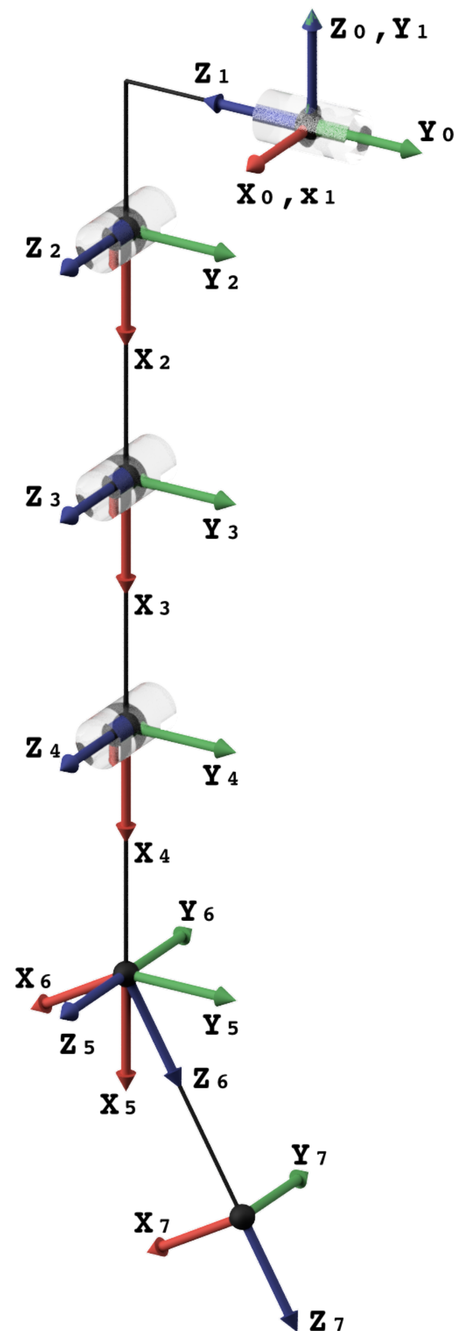


Figure 3.3: Robot-leg Configuration

Since almost all dimensions of this robot are found parametrically, the length of the leg can be chosen arbitrarily. To keep the robot portable but still visually striking, a total length for the leg of 800mm was decided. The total length of the links was found using the proportions found in the anatomical analysis in *Table 3.4*.

Link:	Femur	Tibia	Tarsal	Phalanges
Length [mm]:	263.2	291.44	160.72	84.72

Table 3.4: Actual link lengths

Degree of actuation

Since the configuration has three actuators in one plane and one in a perpendicular one, the robot is under-actuated when looking at the complete robot. By looking at the robot as a planer robot, the robot can be seen as over-actuated with only the three outermost actuators.

3.3.4 Leg Design

The leg was designed around the possibility to change the length or proportions of the leg. Therefore the link designs were made into separate pieces. This also simplified the manufacturing process as the 3D printers available had limited printing volume. The 3D-print slicing parameters were tweaked with thicker walls and a higher percentage of infill to get the most strength possible for each part. The orientation of the part on the 3D-print build-plate was also considered. This was still limited by the time restraints MAKE@NTNU has for each printing session.

Motor encoder cap

The motors only have an axle on one side, and the entire weight of the robot would therefore rest on this one axle. A new plastic cap for the encoder was designed with an axle to remedy this. This way, the weight could be distributed on both sides of the motor. The axle and upper link were designed so that a good clearing would exist between the parts. The parts experienced little friction but could be made smoother by applying some dry lubricant, like graphite, to reduce it further. The end-cap design can be seen in *Figure 3.4*.

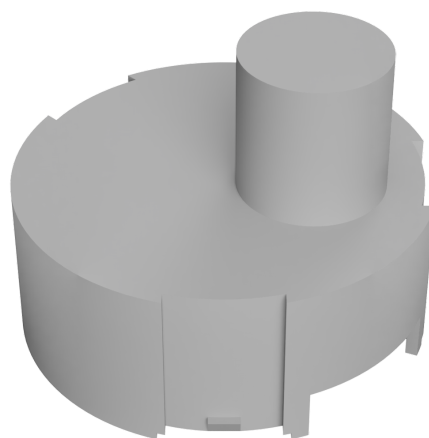


Figure 3.4: Link Design - Motor encoder cap

Upper link

The upper link connects the mounting hub on the motor axle to the middle link section. This part was designed with a recessed hole to fit the mounting hub to minimize the total width of the joints. Mounting holes were added to the bottom and backside of the part to enable mounting both downwards pointing links: like the femur, tibia, and end effector; and the backward-facing hip.

The biggest challenge with this part was printing it in a way that maximized the width of each printed layer. The body of the part is 10mm thick, and the design includes $90°$ angles. Therefore, angling the design $45°$ of the 3D-printers build plate will increase the width of the layer from 10mm to about 14mm. This increase will help strengthen the layer adhesion and, in turn, increase the strength of the part.

The clearance between the upper and lower link with the motor attached is quite small. Therefore, a correct angle when mounting is needed together with a small amount of force. The final piece is the bracket to lock the upper link to the end-cap axle. Since the piece only measures 10mm, a captured nut solution was ruled out not to create weak spots in the material. A solution where the plastic itself was threaded so that the screw mated directly to the plastic was chosen. A chamfer was added to each side of the mounting hub to ease the assembly of the part. A chamfer was also added to the inside of the end cap axle hole. Both of these chamfers ensure that the part can be mounted without bending or breaking. The design of the upper link can be seen in *Figure 3.5*.

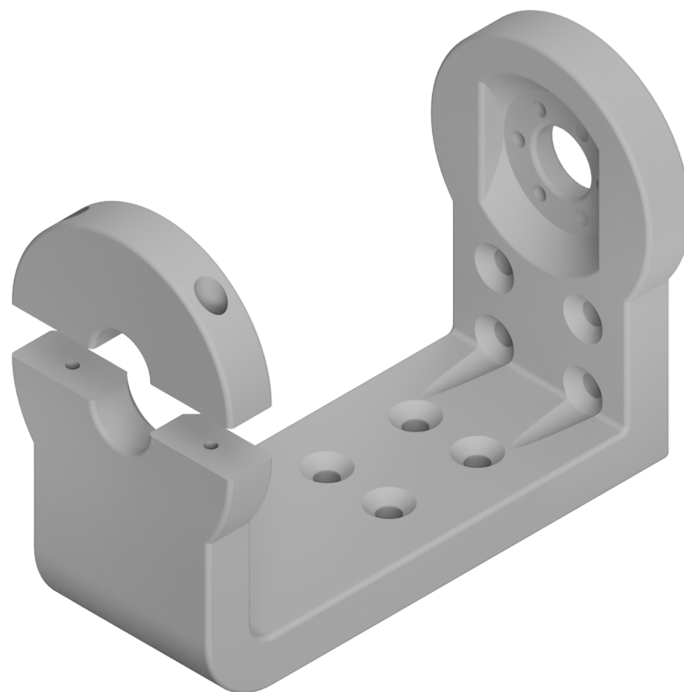


Figure 3.5: Link Design - Upper link

Lower link

The lower link connects the motor to the middle link section. The reasoning behind the design of the part is similar to that of the upper link. The main difference is that it screws directly into the motor on one side and that the bracket goes around the body of the motor. The bracket then helps to stabilize the joint and spread the load on the part. The design of the lower link can be seen in *Figure 3.6*.

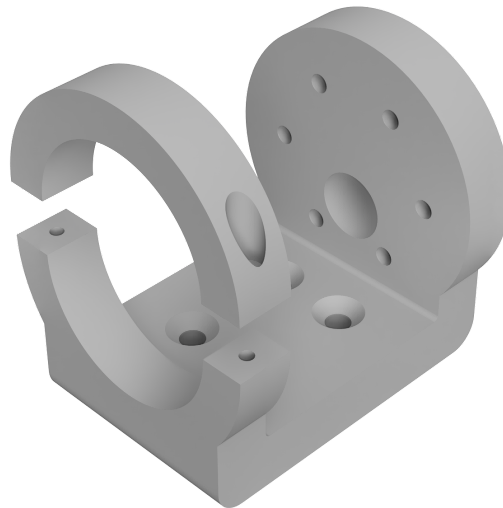


Figure 3.6: Link Design - Lower link

Middle link design

A simple, sleek design with a hollow center to facilitate neater cabling for the motors and encoders was chosen for the links. Due to the hollow center, the design was split in half so that the support structure needed in the 3D printing process could be removed. Since the wires are entering opposite sides of the link, a single flipped design could be used for both sides. Both the femur and Tibia links use the same design but with different lengths based on the proportions found in the anatomical analysis. The finished design for the femur and tibia can be seen in *Figure 3.7*.

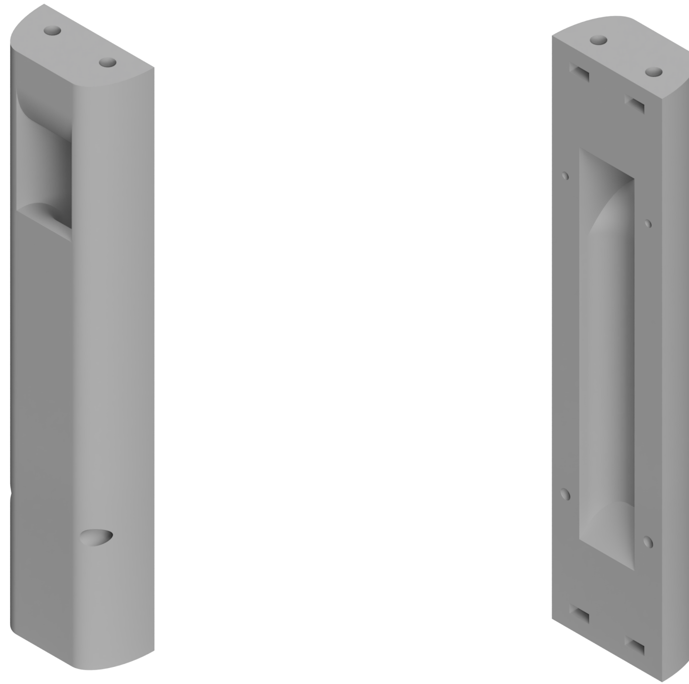


Figure 3.7: Link design - Femur and tibia

The hip link is needed for connecting the base link with the rest of the leg. It needs to do this without building too much in any direction. A design was made that would fit within the dimensions of the joint pieces so that it would not impede any movement of the leg. The hip design can be seen in *Figure 3.8*.

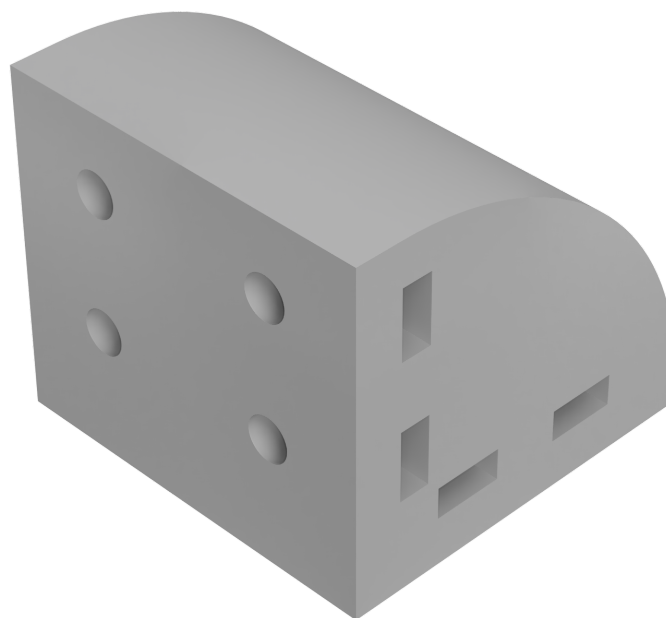
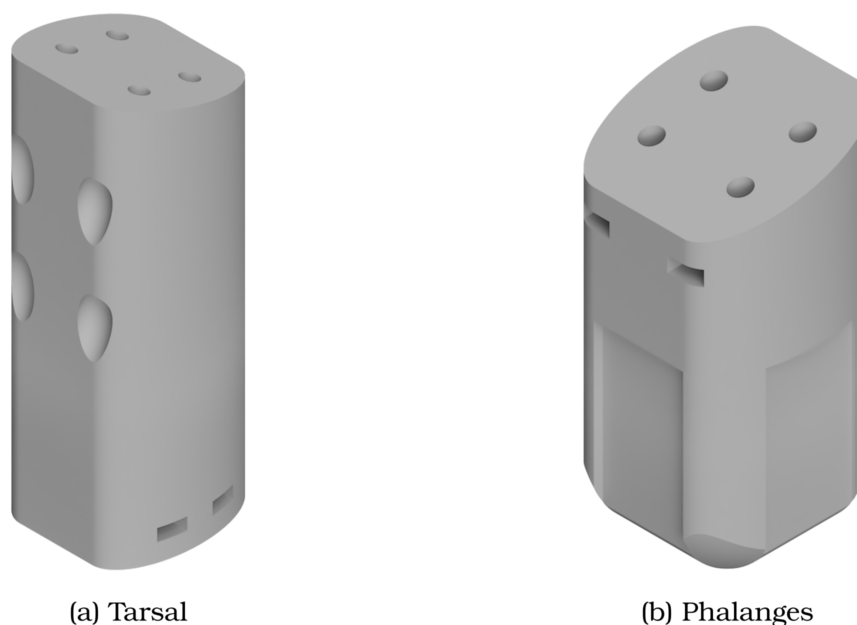


Figure 3.8: Link design - Hip

End effector

The last link and end effector did not need a hollow center to accommodate any wire harness. Therefore the parts were designed without the split used in the femur and tibia designs. This change meant that the captured nut had to be inserted from the outside of the design. This last link represents both the tarsal bones and phalanges from the anatomical analysis. As stated in this analysis, the angle between the tarsal and phalanges varies greatly at different stages of the gait. Since the robot has this as a fixed joint, the angle was set as a parameter in Fusion 360 together with the length of both parts. This enables the printing of new parts with different angles to be printed for future testing. In the final prototype, this angle was set to 30°. On the part where the end effector comes in contact with the ground, a 3.5 mm rubber sole was added to increase the friction. The design of the end effector can be seen in *Figure 3.9*.



(a) Tarsal (b) Phalanges

Figure 3.9: Link design - End effector

3.3.5 Stand Design

The primary responsibility of the stand is to secure freedom of movement and stability for the leg. It also serves as a platform for the electronic hardware used in this project. In addition to 3D-printed parts, the stand consists of two sets of aluminum pipes, used as reinforcement for the top plate and as legs for the stand, and four ball-casters to allow the robot leg to move the stand. Each support leg is tilted 20° out diagonally from the corners of the plateau. This enables greater maneuverability for the robot leg by widening the area between the aluminum pipes and improving the stability of the stand.

Top plate

The top plate had to be designed so that the electronic hardware would fit neatly on the plateau, as well as having easy wiring access to the leg. A 20×20 cm quadratic shape was chosen as a reasonable build. To pull wires from the actuators to the drivers and controllers, a rectangular 25×35 mm hole was placed at a logical location. The motor drivers only had two mounting holes, so two rectangular extrusions were added as support. To make sure the robot leg had a robust mounting point, an 18×4 cm metal panel was added on the underside of the plate.

The top plate was printed in PETG due to being the component with the most mounted parts, including 44 screw holes. This was done to compensate for the weaknesses added by all the holes but also to add general strength and stiffness. The design of the top plate can be seen in *Figure* 3.10 and *Figure* 3.11.

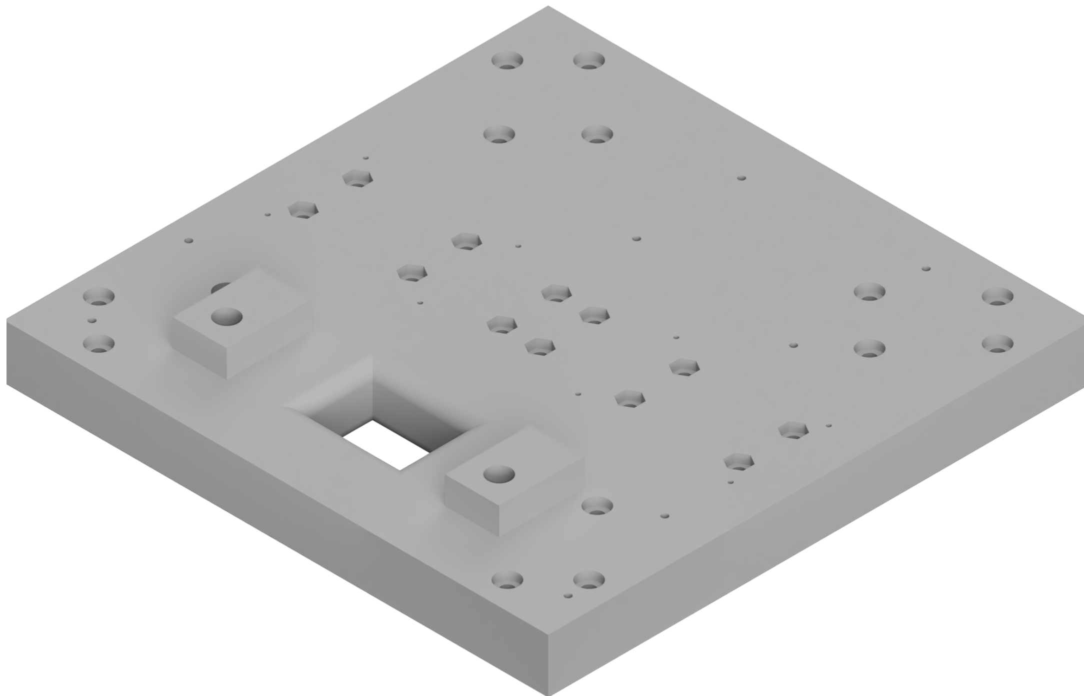


Figure 3.10: Stand Design - Top Plate Top

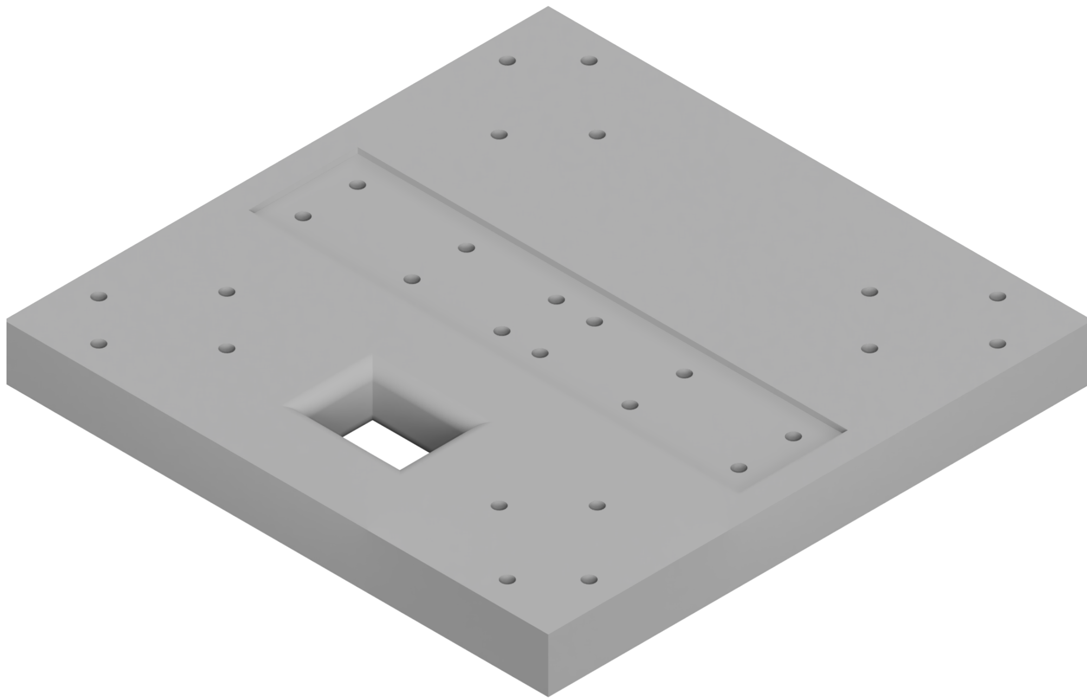


Figure 3.11: Stand Design - Top Plate Bottom

Plate to pipe connector

A connector to mount the aluminum pipes were added in each corner on the underside of the top plate. The connectors are linked with each other through additional metal pipes, adding to the sturdiness of the stand. The lower part of the connectors, mounted to the pipes, is tilted at a $20°$ angle diagonally outwards from the corners of the plate. Each metal pipe is fastened with a screw and a captured nut. The connectors themselves are mounted to the top plate using four screws and captured nuts. The design of the connector can be seen in *Figure 3.12*.

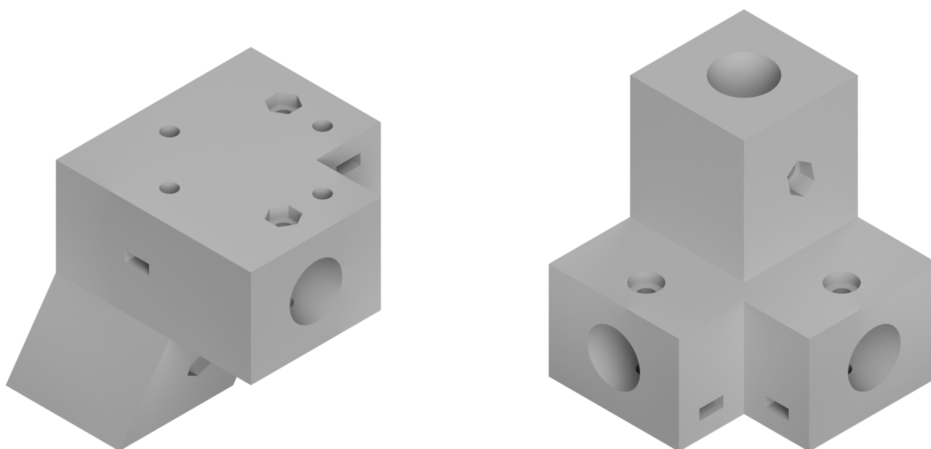


Figure 3.12: Stand Design - Pipe Connector

Caster to pipe connector

The connector between the ball casters and the aluminum pipe is angled $20°$ in the opposite direction to the pipes. This allows the wheel to meet perpendicular to the ground. The aluminum pipe and ball casters are both fastened using captured nuts. The design of the connector can be seen in *Figure 3.13*.

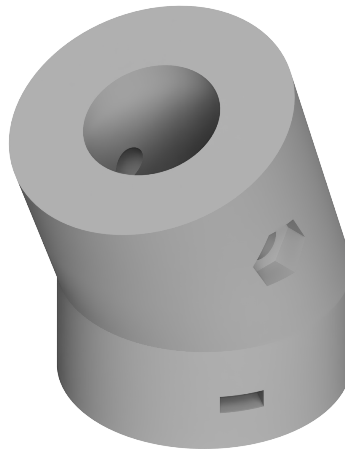


Figure 3.13: Stand Design - Castor wheel connector

3.3.6 Complete Assembly

The robot leg and stand is assembled as shown in *Figure 3.14*. The leg is mounted to the underside of the stand using a spacer to allow free movement of the first joint and the metal plate mounted to the underside of the top plate.

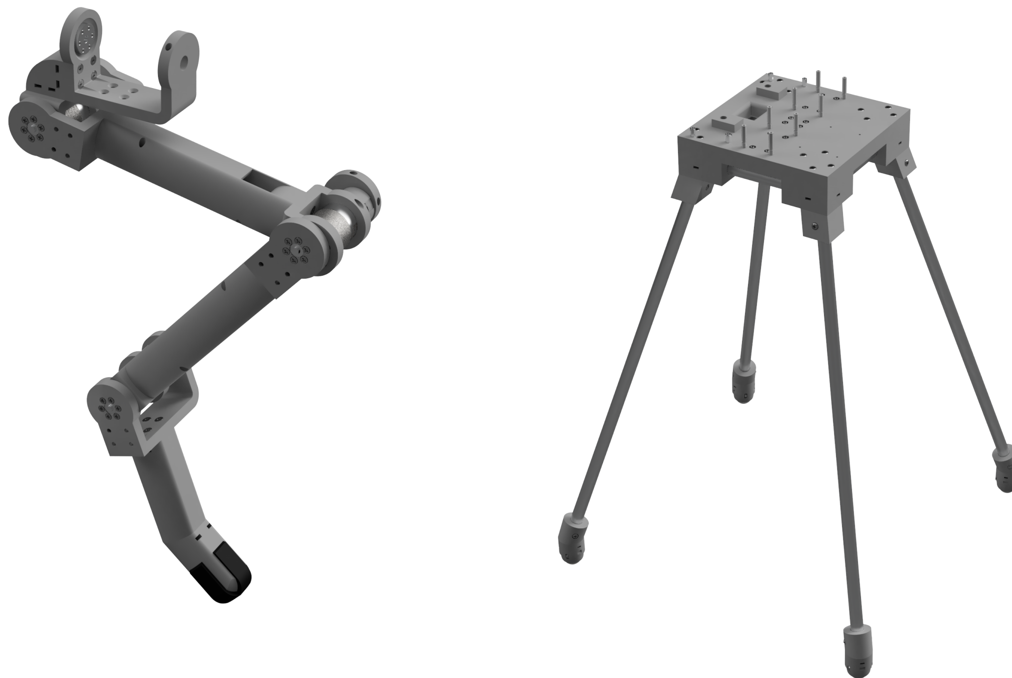


Figure 3.14: Complete Assembly - Leg and Stand Design

3.3.7 Order of operations

Due to the separate joint and link design and the use of captured nuts to connect them, the entire design has a specific order of operations for the final assembly. An example of this is that the joints need to be secured to the links before mounting the motors. This is because the countersunk screws that mount them together rest between the motor and joint after assembly. The mounting of the leg to the stand requires the attachment of the base_link and hip to the top plate through the metal plate. The rest of the leg has to be joined piece by piece, in no particular order, until the assembly is complete.

3.3.8 URDF Export

To simplify creating a robot model to use in ROS and get an accurate simulation of the model, a script to export a URDF file directly from Fusion 360 was used (Fischer 2021). A strict structure had to be followed, and all joints had to be either rigid, rotating, or linear to use the script. A starting point had to be defined as "base_link," and all joints would have to fork out from this part. This structure prevents circular links that could have broken the model. In this design, a spacer placed between the top plate and robot leg was chosen as a natural base_link. The link tree would look something like in *Figure 3.15*. Fusion 360 creates the moment of inertia matrices for all components in its designs. By selecting the real-world material for each part, these calculations can achieve better accuracy. How the robot model is implemented will be presented in n *chapter 7*.

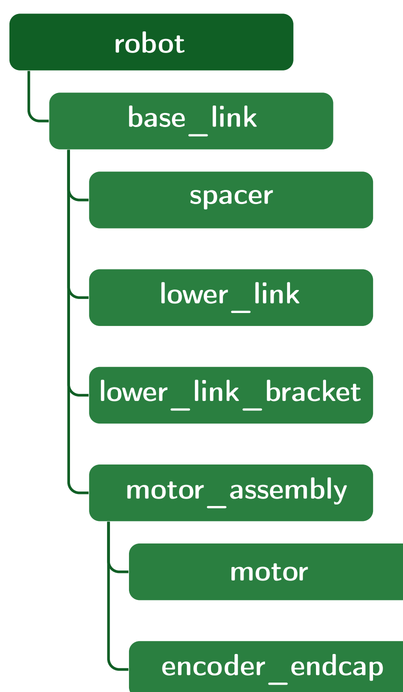


Figure 3.15: Part configuration-tree in Fusion 360

3.4 Analysis and Discussion

3.4.1 Anatomical and Gait Analysis

In an actual animal, the hip is a ball and socket joint. In robotics, this is considered a spherical joint that needs three degrees of freedom to be emulated. As this robot leg only has two, some natural motions are impossible. The same can be said for the ankle and missing degrees of freedom. Even though this joint is not a ball and socket joint, it is often modeled with three degrees of freedom to make all motion possible, like the wrist. Finally, as proposed in the anatomical analysis, the phalanges angle changes during the normal gait cycle, even modeling all phalanges as one means adding another degree of freedom to the system.

All these extra degrees of freedom would add four more actuators to the leg, bringing the total links in the Denavit-Hartenberg table to at least eleven. This would make the mathematical model increasingly complex with the inverse kinematics hard to solve and make the robot much heavier and bulky needing smaller actuators.

Creating four identical legs (two mirrored) and mounting them to a free-floating base would be a logical goal for future research. This quadruped model would enable a natural gait where the interaction between the legs and the body would come into play. In addition, this would also complicate all mathematics and make a simple gait analysis, like the one presented in this thesis, be less optimal. A simple visualization can be seen in *Figure 3.16*.

Figure 3.16: Quadruped robot visualization

3.4.2 Method of Mounting

It was considered using linear rails instead of a stand with wheels as support for the robot early in the design process. A two-railed track with the robot leg in the middle would probably require less 3D-printed material than a stand. The bulk of the material would be the rails and a 3D-printed mount for connecting the leg. This would give more stability to the robot, as there would not be any concern with the level of friction in relation to ball casters or the possibility for the stand to tip over.

The rail-based solution eliminates any movement in the xz-plane, making the first joint redundant. Sacrificing this degree of freedom would make this a two-dimensional robot in terms of future work or research. In addition to expanding the robot's freedom of movement, the stand design requires less setup and is easier to carry. This improves the ability of the robot to be used for educational and demonstrative purposes. Since the goal of the thesis is to create a robot for future use in demonstrating and analyzing gaits, the stand solution gives most options in poses and possible waypoints.

3.4.3 Leg Design

During assembly, it was discovered that the order of operations for mounting the leg together meant that everything had to be assembled in a specific order. For example, link_1 had to be connected to the base_link before the stand was connected. This complicates the assembly process and makes swapping parts or taking the robot apart harder.

Upper link design

The upper-link design with the recessed hole for the mounting hub was one of the designs that made this problematic. When the upper and lower links were properly mated, only about 2 mm was set aside as clearance for the parts to move freely. This clearance meant that there was too little space when putting them together. Manually chamfering the edges of the recessed hole made this easier, but only so much plastic could be removed before removing the outer shell from 3D printing. Later the entire design was changed to the one presented in *Figure 3.5*. The old design can be seen in *Figure 3.17*.

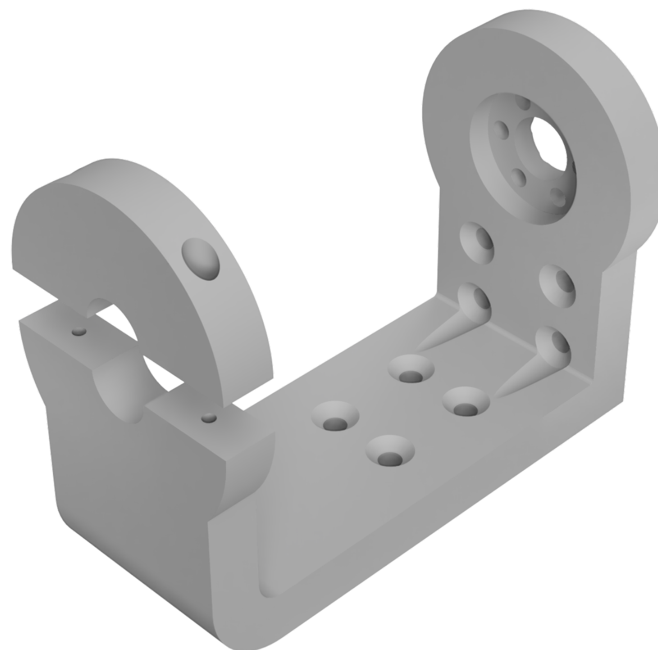


Figure 3.17: Link Design - Upper link, old design

When designing the parts, many measurements were taken from the supplier's websites. These measurements include the recommended through holes for the screws. While these dimensions probably work as intended on metal, they were loose for this project and introduced some play between the part and the mounting hub. Therefore, the through holes were reduced in the new design to make the holes clamp tight around the screw.

Lower link design

During testing, the force of the movement stripped the bracket threads that supported the first actuator in the chain. A quick re-design (*Figure 3.18*) that increased the contact area for the bracket and doubled the number of screws used was made. This design still has threaded plastic, but it will hold temporarily, with the leg operating at a lower speed. Finding a design that uses threaded inserts or captured nuts would be advised for further work.

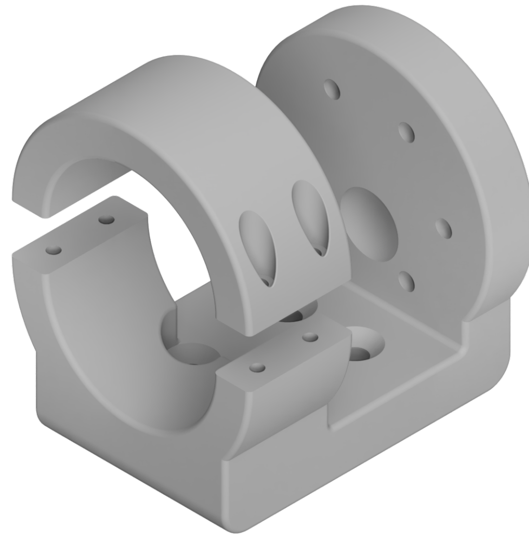


Figure 3.18: Link Design - Lower link, new design

Asymmetric link design

When designing the link, the idea was to make it primarily symmetric, resulting in a theoretical angle span of $\pm 90°$ for the forward/backward motion of the hip and about $\pm 15°$ for the sideways motion depending on the configuration of the other joints. The rest of the joints have a theoretical span of $\pm 125°$. As the robot is meant to emulate a four-legged biological animal, many of these angles would not be reached during normal gaits. Therefore a joint design where the span was not symmetric could increase the angles available for the gait. An example of this is when the leg returns from the point furthest back in the gait to the front. The current design gives less than 10 cm clearance to the ground with both the knee and ankle joint contracted. By changing the design to something asymmetric like in *Figure 3.19*, the span could be extended to about $180°$ in the direction needed for the link.

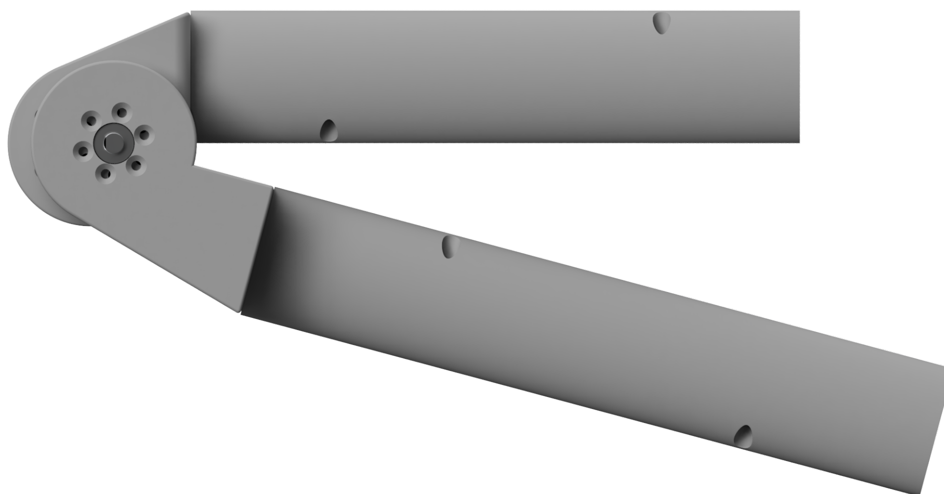


Figure 3.19: Asymmetric link design

Future work

Future expansion of the project could include sensory feedback on how hard the foot is pressing down on the surface. This feedback would enable better control of how the robot interacts with the ground and ensure that the robot gets enough friction when moving. Adding this feature would mean redesigning the tarsal and phalanges parts to facilitate the wires and placement needed for the sensor. This expansion is further explored in *subsection 6.5.8*.

3.4.4 Stand Design

The design of the plateau ended up in a 20×20 cm quadratic configuration. These dimensions were mainly chosen due to the physical limitations of the 3D printers. It would be possible to increase the width of the plate by using a larger printer, making room for more movement for the θ_2 angle on the robot. A larger top plate would also make it easier to place the electronic components in an orderly disposition.

Another way to make more room for θ_2 is to increase the tilt of the pipe connector. The current tilt of $20°$ allows for a rotation of $\pm 17°$ for the leg on the xz-plane. This flexibility is enough to turn the stand and enables two-dimensional motion on the ground. A larger tilt could let it turn faster. However, if this angle is increased, the stand support legs must be made longer for the robot base to remain at the same height. Longer support legs increases the torsion force in the plate-pipe connector, making it more prone to breakage. This again could be resolved by making a more robust connector.

Caster wheels

The caster wheels were added to lessen friction between the stand and ground. The balls in the casters rest on three ball bearings to reduce friction further. The casters rotate freely when tested independent of the stand, but the weight of the stand seems to introduce more friction making it harder to move. However, the wheels still reduce the friction compared to having no wheels, so it is still a net positive. Reducing the friction beyond the current level might need to be considered when the robot leg is fully operational.

3.4.5 3D-Print and Material

Due to the number of parts, the single operational 3D printer at the department would not have the capacity to print all the parts in the given time. Therefore most of the parts were printed at MAKE@NTNU. MAKE have time restrictions of 6 hours during the day and 16 hours for overnight printing. Many parts filled up the 16-hour limitation even at lower settings, and some needed over 24 hours to print with acceptable settings. Even with some special approval to print the most extensive parts on weekends, most parts were sliced with settings lower than optimal. If the parts

were remade or upgraded, thicker walls and thinner layers should be considered to strengthen the model.

Another method previously mentioned is to angle the part on the 3D-printers build plate. Angling the part is a good way to increase the strength of its layers, but it also makes it more prone to printing failures and increases the print time significantly. Because of this, only some parts were printed this way. This would also be an aspect to consider if the parts were to be replaced. In the end, the final pieces printed were printed on their sides so that the layers would line up with the longest sections of the parts. This orientation seems to increase the strength beyond the 45° tilt used for most of the parts. The difference can be seen in *Figure 3.20*.

A minor detail that can be seen on the finished physical model is that all parts are printed in different colors. The reason for this comes from printing most parts at MAKE@NTNU, where the general rule is to use up any open filament spools before opening new ones. This only has an aesthetic effect since the PLA filament in different colors should have the same properties. Future work could include painting the parts or investing in filament for new parts if a more coherent color scheme is preferred.

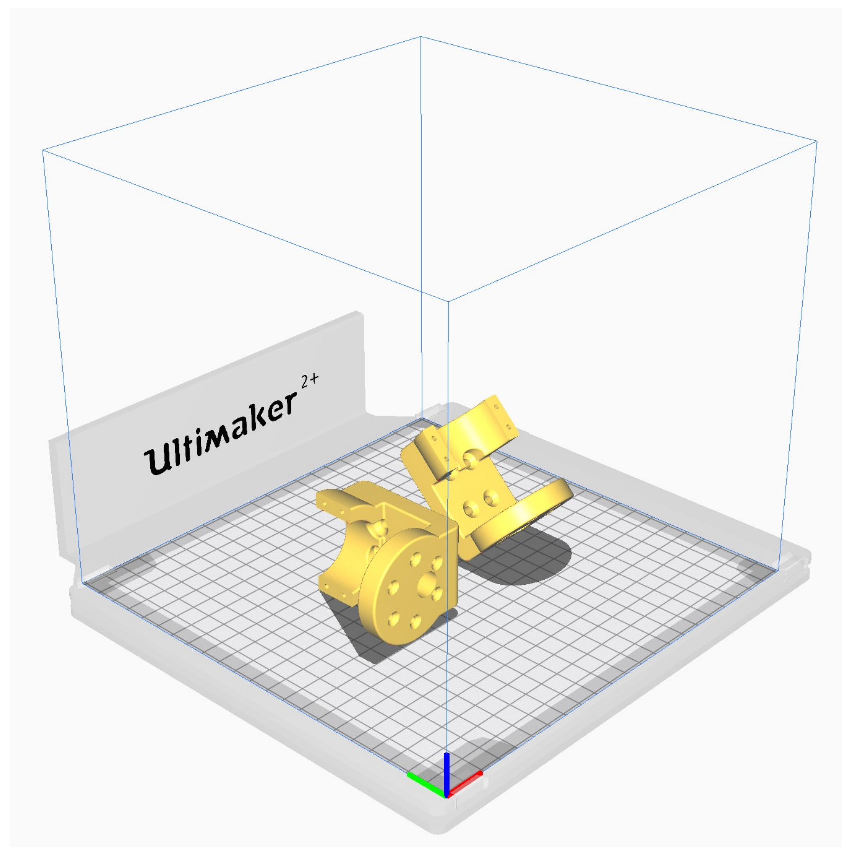


Figure 3.20: Design on its side and tilted 45° on printer build plate

Other than the cost perspective, one of the main reasons for going with 3D-printed parts is the weight of the final leg. The downside to this is that there is little control over the physical properties of the robot. The layers in a 3d-printed part are a weak

point that, on several occasions, caused parts to split during assembly. A better solution for a project like this would be to cut the parts from nylon. Nylon parts would require stronger motors since solid nylon has a greater weight than standard 3D-printed parts. Another option would be to completely design each part down to the infill so that the inertia matrix for each link could more accurately be determined. By 3D printing and designing the links in separate parts, it would be easy to re-design parts if needed. This will make things simpler when finding new actuators for the robot.

The screws were fastened directly to the plastic for some of the connections. Making threads in plastic can be challenging due to the plastic being soft. One of the holes for the standoffs got stripped and had to be glued in place. For when captured nuts could not be used, a better solution would be to use threaded inserts or standoffs with threaded holes on each side and run a screw through the plastic from the other side.

Finally, the choice of plastics itself introduced non-planned issues. Even though PLA is reasonably rigid, some flex was detected on the assembled model. This flex means that that accurate angles will be hard to achieve. A stronger material like aluminum could be chosen for some or all parts of the robot in conjunction with stronger actuators.

3.4.6 Interdisciplinary Project

A large part of this project can be said to be interdisciplinary due to it including parts from automation, electronics, industrial design, material, and mechanical engineering. Therefore, if this project were to be repeated or built upon, this should be considered. One way to do this is to have separate teams from the different disciplines that would work towards improving their separate parts. Another would be to have an interdisciplinary team that could see the whole scope of the project and thus formulates a better way forward.

Involving more study programs would possibly prevent some of the issues that arose during this project or at the very least mitigate the effects of said issues. For example, students with material or mechanical background could run structural analyses to determine the minimum thickness and infill for the parts. An industrial design student could more easily design the parts and have the prerequisite knowledge to avoid the strict order of operations that must be followed to assemble the robot leg. A final advantage would be that each team member would work more closely with their area of study so that their skill and knowledge in their field would better show in the finished work.

3.4.7 URDF-Export

At the start of the project, the script used was *fusion2urdf* by Toshinori Kitamura (Kitamura 2021). This script had some design specifications that had to be strictly

followed. For example, the base link component had to be defined as "base_link," and no nested or linked components could be used. These limitations meant that all designs had to be done in a single file and that only one person could edit the design at a given time. A new version of the script, forked from the original, by Florian Fischer, was found during the project that made nested and linked components possible (Fischer 2021). This new script enabled a better workflow and the final file structure. Unfortunately, during the work on the model, Autodesk updated how coordinates were calculated in their API. The new method used to calculate coordinates meant that the script used stopped working. This problem cost the project about 25 hours in troubleshooting before the error was found. When asked, Fischer updated his script to work with the changes and prevented further hours used to make a URDF model manually.

The inertia matrices that Fusion 360 creates assume massive parts. This means that the mass and CoM calculations will be wrong when 3D-printing with lower than 100 % infill. To counteract this, one could design all the infill directly in Fusion and then print using 100 % infill in Cura. By doing it this way, one could also perform strength analysis for each part in Fusion. Performing a strength analysis will enable possible problems to be addressed before printing the parts.

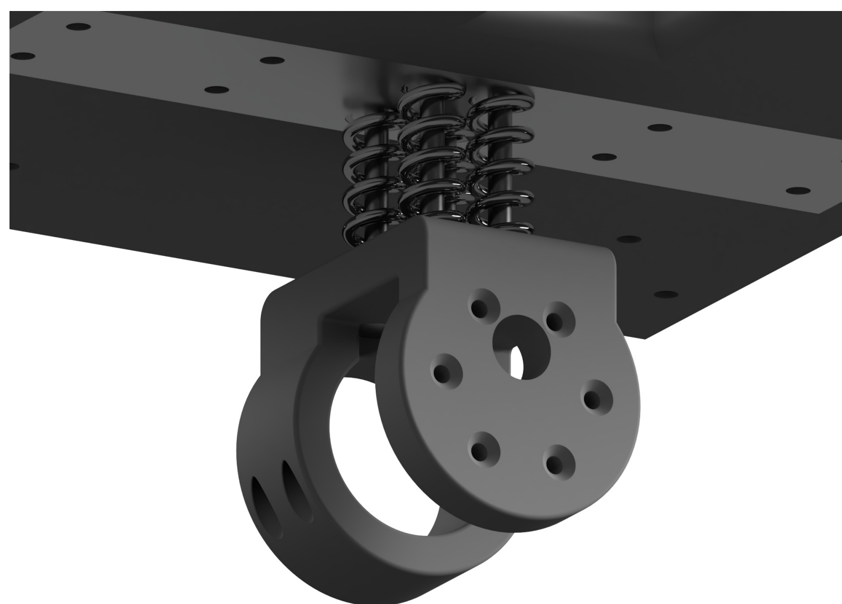


Figure 3.21: Possible spring dampening system

3.4.8 Spring Dampening Addition

In animals, the hip is not at a fixed height. Therefore, it will move up and down during the progress of the gait. The hip also cushions the force spike when the leg contacts the ground. Because of this, a dampening system between the leg and stand could be implemented like pictured in *Figure 3.21*.

3.5 Chapter Conclusion

After the redesigns presented in the results and discussion were implemented, the design of the robot satisfied all aspects of the project assignment. However, issues still exist, and several improvements to the design are discussed. The robot has a limited range due to the stand and the maximum range of movement for each joint.

As presented, the ability to export a URDF file for use with ROS has been a great help and has meant that better visuals and physics for the simulation could be done. In addition, designing the parts to be massive and not semi-hollow like the current parts would improve the control over both the moments of inertia for the URDF export and the general stiffness of the robot.

Due to having 3D-printed most of the parts, the robot displayed more material flex than wanted. Therefore, more robust materials or strengthening brackets should be considered for further work.

Chapter 4

Mathematical Model

4.1 Introduction

Mathematical expressions and algorithms can model and simulate a robot's movement. This chapter covers the theory and calculations of the mathematical framework for the proposed robot leg design. The mathematical framework can be used for more optimal control and simulation. The kinematics, path, and trajectory of the leg will be calculated and visualized using Matlab combined with other tools. Finally, the groundwork for future expansions will be laid down. This will involve finding more optimal control and trajectory planning methods based on more advanced mathematical models. This chapter is heavily based on the lecture series by Anstensrud 2020a, the books by Spong, Hutchinson and Vidyasagar 2006, Egeland and Gravdahl 2002; and a paper on 3R kinematics by Kumar 2021.

4.2 Method and Software

4.2.1 Method

This part of the project was completed using methods and knowledge acquired in the electrical engineering study's mathematics and robotics courses. The kinematics will be found using trigonometry and matrix manipulations, along with conventions and rules from the field of robotics. All parts of this chapter were implemented using Matlab and imported toolboxes. This is also a part of the project where version control was implemented with Git to safeguard against conflicting code.

4.2.2 Software

Name	Description	Documentation
MathWorks MATLAB	Programming language and platform	MathWorks 2021a
Robotics System Toolbox	Functions for robot integration in Matlab	MathWorks 2021b

4.3 Theoretical Framework

4.3.1 Rigid Transformation

Describing an object's position and orientation given in a different coordinate system is useful in applications such as robotics. It is easier to describe the physical laws acting on an object given in its own coordinate system and then transform it to the coordinate frame of interest. One example of this is when describing the forces acting on an airplane. An observer on the ground is in the center of the coordinate frame a placed on the earth's surface. Describing the forces on the airplane is done in coordinate frame b, placed at the CoM for the airplane. If the observer wants to describe the forces acting on the airplane in coordinate system a, a rigid transformation between the coordinate systems is needed (Egeland and Gravdahl 2002).

Rotation

Describing a rotation in three dimensions is usually done by using a rotational matrix, $\boldsymbol{R} \in \mathbb{R}^{3\times3}$, with and a three-number representation. These three numbers can represent the rotations roll ϕ, pitch θ, and yaw ψ, describing how the object rotates around the $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$-axis. The rotation matrix can be used to describe any rotation, usually done by a multiplication of multiple basis rotations given by roll, pitch, and

yaw:

$$\boldsymbol{R}_1(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$

$$\boldsymbol{R}_2(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$\boldsymbol{R}_3(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Every possible rotation can be described by the multiplication $\boldsymbol{R}_1(\phi)\boldsymbol{R}_2(\theta)\boldsymbol{R}_3(\psi)$

A vector $\boldsymbol{v}$ given in coordinate frame b can be described in coordinate frame a by the rotation matrix $\boldsymbol{R}_b^a$ describing the rotation from a to b.

$$\boldsymbol{v}^a = \boldsymbol{R}_b^a \boldsymbol{v}^b$$

Translation

A translation in kinematics refers to a rigid body's movement along an axis i for a distance d_i. The movement described in $\boldsymbol{d}$ is a vector consisting of the distance traveled along the $\boldsymbol{x}$-, $\boldsymbol{y}$- and $\boldsymbol{z}$-axis as shown in *Equation 4.1*.

$$\boldsymbol{d} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4.1}$$

Homogeneous transformation

The complete movement of a rigid body can be described by a homogeneous transformation matrix $\boldsymbol{H}$. This matrix, shown in *Equation 4.2*, includes both the rotation and translation to present a compact description of movement. (Anstensrud 2020d)

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{d} \\ 0 & 1 \end{bmatrix} \tag{4.2}$$

$$\boldsymbol{R} \in SO(3) \qquad \boldsymbol{d} \in \mathbb{R}^{3\times 1}$$

4.3.2 Denavit-Hartenberg Convention

The Denavit-Hartenberg convention is used to reduce the number of subsequent rotations and translations needed to define the configuration of a robot. This convention

also enables a shared reference point that makes the robot easier to understand for people outside of the project group. The convention is based around a set of two rotations and translations in a set order given by:

1. Rotation θ around the joints z-axis
2. Translation d along the joints z-axis
3. Rotation α around the joints x-axis
4. Translation a along the joints x-axis

The order of rotation and translation along the same axis is arbitrary, but the order with the z-axis first is set and needs to be followed to satisfy the convention. In addition to the rotations and translations, there are two assumptions that need to be fulfilled:

- DH1: The axis x_i have to be perpendicular to the axis z_{i-1}
- DH2: The axis x_i have to intersect the z_{i-1} axis

If both these assumptions are fulfilled, the rotation and translations given above will result in *Table 4.1* which will then be a unique description of the robots configuration (Spong, Hutchinson and Vidyasagar 2006, p. 68-78).

Link	θ_i	d_i	α_i	a_i
1	θ_1	d_1	α_1	a_1
2	θ_2	d_2	α_2	a_2
3	θ_3	d_3	α_3	a_3
4	θ_4	d_4	α_4	a_4

Table 4.1: Denavit-Hartenberg Table for an arbitrary system

4.3.3 World vs base frame

One concept worth diving into is the difference between world and base frame. The world frame is a fixed point that does not move or rotate with the robot. The base frame is sometimes called the robot frame and is the frame where the robot starts. This frame will not move or rotate in reference to the robot but can move in reference to the world frame depending on the type of robot. For example, for a robotic arm, these two can sometimes overlap, but for a movable robot, the two frames will usually move in relation to each other (Rust et al. 2018). A base frame will be used in this project, and a possible world frame will be discussed.

4.3.4 Forward Kinematics

The forward kinematics of a robot refers to solving the position of the end effector using the rotation and translation of each joint as parameters.

Trigonometric method

For systems operating in two-dimensional movement, the easiest way to solve the forward kinematics is by using trigonometric functions. Starting from the origin, one has to calculate the end effector's Cartesian x and y positions separately. The x value is calculated recursively for n number of links along the horizontal axis from the origin, shown in *Equation 4.3*.

$$x = \sum_{k=1}^{n} a_k \cos(\sum_{i=1}^{i=k} \theta_i) \tag{4.3}$$

The y value is calculated using the same trigonometric logic but going in a vertical direction using sine instead of cosine, shown in *Equation 4.4*.

$$y = \sum_{k=1}^{n} a_k \sin(\sum_{i=1}^{i=k} \theta_i) \tag{4.4}$$

There is no general formula for calculating the forward kinematics of a three-dimensional robot using trigonometric functions, but the same principles can be used. By using the $\boldsymbol{xy}$ projection as a frame of reference combined with the use of Pythagoras Theorem, the length of the robot in the projection can be defined as the radius r. This radius can then be treated as one of the axes in an $\boldsymbol{rz}$ projection.

$$|r| = \sqrt{a^2 + b^2} \tag{4.5}$$

This method requires individual adjustments, and its complexity may vary according to the robot's configuration and the number of joints.

Matrix method

For three-Dimensional systems with multiple degrees of freedom, it is usual to solve the forward kinematics using matrix calculations with the parameters from the DH-table 4.1. This method establishes the homogeneous transformation matrix $\boldsymbol{A}_i$ for each joint of the robot.

$$\boldsymbol{A}_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sequentially multiplying the $\boldsymbol{A}$-matrices produces the matrix $\boldsymbol{T}$ describing the transformation between the joints. For example, doing this with all the robot joints pro-

duces the transformation matrix for the end effector given in the base frame.

$$\boldsymbol{T}_i = \sum_{n=1}^{n=i} \boldsymbol{A}_n \tag{4.6}$$

The direct kinematics is derived from the transformation matrix' last column, where the three first elements correspond with the end effectors Cartesian coordinates (Anstensrud 2020d).

$$\begin{bmatrix} T_{14} \\ T_{24} \\ T_{34} \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4.7}$$

4.3.5 Inverse Kinematics

The inverse kinematics of a robot cover the conversion of Cartesian coordinates to joint parameters. It can be quite a tricky task depending on the complexity of the robot.

Before calculating the inverse kinematics, the forward kinematics need to be determined. The kinematics will provide the equations needed to solve the joint parameters.

Although the forward kinematics is solved for one solution, the inverse kinematics may have anywhere from zero to multiple, or even infinite, solutions depending on its configuration. Multiple solutions are available because different combinations of joint values can produce the same coordinates for the end effector.

For example will a two-dimensional robot with two angular joints and identical links, have the same end effector coordinates for $\theta_1 = \frac{\pi}{4} \quad \theta_2 = -\frac{\pi}{4}$ and $\theta_1 = -\frac{\pi}{4} \quad \theta_2 = \frac{\pi}{4}$

A set of coordinates outside the robot's workspace will produce zero solutions for the inverse kinematics. The inverse kinematics do not factor in physical obstructions. These constraints must be taken into consideration when finding the solution.

Solving the position of the joint values usually requires the use of trigonometric identities, being more complex the more degrees of freedom the robot has (Anstensrud 2020e).

4.3.6 Velocity Kinematics

The velocity kinematics refers to the linear and angular velocity of the end effector in relation to the velocities of the joints. Velocity kinematics are calculated using the Jacobian matrix. The relationship between the body velocity $\boldsymbol{\xi}$ and state velocities $\dot{\boldsymbol{q}}$, with respect to the Jacobian $\boldsymbol{J}$ is shown in *Equation 4.8*.

$$\boldsymbol{\xi} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{4.8}$$

$$\boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{v}_n^0 \\ \boldsymbol{\omega}_n^0 \end{bmatrix} \qquad \boldsymbol{J}(\boldsymbol{q}) = \begin{bmatrix} \boldsymbol{J}_v(\boldsymbol{q}) \\ \boldsymbol{J}_\omega(\boldsymbol{q}) \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{\boldsymbol{6}\times\boldsymbol{n}}$$

The Jacobian matrix uses values from the transformation matrices. Hence, the direct kinematics first need to be calculated. When the values needed are obtained, the matrix is determined by following the steps shown in *Equation 4.9* and *Equation 4.10*.

$$\boldsymbol{J}_{\boldsymbol{v}_i} = \begin{cases} \boldsymbol{z}_{i-1} \times (\boldsymbol{o}_n - \boldsymbol{o}_{i-1}) & \text{For rotating joint i} \\ \boldsymbol{z}_{i-1} & \text{For prismatic joint i} \end{cases} \tag{4.9}$$

$$\boldsymbol{J}_{\boldsymbol{\omega}_i} = \begin{cases} \boldsymbol{z}_{i-1} & \text{For rotating joint i} \\ 0 & \text{For prismatic joint i} \end{cases} \tag{4.10}$$

The fully calculated matrix can then be multiplied with the vector of joint derivatives, producing the body velocity matrix $\boldsymbol{\xi}$. A robot's body velocity refers to the speed of the end effector relative to its base (Anstensrud 2020f).

The process of finding the body velocity can be reversed to find the joints angular velocities, although it requires an inverse Jacobian matrix which often does not exist. For a non-square Jacobian matrix, the method of pseudoinverse matrices can be used as shown in *Equation 4.11*. The calculation of a pseudoinverse matrix is beyond the scope of this thesis but is explained in Krishna 2021.

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{+}\boldsymbol{\xi} \tag{4.11}$$

4.3.7 Waypoints and path

A robot's movement includes several positions it has to occupy to complete its cycle. The positions described by the general coordinates $\boldsymbol{q}$, or by Cartesian coordinates for the end effector, translate to the different waypoints. The sequence of these waypoints is the path of the robot.

The planning of a robot's path has to take into consideration its physical environment. For example, there could be a risk of joints colliding with each other or other physical boundaries, causing material or human damage. It is therefore essential to take precautions when choosing a robot's waypoints (Anstensrud 2019).

4.3.8 Trajectory

A robot's trajectory refers to its movement, speed, and acceleration between waypoints. For a robot moving on a path, an infinite number of trajectories can be generated between each waypoint. Given the range of possibilities, it is not easy to find an optimal trajectory.

One way to generate a possible trajectory is using a polynomial with limitations on start and stop values. For example, a quintic polynomial allows for limitations for angles, velocities, and accelerations at each waypoint, as well as the time duration between waypoints.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{4.12}$$

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \tag{4.13}$$

$$\ddot{q} = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \tag{4.14}$$

The equations can be solved with matrix calculations as shown in *figure* 4.1.

q_0:	start position	q_f:	stop position
$\dot{q}_0$:	start velocity	$\dot{q}_f$:	stop velocity
$\ddot{q}_0$:	start acceleration	$\ddot{q}_f$:	stop acceleration
t_0:	start time	t_f:	stop time

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2t_0 & 3t_0^2 & 4t_0^3 & 5t_0^4 \\ 0 & 0 & 2 & 6t_0 & 12t_0^2 & 20t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} q_0 \\ \dot{q}_0 \\ \ddot{q}_0 \\ q_f \\ \dot{q}_f \\ \ddot{q}_f \end{bmatrix}$$

Figure 4.1: Trajectory equations expressed with matrices

With given start and stop values, the solutions produce functions with respects to time for angle, velocity, and acceleration between the waypoints. These can be implemented as references in possible control systems for the robot (Anstensrud 2019).

4.3.9 Lagrangian mechanics

Lagrange introduced analytical mechanics in his paper Mecanique Analytique (Lagrange 1788). He provided an alternative way to derive the equations of motion based

on algebraic operations, compared to the well-known Newton-Euler method based on the vector formulation of Newton's second law, called newtonian mechanichs. Newton-Euler equations of motion are well documented and will not be discussed in this paper. More information on the Newton-Euler formulation can be found in Spong, Hutchinson and Vidyasagar 2006. Lagrange's formulation is based on kinetic T and potential P energy, which is of importance in control theory as many controller designs are energy-based, (Egeland and Gravdahl 2002). Although the methods to derive the equations of motion are different. Newton-Euler and Euler-Lagrange derive equivalent equations of motion. However, the Lagrangian approach is usually advantageous for complex systems such as multi DOF robots.

The differences between the kinetic and potential energy are called the Lagrangian $\mathcal{L}$ of the system, found by *Equation 4.15*. It is found as a term of generalized coordinates $(q_1, ..., q_n)$. The Euler-Lagrange equations of motion are then found using the Lagrangian. Deriving the equations of motion given in *Equation 4.16* is well documented, such as in Egeland and Gravdahl 2002 and will not be done in this thesis. τ represents the generalized forces associated with its generalized coordinate. An example of a generalized force is motor torque. Unconstrained Lagrange results in a second-order ordinary differential equation.

$$\mathcal{L} = T(\dot{q}, q) - V(q) \tag{4.15}$$

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}(q, \dot{q})}{\partial \dot{q}}\right) - \frac{\partial \mathcal{L}(q, \dot{q})}{\partial q} = \tau \tag{4.16}$$

4.4 Results and Empirical Findings

4.4.1 Denavit–Hartenberg Parameters

Due to the chosen design, extra frames are needed to explain the orientation and placement of the joints. The complete Denavit–Hartenberg table can be seen in *Table 4.2*. Link 1 describes the rotation from a natural xyz-coordinate system to the starting point for the robot, which has the z-axis pointing backward. Links 2 through 5 describe the links between where the actuators are placed, and θ^*_{2-5} are the actuator angles. Link 6 describes the rotation needed to have the final z-axis point out through the tip of the end effector, and link 7 describes the final translation to the end effector tip.

	d	θ	a	α
1	0	0	0	$\frac{\pi}{2}$
2	d_2	$\theta_2^* - \frac{\pi}{2}$	a_2	$-\frac{\pi}{2}$
3	0	θ_3^*	a_3	0
4	0	θ_4^*	a_4	0
5	0	θ_5^*	a_5	0
6	0	θ_6	0	$-\frac{\pi}{2}$
7	d_7	0	0	0

Table 4.2: Symbolic Denavit Hartenberg table for this specific configuration

The link lengths are described by a_{2-5}, d_2 and d_7 where the variables correspond to the physical model as follows:

a_2: Hip height
a_3: Femur
a_4: Tibia
a_5: Tarsal
d_2: Hip length
d_7: Phalanges

In this list, "hip-length" is the distance between joint two and three along y_0, "hip height" is the distance between the same joints along z_0. As can be seen from the Denavit–Hartenberg table and *Figure 3.3*, the base of the robot floats in mid-air. This means that all Cartesian coordinates must be given in relation to this floating base frame. As can be seen in the Denavit–Hartenberg table, θ_6 has not been defined. In this thesis, this angle is given as $\theta_6 = \frac{\pi}{3}$, but can be changed in future research. This angle represents the bending of the phalanges and, as discussed in the anatomical analysis, is not fixed in normal gaits.

4.4.2 Forward Kinematics

Matrix method

Using the method described in the theory part of this section, the $\boldsymbol{A}$ and $\boldsymbol{T}$ matrices, given the robots Denavit-Hartenberg parameters, could be calculated. To do the calculations some simple functions were created in Matlab that created $\boldsymbol{A}$-matrices for a given link (*Code snippet F.1*), one that created all $\boldsymbol{A}$-matrices for the configuration (*Code snippet F.2*), and a function that returned all $\boldsymbol{T}$-matrices from base frame to each joint (*Code snippet F.3*). All functions were coded so that all configurations would be calculated correctly, and any changes made to the configuration would not break the code. The resulting $\boldsymbol{A}$-matrices can be seen in *Figure 4.2* and the $\boldsymbol{T}$-matrices in *Figure 4.3*.

$$
\boldsymbol{A}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\boldsymbol{A}_2 = \begin{bmatrix} \sin(\theta_2) & 0 & \cos(\theta_2) & a_2\sin(\theta_2) \\ -\cos(\theta_2) & 0 & \sin(\theta_2) & -a_2\cos(\theta_2) \\ 0 & -1 & 0 & d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
\boldsymbol{A}_3 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & a_3\cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & a_3\sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\boldsymbol{A}_4 = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & a_4\cos(\theta_4) \\ \sin(\theta_4) & \cos(\theta_4) & 0 & a_4\sin(\theta_4) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
\boldsymbol{A}_5 = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & a_5\cos(\theta_5) \\ \sin(\theta_5) & \cos(\theta_5) & 0 & a_5\sin(\theta_5) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\boldsymbol{A}_6 = \begin{bmatrix} \cos(\theta_6) & 0 & -\sin(\theta_6) & 0 \\ \sin(\theta_6) & 0 & \cos(\theta_6) & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
\boldsymbol{A}_7 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_7 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Figure 4.2: $\boldsymbol{A}$-Matrices for robot configuration with symbolic calculations

Both the $\boldsymbol{A}$- and $\boldsymbol{T}$-matrices are shown symbolically with the parameters described in the section on Denavit–Hartenberg Parameters. When using these functions to find the position of each joint and end effector, real values for the current configuration were inserted. As far as possible, all functions were created to calculate everything, both symbolic and with real values.

$$\boldsymbol{T}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{T}_2 = \begin{bmatrix} s_2 & 0 & c_2 & a_2 s_2 \\ 0 & 1 & 0 & -d_2 \\ -c_2 & 0 & s_2 & -a_2 c_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_3 = \begin{bmatrix} c_2 c_3 & -s_2 s_3 & c_2 & s_2(a_2 + a_3 c_3) \\ s_3 & c_3 & 0 & a_3 s_3 - d_2 \\ -c_2 c_3 & c_2 s_3 & s_2 & -c_2(a_2 + a_3 c_3) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_4 = \begin{bmatrix} s_2 c_{34} & -s_2 s_{34} & c_2 & s_2(a_2 + a_3 c_3 + a_4 c_{34}) \\ s_{34} & c_{34} & 0 & a_3 s_3 + a_4 s_{34} - d_2 \\ -c_2 c_{34} & c_2 s_{34} & s_2 & -c_2(a_2 + a_3 c_3 + a_4 c_{34}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_5 = \begin{bmatrix} s_2 c_{345} & -s_2 s_{345} & c_2 & s_2(a2 + a3c_3 + a4c_{34} + a5c_{345}) \\ s_{345} & c_{345} & 0 & a3s_3 + a_4 s_{34} + a5s_{345} - d2 \\ -c_2 c_{345} & c_2 s_{345} & s_2 & -c_2(a2 + a3c_3 + a4c_{34} + a5c_{345}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_6 = \begin{bmatrix} s_2 c_{3456} & -c_2 & -s_2 s_{3456} & s_2(a_2 + a_3 c_3 + c_{34} + a_5 c_{345}) \\ s_{3456} & 0 & c_{3456} & a_3 s_3 + a_4 s_{34} + a_5 s_{345} - d_2 \\ -c_2 c_{3456} & -s_2 & c_2 s_{3456} & -c_2(a_2 + a_3 c_3 + a_4 c_{34} + a_5 c_{345}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_7 = \begin{bmatrix} s_2 c_{3456} & -c_2 & -s_2 s_{3456} & s_2(a_2 + a_3 c_3 + a_4 c_{34} + a_5 c_{345} - d_7 s_{3456}) \\ s_{3456} & 0 & c_{3456} & a_3 s_3 + a_4 s_{34} + a_5 s_{345} + d_7 c_{3456} - d_2 \\ -c_2 c_{3456} & -s_2 & c_2 s_{3456} & -c_2(a_2 + a_3 c3 + a_4 c_{34} + a_5 c345 - d_7 s_{3456}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.3: $\boldsymbol{T}$-Matrices for robot configuration with symbolic calculations

Trigonometric method

A trigonometric approach to calculate the forward kinematics of the robot was first used. Two projections were used to simplify the calculations. As seen in *Figure 3.3*, the first actuator works at a perpendicular direction compared to the rest and was therefore isolated in a $\boldsymbol{xz}$ projection as seen in *Figure 4.4*. From this, equations for x (*Equation 4.17*) and z (*Equation 4.18*) could be found using the length of the leg r in its current configuration and the angle of the first actuator: θ_2.

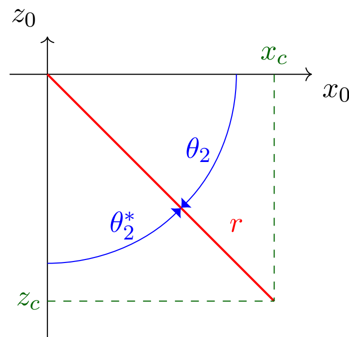


Figure 4.4: Forward Kinematics - xz-projection

$$x = r * sin(\theta_2) \tag{4.17}$$

$$z = -r * cos(\theta_2) \tag{4.18}$$

By removing the θ_2 calculation from the rest of the joints, the robot can now be projected in the yr-plane where r is the direction the leg is pointing from *Figure 4.4*. This projection leaves a configuration with three unknown angles in a two-dimensional plane as seen in *Figure 4.5*.

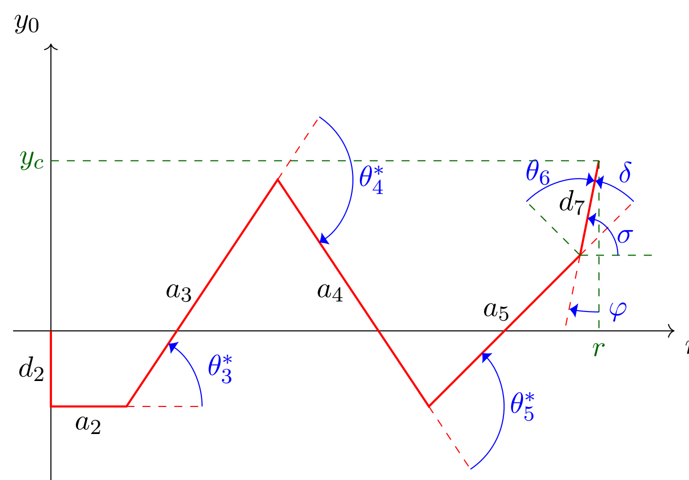


Figure 4.5: Forward Kinematics - yr-projection

Some of the angles in the yr-projection need to be explained as they are not that intuitive.

- φ: The angle between the ground and the robot end effector as defined in the gait analysis.
- δ: The angle of the end effector as defined in the physical model (used for the r length calculation).
- σ: The angle between the r-axis to the end effector (Used for finding θ_5^*).

Due to the rotation on link 6 in the Denavit-Hartenberg table, the angle θ_6 gets defined differently than θ_{3-5}. Since θ_6 is defined negative due to the rotational direction, the relationship between θ_6 and δ is like stated in Equation 4.19.

$$\delta = \frac{\pi}{2} + \theta_6 \tag{4.19}$$

An expression for the length r of the robot in the current position could then be determined as seen in Equation 4.20.

$$r = a_2 + a_3cos(\theta_3) + a_4cos(\theta_3 + \theta_4) + a_5cos(\theta_3 + \theta_4 + \theta_5) + d_7cos(\theta_3 + \theta_4 + \theta_5 + \delta) \tag{4.20}$$

The same method could then be applied to finding an expression for y as seen in Equation 4.21.

$$y_c = -d_2 + a_3sin(\theta_3) + a_4sin(\theta_3 + \theta_4) + a_5sin(\theta_3 + \theta_4 + \theta_5) + d_7sin(\theta_3 + \theta_4 + \theta_5 + \delta) \tag{4.21}$$

4.4.3 Inverse Kinematics

The inverse kinematics for this robot was calculated using a method described by Vijay Kumar (Kumar 2021). Using Figure 4.4 an expression for the actuator angle θ_2^* can be calculated as seen in Equation 4.22. Due to the rotation on link 2, this angle will have to be modified to $\theta_2 = \theta_2^* - \frac{\pi}{2}$ for plotting purposes.

$$\theta_2^\star = atan2(x_c, z_c) \tag{4.22}$$

In the $\boldsymbol{yr}$-plane, there are three unknown angles and only two expressions. The known angle φ from the gait analysis will have to be used to solve the inverse kinematics. This angle can be represented by the equation $\varphi = -(\theta_3+\theta_4+\theta_5+\theta_6)$. The inverse kinematics equation for θ_3 is calculated in Equation 4.23.

$$\begin{gathered}
\theta_3^\star = \gamma \pm cos^{-1}\left(\frac{-(r'^2 + y'^2 + a_3^2 - a_4^2)}{2a_3\sqrt{r'^2 + y'^2}}\right) \\
where \\
\gamma = atan2\left(\frac{-y'}{\sqrt{r'^2 + y'^2}}, \frac{-r'}{\sqrt{r'^2 + y'^2}}\right) \\
y' = y_c + d_2 + a_5sin(\varphi + \theta_6) - d_7sin(\frac{\pi}{2} - \varphi) \\
r' = r_c - a_2 - a_5cos(\varphi + \theta_6) - d_7cos(\frac{\pi}{2} - \varphi)
\end{gathered} \tag{4.23}$$

Combining the φ expression with the ones for r (*Equation 4.20*) an y (*Equation 4.21*) and reordering them for sinus or co-sinus expressions containing θ_4 gives the opportunity to use the inverse tangent function to find θ_4 as in *Equation 4.24*.

$$\theta_4^\star = atan2\left(\frac{y' - a_3 sin(\theta_3)}{a_4}, \frac{r' - a_3 cos(\theta_3)}{a_4}\right) - \theta_3 \tag{4.24}$$

Now that expressions for θ_3 and θ_4 have been found, a reordering of the equation for φ gives the solution for θ_5 as given by *Equation 4.25*.

$$\theta_5^\star = -(\varphi + \theta_3 + \theta_4 + \theta_6) \tag{4.25}$$

In Matlab, both the inverse kinematic results for a given Cartesian position of the end effector and a set of waypoints were needed. *Code snippet F.5* implements the equations listed for θ_{2-5} and returns a momentary DH-table for the current position. By using momentary DH tables, the same plotting function can be used to visualize the robot in starting position and all momentary positions. *Code snippet F.6* takes all Cartesian waypoints and uses the inverse calculation function to convert the Cartesian waypoints to angular ones.

The complete calculations for forward and inverse kinematics using trigonometry can be seen in *Appendix G*.

4.4.4 Velocity Kinematics

A function (*Code snippet F.4*) was created to calculate the Jacobi matrix. As stated, the Jacobi matrix can be used to find the inverse kinematics for the robot by finding its pseudo inverse. This was done by taking advantage of the built-in function *pinv()* in Matlab. To implement this, one would have to find expressions for the linear and angular velocities for the end effector in the base frame. By only looking at movement along the y-axis and setting a constant velocity, this could be achieved by using the projection in *Figure 4.6*. In this figure, the yz-axis is the base frame of the robot, and the value b is the height from the ground to the first joint. y_0 and y_1 is the start and stop point for the end effector, and the red line between them is the desired linear path.

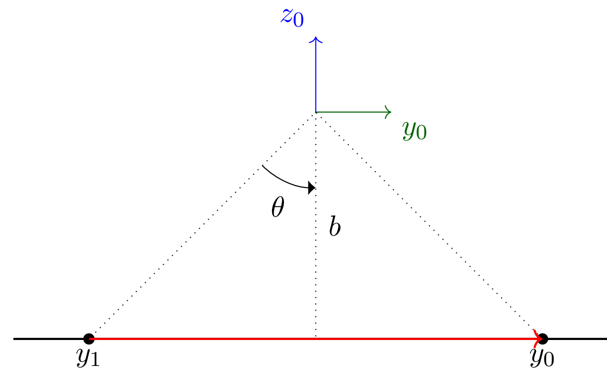


Figure 4.6: Linear and angular movement

Using trigonometry and finding the derivative gives expressions for linear velocity in the y_0 direction and angular velocity around the x_0 axis (*Equation 4.26*).

$$\begin{aligned} \dot{y}(t) &= \frac{y_1 - y_0}{t_{lim}} \\ \dot{\theta}_x &= \frac{y_1 - y_0}{bt_{lim}\left(\frac{1}{b^2}\left(\frac{y_1-y_0}{t_{lim}}t + y_0\right) + 1\right)} \end{aligned} \tag{4.26}$$

Taking these expressions and inserting them into the column vector for body velocity $\boldsymbol{\xi}$ while setting the other values as zero allows calculating the angular velocities for all joints. This work was not implemented for the simulations in Matlab but is included in the code presented in the git repository (Arnesen, Grinde, Hovland and Vestland 2021b).

4.4.5 Path planning

Waypoints

The Cartesian waypoints from the gait analysis are used as a path for the trajectory planning of this robot. Using a function (*Code snippet F.6*) that implements the inverse kinematics, the matrix of Cartesian points are converted to angles for further calculations. A visual control plot was generated for the robot pose in each waypoint to uncover possible problems (*Figure 4.7*).

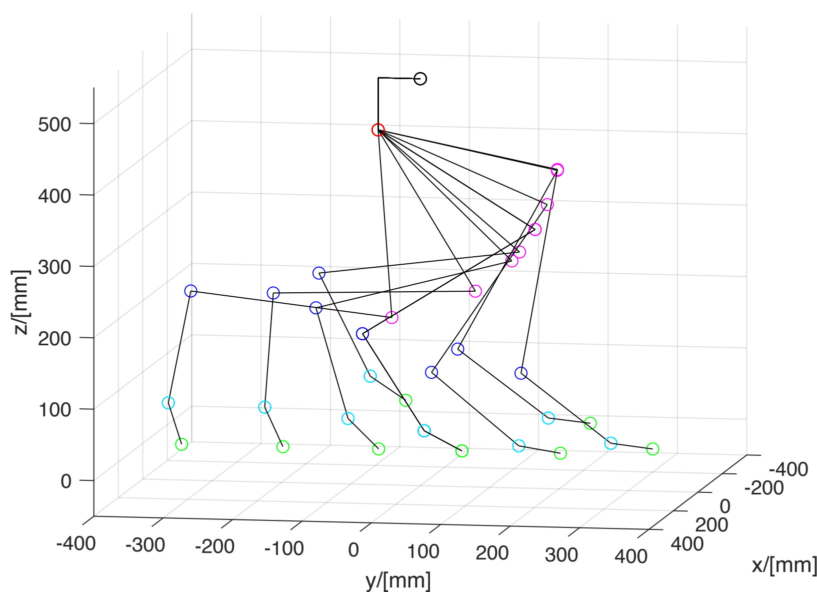


Figure 4.7: Robot plotted for each waypoint generated

Trajectory

The trajectory is solved in Matlab using quintic polynomials described in the theory section. One value was given for the velocity limitation in each waypoint to simplify the calculation. Since the angles might be going in both positive and negative rotational direction and even change direction in a point, a function (*Code snippet F.12*) was created to fix the sign of the velocity. This function would check the direction of movement and give the velocity a negative sign if the movement was negative on both sides of the waypoint. If the movement changed direction, the velocity was set to 0 rad/s to avoid overshoot.

A function that takes in the start and end parameters for angle, time, velocity, and acceleration and creates a symbolic expression with respect to time was made, (*Code snippet F.7*). The function sets the values for velocity or acceleration to zero for the start and end parameters if the function does not receive its values. The function returns expressions for angle, velocity, and acceleration. Another function (*Code snippet F.8*) was made that takes in all waypoint angles created by the inverse kinematics waypoint function (*Code snippet F.6*) and a list of time limits that each trajectory must adhere to. It has two optional input variables for the start and end values for velocity and acceleration. This function uses the trajectory function to return a matrix of symbolic expression for all trajectories, for all joints on the path.

Finally, the symbolic expressions are fed into a function (*Code snippet F.9*) with the time limits and a variable to set the number of time steps between each via points for plotting. This function returns a matrix where each row contains a list of joint angles for that specific joint. It also returns a timeline that contains the corresponding timestamp for each angle. Plotting the angle (*Figure 4.8*), velocity (*Figure 4.9*) and

acceleration (*Figure 4.10*) for each actuator enables spotting impossible angles or unwanted responses for velocity and acceleration. In this simulated gait, all waypoint velocities were set to $0\frac{rad}{s}$

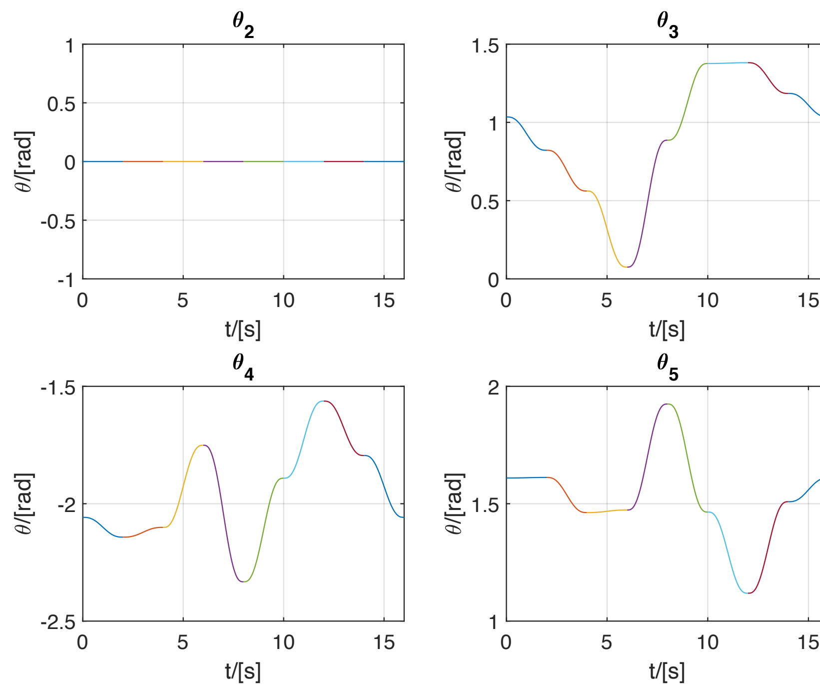


Figure 4.8: Actuator angles for complete gait cycle

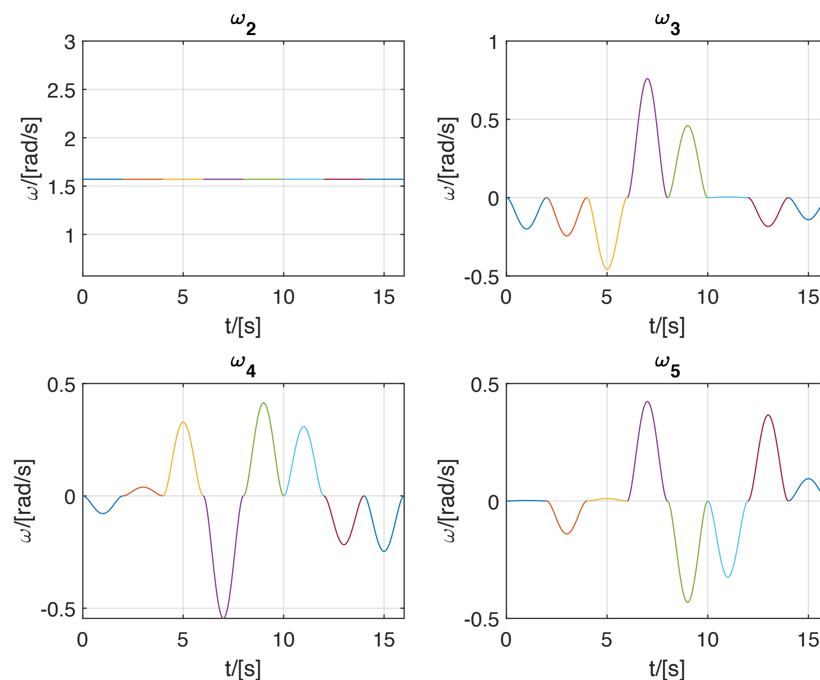


Figure 4.9: Actuator velocities for complete gait cycle

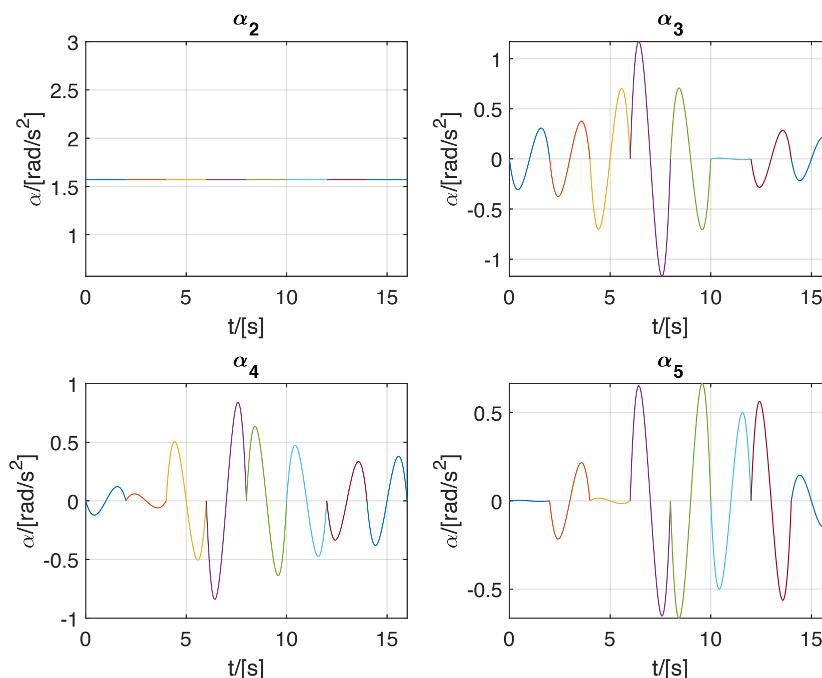


Figure 4.10: Actuator accelerations for complete gait cycle

4.4.6 Plotting Functions

Several functions were created to visualize the results from the calculations. *Code snippet F.10* takes in a momentary DH-table and runs it through the T and A matrix functions. From the T matrices, it gets the Cartesian position of all links in the DH-table. Link two has to be plotted separately to accommodate the hip design of the robot. The configuration is plotted in a simple "stick-figure" presentation of the robot leg. To analyze the resulting gait *Code snippet F.11* was created. This function takes in the original DH-table and base height in addition to the discrete angles and timeline created by *Code snippet F.9*. The animation function then runs through the timeline, creates a momentary DH-table for each time increment, and uses the *plotRobot* function to draw the momentary configuration. Then the plotted image is stored in a vector, and the plot is cleared when starting a new cycle. When the process is completed, the vector is saved as a MPEG-4 video file.

4.4.7 System dynamics

As discussed in *subsection 4.5.5*, modeling the dynamics of this system is rather complex. Looking at a simplified model of the system allows for the use of well-known and more trivial modeling principles. For demonstration purposes, the leg was modeled as a robot manipulator attached to a base frame, represented by the hip's connection to the supporting system as presented in chapter 3. Modeling dynamics for robotic manipulators are well documented. The method used in this chapter is based upon the principals from Spong, Hutchinson and Vidyasagar 2006, Freidovich 2017 and

Egeland and Gravdahl 2002.

The simplified system contains four rotary joints connected to rigid body links. A vector of generalized coordinates $\boldsymbol{q}$ represents the angle of joint i, denoted as $\boldsymbol{q}_i$. The four rotary joints are $\boldsymbol{q}_2^*...\boldsymbol{q}_5^*$, while the other joints in *Equation 4.27* are fixed.

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2^* \\ q_3^* \\ q_4^* \\ q_5^* \\ q_6 \\ q_7 \end{bmatrix}. \tag{4.27}$$

The kinetic energy of a link will be given by its angular velocity $\boldsymbol{\omega}_i$ and the linear velocity $\boldsymbol{v}_i$, both given by the Jacobian matrices previously found. $\boldsymbol{I}_i$ is the moment of inertia. By substituting $\boldsymbol{\omega}$ and $\boldsymbol{v}$ with their linear combination of the time derivative given by the Jacobian matrices, we find the kinetic energy. Velocities and moments of inertia are given around the CoM. The Lagrangian of the system is then found by evaluating *Equation 4.15*. Kinetic and potential energy are found using physical properties such as the moments of inertia and the kinematics of the robot. The moments of inertia are provided in the Matlab section in the git repository.

The kinetic energy of a link will be given by its angular velocity $\boldsymbol{\omega}_i$ and the linear velocity $\boldsymbol{v}_i$, both given by the Jacobian matrices previously found. $\boldsymbol{I}_i$ is the moment of inertia. By substituting $\boldsymbol{\omega}$ and $\boldsymbol{v}$ with their linear combination of the time derivative given by the Jacobian matrices, we find the kinetic energy. Velocities and moments of inertia are given around the CoM.

$$\boldsymbol{T} = \frac{1}{2} m_i \boldsymbol{v}_i^2 + \frac{1}{2} m_i \boldsymbol{I}_i \boldsymbol{\omega}_i^2 \tag{4.28}$$

$$\boldsymbol{v}_i = \boldsymbol{J}_{\boldsymbol{v}_i}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{4.29}$$

$$\boldsymbol{\omega}_i = \boldsymbol{J}_{\boldsymbol{\omega}_i}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{4.30}$$

The generalized coordinates and their derivative then give the total kinetic energy of the leg.

$$\boldsymbol{T} = \frac{1}{2}\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{4.31}$$

$$\boldsymbol{M}(\boldsymbol{q}) = \sum_{i=1}^{n} [m_i \boldsymbol{J}_{\boldsymbol{v}_i}^T(\boldsymbol{q}) \boldsymbol{J}_{\boldsymbol{v}_i}(\boldsymbol{q}) + \boldsymbol{J}_{\boldsymbol{\omega}_i}^T(\boldsymbol{q}) \boldsymbol{I}_i \boldsymbol{J}_{\boldsymbol{\omega}_i}(\boldsymbol{q})] \tag{4.32}$$

$\boldsymbol{M}$ is the $n \times n$ symmetric mass matrix and is positive definite, (Egeland and Gravdahl 2002).

The potential energy of a rigid link will only be given by gravity. As the Lagrangian states, the potential energy is only given by the generalized coordinates $\boldsymbol{q}$ and not their derivative $\dot{\boldsymbol{q}}$.

$$\boldsymbol{V}(\boldsymbol{q}) = \sum_{i=1}^{n} [m_i \boldsymbol{g}_i \boldsymbol{p}_i(\boldsymbol{q})] \tag{4.33}$$

Where:

- m is the mass for the link
- $\boldsymbol{g}$ is the gravity vector
- $\boldsymbol{p}$ is the CoM coordinates given by the direct kinematics

The Lagrangian of the robot leg is then found by *Equation 4.34*, and the equations of motion can then be calculated from *Equation 4.16*.

$$\boldsymbol{\mathcal{L}} = \frac{1}{2} \dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q}) \dot{\boldsymbol{q}} - \boldsymbol{V}(\boldsymbol{q}) \tag{4.34}$$

It can be shown that for any robotic manipulator and in general for a mechanical system with kinetic energy in the same form as *Equation 4.31*. That the generalized forces acting on the joints and thus the equations of motion can be written as *Equation 4.35*, (Spong, Hutchinson and Vidyasagar 2006).

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q}) \ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) \tag{4.35}$$

Where:

- $\boldsymbol{g}$ is the gravity vector
- $\boldsymbol{C}$ is the matrix of centrifugal and Coriolis effects
- $\boldsymbol{M}$ is the mass matrix given by *Equation 4.32*

As discussed in *subsection 4.5.5* fully calculating the equations of motions is a bit out of the scope of this thesis. More information on how to calculate the $\boldsymbol{C}$ matrix is available in textbooks, such as Spong, Hutchinson and Vidyasagar 2006 and Freidovich 2017. A Matlab template for calculating the equation of motion and the contents of the moments of inertia matrices is presented in the Matlab section of the Git hub repository.

4.5 Analysis and Discussion

4.5.1 Denavit–Hartenberg Parameters

As stated, the chosen configuration places the robot floating in the air with all Cartesian z-coordinates being negative. A d_1 translation corresponding to the distance from the ground to the stands base link could be added to ground it. This would enable setting values in relation to the ground when making paths. In this thesis, it was decided that using the floating base frame was the most intuitive. If this were to change with future use, a simple edit to the Denavit–Hartenberg, inverse kinematics and plotting functions in Matlab would be needed. For the forward kinematics, *Eq 4.18* would have to be updated to accommodate this.

4.5.2 Velocity Kinematics

In this project, the Jacobi matrix was calculated but only used for an alternative inverse kinematics route that was not implemented. However, it was essential to include as most future work regarding model-based controllers and simulations depends on it. Also, the alternative inverse kinematics solution could help make a simple linear Cartesian movement without having to replace large portions of the code.

Path and trajectory

As mentioned, the desired path is found by mimicking the domestic cat. The parameters for timing, velocity, and acceleration are, on the other hand, arbitrarily chosen to create a smooth gait in simulations. If the future goal for the robot is a more biomimetic gait, further gait analysis and other algorithms for calculating the trajectories are needed.

Another aspect of biomimicry is that the leg is not connected to an actual body. This means that the hip joint does not move with the motion of the complete animal. Some of this could be improved by implementing the spring dampening system discussed in the design chapter (*section 3.4*). However, this still does not completely make up for the moving body.

Linear trajectories

As can be seen in *Figure 4.11* the quintic trajectories for the angles do not take the ground into account. As the ground in this configuration is set at -520 mm, the end effector will attempt to either dig into the ground or lift the stand.

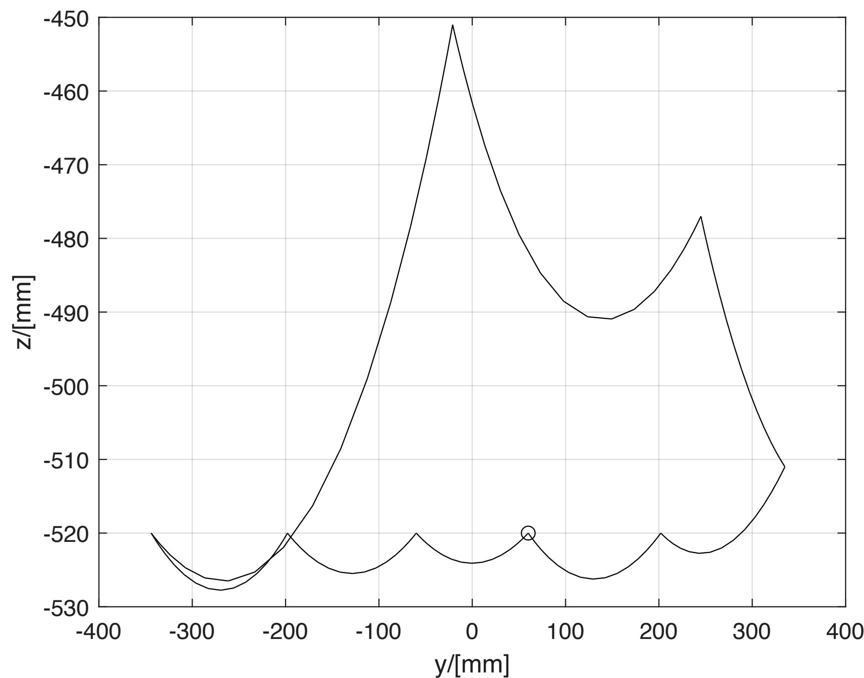


Figure 4.11: End effector coordinates for complete gait

As discussed in the design chapter, if the spring dampening system were in place, this would probably remove some if not all of this effect. This trajectory would either mean that the end effector has different friction to the ground at different parts of the trajectory. If the linear Cartesian trajectories described in the section on velocity kinematics (*subsection 4.4.4*) were implemented, the curved movement could be eliminated, and constant friction could be achieved.

Impossible configurations

Since the trajectory planning uses quintic polynomials with only start and stops parameters for angle, velocity, acceleration, and time limits. This means that no minimum and maximum values for the angles or Cartesian coordinates are given. The resulting trajectories could include collisions with either the ground or the robot itself. As the robot is currently controlled with ROS, this is not a problem as ROS has built-in collision avoidance and uses other kinematics solvers. If future work moves away from using ROS as a trajectory planner and implements more advanced control in Matlab, this would have to be addressed.

4.5.3 Trajectory Optimization

As discussed, the leg configuration and gaits planned in this thesis are based on animal's biomimetic properties, specifically the cat. However, gaits planned by solving an optimization problem would provide a more optimal gate for a given robot configuration and set constraints. Furthermore, a discussion can be made that a cat's

gait is the same as the optimal solution due to evolutionary properties. For robot locomotion, finding a gait that minimizes motor torque or follows other constraints is usually preferable. Solving optimization problems for trajectory planning is out of the scope of this thesis but will be briefly discussed for further development in the following section.

Planning a gate based on an optimization problem, formulated as a nonlinear program (NLP), requires knowledge about the dynamic and kinematic model, which are not found in this thesis. Solving a nonlinear optimization problem can be done using an NLP solver and constraints to describe the desired outcome. Simplifying the problem is usually done to reduce commuting time as the field of possible optimal solutions shrinks. For example, these simplifications can limit the rotation and height of the base link so that jumping or rotating solutions are not included. Initial guesses can also be made for the same purpose. The hybrid system presented in this thesis would require a multi-phase trajectory due to contact points. Solving hybrid systems such as this is usually done by collocation methods such as implicit Runge-Kutta. More information on this topic can be found in Nocedal and Wright 2006 and how it may be used for optimal control is presented in Aksel N. Heirung 2016.

4.5.4 Code optimization

Functions were created to do most tasks so that no unnecessary lines were repeated. This also simplifies the code. Since the algorithm involves running through many large for-loops involving large matrices and symbolic calculations, the code's run time can get quite extensive. Even after optimizing with preallocating all variables, the code takes several minutes (including plotting) from desired waypoints to symbolic expressions for trajectories using eight waypoints for the complete gait. Since only the angles for the four actuators are changing during the gait, the functions could be optimized for this specific robot by only feeding these angles into the various functions. This would reduce the momentary values from a 7×5 matrix to a 4×1 column vector.

In addition to the big momentary DH-table, the base height was, as described, not included in the original DH-table. This meant that the base height would be fed into every function that visualizes the robot. By adding a line to adjust the logical desired z value from the base frame to the world frame, the DH-table could be completed with the base height included. This would involve updating all the plot functions but would result in a more streamlined code.

4.5.5 Hybrid systems

Modeling the system dynamics is needed for simulation and allows for the implementation of model-based controllers, discussed in chapter five. Accurately modeling the complete system introduces complexities that are not very trivial to solve and out of the scope of this thesis. In *subsection 4.4.7* the system was simplified as a robot manipulator. This simplification was made to present an overview of how to approach

modeling the dynamics of a robot leg. If model-based controllers are to be implemented in further work on this project. The principles described can be used as a baseline. Literature on this topic is also presented. A brief overview of the different properties needed to be included in the model will be discussed.

The robot leg, with its support system, is a hybrid system. A hybrid system has to be modeled as a discrete model with a continuous mode and a discrete mode. A simple example of a hybrid system is a bouncing ball, where the ball has a continuous mode between each bounce; however, when the ball touches the ground, the model undergoes a discrete change. In this case, the model will change when the leg touches the ground. This can be described as a "contact problem" at $z = 0$. Conditions like this can be calculated from the kinematics and implemented in software. Modeling and simulating friction can also pose a problem. The model used in this thesis will have friction between the ground and both the wheels and leg. Successfully detecting a change in the model, including jumps from static to dynamic friction in the simulation scheme, is necessary. If the step-time is not small enough around these discrete changes, the simulated system will not behave properly. The controller for a hybrid system would also need to be hybrid, and this could introduce problems when transitioning between modes. Effects of linear and angular motion on the supportive system were also not addressed in *subsection 4.4.7*. A more detailed explanation of mathematical modeling and control of a hybrid robot leg system is presented in Sobotka 2007.

4.5.6 Experimental modeling

When the system dynamics are unknown experimental modeling can be used to estimate transfer functions. The unknown transfer functions can be modeled as black boxes with output and input. Performing impulse or step responses on the input of this black box will give a measurable response to the output signal. There are different methods to estimate transfer functions for these responses. The impulse and step method are described in Hveem and Bjørvik 2014. Measuring how the system responds to changes in input frequency could also be used to plot a bode diagram, which can be used to estimate a transfer function.

Experimental modeling is not done in this thesis due to the efficiency of the PID controller and due to properties of the dynamical system. These properties include a hybrid model with an inconsistent gravity force, Coriolis effects, and centrifugal forces on the joints that would cause inaccurate models when not operating within a small constraint area. Trying to model all the cross-coupling in this MIMO system would become quite cumbersome and not very efficient. This is further discussed in *subsection 5.3.1*.

4.5.7 Physical simulation in Matlab

Matlab includes some powerful packages for working with robotics. This includes a toolbox that would enable importing the URDF generated in Fusion 360, adding

physical properties, and simulating the drive train (MathWorks 2021b). Using this package, one could synchronize the Simulink model with *Gazebo* in ROS so that the controller in Matlab would directly work with the physical model. This would just show the physical model mimicking the simulated one and presents no feedback from the real system. The imported model for the complete quadruped can be seen in *Figure 4.12*. Another option for this would be to set up Matlab as a ROS node so that measured values could be fed back into Matlab. Then the controllers in Matlab would send the control signal back to the motor drivers through ROS. Either way, doing a completely simulated model in Matlab and Simulink would have been a great way to figure out problems before building a physical model and would have helped with choosing the actuators needed.

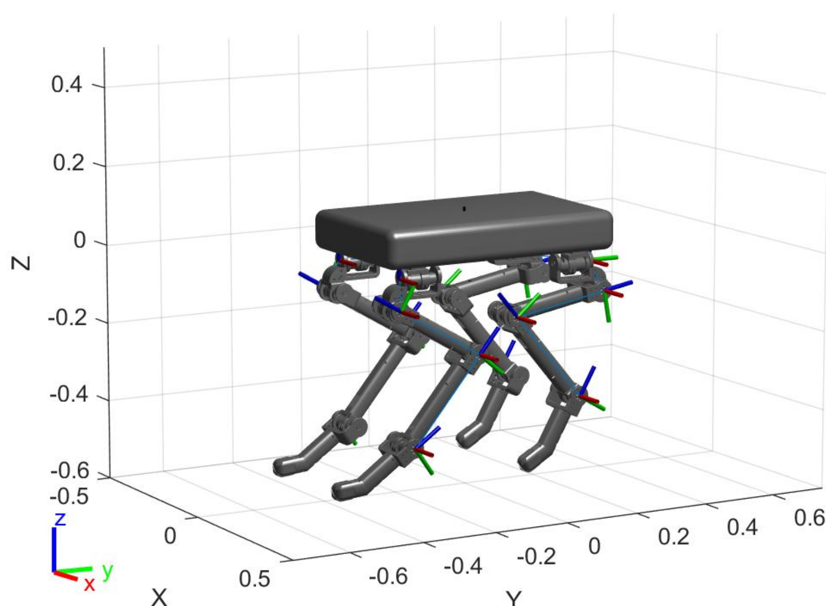


Figure 4.12: Imported URDF in Matlab

4.6 Chapter Conclusion

The mathematical model created includes the forward and inverse kinematic, path and trajectory planning, and proposed velocity kinematics for the robot configuration. It also contains a description for further modeling.

The complete calculation successfully plans and plots the trajectories between set waypoints and allows for time, speed, and acceleration limitations. This includes inverse kinematics from Cartesian waypoints to angles, trajectory planning between actuator angles, and forward kinematics for plotting and simulation.

Alternative inverse kinematics using body velocity are explored but not implemented.

This would allow for linear Cartesian movement of the end effector. Modeling the system dynamics is also explored but not fully completed.

By importing the URDF file generated in Fusion 360, a model of a complete four-legged robot was implemented. This enabled visualizing the robot using the same calculations done before. The toolbox used for this also includes tools for modeling the physical aspects of the actuators and how gravity would affect the robot.

In the end, as presented in the control chapter, the mathematical model was not used in the controller implementation. However, the completed work lays the foundation for more optimal control and simulation. Further work could elect to use Matlab as a node for ROS to enable the position and speed controllers to run directly in the program.

Chapter 5

Control

5.1 Introduction

For a robot leg to track a planned trajectory, it is necessary to control each joint. The inverse kinematics previously found provides the different angles necessary to reach a specific point. A gait, or a trajectory, will consist of different points to move through. This is discussed previously. Controlling the joints reliably to follow the trajectory for each state will make the robot execute the planned gait. This chapter will assume that the reader has a general understanding of control theory and the PID controller. However, some core concepts will be briefly explained to make it easier to follow the more advanced topics and reasoning behind the different choices. The theoretical framework is readily explained in textbooks that cover general control theory, e.g., Hveem and Bjørvik 2014. Based on what is presented in this chapter, the results of the fully implemented control system will be provided in the main results, as it is dependent on multiple parts of the project.

5.2 Theoretical Framework

5.2.1 Feedback Loop

In control theory, a negative feedback loop provides the difference between a target and the measured value. The measured value can be a signal from a sensor, or it can be estimated based on the mathematical model. The negative feedback loop is the core principle behind many different control strategies, such as the PID controller. A general example of how the feedback loop is implemented in control systems is shown in figure 5.1.

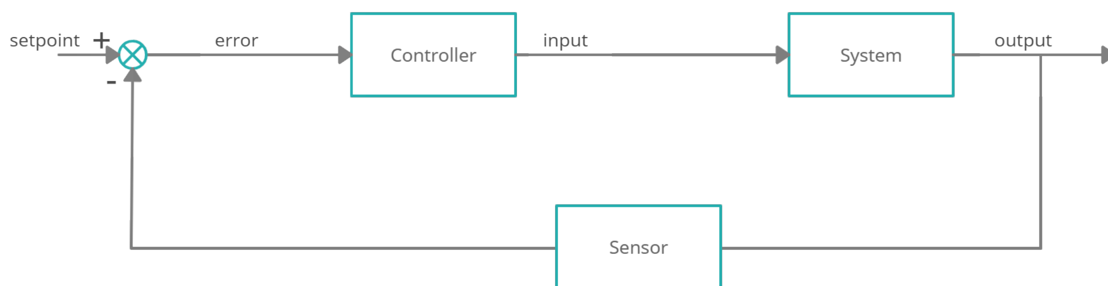


Figure 5.1: Negative feedback loop

5.2.2 PID

The PID controller works by implementing a feedback loop and consists of three different components, which can be described by breaking down the summation form of the PID. The three components are; proportional, integral, and derivative.

$$u_{pid} = u_p + u_i + u_d \tag{5.1}$$

In the time domain, the PID continuously calculates an error given by the feedback loop. This error $e(t)$, is then used to calculate the P, I, and D terms, which adds up to the actuating signal u.

The proportional term multiplies a constant K_p with the error.

$$u_p = K_p e(t) \tag{5.2}$$

The Integral term multiplies the integral of the error with a constant T_i.

$$u_i = \frac{K_p}{T_i} \int_0^t e(\tau)\, d\tau \tag{5.3}$$

The derivative term only considers the error rate and multiplies it with a constant T_d.

$$u_d = T_d K_p \frac{de(t)}{dt} \tag{5.4}$$

The proportional constant is used to tune how quick the response will be. The integration constant is often tuned to get rid of the steady-state error. The derivative constant is usually tuned to reduce overshoot by limiting rapid changes when the error rate is significant. A simple illustration of how these terms change the response of a unit step on the setpoint is shown in *Figure 5.2*. Note that the parameters are not tuned to achieve any specific response.

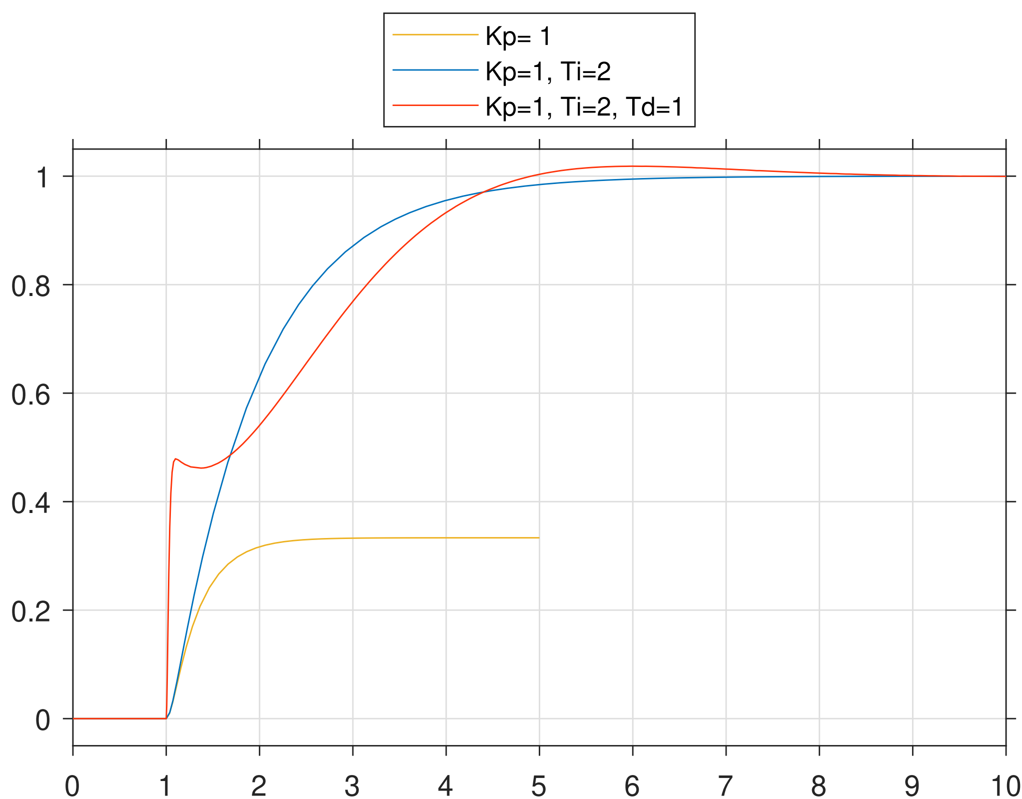


Figure 5.2: PID illustration

5.2.3 Pole Placement

A pole placement controller is a form of full feedback control that uses the system's states instead of its output(s) to control the system dynamics. The controller is implemented by adding a feedback loop and a gain matrix. Implementing a pole placement controller is only possible if the system is controllable, meaning the control signal can reach all possible states. *Equation 5.5* shows how the gain matrix $\boldsymbol{k}$ is found, using the system matrix $\boldsymbol{A}$, the signal matrix $\boldsymbol{B}$, and the desired closed-loop poles $\boldsymbol{P}$. A

system with a state feedback controller is shown in figure *Figure 5.3*

$$P = |\lambda I - (A - Bk)| \tag{5.5}$$

$$u = -kx$$

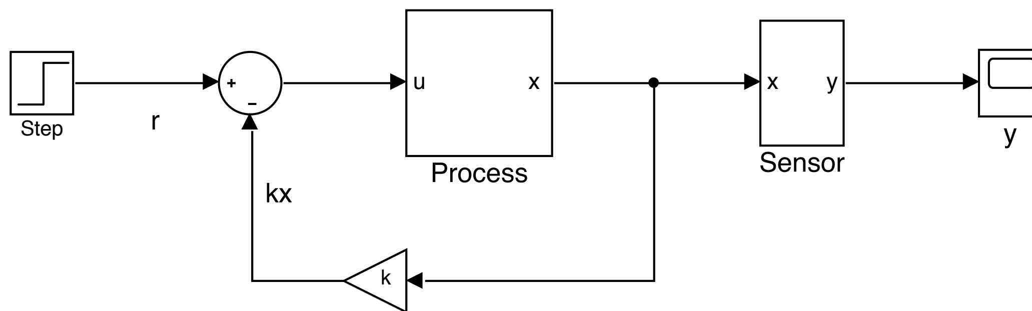


Figure 5.3: State feedback controller

The pole assignment is what decides the desired performance. There are different methods to determine the poles. For example, a practical solution for a faster closed-loop response is to set the desired poles $2 - 5$ times the system poles. (Anstensrud 2020b)

In instances where some of the systems states can not be measured, the implementation of an estimator is required. The estimator runs parallel to the process using input and output values to estimate a state $\hat{x}$ in real-time. A system with a state estimator is shown in *Figure 5.4*, (Anstensrud 2020c).

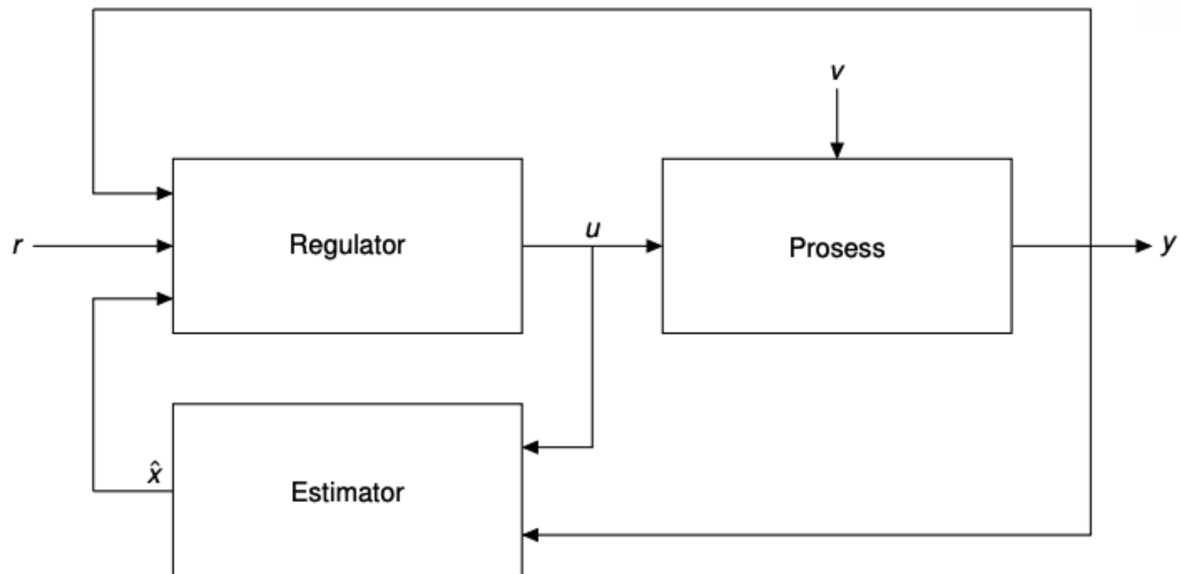


Figure 5.4: State Estimator, (Anstensrud 2020c)

5.2.4 Linear-Quadratic Regulator

The theory presented in this chapter is found in Douglas 2019 and more comprehensively in kwakernaak 1972, chapter 3.

Although the method of deciding poles for a state feedback controller is easy to use, it is not a method usually implemented for optimal control. An LQR is a more suitable method to find an optimized gain matrix, using a cost function instead of placing poles. The cost function shown in *Equation 5.6* accounts for performance and actuator effort and decides an optimal gain based on the weighted priorities. $\boldsymbol{Q}$ and $\boldsymbol{R}$ represent the weighted performance and actuator effort respectively of the controller.

$$\boldsymbol{J}(\boldsymbol{u}) = \int_0^{\infty} [\boldsymbol{x}^T(t)\boldsymbol{Q}\boldsymbol{x}(t) + \boldsymbol{u}^T(t)\boldsymbol{R}\boldsymbol{u}(t)]dt \tag{5.6}$$

$\boldsymbol{x}(t)$ refers to the trajectory for the system states. For robot movement, it is generally generated by solving an optimization problem. Since $\boldsymbol{Q}$ is a scaling matrix, the weighted priorities to each state can be decided. A higher value in a state's respective element produces a faster convergence and better trajectory tracking.

The $\boldsymbol{R}$ matrix is also a scaling matrix, which size depends on the number of inputs $\boldsymbol{u}$. This makes it possible to decide the level of effort in each actuator. Higher values in the $\boldsymbol{R}$ matrix entail less actuator effort.

Finding the gain matrix in an LQR is a simple process in Matlab using the function $>> \boldsymbol{k} = lqr(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{Q}, \boldsymbol{R})$

5.2.5 Digital Control Systems

Complex systems often consist of analog components and digital computers. A digital computer allows for more complex and re-programmable behavior. Communication between analog and digital components works by using DACs and ADCs. These are converters that convert between digital or analog signals. When sampling a continuous signal, the signal is replicated as a sequence of numbers in the discrete domain $\mathcal{Z}$. Where $Z = e^{sT}$.

Converters

When supplying power to electrical machines, the most common method used to convert a discrete signal is PWM. PWM is done by rapidly changing between the max output and zero. How rapid the signal is changing is given by the period time T, and the proportions between on and off are given by the duty cycle D. Using a high PWM frequency will make the electrical machine operate as if it were a smooth signal due to the machines' inductive traits.

Most digital computers used in control systems have an integrated DAC. In the Arduino boards, the DAC is realized by the successive approximation method. The successive approximation method consists of a binary search. A comparator compares the bytes of the signal from the DAC converted by an ADC with the original digital signal. This method is a quick converter as long as the input signal is constant during the converting cycles.

Sampling and time delay

Sampling time is an important topic when it comes to digital systems. The sample time describes how rapidly the digital system is sampling the analog signal. Sampling can be done by a sample and hold circuit, which introduces additional dynamics in the system. An example of this is the zero-order hold which holds the value over one sample interval, with dynamics represented by F_{ZOH}.

$$F_{ZOH} = \frac{1 - e^{-Ts}}{s} \tag{5.7}$$

Having a sufficient sampling time is necessary to avoid problems with aliasing. A sufficient sampling frequency is given by the Nyquist-Shannons sampling theorem. Stability in the Z domain is also largely dependent on a sufficient sampling frequency due to the traits of the Z transform. In practice, the sample time is determined by the computing cost and the speed of the CPU. A comprehensive treatment of digital signal processing is a bit beyond the scope of this thesis. For more information on this topic, please refer to textbooks about digital signal processing such as (McClellan, Schafe and Yoder 2018).

Digital control system introduces additional time delay into the system, which can

cause stability problems. This time delay is entirely dependent on the sampling time h. A simplified but practical method to find an approximate value for the additional time delay based on the padé-approximation is described in Hveem and Bjørvik 2014.

$$\tau_{controller} = 1.5h \tag{5.8}$$

The constant 1.5 accounts for the calculation time as well as the speed of the ADC and DAC converters.

5.2.6 MIMO Systems

Multiple-input and multiple-output systems have more than one input variable or more than one output variable. MIMO systems often complicate the control due to cross-couplings. Cross-couplings describe how the different process variables interact with each other. When one setpoint is changed, every other process variable with cross-coupling to that variable will change. As a result, each controller will then observe a more complex system, which can make robust control difficult. A decoupling filter may be introduced to help fix this. However, implementing a decoupling filter requires knowledge of how the variables interact with each other. Model-based methods like RGA and singular-value analysis can be used to describe the couplings. These methods can help to decide on which inputs or outputs to remove for more robust control.

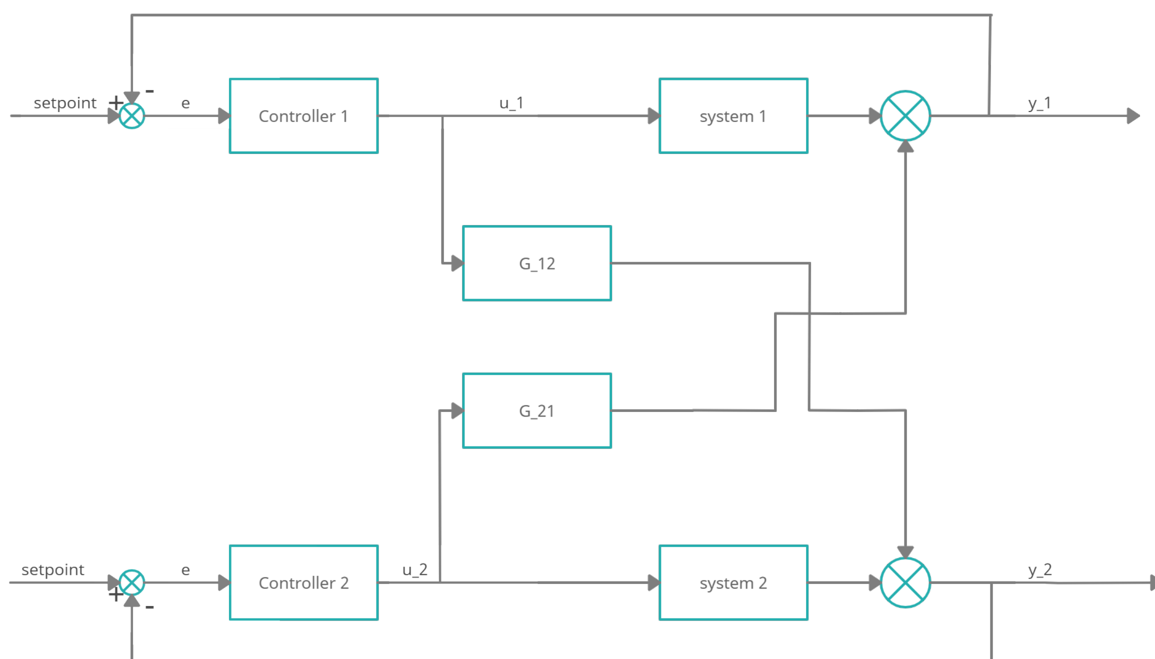


Figure 5.5: MIMO system

5.2.7 Closed Loop Response

When tuning a controller, a specific response for the closed-loop system is usually desired. In control theory, we often chose between a fast response with overshoot, called a second-order response, or a slower response without overshoot, called a first-order response. A stable second-order response is described by a complex conjugate pair in the left-hand plane and a first-order by reel poles.

5.3 Discussion and Results

5.3.1 Choice of Controller

When choosing the controller for the robot leg, the PID and LQR controller were evaluated. The PID controller was selected due to the difficulty of modeling the system dynamics for the hybrid system discussed in *subsection 4.5.5*. Using a PID controller allows for easy implementation in digital control systems and does not require an accurate dynamic model of the system. The joints are controlled by independent joint PID control. Position control of a brushed DC motor is an integrator by nature, and the controller should not need an additional integrator to avoid steady-state error from a unit step at the setpoint, (Hveem and Bjørvik 2014). However, an integrator was necessary for avoiding steady-state error due to significant disturbance.

Independent joint control trivializes the multivariable system. Trying to model it as a nonlinear MIMO system would make it unnecessarily complicated. For example, the cross-couplings in a robot leg, given by the equations of motion, are rather complex. The equations of motion and the nonlinearities are described in *subsection 4.4.7*. When controlled by independent joint control, the cross-couplings are represented by a disturbance. The controllers are then tuned separately to suppress this disturbance. The integration term of the controller counteracts the disturbance, which mainly consists of the gradient of the gravity vector. *Figure 5.6* shows a block diagram for independent joint control of a given motor.

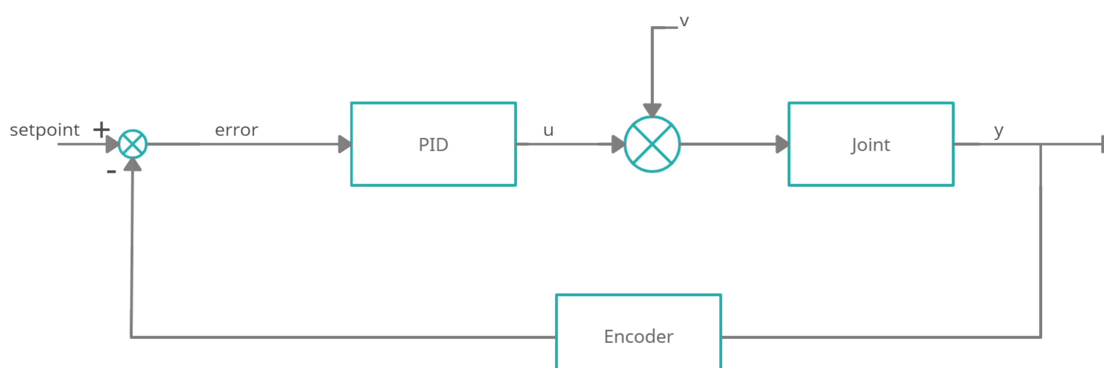


Figure 5.6: Block diagram for independent joint PID

5.3.2 Discrete PID

In this thesis, a digital PID controller was used, programmed in the Arduino Language. A digital PID was achieved by discretizing the equations given in the time domain. This was done by substituting $t \rightarrow nT$. Where:

- T = sampling time
- n = n-th time step.

Applying this substitution directly to equation 5.2 gives a discrete equation representing the proportional term.

$$u_p(nT) = K_p e(nT) \tag{5.9}$$

Representing an integral in discrete time can be done by a summation. By using backward difference, the integral in equation 5.3 was written as a summation and implemented in the digital controller.

$$u_i(nT) = \frac{K_p T}{T_i} \sum_{k=0}^{n-1} e(kT) \tag{5.10}$$

The derivative term can be expressed discreetly as the change between two time-steps over the sample time. Applying this to equation 5.4 gives a discrete derivative term.

$$u_d = T_d K_p \frac{e(nT) - e((n-1)T)}{T} \tag{5.11}$$

When implementing a digital PID controller, it is common to add extra features to make it more robust. One of these is anti-windup on the integrator. Anti-windup stops the integrator when the control signal has reached its maximum or minimum value. Successful implementation of anti-windup prevents windup in the integrator, which would worsen the control. How windup on the integrator worsens the control by introducing additional delay for a response is illustrated in *Figure 5.7*. Implementing a filter for the derivative term of the PID is also commonly done. However, in this thesis, incremental encoders were used. Sensor data from incremental encoders are not noisy, and thus a derivative filter is not necessary. The working principles of incremental encoders are discussed in *subsubsection 6.3.7*.

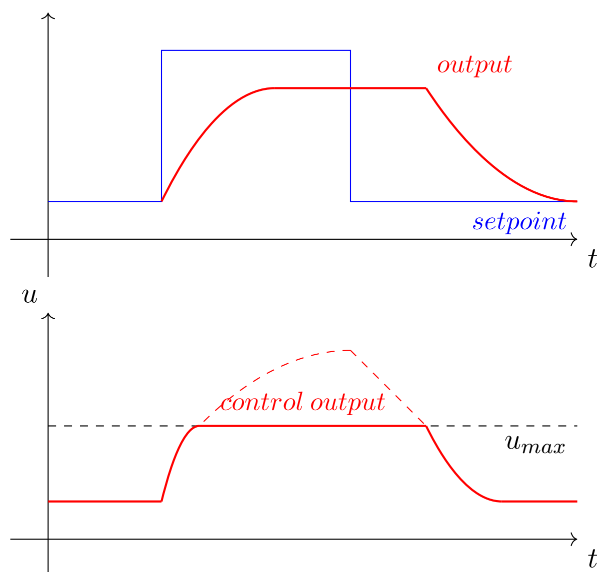


Figure 5.7: Windup

5.3.3 Controller Tuning

When tuning the controller, the target was a first-order response with a fast rise time and no overshoot. This response gave the best results when tracking a trajectory. Not having overshoot eliminated some of the jerk as setpoints got updated along the trajectory. The controller parameters were first tuned using the Ziegler-Nichols method on the brushed DC-Motors without load (Hveem and Bjørvik 2014). ZN gave a good starting point for further tuning. The system dynamics were unknown when tuning the controller. This limited the use of other tuning principles that use either a Transfer function or some other system model, like the SIMC method. When the load was attached and the robot fully assembled, the step-response method was used (Hveem and Bjørvik 2014). The ZN method is unpractical for the assembled robot due to the risk of links colliding and losing control when the system is close to unstable or if it became unstable during testing and could not be used. The parameters were manually tuned to give the desired first-order response with little overshoot by comparing the two methods.

Controlling velocity was done by ramping the setpoint instead of having a large step every time it was updated. The desired speed determines the rate at which the setpoint is ramped. This desired speed is planned alongside the position by the trajectory planner. By tuning the controller to follow the ramped setpoint, the result is a smooth gate without sudden jumps at setpoint updates. The response from this controller, implemented in a digital control system described in chapter six, is presented in the main results.

Chapter 6

Embedded Systems

6.1 Introduction

An embedded system is a microprocessor-based system with integrated memory and peripheral devices that are built to control a function or range of functions, and are not designed to be programmed by the end-user in the same way that a PC is (Heath 2003). It is estimated that ninety-eight percent of produced microprocessors are used in embedded circuits and only 2 percent in computers, (Barr 2021). Embedded systems usually control physical operations through the use of an embedded machine. In complex embedded systems, communication between the different microprocessors or the outside world is done using peripheral devices. This project's embedded system controls four brushed DC motors using incremental encoders and the Arduino platform, which communicates with ROS running on a PC as described in chapter 7. This chapter will cover the necessary theoretical framework to understand the implemented embedded system, its flaws, and the reasoning behind the different decision.

6.2 Research Method and Equipment

6.2.1 Method

The knowledge and methods acquired during the electric circuit analysis, electronics, and computer technology classes were used to design and program the electronics in this chapter. In addition, further research was conducted for topics that went beyond the scope of these curricula. All parts were connected using spring-loaded connectors or soldered together. The programming was done using Visual Studio Code while compiling, and upload was done using Arduino IDE. Version control was done through Git.

6.2.2 Equipment

Name	**Description**	**Documentation**
Arduino Mega		Arduino 2021a
Arduino Nano		Arduino 2021b
Assorted Wires		
Capacitor, $460\ \mu$F, 16 v		
Mean Well power supply	24 V, 400 W	Mean Well 2021
Metcal MX-500	Soldering station	Metcal 2021
Pin headers		
Pololu Dual G2	High-Power Motor Driver 24v14 Shield for Arduino	Pololu 2021b
Pololu 37D	24 V, 150:1 Metal Gearmotor with 64 CPR encoder	Pololu 2021a
Pololu mounting hub	Universal Aluminium Mounting Hub for 6mm Shaft, M3 Holes	Pololu 2021c
Protoboard		
Schottky diode, bat43		

6.2.3 Software

Name	**Description**	**Documentation**
Arduino IDE		Arduino 2021c
DIYLC	Layout creator	DIY Fever 2021
Gears App	Gear design program by Drive Train Hub	Drive Train Hub 2021
Visual Studio Code		Microsoft 2021

6.2.4 Arduino libraries

Name	Description	Documentation
DualG2HighPowerMotorShield		Pololu 2020
EEPROM		Arduino 2021d
rosserial		Purvis et al. 2021
stdlib		Michalkiewicz and Wunsch 2017
Wire		Arduino 2021e

6.3 Theoretical Framework

6.3.1 Actuator Types

Brushed DC-motors

In a simplified form, brushed DC-motors consist of a rotor, a stator, and brushes that enable an electrical connection to the rotor through a commutator ring. The stator is made up of either permanent magnets or coils that produce a magnetic field when a current runs through them. The stator encloses the rotor so that the rotor can rotate within the magnetic field. The rotor consists of one or more wire loops that make a complete circuit through the brushes connected to the motor driver. The current running through the rotor windings creates a magnetic field on its own. The two magnetic fields generate a force on the rotor perpendicular to the stator magnetic field and, in turn, creates a rotating movement. Slits in the commutator ring enable the current direction to change so that continuous rotation can be achieved. This mechanical connection introduces the problem of wear on the motor. In industrial settings, the brushes would have to be changed and the commutator ring cleaned to ensure optimal performance. Due to this, brushed motors have been changed for brushless types in many applications. The brushed motors are often cheaper than the alternatives. The cost, combined with less complex driver systems, makes it an option in some situations (Umans 2013).

Brushless DC-motors

The working principle behind the brushless DC motor is the same as for the brushed variant. The main difference is that the rotor windings are replaced with permanent magnets in a brushless DC motor. This makes the brush and commutator ring obsolete. As a result, the stator of this type of motor often has many more winding pairs, and the moving magnetic field is controlled by powering the stator coils in a set pattern (Moreton 1999). Thanks to this configuration, many brushless DC motors are

designed with the rotor outside the stator. This is called an outrunner motor in opposition to the traditional inrunner configuration. This outside rotating configuration has become popular in drone and robot designs. One example of this type of motor is the actuator designed for MIT's Cheetah robot (MIT 2021).

Stepper motors

Stepper motors are designed with the same principles as the brushless DC motor. Several stator coils are used to control the position of the permanent magnets in the rotor. The difference is that stepper motors are designed to hold the rotor in a set position, where the brushless variant is designed to make movement between the poles smoother. This gives the stepper motor a stronger hold torque. Stepper motors have been used in walking robots and offer good accuracy in terms of joint angle. The accuracy is limited to the number of coils in the stator, but using half steps, where two coils are powered simultaneously, doubles the number of steps per rotation. Due to the actuator design, the stepper motor is prone to slipping if the torque nears the torque limits. An encoder to measure the angle is therefore often used as an additional measurement. The torque of the stepper motor drops as the speed increases, making it unsuited for high-speed applications (Freimanis 2021).

Servo motors

Servo motors are brushed or brushless motors with a built-in encoder providing a feedback loop to work with angles instead of rotation. This paragraph, however, will deal with the type of servo motors with a built-in feedback loop and regulator. These types of actuators are popular in hobby projects as angle-regulation is done automatically. They are also often used in spider-/insect-like robots like the MX-Phoenix by Zenta Robotic Creations (Halvorsen 2021). Due to having all controllers internal, these servos only take in a PWM signal that gives the desired angle, and the controller does the rest. Most common servos do not have feedback outputs providing the actual angle (Hughes and Drury 2013).

6.3.2 Back Driven actuators

Back-drive-ability is the ability to turn the motor axle when the motor is not powered. This can be from the mechanical load on the motor itself or from external manipulation. This ability can be both positive and negative depending on the robot's area of use. Back-drive-ability can prevent the robot from reaching its desired Cartesian coordinates but can also prevent damage to the robot and to humans in contact with the robot. A high gear ratio will make back driving harder to achieve (Sol 2021).

6.3.3 Braking

Braking involves locking the motor in its current position and can be done mechanical or electrical. Mechanical braking involves applying friction by pressing two plates

together or by having teeth locking the parts together. Both these methods usually use a spring-loaded or magnetic mechanism to engage (Dragone 2021). Electrical braking involves reversing the function of the motor and turning it into a generator. The energy can either be burnt off as heat in resistors or used to recharge batteries. Electrical braking is used more in electrified transportation and not so much in robotics (Mohan, Undeland and Robbins 2002). As mentioned in the back-driven actuators section, a high gear ratio will make the actuators hard to rotate. This comes from the friction created between the gears and will therefore vary by the type of gear and the gear ratio. The high gearing used in this project can mimic a braking system but not completely replace it. One of the more common usages of braking in robotics is to lock the robot in place if the power cuts out, preventing the robot from collapsing (Dragone 2021).

6.3.4 Gearing

Gearing is used for increasing or decreasing the speed of a connected motor. The torque a motor can deliver is proportional to the gear ratio, meaning that a high gear ratio will enable the motor to deliver more torque. Different gear types exist that have varying properties pertaining to size, max gear ratio, efficiency, and backlash. In this section, the theory for some of the more common types used in rotational joints will be presented.

Backlash

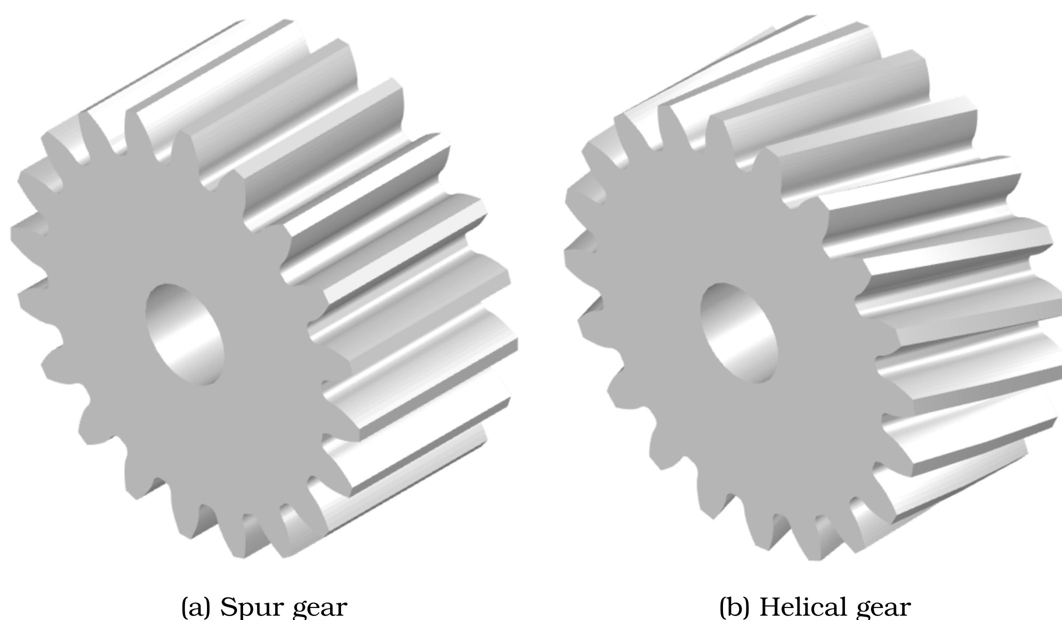
Theoretical, no gaps are needed between the teeth of connected gears. However, manufacturing tolerances and needed room for lubricants between the gears ensure that all traditional gears have some gaps. The rotational play between the gears is called backlash and can lower the accuracy and precision of the angles wanted for each actuator (Jelaska 2012).

Spur gears

Spur gears are the most common type of gear and what most people would picture when imagining a gear. The teeth, or cogs, run along the outside or inside of the gear wall. The teeth run perpendicular to the gear face. An example of spur gear with the teeth on the outside can be seen in *Figure 6.1a*. Spur gears are noisy due to the teeth colliding head-on and can cause problems with vibration. (Jelaska 2012).

Helical gears

Helical gears are designed much like spur gears, with the teeth manufactured with an angle to the gear face. Due to this design, the teeth of a helical gear join together gradually along the width. This means that the helical gear creates less noise and vibrations than a spur gear. An example of a helical gear can be seen in *Figure 6.1b* (Jelaska 2012).



(a) Spur gear (b) Helical gear

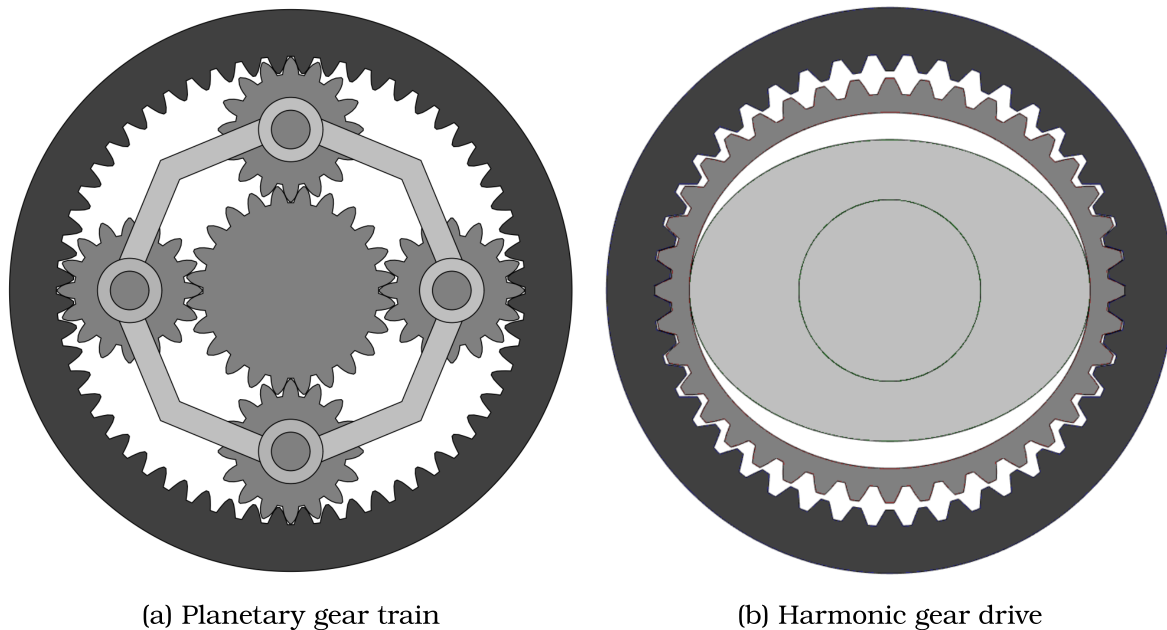
Figure 6.1: Different gear designs

Planetary gear train

A planetary gear train (or epicyclic gear train) consists of a central spur or helical gear called a sun gear. In robotics, this is usually the gear connected to the motor. Several "planetary" gears of the same type as the sun gear are connected together by a carrier that forces equal spacing around the sun gear surrounding the central gear. Then sits a ring gear that holds the planetary gears in place. Where the output is connected varies between the planetary carrier or the ring gear depending on the application. The gear train can be seen in *Figure 6.2a*. This type of gear train can achieve higher efficiency than many other designs (Jelaska 2012).

Harmonic gears

Harmonic gearing (or strain wave gearing) is an alternative to planetary gears. This type has a central hub and outer ring (annulus) like the planetary gears but no sun or planetary gear. Instead, the central hub is connected to an oval piece called a wave generator. Then a flexible toothed rim, with fewer teeth than the annulus ring, is added between the wave generator and the stationary annulus ring gear. Ball bearings are placed between the flexible rim and the wave generator (*Figure 6.2b*). When the wave generator rotates, it pushes the teeth of the flexible rim into the outer gear. Because the wave generator has fewer teeth than the annulus ring, the rim rotates to hit the outer gears. This creates a high gear ratio and close to zero backlash on the flexible rim. The output axle is often mounted directly to the flexible rim (Jelaska 2012).



(a) Planetary gear train (b) Harmonic gear drive

Figure 6.2: Different gear drives

6.3.5 Inter-Integrated Circuit

The I2C bus is one of the most well-known and simple serial communication methods. The bus consist of a clock (SCL) and a data (SDA) line, pulled up with resistors. It allows for communication through a Master and Slave setup. The master node generates the clock and initiates communication with slaves, and the slave node receives the clock and responds when addressed by the master. This bus connection allows for four different communication modes on the bus.

1. Master transmitting to a slave node.
2. Master receiving from a slave node.
3. Slave transmitting to master node.
4. Slave receiving from the master node.

Each message starts with a START condition and the slave address and ends with a STOP condition. In addition to the SDA and SCL pin, the signal ground should be shared for devices on the same bus. I2C communication is best suited for short-range connections (Heath 2003).

6.3.6 Serial Peripheral Interface

The SPI bus is another well-known communication protocol. Like the I2C bus, it has a clock (SCLK) and a data line for the master to slave (MOSI). However, it also consists of lines for communication to master from slave (MISO) and slave select (SS). Having different lines for communication from master to slave and slave to master enables SPI to send data in a continuous stream, making it faster than protocols that send data in packages. In addition, the slave select line makes addressing the slaves easier.

SPI communication is suited for systems that require higher speed than the I2C bus, (Campbell 2021).

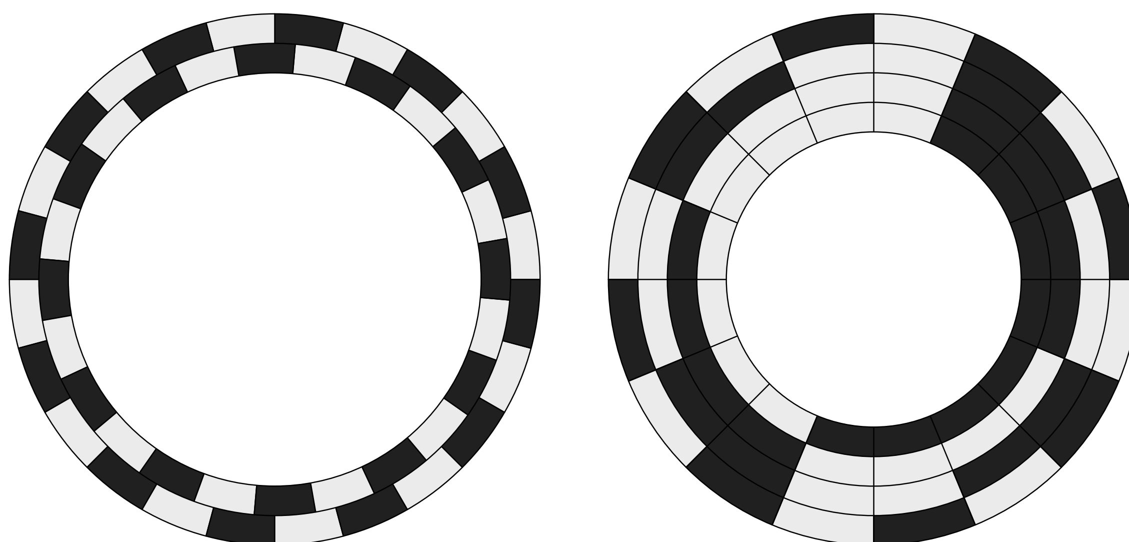
6.3.7 Encoders

Incremental Encoders

The incremental encoder, often called a rotary encoder, is a cheap and simple encoder. It provides changes in position through two output signals, Pulse A and Pulse B. The pulses are generated by having encoder discs like in *Figure 6.3a*, where sensors detect the gaps in the concentric rings. The absolute position can be tracked through the use of an incremental encoder interface. The encoder reports changes in position nearly instantaneously and is well suited for high-speed applications that require precision. The phase difference between Pulse a and Pulse B determines the direction of rotation. A difference of $+90°$ indicates clockwise rotation and $-90°$ counterclockwise. The frequency indicates the velocity. The resolution of an incremental encoder is specified in pulses per revolution (PPR) (Craig 2021).

Alternative Encoders

An alternative to incremental encoders is to use absolute encoders. An absolute encoder has rings laid out on the encoder disc in a way to represent bits in binary or Gray code (*Figure 6.3b*). This way, one would always know the angle of the actuator without having to start at a known angle like for an incremental encoder. However, the accuracy of an absolute encoder depends on the number of encoder rings. This means that this encoder can become too big if higher accuracy is needed (Craig 2021).



(a) Incremental encoder disc with two channels

(b) Absolute binary encoder disc with 4 bit accuracy

Figure 6.3: Different gear drives

Another alternative is to use a continuous potentiometer connected to the motor axle. By connecting the middle terminal to the microcontroller's analog input, and positive supply and ground to the others, one can read a binary value representing the absolute position of the actuator. The accuracy of this solution depends on the bit resolution of ADC used in the microcontroller.

6.3.8 Arduino Platform

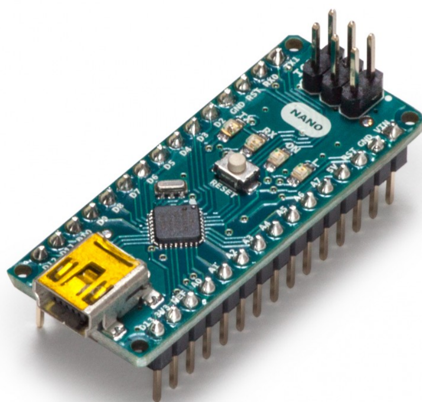
The Arduino platform is based on AVR ATmega microcontrollers equipped with digital and analog input/output pins. It features serial communication interfaces. The boards also have built-in memory such as flash for firmware uploaded by its boot loader and EEPROM support. The boards are programmed using the "Arduino language" which is a standard API that uses C and C++. The internal clock speed of the ATmega chips used in the Arduinos is 8 kHz, but the boards are equipped with external 16 kHz crystals to double the processing speed.

Arduino Nano

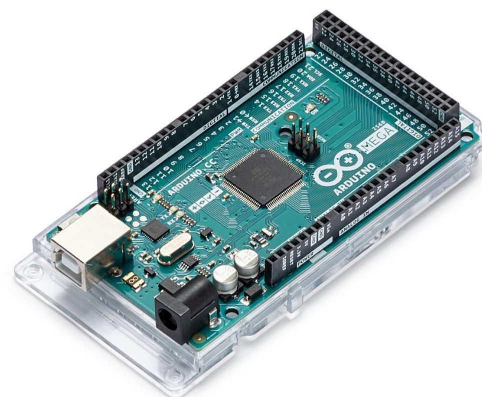
The Arduino Nano (*Figure 6.4a*) has 22 digital and 8 analog pins. It has a 32 kB flash memory to store programs and has a single RX/TX serial port.

Arduino Mega

The Arduino Mega (*Figure 6.4b*) has more digital pins (54) and analog pins (16) than the Nano. It also has more flash memory (256 kB) for more extensive programs. In addition, the Mega has the possibility for 4 RX/TX serial ports.



(a) Arduino Nano (Arduino 2021b)



(b) Arduino Mega (Arduino 2021a)

6.3.9 EEPROM

Electrically Erasable Programmable Read-Only Memory is an integrated circuit that consists of a re-programmable ROM. EEPROM is a non-volatile memory that allows bytes to be stored even when power is lost. Bytes are stored by using Floating-Gate

transistors witch keep their charge unchanged for long periods without current connected. Writing to the EEPROM is slower than other memory types like flash. Flash memory has a higher speed but is limited to around ten thousand writing cycles. EEPROM has a longer lifetime, often as high as one million cycles. Higher lifetime makes the EEPROM more suited to store a small number of parameters and history that changes more frequently. Both flash and EEPROM have an unlimited amount of reads, (Williams 2014).

6.4 Results and Empirical Findings

6.4.1 Embedded circuit schematic

The robot's electronics are laid out as seen in *Figure 6.5* where the Arduino Mega is the communication hub that connects ROS to the motor drivers and Arduino Nano slaves. The communication between the Mega and the Nano controllers is done via the I2C protocol. Here the SDA and SCL lines marked are in green and yellow in the schematic. The motor drivers are running directly from the Mega, and each motor encoder is connected to its separate Nano.

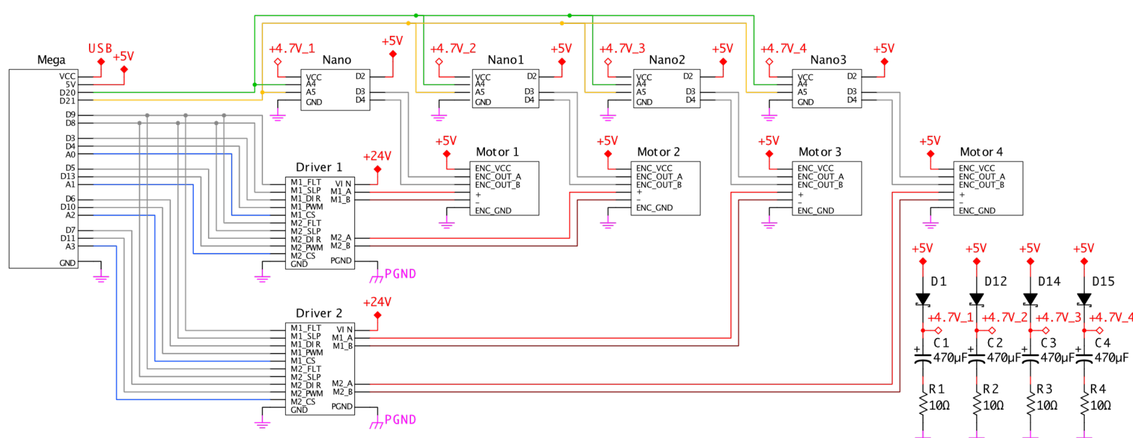


Figure 6.5: Embedded circuit schematic

The Arduino Mega comes with a mounting hub and has female headers already mounted, while the Nano comes with male headers. Therefore, each Nano was fitted to a protoboard that contains the controller, power buffer, and headers for connectivity. Two sets of two boards are stacked as seen in *Figure 6.6*.

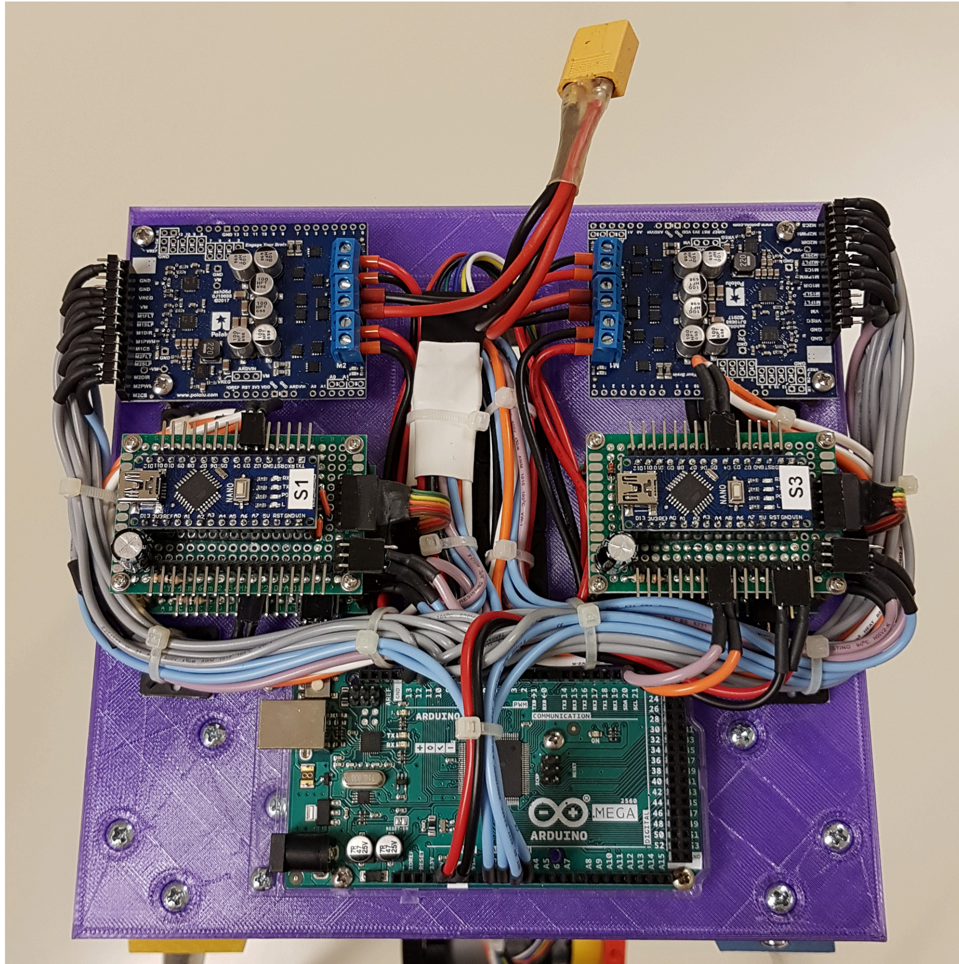


Figure 6.6: On-board electronics

Since I2C requires all units connected in parallel, a bus for SDA, SCL, and 5 V and ground were pulled from the Mega through all protoboards to simplify cabling. By powering all Nano boards and encoders from the Mega's 5 V output, only the USB cable are needed to power the controllers. The complete wiring for the Nano boards can be seen in *Figure 6.7*.

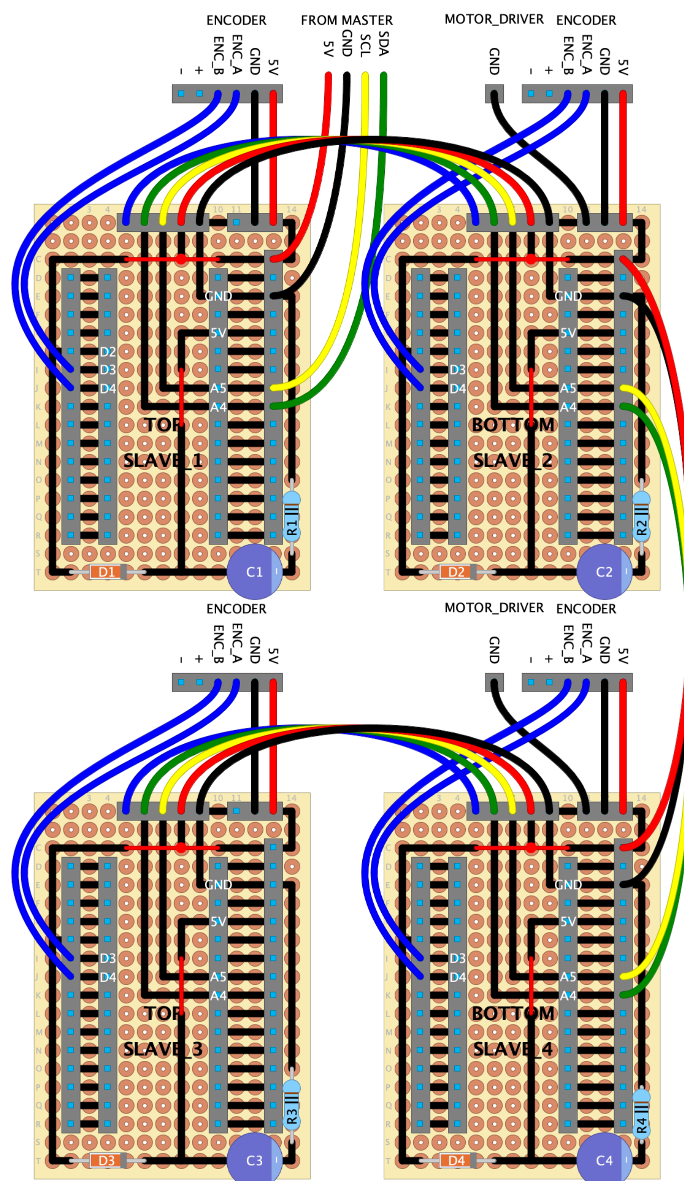


Figure 6.7: Arduino Nano circuit board

The motor drivers are powered through an external 24 V power supply, and the power ground is kept separate from the signal ground. The positive wire from the power supply is wired through an emergency stop (not in the schematic) so that the actuator power can be cut if anything goes wrong. The emergency stop was mounted temporarily to the power supply as seen in *Figure 6.8* to easy transport.

Figure 6.8: Power supply and emergency stop

6.4.2 Actuators

Due to budget constraints, the range of available motors was very limited. After some searching and deliberation, the decision was made to go with brushed gear motors with built-in incremental encoders. $150:1$ geared 24 V motors were bought from Pololu (Pololu 2021a) to have the best chance of getting strong enough motors. Unfortunately, these motors turned out to be too weak to drive the two first joints reliably. During the testing, two of the motors were damaged and could not be used further. Both ended up with the last gear in the gear train being damaged, and one of the motors burnt out. The actuators were then re-arranged so that the damaged actuators were placed as the first two actuators. As a result of this, no movement in the first joints could be used from this point on.

6.4.3 Embedded Controller

The purpose of the embedded controller is to actuate the embedded DC motors using motor drivers. The Arduino library for the motor drivers was used to set the output calculated from the discrete PID algorithm described in *subsection 5.3.2*, including anti-windup for the integrator. PID control calculated effort based on joint position

feedback. Because the DC motors all share the same properties, an actuator class was created. This class is a general actuator class and should work for other motors as well. It supports set & get functions to access the individual private values of the objects and a PID algorithm. The full properties of the class can be inspected in *Code snippet D.1* and *Code snippet D.2*. An array containing four actuator objects was created. This made it possible to access the different objects in a structured manner, and independent joint control was implemented by computing PID while looping through the list. The essential of this code is shown in *Code snippet 6.1*. This algorithm only shows how the effort is set to actuator 1. Since each motor driver supported two actuators, a separate if-statement had to be made for all four cases. More details can be found in *Code snippet D.4*.

Code snippet 6.1: Creating a object list used to control the four actuators

```
1  Actuator actuators[4] = {Actuator(8), Actuator(9), Actuator(10),
      Actuator(11)};
2  void controllActuators(Actuator actuators[]) {
3    for (int i = 0; i < numActuators; i++) {
4      actuators[i].readAngle();      // Stores angle from encoder
5      actuators[i].computePID();     // Comute pid and store output
6      if ( i == 0) {                 // Set speed to motor 1
7          md1.setM1Speed(actuators[i].getEffort());  // Motor 1 is
              driver 1, M1
8          }
9      }
10 }
11 void loop() {
12   stopIfFault();                  // Stops if fault
13   controllActuators(actuators);   // Pid controll on all the acuators
14   rosPub();                       // Publish to topics
15 }
```

The average loop time of the controller was found to be around $3 - 4$ ms depending on the calculations done, which gives the digital controller a time delay of $4.5 - 6$ ms calculated from equation *Equation 5.8*. Controller tuning was done as described in *subsection 5.3.3*, and the time delay did not make the system hard to control.

6.4.4 Incremental encoder Interface

An interface for the incremental encoder was added to keep track of the joint positions given by the changes in position. This was done by having a separate Arduino Nano board connected to each encoder. The sketch uploaded to slave one is shown in *Code snippet D.5*. The two pulses were recorded, and changes were saved as a counter. The joint position can be directly calculated from this counter variable by using the

properties of the encoder, described by *Equation 6.1*.

$$\theta = \frac{360°}{PPR} I \quad (6.1)$$

Where:

- PPR = Pulses per revolution, witch is 9600 if all flanks are counted.
- I = number of increments stored in the counter variable.

The counter variable is transmitted to the Arduino Mega, using I2C communication, running the control algorithm, and communicating with ROS. When sending the counter integer, it is divided into two bytes consisting of 8 bits and sent as a lower and upper byte described in *Code snippet 6.3*, before they are joined as an integer in the master node, *Code snippet 6.2*.

Code snippet 6.2: Receiving integer from slave

```
counter = Wire.read() << 8 | Wire.read(); // Read upper/lower bytes
```

Code snippet 6.3: Transmitting integer from slave

```
void requestEvent() {
  uint8_t buffer[2];

  buffer[0] = counter >> 8;      // Store the int as 2 bytes
  buffer[1] = counter & 0xff;

  Wire.write(buffer, 2);         // Respond with message of 2 bytes
}
```

Auto saving on power loss

The ATmega processor in the Arduino has a built-in EEPROM rated for 100.000 cycles and can store 512 bytes. This makes it easy to store and read encoder values using the Arduino. The Arduino has support for an EEPROM library that includes reading, writing, clearing, and updating the EEPROM. The *get* function checks the value stored in the address and only writes to the EEPROM if it has changed. When power is lost, the encoder increments are saved to the EEPROM using the *put* function. On program startup, the encoder increments initial value is read from the same EEPROM address. As the encoder increments are stored as a single integer on each Arduino Nano on shutdown, EEPROM speed or life cycles is not a problem. Power loss was detected using an interrupt pin on the falling edge connected to the 5v bus. Schematics for power loss detection and the power buffer can be seen in *Figure 6.9*.

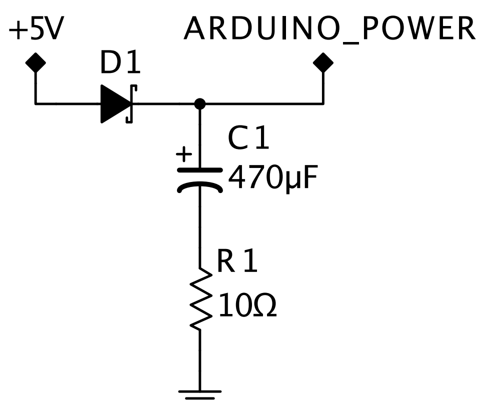


Figure 6.9: Power buffer

Code snippet 6.4: Saving to EEPROM

```
1 attachInterrupt(digitalPinToInterrupt(2), saveToERPROM, FALLING);
2 void saveToERPROM(){                    //ISR function for interupt
3     EEPROM.put(0, counter);             // Store counter
4     delay(1000);                        // Wait to die
5 }
```

6.5 Analysis and Discussion

6.5.1 Actuators, Gearing and Belt Drive

Due to budget constraints, the actuators available would not be strong enough without a high gear ratio. The $150 : 1$ gear ratio in the actuators used in this project meant that they were harder to turn manually. Initially, the actuators were too hard to turn by hand, but they seemed to loosen up after some use, and it was possible to pose the robot manually. Even with a $150 : 1$ gear ratio, the actuators are operating on their limits to hold up the leg in the common poses found in the trajectory.

As said in the results section, the strain on the actuators was too much, and two actuators were irreversibly damaged during testing. Both times, the damaged actuator was the one placed in joint two, which made sense as the joint takes almost the entire load of the leg in the air. The motor drivers' output was monitored but never showed any indication that the actuator was about to be overheated. For future work, this would have to be addressed by upgrading the actuators and implementing the current sensor abilities of the motor drivers.

Higher gearing means more of the actual cogwheels. As a result, each cogwheel adds more backlash to the system. This backlash manifests in movement in the joint that the encoders will not register. In this project, this non-measured angle seemed to be about $2-4°$ and will affect the movement greatly. Better, more expensive gearing would help or eliminate the backlash, as the gears used in the actuators are both helical

and spur type gears (Pololu 2021a). If strong enough actuators could be sourced, the robot could be direct driven. Alternatively, a planetary gearbox could be used for higher accuracy or a harmonic gearing that would eliminate backlash.

Alternative Actuators

Brushed DC motors are among the cheapest actuators when looking at stall torque versus cost. The main drawback is that the brushes will wear out over time, and the torque will decrease. With brushless motors, this would not be an issue. In addition, brushless motors usually weigh less than brushed of equal stall torque. The weight reduction achieved would help the robot move more easily, pulling less current and exerting less torque. Brushless motors also can achieve higher efficiency than brushed to reduce the current draw further.

If one wanted to step back on the complexity of the robot, other actuator types could be considered. For example, stepper motors could be controlled by the number of steps or half steps with a known angle for each step. As mentioned in the actuator theory, the stepper motor would have a higher holding torque making the robot more stable. Servo motors would simplify the control in the same way as both stepper and servo motors only need the desired position converted to either the number of steps or duty cycle for the servos PWM signal.

Belt Drive

One way to limit the weight, and thus the torque needed, is to place the actuators further up the leg and use a belt or possibly chain drive to transfer the torque to the joint. An example of this is the MIT Cheetah (Katz, Carlo and Kim 2019) where all three actuators are placed at the hip and belt drive is implemented to drive the lower joint. This configuration allows the leg to be extremely light, and minimal torque would be needed to lift it. The total weight would still be the same or slightly heavier due to the belts. If the upgraded actuators still were too weak, one could implement a gearing system for the belt drive where the toothed pulley on the actuator is smaller than the one on the joint. This would work like having gearing installed.

6.5.2 Mounting Hub

After the first assembly, many sources of backlash were felt on the joints. Some were found to be the gear backlash and too wide screw holes, as discussed previously. However, some of the backlashes seem to come from the mounting hub itself. It was found that the set screws locking the round hole of the mounting hub to the D-shaped axle of the actuator would loosen as the robot was manipulated. When the set screw loosened, it introduces some play between the axle and mounting hub. Some thread-locking fluid was added to the set screws to fix this. Medium strength was used so that the hub could be removed in the future. A future safe solution would be to have new mounting hubs made with a D-shaped hole to fit the actuator axle better.

6.5.3 Encoders

The code used to count and ascertain the direction of the motor only counts the rising and falling edge of pulse train A. This means that only half of the possible accuracy in measuring the angle is used. Pulse train B is then only used to view the direction. The accuracy could theoretically be doubled by counting the rising and falling edges for pulse train B. As the situation stands with the excessive backlash in the gearing and flexing of the plastic, this increase of accuracy would probably not be needed. Because of this, the final possible accuracy became $0.075°$.

As the project uses incremental rotary encoders, a solution to store the final angles had to be developed. If the robot was manipulated in between the sessions, these angles would be incorrect, and a way to reset the angles to a known angle would be needed. A solution with end stop sensors will be discussed later. Another possibility to consider if the actuators were to be changed is to use an absolute encoder or a continuous rotational potentiometer. Both these types would give the angular position of the joints as soon as they have power, and one would not need to reset them.

6.5.4 Circuit Design

When designing the embedded circuit, several different hardware limitations had to be handled. Some of these include:

1. Limited interrupt pins and clock frequency
2. EEPROM life cycles
3. Dual Motor drivers

Limited interrupt pins and clock frequency

A limited number of interrupt pins on the Arduino Mega made it necessary to use individual microcontrollers as Encoder Interfaces. The maximum angular velocity of the Polulu motors used is 7.12 rad/s, which would result in new pulses from the encoder every 0.74 ms. This is faster than the loop time of the Arduino Mega, and thus increments would be lost. However, the loop time in the slaves is fast enough as they perform no heavy calculations. The interrupt pins on the slaves were used to detect power loss, as discussed under EEPROM implementation. Communication between the Arduino Mega and Nanos was done by using I2C. Different communication peripherals such as SPI were considered, but the simple implementation of the I2C bus and wire library made it the obvious choice for this thesis.

EEPROM life cycles

Saving encoder data and loading these is crucial for a smooth startup. The internal EEPROM on the Arduino boards supports a limited number of value updates. Continuously updating the encoder data to the EEPROM would wear it down quickly and slow down the program significantly. In addition, as previously discussed, a slower

program could result in lost pulses from the encoder. Saving the data to a directly connected SD card reader was also considered but deemed too slow for continuous saving. Using an external EEPROM or SD card reader connected to the I2C bus would allow for continuous saving as it would not slow down the program too much. This implementation would need extra hardware and is therefore not implemented. Detecting power loss and saving the data was the obvious choice since this means no interruption during regular operation. Using a battery to keep the power on power loss was considered. However, adding a small capacitor and preventing it from discharging into the 5v bus using a Schottky diode provided a low-cost and lasting solution. A Schottky diode minimizes the voltage drop over the diode, giving the slaves a 4.7 v supply voltage. As mentioned, detecting power loss was made using a falling edge interrupt pin. This results in immediate data storing on power loss using minimal extra hardware, making it the best choice in this thesis.

Dual motor driver

During the preliminary phase of this project, dual motor drivers were chosen compared to mono drivers due to budget constraints. This resulted in all control being done in the Arduino Mega, as it has enough memory to handle the program size. Using four mono motor drivers would make it possible to implement control from each slave, which would remove transport delay between the master and the slaves. The control algorithm would also operate for single joints, which would increase performance due to reduced loop time. In addition, having control done by the slaves would open up for having the master as a communication link between the slaves and ROS interfaces.

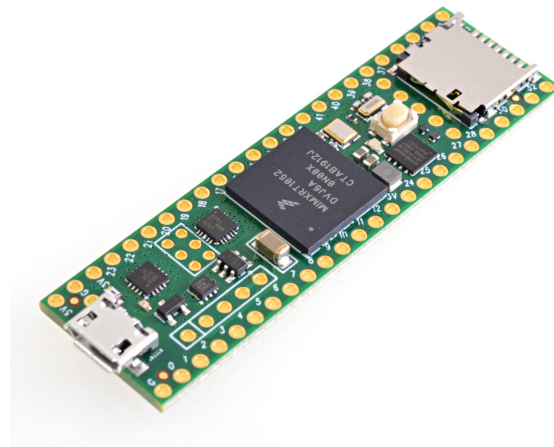
The dual drivers were made as a shield for the Arduino Uno. Using them with the Mega board meant that additional wiring had to be done. If control is moved to the slaves in future work, mono motor drivers would be required.

6.5.5 Choice of Microcontrollers

By using Arduino Mega and Nano, many premade functions and resources are available. The downside is that these microcontrollers run at 16 MHz, and loop time could be an issue. During testing of the controller on the actual robot, this manifested in a noticeably delay between ROS setting a new setpoint and the controller reacting to it. This meant that the robot exhibited a jerking motion between each setpoint. The Arduino family of controllers uses a PWM frequency of 490 Hz as standard. This is well below the 5 kHz specified in the Arduino library for the motor drivers (Pololu 2020). The frequency on the Arduino can be adjusted up to 20 kHz, but this would prevent both normal serial communication and ROS-serial. Using this low frequency means that noise from the motors could be a problem. Another issue is that the control signal represented by the PWM signal would appear wavy by the motors and constantly try to move slightly, drawing more power (Mohan, Undeland and Robbins 2002).

For further work, upgrading at least the Arduino Mega would be recommended. The most natural alternatives would be an ESP32 module (Espressif 2021) or a Teensy (PJRC 2021). The ESP32 has a clock speed of up to 240 MHz. It also has the option for both WiFi and Bluetooth for easier connectivity between ROS and the controller. The Teensy 4.1 does not have wireless communications, but it has a clock speed of 600 MHz and built-in support for Ethernet.

(a) ESP32 (Espressif 2021)



(b) Teensy 4.1 (PJRC 2021)

Another choice would be to use a Raspberry Pi (The Raspberry Pi Foundation 2021). Since the Raspberry Pi is a small computer with an operating system, integration with ROS is simpler. ROS can be installed directly on the device, and latency in the serial communication between the current Arduino Mega and a computer would not be an issue. The Raspberry also has both Bluetooth and WiFi and a CPU with a 1.5 GHz clock speed. Even though the Raspberry has a higher clock speed than the Teensy, some of the computing power for the Raspberry will be used to run the operating system.

Figure 6.11: Raspberry Pi 4 (The Raspberry Pi Foundation 2021)

The choice between these or other options depends on what aspects are prioritized. For example, going with some of the faster microcontrollers could eliminate the need for using one Arduino Nano on each encoder. Alternatively, one could change the Nano for a faster model and run the regulator for each motor separately.

The biggest reason to go wireless is that the computer would not need to be tethered to the robot leaving only the power cable. On the other hand, wireless communications are often slower than wired, so this would have to be considered. Furthermore, removing the wired communication would eliminate any noise from using a long USB cable.

Since the power is distributed through the Arduino Mega, another solution for powering the controllers would be needed if the Mega was to be changed out. The easiest solution would be to use the onboard regulator included in the motor drivers. The only problem with this solution is that the emergency stop cuts power to the drivers. If the emergency stop switch were moved to after the drivers changing it for a 4-pole normally closed switch, this would still work, and the drivers would be able to send status to the controllers even after the emergency stop was engaged.

6.5.6 Current Sense

The motor drivers have built-in current sensors that can be used to limit the current to the motors. However, because the PWM frequency was low, this measurement would be unstable. If better microcontrollers could be used, this would be one of the upgrades recommended and could have saved the damaged actuators.

6.5.7 Software Design

The software running in the embedded microcontrollers was made as modular as possible. For this reason, a custom actuator class was made. By taking the individual properties such as slave addresses, control parameters, and gear ratios as parameters in the constructor, the individual properties of the motor were easily managed. If a different gear ratio or encoder accuracy were to be used in the future, it could easily be managed by the gear ratio properties of the actuator class. This supports different PPR for each individual motor as well. Implementing PID control from an existing Arduino library was discussed. The benefits of a custom PID algorithm became apparent. These benefits include the possibility of a derivative filter, anti-windup, adaptive control by changing parameters, gravity compensation, limiting setpoint rate, and more. In the future development of this robot, more features can easily be implemented to the PID.

When writing modular code, dividing the different parts of the program into separate files makes for readable and easily modifiable code. In the git repository, two different versions of the main program are available, one with ROS control and one controlled from the serial monitor. The Arduino main program controlled from the serial monitor is provided in the git repository as it makes expanding the project easier since the serial monitor can be used for printing results to verify them. *Rosserial* does not work in parallel with Arduino serial monitor. The serial communication version also includes support for writing joint values and a plot of these to two I2C Oled displays. This feature was removed from the ROS version to improve loop time.

Communication with ROS was easy to implement as the *rosserial* library was an extension to how nodes were programmed using C++ and python. How these nodes are programmed and the *rosserial* communication is explained in *chapter 7*. Implementation of multi-thread programming would be beneficial if a microcontroller with more cores was used, such as the esp32. As the Arduino Mega only has one core, multi-thread benefits using an Arduino library such as *FreeRTOS* are limited due to an already low clock speed. Multiple cores could allow for one core to do calculations and the other one to handle communication.

As discussed, actuating the DC motors is done using motor drivers. These drivers have Arduino support using the *DualG2HighPowerMotorShield* Arduino library. The shield was designed for use on a single Arduino Uno. As a single Arduino Mega is used for two drivers in this project, the input and output pins had to be reprogrammed. This is done in the variable file shown in *Code snippet D.3*. Controlling both drivers from one Arduino Mega made it necessary to check each motor and assign the correct motor driver and motor number when PID output was set.

6.5.8 Future Expansions

Force sensor

One key element that was not explored in the design of the robot was force feedback from the end effector to ensure a good balance between friction for movement and torque needed. One way to measure this is to attach a force-sensitive resistor or load cell underneath the end effector or between some of the parts for protection. Another way would be to make a flexible part attached to the end effector with a gap. Then one could measure the gap width and calculate the force needed for the compression.

Gyro and Accelerometer

One possible future goal for this robot would be to mount four legs to a body and have it stand and walk without using the stand. This can be achieved by adding some additional sensors to keep track of the orientation and acceleration in a 3D environment. The interconnectivity of the four legs, and relations between the robot body and ground, could be addressed and accounted for by adding a gyroscope and accelerometer, or simply an IMU. These additional sensors would give the control system additional information, which can be used for different control strategies.

End stop sensors

Currently, the starting position of the actuator angles is set by either placing the leg in the desired position and telling the controller the current angles or by always remembering the last position before the controller loses power. Both these methods require some procedure for shutting down, starting up, and storing the robot. A better solution would be to add a sensor that can detect the angles of the leg without having to manually measure them or having to rely on the leg staying stationary between uses. By having all actuators contract the leg upwards, micro switches, magnetic reed switches, or hall effect sensors can be used to detect when they reach the maximum/minimum angle for each joint. This procedure could be implemented during the startup of the controllers ensuring that the robot always starts in the same pose with all angles correctly set.

Navigational Sensors

Navigational sensors like lidar and cameras enable feedback on a robot's surroundings, making it possible to maneuver physical obstructions. This feature would be necessary if the robot is ever made autonomous. Paired with a system that manages the input from the sensor, this would be a natural expansion in later stages of the robot development. For example, placing the sensor or camera in front of the top plate tilting downwards would allow the robot to calculate immediate obstructions in its path. An alternative is placing a spinning sensor or camera at the top of the robot. This would provide a 360-degree view of its environment but could limit the perception

of immediate ground-level obstacles given the robot's height.

The use of navigational sensors in combination with SLAM algorithms are discussed in *subsubsection 7.5.6*.

Battery

A logical final expansion for the robot would be to make it completely untethered by adding a battery pack to it. Combined with wireless communications, this would let the robot walk without hindrance or range limitations. A prerequisite for adding a battery would be stronger actuators and more robust material used for the links as a battery would add significant weight to the top of the robot. Since lithium batteries would be the best choice in terms of energy density, additional precautions would have to be taken to store and charge the batteries.

6.6 Chapter Conclusion

As already discussed, many aspects of the embedded systems were undersized for what was needed. For example, the actuators were not strong enough to reliably reach and hold all desired poses and the microcontrollers were not fast enough to ensure smooth control. Despite this, the remaining parts of the embedded system worked as intended.

Angle measurements were created in the slave units, using the motor's built-in encoders, and transmitted to the master controller that again transmitted them to ROS on a computer. Only half of the possible accuracy was used, but this was not deemed necessary due to backlash and mechanical play.

The control of the actuators worked as intended by ROS sending out setpoints for each joint and the actuating signal being sent from the master unit to the motor drivers. However, the loop speed of the controller introduced significant transport delay and made smooth control of the actuators difficult to achieve.

Storage of final actuator angles with minimal components worked well and solved the problem related to the initial value for the incremental encoder. However, better fastening to ensure the links were locked in place, e.g., 3D-printed locking brackets, should still be considered moving forward.

Some of the shortcomings (i.e., actuators and encoders) can be attributed to budgetary constraints. In contrast, others (i.e., microcontrollers and the communication method) were due to time constraints and the extent of knowledge for the participants.

Since the actuators were not strong enough for the task selected, more care should have been taken when sourcing them. Even to the point of scaling back the project to accommodate better ones if restricted by the budget.

Chapter 7

Robot Operating System

7.1 Introduction

Robot Operating System, or ROS for short, is an open-source collaborative project that aims to make robot software development easier for everyone. The name "robot operating system" is somewhat misleading as ROS is not truly an operating system in the definition of the word. ROS is more like a framework, often termed as a "middleware" based on an already existing operating system. It is called a middleware because it functions as a pipeline between two or more software programs to send and receive information with each other efficiently. This communication infrastructure is described in more detail in *subsection 7.3.1*. Ubuntu is the primary operating system that officially supports ROS. ROS can also run on other well-known operating systems such as Windows and macOS, but it is then dependent on volunteers from the ROS community for continued support.

Besides the communication infrastructure in which information is shared between programs, the ROS core package also includes some basic client libraries, development tools, and a few other starting software packages. This provides the necessary tools and generic functionality that is sufficient to act as a framework for writing one's own robot software. The reason for creating an individual robot software is that robots can come in a wide variety of shapes and forms with entirely different purposes, forms of locomotion, communication, et cetera. Creating a general-purpose robot software that will work for any robot is therefore essentially impossible. However, by using ROS, one does not have to reinvent the wheel to make their own robot software but can build upon the foundation that ROS provides. The open-source concept is what keeps the ROS community thriving, as the international community of both independent and established developers works like an ecosystem that will hopefully continue to develop robot software for eventually any robot imaginable.

In this project, ROS uses a robot model that can be simulated and tested before building the physical robot leg. ROS is also used to generate a trajectory based on the

configuration of the robot model, which is sent to the controllers in the joints to track. Further development is taken into consideration, and various improvements to the project are suggested in the discussion section of this chapter.

7.2 Research Method and Equipment

7.2.1 Method

The beginner and intermediate tutorials created by the ROS team were used to get familiar with the ROS structure and concepts (Open Robotics 2021n). In addition, these tutorials provided the basic knowledge to start learning more project-specific features. Furthermore, available documentation and official guides were used for the different packages.

7.2.2 ROS packages

Name	Description	Documentation
MoveIt	Motion planning and manipulation	Ioan A. Sucan and Sachin Chitta 2021b
Plotjuggler	Plot graphs from published data	Faconti 2021
rosbash	Support for the rosrun command	Open Robotics 2021o
ros_controllers	Support for necessary controllers	Chitta et al. 2017
ROS Noetic full-desktop	ROS distro and basic packages	Open Robotics 2021m
rosserial	Serial communication with Arduino	Purvis et al. 2021 Ioan A. Sucan and Sachin Chitta 2021b

7.3 Theoretical Framework

7.3.1 Communication Infrastructure

The communication infrastructure that comes with ROS consists of a collection of communication tools. These are the cornerstones of any ROS application, and it is essential to understand how they work when using ROS for developing robot software.

Nodes

In ROS, a node is a process that performs computation and plays a relatively small and individual role as part of a larger software system. A robot application usually consists of many different processes that need to work together to give the robot full functionality. With the help of ROS, they are able to communicate with each other through a node graph architecture. The advantage of having multiple nodes run their own separate processes is making the overall system more fault-tolerant. If one node crashes, it does not stop the other nodes from running. As long as the most critical nodes keep working, the application as a whole can keep running at sub-optimal functionality. An example of this is a mobile robot with a mounted camera that an operator

remotely controls through a screen with the video feed. Should the image processing node for some reason stop working, the operator would still be able to control the robot through a line of sight, though at perhaps a less functional capacity. The ROS nodes can be written in several different programming languages with the use of ROS client libraries. The two most common are *roscpp* and *rospy*, which make it possible to write nodes in C++ and Python respectively. ROS communication has the attribute of being language-independent, meaning that nodes can easily communicate regardless of the programming language in which they are written. Nodes communicate with one another using the ROS communication tools known as messages, topics, services, actions, a parameter server, and a ROS Master node (Open Robotics 2021i).

ROS Master

The ROS Master is a node that is initialized at the startup sequence of every ROS system. The ROS Master is, as the name implies, the master in a master/slave relationship, while all the other nodes in the system are considered slaves. The master node serves as a form of communication hub for all the other nodes, similar to how a Domain Name System works. The master contains a registry with the name and location of every single node, topic, and service in the ROS system. Since every name in the ROS system is exclusive, nodes can refer to topics and services by name, and the master will translate it and relay their location. Once the locations are known, the nodes can communicate with each other peer-to-peer through the topics and services. The master also runs the parameter server, which will be discussed later in the subsection (Open Robotics 2021g).

Messages

Nodes communicate by passing data in the form of messages. A message is a simple data structure with its own distinct name and consists of various fields with specified data types. The supported data types are the standard primitive ones: integer, floating-point, boolean, et cetera. In addition, arrays of these primitive data types are also supported. The various packages in ROS include many predefined message types, but it is unproblematic to define custom messages for one's own application (Open Robotics 2021h).

Topics

When messages travel between nodes, they go through what is called topics. Topics function as a type of channel that nodes can either transmit messages to or receive messages from. In ROS, this is called publishing and subscribing to a node. Each topic has a distinct name to clarify what kind of data is published to it. Nodes can then easily acquire the data they need from other nodes by subscribing to the relevant topics. A topic can both be subscribed to and published to by multiple nodes simultaneously, kind of like how radio broadcasting works, where the name of the topic

is the frequency that the publisher nodes transmit to and subscriber nodes listen in on. This form of communication is asynchronous, meaning that there is no order in which operations need to happen. This makes for fast transmission of data, which is necessary for robot control systems, but does not offer any form of confirmation on sent or received messages. Like devices listening in on the same radio station, the subscriber nodes are unaware of the existence of each other as well as the nodes publishing to the topic. The publisher nodes transmitting messages to the topic are also unaware of each other. This is an important aspect of ROS, as the decoupling of the production of information from the consumption makes the system as a whole more robust (Open Robotics 2021u).

Services

While topics use a form of fire-and-forget, one-way communication model, services provide a means for a request-reply interaction between two nodes. This does not imply that a direct connection is established between two nodes. They still do not know of each other's existence. In this model, the service is attributed to a specific node, making the node the server in a server/client relationship. Any other node can function as a client by sending a request message with the appropriate data structure for the given service. The server node will process the request and return a reply message. This communication is synchronous, meaning that operations must happen in a given order. After a client node has sent a request to a server node, it is essentially blocked until it has received the server node's reply. Therefore, this form of communication is only recommended for processes that do not require a long time for the server node to compute (Open Robotics 2021s).

Actions

The ability to send requests to nodes to perform specific tasks and receive a form of reply when the task is finished is extremely useful and a necessity in larger ROS-based systems. Still, the fact that services block the client node while it is waiting for a reply can cause some problems when dealing with longer tasks. This issue is addressed with the implementation of ROS actions. Actions are built upon topics and provide a form of asynchronous server/client relationship between nodes. Instead of a request message, the action client node can send a goal to the action server node, which then starts executing the task. While the server is working, the client is free to do its own thing. It can even cancel the process that the server is working on at any time, or it can send the server a new goal that it will immediately start working on instead. Since the process can take a long time to execute, the server also has the means of continually sending feedback messages to the client, for example, to let it know how far along it is in the process. When the action server has completed the task, it sends a result message back to the client, similarly to how a service would, saying, for example, that the task was completed successfully (Open Robotics 2021a).

Parameter server

The parameter server is a register of global variables that are available to every node in the current instance of the ROS system. It is created in the ROS Master at the startup of a session and can be accessed and modified at any time during the session. The parameter server is good for storing variables that seldom change their value and are relevant to many different nodes. This way, the variables can be defined in one place instead of locally in every single node. Values that are typically found on the parameter server are configuration parameters like the name of the robot, link lengths, operating frequencies of sensors, filesystem paths, et cetera (Open Robotics 2021k).

7.3.2 Filesystem

Workspace

A workspace is a repository where one would organize a ROS project. The workspace is comprised of three folders: *build*, *devel* and *src*. The *src* folder is created manually and is where the packages that comprise the project's software ecosystem are located. The *build* and *devel* folders can be automatically generated by the ROS build system, *catkin*. Inside the generated *devel* folder are several *setup.*sh* files. One of these files must be sourced to be able to utilize the packages in the workspace. For an overview of how *catkin* works, consult the ROS wiki by Open Robotics 2021b.

Packages

ROS software is organized in the form of packages, which is the smallest unit that ROS software can be built and released in. A package can vary largely in its contents. However, a rule of thumb is that it should provide enough functionality to be useful but compact enough to be reused by other software easily. Packages tend to contain some common files and directories. A few examples of this are the folders called *config*, *launch*, *urdf* and *scripts*, that contain configuration files, launch files, URDF files and executable python scripts respectively. It is not a requirement for packages to contain nodes. It can, for example, hold just a collection of datasets as long as it makes for a useful module. Packages always have a *package.xml* file in their root folder that acts as the packages manifest. It provides meta information about the package, including its name, version, description, license information, maintainer, et cetera (Open Robotics 2021d). Packages also always contain a *CMakeLists.txt* file. This gives information to the *CMake* build system, which *catkin* is based on, that describe how to build the code and where to install it to (Open Robotics 2021c). (Open Robotics 2021j).

Unified Robot Description Format

URDF is built using XML code to describe the properties of a robot. This includes links and joints and how they are connected. For links, this is done by adding inertial elements, visual elements, and collision elements. For joints, this is done by adding

the type of joint, origin, parent and child link, and the axis it rotates around. Physical properties are only needed for simulations. Visualizing the robot can be done with only visual properties.

Inertial values consist of the physical properties mass, origin, and moment of inertia. Origin describes where the link has its origin, and the angular mass inertia around this origin is described by the moment of inertia matrix. For a rigid body, the moment of inertia describes the torque needed for a desired angular acceleration. For rigid bodies that are free to rotate in three dimensions, $\mathbb{R}^{3\times3}$, the moment of inertia is given by a 3×3 matrix.

Visual properties consist of; an origin, a geometry tag, and material. The geometry tag describes the geometry of the link. It is possible to use different shapes like "box" or simply exported STL files from the 3D model to get the most accurate visual representation. The material tag describes the visual color, while the physical impact of the material is found in the inertial properties. Completing the description of a link is done by adding the collision tag. The collision tag describes how it collides, which can also be done by an STL tag. Describing the collision with the STL files allows for more accurate collision checking than a box description and can be done using a low poly representation to save computing power.

Joints describe how the links are connected, which can be done by either a fixed or continuous joint. Fixed joints describe how two links are attached to each other rigidly, and continuous joints describe how two links are attached to each other via a rotation that may change. They both need an origin to describe where the two links are connected, but continuous joints also need to describe the axis of the rotation.

Launch Files

Single nodes can be started through the terminal with the command "rosrun". If several nodes are started in this fashion in individual terminals and correct order, they will eventually resemble a full application. Even so, the amount of different running terminals on the desktop can easily cause disorganization. Additionally, there are probably several parameters that need to be defined manually for the application to work, so this approach can be really tedious and impractical. The solution for this is using what is known as launch files. In a launch file, one can write a call for any number of nodes in a specific order as well as defining any number of parameters for the parameter server. Then, all the parameters can be set and all the nodes can be run in the specified order from this single file with the help of the terminal command "roslaunch". Launch files can also be launched from within other launch files. An application can then have several different launch files for starting isolated parts of the application, which makes for easier debugging, while for example having one main launch file that starts the whole application. This way, launch files make the process of starting an application much tidier, as well as saving the user a lot of time (Open

Robotics 2021p).

7.3.3 Project-essential applications

Following are descriptions of some fundamental ROS applications that play a big part in the completion of this project.

rqt

rqt is software that offers a graphical user interface with access to various tools in the form of plugins. It can make many ROS operations simpler to perform and information easier to break down for the user. Examples of this are the *Message Publisher* plugin and the *Node Graph* plugin. The *Message Publisher* makes it possible to choose a message type, specify its field values and publish it to a selected topic, all through a menu-driven interface. This saves time compared to publishing messages by manually typing out commands in a terminal. The *Node Graph* shows a graph structure of how all the currently running nodes are connected and to which topics they are publishing and subscribing. This paints a much clearer picture of the system's information flow. Generally, the *rqt* plugins make many aspects of ROS more user-friendly (Open Robotics 2021q).

RViz

RViz is a three-dimensional visualization tool for ROS. The appearance of a robot can be hard to imagine through only a URDF file, but with *RViz*, and help from the *robot_state_publisher*- and *tf2*-packages, the complete robot state is visualized. *RViz* can visualize the 3D shape of every link described in the URDF, though the composition of the links may be incorrect. The *robot_state_publisher* node can assist with this problem by translating the URDF files joint specifications, as well as listening for updates to the joint positions on the *joint_states* topic. The node combines the values of all the different joints on the robot and calculates the forward kinematics to get the correct coordinate frame of every link (Open Robotics 2021l). The coordinate frames can then be managed by the *tf2* library, which essentially functions as a database that keeps track of every transform in the robot system at all times (Open Robotics 2021t). These transforms can be subscribed to by the *RViz* node so that every link is connected in the right place with the current joint configuration. Lastly, with a joint state publishing node, for example, from the *joint_state_publisher_gui*-package, one can easily explore every possible configuration the robot may have visually through *RViz* (Open Robotics 2021f). *RViz* is a useful tool, not only for developing robot designs. It can also be modified with a large collection of plugins and act as a graphical user interface to interact with robots in a variety of different ways. *MoveIt* is an example of a ROS package that utilizes a modified form of *RViz* as a user interface and will be discussed later in this subsection (Open Robotics 2021r).

Gazebo

Gazebo is an open-source robot simulation software with high-quality graphics and a robust physics engine. Good simulations with realistic physics can be essential in robotic systems development as it offers the ability to test algorithms and robot design iterations rapidly. ROS can be integrated with *Gazebo* with a set of ROS packages named *gazebo_ros_pkgs*. The URDF also needs elaborating upon as the physical properties of all the links need to be specified since it will affect the behavior of the simulated robot. *Gazebo* will then make it possible to see how the robot will theoretically react to gravity, wind, and other physical disturbances that the robot may be influenced by in its planned working environment. Another use for *Gazebo* is to test the robot's control system by implementing simulated actuators and control them with the controllers provided by the *ros_control*-package. This package offers tools to create custom controllers with ROS, but it also contains a collection of generically functional controllers that can be used as-is. The primary controllers that are available are effort-, position- and velocity-controllers, named after the unit of their output value. The effort is measured in either force or torque, depending on the type of actuator, and is the most commonly used. These controllers are typically PID controllers with feedback loops. However, there are also a more advanced group of controllers called joint trajectory controllers. These have extra functionality for splining an entire trajectory instead of just a single setpoint. This is very useful for robots that need to perform more delicate and complicated movements. *Ros_control* is able to send and receive commands to actuators through a hardware interface that describes the relationship between actuators and joints. Many hardware interfaces are available via the *Hardware Resource Manager*, which is included in *ros_control*. To implement this, the URDF needs to be extended to include transmission elements for every joint that is linked to an actuator. The transmission element specifies the hardware interface that is being used. The transmission element can also specify mechanical reduction between joint space and actuator space (Chitta et al. 2017) to accommodate for physical mechanisms like gear ratios that affect the relationship between actuators and joint positions. Finally, for *Gazebo* to be fully integrated with *ros_control*, the *gazebo_ros_control* plugin needs to be added to the URDF (Open Robotics 2021v). With all these features, *Gazebo* can be used to develop a robot-proof of concept completely digitally, and thus save a lot of time and resources in the design phase of a project by not having to make as many physical prototypes as one would otherwise (Open Robotics 2021e).

MoveIt

One of the most difficult problems for robot manipulation is when a robot requires delicate movements to perform tasks. This is often the case even for tasks that will seem trivial to humans, like picking up an object with a gripper or moving a manipulator past an obstacle without colliding with it. Without a human operator controlling it, the robot needs to be able to plan its own trajectories to satisfy the goal of the task.

This is more complicated than it first sounds, as the trajectory from one location to another can have an unlimited amount of solutions, and the robot must be able to determine which one is the best. Motion planning algorithms is a very large field in and of itself, and a comprehensive treatment of this topic is beyond the scope of this thesis. After the motion is planned, it also needs to be executed by the robot, which often requires multiple timed adjustments in every active joint during the full range of the motion. Thankfully, *MoveIt* is a state-of-the-art robot manipulation software available to ROS that specializes in robot motion planning. The company behind it, PickNik Robotics, has made the software open source and fully customizable, thereby saving robot developers potentially months of development time. The software incorporates motion planners that give time-parameterized trajectories, 3D perception for sensors, common inverse kinematic solvers for even over-actuated manipulators, and collision detection capabilities. Despite the many advanced features, *MoveIt* is fairly user-friendly as it comes with its own setup assistant that guides the user to use the platform on any robot. Its only requirement is the URDF of the robot. The planning groups, controllers, sensors, end effectors, and more can be configured in the setup assistant. The full *MoveIt*-application is a large collection of packages, each with its own collection of nodes that run in unison while the application is active. A simplified overview of the flow of information shown in *Figure 7.1* should make things more clear. In the middle of the diagram is *MoveIts* primary node which is called *move_group*, and is the one that provides the robot interface command the robot. The user is able to interact with the node with one of three user interfaces, where one of them works through *RViz* with a plugin, as was mentioned earlier. The interactions usually consist of sending motion plan requests. These are either a goal for new joint positions that can be solved with the use of forward kinematics or a new end effector pose that can be solved with inverse kinematics. The *Move Group* fetches a *Planning Scene*, which is a description of the environment in which the robot must plan its motions. The *Move Group* then conveys command goals and the *Planning Scene* to a *Planning Pipeline* that consist of a *Planning Library* and *Planning Request Adapters*. The *Planning Library* takes the motion plan request and generates a possible trajectory for the robot to follow. There are a few libraries available, and the *Open Motion Planning Library*, or *OMPL* for short, is the one that is most commonly used. The *Planning Request Adapters* allow for pre-and post-processing operations to the planned trajectory. When the trajectory is determined, the *Trajectory Execution Manager* can transmit the necessary information through an action interface to the robot's controllers so that it can follow the planned trajectory. For further details consult the documentation on *MoveIts* homepage by Ioan A. Sucan and Sachin Chitta 2021b.

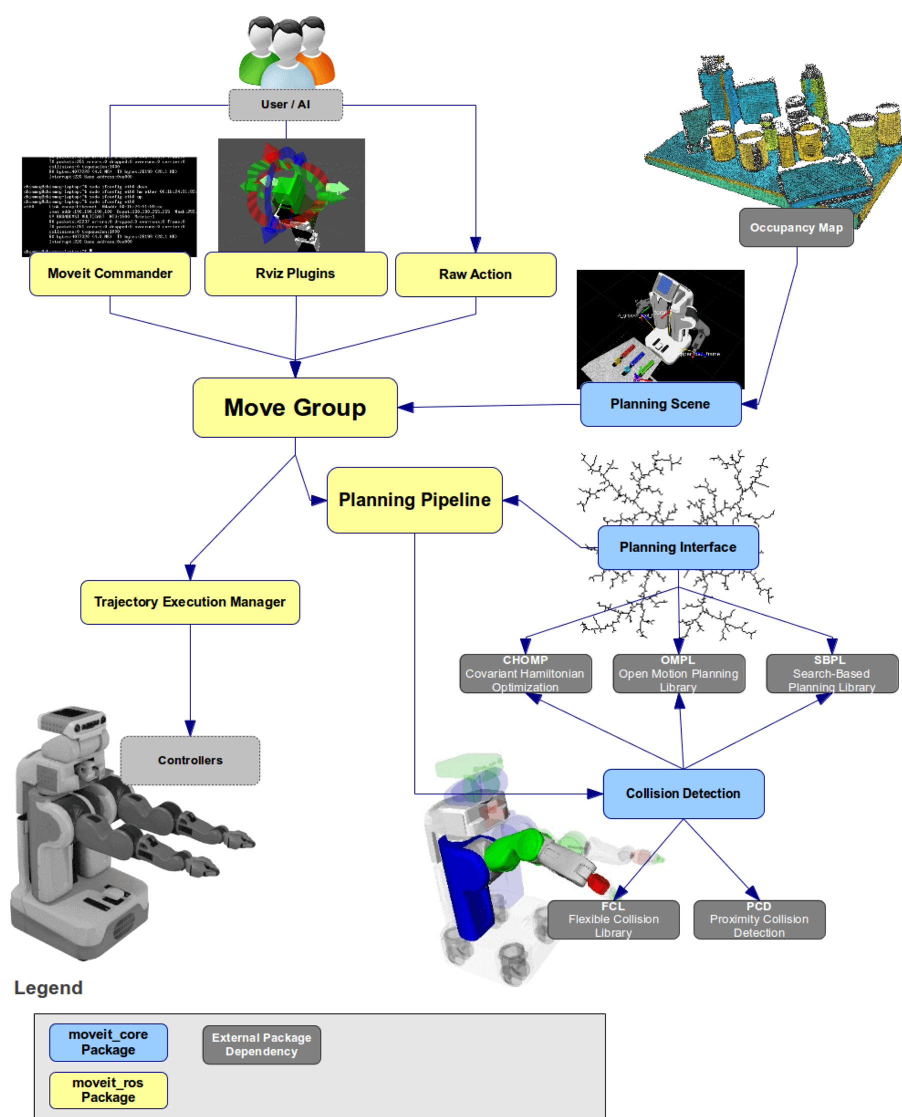


Figure 7.1: *MoveIt* pipeline overview (Ioan A. Sucan and Sachin Chitta 2021a)

7.4 Results and Empirical Findings

7.4.1 Source directory

The finalized source directory of the project workspace, with all its packages and sub folders, is shown in *Figure 7.2*. The contents of all the folders are available for examination on the projects GitHub repository (Arnesen, Grinde, Hovland and Vestland 2021b).

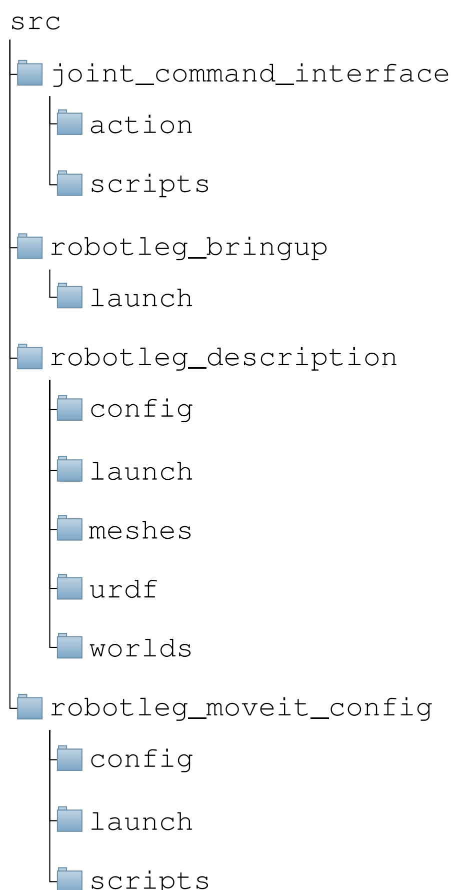


Figure 7.2: Source directory

7.4.2 Simulation

To simulate the robot in *Gazebo*, a launch file called *gazebo.launch* was made in the *robotleg_description* package. *gazebo.launch* starts by including another launch file from the *gazebo_ros* package called *empty_world.launch*, which simply starts an empty *Gazebo* simulation window. To be able to edit and save the simulation configuration, for example the starting view, a folder named *worlds* was created. There, the configuration can be saved and used as a launch argument. The robot model can then be spawned. The path to the robot's main URDF file is defined in the parameter server under the name *robot_description* so that it can easily be found by other nodes in the system. An executable from the *gazebo_ros* package called *spawn_model* then uses this parameter to have the robot model appear in the simulation window. The URDF describes the robot leg as fully extended so that the joints have a clear and recognizable zero-configuration. Since the extended leg is longer than the height of the stand, the robot model needs to be spawned slightly up in the air so that parts of the robot will not appear in the ground, as it can crash the simulation. This spawn position can be specified in the launch arguments of *spawn_model*. The simulation window on startup is shown in *Figure 7.3*.

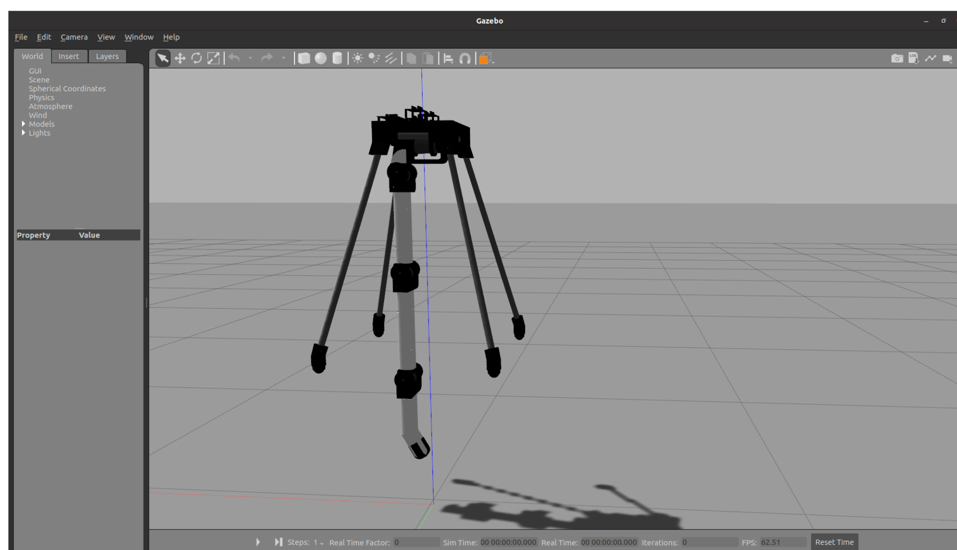


Figure 7.3: *Gazebo* simulation window on startup

Time will start by pressing the small play button on the bottom left of the window, and the model will fall to the floor. Additionally, the leg will collapse as there are no controllers to hold the joints in place. Controllers are added to the leg joints by first creating a controller configuration file stored in a new folder called *config*. The configuration files for controllers are by standard written as *.yaml* files, and the first file created is called *position_controller.yaml*. This file is filled with the parameters for a joint state controller as well as a position controller for every joint that is to be controlled. The joint state controller publishes the state of all joints registered to a hardware interface to the *joint_states* topic. The position controllers are PID controllers that use the joint position as a setpoint to determine effort. The PID parameters are specified in the *.yaml* file. Another launch file called *controller.launch* is created to attach the controllers to the simulated robot. This launch file first loads the *position_controller.yaml* file to the ROS parameter server. Then, the *spawner* executable from the *controller_manager* package is called upon with the name of the controllers from the configuration file as launch arguments. Lastly, the launch file calls upon the *robot_state_publisher* to convert joint states to transforms. This is relevant if, for example, *RViz* is going to be used. The *controller.launch* file is included at the bottom of the *gazebo.launch* file that was created earlier so that the controllers are launched together with the simulation. The reason for having the controllers in a separate launch file is to make debugging the package easier. Now, when launching the simulation, the leg joints are firm, and so the whole model will tip over when it touches the ground. There should therefore be published new setpoints to the controllers' command topics so that the robot has a more functional configuration. This can be easily done by running an *rqt* node with the *Message Publisher* plugin. The robot is shown in a standing position with the help of position controllers and the *Message Publisher* in Figure 7.4. The robot can perform simple movements with this setup.

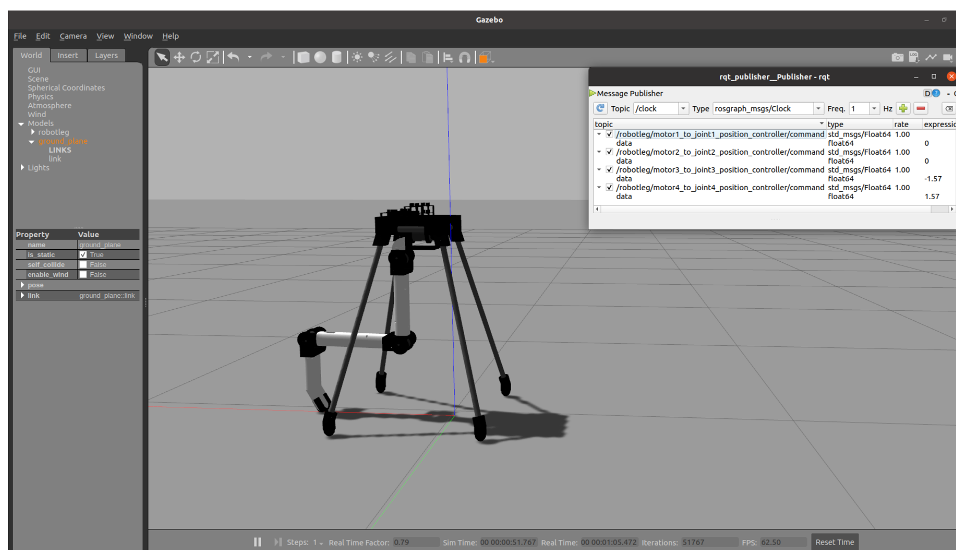


Figure 7.4: *Gazebo* simulation with position controllers

7.4.3 Motion planning and trajectory execution

MoveIt was implemented to have the robot perform more advanced movements enabling it to go forwards. By having *MoveIt* plan and execute a trajectory imitating a gait, the robot leg would, in theory, drag and push itself forward. For the robot to support trajectory execution, the controllers would first need to be shifted from joint position controllers to joint trajectory controllers. This was done by creating a new controller configuration file called *trajectory_controller.yaml* with the relevant parameters for a trajectory controller. The *controller.launch* file must then be altered to load this new file instead of *position_controller.yaml* to the ROS parameter server. The *spawner* must also be launched with the name of the trajectory controller as an argument instead of the position controllers. The joint state controller is still needed to publish joint states to the *joint_states* topic. The controllers will then function as action servers and wait for an action client to publish an input on the *goal* topic. This action client is created by *MoveIt*.

To use *MoveIt*, a *MoveIt* configuration package must be created in the workspace. This package can be configured and generated by launching the *MoveIt Setup Assistant* that comes with installing *MoveIt*. The setup assistant needs to know the location of the robot's URDF, which is in the *robotleg_description* package. It is then possible to let the setup assistant check for potential self-collisions, create planning groups with the four joints that are to be considered in motion plans, and define named robot states that are going to be used often. The last thing that needs to be configured for the robot leg is a *FollowJointTrajectory* controller, which can be automatically added for the planning groups that have been defined by simply clicking a button in the setup assistant. After adding meta information about the author of the package, the package can be generated. The name of the package follows a standard and is called *robotleg_moveit_config*.

The *MoveIt* package comes with a large collection of launch and configuration files. One of the generated launch files, called *demo.launch*, lets the user try out the motion planner on a dummy robot that does not let the environment restrict its movements. However, the package does not come with a launch file that connects the motion planner to a real or simulated robot. This launch file must be created manually and is given the name *robotleg_planning_execution.launch*. This launch file first runs a *joint_state_publisher* node that publishes the joint states of the robot to the *joint_states* topic so that they are available to other nodes in the system. Then, the launch file includes two other launch files called *move_group.launch* and *moveit_rviz.launch*, that were automatically generated when creating the package. *move_group.launch* initializes *move_group*, which is the primary *MoveIt* node. *moveit_rviz.launch* opens *RViz* with the *MotionPlanning* plugin for interacting with the *move_group* node. By launching *gazebo.launch* from the *robotleg_description* package followed by *robotleg_planning_execution.launch* from the *robotleg_moveit_config* package, the simulated robot can be manipulated with *MoveIt*.

When configuring the package through the setup assistant, there was an option to define named states for the robot. A state called *front_step* and a state named *back_step* was defined to act as the start and end configuration of a gait. By moving the robot leg to the *front_step* state and planning a trajectory to the *back_step* state, the robot leg would in theory perform a movement resembling a gait and push itself forward. The planned trajectory is visualized in *Figure 7.5*.

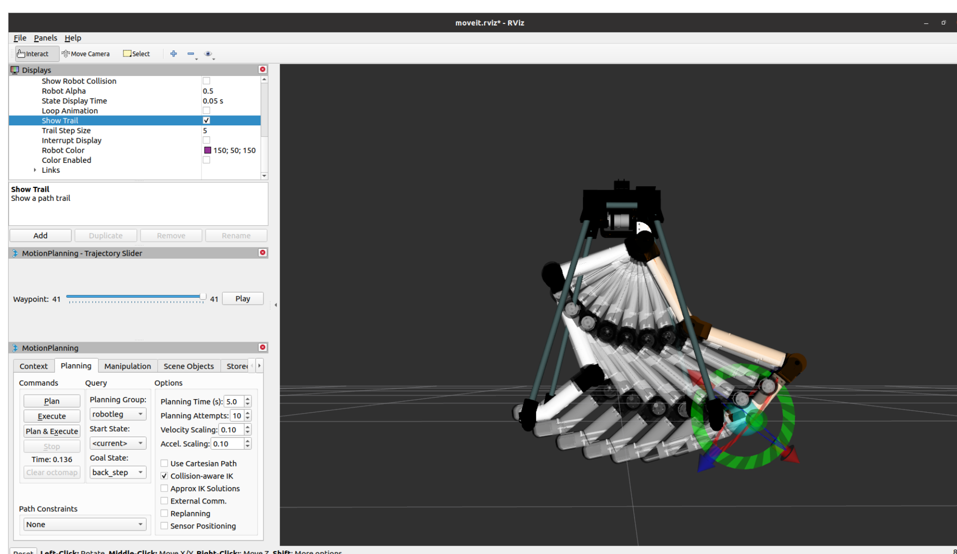


Figure 7.5: Planned trajectory generated by *MoveIt*

A problem was discovered with this plan. Part of the trajectory goes below the ground level referenced to the stand. Since the hip is fixed to the base of the stand, and the whole stand is rigid, the legs of the stand will be lifted off the ground as the robot leg pushes down during the execution of the trajectory. The result was that the model would always tip over. This brought to attention a flaw with the design, which is that

the height of the stand, thus the hip, is static. This is not the case in a natural gait, as the hip would shift slightly vertically during the motion. This is discussed in *subsection 3.4.8*. A more advanced path planning was needed to fix this problem. The solution was to use the functionality known as Cartesian planning. Cartesian planning ensures that the end effector moves in a straight line from start pose to end pose, thereby preventing the leg from lifting the stand off the ground. There are several Cartesian planners that have been developed by the global *MoveIt* community in collaboration with PickNik Robotics (Dave Coleman, Mark Moll and Andy Zelenak 2021). However, they are currently works in progress and can sometimes have trouble planning complete trajectories. The planned trajectory with Cartesian path enabled is visualized in *Figure 7.6*.

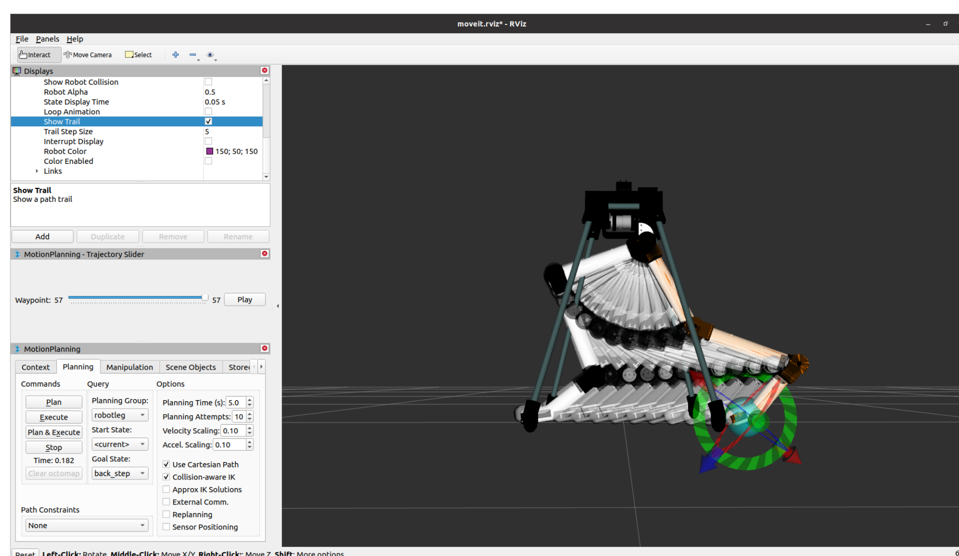


Figure 7.6: Planned trajectory with Cartesian path enabled

For the robot leg to move a considerable distance, the motion would have to be repeated several times. This is not intuitive through the *RViz* GUI. Therefore, an action interface to the *move_group* node is created with a Python script called *loop_gait.py* and the *moveit_commander* package. Many parts of the code is modified from the official *Move Group Python Interface* tutorial by Acorn Pooley and Mike Lautman 2021. As the name of the script implies, the robot leg is looped through a gait. The gait has four named poses as waypoints: *back_step*, *back_raised*, *front_raised* and *front_step*. The trajectories from *back_step* to *back_raised*, *back_raised* to *front_raised* and *front_raised* to *front_step* use regular planning, while the trajectory from *front_step* to *back_step* uses Cartesian planning. These trajectories are executed in sequence inside a while loop. The Cartesian path planning needs an exact start pose and end pose before computing a trajectory. Therefore, the script sends the robot leg to the *back_step* pose first so it can copy the pose to a variable called *end_step*. The gait is performed successfully most of the time, although the controllers sometimes have trouble executing the full Cartesian trajectory. The result is showcased in *Figure 7.7*, where the robot leg has

traveled from the vertical blue line by only running the script in *Code snippet E.2*.

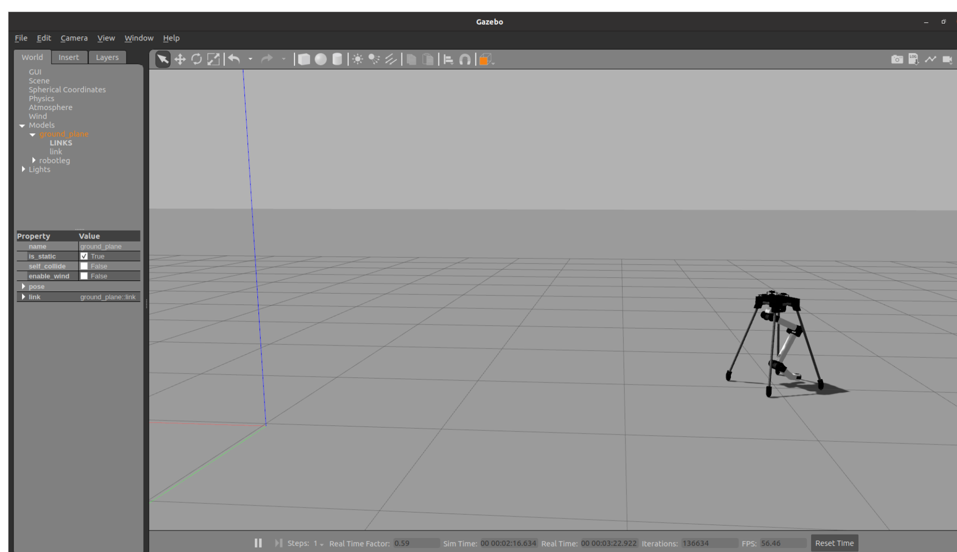


Figure 7.7: A distance traveled by only running loop_gait.py

7.4.4 Joint command interface

A joint command interface was implemented to handle communication between *MoveIt* and the embedded serial node. *MoveIt* expects communication with the *follow_joint_trajectory* action server to send the planned path. This action server was programmed using Python and the *rospy* library. Executing a trajectory is done by implementing an execute callback function that receives the planned trajectory and while subscribing to the actual joint positions, loops through it. When every joint is within a tolerance, the following point is sent. Since there is currently no support for effort measurement, only the velocity and position are being published to their respective topics and subscribed to by the serial node. The details of the Python script are shown *Code snippet E.1*.

Another crucial part of this joint command interface is to handle unexpected behavior properly. A timer is set to exit the execution callback function if the trajectory is not finished within a given timer. The success feedback is sent to *MoveIt* when the trajectory is stored in the action server. Properly navigating the trajectory is up to the embedded controllers and is monitored by the joint command interface. Dynamical startup positions were added, making it possible to start in any position, as long as the joints are within their respective bounds.

7.4.5 Hardware interface

A hardware interface was set up by initializing Arduino as a serial node using the *rosserial* library. Using Arduino as a serial node allows for communication between ROS and embedded sensors and actuators. *rosserial* connects the Arduino microcontroller to the master node allowing access to the topic like any other node in the ROS system.

The communication between the PC and the Arduino operates as regular serial communication, using a set baud rate. In this thesis, a baud rate of 115200 is chosen. This means that the serial port can handle 115200 bits per second, representing the amount of data for 114 float data types being handled per second. The *rosserial* library does not currently support action server functionality, only publishing and subscribing.

When launching the serial node, the different publishers and subscribers are initialized with their respective topics and expected data types. The data types used are included as standard messages in the *rosserial* library and are the same as what is used in the Python script mentioned using *rospy*. The subscriber nodes must also contain a callback function with a node handle to handle the incoming data and store it. The incoming data is stored in the list of actuator classes described in *subsection 6.4.3*. When communicating with the ROS master, a node handle for the current node is initialized. This node needs to synchronize with the master frequently enough to keep the same clock and sync information. In the serial node, this is done by calling *nh.spinOnce()*. The terminal command shown in *Code snippet 7.1* was used to reduce serial latency. Details on how the node is initialized, the publisher and subscriber nodes can be viewed in *Code snippet D.4*.

Code snippet 7.1: Low latency mode

```
1  $ setserial /dev/<tty_name> low_latency
```

The communication latency between the serial node running in the embedded system and the ROS master is low. As shown in *Figure 7.8*, the latency average was around 4.63 ms. However, this average was heavily inflated by one single bit. The latency for most of the data sent was between 0 and 0.01 ms after the low latency mode was enabled. The latency check probably failed before completing due to compunction dropout; this happened multiple times.

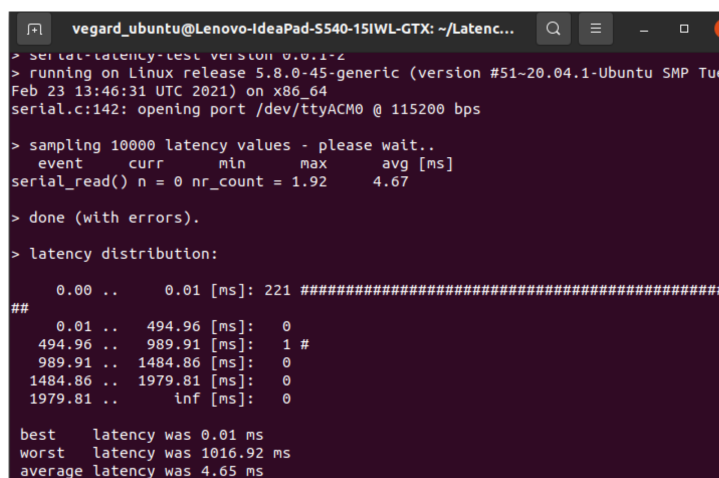


Figure 7.8: Latency test

7.4.6 GUI

Visualizing the state of the real robot is done in *RViz* when using *MoveIt*. The *joint_state* topic published by the serial node is transformed to display the robot state using the URDF model. For monitoring process values on a more detailed level, the *plotjuggler* package was used. *plotjuggler* allows the user to subscribe to all active topics and plot the values in different graphs. An example of how the *plotjuggler* window looks like while plotting the *joint_states* and *setpoint2arduino* topics are shown in *Figure 7.9*. The *RViz* window displaying the robot state based on the URDF model is also shown in *Figure 7.9*.

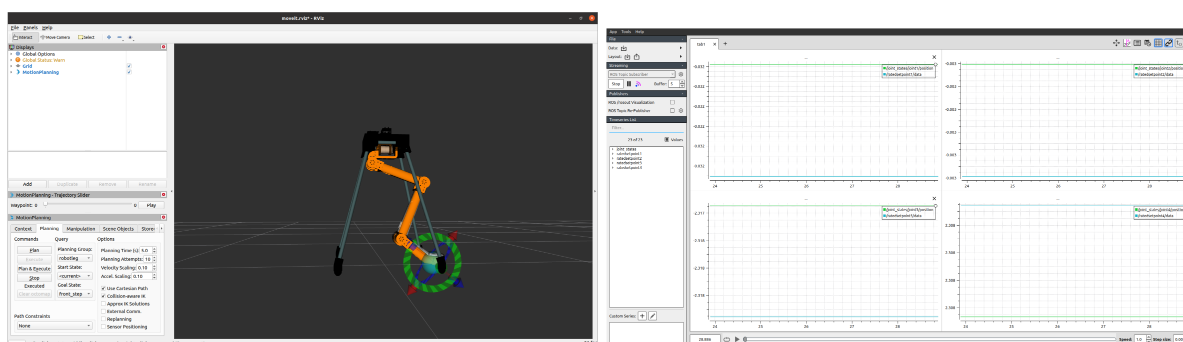


Figure 7.9: Displaying information using *RViz* and *plotjuggler*

7.4.7 Launching physical robot

The different launch files needed to execute a trajectory to the embedded actuators were combined into one file in a separate package. By launching *robotleg.launch* in the *robotleg_bringup* package, all communication is initialized. The content of this launch file and its including launch files can be found in the catkin workspace of the ROS section on the GitHub repository (Arnesen, Grinde, Hovland and Vestland 2021b). Visualizing correctly initialized communication is done by checking the *rqt_graph*. The correct *rqt_graph* and further instructions for properly initializing communication and executing a trajectory to the robot are shown in the user manual, *Appendix H*.

7.5 Analysis and Discussion

7.5.1 ROS version and distribution

The choice of which ROS distribution to use had to be made. As the team members had little previous experience with ROS, some research on the main differences between ROS and ROS 2 and their different distributions was done. At the start of this project, ROS 2 was fairly new, which has its advantages and disadvantages. Developing the project using ROS 2 would future proof the system as ROS 2 is the future of ROS. Another benefit of ROS 2 is that it would allow the team to use the more familiar operating system, Windows. However, the stable version of ROS 2 is recently published. This means that existing projects and documentation are sparse compared to ROS.

As the bachelor's thesis project period is approximately five months, the benefits of using ROS 2 did not outweigh the good documentation and examples available for ROS. If in the future development of this project, a change to ROS 2 is made, implementation of the *ros1_bridge* package will allow for communication between the two different ROS versions, making the transition smoother (Thomas et al. 2021). In the end, ROS Noetic, the newest ROS distribution and what is recommended on the ROS website, was chosen. Ubuntu 20.04 LTS is the recommended operating system running Noetic. ROS Noetic has a planned EOL, end of life, in May 2025, and combined with the long-time support version of Ubuntu, should allow for the same system to be used for years even though ROS 2 was not used.

7.5.2 Motion Planning

Most of the testing of motion planning with ROS was done with *MoveIt* through the *RViz* plugin interface. This interface was very user-friendly and therefore suitable for beginners of ROS. However, the interface is more fitting for a manipulator arm that requires delicate movements to perform a task with some type of tool on the end effector, for example a gripper that picks up and moves loose objects around. The end effector on the robot leg does not have a tool. It only has a rubber sole, which is used for dragging and pushing itself along the ground. The motions needed for the leg to fulfill its purpose of locomotion is largely dependent on the quantity of motion cycles and not as much on having a visualized environment to work in. Further work with the motion planning of the robot could therefore be done with the use of other interfaces. Examples of this being the C++-based move group interface or the Python-based move group interface, *MoveIt Commander*, which was barely explored in this project.

The planned trajectories in this project are all exclusively made in the walking direction of the robot, even though the robot has a hip joint that would allow for small movements to its sides. This joint could perhaps be used for turning the robot by planning and executing small movements perpendicular to the walking direction, a short distance either in front or behind its center of gravity. A difficulty with using a four DOF robot is that the orientation of the end effector can not be independent of all directions of translation. This would require at least six DOF. In the case of the robot leg, any movement to either of its sides will give the end effector a new orientation. An example of this is visualized from the front of the robot in *RViz* in *Figure 7.10*. This can cause difficulties when trying to plan in Cartesian space since it would require knowing the exact orientation of the end effector in the goal pose of the path. This issue is addressed by some Cartesian planners that have the property of being *underconstrained*. These do not require a pose to be fully specified. This means, for example, that the translation of an end effector can be set with an arbitrary orientation, or in the robot leg's case, whatever the orientation needs to be to support the translation. This functionality could be implemented in future work. A table showing

different Cartesian planners, and if they have the *underconstrained* property, is found on this blog post by Dave Coleman, Mark Moll and Andy Zelenak 2021.

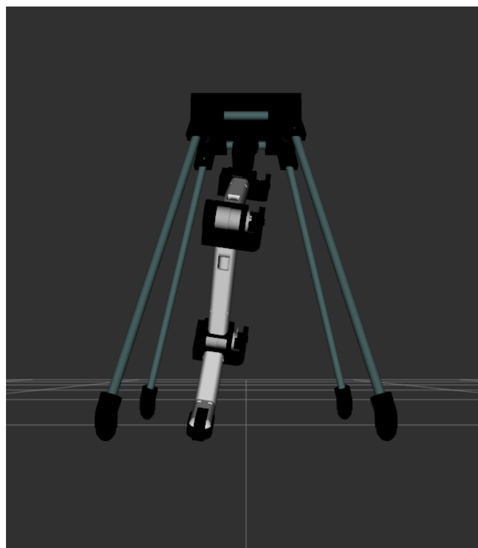


Figure 7.10: Example of the end effectors orientation being dependent on the hip joint

7.5.3 Controller implementation

Implementing control as a ROS node running on the computer would allow the controller algorithms to utilize more processor cores and faster processing speed, represented by the clock frequency of the CPU. Doing this would also free up processing power for other tasks in the already bottlenecked microcontrollers. Implementing control in this way would not make the software any more complex. The controller code running on the Arduino could be moved a layer up as a ROS C++ node. Instead of publishing setpoints from the joint command interface to the Arduino, the controller node would publish the desired actuating signal. This was not implemented because of stability issues with the *rosserial* communication, and communication dropouts would cause faulty control. Due to previous experience, implementing control in the Arduino made it possible to get more familiar with the other aspects of ROS. When the controller algorithm is placed close to the process, the current setpoints are kept during synchronization issues.

If state feedback controllers were implemented and the communication dropouts removed. The controller could be implemented by running Matlab and Simulink as their independent nodes and publishing the desired actuation signal. Implementing state feedback controllers on a microcontroller running C or C++ code could become quite substantial for a dynamic system like the robot leg developed in this project. They could require a lot of memory and computing power, especially when inverting matrices. However, the ROS toolbox for Matlab and Simulink could make this more manageable. There are also many features to aid controller tuning included in the different ROS packages. An example of this is an auto-tuner for the PID controller. A control manager could also be implemented. This would allow for easy swapping

between different controllers.

7.5.4 Hardware interface

Implementing a robust hardware interface to handle communication between the hardware layer and ROS is needed. The serial node used in this thesis can not operate as an action server and needs a joint command interface to handle communication with *MoveIt*. As mentioned, the joint command interface implemented in this thesis is simple and not suitable for more advanced operations. The implemented interface does not allow for re-planning as it stores the path planned from *MoveIt* and loops through it. If features like obstacle avoidance or re-planning based on joint feedback were added, this interface would need to be rewritten. Implementing re-planning was not done in this thesis because the objective was to implement a set gait, and synergistic features like obstacle avoidance were out of the scope due to lack of sensors. Suppose a different communication method was used between the PC and microcontrollers, e.g. if the microcontroller did not have support for direct integration with ROS as a node. An open-source boilerplate template for implementing a more robust hardware interface could be used with support for a control manager. The ROS Control Boilerplate supports trajectory execution from *MoveIt* and controller swapping using ROS Control (Coleman 2015).

7.5.5 Serial comunication

Communication between the Arduino boards and the PC using the *rosserial* library worked well in general. The problems encountered while using the Arduino serial communication based on the UART protocol were mainly communication dropouts. These dropouts were probably the cause of synchronization issues caused by communication with the ROS master by using *rosserial*. A deep dive into the UART protocol and communication with ROS is out of the scope of this thesis. In the serial node, the *nh.spinOnce()* command synchronized the node with the master. Synchronization issues can occur if this does not happen frequently enough.

Another problem encountered with the *rosserial* library is that the serial node only has support for publishing and subscribing. Ideally, the serial node would have support for an action server functionality. This would allow the serial node to communicate with *MoveIt* using the *follow_joint_trajectory* action directly. If faster communication was desired, a separate PCI-Express RS232 card could be installed on the computer. While using the Pololu motors and the Arduino Mega, the communication speed was not a limiting factor. The rate at which the Mega publishes and subscribes is not quick enough to warrant higher communication speed. If faster motors were used, a faster microcontroller and communication could be required for robust control.

7.5.6 Features for further work

Moving away from MoveIt

MoveIt is a great tool for easy control of robot manipulators and made testing trajectory execution on the robot leg in this thesis easier. However, for legged locomotion, a continuous gait often planned by solving an optimization problem as described in subsection 4.2.1 is preferable. Implementing trajectory execution based on a gait given in continuous time $x(t)$ and using a controller like the mentioned LQR would not require *MoveIt*. ROS integration with Matlab and Simulink could be implemented to support this new control strategy. Doing this requires an accurate dynamic and kinematic model and thus was not done in this thesis. This would allow for a more optimal gait based on the cost function, allowing for minimizing motor torque. Optimizing the trajectory for legged robots may be done by the use of ROS packages like *towr* (Alexander W Winkler, Bellicoso, Hutter and Buchli 2018). These trajectories can be visualized in *RViz* by plotting the planned gait and contact points with the use of ROS packages like *xpp* (Alexander W. Winkler 2017).

ROS Navigation and SLAM

MoveIt has integrated support for obstacle avoidance, which can be implemented by using SLAM to map the surroundings. Furthermore, the map made by the SLAM algorithm can be visualized in *RViz*, and taken into consideration by the trajectory being executed from *MoveIt*. Various SLAM algorithms exist in the form of ROS packages. For example, a laser-based slam algorithm is Open SLAM's *gmapping* (Gerkey 2019). Implementation of SLAM for the robot in this thesis would require a laser sensor, as discussed in *subsubsection 6.5.8*. For mobile robots, like the one in this thesis, ROS navigation may be implemented to allow for autonomous navigation using the map generated of the surroundings, odometry data, and a goal pose to generate a safe velocity command (Marder-Eppstein 2020). SLAM and navigation may be used together as the localization part of SLAM can help determine any errors from the odometry data caused by a faulty model or encoders.

Complete robot configurations

A quadruped model using four of the legs created in this thesis was made as a suggested application of the leg for further research. Using the same principles as previously discussed, a simulation in *Gazebo* with move groups in *MoveIt* were made, shown in *Figure 7.11*. These are also provided in the Git repository in the *catkin_ws_quad* directory. Implementation of the features discussed previously in this subsection may be advantageous for fully implementing locomotion with localization and environment mapping.

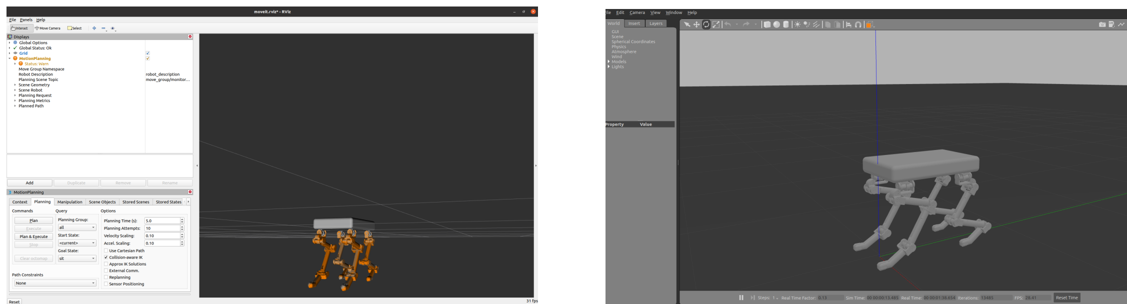


Figure 7.11: Quadruped robot simulation in *Gazebo* and move groups in *MoveIt*

7.6 Chapter Conclusion

The implementation of ROS in this project worked as intended. Rosserial communication was set up to communicate with the embedded microcontrollers, and a low latency meant that the communication speed was sufficient. Problems with communication dropouts and synchronization were worked around with the implementation of the joint command interface. The joint states published by the Arduino were reliably previewed in *RViz* using the URDF model. *MoveIt* allowed for motion planning, including Cartesian path planning, which was implemented for executing a functional gait. Simulation in *Gazebo* was successfully implemented with the use of an accurate robot model. How Matlab may be used with ROS was discussed and presented for further work.

Chapter 8

Results and Empirical Findings

8.1 Physical Model

The resulting physical model of this project consists of the robot leg and stand comprised mostly of 3D-printed parts and the embedded system of microcontrollers, motor drivers, actuators, and a wiring harness. As one of the more critical parts, the finished joint can be seen in *Figure 8.1*. The complete model was build according to the order of operations presented in the design chapter and can be seen in *Figure 8.2*.

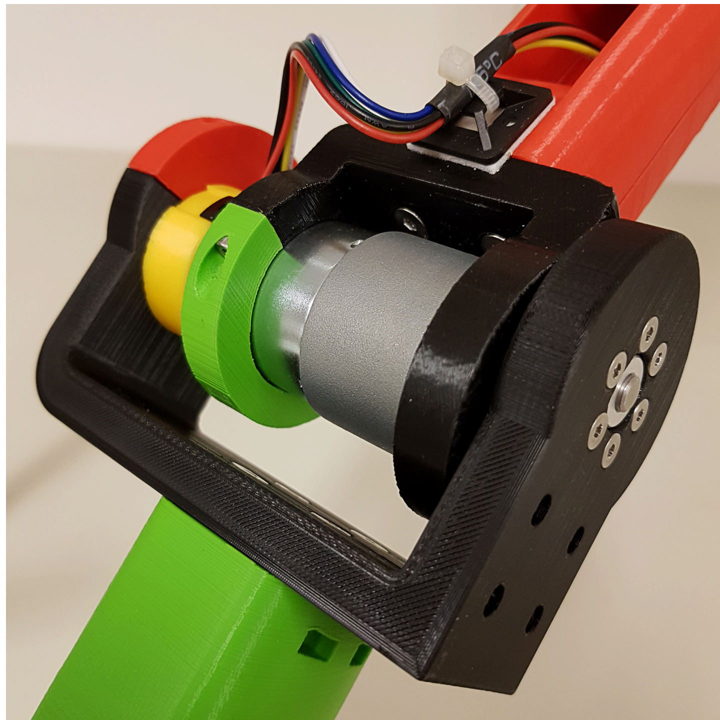


Figure 8.1: Physical model - Joint

Figure 8.2: Physical model - complete

8.2 Gait Execution

The executed gait is a result of the design, control strategy, and the embedded system. The controller, described in chapter 5, implemented as a low-level discrete controller in the embedded system, described in chapter 6, was tuned for the robot to follow a planned trajectory executed by ROS, described in chapter 7. The trajectory tracking had to be as smooth as possible due to restrictions posed by design and hardware. The result of the executed gait is presented in *Figure 8.3*. A graph made from data published to ROS topics containing setpoints, ramped setpoints, and actual joint states show how the controller tracks the trajectory. Angles published to ROS topics are given in radians. A video of the executed trajectories from early testing, before the motors got damaged, is also presented with the hand-in of the thesis.

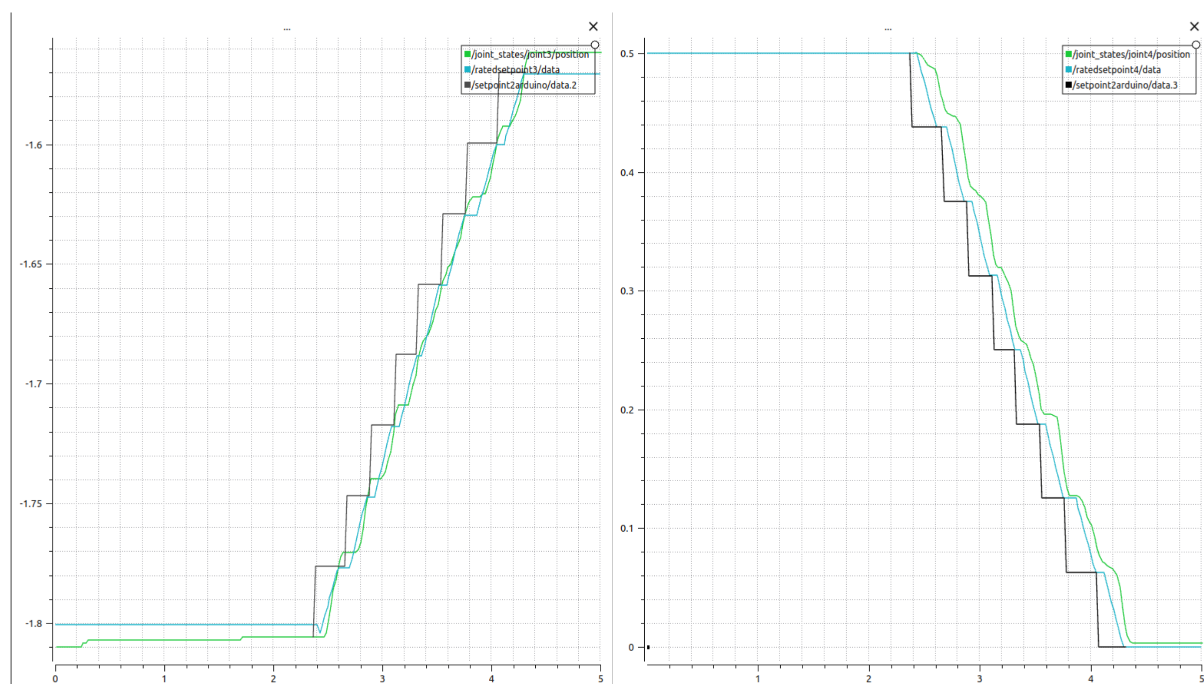


Figure 8.3: Joint 3 and 4 tracking trajectory

Chapter 9

Analysis and Discussion

The development of a robot from scratch is a comprehensive and interdisciplinary task. Finding optimal solutions in terms of control, design, and electronics involves several disciplines and usually requires more time and planning than afforded during a bachelor's thesis. Given the broad breadth of this project, some of the individual parts were probably not given enough consideration. This was mainly due to the project's time frame and the knowledge gaps of the people involved. As mentioned in the chapter on the design (*chapter 3*), involving people from other fields of study would perhaps close some of the gaps mentioned. This solution would free up time for the participants to work in their fields of expertise. As a result, more extensive research could have been performed on each separate part, and several of the proposed expansions could have been undertaken.

Budget constraints affected all parts of the project, as can be seen in the individual discussions for the different parts. This meant that all choices made had to take budgeting into account, resulting in less than optimal actuators and material choices. Another approach that could have mitigated some of the budgeting issues was to scale back the project's scope and focus more on the specific functionalities that would make up a complete robot. A project like this could involve creating one functioning joint with a more complete mathematical model that would allow a more optimal controller like the LQR. Only using one actuator would mean that a higher quality one could be used, as discussed in the embedded systems (*chapter 6*). Generally, a larger budget could enable the purchase of additional sensors, more powerful actuators and using sturdier materials than 3D-printed plastic parts. Adding more parts would have expanded the total scope of the project and required either more people or a down-scaled physical product.

9.1 Future Work

The expansions and improvements discussed for future work in the previous chapters are suggestions for the future development of the robot leg. Some of these are sugges-

tions, and some are mandatory improvements to achieve normal functionality. Mandatory improvements would be addressing the weak actuators, where two of them are currently permanently damaged. Motor upgrades, microcontroller upgrades, and additional features could require new sensors or a new controller algorithm. A decision about the suggested upgrades will have to be made by the future developers of this project.

Chapter 10

Conclusions

To sum up this project, some key conclusions will now be drawn. The design came out as intended, with all parts mating correctly. Some adjustments were needed, and material flex was greater than anticipated. The mathematical model created simulations showing that the kinematic calculations were correct. In the end, it was not implemented on the physical robot.

In the embedded system, all communication was set up successfully, and all parts of the electronics functioned adequately. The microcontrollers were pushed to their limits, so some optimization or an upgrade would be needed for future work. The control system is fully operational and able to track a smooth trajectory. However, stronger actuators are required for the hip joints as they suffered damage during testing and were rendered nonfunctional. To support further research, the strength of at least these two motors will have to be addressed. Other physical properties of the current design, i.e., gearing, will also need to be addressed to improve accuracy for research requiring high precision.

Simulations in ROS are operational and available for further research. Furthermore, the framework presented in chapter 4 and 7 are thoroughly explained and can be used for educational purposes.

In conclusion, the project statements have been solved, except for the problems introduced by the hip joint actuators. This is mainly due to budgeting and a lack of experience with torque sizing for gearing systems. Due to the limitations of the physical model, this thesis should be viewed as a pilot project for future work.

Bibliography

[1] Acorn Pooley and Mike Lautman. *move_group_python_interface_tutorial.py*. URL: https://github.com/ros-planning/moveit_tutorials/blob/master/doc/move_group_python_interface/scripts/move_group_python_interface_tutorial.py (visited on 11th May 2021).

[2] Affinity. *Designer*. URL: https://affinity.serif.com/en-gb/designer/ (visited on 2nd April 2021).

[3] Bjarne Foss andTor Aksel N. Heirung. *Merging Optimization and Control*. 1st ed. Springer, 2016. ISBN: 978-82-7842-200-7.

[4] Torleif Anstensrud. *Lecture series in TELE 3003, Industriell Automatisering - Robotteknikk*. 2020.

[5] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Multivariable Systemer, Forelesning 5, Modalregulering*. September 2020.

[6] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Multivariable Systemer, Forelesning 7, Tilstandsestimering*. September 2020.

[7] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Robotteknikk, Forelesning 3, Direktekinematikk*. October 2020.

[8] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Robotteknikk, Forelesning 4, Inverskinematikk*. November 2020.

[9] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Robotteknikk, Forelesning 6, Banegenerering*. November 2019.

[10] Torleif Anstensrud. *TELE 3003, Industriell Automatisering - Robotteknikk, Forelesning 6, Hastighetskinematikk*. November 2020.

[11] Arduino. *ARDUINO MEGA 2560 REV3*. URL: https://store.arduino.cc/arduino-mega-2560-rev3 (visited on 30th April 2021).

[12] Arduino. *ARDUINO NANO*. URL: https://store.arduino.cc/arduino-nano (visited on 30th April 2021).

[13] Arduino. *Arduino Software (IDE)*. URL: https://www.arduino.cc/en/Guide/Environment (visited on 30th April 2021).

[14] Arduino. *EEPROM Library*. URL: https://www.arduino.cc/en/Reference/EEPROM (visited on 30th April 2021).

[15] Arduino. *Wire Library*. URL: https://www.arduino.cc/en/reference/wire (visited on 30th April 2021).

[16] Henrik Moe Arnesen, Kristian Grinde, Vegard Hovland and Even Vestland. 'Development of Biomimetic Robot Leg With ROS Implementation - Preliminary Project'. 2021.

[17] Henrik Moe Arnesen, Kristian Grinde, Vegard Hovland and Even Vestland. *E2103 repository*. https://github.com/VegardHovland/E2103-Bachelor. 2021.

[18] AutoDesk. *Fusion 360*. URL: https://www.autodesk.com/products/fusion-360/overview (visited on 20th January 2021).

[19] Autodesk. *Parameter I/O*. URL: https://apps.autodesk.com/FUSION/en/Detail/Index?id=1801418194626000805 (visited on 30th March 2021).

[20] Michael Barr. *Embedded Systems Glossary*. URL: https://barrgroup.com/embedded-systems/glossary (visited on 20th April 2021).

[21] Boston Dynamics. *Spot*. URL: https://www.bostondynamics.com/spot (visited on 29th January 2021).

[22] Scott Campbell. *BASICS OF THE SPI COMMUNICATION PROTOCOL*. URL: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/ (visited on 12th May 2021).

[23] Sachin Chitta et al. 'ros_control: A generic and simple control framework for ROS'. In: *The Journal of Open Source Software* (2017). DOI: 10.21105/joss.00456. URL: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf.

[24] Dave Coleman. *ros_control_boilerplate*. https://github.com/PickNikRobotics/ros_control_boilerplate. 2015.

[25] Kevin Craig. *Optical Encoders*. URL: http://engineering.nyu.edu/mechatronics/Control_Lab/Criag/Craig_RPI/SenActinMecha/S&A_Optical_Encoders.pdf (visited on 2nd May 2021).

[26] Dave Coleman, Mark Moll and Andy Zelenak. *Guide to Cartesian Planners in MoveIt*. URL: https://picknik.ai/cartesian%20planners/moveit/motion%20planning/2021/01/07/guide-to-cartesian-planners-in-moveit.html (visited on 11th May 2021).

[27] DIY Fever. *DIY Layout Creator*. URL: http://diy-fever.com/software/diylc/ (visited on 30th April 2021).

[28] Brian Douglas. *State Space, Part 4: What Is LQR control?* February 2019. URL: https://www.youtube.com/watch?v=E_RDCFOlJx4&t=90s (visited on 6th May 2021).

[29] Rocco V. Dragone. *Choosing the Right Brake for Robotic Applications*. URL: https://www.techbriefs.com/component/content/article/tb/supplements/md/features/articles/28812 (visited on 20th April 2021).

[30] Drive Train Hub. *Gears App*. URL: https://gears.drivetrainhub.com/ (visited on 2nd May 2021).

[31] Michael Dwamena. *PLA vs ABS vs PETG vs Nylon – 3D Printer Filament Comparison*. URL: https://3dprinterly.com/pla-vs-abs-vs-petg-vs-nylon/ (visited on 2nd April 2021).

[32] Olav Egeland and Jan Tommy Gravdahl. *Modeling and Simulation for Automatic Control*. 1st ed. Tapir Trykkeri, Trondheim, Norway, 2002. ISBN: 82-92356-01-0.

[33] Espressif. *ESP32 WROOM 32E Datasheet*. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf (visited on 28th April 2021).

[34] Davide Faconti. *PlotJuggler*. https://github.com/facontidavide/PlotJuggler. 2021.

[35] Florian Fischer. *fusion2urdf*. https://github.com/SpaceMaster85/fusion2urdf. 2021.

[36] Leonid B Freidovich. 'CONTROL METHODS FOR ROBOTIC APPLICATIONS'. In: *UMEÅ University Lecture notes* 1.1 (2017), pp. 9–177. DOI: hhttp://acs.pollub.pl/pdf/v15n2/8.pdf.

[37] Mariss Freimanis. *Advantages & Disadvantages of Stepper motors & DC servo motors*. URL: https://www.machinetoolhelp.com/Automation/systemdesign/stepper_dcservo.html (visited on 24th January 2021).

[38] Brian Gerkey. *gmapping*. 2019. URL: http://wiki.ros.org/gmapping.

[39] Katarzyna Gospodarek. 'Determination of Relative Lengths of Bone Segments of the Domestic Cat's Limbs Based on the Digital Image Analysis'. In: *Applied Computer Science* 12.2 (2019), pp. 89–97. DOI: hhttp://acs.pollub.pl/pdf/v15n2/8.pdf.

[40] Kåre Halvorsen. *MX-Phoenix Hexapod*. URL: http://zentasrobots.com/mx-phoenix-hexapod/ (visited on 25th January 2021).

[41] Steve Heath. *Embedded Systems Design*. 2nd ed. Newnes, 2003. ISBN: 0-7506-5546-1.

[42] Milton Hildebrand. 'An analysis of body proportions in the Canidae'. In: *American Journal of Anatomy* 90.2 (1952), pp. 217–256. DOI: https://doi.org/10.1002/aja.1000900203. eprint: https://anatomypubs.onlinelibrary.wiley.com/doi/pdf/10.1002/aja.1000900203. URL: https://anatomypubs.onlinelibrary.wiley.com/doi/abs/10.1002/aja.1000900203.

[43] Austin Hughes and Bill Drury. *Electric Motors and Drives: Fundamentals, Types and Applications*. eng. Oxford: Elsevier Science & Technology, 2013. ISBN: 0080983324.

[44] Per Hveem and Kåre Bjørvik. *Reguleringsteknikk*. 1st ed. kybernetes forlag, 2014. ISBN: 978-82-92986-21-9.

[45] Ioan A. Sucan and Sachin Chitta. *Concepts*. URL: https://moveit.ros.org/documentation/concepts/ (visited on 8th May 2021).

[46] Ioan A. Sucan and Sachin Chitta. *MoveIt*. URL: https://moveit.ros.org/ (visited on 8th May 2021).

[47] Damir Jelaska. *Gears and Gear Drives*. John Wiley & Sons Ltd, 2012. ISBN: 9781119941309.

[48] B. Katz, J. D. Carlo and S. Kim. 'Mini Cheetah: A Platform for Pushing the Limits of Dynamic Quadruped Control'. In: *2019 International Conference on Robotics and Automation (ICRA)*. May 2019, pp. 6295–6301. DOI: 10.1109/ICRA.2019.8793865.

[49] Toshinori Kitamura. *fusion2urdf*. https://github.com/syuntoku14/fusion2urdf. 2021.

[50] Sai Krishna. *Jacobian*. URL: https://www.rosroboticslearning.com/jacobian (visited on 6th May 2021).

[51] Vijay Kumar. *Planar Robot Kinematics*. URL: https://www.seas.upenn.edu/~meam520/notes/planarkinematics.pdf (visited on 19th April 2021).

[52] Huibert kwakernaak. *Linear Optimal Control Systems*. 1st ed. Wiley-Interscience, 1972. ISBN: 978-0471511106.

[53] Joseph-Louis Lagrange. *Mécanique Analytique*. 1st ed. Chez la Veuve Desaint, 1788. ISBN: 978-2876470514.

[54] Eitan Marder-Eppstein. *Navigation*. 2020. URL: http://wiki.ros.org/navigation.

[55] MathWorks. *Matlab*. URL: https://se.mathworks.com/products/matlab.html (visited on 20th January 2021).

[56] MathWorks. *Robotics System Toolbox*. URL: https://mathworks.com/products/robotics.html (visited on 11th May 2021).

[57] James H. McClellan, Ronald W. Schafe and Mark A. Yoder. *Digital Signal Processing First*. 2nd ed. Pearson, 2018. ISBN: 9781292113876.

[58] Mean Well. *400W Single Output Switching Power Supply*. URL: https://www.elfadistrelec.no/Web/Downloads/_t/ds/ERPF-400-24_eng_tds.pdf (visited on 30th April 2021).

[59] Metcal. *MX-500P Power Supply*. URL: https://www.okinternational.com/hand-soldering-systems/id-MX-500P/MX-500P_Power_Supply (visited on 30th April 2021).

[60] Marek Michalkiewicz and Joerg Wunsch. *stdlib*. https://github.com/Patapom/Arduino/blob/master/Libraries/AVR%20Libc/avr-libc-2.0.0/include/stdlib.h. 2017.

[61] Microsoft. *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 30th April 2021).

[62] MIT. *Optimal Actuator Design*. MIT Biomimetic Robotics Lab. URL: https://biomimetics.mit.edu/research/optimal-actuator-design (visited on 24th January 2021).

[63] Ned Mohan, Tore M. Undeland and William P. Robbins. *Power Electronics: Converters, Applications, and Design*. 3rd ed. John Wiley & Sons Inc, 2002. ISBN: 9780471226932.

[64] Peter Moreton. *Industrial Brushless Servomotors*. eng. 1st ed. Newnes Power Engineering Series. Oxford: Elsevier Science & Technology, 1999. ISBN: 0750639318.

[65] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. 2nd ed. Springer, 2006. ISBN: 978-0387-30303-1.

[66] Open Robotics. *actionlib*. URL: http://wiki.ros.org/actionlib (visited on 20th April 2021).

[67] Open Robotics. *catkin*. URL: http://wiki.ros.org/catkin (visited on 10th May 2021).

[68] Open Robotics. *catkin/CMakeLists.txt*. URL: http://wiki.ros.org/catkin/CMakeLists.txt (visited on 9th May 2021).

[69] Open Robotics. *catkin/package.xml*. URL: http://wiki.ros.org/catkin/package.xml (visited on 9th May 2021).

[70] Open Robotics. *Gazebo*. URL: http://gazebosim.org/ (visited on 4th May 2021).

[71] Open Robotics. *joint_state_publisher*. URL: http://wiki.ros.org/joint_state_publisher (visited on 4th May 2021).

[72] Open Robotics. *Master*. URL: http://wiki.ros.org/Master (visited on 25th April 2021).

[73] Open Robotics. *Messages*. URL: http://wiki.ros.org/Messages (visited on 10th April 2021).

[74] Open Robotics. *Nodes*. URL: http://wiki.ros.org/Nodes (visited on 9th April 2021).

[75] Open Robotics. *Packages*. URL: http://wiki.ros.org/Packages (visited on 9th May 2021).

[76] Open Robotics. *Parameter Server*. URL: http://wiki.ros.org/Parameter%20Server (visited on 22nd April 2021).

[77] Open Robotics. *robot_state_publisher*. URL: http://wiki.ros.org/robot_state_publisher (visited on 4th May 2021).

[78] Open Robotics. *ROS Noetic*. URL: http://wiki.ros.org/noetic (visited on 10th May 2021).

[79] Open Robotics. *ROS Tutorials*. URL: http://wiki.ros.org/ROS/Tutorials (visited on 10th May 2021).

[80] Open Robotics. *rosbash*. URL: https://wiki.ros.org/rosbash (visited on 19th May 2021).

[81] Open Robotics. *roslaunch*. URL: http://wiki.ros.org/roslaunch (visited on 26th April 2021).

[82] Open Robotics. *rqt*. URL: https://wiki.ros.org/rqt (visited on 10th May 2021).

[83] Open Robotics. *rviz*. URL: http://wiki.ros.org/rviz (visited on 4th May 2021).

[84] Open Robotics. *Services*. URL: http://wiki.ros.org/Services (visited on 20th April 2021).

[85] Open Robotics. *tf2*. URL: http://wiki.ros.org/tf2 (visited on 4th May 2021).

[86] Open Robotics. *Topics*. URL: http://wiki.ros.org/Topics (visited on 20th April 2021).

[87] Open Robotics. *Tutorial: ROS Control*. URL: http://gazebosim.org/tutorials?tut=ros_control&cat=connect_ros (visited on 4th May 2021).

[88] PJRC. *Teensy 4.1 Development Board*. URL: https://www.pjrc.com/store/teensy41.html (visited on 28th April 2021).

[89] Pololu. *150:1 Metal Gearmotor 37Dx73L mm 24V with 64 CPR Encoder (Helical Pinion)*. URL: https://www.pololu.com/product/4697 (visited on 30th April 2021).

[90] Pololu. *Arduino library for the Pololu Dual G2 High Power Motor Driver Shields*. https://github.com/pololu/dual-g2-high-power-motor-shield. 2020.

[91] Pololu. *Pololu Dual G2 High-Power Motor Driver 24v14 Shield for Arduino*. URL: https://www.pololu.com/product/2516 (visited on 30th April 2021).

[92] Pololu. *Pololu Universal Aluminum Mounting Hub for 6mm Shaft, M3 Holes (2-Pack)*. URL: https://www.pololu.com/product/1999 (visited on 30th April 2021).

[93] Mike Purvis et al. *rosserial*. https://github.com/ros-drivers/rosserial. 2021.

[94] Raise3D. *Pro2 Plus Large Format 3D Printer*. URL: https://www.raise3d.com/pro2-plus/ (visited on 2nd April 2021).

[95] R. Rust et al. *COMPAS FAB: Robotic fabrication package for the COMPAS Framework*. https://github.com/compas-dev/compas_fab/. Gramazio Kohler Research, ETH Zürich. 2018. DOI: 10.5281/zenodo.3469478. URL: https://doi.org/10.5281/zenodo.3469478.

[96] Manuel F Silva and J.A Tenreiro Machado. 'A Historical Perspective of Legged Robots'. eng. In: *Journal of vibration and control* 13.9-10 (2007), pp. 1447–1486. ISSN: 1077-5463.

[97] Marion Sobotka. 'Hybrid Dynamical System Methods for Legged Robot Locomotion with Variable Ground Contact'. PhD thesis. Technische Universität München, 2007.

[98] Enrique Del Sol. *Backdrivability*. URL: https://enriquedelsol.com/2017/12/05/backdrivability/ (visited on 20th April 2021).

[99] Ian Sommerville. *Software Engineering*. 10th ed. Pearson, 2015. ISBN: 9781292096131.

[100] Mark W Spong, Seth Hutchinson and M Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: John Wiley & Sons, 2006.

[101] The Raspberry Pi Foundation. *Raspberry Pi 4 Tech Specs*. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (visited on 28th April 2021).

[102] Dirk Thomas et al. *ros1_bridge*. https://github.com/ros2/ros1_bridge. 2021.

[103] Ultimaker. *Ultimaker 2+*. URL: https://support.ultimaker.com/hc/en-us/sections/360003548499-Ultimaker-2- (visited on 2nd April 2021).

[104] Ultimaker. *Ultimaker Cura*. URL: https://ultimaker.com/software/ultimaker-cura (visited on 2nd April 2021).

[105] Stephen Umans. *Fitzgerald & Kingsley's Electric Machinery*. 7th ed. McGraw-Hill Education - Europe, 2013. ISBN: 9780073380469.

[106] Ellitor Williams. *Make: AVR Programming*. 1st ed. Maker Media, 2014. ISBN: 9781449355784.

[107] Alexander W Winkler, Dario C Bellicoso, Marco Hutter and Jonas Buchli. 'Gait and Trajectory Optimization for Legged Systems through Phase-based End-Effector Parameterization'. In: *IEEE Robotics and Automation Letters (RA-L)* 3 (May 2018), pp. 1560–1567. DOI: 10.1109/LRA.2018.2798285.

[108] Alexander W. Winkler. *Xpp - A collection of ROS packages for the visualization of legged robots*. 2017. DOI: 10.5281/zenodo.1037901. URL: https://doi.org/10.5281/zenodo.1037901.

[109] Kan Yoneda and Yusuke Ota. 'Non-Bio-Mimetic Walkers'. eng. In: *The International journal of robotics research* 22.3-4 (2003), pp. 241–249. ISSN: 0278-3649.

Appendix A

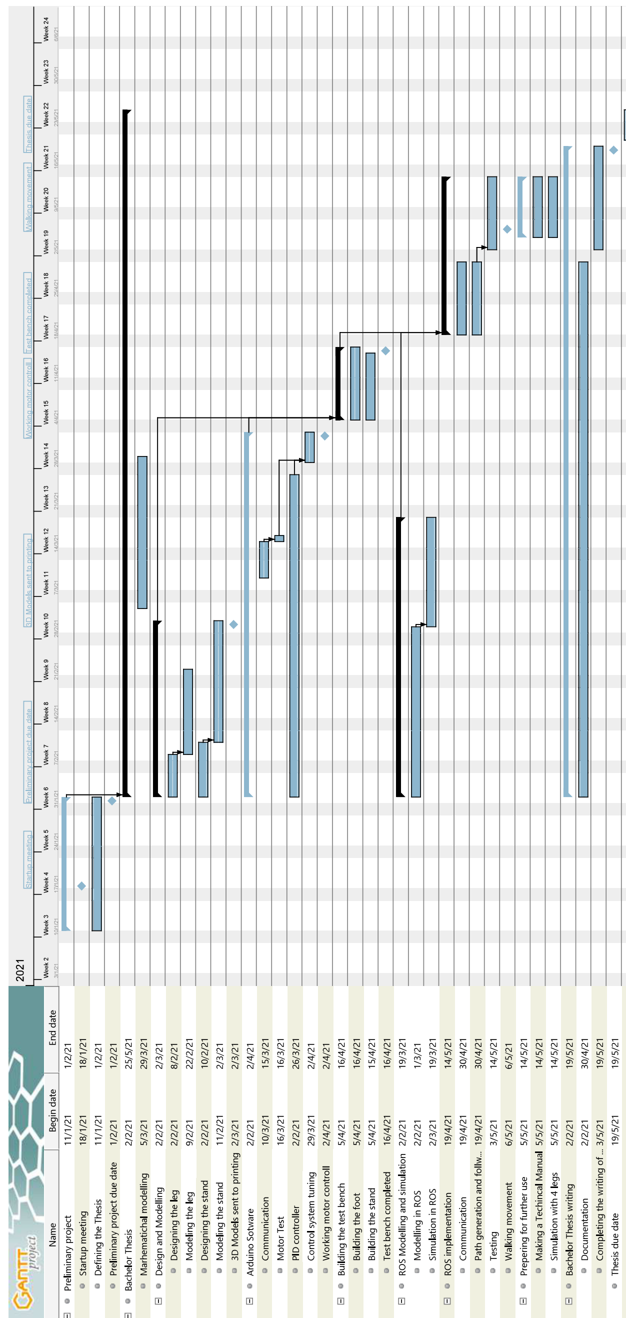
Budget and Record

Description	In	Out	Comment
Departmental financial support	5000kr		
Pololu DC motors		1387.5kr	4 pieces
Pololu motor drivers		867.5kr	2 pieces
Pololu mounting hubs		138kr	2 packs of 2
Pololu caster wheels		310kr	6 pieces
Shipping Pololu		477kr	
MVA Pololu		899kr	
Mean Well 400W power supply		782.5kr	
M4x40 screws		58.5kr	
Sum	5000kr	4920kr	

Table A.1: Bachelor thesis budget in NOK

Appendix B

Gantt



Appendix C

Parts List

Part name	Description	Quantity
Spacer		1
Link 1 - Hip		1
Link 2 - Femur side		2
Link 3 - Tibia side		2
Link 4 - Metatarsal		1
Link 4 - Phalanges		1
Link 4 - Rubber sole		1
Link lower		4
Link upper		4
Encoder end cap		4
Top plate		1
Weight distribution plate		1
Stand to foot connector		4
Stand to ball wheel connector		4
Ball wheel caster		4
Stand support leg	Aluminum pipe	4
Stand reinforcement pipe	Aluminum pipe	4
M2x5 dome head machine screw		8
M2x15 socket head screw		8
M3x10 countersunk screw		24
M3x13 socket head screw		20
M3x15 countersunk machine screw		24
M3 hexagonal nut		12
M4x19 countersunk machine screw		8
M4x20 dome head machine screw		12
M4x20 countersunk machine screw		20
M4x26 dome head machine screw		8
M4x27 dome head machine screw		8

M4x32 dome head machine screw	4
M4x33 dome head machine screw	4
M4x34 dome head machine screw	8
M4x35 countersunk machine screw	4
M4 hexagonal nut	76
Hexagonal spacer M3x10	4
Hexagonal spacer M2.5x6	8
Hexagonal spacer M2.5x20	8
Protoboard 20x14	4
Pin header female 1x15	8
Pin header angled male 1x3	4
Pin header angled male 1x5	4
Pin header angled male 1x15	4
Pin header angled male 1x17	4
Capacitor $470\ \mu$F 16 V	4
Schottky diode	4
Resistor $10\ \Omega$	4
Arduino Mega	1
Arduino Nano	4
Pololu 24 V 14 A motor driver	2
Pololu 24 V 1:150 gear motor	4
Pololu mounting hub	4

Appendix D

Arduino Code

D.1 Master

Code snippet D.1: actuator.h

```
//Header file for acuator class
#ifndef actuator_h
#define actuator_h
#include <Arduino.h>
#include <Wire.h>

class Actuator {
  private:
    byte slaveadress;   // Slave adress for a given actuator
    bool windup;        // windup indication flag
    int counter;        // Store counter from incremental encoder
    int gearRatio;      // The acuators gear ratio
    unsigned int amps;  // Store how much amps it is drawing
    float ang, prevAngle;// The acuators angle and previous angle
    float Kp, Ti, Td;   // Controller constants
    float ui = 0;       // Integrator part
    float ud = 0;       // Derivative part
    float elapsedTime;  // Scan time
    float error, lastError, cumError ; // Store last error
    float prevOut, output, setPoint;   // setpoints and outputs
    float velocity;
    float rateLimit;
    float setpointRated;        // Rate limited setpoint
    float velRef;               // Desiered velocity
    unsigned long currentTime;  // store current time
    unsigned long previousTime; // Keep track of the previousTime
```

```
27
28   public:
29     // Constructor function for actuator class
30     Actuator(byte encAddr, float p, float i, float d, int gr);
31
32     // Set functions
33     void setSetpoint(float r);          // Set setpoint
34     void setRatedSetpoint(float r);     // Set rate limited setpoint
35     void setParameters(float kp, float ti);  // Set PID parameters
36     void setDesieredVelocity(float lim);// Set desiered velocity
37     void setAmps(unsigned int amp);   // Set current readings in [mA]
38     void setSetpointRateLimit();      // Set rate limit for seetpoint
39
40     // Get functions
41     float getKp();                  // Get PID parameters
42     float getTi();
43     float getTd();
44     float getRatedSetPoint();       // Get the rated setpoint
45     float getAngle();               // Get angle
46     float getSetpoint();            // Get actuator setpoint
47     float getVelocity();            // Get current velocity
48     int getEffort();                // Get actuator speed / effort
49     unsigned int getAmps();         // Get current readings in [mA]
50
51     void readAngle();               // read and calculate the angle
52     void computePID();              // Computes the pid algorithm
53 };
54 #endif
```

Code snippet D.2: actuator.cpp

```
//This is the Cpp file for the actuator class
#include <Arduino.h>
#include "actuator.h"
#include <Wire.h>

//Constructor function for the class,
Actuator::Actuator(byte encAddr, float p, float i, float d, int gr)
    {
  slaveadress = encAddr;      // Store actuator encoder slave adress
  Kp = p;                     // Store pid parameters
  Ti = i;
  Td = d;
  gearRatio = gr;             // Store actuator gear ratio
}

//PID algorithm function
void Actuator::computePID() {
  currentTime = millis();             // Get current time
  elapsedTime = (float)(currentTime - previousTime); // Compute
     time elapsed
  rateLimit = abs(velRef / (elapsedTime) * 50); // calc ratelimit
  setSetpointRateLimit();            // Calc limited setpoint
  error = setpointRated - ang;       // Determine error

  if (!windup && Ti > 0.0) {         // intergrator term if not
     windup
    cumError += error * elapsedTime;
    ui =  cumError / Ti;             // Calc integrator term
  }

  if (Td > 0.0) {                    // calc derivative term if on
    ud = ((error - lastError) * (Kp * Td)) / elapsedTime;
  }

  float  out = Kp * error + ui + ud;      // PID output

  if ( out < 400 && out > -400) { // not windup if not within bounds
    windup = false;
  }
  if (out > 400) {                 // if  out of bounds, set windup
    out = 400;
```

```
39      windup = true;
40    }
41    if (out < -400) {
42      out = -400;
43      windup = true;
44    }
45    // Calculate angular velocity rad/s
46    velocity = ((ang - prevAngle) / 57.32 ) / (elapsedTime * 1000);
47
48    lastError = error;              // Remember last error
49    output = out;                   // Store output. MAX 400, MIN -400
50    prevAngle = ang;                // remember prev angle
51    previousTime = currentTime;     // Remember current time
52  }
53
54  //Reads encoder counter from slave and converts to angle
55  void Actuator::readAngle() {
56    Wire.beginTransmission(slaveadress); // Starts transmission
57    int available = Wire.requestFrom(slaveadress, (uint8_t)2); //
        Requests bytes
58
59    if (available == 2) {                 // Checks if 2 bytes avavible
60      counter = Wire.read() << 8 | Wire.read();  // int as combined
          upper and lower byte
61    }
62    else {                                // Error in transmission
63      Serial.print("Unexpected number of bytes received: ");
64      Serial.println(available);
65    }
66    int result = Wire.endTransmission();// End transmission, store
        result
67    if (result) {                        // check if sucessfulll
68      Serial.print("Unexpected endTransmission result: ");
69      Serial.println(result);
70    }
71    ang = (360.0 * (float)counter) / gearRatio;// Converts to
72  }
73
74  // Set function for setpoint
75  void Actuator::setSetpoint(float r) {
76    setPoint = r;                               //Updates setpoint
77  }
```

```
78 // Set function for rated setpoint
79 void Actuator::setRatedSetpoint(float r) {
80   setpointRated = r;                          //Updates rated setpoint
81 }
82 // Set function for parameters
83 void Actuator::setParameters(float p, float ki) {
84   Kp = p;                                     // Updates Kp
85   Ti = ki;                                    // Updates Ti
86 }
87 // Set function for current [mA]
88 void Actuator::setAmps(unsigned int amp) {
89   amps = amp;
90 }
91 // Set function for the desiered velocity
92 void Actuator::setDesieredVelocity(float vel) {
93   velRef = vel;
94 }
95 // Set function for Rate limit for setpoint change
96 void Actuator::setSetpointRateLimit() {
97   float diff = setPoint - setpointRated;
98   if (setPoint > setpointRated) { // If setpoint larger then rated
99     if (diff > rateLimit) {       // increase rated setpoint
100      setpointRated = setpointRated + rateLimit;
101    }
102    else {
103      setpointRated = setpointRated + diff;
104    }
105  }
106  if (setPoint < setpointRated) {  // If setpoint smaller then rated
107    diff = diff * -1.0;
108    if (diff > rateLimit) {        //decrease rated setpoint
109      setpointRated = setpointRated - rateLimit;
110    }
111    else {
112      setpointRated = setpointRated - diff;
113    }
114  }
115 }
116 // Get function for angle
117 float Actuator::getAngle() {
118   return ang;
119 }
```

```
120 // Get function for setoint
121 float Actuator::getSetpoint() {
122   return setPoint;
123 }
124 // Get function for rated setpoint
125 float Actuator::getRatedSetPoint() {
126   return setpointRated;
127 }
128 // Get function for current velocity
129 float Actuator::getVelocity() {
130   return velocity;
131 }
132 // Get functions for PID parameters
133 float Actuator::getKp() {
134   return Kp;
135 }
136 float Actuator::getTi() {
137   return Ti;
138 }
139 float Actuator::getTd() {
140   return Td;
141 }
142 //Get function for speed command / Effort
143 int Actuator::getEffort() {
144   return (int)output;       //Return as integer [-400,400]
145 }
146 //Get function for current [mA]
147 unsigned int Actuator::getAmps() {
148   return amps;
149 }
```

Code snippet D.3: variables.h

```
//Constants
const int numActuators = 4 ;      // Defines the number of actuators
const float startPos[4] = {0 , 0, -90, 40}; // Default Startposition
const char robot_id = "robotleg";// Robot namespace for topic
const char* joint_name[4] = {"joint1", "joint2", "joint3",
   "joint4"}; // Name of joints for topic


//Driver 1, for actuator 1 and 2
unsigned char M11nSLEEP = 8; // Common sleep pin for all 4 actuators
unsigned char M11DIR = 3;    // Directio pin for actuator 1
unsigned char M11PWM = 4;    // PWM pin for actuator 1, pwm fq=980
unsigned char M11nFAULT = 9; // Common fault pin fot all 4 acutators
unsigned char M11CS = A0;    // Pin for current sensing for
   actuator 1

unsigned char M12nSLEEP = 8; // Common sleep pin for all 4 actuators
unsigned char M12DIR = 5;    // Directio pin for actuator 2
unsigned char M12PWM = 13;   // PWM pin for actuator 2, pwm fq=980
unsigned char M12nFAULT = 9; // Common fault pin for all 4
unsigned char M12CS = A1;    // Pin for current sensing actuator 2

//Driver 2, for actuator 3 and 4
unsigned char M21nSLEEP = 8; // Common sleep pin for all 4 actuators
unsigned char M21DIR = 6;   // Directio pin for actuator 3
unsigned char M21PWM = 10;  // PWM pin for actuator 3, pwm fq=490
unsigned char M21nFAULT = 9;// Common fault pin fot all 4 acutators
unsigned char M21CS = A2;   // Pin for current sensing actuator 3

unsigned char M22nSLEEP = 8;// Common sleep pin for all 4 actuators
unsigned char M22DIR = 7;   // Directio pin for actuator 4
unsigned char M22PWM = 11;  // PWM pin for actuator 4, pwm fq=490
unsigned char M22nFAULT = 9;// Common fault pin fot all 4 acutators
unsigned char M22CS = A3;   // Pin for current sensing actuator 4
```

Code snippet D.4: main.ino

```
1 #include <Arduino.h>
2 #include <stdlib.h>
3 #include "actuator.h"
4 #include "DualG2HighPowerMotorShield.h"
5 #include "variables.h"
6 #include <ros.h>
7 #include <sensor_msgs/JointState.h>
8 #include <control_msgs/FollowJointTrajectoryAction.h>
9 #include <control_msgs/FollowJointTrajectoryActionGoal.h>
10 #include <control_msgs/FollowJointTrajectoryGoal.h>
11 #include <trajectory_msgs/JointTrajectoryPoint.h>
12 #include <trajectory_msgs/JointTrajectory.h>
13 #include <std_msgs/UInt16.h>
14 #include <std_msgs/Float32MultiArray.h>
15 #include <std_msgs/Float32.h>
16 #include <std_msgs/Float64.h>
17
18 // Create node handles for publishers
19 sensor_msgs::JointState robot_state;
20 std_msgs::UInt16 curr_reading1;
21 std_msgs::UInt16 curr_reading2;
22 std_msgs::UInt16 curr_reading3;
23 std_msgs::UInt16 curr_reading4;
24 std_msgs::Float32 rated_setpoint1;
25 std_msgs::Float32 rated_setpoint2;
26 std_msgs::Float32 rated_setpoint3;
27 std_msgs::Float32 rated_setpoint4;
28
29 //Create node handle for ros communication
30 ros::NodeHandle  nh;
31
32 // Init motordriverer1 as md1(motor driver 1)
33 DualG2HighPowerMotorShield24v14 md1(M11nSLEEP, M11DIR, M11PWM,
      M11nFAULT,  M11CS, M12nSLEEP,  M12DIR,  M12PWM,  M12nFAULT,
      M12CS);
34 // Init motordriverer2 as  md2(motor driver 2)
35 DualG2HighPowerMotorShield24v14 md2(M21nSLEEP, M21DIR, M21PWM,
      M21nFAULT,  M21CS, M22nSLEEP,  M22DIR,  M22PWM,  M22nFAULT,
      M22CS);
36
37 // Create object array of the 4 actuators
```

```
38 // slaveadress, pid parameters and gear ratio as parameters
39 Actuator actuators[4] = {Actuator(8, 80, 0, 0.005, 3200),
       Actuator(9, 30, 8, 0, 4480), Actuator(10, 20, 20, 0, 4480),
       Actuator(11, 15, 100 , 0, 4480)};
40
41 // Declare functions
42 void controllActuators(Actuator acts[]); // Comute PID on all joints
43 void stopIfFault();    // Disable the motordrivers if fault
44 void setupDrivers();   // Start the motor drivers
45 void setupRos();       // Initialize ros comunication
46 void rosPub();         // Publishes to topics
47 void legCb(const std_msgs::Float32MultiArray& leg); // pos cb
48 void velCb(const std_msgs::Float32MultiArray& vel); // velocity cb
49
50 // Declare ros publishers
51 // Robot state publisher on joint_state topic
52 ros::Publisher pubJoint("/joint_states", &robot_state);
53 // Publishers for current readings 1 - 4
54 ros::Publisher pubCurr1("/current_reading1", &curr_reading1);
55 ros::Publisher pubCurr2("/current_reading2", &curr_reading2);
56 ros::Publisher pubCurr3("/current_reading3", &curr_reading3);
57 ros::Publisher pubCurr4("/current_reading4", &curr_reading4);
58 // Publishers for rate limited setpoints 1 - 4
59 ros::Publisher pubSetPoint1("/ratedsetpoint1", &rated_setpoint1);
60 ros::Publisher pubSetPoint2("/ratedsetpoint2", &rated_setpoint2);
61 ros::Publisher pubSetPoint3("/ratedsetpoint3", &rated_setpoint3);
62 ros::Publisher pubSetPoint4("/ratedsetpoint4", &rated_setpoint4);
63
64 // Declare subscribers
65 // subscriber for planned setpoints
66 ros::Subscriber <std_msgs::Float32MultiArray>
       sub("setpoint2arduino", &legCb);
67 // subscriber for planned velocities
68 ros::Subscriber <std_msgs::Float32MultiArray>
       sub2("velocities2arduino", &velCb);
69
70 void setup() {
71   setupRos();         // initialize ros comunication
72   Wire.begin();       // Initialize wire
73   delay(10);
74   setupDrivers();     // Init drivers and startpositions
75   delay(10);
```

```
76 }
77
78 void loop() {
79   stopIfFault();
80   controllActuators(actuators); // Pid controll on all the acuators
81   rosPub();        // Publish joint states and current readings
82   nh.spinOnce();   // sync with ros, needs to happen frequently
83 }
84
85 //Setup function for roscomunication
86 void setupRos() {
87   nh.getHardware()->setBaud(115200);  // Set baud rate
88   nh.initNode();                      // Initialize serial node
89
90   // Set publishers
91   nh.advertise(pubJoint);
92   nh.advertise(pubCurr1);
93   nh.advertise(pubCurr2);
94   nh.advertise(pubCurr3);
95   nh.advertise(pubCurr4);
96   nh.advertise(pubSetPoint1);
97   nh.advertise(pubSetPoint2);
98   nh.advertise(pubSetPoint3);
99   nh.advertise(pubSetPoint4);
100
101   // Set subscribers
102   nh.subscribe(sub);
103   nh.subscribe(sub2);
104
105   nh.spinOnce();
106
107   // Fulfill sensor_msg/JointState msg
108   robot_state.name_length = 4;
109   robot_state.velocity_length = 4;
110   robot_state.position_length = 4;
111   robot_state.effort_length = 4;
112   robot_state.header.frame_id = "";
113   robot_state.name = joint_name;
114 }
115
116 //Controlls all the actuators
117 void controllActuators(Actuator actuators[]) {
```

```
118   nh.spinOnce();                              // sync to get setpoints
        from topic
119   for (int i = 0; i < numActuators; i++) { // Loops over the 4
        actuators
120     actuators[i].readAngle();               // Get the actuators
          angle
121     actuators[i].computePID();              // Computes output using
          PID
122     nh.spinOnce();
123     if ( i == 0) {                          // Motor 1 is driver 1 M1
124       md1.setM1Speed(actuators[i].getEffort());
125       actuators[i].setAmps(md1.getM1CurrentReading());
126       curr_reading1.data = actuators[i].getAmps();
127       rated_setpoint1.data = actuators[i].getRatedSetPoint();
128     }
129     else if ( i == 1) {                     // Motor 2 is driver 1 M2
130       md1.setM2Speed(actuators[i].getEffort());
131       actuators[i].setAmps(md1.getM2CurrentReading());
132       curr_reading2.data = actuators[i].getAmps();
133       rated_setpoint2.data = actuators[i].getRatedSetPoint();
134     }
135     else  if ( i == 2) {                    // Motor 3 is driver 2 M1
136       md2.setM1Speed(actuators[i].getEffort());
137       actuators[i].setAmps(md2.getM1CurrentReading());
138       curr_reading3.data = actuators[i].getAmps();
139       rated_setpoint3.data = actuators[i].getRatedSetPoint();
140     }
141     else if ( i == 3) {                     // Motor 4 is driver 2 M2
142       md2.setM2Speed(actuators[i].getEffort());
143       actuators[i].setAmps(md2.getM2CurrentReading());
144       curr_reading4.data = actuators[i].getAmps();
145       rated_setpoint4.data = actuators[i].getRatedSetPoint();
146     }
147   }
148 }
149
150
151 //Disable motordriver if fault
152 void stopIfFault() {
153   if (md1.getM1Fault() || md1.getM2Fault()) { // Checks fault on
        driver 1
154     md1.disableDrivers();                       // Disable driver 1
```

```
155      delay(1);
156      Serial.println("M fault");
157      while (1);                                      // Stop program
158    }
159    if (md2.getM2Fault() || md2.getM1Fault()) { // Checks fault on
          driver 2
160      md2.disableDrivers();                           // Disable driver 2
161      delay(1);
162      Serial.println("M fault");
163      while (1);                                      // Stop program
164    }
165  }
166
167  //Setup function for motor drivers
168  void setupDrivers() {
169    md1.init();                             // Init pinmodes driver 1
170    md1.calibrateCurrentOffsets();
171    md2.init();                             // Init pinmodes driver 2
172    md2.calibrateCurrentOffsets();
173    md1.enableDrivers();                    // Enable mosfet 1
174    md2.enableDrivers();                    // Enable mosfet 2
175    for (int i = 0; i < numActuators; i++) {
176      actuators[i].readAngle();             // Read startpos
177      delay(10);
178      actuators[i].setSetpoint(actuators[i].getAngle()); //Initialize
            start setpoints
179      actuators[i].setRatedSetpoint(actuators[i].getAngle());
180    }
181    delay(50);            // Enabeling drivers needs some time
182  }
183
184  // Function for publishing joint states on the joint_state topic
185  void rosPub() {
186    float pos[4];        // Expected size for topic
187    float vel[4];
188    float eff[4];
189
190    for (int i = 0; i < 4; i++) {    // Fulfill the arrays whith
          readings
191      pos[i] = (float)((actuators[i].getAngle()) / 57.32);// store
            angle in rad
192      vel[i] = actuators[i].getVelocity();    // velocity in rad/s
```

```
193     eff[i] = actuators[i].getEffort();      // output from PID
194   }
195   // Fulfill the sensor_msg/JointState msg
196   robot_state.header.stamp = nh.now();
197   robot_state.position = pos;
198   robot_state.velocity = vel;
199   robot_state.effort = eff;
200
201   pubJoint.publish( &robot_state);    // Publish joint states
202   pubCurr1.publish( &curr_reading1); // Publish current readings 1
        to 4
203   pubCurr2.publish( &curr_reading2);
204   pubCurr3.publish( &curr_reading3);
205   pubCurr4.publish( &curr_reading4);
206
207   pubSetPoint1.publish( &rated_setpoint1); // Publish rate limited
        setpoints
208   pubSetPoint2.publish( &rated_setpoint2);
209   pubSetPoint3.publish( &rated_setpoint3);
210   pubSetPoint4.publish( &rated_setpoint4);
211
212   nh.spinOnce();                            // Sync with ros
213 }
214 // calback for setponts
215 void legCb(const std_msgs::Float32MultiArray& leg) {
216   for (int i = 0; i < 4; i++) {
217     float rad = leg.data[i];    // read radians planned in moveit
218     float deg = rad  * 57.32;   // Convert to degrees (180/3.14)
219     actuators[i].setSetpoint(deg); // Get angles from publisher node
220   }
221 }
222 // calback for velocities
223 void velCb(const std_msgs::Float32MultiArray& vel) {
224   for (int i = 0; i < 4; i++) {
225     actuators[i].setDesieredVelocity(vel.data[i]);  // Set desiered
          velocity
226   }
227 }
```

D.2 Slave

Code snippet D.5: slave1.ino

```
#include <Wire.h>
#include <EEPROM.h>            // Liberary for EEPROM saving
#define slaveAddr 8
//Constants:
int pinA = 3;                  // Encoder pin for A puls
int pinB = 4;                  // Encoder pin for B puls
//Variables:
int counter;               // store the incremental encoders counter
int aState;                    // Store the state of the puls
int aLastState;                // Save last state of the puls
void setup() {
  pinMode (pinA, INPUT);       //Defines the input pins
  pinMode (pinB, INPUT);
  EEPROM.get(0, counter);      //Get last stored counter value
  Wire.begin(slaveAddr);
  Wire.onRequest(requestEvent);          // On request from master
     function
  aLastState = digitalRead(pinA);        // Reads outputA initial
     state
  attachInterrupt(digitalPinToInterrupt(2), saveToERPROM, FALLING);
}
void loop() {
  aState = digitalRead(pinA);              // State of puls A
  if (aState != aLastState) {              // Checks if a pulse has
     happened
    if (digitalRead(pinB) != aState) {    // checks if clockwise
       rotation
      counter ++;
    } else {
      counter --;
    }
  }
  aLastState = aState;                      // Saves previous state
}
void requestEvent() {
  uint8_t buffer[2];
  buffer[0] = counter >> 8;                // Store the int as 2 bytes
  buffer[1] = counter & 0xff;              // each byte is 8 bits
```

```
35    Wire.write(buffer, 2);                    // Respond with message of
         2 bytes
36  }
37  void saveToERPROM(){                        //ISR function for
       interupt pin
38    EEPROM.put(0, counter);
39    delay(1000);                              // Wait to die
40  }
```

Appendix E

Python Code

Code snippet E.1: command interface

```
%\begin{python}
#! /usr/bin/env python
import rospy
import actionlib
from std_msgs.msg import Int32
from sensor_msgs.msg import JointState
from control_msgs.msg import FollowJointTrajectoryGoal
from control_msgs.msg import FollowJointTrajectoryActionGoal
from std_msgs.msg import Float32MultiArray
import time
from trajectory_msgs.msg import JointTrajectoryPoint
from control_msgs.msg import (
    FollowJointTrajectoryAction,
    FollowJointTrajectoryFeedback,
    FollowJointTrajectoryResult,
    FollowJointTrajectoryGoal
)

class JointTrajectoryActionServer(object):
    def __init__(self, controller_name):
        rospy.init_node('robotleg_interface')        # init ros node
        self._action_ns = controller_name + '/follow_joint_trajectory'
           #set ns
        self._as =
           actionlib.SimpleActionServer(self._action_ns,FollowJointTrajectoryAction,ex
           = False)         # Settings for action server
        self._as.register_goal_callback(self.goalCB) # Register goal cb
           function
        self._action_name = rospy.get_name()
        self._as.start()                             # Start actionserver
        self._feedback = FollowJointTrajectoryFeedback
        self._result = FollowJointTrajectoryResult
        self.pos = Float32MultiArray()               # Poisiton being
           published
        self.speed = Float32MultiArray()             # velocity being
           published
        self.i=0                                     # Counter for
           viapoint looping
        self.timer = 0
```

```
33          self.pos.data = [0, 0, -1.57, 0]              # init pos for
               joint positions
34          self.jointStates = [0, 0, -1.57, 0]
35          self.jointVel = [0, 0, 0,0]                   # init vel for
               joint states
36          rospy.Subscriber('/joint_states', JointState, self.get_joints) #
               define joint_states subscriber
37          self.speed.data = [0.0, 0.0, 0.0, 0.0]        # init speed
38          self.pos.data = self.jointStates
39          self.speed.data = self.jointVel               # store positions
               from harware
40          self.pub = rospy.Publisher('/setpoint2arduino',
               Float32MultiArray, queue_size=1000)  # Publisher for
               setpoints to arduin
41          self.pub2 = rospy.Publisher('/velocities2arduino',
               Float32MultiArray, queue_size=1000)     # Publisher for
               velocities to aruino
42          rospy.loginfo('Successful init')
43          rospy.spin()                                  # Loop ros
               comunication
44
45  # Callback function for executing trajectory
46      def execute_cb(self, goal):
47          joint_names = goal.trajectory.joint_names    # Get joint names
48          trajectory_points = goal.trajectory.points   # get trajecotry
49          self.viapoints = trajectory_points           # store as viapoints
50          self._goal = self._as.set_succeeded()        # Sucessfully got
               viapoint
51          self.i = 0                                   # reset
52          self.timer = time.time()                     # start timer
53          while self.i < len(self.viapoints):          # Loop over all the
               viapoints
54              for j in range(4):  # loop over 4 joints and get curr
                   setpoints
55                  self.pos.data[j] = self.viapoints[self.i].positions[j]#
56                  self.speed.data[j] = self.viapoints[self.i].velocities[j]
57                  if self.i == (len(self.viapoints) - 1): # avoid stopping
                       to early
58                      self.speed.data[j] = self.viapoints[self.i
                           -1].velocities[j]
59              rospy.spin                          # Read new pos from
                   sensors
60              time.sleep(0.02)                    # sleep long enough to
                   get new joint_states
61              rospy.sleep                         # continue program
62              self.pub.publish(self.pos)          # Publish current
                   setpoints
63              self.pub2.publish(self.speed)       # Publish speeds
64              if self.tol():                      # Check if close enough
                   to setpoint
65                  self.i = self.i + 1             # loop to next setpoints
                       in path
66              if (time.time()- self.timer) > 10: # exit if not completed
                   within 10 sec
67                  break
68          self.speed.data = self.viapoints[self.i - 1].velocities # set
               final speed
69          self.i=0                # Reset counter for next trajecotry
```

```
            execution
70
71      def goalCB(self):          # Goal callback function not used
72          self._goal = self._as.accept_new_goal()      #  Accept next goal
73          # Do something with goal, not used
74
75      def get_joints(self, msg):
76          self.jointStates = msg.position
77          self.jointVel =   msg.velocity   # Get joint states
78
79      def tol(self):
80          tolerance = True
81          for indeks in range(4):
82           if abs(self.pos.data[indeks] - self.jointStates[indeks]) >
                0.035:
83              tolerance = False # if not within tolerance curently 2
                   degrees
84          return tolerance
85
86  if __name__ == '__main__': # Run class constructor on program start
87      JointTrajectoryActionServer('robotleg/robotleg_controller')
```

Code snippet E.2: loop_gait.py

```
#!/usr/bin/env python3

# Software License Agreement (BSD License)
#
# Copyright (c) 2013, SRI International
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
#  * Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
#  * Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
#    with the distribution.
#  * Neither the name of SRI International nor the names of its
#    contributors may be used to endorse or promote products derived
#    from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
# Author: Acorn Pooley, Mike Lautman
#
# Modified by Kristian Grinde

## To use the Python MoveIt interfaces, we will import the
    `moveit_commander`_ namespace.
## This namespace provides us with a `MoveGroupCommander`_ class, a
    `PlanningSceneInterface`_ class,
## and a `RobotCommander`_ class. More on these below. We also import
    `rospy`_ and some messages that we will use:

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi, tau, dist, fabs, cos
from std_msgs.msg import String

class MoveGroupPythonInterface(object):
  """MoveGroupPythonInterface"""
```

```
54    def __init__(self):
55      super(MoveGroupPythonInterface, self).__init__()
56
57      ## First initialize `moveit_commander`_ and a `rospy`_ node:
58      moveit_commander.roscpp_initialize(sys.argv)
59      rospy.init_node('move_group_python_interface', anonymous=True)
60
61      ## Instantiate a `RobotCommander`_ object. Provides information such
            as the robot's
62      ## kinematic model and the robot's current joint states
63      robot = moveit_commander.RobotCommander()
64
65      ## Instantiate a `PlanningSceneInterface`_ object.  This provides a
            remote interface
66      ## for getting, setting, and updating the robot's internal
            understanding of the
67      ## surrounding world:
68      scene = moveit_commander.PlanningSceneInterface()
69
70      ## Instantiate a `MoveGroupCommander`_ object.  This object is an
            interface
71      ## to a planning group (group of joints).  In this tutorial the
            group is the primary
72      ## arm joints in the robotleg robot, so we set the group's name to
            "robotleg".
73      ## If you are using a different robot, change this value to the name
            of your robot
74      ## arm planning group.
75      ## This interface can be used to plan and execute motions:
76      group_name = "robotleg"
77      move_group = moveit_commander.MoveGroupCommander(group_name)
78
79      ## Create a `DisplayTrajectory`_ ROS publisher which is used to
            display
80      ## trajectories in Rviz:
81      display_trajectory_publisher =
            rospy.Publisher('/move_group/display_planned_path',
82                                                             moveit_msgs.msg.DisplayTrajectory,
83                                                             queue_size=20)
84
85
86      ## Getting Basic Information
87      ## ^^^^^^^^^^^^^^^^^^^^^^^^^
88      # We can get the name of the reference frame for this robot:
89      planning_frame = move_group.get_planning_frame()
90      print("============ Planning frame: %s" % planning_frame)
91
92      # We can also print the name of the end-effector link for this group:
93      eef_link = move_group.get_end_effector_link()
94      print("============ End effector link: %s" % eef_link)
95
96      # We can get a list of all the groups in the robot:
97      group_names = robot.get_group_names()
98      print("============ Available Planning Groups:",
            robot.get_group_names())
99
100     # Sometimes for debugging it is useful to print the entire state of
            the
```

```
101        # robot:
102        print("============ Printing robot state")
103        print(robot.get_current_state())
104        print("")
105
106        # We can get a list of all the named joint targets:
107        print("============ Printing named targets")
108        print(move_group.get_named_targets())
109        print("")
110
111        # Misc variables
112        self.robot = robot
113        self.scene = scene
114        self.move_group = move_group
115        self.display_trajectory_publisher = display_trajectory_publisher
116        self.planning_frame = planning_frame
117        self.eef_link = eef_link
118        self.group_names = group_names
119
120    # Plan and execute a trajectory to a named group state
121    def go_to_named_target(self, target):
122
123        # Update the joint values of a group state to the joint target
124        joint_goal = self.move_group.get_current_joint_values()
125        joint_goal = self.move_group.get_named_target_values(target)
126
127        # Plan and execute to the joint target
128        self.move_group.go(joint_goal, wait=True)
129
130        # Make sure there is no residual movement
131        self.move_group.stop()
132
133    # Copy the start and end pose of the Cartesian path
134    def calibrate_cartesian_path(self):
135
136        # Copy the end pose of the Cartesian path to a variable
137        self.go_to_named_target('back_step')
138        end_step = self.move_group.get_current_pose().pose
139
140        # Follow the gait pattern to avoid collisions
141        self.go_to_named_target('back_raised')
142        self.go_to_named_target('front_raised')
143
144        # Copy the start pose of the Cartesian path to a variable
145        self.go_to_named_target('front_step')
146        start_step = self.move_group.get_current_pose().pose
147
148        # Make an array of the waypoints used to compute a Cartesian path
149        waypoints = []
150        waypoints.append(copy.deepcopy(start_step))
151        waypoints.append(copy.deepcopy(end_step))
152
153        (plan, fraction) = self.move_group.compute_cartesian_path(
154                                        waypoints,    # waypoints to follow
155                                        0.1,         # eef_step
156                                        0.0)          # jump_threshold
157
158        return plan, fraction
```

```
159
160
161     def execute_plan(self, plan):
162
163         ## Executing a Plan
164         ## ^^^^^^^^^^^^^^^^
165         ## Use execute if you would like the robot to follow
166         ## the plan that has already been computed:
167         self.move_group.execute(plan, wait=True)
168         self.move_group.stop()
169         ## **Note:** The robot's current joint state must be within some
               tolerance of the
170         ## first waypoint in the `RobotTrajectory`_ or ``execute()`` will
               fail
171
172
173 def main():
174     try:
175
176         # Begin the gait by setting up the moveit_commander ...
177         gait = MoveGroupPythonInterface()
178
179         # Start by planning the Cartiesian path and saving the plan to a
               variable
180         cartesian_plan, fraction = gait.calibrate_cartesian_path()
181
182         # Perform all the movements of the gait in sequence and repeat until
               interrupted
183         while True:
184
185             gait.execute_plan(cartesian_plan)
186
187             gait.go_to_named_target('back_raised')
188
189             gait.go_to_named_target('front_raised')
190
191             gait.go_to_named_target('front_step')
192
193     except rospy.ROSInterruptException:
194         return
195     except KeyboardInterrupt:
196         return
197
198 if __name__ == '__main__':
199     main()
```

Appendix F

Matlab Code

Code snippet F.1: calculation of $\boldsymbol{A}$-matrix

```
function [A] = calcA(d, theta, a, alpha)
%calcA creates the transfer matrix between two joints using Denavit-
%Hartenberg parameters

A = [cos(theta) -sin(theta)*cos(alpha)  sin(theta)*sin(alpha) a*cos(theta);
     sin(theta)  cos(theta)*cos(alpha) -cos(theta)*sin(alpha) a*sin(theta);
              0             sin(alpha)             cos(alpha)            d;
              0                      0                      0           1];
end
```

Code snippet F.2: Symbolic calculation of all $\boldsymbol{A}$-matrices

```
function [A] = symCalcA(dh)
%symCalcA calculates all transfer matrices between two joints for the
%entire robot configuration using Denavit-Hartenberg parameters. By
%declaring "A" symbolic, the function enables symbolic calculation but it
%can also be used numerically.

    n = length(dh(:,1)); %Number of links
    A = sym('A', [4 4 n]);

    for i = 1:1:n
        d = dh(i,1);
        theta = dh(i,2);
        a = dh(i,3);
        alpha = dh(i,4);
        A(:, :, i) = calcA(d, theta, a, alpha);
    end
end
```

Code snippet F.3: Symbolic calculation of all T-matrices

```
function [T] = symCalcT(dh)
%symCalcT calculates all T matrices from base link to end effector for any
%robot configuration. By declaring A symbolic, the function enables
%symbolic calculation but it can be used numerically

    A = symCalcA(dh);

    n = length(dh(:,1)); %number of links
    T = sym('T', [4 4 n]);

    T(:,:,1) = A(:,:,1);
    for i = 2:1:n
        T(:,:,i) = T(:,:,i-1)*A(:,:,i);
    end
end
```

Code snippet F.4: Symbolic calculation of J-matrix

```
function [J] = symCalcJ(dh)
%sumCalcJ calculates the Jacobi matrix for any given robot configuration
%   By defining the Denavit Hartenberg table with an extra row for
%   rotational or translational joints the Jacobi matrix can be calculated
%   directly from the DH-table

    n = length(dh(:,1)); %number of links
    Jv = sym('Jv', [3 n]);
    Jw = sym('Jw', [3 n]);

    T = symCalcT(dh);

    for i = 1:n
        type = dh(i,5);
        if lower(type) == "rot"
            if i == 1
                Jv(1:3, i) = cross([0;0;1],(T(1:3, 4, n) - [0;0;0]));
                Jw(1:3, i) = sym([0;0;1]);
            else
                Jv(1:3, i) = cross(T(1:3, 3, i-1),(T(1:3, 4, n) - (T(1:3, 4,
                    i-1))));
                Jw(1:3, i) = T(1:3, 3, i-1);
            end
        else
```

```
24            if i == 1
25                Jv(1:3, i) = [0;0;0];
26            else
27                Jv(1:3, i) = T(1:3, 3, i-1);
28            end
29            Jw(i,1:3) = 0;
30        end
31    end
32
33    J = [Jv;Jw];
34 end
```

Code snippet F.5: Inverse kinematics for the configuration using geometry

```
1  function [dh] = invKinCalc(x,y,z,phi,dh,pi)
2  %invKinCalc calculates robot angles given desired values for x, y, z, phi
3  %   This function returns a full Denavit-Hartenberg table for all momentary
4  %   values at the current stance.
5
6      if ~exist('pi','var')
7        pi = pi;
8      end
9
10     theta2 = dh(2,2) + atan2(x,-z);
11     dh(2,2) = quadCheck(theta2);
12
13     r = sqrt(x^2+z^2);
14     ym = y+dh(2,1)+dh(5,3)*sin(phi+dh(6,2))-dh(7,1)*sin(pi/2-phi);
15     rm = r-dh(2,3)-dh(5,3)*cos(phi+dh(6,2))-dh(7,1)*cos(pi/2-phi);
16
17     gamma = atan2(-ym/sqrt(rm^2+ym^2),-rm/sqrt(rm^2+ym^2));
18     theta3 = gamma -
           acos(-(rm^2+ym^2+dh(3,3)^2-dh(4,3)^2)/(2*dh(3,3)*sqrt(rm^2+ym^2)));
19     dh(3,2) = quadCheck(theta3);
20
21     theta4 =
           atan2((ym-dh(3,3)*sin(dh(3,2)))/dh(4,3),(rm-dh(3,3)*cos(dh(3,2)))/dh(4,3))-dh(3,2);
22     dh(4,2) = quadCheck(theta4);
23
24     theta5 = -(phi+dh(3,2)+dh(4,2)+dh(6,2));
25     dh(5,2) = quadCheck(theta5);
26 end
```

Code snippet F.6: Using the inverse kinematics function to return a matrix containing all angles for all via points

```
function [theta] = invKinViasCalc(xVias,yVias,zVias,phiVias,dh,pi)
%invKinViasCalc Creates viapoints using the inverse kinematics function
%   This function returns a matrix containing angles for each actuator in
%   each via point

    if ~exist('pi','var')
      pi = pi;
    end

    n = length(xVias);
    theta = sym('theta', [4 n]);
    for i = 1:n
        dhMom = invKinCalc(xVias(i),yVias(i),zVias(i),phiVias(i),dh,pi);
        theta(:,i) = dhMom(2:5,2);
    end
end
```

Code snippet F.7: Symbolic calculation of quintic polynomial between two angles

```
function [thetaFunc,velFunc,accFunc] = tradjCalcQuint(theta, time, vel, acc)
%pathCalc calculates the quintic trajectory between two points
%   Returns symbolic trajectory functions using a quintic function. This
%   function enables the setting of start/stop times and acceleration but
%   sets them to 0 when the values are missing.

    t = sym('t');

    if ~exist('vel','var')
     % Velocity parameters is not set, defaults to 0
      vel(1) = 0;
      vel(2) = 0;
    end
    if ~exist('acc','var')
     % Acceleration parameters is not set, defaults to 0
      acc(1) = 0;
      acc(2) = 0;
    end

    c=[theta(1) vel(1) acc(1) theta(2) vel(2) acc(2)]';
    b = [1 time(1) time(1)^2 time(1)^3 time(1)^4 time(1)^5; 0 1 2*time(1)
        3*time(1)^2 4*time(1)^3 5*time(1)^4; 0 0 2 6*time(1) 12*time(1)^2
```

```
        20*time(1)^3; 1 time(2) time(2)^2 time(2)^3 time(2)^4 time(2)^5; 0 1
        2*time(2) 3*time(2)^2 4*time(2)^3 5*time(2)^4; 0 0 2 6*time(2)
        12*time(2)^2 20*time(2)^3];
22      a = b\c;
23
24      thetaFunc = a(1) + a(2)*t + a(3)*t^2 + a(4)*t^3 + a(5)*t^4 + a(6)*t^5;
25      velFunc = a(2) + 2*a(3)*t + 3*a(4)*t^2 + 4*a(5)*t^3 + 5*a(6)*t^4;
26      accFunc = 2*a(3) + 6*a(4)*t + 12*a(5)*t^2 + 20*a(6)*t^3;
27  end
```

Code snippet F.8: Symbolic calculation of all polynomials for the configuration

```
1   function [thetaFuncs,velFuncs,accFuncs] = pathCalcTot(thetaVias,timeLimits)
2   %pathCalcTot calculatesthe total path functions for alle via points
3   %   Using the pathCalc function, this function returns the symbolic
4   %   functions with respect to t for angle, velocity and acceleration for
5   %   all trajectories between via points.
6
7       n = length(timeLimits);
8       m = length(thetaVias(:,1));
9
10      timeLine = zeros(length(timeLimits)+1);
11      timeLine(1) = 0;
12
13      thetaFuncs = sym('thetaFuncs', [m,n]);
14      velFuncs = sym('velFuncs', [m,n]);
15      accFuncs = sym('accFuncs', [m,n]);
16
17      for i = 1:length(timeLimits)
18          timeLine(i+1) = timeLine(i)+timeLimits(i);
19      end
20
21      for i=1:n           %Number of via points
22          for j = 1:m     %Number of actuators
23              [thetaFuncs(j,i), velFuncs(j,i), accFuncs(j,i)] =
                    pathCalc([thetaVias(j,i) thetaVias(j,i+1)], [timeLine(i)
                    timeLine(i+1)]);
24          end
25      end
26  end
```

Code snippet F.9: Converting the symbolic functions into arrays of angles for plotting

```
1   function [thetaDiscrete, timeLine] = pathDiscrete(thetaFuncs,timeLim, timeStep)
```

```
2  %pathDiscrete creates an array of angles for each function in thetaFuncs
3  %   This function takes the symbolic functions in thetaFuncs and creates
4  %   angles given the time step. It also returns a timeline of all
5  %   timestamps where angles were calculated.
6
7      numVias = length(timeLim);
8
9      time = cell(1,numVias);
10     time{1} = 0:timeStep:timeLim(1);
11     numVals = length(time{1});
12     for i = 2:numVias
13         time{i} = time{i-1}(end):timeStep:time{i-1}(end)+timeLim(i);
14         numVals = numVals + length(time{i});
15     end
16
17     thetaPath = cell(1,numVias);
18     for i = 1:numVias
19         numTimeSteps = length(time{i});
20         for j = 1:numTimeSteps
21             t = time{i}(j);
22             thetaPath{i}(:,j) = subs(thetaFuncs(:,i), t);
23         end
24     end
25
26     thetaDiscrete = thetaPath{1};
27     timeLine = time{1};
28     for i = 2:numVias
29         thetaDiscrete = [thetaDiscrete thetaPath{i}];
30         timeLine = [timeLine time{i}];
31     end
32 end
```

Code snippet F.10: Ploting function for the robot in its current configuration

```
1  function [] = plotRobot(dh, baseHeight)
2  %plotRobot Plotting robot in current configuration
3      hold on
4      grid on
5      xlim([-400 400]);
6      ylim([-400 400]);
7      zlim([-50 550]);
8      xlabel('X');
9      ylabel('Y');
```

```
10      zlabel('Z');
11
12      T = symCalcT(dh);
13      n = length(dh(:,1)); %number of links
14
15      for i=1:n
16          x(i) = T(1,4,i);
17          y(i) = T(2,4,i);
18          z(i) = T(3,4,i);
19
20          c = {'k' 'r' 'm' 'b' 'b' 'c' 'g' 'k'}; %Color for each joint and end
                effector
21          plot3(x(i), y(i), z(i)+baseHeight, 'color', c{i}, 'marker', 'o')
                %Justerer for aksene til DH og høyden til stativet
22          if i==2
23              plot3([0 0],[0 y(i)],[baseHeight baseHeight], 'k');
24              plot3([0 x(i)],[y(i) y(i)],[baseHeight z(i)+baseHeight], 'k');
25          elseif i>1
26              plot3([x(i-1) x(i)], [y(i-1) y(i)], [z(i-1)+baseHeight
                    z(i)+baseHeight], 'k')
27          end
28
29      end
30  view([100 10])
31  end
```

Code snippet F.11: Animated plotting of robot

```
1   function [] = plotAnimation(dh, baseHeight, thetaDiscrete, timeLine, fileName)
2   %plotAnimation creates and exports an animation of the robot
3   %  The thetaDiscrete matrix includes actuator angles for all time steps in
4   %  a complete gait.
5
6       figure()
7
8       %Preallocation
9       numFrames = length(timeLine);
10      frames = struct('cdata',cell(1,numFrames),'colormap',cell(1,numFrames));
11      dhMom = dh;
12      for i = 1:numFrames
13          clf;        %Clearing plot values
14          dhMom(2:5,2) =  thetaDiscrete(:,i);
15          plotRobot(dhMom, baseHeight);
```

```
16          frames(i) = getframe(gcf);
17      end
18
19      video = VideoWriter(fileName, 'MPEG-4');
20      video.FrameRate = 10;
21
22      open(video)
23      writeVideo(video,frames);
24      close(video)
25  end
```

Code snippet F.12: Determining the velocity direction if any for waypoints

```
1   function [velVias] = velViasCalc(thetaVias,velLim)
2   %velViasCalc generates velocity limitations for all waypoints
3   %   This function takes an absolute value for the velocity limit and adds
4   %   the correct sign for the direction of the joint. If the direction
5   %   changes the velocity is set to 0.
6
7       [numActuators, numVias] = size(thetaVias);
8       velVias = zeros(numActuators, numVias);
9
10      for i = 1:numActuators
11          for j = 2:numVias-1
12              if thetaVias(i,j-1) > thetaVias(i,j) && thetaVias(i,j) >
                    thetaVias(i,j+1)
13                  velVias(i,j) = -velLim;
14              elseif thetaVias(i,j-1) < thetaVias(i,j) && thetaVias(i,j) <
                    thetaVias(i,j+1)
15                  velVias(i,j) = velLim;
16              else
17                  velVias(i,j) = 0;
18              end
19          end
20      end
21  end
```

Appendix G

Inverse Kinematics

The complete trigonometric calculation of the forward and inverse kinematics for the robot leg used in this thesis can be found in this appendix. As noted in the chapter on the mathematical model, the kinematics is easier to solve splitting it up into two projections. The first projection is of the xz-plane and can be seen in *Figure G.1*.

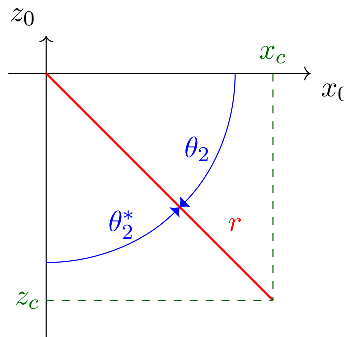


Figure G.1: Forward Kinematics - xz-projection

Using this projection both the direct kinematics for x and z can be found (*Eq G.1*) and the inverse kinematic function for θ_2 can be worked out (*Eq G.2*).

$$x = r * sin(\theta_2)$$
$$z = -r * cos(\theta_2) \tag{G.1}$$

$$\theta_2^{\star} = atan2(x_c, z_c)\theta_2 = \theta_2^* - \frac{\pi}{2} \tag{G.2}$$

In the angle expression for actuator 2, separate equations for θ_2^* and θ_2 is made. The θ_2^* is the actual actuator angle and θ_2 takes into consideration that the leg is pointing down in its natural state. This last angle is used for plotting only.

The other projection is in the yr-plane, where r is pulled from the xz-projection. This projection can be seen in *Figure G.2*.

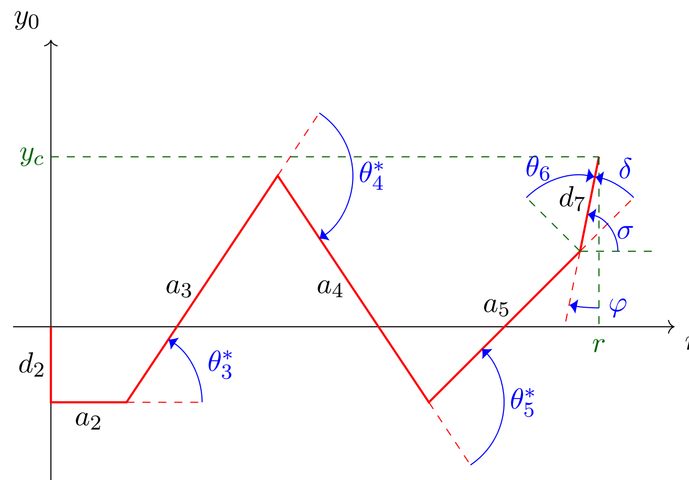


Figure G.2: Forward Kinematics - yr-projection

Some of the angles in the yr-projection need to be explained as they are not that intuitive.

- φ: A known angle between the ground and the robot end effector as defined in the gait analysis.
- δ: The angle of the end effector as defined in the physical model (used for the r length calculation).
- σ: The angle between the the r-axis and the end effector.

As can be seen in *Eq* G.3, using the known angles θ_6 and φ, an expression that includes the 3 unknown actuator angles can be worked out. This makes sure that there is three equations for the three unknown variables and that the equation set can be solved. Its worth noting that θ_6 is negatively defined and will have a negative value.

$$\begin{aligned}
\sigma = \pi - \frac{\pi}{2} - \varphi = \theta_3^* + \theta_4^* + \theta_5^* + \delta \\
\delta = \frac{\pi}{2} + \theta_6 \\
\varphi = -(\theta_3^* + \theta_4^* + \theta_5^* + \theta_6)
\end{aligned} \tag{G.3}$$

Using the fact that all joints connect to where the last link end, an equation can be formed that describes the y and r value of the end effector in the current pose. The

calculation for the y-value can be seen in *Eq* G.4.

$$\begin{aligned}
y_c &= -d_2 + a_3 sin(\theta_3^*) + a_4 sin(\theta_3^* + \theta_4^*) + a_5 sin(\theta_3^* + \theta_4^* + \theta_5^*) + d_7 sin(\theta_3^* + \theta_4^* + \theta_5^* + \delta) \\
y_c &= -d_2 + a_3 sin(\theta_3^*) + a_4 sin(\theta_3^* + \theta_4^*) + a_5 sin(-(\varphi + \theta_6)) + d_7 sin(\frac{\pi}{2} - \varphi) \\
y_c + d_2 + a_5 sin(\varphi + \theta_6) - d_7 sin(\frac{\pi}{2} - \varphi) &= a_3 sin(\theta_3^*) + a_4 sin(\theta_3^* + \theta_4^*) \\
&where \\
y_c' &= y_c + d_2 + a_5 sin(\varphi + \theta_6) - d_7 sin(\frac{\pi}{2} - \varphi)
\end{aligned} \tag{G.4}$$

To ease in calculating the inverse kinematics, all known values were placed in the y' variable and the final equation can be seen in *Eq* G.5.

$$y_c' - a_3 sin(\theta_3^*) = a_4 sin(\theta_3^* + \theta_4^*) \tag{G.5}$$

Using the same method for the r-value gives the equation in *Eq* G.6.

$$\begin{aligned}
r &= a_2 + a_3 cos(\theta_3^*) + a_4 cos(\theta_3^* + \theta_4^*) + a_5 cos(\theta_3^* + \theta_4^* + \theta_5^*) + d_7 cos(\theta_3^* + \theta_4^* + \theta_5^* + \delta) \\
r &= a_2 + a_3 cos(\theta_3^*) + a_4 cos(\theta_3^* + \theta_4^*) + a_5 cos(-(\varphi + \theta_6)) + d_7 cos(\frac{\pi}{2} - \varphi) \\
r - a_2 - a_5 cos(\varphi + \theta_6) - d_7 cos(\frac{\pi}{2} - \varphi) &= a_3 cos(\theta_3^*) + a_4 cos(\theta_3^* + \theta_4^*) \\
&where \\
r' &= r - a_2 - a_5 cos(\varphi + \theta_6) - d_7 cos(\frac{\pi}{2} - \varphi)
\end{aligned} \tag{G.6}$$

As with the y equation, all known values are collected in the r' variable and the final equation can be seen in *Eq* G.7.

$$r' - a_3 cos(\theta_3^*) = a_4 cos(\theta_3^* + \theta_4^*) \tag{G.7}$$

Taking both the equation for y' (*Eq* G.5) and r' (*Eq* G.7), an equation for θ_3 can be formed by combining them. Squaring both equations and adding them together in *Eq* G.8 results in an expression that can be used further.

$$\begin{aligned}
(y_c' - a_3 sin(\theta_3^*))^2 + (r' - a_3 cos(\theta_3^*))^2 &= (a_4 sin(\theta_3^* + \theta_4^*))^2 + (a_4 cos(\theta_3^* + \theta_4^*))^2 \\
y_c'^2 - 2a_3 y' sin(\theta_3) + a_3^2 sin^2(\theta_3) + r'^2 - 2a_3 r' cos(\theta_3) + a_3^2 cos^2(\theta_3) &= a_4^2 \\
-2a_3 r' cos(\theta_3) - 2a_3 y' sin(\theta_3) + r'^2 + y'^2 + a_3^2 - a_4^2 &= 0
\end{aligned} \tag{G.8}$$

This leaves us with an equation on the form $P cos\theta + Q sin\theta + R = 0$ which can be solved

like shown in *Eq* G.9.

$$\gamma = atan2\left(\frac{Q}{\sqrt{P^2+Q^2}}, \frac{P}{\sqrt{P^2+Q^2}}\right)$$
$$\gamma = atan2\left(\frac{-2a_3y'}{\sqrt{(-2a_3r')^2+(-2a_3y')^2}}, \frac{-2a_3r'}{\sqrt{(-2a_3r')^2+(-2a_3y')^2}}\right)$$
$$\gamma = atan2\left(\frac{-y'}{\sqrt{r'^2+y'^2}}, \frac{-r'}{\sqrt{r'^2+y'^2}}\right) \tag{G.9}$$

$$\theta_3 = \gamma \pm cos^{-1}\left(\frac{-R}{\sqrt{P^2+Q^2}}\right)$$
$$\theta_3 = \gamma \pm cos^{-1}\left(\frac{-(r'^2+y'^2+a_3^2-a_4^2)}{2a_3\sqrt{r'^2+y'^2}}\right)$$

The inverse kinematics equation for θ_3 gives two solutions. After simulating the solutions, the negative solution was found to be the correct for achieving the desired pose of the robot. solving *Eq* G.5 and *Eq* G.7 for the sin and cos expressions containing θ_4 gives:

$$sin(\theta_3+\theta_4) = \frac{y' - a_3 sin(\theta_3)}{a_4}$$
$$cos(\theta_3+\theta_4) = \frac{r' - a_3 cos(\theta_3)}{a_4}$$

This can be solved using inverse tangent in *Eq* G.10.

$$\theta_4 = atan2\left(\frac{y' - a_3 sin(\theta_3)}{a_4}, \frac{r' - a_3 cos(\theta_3}{a_4}\right) - \theta_3 \tag{G.10}$$

Finally, using *Eq* G.3, θ_5 can be found in *Eq* G.11

$$\theta_5 = -(\varphi + \theta_3 + \theta_4 + \theta_6) \tag{G.11}$$

Appendix H

User Manual

H.1 Introduction

This user manual provides a detailed description of how to simulate and control a 4 DOF robot leg using ROS with *MoveIt*. ROS Noetic and required packages specified in the equipment section should be installed. The joints of the robot leg should only operate within a natural gait as the motors currently equipped are not strong enough to handle the extensive gravitational load. The safety mount keeping the leg in place should not be removed before the Arduino boards are powered. Doing this would result in losing the current position, which would require a re-calibration of the start position. The emergency power button should be used if the leg does not operate as expected and control is lost. Do not turn on the power until everything is set up properly.

H.2 Equipment

Name	**Description**
MoveIt	Motion planning and manipulation
Robot leg	4 DOF Robot leg
rosbash	Needed for the rosrun command
ros_controllers	Support for necessary controllers
ROS Noetic full-desktop	ROS LTS distribution
PC with Ubuntu	Tested for Ubuntu 20.04
rosserial	Allows for serial communication with ROS
Plotjuggler	Plot information published on topics

H.3 Create Catkin workspace

Create a catkin workspace and clone repository.

Code snippet H.1: Create catkin workspace

```
$ mkdir -p ~/catkin_ws/
$ cd ~/catkin_ws/
$ git clone https://github.com/VegardHovland/src
$ catkin_make
```

Source the setup file, and confirm correct package path.

Code snippet H.2: Source setup file

```
$ source devel/setup.bash
$ echo $ROS_PACKAGE_PATH
```

H.4 Running simulation

To run a simulation of the robot leg in *Gazebo*, a slight modification needs to be made in one of the launch files. In *robotleg_planning_execution.launch*, in the *robotleg_moveit_config* package, the call for the *joint_state_publisher* node needs to be uncommented. This needs to be commented again if using the real robot.

Start by making sure the workspace is sourced and launch the *gazebo.launch* file.

Code snippet H.3: Launch Gazebo simulation

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roslaunch robotleg_description gazebo.launch
```

Wait for the simulation window to load in. Unpause the simulation and let the robot fall to the ground. The model will be reset after giving the leg a better configuration. Now open a new terminal and make sure to source the workspace before launching *MoveIt*.

Code snippet H.4: Launch MoveIt

```
$ cd ~/catkin_ws
$ source devel/setup.bash
$ roslaunch robotleg_moveit_config robotleg_planning_execution.launch
```

Use the motion planning panel to plan and execute any named goal state except *zeros*. Then, expand *Edit* in the top left corner of the *Gazebo* window and press *Reset Model Poses*. The robot should now be upright and the simulation is free to be explored at own will.

To perform the step motion, send the robot leg to the *front_step* state. Check the box called *Use Cartesian Path*, and plan with *back_step* as the goal state. If the planning or execution of this trajectory for some reason does not work, try setting the start state as *front_step* instead of <*current*>. If this also does not work, plan and execute a new trajectory to *front_step* and try again.

To have the robot loop a walking motion, open a new terminal and run the *loop_gait.py* executable script after sourcing the workspace. Make sure to have the robot in the *back_step* state first, as the script will start by moving the leg to this state, which might tip the robot depending on the start state.

Code snippet H.5: Loop gait

```
1 $ cd ~/catkin_ws
2 $ source devel/setup.bash
3 $ rosrun robotleg_moveit_config loop_gait.py
```

The loop can be canceled by pressing Ctrl+Z in the terminal where the script is running.

H.5 Executing on real robot

The first step to execute a planned gait on the robot leg is to ensure that the USB is plugged into the Arduino mega and the PC. To validate the connection, ensure that all five Arduino boards have their light indicators running. When the boards are correctly powered, the security strap can be removed, and the leg safely set to a given start position. The start position must satisfy the upper and lower bounds set by the individual joints. *Figure H.2* shows two examples of good start positions. As long as the Arduino boards are powered, the encoders will store their value for any given start position to be initialized. Starting serial communication, launching *MoveIt*, and initializing communication with the hardware interface is done by launching the *robotleg.launch* file.

Code snippet H.6: Launch robotleg.launch

```
1 $ roslaunch robotleg_bringup robotleg.launch
```

The correct starting position is confirmed by checking the robot state in *RViz*. When the launch program has successfully launched, validating successful communication is done by checking the *rqt_graph*.

Code snippet H.7: *rqt_graph*

```
1 $ rosrun rqt_graph rqt_graph
```

The rqt graph should look like the one in *Figure H.1* if set up correctly.

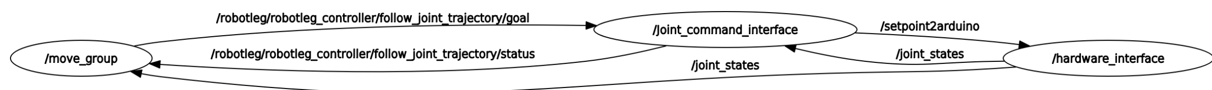


Figure H.1: *rqt_graph*

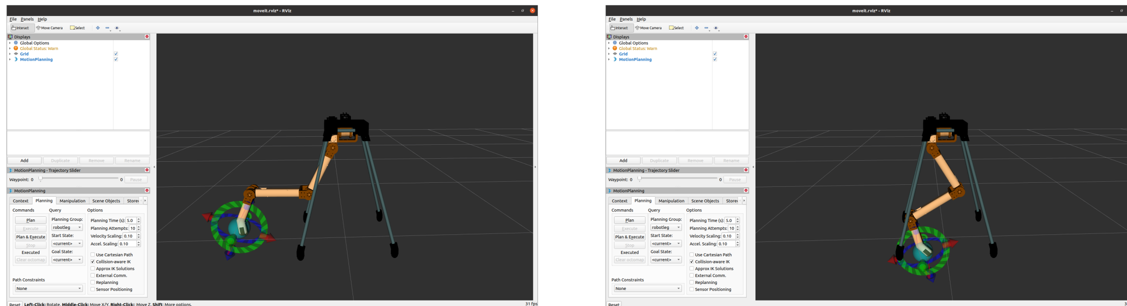


Figure H.2: Suggested start positions

H.5.1 Plot data from topics

In addition to the robot state displayed in *RViz*, plotting the data sent to the different topics can be done to display data. This is done by a ROS package called *plotjuggler*. How to install *plotjuggler* is shown on their GitHub repository, Faconti 2021. Running *plotjuggler* is done by running *Code snippet H.8* in a new terminal window.

Code snippet H.8: Run *plotjuggler*

```
1  $ rosrun plotJuggler plotJuggler
```

Loading the layout file in the source folder will subscribe to the *joint state* and *setpoint2arduino* topics. The response of the system can be monitored when executing a trajectory. How the *plotjuggler* window should look is shown in *Figure H.3*.

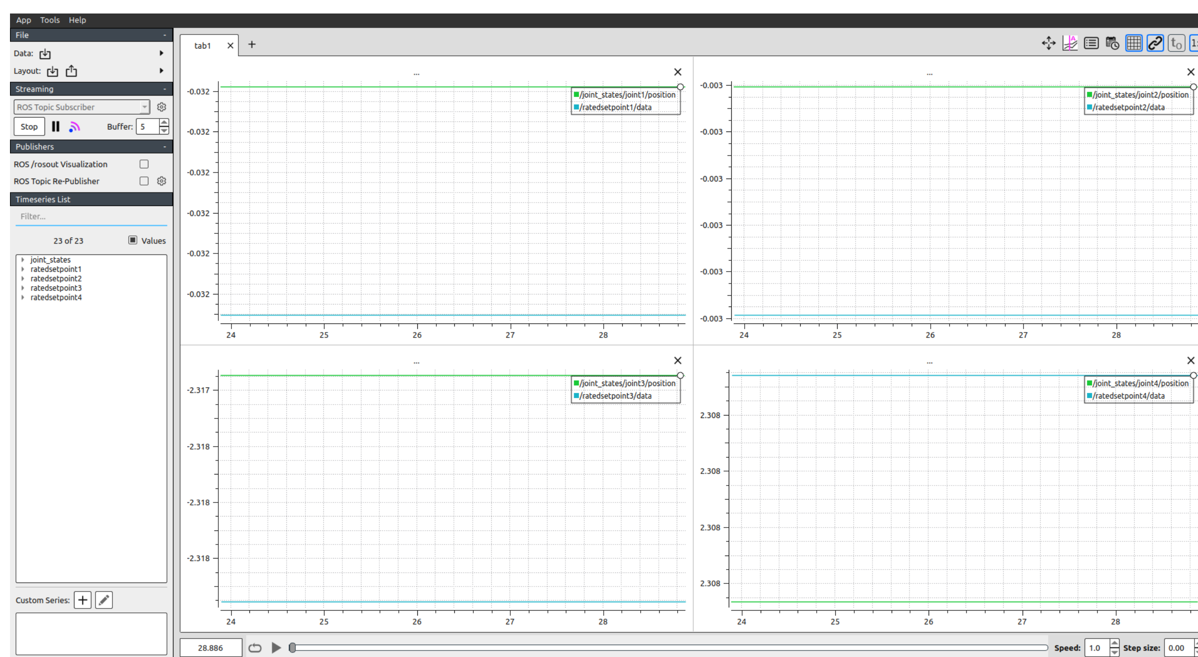


Figure H.3: *Plotjuggler* layout

H.5.2 Executing trajectory

Before turning on the power, unplug and plug in the Arduino to set its new starting position. Validate that the position is kept in *RViz*. The power button for motor supply voltage can now be turned on. Nothing should happen. Executing a trajectory is done using *MoveIt*. Pre-set positions are already initialized. Start by selecting the "middle step" position and pressing "plan" to verify that a path is available from the start position. If a path is available, execute the trajectory by pressing "plan and execute". The robot leg should now move to the middle step position. Repeat this step for the "front step" position. When the leg is in the "front step" position, select Cartesian planning. This will make the robot go in a linear path. Then select goal position to "back step" and verify that a plan is available. Execute the step by pressing "plan and execute". Running this gait in a loop is done by executing a script in a separate terminal, *Code snippet H.9*.

Code snippet H.9: Run gait loop

```
1  $ rosrun robotleg_moveit_config loop_gait.py
```

Development of Biomimetic Robot Leg With ROS Implementation

Henrik Moe Arnesen, **Kristian Grinde**, **Vegard Hovland** and **Even Vestland**
Department of Engineering Cybernetics, Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

Motivation

Four-legged robots have in recent years become popular as a platform for future use among humans. Common for many of them is a configuration with two degrees of freedom in walking direction and one degree of freedom perpendicular to this to make the robot more agile. The hypothesis that inspires this project is the following:

"If adding another degree of freedom, and making the robot more anatomically correct compared to quadrupedal animals, can give the robot increased agility and mobility?"

Although the research itself is beyond this project's scope, the goals were to develop a test bench for a biomimetic robot. This way, different gaits could be executed and analyzed. In addition, the robot could also serve as an educational example in various engineering disciplines.

The project culminated in the bachelor's thesis for the group in electrical engineering with a specialization in Automation [1].

Design

The goal of the design was to make a biomimetic robot leg modeled after feline proportions.

- An anatomical analysis was undertaken to find correct proportions for the feline leg.
- Using mostly 3D printed parts, the individual links and joints were connected with screws to form the leg.
- Hollow links allow hiding the cables.
- Having the leg mounted to a wheeled stand would enable a normal gate to move the robot.
- The stand consists of a 3D-printed top plate connected to four caster wheels via metal pipes. The top plate doubles as a mounting plate for the electronics and wiring.
- PLA was used for most 3D-printed parts except top plate made in PETG for strength.
- An URDF description of the robot with one and four legs was created for later use with ROS and Matlab.

Embedded System

The embedded system powering the robot consists of four actuators with built-in incremental encoders, two dual motor drivers, an Arduino Mega, and four Arduino Nanos.

- Motors used was 24V brushed DC gear motors with a gear ratio of 1:150 to improve torque and 68PPM encoders.
- To read the incremental encoders, four Arduino Nanos transmits the pulse counts to the master controller.
- An Arduino Mega concerts the pulse counts to angles and acts as a communication hub between ROS and the motor drivers.
- The Mega also powers the slaves and encoders, with a power buffer so that the Nanos has time to store values to EEPROM.
- Communication between the Arduino Mega and Nanos is done via the I2C protocol.
- Each individual actuator is controlled via independent joint PID controllers with anti-windup in the Mega.

Results and Future Work

The robot leg was built as shown in *Figure 1* and with some tweaks made operational. It turned out that the gearing for the motors could not handle the torque needed to hold the leg up and two of the actuators became damaged. This resulted in only the knee and ankle joints remaining operational.

Fig. 1: Finished physical robot

With only two operational actuators, the goal of creating a test bench for future research in gait analysis was not fulfilled. However, the control of the robot was proven to work before the failure. Future work would include finding stronger actuators and adjusting the holders.

The models in both ROS and Matlab works as intended and enables them to be used for future work and educational purposes. The entire project is thoroughly documented using Git [2]. Since the robot is a pilot project, adjustments and perhaps a recreation with better material and hardware is possible and may be required in the future.

ROS

ROS was used to simulate and send setpoints to the physical robot. Cartesian planning in *MoveIt* allowed for executing a functional gait.

- *MoveIt* was set up to plan and execute trajectories
- Simulation was done in *Gazebo*

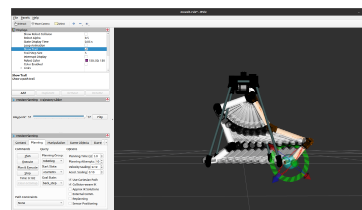


Fig. 2: MoveIt

Mathematical Framework

With the hope of future implementation with ROS, the mathematical model including kinematics was created in Matlab.

- The forward kinematics was found using trigonometry and the Denavit Hartenberg Convention.
- The inverse kinematics was calculated using the found trigonometric expressions.
- A gait analysis of feline movement gave Cartesian waypoints for the end effector.
- Using the inverse kinematics, this was converted to angular waypoints.
- Trajectories between waypoints were calculated using quintic polynomials.
- The robot's Jacobian matrix and body velocity was calculated, enabling alternative inverse kinematics for linear Cartesian trajectories of the end effector.
- A model using the URDF description of the robot was was created to allow simulations (*Figure 3*).

Fig. 3: Four legged model

In the end, Matlab was not implemented as a ROS node for the physical robot.

Acknowledgments

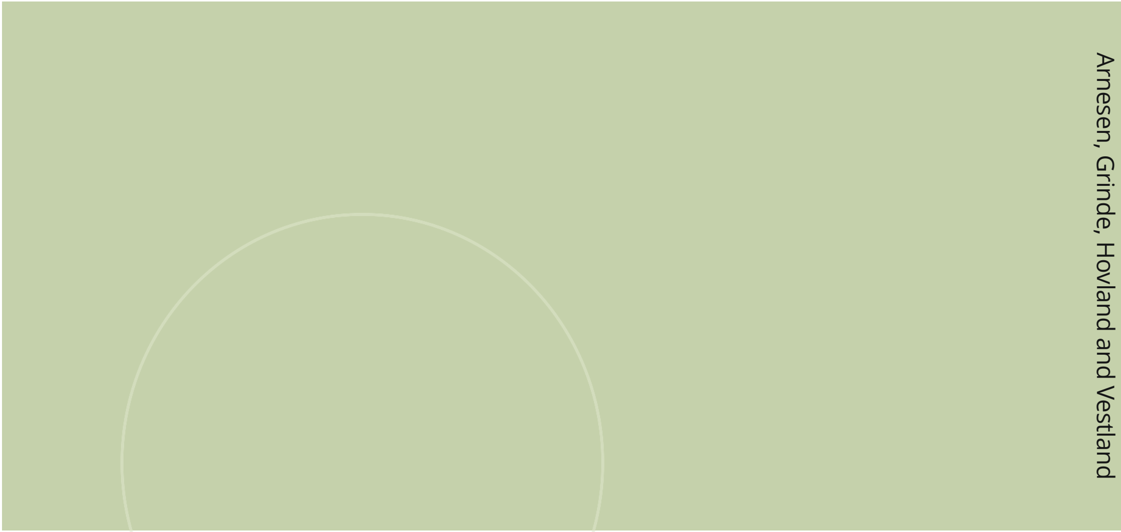
For help and assistance during this thesis, we would like to thank the department of engineering cybernetics and especially the technical and mechanical workshops for lending us equipment and workspaces. Another thanks must be given to MakeNTNU for the 200+ hours of printing time needed to print the parts for our robot. Furthermore, we would like to thank Florian Fischer for his critical help with the creation of the URDF for our robot and Mathias Hauan Arbo for his much-appreciated guidance in ROS.

Finally, we would like to thank our supervisor Torleif Anstensrud for his sound advice, help, and guidance throughout this project.

References

[1] Henrik Moe Arnesen, Kristian Grinde, Vegard Hovland, and Even Vestland. *Development of Biomimetic Robot Leg With ROS Implementation*. 2021.

[2] Henrik Moe Arnesen, Kristian Grinde, Vegard Hovland, and Even Vestland. *E2103 repository*. `https://github.com/VegardHovland/E2103-Bachelor`. 2021.



Arnessen, Grinde, Hovland and Vestland

Development of Biomimetic Robot Leg with ROS Implementation

NTNU

Kunnskap for en bedre verden