Bimodal Quadruped Robot

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

By:

Hushmand Esmaeili (RBE)
Yuen Lam Leung (RBE)
Aadhya Puttur (CS)

Project Advisors:

Mohammad Mahdi Agheli Hajiabadi
William Michalson
Andre Rosendo

Sponsored By:

ODrive Robotics, Protolabs, MathAltitude School of Mathematics, Greensea Systems

Date: April 26 2023

Abstract

The focus of this project is on the development of a quadruped robot named Solo 12, which is designed to switch seamlessly between quadrupedal and bipedal modes of locomotion. This adaptability makes it ideal for navigating unstructured environments. The project delves into the technical aspects of the robot's development, including hardware architecture, electronics installation, and software stack with an emphasis on locomotion controls. Additionally, the project explores the use of computer vision technology for person tracking.

Acknowledgements

The BiQu MQP team would like to thank our project advisors Professor Agheli, Professor Michalson, and Professor Rosendo for their guidance and support throughout the project. This endeavor would not have been possible without their practical suggestions and extensive knowledge in the related field.

We would also like to thank our sponsors: ODrive Robotics, ProtoLabs, MathAltitude School of Mathematics, and Greensea Systems. Their generous donations have allowed us to turn our vision into reality. We would also like to thank Daniel Ali Tribaldos and Professor Nycz at PracticePoint for their assistance in printing the robot chassis, and Dr. Stults from WPI Academic and Research Computing (ARC) for the consultation. We would also like to thank the Open Dynamic Robot Initiative community for their valuable advice.

Lastly, we would like to thank Ethan Chandler for his time and helpful contributions.

Contents

List of Tables

List of Figures

1 Introduction

The bimodal quadruped robot is an adaptation of a project by a group of undergraduate WPI students called the Multi-modal Locomotion Robot [4]. The purpose of this endeavor is because of the various legged robots that exist in order to maneuver in different terrains. This project focused on extending an existing quadruped platform into a bipedal stance using an 8 Degrees-of-Freedom(DOF) robot. This was not achieved physically but the idea was adapted in simulation.

For this current project, we wanted to make adjustments to this robot as there were quite a number of hardware and communication issues with the previous team. We aimed to extend the robot to a 12-DOF robot by using a similar existing design. A new hip joint was added to the robot, whereas before the robot was limited to only moving back and forth, giving it the capability to turn.

Additionally, we introduced vision to the robot and it was a significant improvement in this project. We integrated a Intel RealSense D415 Depth camera into the robot's sensory system, which allowed it to perceive its environment in 3D and capture depth information. This enabled the robot to detect and track objects, specifically humans, in real time.

With the vision capabilities, we aimed to make the robot autonomous by implementing a person-following behavior. The robot was programmed to detect and track a person using the depth camera, and then adjust its movements accordingly to keep a safe distance and follow the person's movements. This added a new level of autonomy to the robot, allowing it to operate in dynamic environments and interact with humans in a more intuitive way.

To accommodate the additional hardware and communication requirements, we made necessary adjustments to the robot's design and control system. This included updates to the robot's kinematic model and control algorithms to account for the increased degrees of freedom and the integration of vision data.

The extended robot with 12-DOF and vision capabilities showed promising results in simulation, demonstrating improved locomotion capabilities and autonomous behavior. Further work could involve physical implementation and testing of the robot, refining the control algorithms, and exploring other potential applications such as human-robot interaction, navigation in complex environments, and object manipulation. Overall, this project aimed to improve the existing bimodal quadruped robot by addressing hardware and communication issues, adding perception system, and extending its locomotion capabilities, paving the way for potential advancements in quadrupedal robotics research.

1.1 Project Objectives

The project aims to upgrade an existing 8-DOF quadruped robot to a 12-DOF system, with the goal of improving its mobility. This upgrade will involve adding additional joints and actuators to the robot's legs, allowing for increased freedom of movement and more advanced locomotion capabilities.

In addition to the hardware upgrade, the project also seeks to implement a dynamic control system that enables the robot to navigate effectively on various surfaces, including rough terrain, slopes, and to avoid obstacles. This control system will utilize sensor inputs and advanced algorithms to dynamically adjust the robot's gait and motion in real time, allowing for adaptive locomotion and improved stability in different environments.

Furthermore, the project aims to extend the robot's perception capabilities by incorporating person-tracking functionality. This will involve integrating sensors such as cameras or depth sensors into the robot's system, enabling it to detect and track individuals in its environment. This new perception system will allow the robot to interact with humans in a more intelligent and responsive manner, opening up potential applications in areas such as human-robot interaction, surveillance, and autonomous navigation in human-populated environments.

1.1.1 Upgrading to 12 DOF

Upgrade the existing 8 DOF quadruped robot to a 12-DOF configuration, enhancing its mobility and versatility.

1.1.2 Implementing a Dynamic Control System

Implement a dynamic control system that enables the robot to navigate effectively on different surfaces, including rough terrain, slopes, and obstacles.

1.1.3 Incorporating Vision for Autonomous Movement

Extend the robot's perception capabilities by incorporating person tracking functionality, allowing it to detect and track individuals in its environment using sensors such as cameras or depth sensors.

2 Background

2.1 Literature Review

2.1.1 Dynamic Locomotion Controls

There have been several attempts to design a control system for dynamic locomotion of quadrupeds. To fully take advantage of the hardware capabilities of quadruped robots, controllers need to be designed and implemented to address challenging issues of dynamic locomotion. Some of these challenges include body stabilization in stance periods, swing phases, high-speed swing phases, and different gaits. The problem can be extended to locomotion in rough terrain that introduces greater external disturbances, and even recovery from pushing or slippage. Different speeds or terrains may then require different gaits and even compliant legs.

The challenges described can be addressed at different levels; hence a control system framework and architecture are required to decouple the controller into different levels and tasks. In [5], the authors adopt a hierarchical control structure to simplify the control framework. Different level planners and controllers are independent from each other, but work with each other to make up the complete control system. The authors reference a control architecture for quadruped locomotion in rough terrains. The controller consists of a higher-level planner, a low-level planner, and a low-level controller. The higher-level planner plans a set of footsteps across a terrain, achieved through a terrain feature extraction module combined with a body path and footstep planners. The low-level planner plans trajectories for the robot's feet and center of gravity. Finally, the low-level controller keeps track of the desired trajectories using different closed-loop systems, including a PD controller, closed-loop foot placement, and a body stabilization module. The authors in [6] provide another example of a hierarchical control system, focused on the controller for a single leg. In this paper, the higher-level controller tunes the various control parameters used by the low-level controller, including the phase of the leg's motion, stride frequency, and ellipse dimensions. The low-level controller is responsible for the trajectory planning of the "toe" or end-effector of the foot, the inverse kinematics (IK) to convert to the trajectory of the joint positions, and an active-compliance block to compute torques to compliantly drive the joints.

For the low-level controller, many different approaches can be taken to achieve body stabilization and compliance of the legs. The researchers in [7] employ Virtual Model Control (VMC) to achieve dynamic balance of the whole body of quadrupeds at trot gaits. For each leg, they designed a VMC for the swing phase and stance phase, which they considered virtual stiffness and damping. For the whole body, they utilized VMC to achieve attitude control, including orientation. Although the VMC approach has a lower computational cost than traditional inverse dynamics approaches, the tracking performance of the proposed controller is not better.

An approach that our team considered was to employ a specific control method for each independent block in the hierarchical structure. In this framework, and to address the challenge of reducing ground reaction forces at the lowest level of the controller, [8] introduces a compliance-force control method for a quadruped robot using VMC. Each leg was modeled as two sets of virtual spring and damper systems. The virtual leg impedance is created in the Cartesian (i.e. task) space, and an impedance controller is used for each leg, which is the low-level controller. For higher-level planners and controllers blocks in the hierarchical structure, many different approaches have been adopted including CPG-inspired trajectory generators [9].

An example of a complete and cohesive controller for dynamic locomotion is introduced in [10]. The authors in the paper propose a controller that combines whole-body control (WBC) and model-predictive control (MPC). Unlike existing WBCs that attempt to track body trajectories, their controller focuses on the reaction force command. The MPC block finds optimal reaction forces over some time horizon with a simple model, and the WBC block computes joint torque, position, and velocity commands based on optimized reaction forces. This novel controller was tested on the MIT Mini-Cheetah. This approach looks like a promising solution for our Solo12 robot. The implementation of such a control pipeline was also implemented by the creators of the Solo12 within the framework of Open Dynamic Robot Initiative (ODRI) [11]. This confirmed to us that the WBIC with MPC control framework would be the approach we would attempt to integrate into our Solo12 robot.

2.1.2 Model Predictive Control and Whole-Body Impulse Control Framework

The method presented in [10] aims to simplify dynamic locomotion control by dividing it into two controllers. The first controller uses an MPC with a simple lumped mass model to find the optimal reaction force profiles for a given trajectory, while the second controller uses whole-body control with a more accurate dynamics model and high frequency feedback control to achieve high bandwidth control.

The objective of MPC in the control framework in [10] [12] is to find reaction forces which would allow our robot to follow a given trajectory. As mentioned, the MPC uses a simple lumped mass model:

$$m\ddot{\mathbf{p}} = \sum_{i=1}^{n_c} \mathbf{f}_i - \mathbf{c}_g \tag{1}$$

,

$$\frac{d}{dt}(\mathbf{I}\omega) = \sum_{i=1}^{n_s} \mathbf{r_i} \times \mathbf{f}_i \tag{2}$$

where $\mathbf{p}$, $\mathbf{f}_i$ and $\mathbf{c}_g$ are 3D vectors representing the robot position, reaction force of the i^{th} leg, and gravitational acceleration with respect to the global frame. $\mathbf{I}$ is the rotation inertia tensor and ω is the angular velocity of the body. $\mathbf{r}_i$ is the position of the i^{th} leg contact point with respect to the center of mass (CoM) of the robot.

To ensure a convex optimization problem, three simplifications are made to the lumped mass model, including small roll and pitch angles, states close to the commanded trajectory, and small pitch and roll velocities. A quadratic programming problem is used to minimize a cost function. By applying several simplifications to the inverse dynamics (i.e., approximating the multi-body system as a rigid body with instantaneous states to keep the time-varying system linear), the approach keeps the formulation convex, and hence, the problem is fast to solve and can be solved by a unique global minimum.

The MPC formulation is given by:

$$\begin{aligned}
\min_{\mathbf{x},\mathbf{f}} \quad & \sum_{k=0}^{m} ||\mathbf{x}(k+1) - \mathbf{x}_{ref}(k+1)||_{\mathbf{Q}} + ||\mathbf{f}(k)||_{\mathbf{R}} \\
\text{s.t.} \quad & |f_x| \leq \mu f_z \\
& |f_y| \leq \mu f_z \\
& f_z > 0
\end{aligned} \tag{3}$$

where $\mathbf{x}(k+1)$ is the state at time $k+1$ and $\mathbf{f}(k)$ is the control input at time k. The constraints represent an approximation of friction cones at the feet of the robot to prevent slippage. The predictive controller aims to guide a system along a desired trajectory while minimizing control effort and adhering to constraints. It achieves this by finding the best sequence of control inputs over a prediction horizon. This enables the controller to plan for periods of flight and regulate the states of the system during a gait when the system is underactuated. The controller also seeks the best solution if the system cannot exactly track the reference trajectory. Our attempt to implement MPC in MATLAB is described in Section 3.2.2.

The WBIC block utilizes the optimal reaction forces computed in the MPC block to determine joint position, velocity, and torque commands. To calculate joint position, velocity, and acceleration, they employ an inverse kinematics algorithm that prioritizes task execution. In order to determine the torque command, they employ quadratic programming to obtain the reaction forces, which not only minimizes errors in acceleration command tracking and reaction force command tracking, but also meets the inequality constraints of the resulting reaction forces. The multi-body dynamics can be written as:

$$\mathbf{A}\begin{bmatrix}\ddot{\mathbf{q}}_f \\ \ddot{\mathbf{q}}_j\end{bmatrix} + \mathbf{b} + \mathbf{g} = \begin{bmatrix}\mathbf{0}_6 \\ \boldsymbol{\tau}\end{bmatrix} + \mathbf{J}_c^T\mathbf{f}_r \tag{4}$$

where $\mathbf{A}$, $\mathbf{b}$, $\boldsymbol{\tau}$, $\mathbf{f}_r$, and $\mathbf{J}_c$ are the generalized mass matrix, Coriolis force, gravitation force, joint torque, augmented reaction force and contact Jacobian. $\ddot{\mathbf{q}}_f$ is the acceleration of the floating base and $\ddot{\mathbf{q}}_j$ is the vector of joint velocities. The final step of the WBIC block is to compute a torque command from the reaction forces, $\mathbf{f}_r$, and the configuration space acceleration, $\ddot{\mathbf{q}}$.

2.1.3 Computer Vision Applications on Machine

When we talk about computer vision, it usually involves getting visual input and inferring some type of information from it. Computer vision replicates the human vision system by creating a system that processes, analyzes, and understands digital images. It can help a machine automate tasks for us by helping a machine understand an image; this is called machine vision [13]. Machine vision or machine perception is a system that has computer vision capabilities and uses that information to make a decision or perform a specific action. While computer vision imitates the human vision system, machine perception mimics human judgment. Machine perception is exactly what we want to achieve with our bimodal quadruped robot. The robot should be able to process, analyze, and extract information from its surroundings and be able to perceive its environment to make certain decisions on its direction or movement. Especially since this robot has hybrid locomotion, analyzing the environment to change different modes of locomotion is essential.

Our goal for the quadrupedal bimodal robot is to be able to follow a fixed person.

2.1.4 Solo12

The Solo12 is an Open Toque-Controlled Modular Robot Architecture for Legged Locomotion Research. The design is developed by the Open Dynamic Robot Initiative (ODRI), which is a collaboration between the Motion Generation and Control Group, the Dynamic Locomotion Group and the Robotics Central Scientific Facility at the Max-Planck Institute for Intelligent Systems, the Machines in Motion Laboratory at New York University's Tandon School of Engineering and the Gepetto Team at the LAAS/CNRS [3]. The Solo12, as shown in figure 1 , is 12 Degrees-of-freedom (DOF) quadruped robot that has a weight of 2.5kg and a dimension of 30 x 45 x 34cm. The robot can perform a diverse range of motion, including push recovery, walking on unknown terrain, and high jump (65-100cm). It is a low-cost, lightweight, and highly-dynamic quadruped platform that can be built with in-house manufacturing and off-the-shelf components. The details of developing the Solo12 platform are available on the Open Dynamic Robot Initiative's GitHub repository.

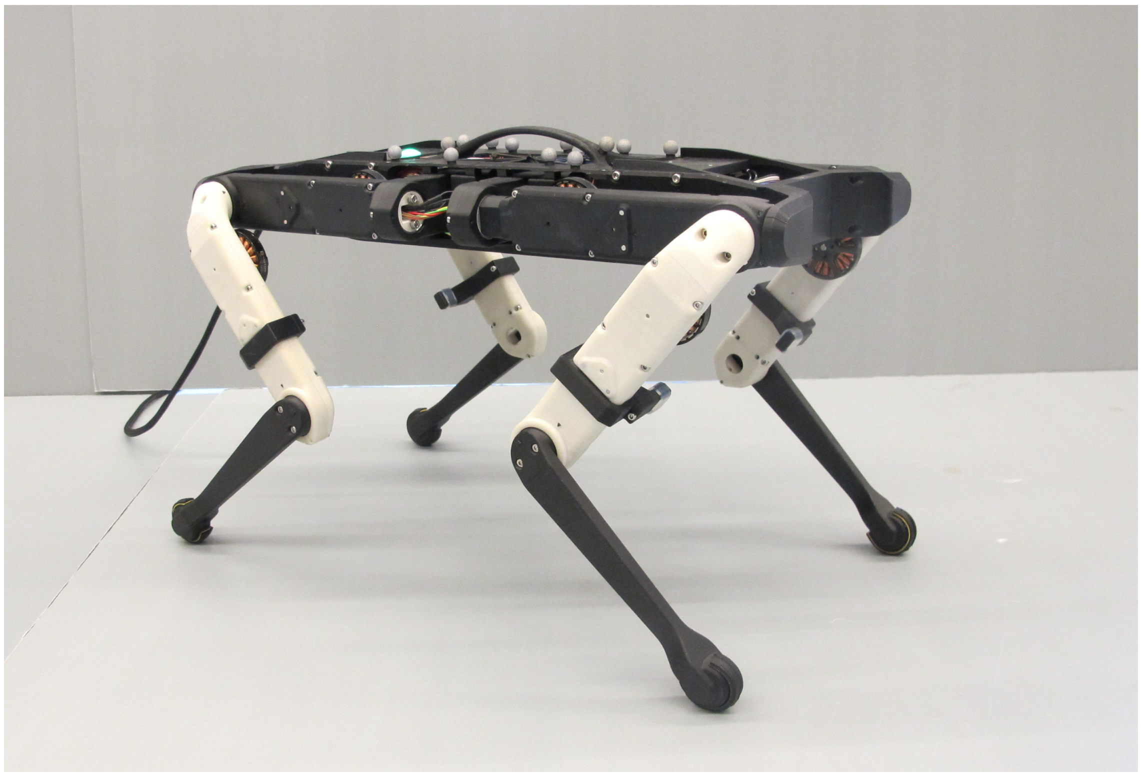


Figure 1: Solo12[3]

2.2 Funding and Partnership

Proposed Budget	
Components	Budget
Actuator Modules	$900
Electronics	$500
Hardware	$150
Fabrication	$750
Total Cost	$2300

Table 1: Proposed Budget

In order to gather all the components for the physical robot, the team proposed a budget of $2,400. The breakdown of the budget is shown in Table 1, consisting the cost of producing the actuator modules, electronics, hardware, and fabrication. The proposed budget exceeded the team's initial WPI funding of $850, therefore, we have put together

Actual Budget	
Components	Budget
Actuator Modules	
Motors	$312
Encoder	$568
Transmission System	$261
Total	$1141
Electronics	
Motor Controller	$15
Microcontroller	$45
Camera	$358
Miscellaneous	$45
Total	$463
Hardware	
Fastener	$200
Fabrication	
Material	$240
Total Cost	$2044

Table 2: Actual Cost

Funding and Sponsorship	
WPI Funding	$850
Dr. Glenn Yee Project Award	$1000
ODrive Robotics	$500
ProtoLabs	$350
MathAltitude School of Mathematics	$100
GreenSea System	$100
Total Funding	$2400

Table 3: Funding and Sponsorship

a grant proposal to seek funding from external sponsors. Through communicating with potential external sponsors, we were able to raise $1,050 funding, which include product sponsorship and partnership. One of our team members, Hushmand Esmaeili, was awarded $1,000 from the Dr. Glenn Yee Project Award. With WPI funding, external funding, and the award, the team was able to obtain $2,400 in funding as shown in Table 3.

As shown in Table 2, the total actual cost of the robot is $2,044. While the actual spending for the actuator modules, electronics, and hardware exceeded the proposed budget, the partnership with the sponsors significantly reduced the budget allocated for fabrication.

2.3 Limitations of Past MQP

We briefly discussed the past MQP (Multi-modal Locomotion Robot) and how this project aims to extend the project from an 8-DOF robot to a 12-DOF robot. There are other limitations we had from utilizing the robot chassis from past MQP. Without the additional hip joint, it would be difficult to turn which would aid the robot to avoid obstacles. The past MQP also used the Robot Operating System (ROS) framework for node communications. The delays in node communication led to problems with timing mismatch. The robot electronics were custom-built and briefly documented, making it difficult to replicate the process and learn how to use them for this current MQP and future MQPs.

In previous discussions, we recognized the limitations of the previous Multi-modal Locomotion Robot (MQP) project, where the 8-DOF robot could only move forward and backward. This current project aims to extend the robot to a 12-DOF system. One significant limitation was the absence of an additional hip joint, which hindered the robot's ability to navigate around obstacles effectively [14]. Furthermore, the use of ROS as the primary communication method with the motors resulted in timing issues due to delays in data transmission. Additionally, the custom-built robot electronics had insufficient documentation, posing challenges for replication and utilization in the current MQP and future projects.

3 System Design

3.1 Hardware Architecture

The modification of the 8-DOF robot to a 12-DOF robot requires additional hardware to accommodate the four new hip joints, including additional motor controllers and actuator modules. Utilizing existing hardware available from the previous MQP and additional electronics, the hardware architecture of the BiQu robot is redesigned to address preexisting problems and cooperate with the new software architecture.

3.1.1 Materials

One of the major issues of the 8-DOF robot built by the past MQP team is the chassis material. The 8-DOF robot has a PLA chassis, which has a Heat Deflection Temperature of 50 °C. The heat from the motor running caused the PLA shell to melt and solidify once cool. Apart from shell melting, one of the actuator modules broke off during transportation. Seeing how frequently the shells are damaged, we decided to reprint all the shells in a different material with higher strength and heat tolerance.

The ODRI community recommended printing the chassis and shells in PC-ABS, which combined the properties of ABS and PC. PC-ABS has a higher heat tolerance, and at the same time, higher tensile and flexural strength than PLA. We compared the properties of the materials in Table 4. For ABS and PC, we were able to produce some test prints using the Ultimaker 3 at the Makerspace Prototyping Lab. We consulted multiple advisors and manufacturing labs on campus, and were not able to locate a printer that could provide and print PC-ABS. We then reached out to PracticePoint, who suggested us to look into the material Onyx. Figure 2 shows a comparison of the shell printed in ABS and in Onyx. The ABS shell had visible layers and required extra deburring to ensure the components of actuator modules sit nicely on the shell, while the Onyx shell was smoother surfaces and cleaner edges. Given the physical properties of Onyx, we evaluated that it would be a good choice of material.

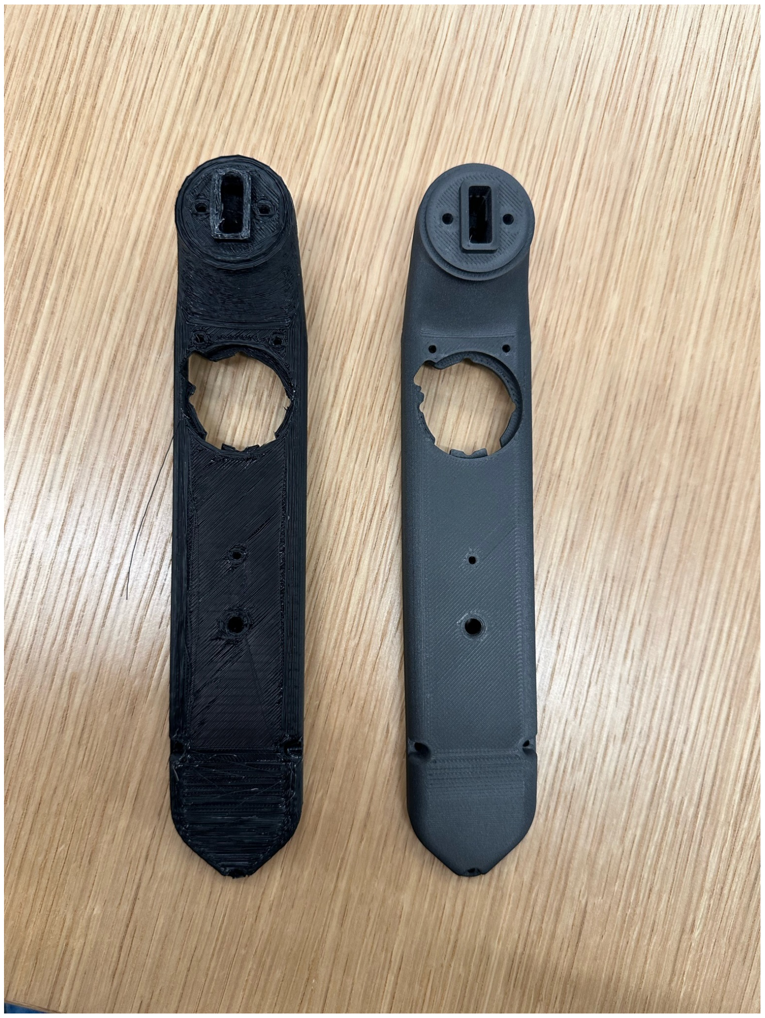


Figure 2: Comparison of ABS (left) and Onyx (right) prints

3.1.2 Fabrication

To fabricate the components with the materials that we have chosen, we researched methods that can produce high-quality parts at a low cost. For the chassis and the shell of each actuator module, we were able to utilize the resources at PracticePoint, which has a

Comparison of Materials					
Material	PLA	ABS	PC-ABS	PC	Onyx
Density (g/cm3)	1.3	1.02	1.1	1.15	1.2
Tensile Strength (mPA)	37	57	41	68	40
Flexural Strength (mPA)	43.6	75	68	104	71
Heat Deflection Temp (°C) @67psi	50	100	110	138	145

Table 4: Comparison Table of Materials [1][2]

Markforged Mark Two that is equipped with Onyx. For the high-strength and high-precision transmission parts, the team first consulted the Academic Research & Computing Center (ARC). The ARC has a Formlabs Form 2 printer that is able to print a variety of Formlabs Resin. To minimize cost, we experimented printing the transmission parts with Tough Resin at the ARC. Eventually, the partnership with Protolab provided an opportunity to print the transmission part with Accura Xtreme 200 using Stereolithography (SLA). Additional steps are required to modify the printed parts, including installing helical inserts, and reducing the diameter of the pulleys using a CNC lathe.

3.1.3 Hardware Acquisition

Motor Controller. To extend the degrees of freedom of the robot within a manageable budget, we decided to make use of most electronics from the chassis of the 8-DOF quadruped robot. The past MQP team utilized four ODrive motor controllers. ODrive motor controller provides functionality such as position control, velocity control, and torque control on brushless DC motors. Each ODrive motor controller is connected to two motors and two encoders, therefore, controlling eight joints in total. The four additional hip joints would require two additional ODrive motor controllers. The team reached out to ODrive Robotics, who were generous enough to sponsor two motor controllers for us.

Single Board Computer. The integration of dynamic controllers and computer vision would require more computing power from the single board computer. The Raspberry

Pi 4B - 8GB RAM that the past MQP team used for running the software stack was sufficient in this case, therefore, we continued using the same Raspberry Pi 4B for this project.

Microcontroller. In order to interface the six ODrive motor controllers with the software stack, six UART communications are required. The past MQP team utilized a Teensy 4.1, which supports eight UART connections. Despite the pre-existing Teensy 4.1 from the past MQP team was experiencing hardware failure, we purchased and continued using Teensy 4.1 in this project for the same reason: there are sufficient UART ports on the Teensy 4.1 to bridge communication between Raspberry Pi 4B and the six ODrive motor controllers.

Actuator Modules. Following the open-sourced design provided by ODRI, the extension from the 8-DOF quadruped, Solo8, to the 12-DOF quadruped, Solo12 requires four extra actuator modules and a reprint of the entire chassis. An actuator module consists of a brushless motor (T-Motor Antigravity 4004, 300KV), a high-resolution optical encoder with an index pulse, and a 5000 pulse-per-revolution code wheel mounted to the motor shaft. It has a 9:1 dual-stage timing belt transmission consisting several motor shafts and pulleys. To minimize cost, we started off by reorganizing the inventory to look for reusable parts. We were able to obtain one extra motor from the past MQP team's inventory. The motors, encoders, code wheels, and timing belts are off-the-shelf products, which we were unable to reduce cost. We figured the most cost-consuming components of the actuator modules are the shafts and pulleys used in the transmission system. The shafts and pulleys are small and require high precision, we will either have to order a form cutter or utilize a wire EDM machine. We consulted some members at the Washburn Machine Shop on campus and Professor Michalson, and learned that machining these parts in-house would be a challenge. We looked into another option: purchasing preassembled plug-and-play encoder kits from PWB Encoders. The drawback of this option is the uncertainty in delivery time with foreign shipping. After considering the differ in cost, as shown in table 5, we ended up purchasing the preassembled encoder kits which would reduce the cost by $300.

Cost of Actuator Modules			
Components	Quantity	Unit Cost	Unit Cost
Encoder (Broadcom AEDT-9810-Z00)	4	$36	
CodeWheel (625cpr, ID 7mm / OD 25,56mm)	4	$30.1	
CodeWheel Mount	4	$1	
Motor shaft (4mm stainless steel rod)	4	$24	
Motor Pulley	4	$48	
Center Pulley	4	$83	
PWB Encoder Kit	4		$121.4
Foreign Shipping Fee	1		$77.6
Total Cost		$884.4	$563.2

Table 5: Cost of Actuator Modules

Cameras. One of the objectives of this project is to implement perception into the BiQu robot, therefore, we have to acquire cameras that are able to supply data for computer vision. The team was able to obtain an Intel RealSense D415 Depth Camera from one of our advisors, Professor Michalson. In B-term, we considered using a Lord MicroStrain IMU, which the MIT Mini Cheetah and the original Solo12 design use. However, its cost of $850+ was way above our budget. Therefore, we decided to acquire another camera, the Intel RealSense Tracking Camera T265, which provides vision-fused IMU data for a price cut of $500+.

3.1.4 Electronics Housing

On the previous 8-DOF version robot, all of the electronics are tethered: the Raspberry Pi, Teensy, ODrive motor controllers, and the power supply. With the conversion from the Solo8 to the Solo12 design, the team redesigned the chassis to house all the electronics onboard. The calculation of optimal payload is done using the Cheetah Software from MIT Biomimetics. We extended the floating-base dynamic class to include the Solo12 dynamic model, which helped us in determining that the Solo12 can have a payload of approximately 2kg.

With six motor controllers, each connecting four sets of cables from the motor and encoders, the cables must be organized inside the chassis to prevent tension and interference with the motors. As shown in figure 3, the cables are organized according to the module. To work better with the software stack, each ODrive motor controller is designated with a module: Hip AA, Hip FE, or Upper Leg. With the cables coming from both sides of the robot, we organized the cables to keep them away from the motors.

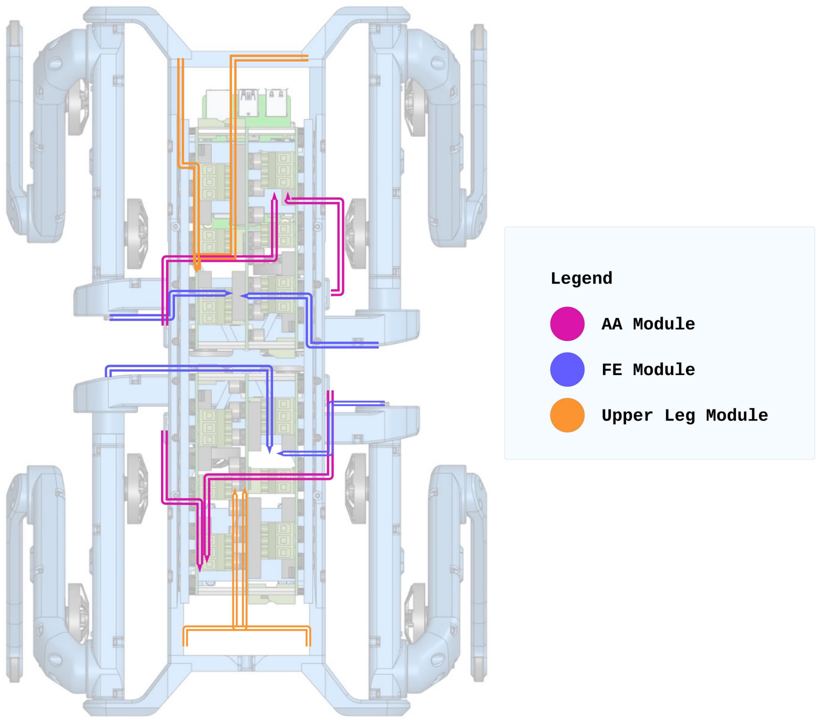


Figure 3: Wire Organization

3.1.5 Wiring

Some of the changes in wiring from the previous 8-DOF version robot include eliminating the block for servo (24V-to-8.4V Converter and Servos), and using SPI communication between Raspberry Pi and Teensy 4.1 instead of UART. The overall electrical wiring diagram is shown in figure 4. The power supply was divided into two routes: 24V and conversion to 5V. The ODrive motor controllers have an input voltage range from 12V to 56V, while the

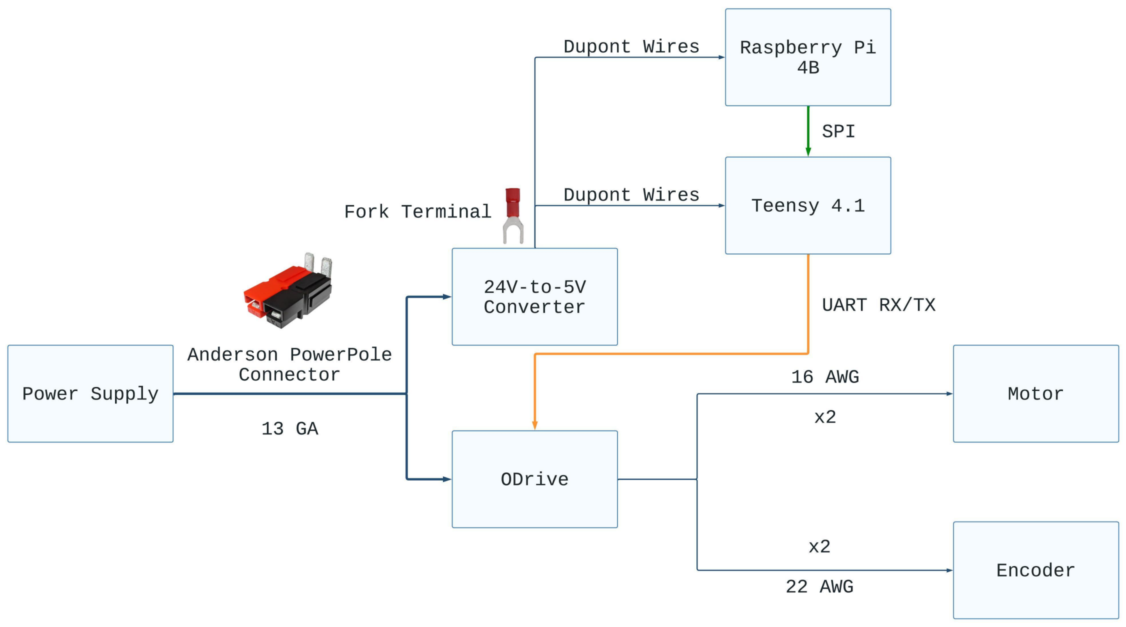


Figure 4: Electrical Diagram

Raspberry Pi and Teensy required an input voltage of 5V. The power cables were connected using Anderson PowerPole connectors, and the DuPont wires for supplying power to the microcontrollers were connected using fork terminals.

The Raspberry Pi communicated with six ODrive motor controllers via Teensy. The Teensy was used to bridge the communication because it supports eight UART serial communication. As shown in figure 5, the Raspberry Pi utilized SPI communication to send and receive data to/from Teensy, while Teensy used six UART serial communication to send and receive data to/from the ODrive motor controllers.

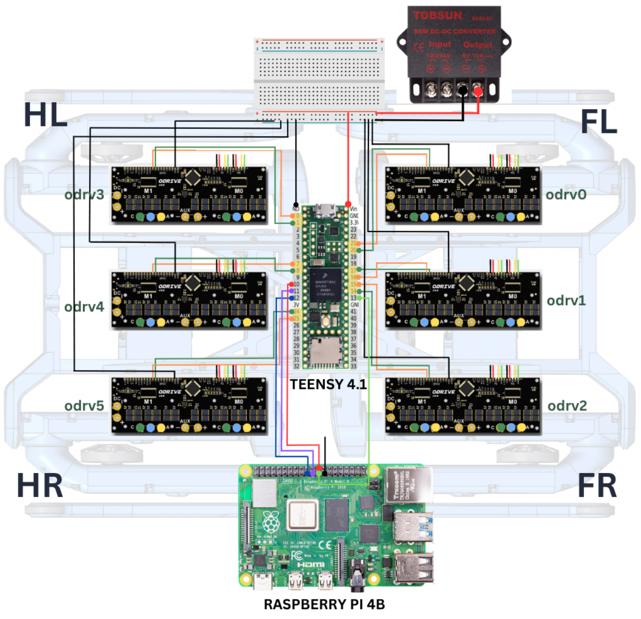


Figure 5: Wiring Diagram for UART and SPI Communication

3.2 Software and Control Architecture

Our team went through different design iterations. During the preliminary design phase in our original proposal, we planned the software architecture to run on an Arduino and a Pi 4, following the design of the previous MQP. We decided to house the code on the Pi 4 within ROS2, which we chose over ROS1 due to its long-term prospects. We determined that the high-level controller and computer vision modules should be implemented on the Pi, while only the low-level controller needed to be implemented on the Arduino. This was because the Pi was fast enough for the computationally demanding operations of the computer vision modules, while the low-level controller needed to interact with the joints and therefore required the real-time capabilities of the Arduino. We also planned to do some of the object detection model training through cloud computing. In our design, we ensured that the Arduino would get data from the Pi through the command thread to feed the low-level controller. Additionally, outputs from the Arduino and computer vision modules were to be used in the main thread and would in return feed the Arduino indirectly through the command thread. In the following subsections, we will provide a comprehensive overview of our project's development process. Firstly, we will discuss our initial design and testing of different simulators. Then, we will elaborate on our implementation of an MPC algorithm for the quadruped using MATLAB and the reasons behind our eventual pivot to the MIT Cheetah Software. Lastly, we will detail our design for the low-level control program, which was specifically developed for the Teensy 4.1.

3.2.1 Testing Different Simulators

As part of this preliminary design that was centered around a ROS framework, we tested using the Gazebo simulator with our robot model, as we were most familiar with it. After importing the URDF provided in the Open Dynamic Robot Initiative (ODRI) GitHub repository, modifying the configuration YAML file, and setting up the simulation world, we were able to see a model of the Solo12 in Gazebo.

Simulation Setup. We first encountered issues when the Solo12 would start drifting once the simulation began. We approached this issue in multiple ways. First, we made sure the physics of the world was correct. This included experimenting with an empty simulation world and an existing simulation world from an example program. Then, we checked all the parameters in the URDF against the URDF provided by ODRI. Finally, we turned on a visualization of the inertia of the Solo12 model in Gazebo and noticed deviation, as shown in figure 6. We figured that the URDF provided included incorrect and outdated parameters. We later were able to find a URDF that contains the correct parameters in another repository.

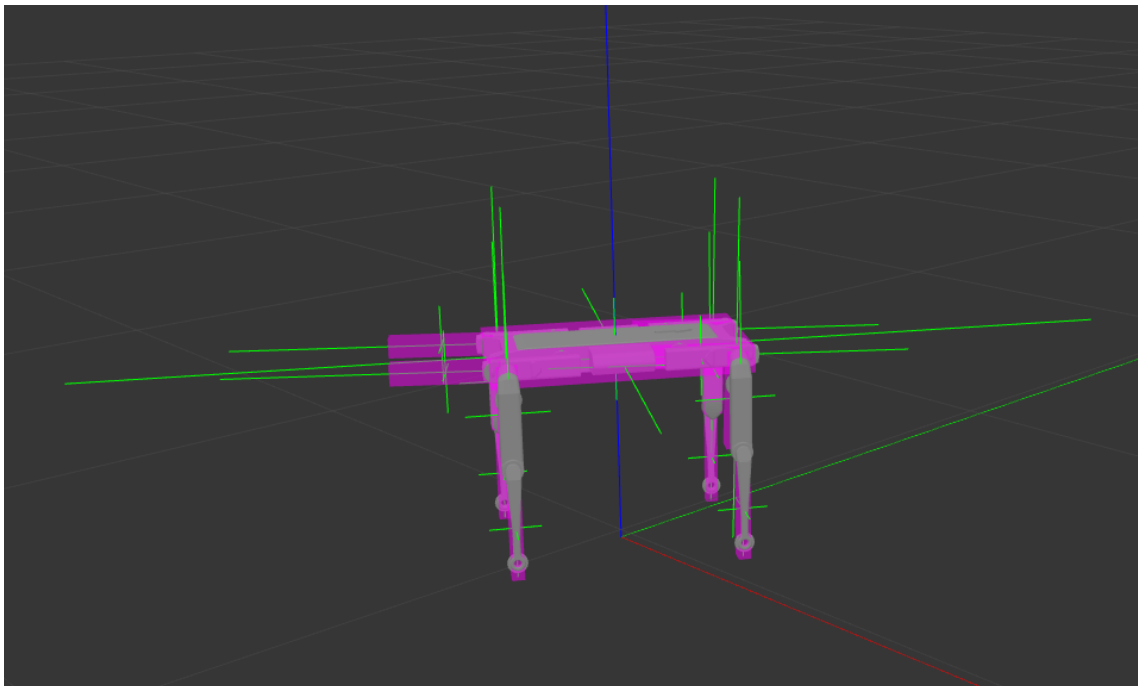


Figure 6: Incorrect Inertia in Gazebo

Controller. The purpose of utilizing a Gazebo simulator was to simulate the joints. To use a controller to control the joints, we can use *ros2_control* framework. The *ros2_control* package has several controller options such as Position Controller, Velocity Controller, Effort Controller, Forward Command Controller, and Joint Trajectory Controller. For this project, we would like to command each joint with an input torque, therefore, we focused on implementing the effort controller and the forward command controller. We were able to command the joints to move with a given torque using the forward command controller. During the process of implementing *ros2_control* and *gazebo_ros2_control*, we noticed that the two packages are still under development. Comparing to the variety and the ease of implementing a controller in *ROS*, there was sufficient support and documentation for *ROS2*.

Gepetto - Pybullet Simulator. The ODRI community has developed a Pybullet simulator with Solo12. We were able to experiment with the simulator after cloning the public GitHub repository. Despite having the Gepetto simulator and gamepad working, we chose C++ as the programming language for the following reasons: unifying the programming language used for high-level and low-level Arduino controller, reducing runtime, and avoiding Python porting issues. Therefore, we decided not to use the Gepetto simulator as it is written with Python.

3.2.2 MPC Implementation in MATLAB

Our initial plan was to implement the MPC and WBIC control framework from scratch. We spent a lot of time attempting to understand how MPC works, specifically the derivation of the single lumped mass model of a quadruped used for the MPC formulation in [12]. We also spent lots of time implementing it in MATLAB by following the paper. Next, we will elaborate on our implementation and simulation in MATLAB.

We used the OSQP quadratic programming (QP) solver for our implementation [15], since there is a MATLAB package for it. In our MPC main function, we first defined constants such as the mass of the robot, the coefficient of friction, maximum reaction forces, inertia tensors, and the weight matrix (tunable). Then, we computed the condensed matrices for **A** and **B**, **A**_qp and **B**_qp. At this point, we defined an arbitrary gait. Next, we computed the **L** and **K** matrices. From these, we computed the **H** and **g** matrices, from the rewritten problem given by [12]:

$$\begin{aligned} \min_{\mathbf{U}} \quad & \frac{1}{2}\mathbf{U}^T\mathbf{H}\mathbf{U} + \mathbf{U}^T\mathbf{g} \\ \text{s.t.} \quad & \underline{\mathbf{c}} \leq \mathbf{C}\mathbf{U} \leq \overline{\mathbf{c}} \end{aligned} \tag{5}$$

At this point, we computed the lower and upper bound. Finally, having set up the problem, we passed the parameters to **osqp** method to solve the optimization problem and find the optimal reaction forces to follow the trajectory. We ran a few tests to verify our formulation and implementation of MPC. In figure 7, we plot the reference trajectory against the trajectory followed by the MPC; the actual initial state is the same as reference trajectory initial state. Figure 8 is similar to the previous figure, except the actual initial state is different to reference trajectory initial state. We observe that the actual trajectory converges to reference trajectory. Figure 9 is just another example of the MPC controller following a reference trajectory.

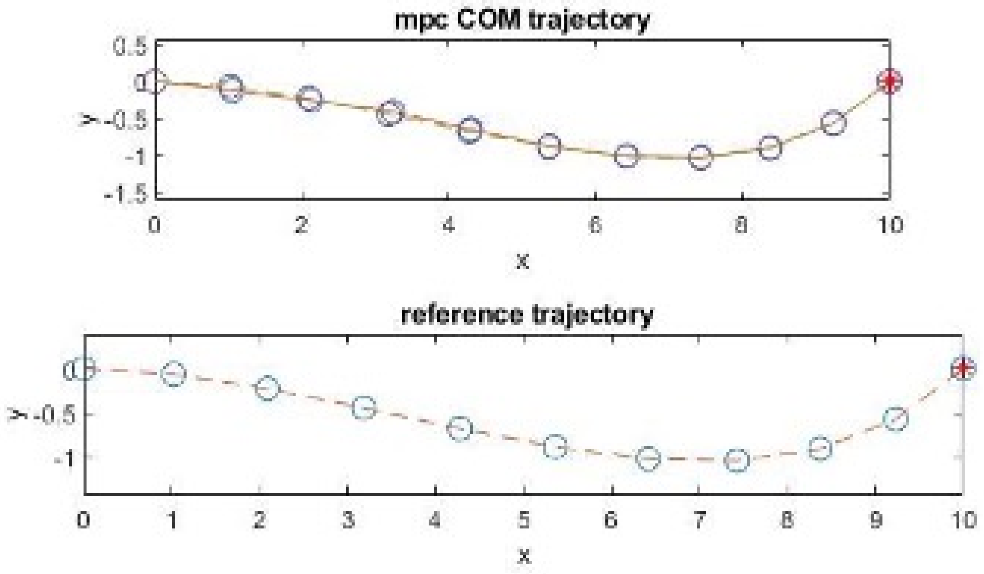


Figure 7: Plot of reference trajectory to follow (dotted) vs. trajectory found by MPC controller (solid). Actual initial state same as reference trajectory initial state.

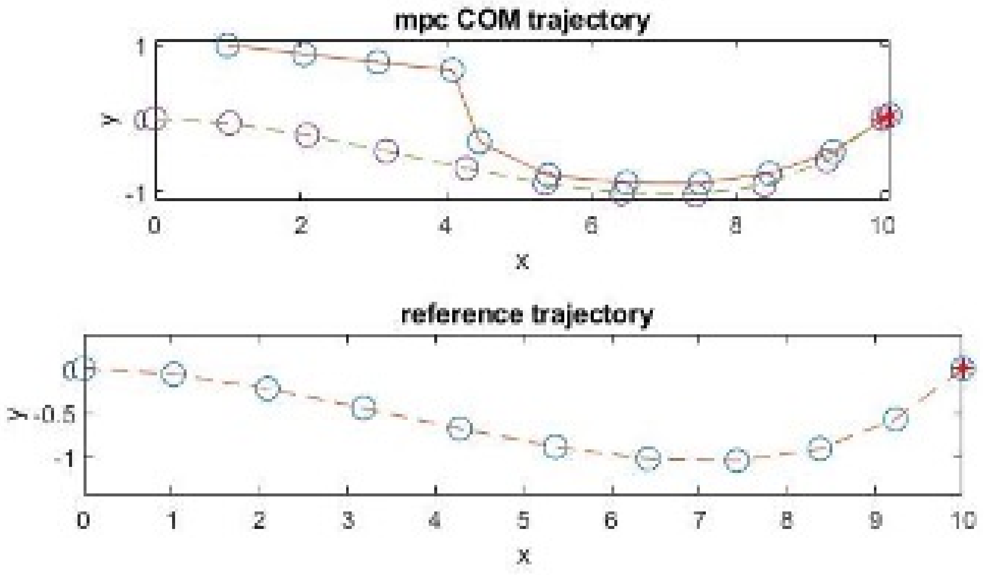


Figure 8: Plot of reference trajectory to follow (dotted) vs. trajectory found by MPC controller (solid). Actual initial state different to reference trajectory initial state. Actual trajectory converges to reference trajectory.

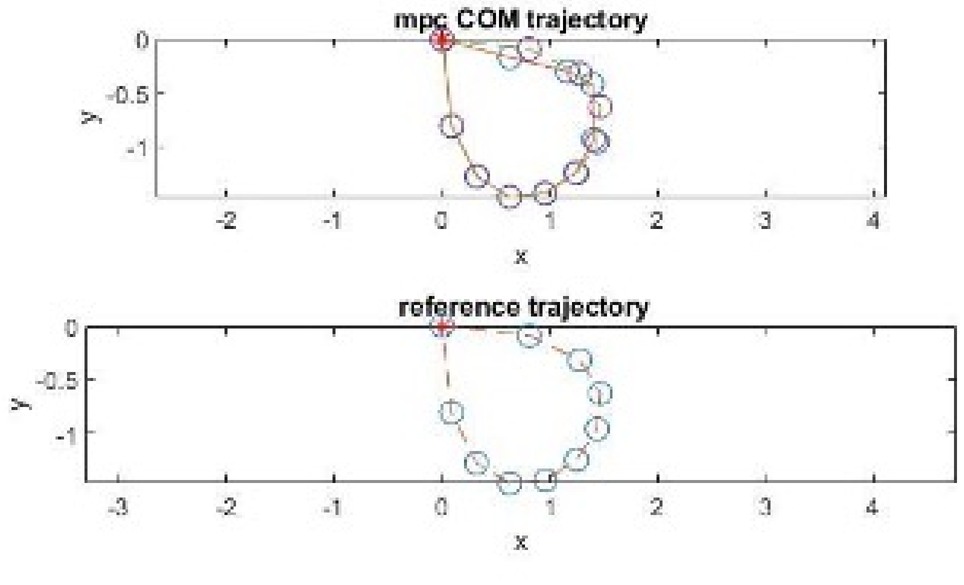


Figure 9: Plot of reference trajectory to follow (dotted) vs. trajectory found by MPC controller (solid). Third experiment.

3.2.3 Pivot to MIT Cheetah Software

After testing different simulators and attempting to implement MPC, we realized that implementing an entire software stack from scratch would be way beyond the scope of an MQP and we would not have enough time. Doing everything from scratch included (and not limited to) deriving and implementing MPC and WBIC from scratch, developing state estimator modules, writing interfacing code for all the hardware, implementing rigid-body dynamic algorithms from scratch, and many other modules. At this point, if we wanted to use ROS as a middleware and Gazebo as a simulator, we would have had to implement the whole control architecture on our own, which, again, we wanted to avoid. Using the Solo12 open-source codebase was an option, since we had tested the robot running in PyBullet. However, we started to research and explore the possibility of using the open-source MIT Cheetah Software. Their software stack provided a complete solution for rigid-body dynamics, simulation, and code to interface with hardware as well as the entire hybrid control framework that we wanted to use for our robot. As the scope of our project broadened in other areas — for example, in upgrading the hardware as well — we decided to adapt and expand our high-level locomotion controller from the open-source MIT Biomimetics Cheetah Software [16] [17].

Brief Overview of Cheetah Software Stack. The Cheetah Software contains common libraries, a robot control framework, and a simulator framework, including dynamics and contact simulation. The software can run on both Mac OS and Linux. The software is split into different modules. There is a common library with utility functions used by other modules. The simulator contains a dynamics simulator, a 3D visualizer, and a user interface (UI) to set parameters and modify the terrain/environment. The Robot Framework module contains a library of code to interface robot control code with the simulator and the actual robot hardware. The User Controller contains the robot control code, which depends on the Robot Framework library. Since the Cheetah Software is a large codebase and more details about it can be found in [17], we will expand on the modules most relevant to our project.

In the Robot Framework module, the main function *main_helper.cpp* parses the command line arguments and starts the appropriate driver. The users can run the physical robot, which creates an instance of *HardwareBridge*, the interface between robot code and robot hardware. This class initializes the hardware of the robot and allows the robot controller to access it. Users can also run the simulation, which creates an instance of *SimulationBridge*, which runs a RobotController and connects it to a Simulator, using shared memory; it is the simulation version of HardwareBridge. Most of the modifications we did were in *HardwareBridge*, though we also had to modify *SimulationBridge* to get our robot model to work in simulation.

An essential aspect of HardwareBridge is that it uses *PeriodicTaskManager* and *PeriodicTask* instances to manage the "connections" with the different hardware modules. PeriodTask is an implementation of a periodic function running in a separate thread. Periodic tasks have a task manager, which measures how long they take to run. Some examples of periodic tasks in HardwareBridge include the SPI task (for SPI communication to the Spine board, and the motor controller) and the SBUS task (for R/C controller communication). The IMU, microstrain, also runs in a separate thread and is initialized in HardwareBridge.

Another class in the Robot Framework library is the *RobotRunner* class, which is initialized in HardwareBridge and SimulationBridge. This is the common framework for running robot controllers, and it is a common interface between control code and hardware/simulation for Mini Cheetah, Cheetah 3, and now also BiQu. Interestingly, RobotRunner is a (inherits from) PeriodTask. RobotRunner **init()** initializes the robot model, state estimator, leg controller, robot data, and any control logic-specific data, while **run()** runs the overall robot control system by calling each of the major components to run each of their respective steps.

The Robot Framework library also includes a couple of different classes used to interface with specific hardware and the ports they are connected to. One that was particularly relevant to our project was the *rt_spi* class, which is used for SPI communication to the Spine board (low-level motor controller). The Spine board is composed of two STM32xx micro-controllers used as custom motor controllers, each communicating to two legs. It uses SPI to communicate with the master board that runs the Cheetah Software (high-level controller). The rt_spi class initializes and opens the SPI driver (spidev in Linux) and sends/receives data through SPI communication. In HardwareBridge, the "spi" periodic task calls **runSpi** (every 0.002 seconds) to perform SPI communication. The SPI message being sent contains desired joint state (i.e. desired joint positions, velocities, feedforward torques and PD gains), while the received message contains feedback joint states (i.e. feedback joint positions and velocities) used for the state estimation module in the Cheetah Software.

The User Controller library contains the MIT_Controller, which is the main controller run in the Mini Cheetah. The user controller contains a main finite state machine (FSM) — in the *ControlFSM* class — that manages the robot's controls. All of the necessary data is passed down to this class and is stored in a struct. This data includes the *Quadruped* object, the *StateEstimatorContainer* object, *LegController* object, *GaitScheduler* object, *DesiredStateCommand* object, *RobotControlParameters* object, *VisualizationData* object, and *MIT_UserParameters* object. The ControlFSM class handles calls to the FSM State functions and manages transitions between all of the states. These states include recovery stand, locomotion, QP stand, vision, and backflip. Each of these states has their own FSM associated with them.

Each state and its corresponding FSM make use of different controllers contained within MIT_Controller user controller. This is where the Convex MPC and WBIC controllers are implemented and used. There are other controllers for other modes, including Balance-Controller and BackFlip. Any future user controller with completely different frameworks can be implemented as a new user controller. As mentioned previously, for this project, we decided that the features and framework proposed in [10], which was implemented in

MIT_Controller, would meet our needs.

Extensions and Modifications to Cheetah Software. As we decided to pivot to the MIT Cheetah Software, we identified the major components that we would either have to modify or add in order to adapt the software to work with BiQu, which includes the following:

- Create a kinematic and dynamic model of BiQu
- Expand simulation to simulate BiQu
- Add interface with T265 camera for pose state estimation (hardware) and integrate into software stack
- Add interface with D415 camera for person tracking (hardware) and integrate into software stack
- Add interface with low-level/motor controller (hardware)
- Add high-level planning module

Figure 10 shows some of the modules that we used straight from the Cheetah Software as well as the specific modifications we did for BiQu. We will expand on the implementations of these changes and modifications in Section 4.3.

3.2.4 Design of Program Flow for Teensy 4.1 Low-level Controller

The main purpose of the program that ran on the Teensy was designed to be twofold. On one hand, the Teensy needed to hold the low-level control, i.e. joint level impedance controller. On the other hand, we designed the Teensy to bridge communication between the high-level controller and the motor controllers.

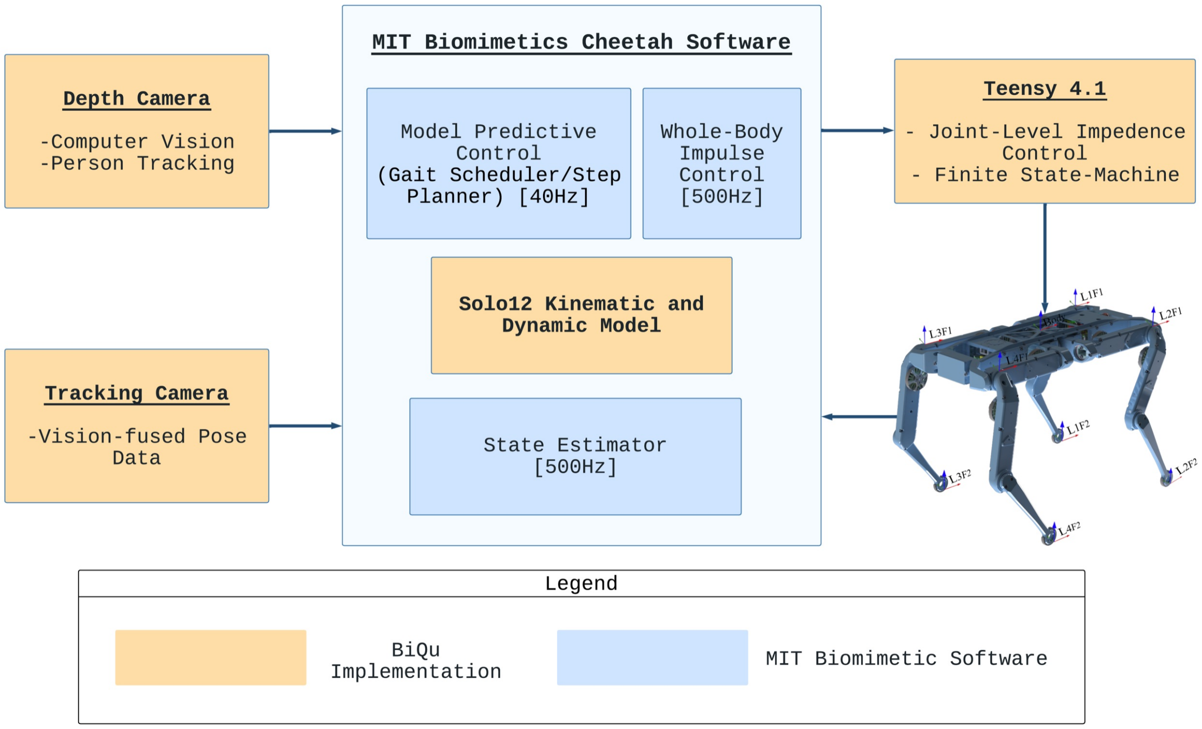

MIT Biomimetics Cheetah Software
Depth Camera
-Computer Vision
-Person Tracking
Model Predictive Control (Gait Scheduler/Step Planner) [40Hz]
Whole-Body Impulse Control [500Hz]
Teensy 4.1
- Joint-Level Impedence Control
- Finite State-Machine
Solo12 Kinematic and Dynamic Model
Tracking Camera
-Vision-fused Pose Data
State Estimator [500Hz]
Legend
BiQu Implementation
MIT Biomimetic Software

Figure 10: Software and control architecture design using Cheetah Software existing modules (in blue) and our modifications and additions (in orange). Low-level controller in Teensy 4.1 also included.

One major aspect of our work was to design an efficient communication pipeline. As mentioned before, our high-level control ran on a Raspberry Pi, the joint-level impedance controller ran on the Teensy, and the motor control was done by the ODrives.

The output of the high-level controller went into the low-level controller as desired joint states and PD gains. More precisely, these were the desired joint positions, desired joint velocities, feedforward torques and PD gains for the joint-level impedance controller. Given our hardware architecture, we needed to transmit this data, which we called "spi_command(s)", from the Raspberry Pi to the Teensy 4.1, since the Teensy held the low-level controller and interfaced with the motor controllers. The Raspberry Pi also needed to get from the Teensy (and the Teensy from the motor controllers) the feedback joint states for its own state estimation module. We simply called this data "spi_data", i.e. feedback joint positions and feedback joint velocities from encoder data. From the desired and actual (feedback) joint states, we computed the desired motor torques, which needed to be sent to the ODrives for motor torque control. The main requirement for this pipeline was that it needed to run at the main controller's frequency: 500Hz, which was 2 millisecond per control loop.

For some context, the joint level impedance controller, or stiffness controller, uses the following equation [18]:

$$\boldsymbol{\tau}_{out} = \boldsymbol{K}_p(\boldsymbol{q}_d - \boldsymbol{q}) + \boldsymbol{K}_d(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}) + \boldsymbol{\tau}_{ff} \tag{6}$$

where $\boldsymbol{K}_p$ and $\boldsymbol{K}_d$ are proportional and derivative gains, $\boldsymbol{q}_d$ and $\dot{\boldsymbol{q}}_d$ are desired joint position and velocity, $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ are actual (feedback) joint position and velocity, and $\boldsymbol{\tau}_{ff}$ is the feedforward torque.

Figure 11 shows our design for the flow of the Teensy program in the form of a flowchart. The main loop is always checking if process_bytes is true; initially it is false. To program will wait for the SPI interrupt service routine that is triggered every time a new byte is received. If the SPI ISR is triggered, we receive the SPI command byte and store it in rx buffer (i.e. a receive buffer) and then we transmit SPI data bytes at the same (response of SPI). We then increment the bytecount by 1. If the bytecount is greater than MAX_BYTES — that is, if we have received the full SPI command struct — we set process_bytes to true. We exit the ISR and go back to the main loop. Now that process_bytes is true, we first process the rx buffer bytes to convert to desired joint states struct. We then get the feedback joint states and load the tx buffer with the feedback joint data (for SPI response in next ISR sequence). We compute the desired joint torques. At this point, we send the joint torque control command to ODrives motor controllers, and set process_bytes to false and bytecount to 0. We repeat the above loop every time we receive new data from the SPI Master (Raspberry Pi). The above flowchart was implemented as described in the Implementation section.

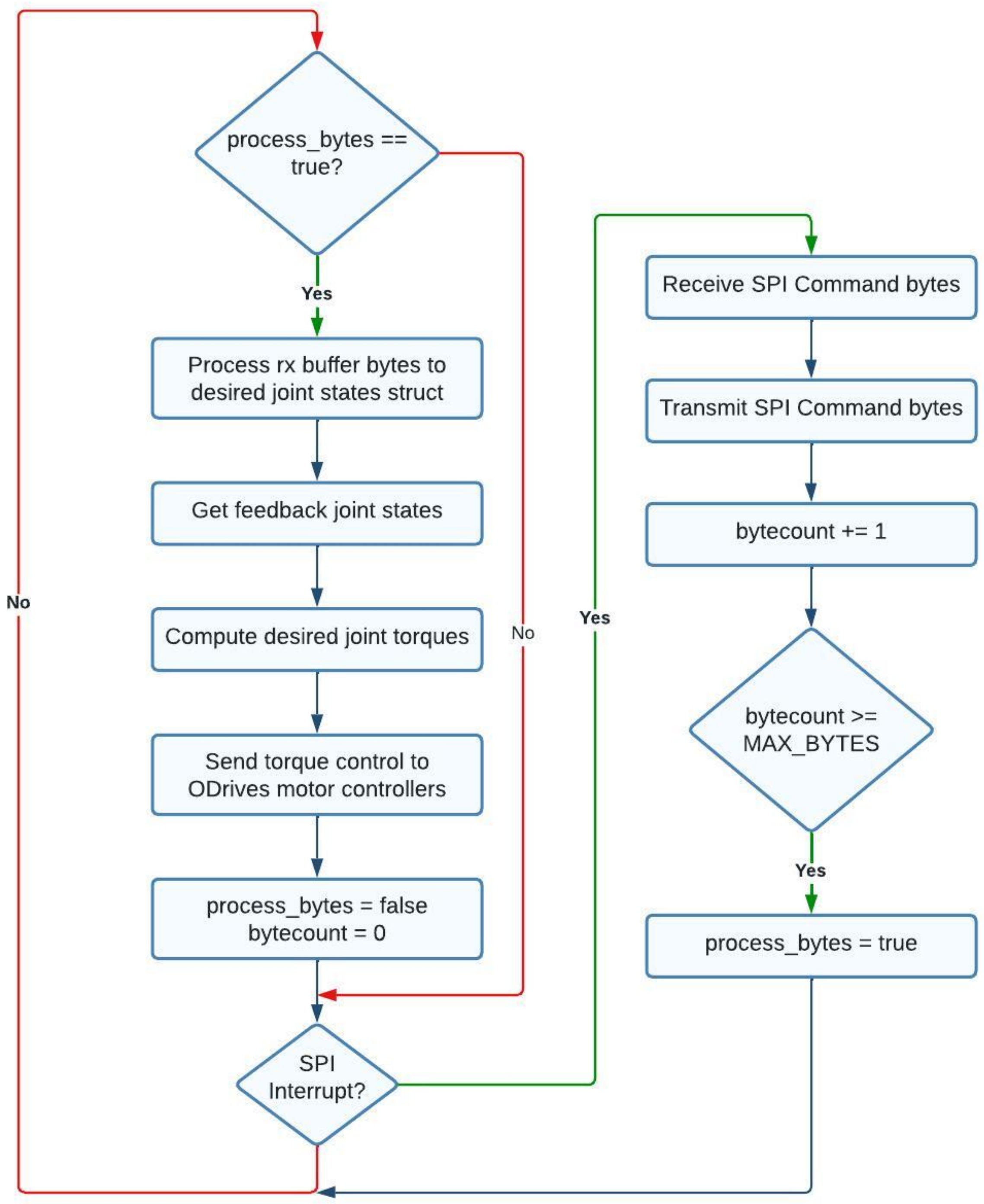

process_bytes == true?
Yes
Process rx buffer bytes to desired joint states struct
Get feedback joint states
Compute desired joint torques
Send torque control to ODrives motor controllers
process_bytes = false
bytecount = 0
No
No
SPI Interrupt?
Yes
Receive SPI Command bytes
Transmit SPI Command bytes
bytecount += 1
bytecount >= MAX_BYTES
Yes
process_bytes = true

Figure 11: Flowchart for Teensy program (low-level controller). The right-hand section shows the sequence inside the SPI interrupt service routine (ISR) and the left-hand section shows the sequence inside the main loop whenever there is new commands to process.

3.3 Computer Vision

Implementing following a person on a quadruped robot requires creating a vision system for the robot to be able to send joint commands to its motors in order to make appropriate adjustments to its direction. This can be broken up into two parts, which are object tracking and object detection. In our implementation, we utilized the D415 camera for object tracking, specifically to track a person's location. The D415 camera, produced by Intel RealSense, is a depth camera that is capable of capturing depth information along with RGB data, making it suitable for accurate tracking of objects in 3D space [19].

3.3.1 Single Object Tracking

Object tracking or visual tracking refers to the idea of locating an object in successive frames of a video and estimating the object's motion. Our goal is to implement this feature on the robot to be able to move in the direction of another person that it is tracking. Object tracking not only is used to locate an object but also can be used to estimate speed and predict direction. In our tracking algorithm, we hope to focus on single object tracking, where we only need to track an object of a single class, which is people. With object tracking, there is much research out there on many visual tracking algorithms such as TLD, MEDIANFLOR, MOSSE and much more. Although, most do not take into account occlusion. In our tracking algorithm, we implement the Minimum Output Sum of Squared Error Filter (MOSSE). To explain the tracking algorithm, first, we will discuss ways we can detect the person. Then we will generalize the steps of object tracking.

3.3.2 Object Detection

"Detection-based algorithms estimate the object location in every frame independently" [20]. Object tracking is much faster than object detection because in single object based tracking, it is building the appearance model as it goes, and is only looking for the object around where it was previously. Whereas in object detection, the model has to be trained to know what object to look for and be able to search the entire window for it. The problem of object detection is a typical problem in computer vision. You Only Look Once (YOLO) is an algorithm that labels objects, fits them in bounding boxes, and tracks them across time. YOLO is currently the fastest (45 frames per second) and the most reliable algorithm. We performed an inference with a pre-trained version of YOLO (YOLOv5) on a video of the WPI campus. YOLOv5 reliably detected individuals, phones, and cars, but other objects like trees were ignored. As we are looking for an algorithm that solely solves the issue of object detection, YOLO seemed like killing a mosquito with a sledgehammer; for the purposes of this project, we do not need the detected object to be labeled while YOLO outputs labeled objects. So, for optimization reason, we looked for alternatives to YOLO.

Alternatively, we tested a R-CNN and considered Fast R-CNN and Faster R-CNN. R-CNN classifies a large number of region proposals (in the order of 1000s) using a selective search algorithm [21]. R-CNN was slow and did not look promising for real-time object detection. But compared to YOLO it detected smaller objects. Fast R-CNN is an improvement on R-CNN as it reduces the number of times region proposals have to be fed to the model. The testing time of Fast R-CNN is about 2.3 seconds. Faster R-CNN is faster because it does not make use of region proposals and goes through the image once [22]. Faster R-CNN, which has a testing time of 0.2 seconds, is twice as fast as Fast-CNN and about 50 times as fast as R-CNN.

Faster R-CNN is fast but not a real-time algorithm, therefore, it is not suited for this project. Since objects need to be detected in real time, we will optimize YOLOv5 for the purposes of this project. We will remove the part of the model that labels the objects and fine-tune it on our campus dataset. We chose YOLOv5 because it is a stable version of YOLO known to be accurate. YOLOv5 implementation in PyTorch facilitates its integration into our software architecture.

Although we initially considered using YOLOv5 for our vision task, we found it to be too complex and computationally demanding for our intended use on a Raspberry Pi. As such, we opted for a simpler approach to detect people in the scene given the limitations of the Raspberry Pi's computational power and the scope of our task. As a result, we decided to use Histogram of Oriented Gradients (HOG), which is a less computationally expensive technique for person detection.

Histogram of Oriented Gradients (HOG) is a feature extraction technique used in computer vision and image processing to represent the local orientation and gradient information of an image [23]. HOG works by dividing an image into overlapping cells, computing the gradients of pixel intensities within each cell, and then constructing a histogram of gradient orientations. These histograms are then concatenated to form a feature vector that represents the image. HOG has been widely used in object detection and recognition tasks, including human detection, pedestrian detection, and face detection, due to its robustness to changes in lighting and appearance, and its ability to capture local gradient patterns that are used for object recognition. This feature vector will be used in machine learning algorithms, such as support vector machines (SVM), to train a person detection model. The trained model can then be used to detect a person in an image by sliding a window across the image, computing HOG features for each window, and classifying the window as containing a person or not based on the trained model [23].

General Steps of Object Tracking

1. `Target Initialization`

2. `Appearance Modeling`

3. `Motion Estimation`

4. `Target Positioning`

3.3.3 MOSSE

Once the person is detected in the frame, we must begin tracking. The first step of any object-tracking algorithm is tracking initialization. Initialization is determining our target which involves drawing a bounding box around it. In MOSSE, this involves cropping an image of the object from the frame for object labeling. The bounding box will be pre-processed and used as our tracking window [24]. We create a synthetic target that generates the desired correlation output and displays the center of the object to locate where the next bounding box should be.

Next, we need to model the appearance of the target object to be able to locate it in the next frame. We do appearance modeling because the object is not going to look perfectly the same in different locations pixel by pixel. These are some of the challenges when dealing with object tracking we must account for illumination, occlusion, deformation, noise corruption, rotation, and speed. We use MOSSE correlation filters to account for this. In MOSSE, this involves taking the cropped image and analyzing the image in the Fourier domain. Appearance modeling involves understanding the features of the object and uses statistical modeling to identify an object effectively [25].

After we have properly modeled the appearance of the object, we will want to estimate its motion. The Motion estimates help us understand the region where we should be searching for the object. Once the possible regions of the location of the object are found, we can look through them and find the object's exact location.

In MOSSE, we can take the output of the correlation from the Fourier domain and convert it back to the spatial domain to find the exact location of the object. Following the guided tutorial how the MOSSE algorithm is created we can find the exact location of the object because, "correlation produces a new image with the dot product for every alignment. The peak in the correlation image is used to find the location of the template in the image" [24].

4 Implementation

4.1 Hardware

For this project, our goal not only includes resolving hardware problems on the previous Solo8 robot, but also focuses on expanding the capabilities of the robot with upgraded hardware. This section details the process of fabricating and assembling the new Solo12 robot.

4.1.1 Fabrication

As mentioned in section 3.1.1, we have decided to print the chassis in Onyx to improve the PLA shell melting problem. To minimize cost, we looked into resources on campus that would allow us to use a Markforged Mark Two printer: Academic & Research Computing (ARC), Robotics Department, and PracticePoint. After a few test prints with Mark Two and communications with the staff at PracticePoint, we purchased an 800cc Onyx filament spool and utilized the printer at PracticePoint to print all the shells and chassis parts of the Solo12. With the help of the ODRI community, we printed the parts with 0.1mm layer height, Onyx material without fiberglass, solid infill, and brim support. Figure 12 shows some of the printed shells. Each actuator module is made up of a shell and a base. Combining the chassis and body structure parts, a total of 45 prints were printed.

Another component that requires manufacturing is the transmission pulleys used in the actuator modules. The pulleys should have high strength and high heat tolerance to prevent deformation and melting. The Solo12 design by ODRI recommended printing in the material Accura Xtreme using a Stereolithography (SLA) printer. The team was not able to find any SLA printing resources on campus and had to look into other materials. Since the tooth profile of the pulley must be precise, concentric, and has minimal visible layers, we could not use a Fused deposition modeling (FDM) printer. We reached out to ARC on campus and learned that we can experiment with the Formlabs Tough Resin, another resin with high impact strength. We researched more on the feasibility of printing the pulleys in Formlabs Tough Resin instead of Accura Xtreme: Both resins are ABS/Polypropylene-like. Formlabs Tough Resin can be printed using a desktop SLA printer, while Accura Xtreme can only be printed with an industrial SLA printer. This means that the cost of printing with Formlabs Tough Resin (\$15) is significantly lower than printing with Accura Xtreme (\$200). The team decided to utilize the ARC printing service and produce the pulleys in Formlabs Tough Resin. However, we were able to obtain a partnership with our sponsor, ProtoLabs, who sponsored to print off the pulleys in Accura Xtreme at their facility.

The transmission pulleys require an extra step of modification in order to fit with the bearings. The printed pulleys have an outer diameter of 25.08mm, while the inner diameter of the bearing is 25mm. We used the CNC lathe at the Washburn Shops to reduce the diameter to about 24.99mm.

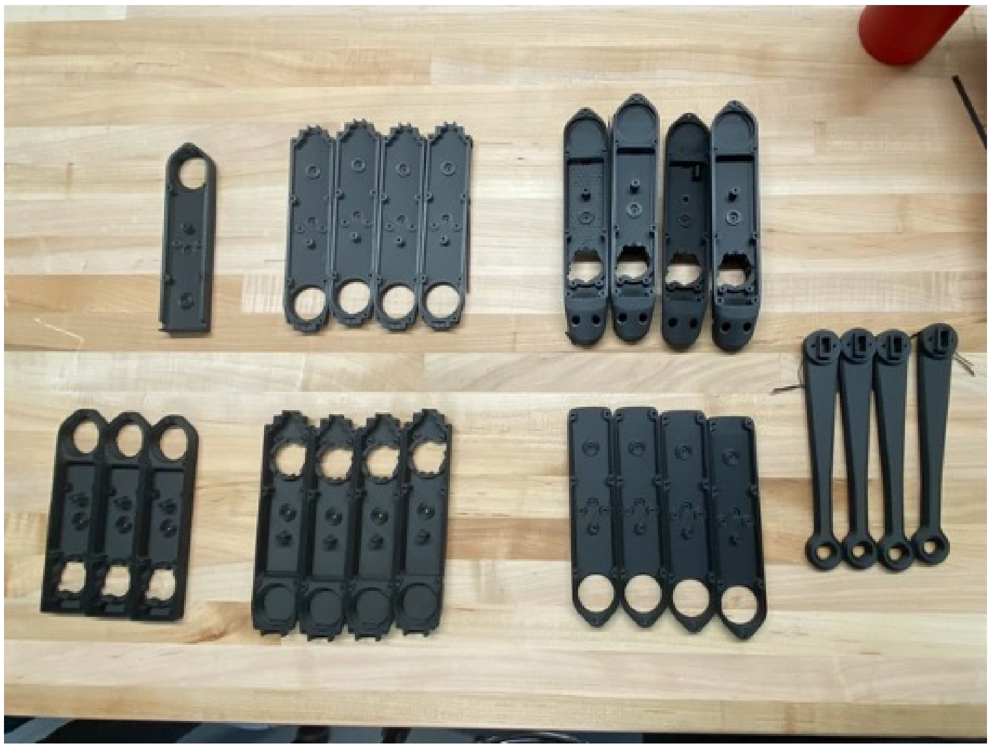
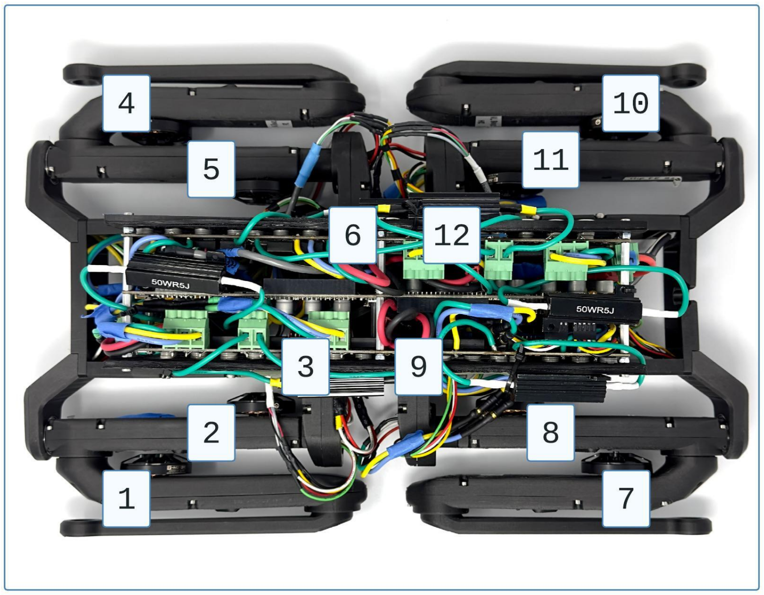


Figure 12: Actuator Module Shells printed in Onyx

4.1.2 Assembly

After collecting all the components for the actuator modules: disassembling the Solo8, purchasing parts, and manufacturing the shells, we followed the instructions provided by ODRI to assemble the robot. The Solo12 consists of 12 actuator modules, labeled as follows in figure 13.



1	Front Left Upper Leg (FLUL)
2	Front Left FE (FLFE)
3	Front Left AA (FLAA)
4	Front Right Upper Leg (FRUL)
5	Front Right FE (FRFE)
6	Front Right AA (FRAA)
7	Hind Left Upper Leg (HLUL)
8	Hind Left FE (HLFE)
9	Hind Left AA (HLAA)
10	Hind Right Upper Leg (HRUL)
11	Hind Right FE (HRFE)
12	Hind Right AA (HRAA)

Figure 13: Labeled Actuator Modules

Actuator Modules The differences between our Solo12 robot and the open-sourced design are the wires. Since we are using the ODrive motor controllers instead of custom micro drivers, we use 22AWG 300V Hook Up wire with a five pins header for the encoder, and 16AWG 600V Silicone Tinned Copper wire with a 4mm connector for the motor. To connect to the ODrive motor controller, the wires are arranged in the order as shown in figure 14. The new encoders that we purchased follow the former color scheme, while the encoders on the previous Solo8 follow the latter color scheme. During the disassembly of Solo8 and the assembly of Solo12, we constantly run into problems where wires are disconnecting at the joints. As shown in figure 15, despite having the protection covering

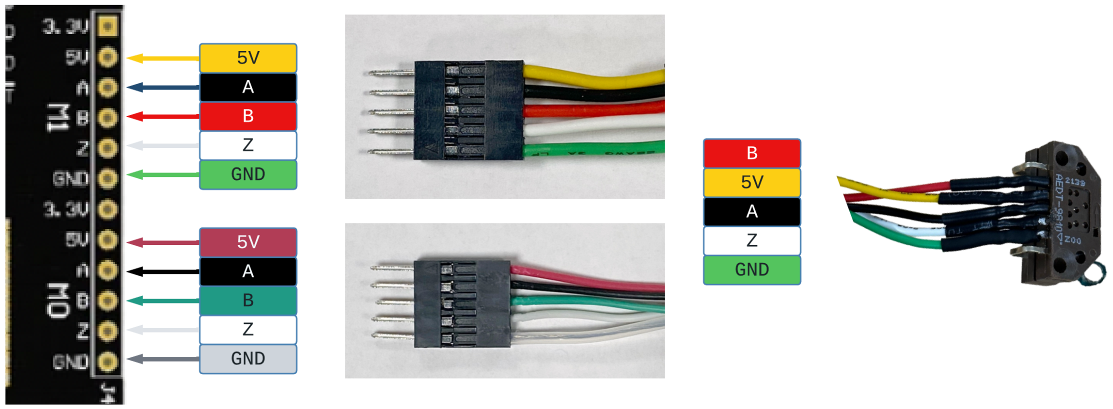


Figure 14: Encoder Wiring Diagram

of heat shrink tubes, the wires broke off at the soldered wire joint that connects the motor phase wires and the extension wires. With the soldered joint hidden under the heat shrink tube, we could only test for continuity during the calibration process. We learned that an indicator that the motor wires are broken is the motor going back and forth and producing a clicking noise during the calibration process. We figured that two factors contribute to the wire breaking: too much tension from bending the wires and solder not secured on the motor phase wires due to a layer of insulation on the wires. Each actuator module has a different shell structure. To avoid bending the motor wires at the joint, we either extend or shorten the motor phase wires so that the joints stay within the shell. Also, the solder does not stick to the insulated motor phase wires, therefore, we used the soldering iron and solder to remove the insulation to expose the silver wire underneath. To minimize tension from bending the wires to fit through the hole in the shell, we used a Dremel tool to open another opening to let the wire through.

Another issue that we keep running into is the code wheel detaching from the motor shaft. The past MQP team purchased the code wheel and motor shaft separately, which require them to use super glue to attach them. In the process of disassembling or installing the shaft, the code wheel will fall off from the mount, as shown in figure 16. We had to glue them together again, which started causing issues such as the code wheel not

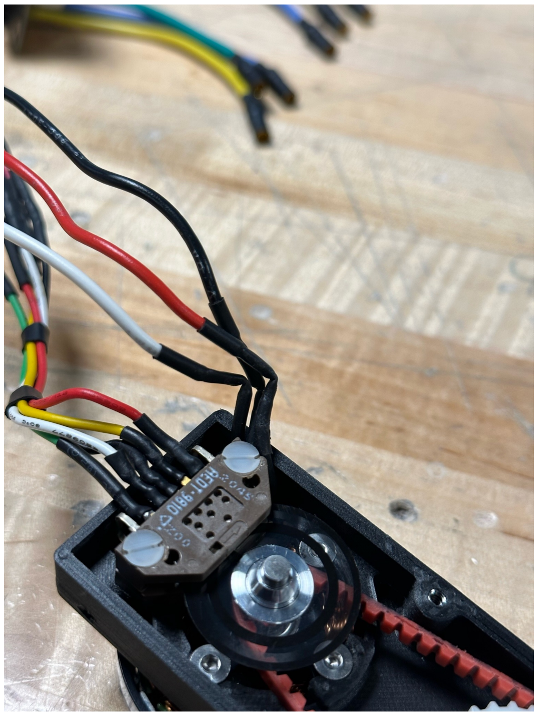


Figure 15: Motor Phase Wire Breaking

sitting flat on the mount due to hardened glue residue or glue stains on the code wheel. Unsure of how the stain or height offset will affect the optical encoder readings, we made several attempts to clean the code wheel with alcohol wipes. Fortunately, we have four extra code wheel kits which came preassembled with metal mounts and were able to replace a few stained code wheels. A fully assembled actuator module is shown in figure 17.

Actuator Module Testing. After we assembled 12 actuator modules, we went through multiple iterations of actuator module testing. We used the *odrivetool*, a PC program that allows us to control the ODrive motor controller via the command line. The *odrivetool* allows us to run calibration, configure parameters, and check encoder counts. We identified three common behaviors of the motor when the calibration failed: Motor moving back and forth with clicking noise, motor complete counterclockwise calibration but does not return to the home position, and motor moving clockwise.

As mentioned in the previous paragraphs, the back-and-forth motion of the motor indicates a discontinuity in one or more motor wires. In our testing, four actuator modules

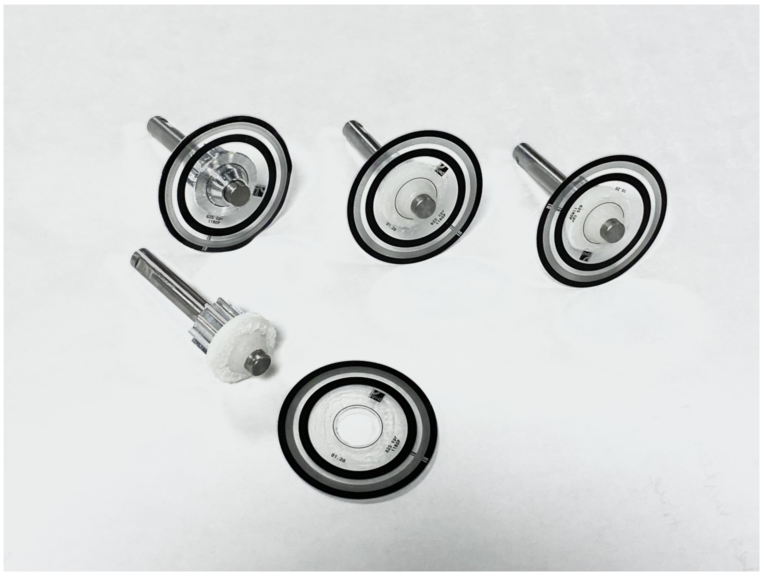


Figure 16: Code Wheels

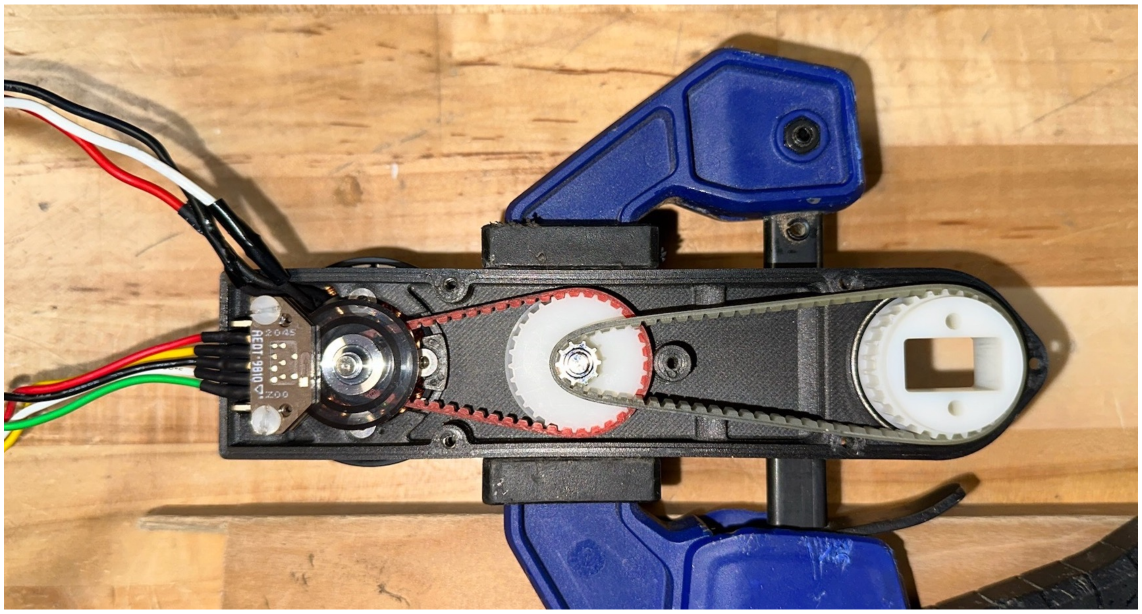


Figure 17: Actuator Module Assembly

encountered this issue: HR Hip AA, HL Hip AA, FL Upper Leg, and FR Hip FE. We were able to fix them by resoldering. Two of the actuator modules displayed the behavior where the motor would rotate counterclockwise but fail to return to the home position. After multiple attempts at repairing, we identified two causes that led to this motion. Since the motor was able to rotate for half a revolution, we figured the problem lies in the encoder.

When one or more encoder wires broke off, or the encoder failed to find the index position on the code wheel, the calibration fails. To resolve this, we first check for the continuity of the encoder wires. Then, we ensure that the encoder index, which is indicated with a blue marker in figure 18, must pass through the encoder during calibration. This consists of figuring out the alignment of the output pulley with the code wheel. The output pulley is connected to the following actuator module, and therefore, its alignment differs in each actuator module. We first set the robot in a home configuration as shown in figure 19, where all legs are pointed straight down, and record the alignment of the output pulley. We then position the output pulley so that when the motor rotates counterclockwise, the code wheel index passes through the encoder. Lastly, two of the actuator modules were rotating clockwise during calibration. We found out that this behavior happens when the order of motor wires is swapped. Swapping the motor wire resolved the problem.

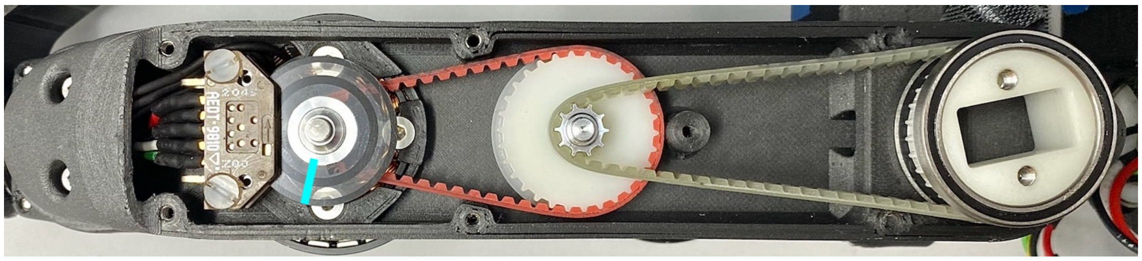


Figure 18: Code Wheel Index Position (Indicated by blue marker)

3-DOF Legs. After making sure each actuator module calibrates successfully, we combine the three modules to form a leg. We encountered another issue here: the wires from the Upper Leg module have to go through the FE module, and the soldered and crimped connectors are not small enough to fit through. Apart from using a Dremel to expand the openings, we also had to desolder and detach the connectors. With the four legs assembled, we were able to put together the robot chassis, as shown in figure 20.

Electronics Housing. Next, we have to house all the electronics onboard. The Solo12 design utilized the space inside the chassis to house the master board and the micro motor drivers. However, due to the size of our ODrive motor controller, we have to place the

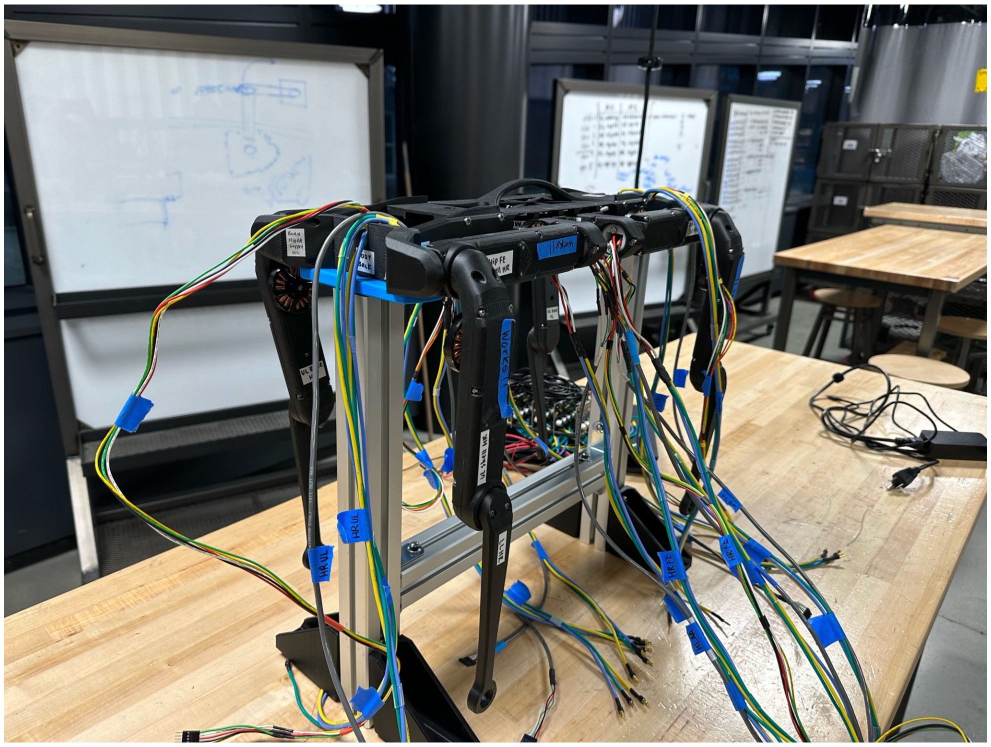


Figure 19: Preassembled Robot in Home Position

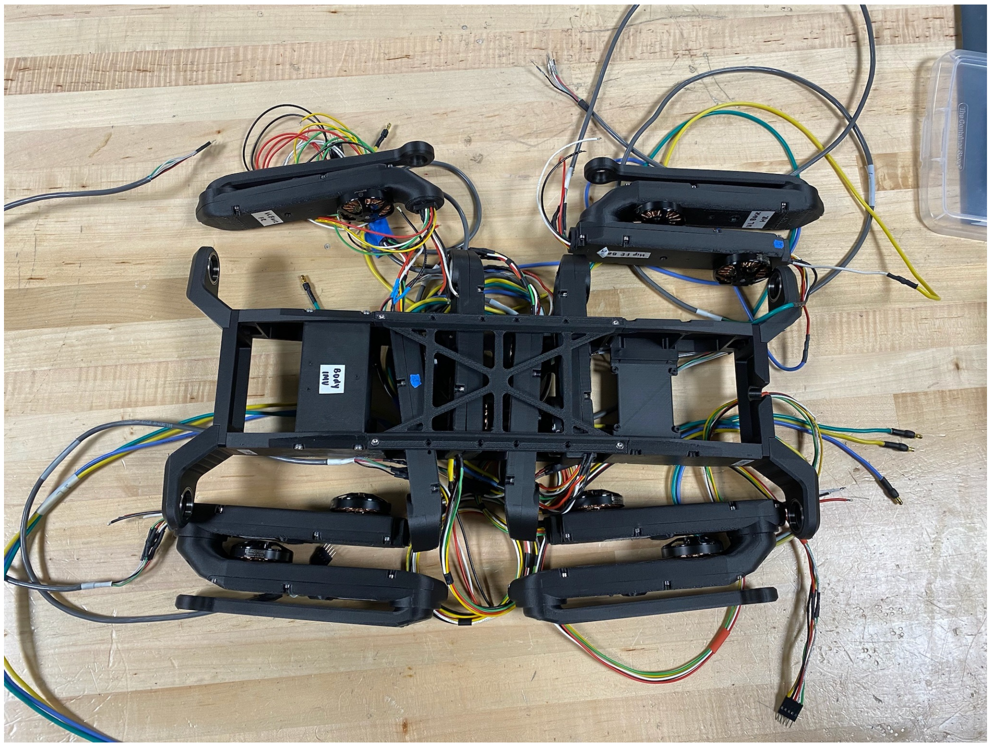


Figure 20: Preassembled Robot in Top View

six ODrive motor controllers on top of the chassis. The top plate is modified into a U-shaped container that will hold the six ODrive motor controllers in place. The first design is shown in figure 21, the three parts, which are printed in PLA using the Ultimaker 3 printer at the Prototyping Lab, have to be held together by epoxy. After assembling the electronics with the housing, we wanted to improve the design so that it is serviceable, meaning that we can take it apart for debugging purposes. We modified and came up with a second design. Figure 22 shows the assembled robot chassis with the first electronics housing design.

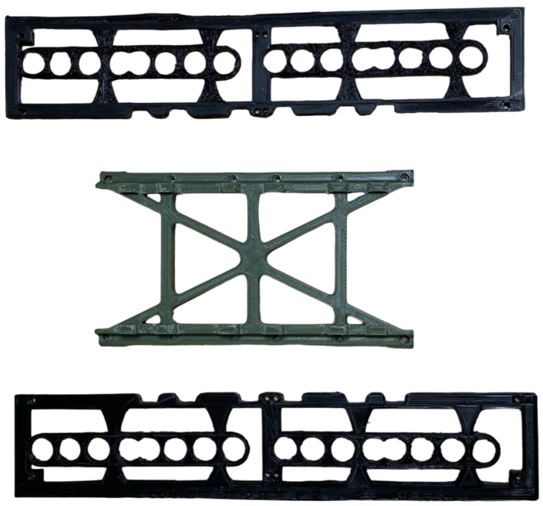


Figure 21: First Design of Electronics Housing

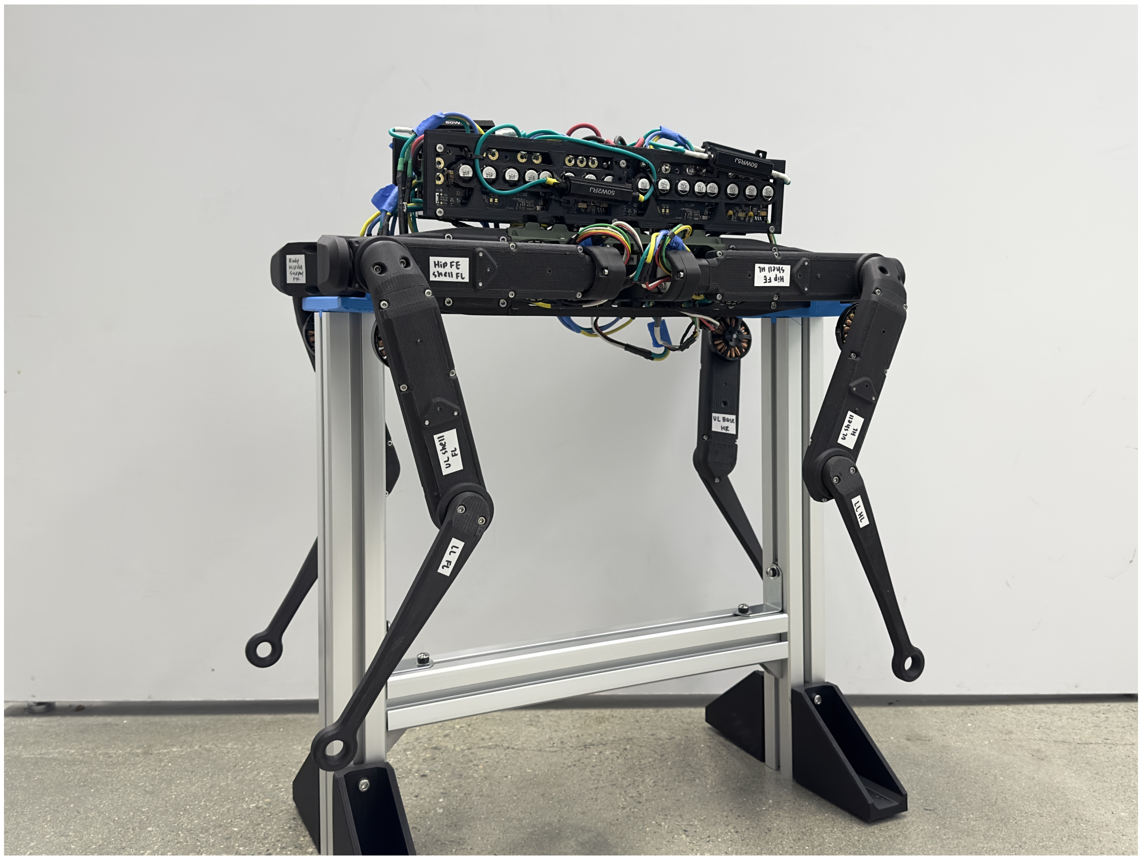


Figure 22: Solo12 with the First Electronics Housing Design

The second design of the electronics housing, as shown in figure 23, consists of the following changes: the parts are held together using 20mm fasteners, and a handle can be attached at the top for carrying the robot. For the second design, the components are printed in Onyx. Figure 24 shows the assembled robot chassis with the second electronics housing design.

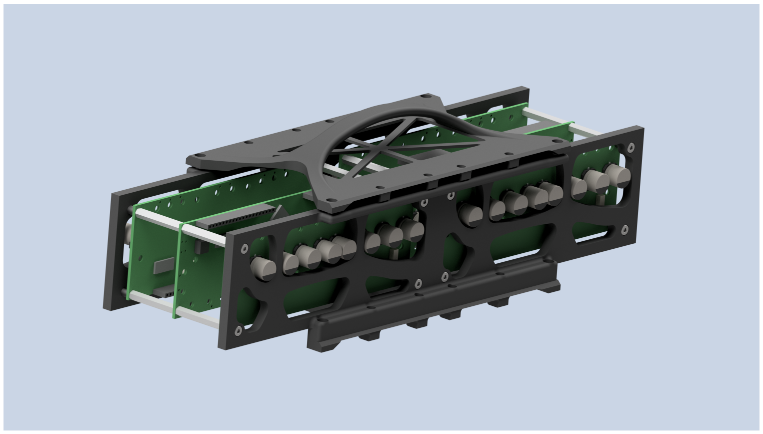


Figure 23: Second Design of Electronics Housing

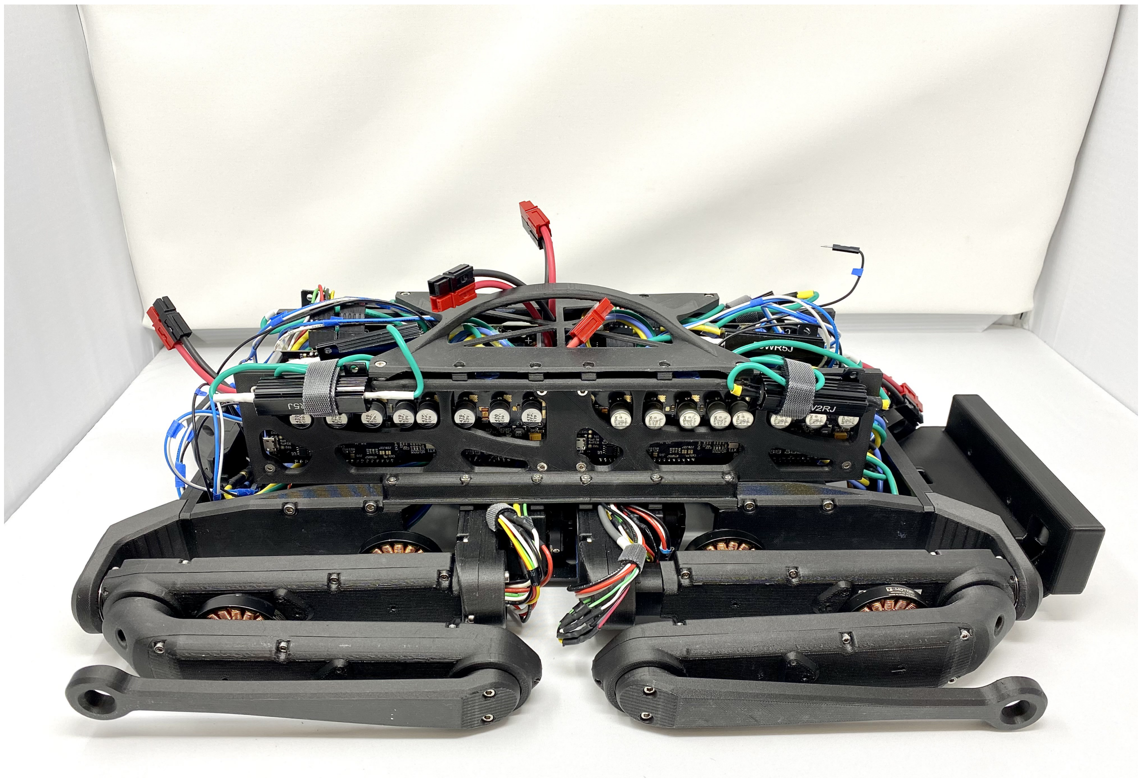


Figure 24: Solo12 with the Second Electronics Housing Design

Camera. We are using two cameras for this project: Depth Camera D415 and Tracking Camera T265. The cameras have to be placed in front of the robot so its view is not blocked. The two cameras are stacked on top of each other and center-aligned. The camera mount shown in figure 25 is attached to the front of the robot chassis and between the FE module. Due to time constraint, we only printed the camera mount in PLA as a test print, and did not attach the camera mount to the chassis.

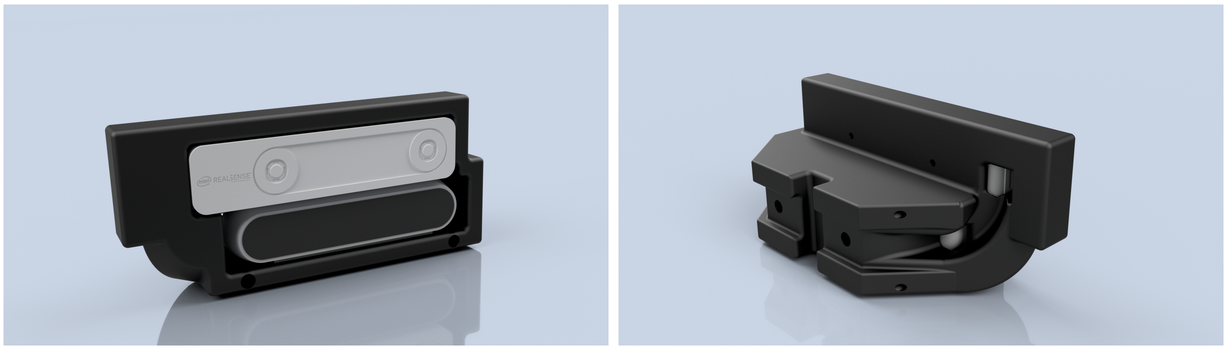


Figure 25: Rendering of Camera Mount

Full Robot Assembly. After putting together all the components, we installed the microcontroller and attached wires for SPI and UART communication. Figure 26 shows the final assembled robot in side view, and figure 27 shows the final assembled robot in top view.

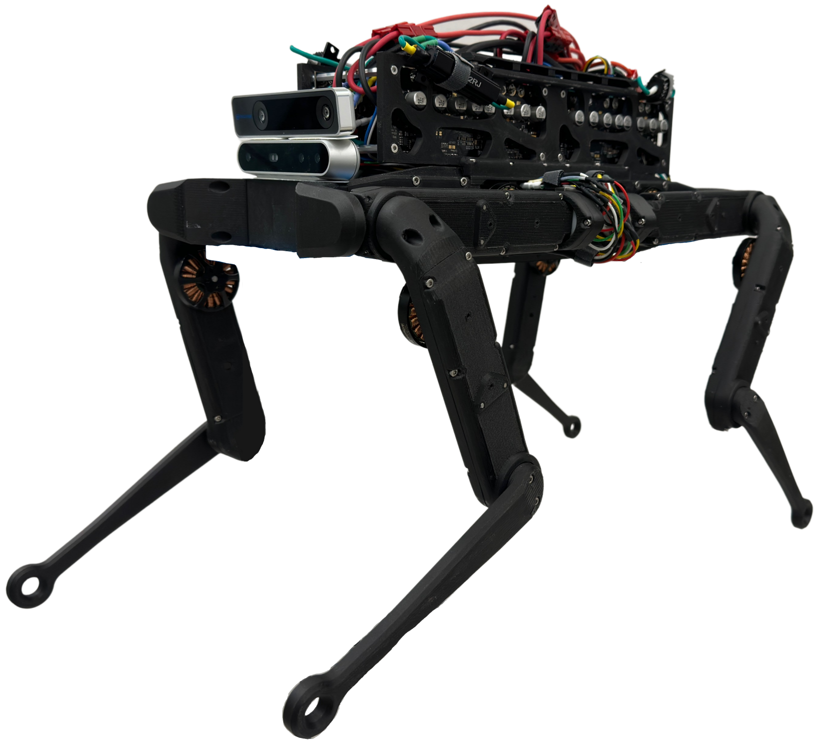


Figure 26: Final Assembled Robot - Side View

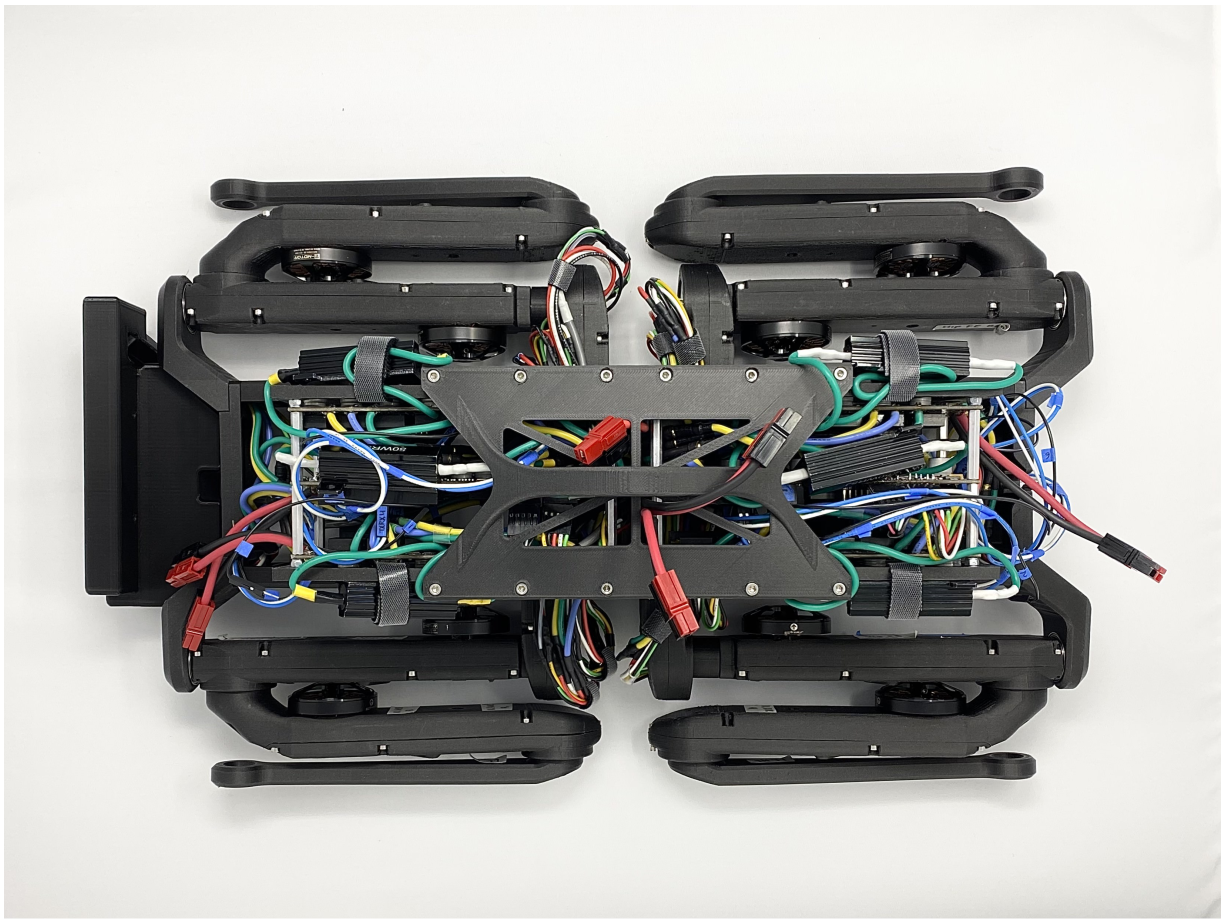


Figure 27: Final Assembled Robot - Top View

4.2 Communication

In the following subsections, we will talk about how we implemented SPI for communication between Raspberry Pi and Teensy, and UART with a custom ODrive firmware for the communication between the Teensy and the six ODrives.

4.2.1 SPI communication between Raspberry Pi and Teensy

The previous team used UART for the communication between Raspberry Pi and Teensy. Our team decided to switch to SPI for this communication for two main reasons. First, SPI is much faster than UART. Second, and more importantly, the Cheetah Software already had foundational code for an SPI driver for communication with microcontrollers dedicated to motor control, for the Spine communication architecture in [26]. All we had to do was to adapt the SPI driver to work with one slave board instead of two. This also meant that we would have to send command messages for all four legs instead of two, which is another modification that we made for the BiQu SPI driver.

When we were implementing and testing SPI communication between the Raspberry Pi and the Teensy, we came across many different challenges. An initial step we did for the Raspberry Pi was to enable the spidev device to enable SPI communication. Next, we had to ensure that the data that the Raspberry Pi sent was correct. We created a test program following the same structure of *rt_spi* class in the Cheetah Software except that it interfaced with a single SPI board, instead of the two that the original Spine communication interface used. One way in which we did initial testing and debugging of the simple test program on the Pi was through an oscilloscope. Figure 28 shows an instance of our team using the oscilloscope to debug the MOSI line of SPI while the Pi sent data over the MOSI line. The bytes that we were sending from our test program matched those of the MOSI line, verifying the right transmission of bytes from Master (Pi) to Slave (Teensy).

While debugging the Slave (Teensy) response, we noticed that we always received four extra bytes, namely 0x0000 and 0xffff. After some research into the SPI Slave library that we were using for the Teensy, we realized that for some reason, the Teensy always sends these extra bytes in the beginning, and that they need to be shifted out. So we modified our implementation on both the Master and Slave sides such that we send dummy bytes in the beginning so as to shift out the extra bytes, before we begin the meaningful transmission of the joint commands structs and joint (feedback) data structs. This is the reason why in our Teensy program, the bytecount starts at -1. The first SPI service routine of a message, since bytecount is -1, the program knows its receiving dummy bytes and ignores the dummy bytes, that is, it does not store in the rx buffer. The variable bytecount is then incremented and from this point onward, the Slave receives meaningful data and stores it in the rx buffer.

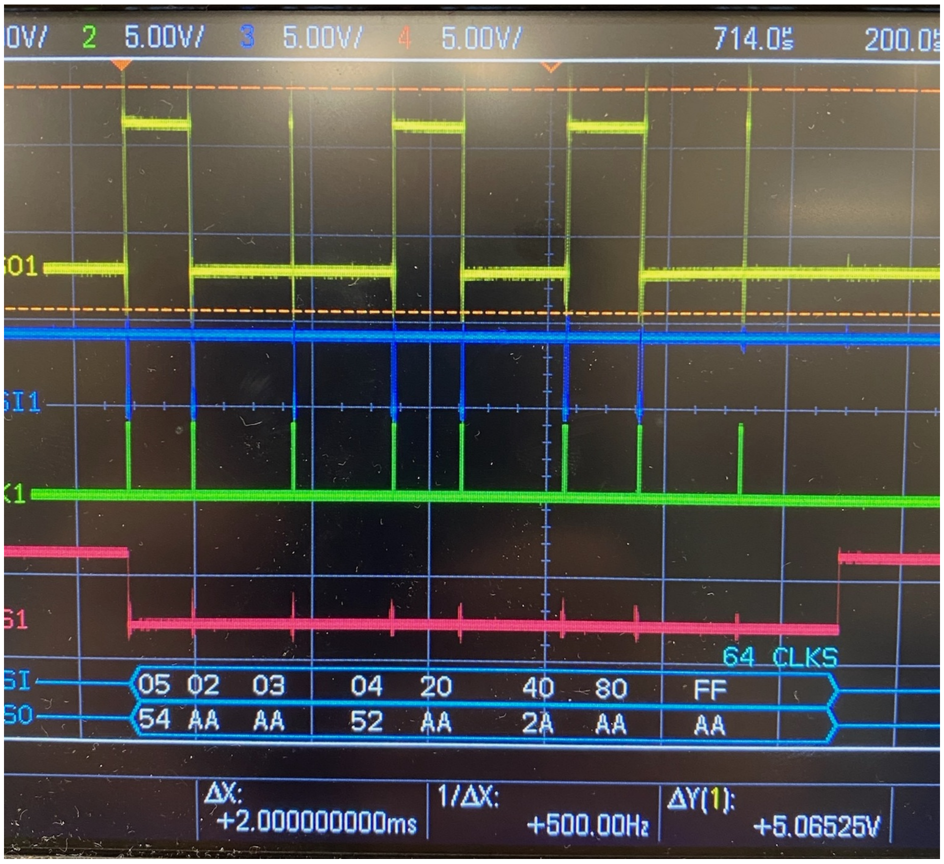


Figure 28: Oscilloscope debugging of MOSI line for Raspberry Pi SPI transmission.

4.2.2 Custom firmware for ODrives

The main high-level controller runs at a frequency of 500Hz, approximately 2 milliseconds per loop. This loop includes communication from Raspberry Pi to Teensy, data processing in Teensy, and communication from Teensy to the ODrive motor controllers. While the SPI communication from Raspberry Pi to Teensy runs at a frequency of 12MHz, the UART communication is taking more time than that. Within one UART communication exchange, the Teensy writes a string of torque commands to the serial port, then writes a string of commands requesting the ODrive to return with joint feedback, and finally reads the serial port to acquire the joint states. We utilized the Arduino interface to briefly measure the duration of this low-level control loop and learned that the process took about 2.5 milliseconds. As we mentioned, the full loop should be completed within 2 milliseconds, therefore, the time taken to complete this portion must be minimized, ideally under 1.5 milliseconds.

We did some research on the ODrive motor controller configuration, and read some forum posts on the ODrive community. We figured we can reduce the time for the low-level control loop with two approaches: Increasing the baud rate and customizing the ASCII protocol to reduce the length of the serial command.

Increasing Baud Rate. We first made changes to the ODrive configurations to increase the baud rate from 115200 to 921600. Despite the concern about dropping bits with a high baud rate, we did not see a significant effect on the transmitted data during testing. We kept the baud rate at 921600 as the changes were able to reduce our loop time by approximately 0.5 milliseconds, from 2.5 milliseconds to 1.9 milliseconds.

Customizing ASCII Protocol. We then modified the ASCII protocol of the ODrive 0.5.1 firmware. Instead of using the default ASCII protocol, we customized the ASCII protocol to fit our usage. When we are sending a command to ask for joint feedback, the default command required an input of motor axis: *'f 0'* for returning the feedback of the motor on axis 0, and *'f 1'* for feedback of the motor on axis 1. For our software, we would always ask for feedback on both axes, which required us to write two separate serial commands and read two separate serial outputs. We modified the ASCII protocol so that we can use *'f'* to ask for feedback on both axis 0 and 1. We then read the string and parsed the output into useful data: *Joint Pos - axis 0, Joint Vel - axis 0, Joint Pos - axis 1, Joint Vel - axis 1.* We also created a batch command that will take a command word and two input torque: *'t input_torque_0 input_torque_1'*, minimizing the number of commands and the number of characters in each command. These changes reduced the duration of the low-level control loop from 1.9 milliseconds to 1.3 milliseconds, achieving a frequency of 770Hz.

Another important implementation detail revolves around how we got feedback on joint states from the six ODrives. The original ODriveArduino interface has **readState** method that calls **readString**, which is a blocking function. Through our testing, we observed that calling **readState** for all six ODrives in the sequence was very slow. It took around 10+ ms to just get the feedback from the ODrives. This did not meet our communication frequency requirement. Therefore, a modification we made is that we instead made our own custom function to get feedback from all ODrives by sending the ASCII batch command and then we read by polling through all ODrives at the same time. The full implementation of the **feedback()** method can be found in our code repository for the BiQu-Arduino-Controller. This new implementation allowed for optimized communication frequency.

One last thing we realized was that the firmware version that we were using still used a UART polling rate of 1 kHz. This meant that the worst-case polling rate for sending or receiving new UART data was 1 ms between each loop. Therefore, we modified the firmware to change the UART polling rate from 1 kHz to 8 kHz. This significantly improved our communication frequency.

4.3 Software

We expanded and adapted the Cheetah Software to work with our robot, making several modifications along the way. The main changes we made included extending the floating base dynamic class and quadruped class to include the kinematic and dynamic model of the Solo12 (BiQu), and modifying the Simulation Framework to include BiQu. Additionally, we developed interfacing code for the T265 camera and integrated it into the software stack for state estimation. We started an initial integration of the D415 camera in the software stack. Another major software development work we did was to develop the low-level controller code and program to run on the Teensy 4.1. Finally, we implemented person-following for the vision module. These modifications and additions were crucial to our project's success, and we will detail the implementation of each one in the following subsections.

4.3.1 BiQu Kinematic and Dynamic Model in Cheetah Software

In the Common library of the Cheetah Software stack, *FloatingBaseModel* is an implementation of a rigid body floating base model class and data structure. The class stores the kinematic tree described in [27]. The tree includes an additional "rotor" body for each body. This rotor is fixed to the parent body and has a gearing constraint. The *Quadruped* class stores all the parameters of a generic quadruped robot and the **buildModel()** method is used to create a floating-base dynamics model of the quadruped. Further, there are utility functions to build Mini Cheetah and Cheetah 3 quadruped objects. Our main work in this section was to build off of this to create a utility function to create a BiQu quadruped object.

The first step in creating this utility function was to create a new BiQu.h file, which holds the BiQu utility function. The inertia parameters of BiQU were determined from CAD; this includes rotor inertias. All (spatial) inertias are taken at the CoM and expressed in the body coordinate system. Other geometric parameters such as body mass, body length, body width, body height, link lengths, locations of all joints, location of rotors, and location of CoM were taken from CAD, existing URDF files of the Solo12, and documentation in the ODRI website [28].

Some other important parameters that we needed to hardcode into the BiQu utility function were those related to the actuators used by our robot, which are the Antigravity MN4004 KV300 motors. These parameters include maximum motor torque, battery voltage, motor torque constant (KT), and motor stator winding resistance (aka. internal resistance). We used the same joint damping and joint dry friction values as those used for the Mini Cheetah and Cheetah 3 utility functions. We found most of the other parameters directly from the Antigravity website under specifications [29]. We calculated the motor torque constant using the following equation [30]:

$$K_t = \frac{3}{2}\frac{1}{\sqrt{3}}\frac{60}{2\pi}\frac{1}{K_v} \tag{7}$$

where K_v is a known parameter with a value of 300. Thus, the resulting torque constant is approximately $K_t = 0.02756\,Nm/A$.

4.3.2 BiQu Simulation in Cheetah Software

One of the major changes we did in Cheetah Software was to expand the Simulator Framework to include the BiQu model. This allowed us to run and test the dynamic locomotion controller on BiQu in simulation as well as to verify its payload capacity based on the dynamic model. Next, we will give some context to the Simulation Framework library and mention some of the implementation details for adding BiQu simulation model.

For some context, the *Simulation* class is the top-level control of a simulation. A simulation includes 1 robot and 1 controller. It does not include the graphics window. The

SimControlPanel class is used for the QT graphical user interface for the simulator. The *Graphics3D* class is the visualizer window for the simulator and implements scroll/pan/zoom. The *DrawList* class stores all the data (except for joint positions) for the robot and is able to load different quadruped robots from the file.

The class that was modified the most was DrawList, where we added a **addBiQu** method. The method loads the BiQu model and builds the draw list. It returns an index number that can later be used to update the position of the robot. Here, for the visualization, we set link offsets, and link colors, and set some other settings like canBeHidden. We also modified the **loadFiles** method to load the BiQu specific OBJ files, including biqu_body.obj, biqu_abad.obj, biqu_upper_link.obj, and biqu_lower_link.obj. These files were generated in SolidWorks by using the Parts to OBJ file converter.

4.3.3 Interface T265 Camera with Cheetah Software

First, we created a *TCam* interface, which can initiate a connection with a T265 camera. It also has a **get_pose** method, which gets the latest pose data from the camera and stores it in the*Pose* struct. This interface includes and makes use of the *librealsense2* library. The next step we took was to integrate and use the TCam interface in HardwareBridge. We created a thread, *_poseThread* that runs **runTCam** in HardwareBridge to update camera pose data. We created an object camVectorNavData that holds the latest camera pose data. This object is passed down to different module include the StateEstimatorContainer and RobotRunner. From here, we created two new state estimators classes, namely *CamOrientationEstimator* and *CamLinearKFPositionVelocityEstimator*. These state estimators use the camera pose data to update the state estimation module of the robot. Although we unit tested the camera functionality to output pose data, we were not able to test our software integration that we just talked about because we were not able to fully run the robot due to other challenges and issues, including time constraints and ODrive limitations.

4.3.4 Initial Integration of D415 Camera in Cheetah Software

In our implementation, we utilized the D415 camera that was mounted on the robot to capture video frames of the environment.

Once the person's location was calculated using the object tracking algorithm, which in our case was the MOSSE algorithm, the angle of the robot with respect to the location of the person was calculated. This angle was determined based on the center of the camera's field of view and the location of the person in the captured frames. The BiQu software, which is responsible for controlling the robot's legs, was then fed with the calculated angle as input.

The BiQu software is capable of processing input data and generating joint commands for controlling the legs of the robot. This approach allowed the robot to dynamically adjust its direction based on the tracked person's movement, enabling it to continuously follow the person's location in real-time

To ensure that the commands were being successfully sent to the trajectory planner in the BiQu software, we performed integration testing. We sent fake person detection data to the planner using a thread that constantly updated a variable, which the robot accessed to retrieve new joint commands for controlling its motors. This testing helped us ensure the successful communication and coordination between the object tracking algorithm, the BiQu software, and the robot's motor control.

4.3.5 Low-level Controller Implementation for Teensy 4.1

Our implementation of the low-level controller and Arduino program for the Teensy 4.1 was exactly how we designed it, which we presented in the flowchart in the System Design section. The program leverages the unit test implementations for SPI communication for Raspberry Pi-Teensy communication and custom ASCII UART protocol for ODrive communication, which was explored in Section 4.2.

4.3.6 Person Following

Detection. We can split this person following problem into two parts: person detection and person tracking. The first part of solution is detecting a person. We perform this by using Histogram of Oriented Gradients.

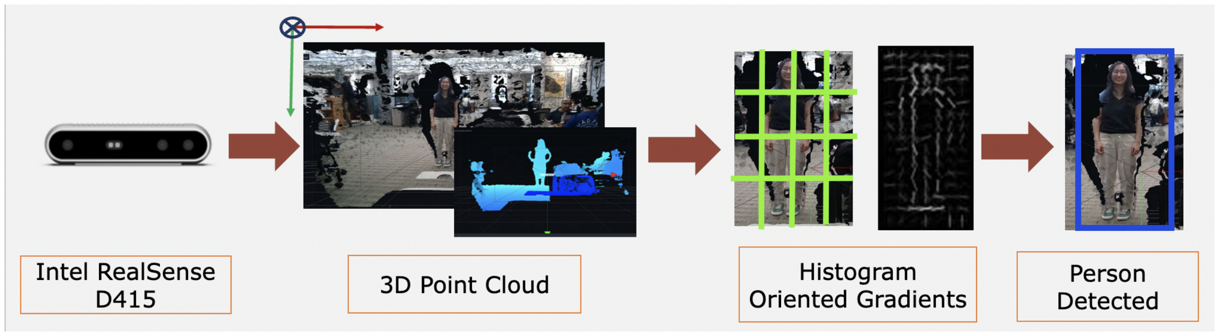


Figure 29: Intel Real Sense D415 Camera Detection

In figure 29, we utilized the Intel RealSense D415 depth camera for the vision component of our project due to the importance of depth data in accurately determining the distance of the target. This camera was selected because one of our project advisors allowed us to borrow it. The depth data captured by the RealSense D415 camera enables the system to accurately perceive the 3D structure of the environment and locate the person.

We performed HOG in order to locate the person. Firstly, image processing needed to be performed to prepare the image such as resizing and normalization. Then we divided the image into overlapping cells and computed the histograms of gradient orientations within each cell. The number of orientations are restricted into bins and the gradients are weighted by their magnitudes.Then in-order to account for illumination and contrast changes in the image, we would perform normalization within each cell.Once the gradients and changes have been accounted for, we then create the feature vectors.This would be performed by concatenating the normalized histograms from all the cells to form a single feature vector that represents the image. Next, we must look for these specific features in the frame and we solve this by using a sliding window. We slide a window across the image and and for each window the HOG features are extracted. To classify whether the window contains a person or not, these features are then fed into a machine learning algorithm, such as a support vector machine (SVM).

MOSSE. Once the person was detected using HOG, the next step was to tracking the moving target. The target box created once the person was detected is pre-processed and set as an initialization image. We used MOSSE algorithm in our computer vision application due to its ability to handle challenges where parts of the object being tracked undergo changes. MOSSE is a robust tracking algorithm that employs adaptive filters to learn the appearance of the target object in real-time. These filters are updated during tracking to account for changes in object appearance due to variations in lighting, scale, rotation, or other factors. MOSSE minimizes the sum of squared error between the filter response and the desired output, making it effective for object tracking tasks in dynamic environments where objects can undergo changes.

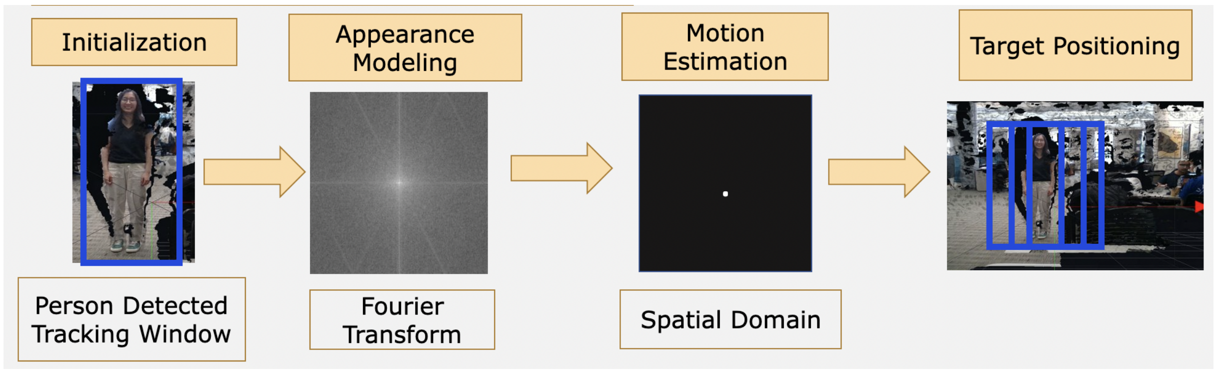


Figure 30: Minimum Output Sum of Squared Error Filter (MOSSE) Tracking

As shown in figure 30, the steps of MOSSE tracking we mentioned before were executed once the person was detected in the frame.

5 Results and Discussion

5.1 Person Following

The MOSSE tracking method, which utilizes online training, has been shown to be more efficient compared to other detection methods that are more costly and time-consuming. MOSSE relies on information from the previous frame to estimate the object's motion and predict its position.

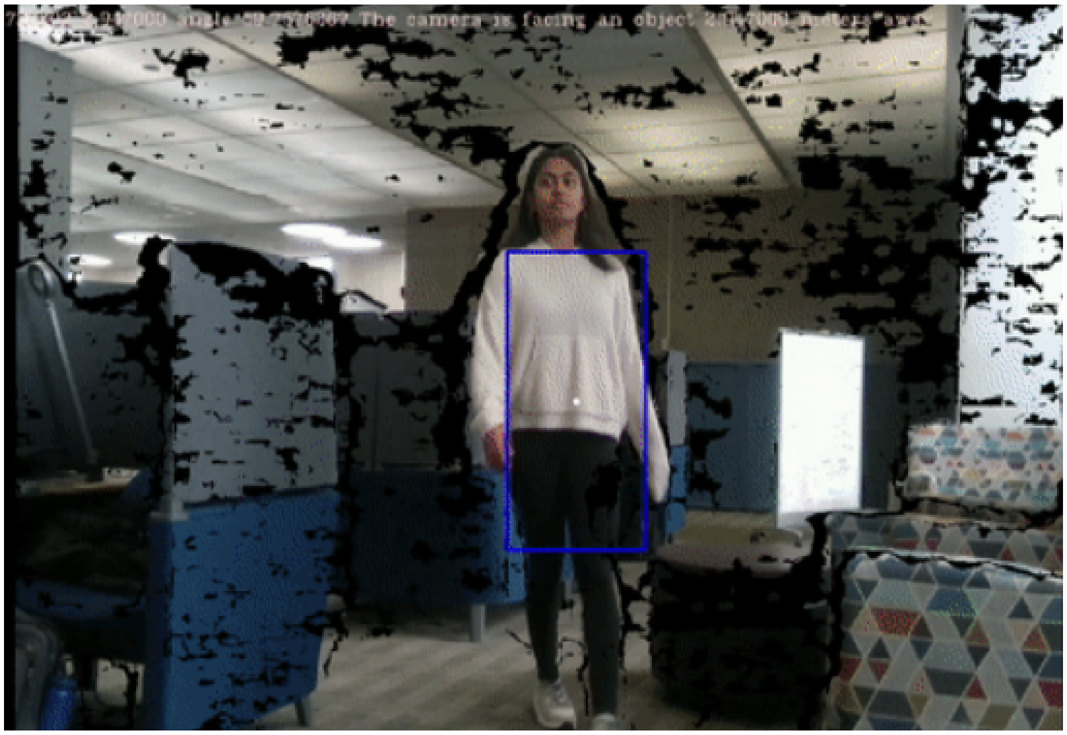
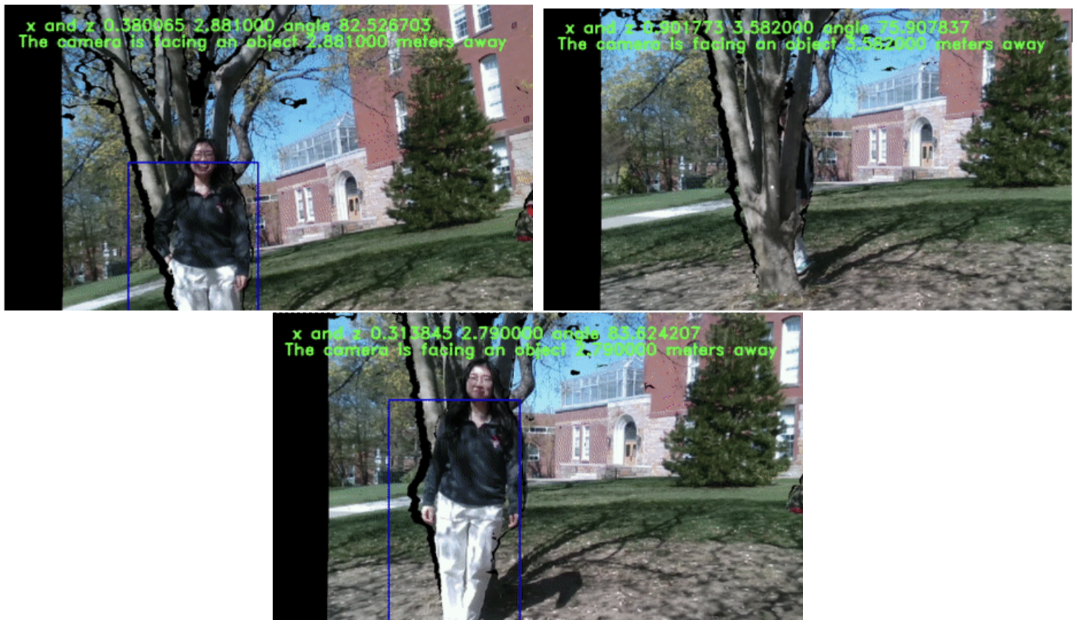


Figure 31: Person Tracking

Observing Figure 32, the camera handles occlusion pretty well and handles tracking at a high speed, making it suitable for real-time applications. The filter is designed to efficiently update the target model and estimate the object's location in each frame, making it capable of handling fast-moving objects, such as people in motion. This allows the algorithm to maintain tracking the object even when it is partially occluded, making it more robust



```
x and z 0.380065 2.881000 angle 82.526703
The camera is facing an object 2.881000 meters away
```

```
x and z 0.901773 3.582000 angle 75.907837
The camera is facing an object 3.582000 meters away
```

```
x and z 0.313845 2.790000 angle 83.624207
The camera is facing an object 2.790000 meters away
```

Figure 32: Target leaves frame and is found again by camera.

compared to other tracking algorithms. It maintains tracking the correct object even when there are changes to the original object in terms of lighting, position, and occlusion.

A number of tests were conducted both outdoors and indoors to observe the different effects of both detecting and tracking. Detection tended to be more difficult when outside giving a rate of 3 false positives ever 5 tests. Tracking tended to be more difficult functioning indoors, giving a rate of 2 false positives every 6 tests. Whereas outdoors, once the person was detected there would be only 1 false positive every 6 tests.

Information of the object's location is present at the top of figure 32. In theory, this data will be passed to the trajectory planning algorithm to pass commands to the motors of the robot.

5.2 Simulation of BiQu Dynamic Locomotion

As mentioned in Section 4.3.2, we extended the Cheetah Software to include the dynamic model of Solo12. We adjusted some parameters and were able to control the Solo12 in the simulation environment. With the correct dynamic model, not only were we able to control the simulated Solo12 with remote control, but we were also able to obtain the joint output torque of the trajectory. We verified the dynamic model by having the Solo12 perform some of the following tasks: Navigating stairs and doing a backflip.

The simulator not only allows us to control the motion of Solo12, but also provides a variety of gait options for experimentation, such as trot and bound. To walk up two consecutive stairs, we tried out several gaits and adjusted gait parameters, for example, the maximum height of the foot trajectory. In figure 33, the robot was able to walk up two consecutive stairs using a trot gait and a foot height of 0.06.

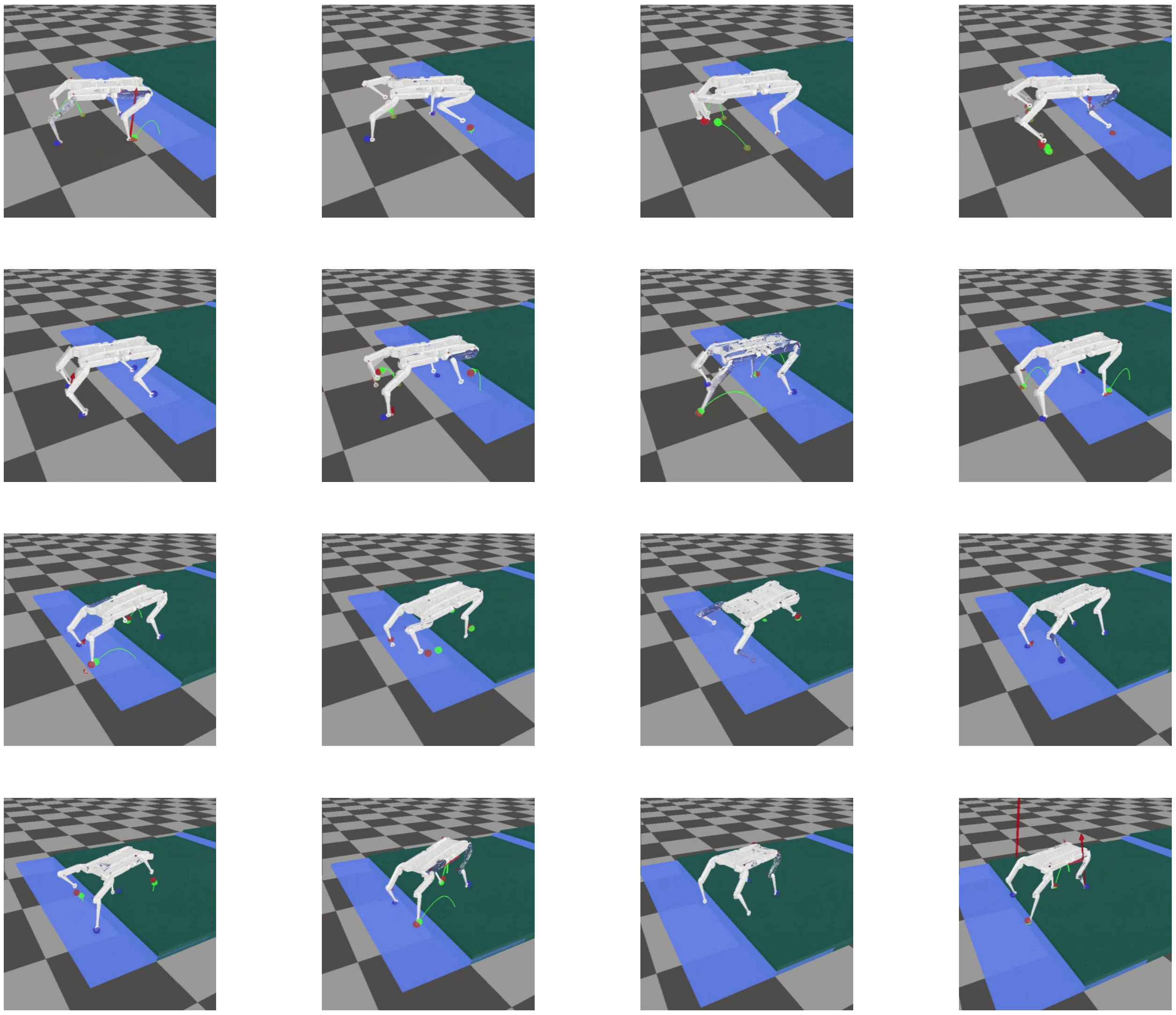


Figure 33: Solo12 Navigating Stairs in Simulation

Figure 34 shows that the Solo12 was able to do a backflip in simulation, which is an indicator of a correct dynamic model. Despite the difference in the electronics housing on the actual chassis and the simulated chassis, we learned that our extension of the Solo12 in the Cheetah Software had a solid foundation that the future team can build on.

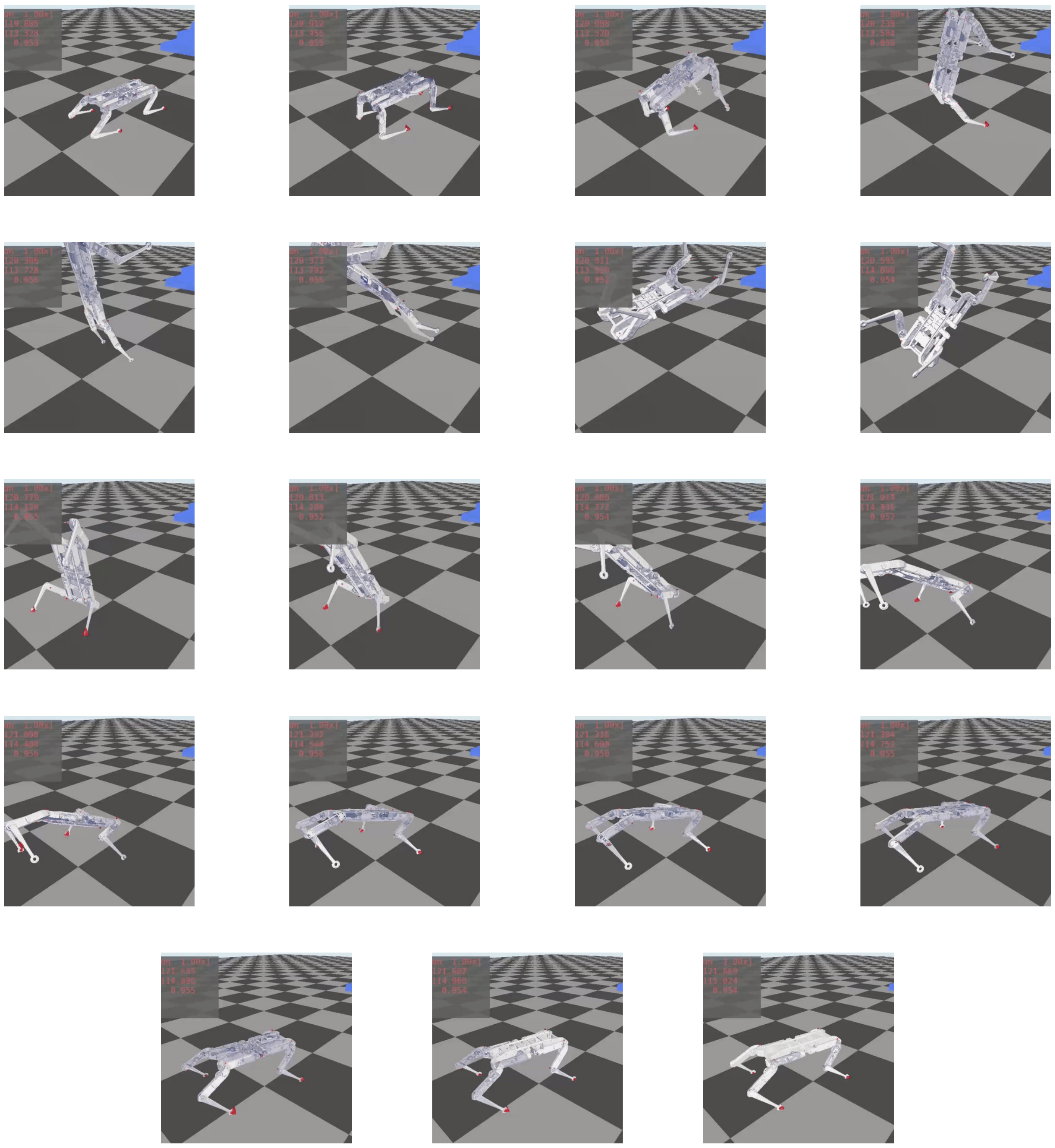


Figure 34: Solo12 Doing Backflip in Simulation

5.3 Single Leg Torque Control for Trot Gait

To test our full communication pipeline, we set up a leg test program on the Raspberry Pi for a trot gait. Figure 35 shows the experimental setup. First, we saved the desired joint state data from a simulation run into a CSV file. We plotted the desired joint positions and velocities for a single leg in trot gait to verify the data collected in the CSV file, as shown in figure 36 and figure 37, respectively.

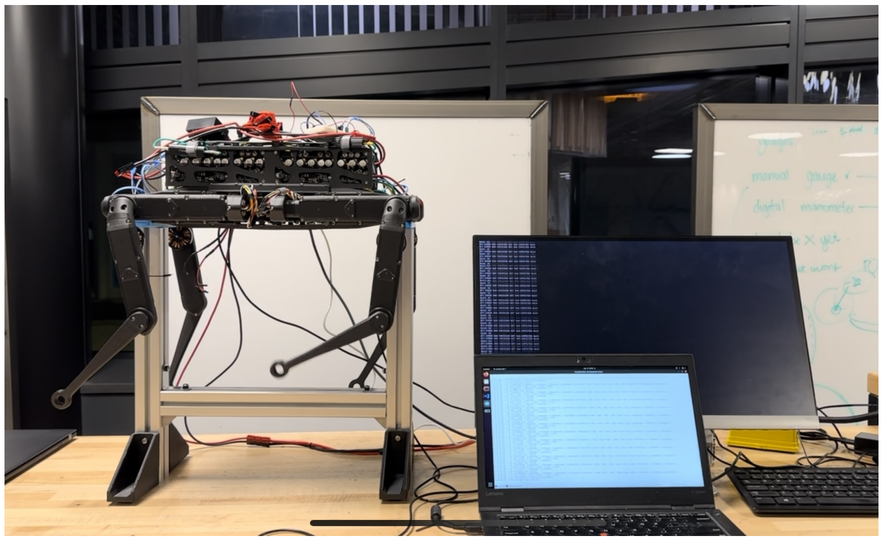


Figure 35: Setup for test of torque control of a single leg for a trot gait. Big monitor shows output from Raspberry Pi receiving feedback joint state data through SPI. Laptop shows output from Teensy receiving desired joint state commands through SPI. Computed torque commands are sent to ODrive motor controllers to command single leg.

We then set up a test program on the Raspberry Pi that imported joint commands from a CSV file and sent these commands to the Teensy every 2ms over SPI. The Teensy then computed the desired joint torques using the impedance controller and sent it over UART to the 6 ODrives. We ran a timer on the Teensy program to evaluate the elapsed time between each run of the full communication pipeline and we printed the results to the Serial debugger. We got an average of about 1 ms (1 kHz frequency) elapsed time. This showed us that the communication pipeline did meet our requirement of at least 0.5 kHz (2 ms per loop). In reality, if we had run all four legs, the elapsed time would have been a little bit longer. However, we did calculate that running all four legs would have taken around about 1.3 ms per loop (0.5 kHz frequency), which would also have met our frequency requirement. This test showed that by shifting from UART to SPI for the Raspberry Pi to Teensy communication and by developing a custom single-character ASCII protocol to dispatch batch torque commands to the six ODrives, we were able to optimize our communication pipeline to meet the controller frequency requirements.

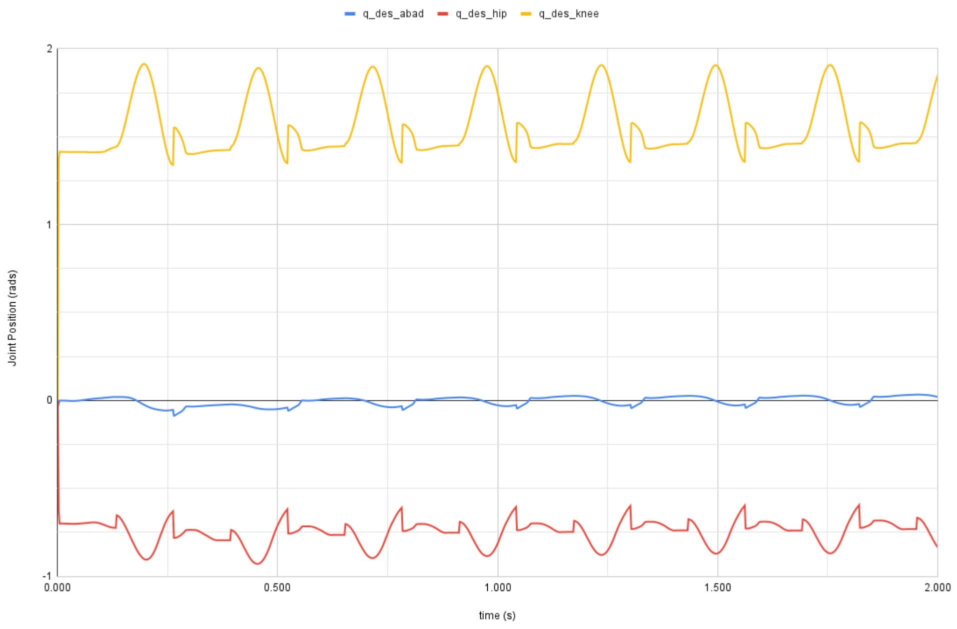


Figure 36: Plot of desired joint positions for a single leg for a trot gait. Desired Hip AA joint position (blue), desired Hip FE joint position (red), and desired knee joint position (yellow) are plotted for total time of 2 seconds. Observe periodic motion for each joint.

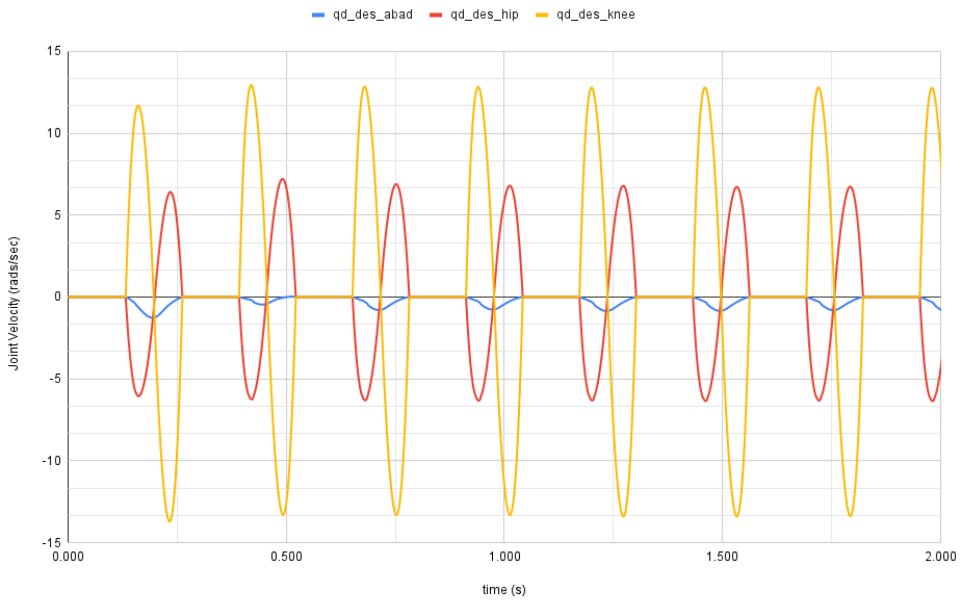


Figure 37: Plot of desired joint velocities for a single leg for a trot gait. Desired Hip AA joint velocity (blue), desired Hip FE joint velocity (red), and desired knee joint velocity (yellow) are plotted for total time of 2 seconds. Observe periodic motion for each joint.

6 Recommendations and Future Work

Several areas of future work that can further enhance the capabilities of the 12-DOF robot. Firstly, further improvements can be made in the perception system for person tracking. This can include incorporating advanced machine learning algorithms for better object detection, tracking, and human pose estimation, enabling the robot to have a more accurate and robust perception of its environment.

Due to time constraints and ODrive limitations, we were not able to fully tune the current control mode of the ODrives for smoother torque control. Additionally, although we unit-tested getting pose data from the T265 camera and integrated it into the high-level software stack, we were not able to test the full robot running. For system improvements, Teensy+ODrives architecture could be replaced with STM32xx MCU for low-level control and follow a similar actuator controller design like in [26]. This would reduce the number of components, hence reducing the weight of the robot and increasing the payload capacity.

Additionally, the high-level planning module can be further developed to enable the robot to autonomously plan and execute complex tasks in unstructured environments. This can involve incorporating motion planning, task planning, and decision-making algorithms to enable the robot to perform tasks such as object manipulation, tool use, and navigation in dynamic environments.

Furthermore, the full autonomy mode of the robot can be improved by integrating more advanced algorithms for environment mapping, localization, and navigation, allowing the robot to autonomously explore and navigate in unknown environments with minimal human intervention.

Moreover, the simulation with dynamic and reactive walking can be further refined by incorporating more realistic and complex environmental dynamics, such as uneven terrain, obstacles, and varying friction coefficients, to better emulate real-world scenarios and evaluate the robot's performance in more challenging conditions.

Finally, further research and development can be conducted to explore potential applications of the 12-DOF robot in various fields, such as disaster response, search and rescue, healthcare, and industrial automation, among others, where its advanced capabilities in perception, planning, and autonomy can be leveraged to improve operational efficiency and safety.

7 Conclusion

In conclusion, our project involved collecting and assembling all the components for the Solo12 robot, including disassembling the Solo8, purchasing parts, and manufacturing shells. We followed the instructions provided by ODRI to assemble the robot, adjusting to some differences in wiring due to our use of ODrive motors. We also made modifications and additions to the Cheetah Software stack, including expanding and adapting the floating base dynamic class and quadruped class to include the kinematic and dynamic model of the Solo12 (BiQu), integrating interfacing code for the T265 camera for state estimation, and developing low-level controller code and Arduino program for the Teensy 4.1.

We implemented person following using HOG and MOSSE and utilized the D415 camera mounted on the robot for capturing video frames of the environment. This handled occlusion very well and it was an efficient tracking method that was successfully used for real-time applications. The person's location was calculated using MOSSE and the angle of the robot with respect to the person's location was used as input for the BiQu software, which controls the robot's legs.

Our implementation of the low-level controller and Arduino program for the Teensy 4.1 followed our designed flowchart in the System Design section. Overall, our project required significant modifications and additions to the hardware and software, and these changes were crucial to the success of our project. With our implemented solutions, we have made progress towards achieving our goals of person detection and tracking with the BiQu quadruped robot.

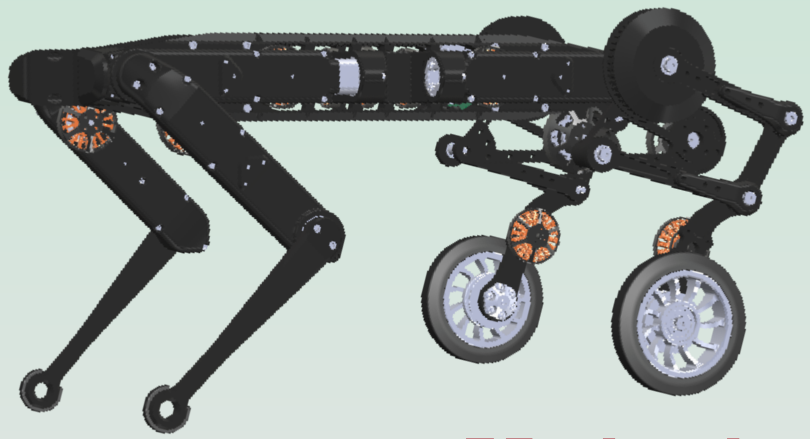
Appendix A

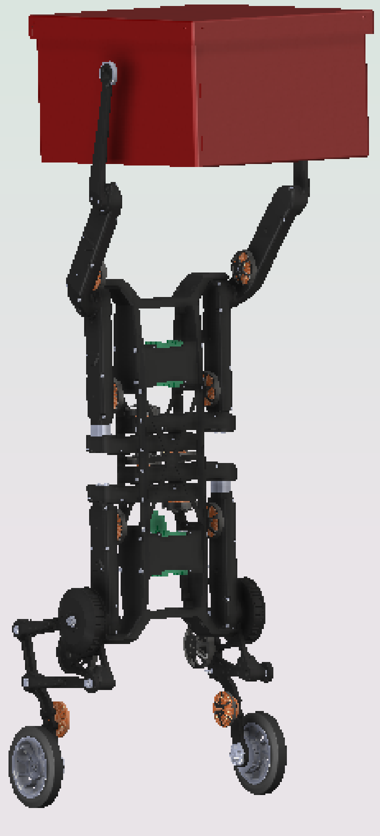
Grant Proposal

Bimodal Quadruped Robot

WPI 2022-2023 Major Qualifying Project

BiQu





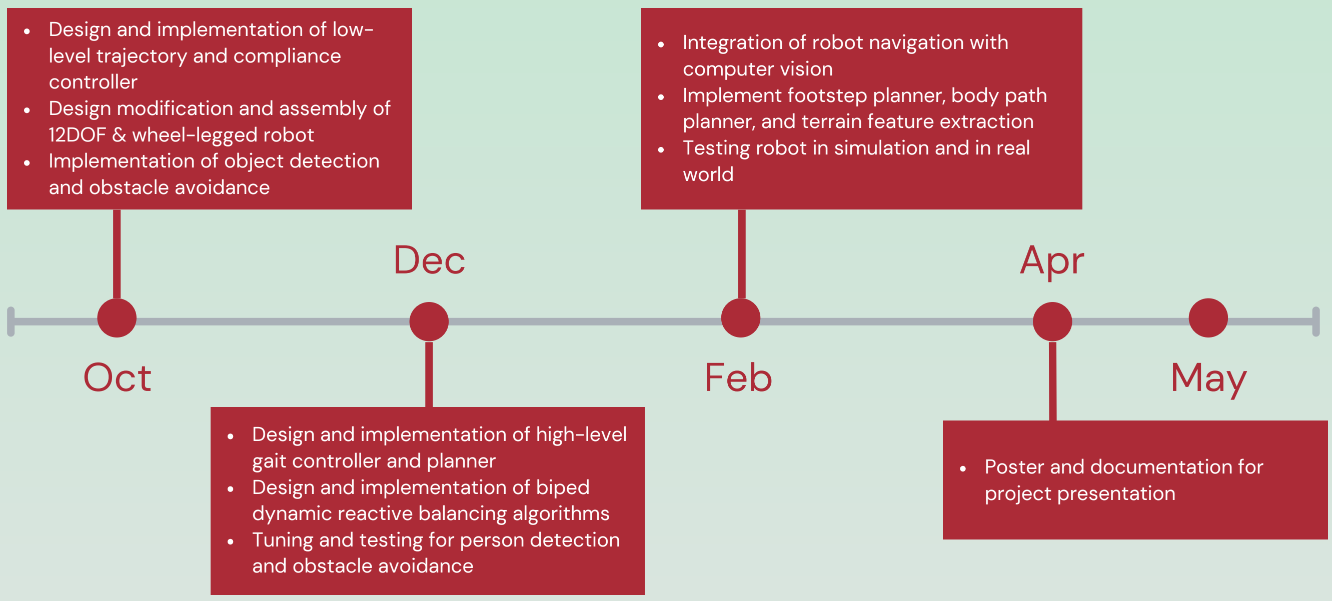
Motivation

Legged robots have been used for workflow automation, deployment in hazardous environments, and variety of other tasks. The task requirements determine which type of legged robot to use, limiting to one form of locomotion. For example, when traversing uneven terrains, one can benefit from the stability of quadruped robots. When it comes to manipulating objects, reaching high objects, or navigating in narrow spaces, the dexterity of bipedal robots might be beneficial. When high-speed locomotion is needed, wheeled robots would be a good choice. What if the robot has all these together? BiQu is a quadruped robot that can stand on two legs and has two wheels on the rear legs. By combining multiple forms of locomotion – walking, crawling, climbing, and driving – the robot is much more versatile with increased mobility. This allows the robot to accommodate a wider range of tasks, increase energy efficiency, and better recover from external disturbances. Furthermore, the integration of Computer Vision enables the robot to perceive obstacles and process terrain features, allowing it to transition between modes and adjust walking gaits to better adapt to the environment.

BiQu Specifications

- 12 Degrees of Freedom
- Hybrid locomotion: quadruped crawling and climbing, bipedal walking and driving
- Dynamic and reactive control system with self-balancing and compliance
- RGBD camera with built-in IMU for obstacle and terrain features detections
- Rigid and lightweight chassis in Onyx and rear legs in aluminium

Timeline



- Design and implementation of low-level trajectory and compliance controller
- Design modification and assembly of 12DOF & wheel-legged robot
- Implementation of object detection and obstacle avoidance

- Design and implementation of high-level gait controller and planner
- Design and implementation of biped dynamic reactive balancing algorithms
- Tuning and testing for person detection and obstacle avoidance

- Integration of robot navigation with computer vision
- Implement footstep planner, body path planner, and terrain feature extraction
- Testing robot in simulation and in real world

- Poster and documentation for project presentation

Cost

Quadruped Chassis	Cost
Actuators	$900
Electronics	$500
Hardwares	$400
Raw Materials	$500
Total	$2,300

The Team

Hushmand Esmaeili
Robotics Engineering

Yuen Lam Leung
Robotics Engineering

Freud Oulon
Robotics Engineering
& Computer Science

Aadhya Puttur
Computer Science

Contact

BiQu is a Major Qualifying Project (MQP) at WPI, a team-based capstone project and research experience. You can contact us via the following email: yleung@wpi.edu. The BiQu team is located in Unity Hall 150 on WPI campus. We will be presenting our work on the annual Project Presentation Day on Apr 21, 2023.

Sponsors

This project would not have been possible without our sponsors. We sincerely thank you for your support!

Appendix B

Bill Of Materials

Actuator Modules

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
Motors	Antigravity MN4004 KV300 - 2PCS/SET	2	T-Motor	155.9	311.8
Encoder Kit	Including: 1x motor shaft assembly with codewheel 625 cpr, 1x center pulley, 1x pre-modified Sensor AEDT- 9810-Z00 with on countersunk mounting holes dia. 3.0mm	4	PWB Encoder GmbH	142	567.82
Timing Belt First Stage	AT3 GEN III - width: 4mm / length: 150mm / 50 teet	4	Belting Online	6.61	26.44
Timing Belt Second Stage	AT3 GEN III - width: 6mm / length: 201mm / 67 teet	4	Belting Online	10.67	42.68
Bearing - Motor Shaft & Center Sh	EZO bearing 8mm x 4mm x 2mm	12	Bearing Direct	3.33	39.96
Bearing - Timing Belt Tensioner	EZO bearing 7mm x 3mm x 3mm	8	Bearing Direct	3.5	28
Bearing - Hip AA	EZO beaing 25mm x 20mm x 4mm ET2520 2Z VA	4	123 Bearing	11.72	46.88
Bearing - Output Shaft	EZO Bearing: 6705-2RS 25x32x4mm	8	123 Bearing	6.8	54.4
				Total	1117.98

Fasteners

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
Helical Insert	M2.5 - 0.45 x 3.8 mm, Packs of 10, 91732A767	3	McMASTER-CARR	6.27	18.81
Socket Head Cap Screw	M5 - 0.8 x 20mm, Stainless Steel, Packs of 40	1	Amazon	9.49	9.49
Helical Insert	M3 Helicoil, Stainless Steel, Packs of 50	1	Amazon	16.36	16.36
Socket Head Cap Screw	M3 - 0.5 x 14 mm, Stainless Steel, Packs of 100	1	McMASTER-CARR	7.36	7.36
Helical Insert	M3 - 0.5 x 6 mm, Stainless Steel, Packs of 10	1	McMASTER-CARR	11.35	11.35
Flat Head Screw	M3 - 0.5 x 5 mm, Stainless Steel, Packs of 100	1	McMASTER-CARR	3.96	3.96
Flat Head Screw	M8 - 1.25 x 25 mm, Stainless Steel, Packs of 5, 9180	1	McMASTER-CARR	8.85	8.85
Socket Head Cap Screws	M3 - 0.5 x 35 mm, Stainless Steel, Fully Threaded	1	Amazon	9.55	9.55
				Total	85.73

Electronics

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
ODrive Motor Controller	Shipping Fee	1	ODrive Robotics	15.25	15.25
Teensy 4.1		1	3DMakerWorld	45.73	45.73
Tracking Camera	Intel® RealSense™ Tracking Camera T265	1	Intel	358.1	358.1
SD Card	Centon 32GB Micro SDHC Card	1	WPI Bookstore	20.00	20
				Total	439.08

Materials

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
Shell Test Prints	ABS	1	MakerSpace Prototyping Lab	23.84	23.84
Onyx Filament	Markforged Onyx Filament - 800cc - Product No. M-	1	MatterHacks	201.88	201.88
Transmission Pulleys	Formlabs Tough Resin, Transmission Pulleys	1	ARC	14.96	14.96
Aluminium Profile	40cm - SureFrame 40 Series light T-slotted rail, 6063-T6 anodized aluminum alloy, cut to length, 40 x 40mm profile, slot size 8.	2	Automation Direct	14	28
Aluminium Profile	35cm - SureFrame 40 Series light T-slotted rail, 6063-T6 anodized aluminum alloy, cut to length, 40 x 40mm profile, slot size 8.	1	Automation Direct	12.55	12.55
Corner Bracket	1-1/2" x 1-1/2" x 15/16", Zinc-Plated Steel, 15705A	2	McMASTER-CARR	6.54	13.08
				Total	294.31

Wiring

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
Wires	Solid Hook Up Wire Kit (Tinned Copper) 22 G	1	Electronix Express	24.99	24.99
Bullet Connectors	USAQ 2mm Bullet Connector, Gold-Plated, 20 Pairs	1	Amazon	9.55	9.55
Dupont Connector Kit	Crimp Connector Kit with 2.54 mm Crimp Pin Connector Housings, Single Row Male Headers, Male/Female Crimp Pins and Ribbon Cable from Plusivo	1	Amazon	10.61	10.61
				Total	45.15

Miscellaneous

Part Name	Description	Quantity	Distributor	Unit Cost ($)	Total Cost ($)
Tap kit	M1 - M3.5 Tap bits	1	Amazon	14.86	14.86
Helicoil Installation Tool	M2.5	1	Amazon	16.28	16.28
				Total	31.14

Overall	**2013.39**

References

[1] Forecast 3D, "3D Printing Materials - Prototype and Production Plastics, Metals, Rubbers."

[2] Markforged, "Markforged - Onyx."

[3] F. Grimminger, A. Meduri, M. Khadiv, J. Viereck, M. Wüthrich, M. Naveau, V. Berenz, S. Heim, F. Widmaier, T. Flayols, J. Fiene, A. Badri-Spröwitz, and L. Righetti, "An open torque-controlled modular robot architecture for legged locomotion research," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3650–3657, 2020.

[4] A. G. Euredjian, R. Mahajan, and A. Gupta, "Multi-modal locomotion robot," May 2021.

[5] J. He, J. Shao, G. Sun, and X. Shao, "Survey of quadruped robots coping strategies in complex situations," *Electronics*, vol. 8, no. 12, p. 1414, 2019.

[6] K. Machairas and E. Papadopoulos, "An active compliance controller for quadruped trotting," *2016 24th Mediterranean Conference on Control and Automation (MED)*, 2016.

[7] G. Chen, S. Guo, B. Hou, and J. Wang, "Virtual model control for quadruped robots," *IEEE Access*, vol. 8, p. 140736–140751, 2020.

[8] B. Jin, C. Sun, A. Zhang, S. Liu, W. Hao, G. Deng, and P. Ma, "Single leg compliance control for quadruped robots," *2017 Chinese Automation Congress (CAC)*, 2017.

[9] V. Barasuol, J. Buchli, C. Semini, M. Frigerio, E. De Pieri, and D. Caldwell, "A reactive controller framework for quadrupedal locomotion on challenging terrain," *2013 IEEE International Conference on Robotics and Automation*, 2013.

[10] D. Kim, J. D. Carlo, B. Katz, G. Bledt, and S. Kim, "Highly dynamic quadruped locomotion via whole-body impulse control and model predictive control," 2019.

[11] P.-A. Leziart, T. Flayols, F. Grimminger, N. Mansard, and P. Soueres, "Implementation of a reactive walking controller for the new open-hardware quadruped solo-12," *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

[12] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[13] D. B. S. Terms and Objects, "Difference between computer vision and machine vision,"

[14] H. Chai, Y. Li, R. Song, G. Zhang, Q. Zhang, S. Liu, J. Hou, Y. Xin, M. Yuan, G. Zhang, and Z. Yang, "A survey of the development of quadruped robots: Joint configuration, dynamic locomotion control method and mobile manipulation approach," *Biomimetic Intelligence and Robotics*, p. 100029, 2022.

[15] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: an operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.

[16] Mit-Biomimetics, "Mit biomimetics cheetah software."

[17] J. D. Carlo, "Software and control design for the mit cheetah quadruped robots," Master's thesis, Massachusetts Institute of Technology, Feb 2021.

[18] R. Tedrake, *Robotic Manipulation*. 2022.

[19] I. R. Depth and T. Cameras, "Depth camera d415,"

[20] B. Jin, C. Sun, A. Zhang, S. Liu, W. Hao, G. Deng, and P. Ma, "Single leg compliance control for quadruped robots," pp. 7624–7628, 2017.

[21] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013.

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.

[23] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," vol. 1, pp. 886–893 vol. 1, 2005.

[24] R. K. Sidhu, "Tutorial on minimum output sum of squared error filter," 2016.

[25] V7, "The complete guide to object tracking [+v7 tutorial],"

[26] B. G. Katz, "A low cost modular actuator for dynamic robots," Master's thesis, Massachusetts Institute of Technology, Oct 2018.

[27] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2007.

[28] Open Dynamic Robot Initiative, "Quadruped Robot 12dof v1," Aug 2022.

[29] T-Motor, "Antigravity MN4004 KV300."

[30] "Where does the formula for calculating torque come from?," Sep 2018.