Additively Manufactured Open-Source Quadruped Robots for Multi-Robot SLAM Applications

Zachary Joseph Daniel Fuge

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Mechanical Engineering

Kaveh Akbari Hamed, Chair
Alexander Leonessa, Co-chair
Dylan Losey

April 30, 2025
Blacksburg, Virginia

Keywords: Quadruped, Multi Robotics, SLAM, Additively Manufactured, Robot

Additively Manufactured Open-Source Quadruped Robots for Multi-Robot SLAM Applications

Zachary Joseph Daniel Fuge

(ABSTRACT)

This thesis presents the design and development of the quadruped robot Squeaky, created as a research and learning platform for single and multi-robot simultaneous localization and mapping (SLAM), computer vision, and reinforcement learning. Affordable robots are increasingly essential for scaling from single-robot to multi-robot applications, as costs can rise exponentially as fleet size increases. SLAM is a critical feature for enabling a robot to perceive and localize itself within its environment, supporting applications such as cave exploration, disaster assistance, and remote inspection. To enhance efficiency, a fleet of robots can collaborate to merge individual maps for multi-robot SLAM. Squeaky is an affordable quadrupedal robot with adaptable hardware and software capable of creating merged maps from multiple robots over a shared network. It is also open-sourced for the benefit of the research and educational community. This work covers Squeaky's full design and development process, with validated results demonstrating the platform's effectiveness for research and educational applications.

Additively Manufactured Open-Source Quadruped Robots for Multi-Robot SLAM Applications

Zachary Joseph Daniel Fuge

(GENERAL AUDIENCE ABSTRACT)

This thesis presents the design and development of a robot named Squeaky, created as a tool for research and learning. Squeaky is a four-legged robot that can help study how robots map and understand their surroundings. As multiple robots become more common in search-and-rescue missions or exploring dangerous environments, affordable robots become increasingly important. This is because the cost of using many robots can rise quickly as more are added to the team. One key capability for robots is the ability to see their surroundings, which helps them navigate and understand their environment by creating maps. When several robots work together, they can combine their maps to understand an area better. Squeaky is designed to be an affordable and flexible robot that can work together with other robots to create shared maps. It is also open-source, meaning others can freely use and improve it for their research. This thesis details the design process of Squeaky and provides results showing its usefulness for both learning and research purposes.

Dedication

This work is dedicated to my family and friends, without whom this would not have been possible. Per Aspera Ad Astra.

Acknowledgments

My journey through graduate school would not have been possible without the connections and friendships I formed. Their diverse backgrounds and perspectives have allowed me to grow personally and academically, helping me refine my abilities and deepen my understanding. For that, I am genuinely grateful.

I want to express my sincere appreciation to my advisor, Dr. Alexander Leonessa, who took a chance on me as an undergraduate student and allowed me to join TREC. This opportunity allowed me to work on various robotic projects, and I am grateful for his constant guidance and support throughout my master's journey.

I would also like to thank my committee members, Dr. Hamed and Dr. Losey, for their valuable insights and perspectives on robotics. Their input has allowed me to approach my project from different angles, further enriching my work.

This journey would not have been possible without the fantastic people I met at the TREC lab. Many of them, with their bright personalities, passion for robotics, and extensive knowledge, fueled my passion and kept me going through these years. I would like to especially recognize Suyeon, Dr. Bhaben Kalita, Sam, Nick, Max, Stephen, Logan, Michael, Melanie, Maddie, An-Chi, Jungsoo, Vitto, Julianna, Carlo, Isaac, Rohaya, Negin, and many others. The times we shared will always be cherished.

I would also like to thank Benjamin Beiter for his early guidance in my project. He helped me stay focused on the bigger picture and, with his support, I could condense and refine my extensive list of ideas. Without his help, I might still be struggling to narrow down my focus.

Thank you to Connor Herron, who took on a mentorship role when I joined TREC as an

undergraduate student. His passion, knowledge, and expertise in humanoid robotics ignited my interest in the field and inspired me to push the boundaries of what is possible. I am truly grateful for all the help and advice he has provided over the years.

I want to express my heartfelt gratitude to my brother, Alex. He has been a constant source of motivation, helping me stay focused and driven throughout my undergraduate and graduate studies. Through his experiences, he has offered invaluable advice, helping me stay grounded and focused on my priorities.

Lastly, I would like to thank my family and friends for their unwavering support throughout this chapter of my life. They have been my rock, providing stability and encouragement that allowed me to concentrate on my research and studies while always being there when needed. Their belief in me has been a constant source of strength and motivation, fueling my journey to completion.

Contents

List of Figures

List of Abbreviations

AM Additive Manufacturing

HL High-Level

LL Low-Level

SLAM Simultaneous Localization and Mapping

Additive Manufacturing (AM): has seen increased adoption as the technology has advanced. Unlike traditional manufacturing methods that remove material, AM builds parts by adding material layer by layer.

Simultaneous Localization And Mapping (SLAM): is the process by which a robot maps and navigates as it traverses an unknown environment. As it traverses this environment, it consistently tracks itself within it.

High-Level (HL): refers to abstract planning and control processes not directly tied to hardware. This part of the system focuses on strategic decision-making for the overall operation.

Low Level (LL): refers to more direct control of hardware, typically referring to motor actuation and sensor reading.

Chapter 1

Introduction

1.1 Background & Motivation

There have been many improvements to SLAM (Simultaneous Localization and Mapping) algorithms in recent years. These algorithms can be classified into two primary approaches for 2D lidars: filter-based and graph-based SLAM. These various approaches of SLAM have been integrated into the ROS framework as packages such as GMapping [1], KartoSLAM[2], Cartographer [3], HectorSLAM [4], and recently SLAM toolbox [5]. With these improvements, SLAM is becoming significantly more reliable and capable, making it possible for multi-SLAM applications, specifically in robotics. When two robots come into the range for peer-to-peer communication, their internal maps are merged [6].

SLAM can be described as the ability of a robot to map and localize itself within its immediate area, either autonomously or via remote control. This is useful in many applications including autonomous navigation [7, 8, 9], construction site inspection [10, 11, 12, 13], agricultural [14, 15, 16], and warehouses [17, 18, 19, 20]. These algorithms require mobile robots capable of navigating complex environments. Specifically, legged robots are being developed to operate within human-centric environments and challenging terrains. However, access to multiple SLAM-capable devices is a considerable financial barrier for research groups and small-scale applications.

SLAM has become the research focus in the last decade because of technological improvements in sensors, hardware, and motors. Especially as these items have seen significant cost, size, and power improvement, this has led to research and development in designing more affordable and capable robots that can test a wide array of SLAM algorithms in a real-world environment. Whereas currently, simulations have been where many algorithms have been tested, which usually concludes that these algorithms will work under ideal conditions. This rarely accounts for other factors that impede successful tests, such as hardware cost and part availability of the robotic platform. Many affordable quadrupeds have been designed to address this gap, but they can still be considered relatively expensive when scaling up a project to a multi-robotic system.

Real-world robotics testing requires substantial financial investment, time, and learning to use the hardware to test new algorithms. Looking at the economic capital requirements, robotic hardware is expensive, especially in prior years when the cost of the base model robots capable of performing SLAM operations ranged from $784 [21] to $7100 [22]. Research labs without the financial resources to acquire physical robots often conduct their research in simulation, leading to the development of robotic models within virtual environments. Simulations provided a more cost-effective alternative to acquiring physical robotic hardware. This meant that testing multi-robotic applications experimentally became an aspiration for many labs because the cost per robot was prohibitive. This is why the primary motivations for this work were to research and develop an affordable quadruped that could be expanded into multi-robotic applications for SLAM in real-world environments. Squeaky is an alternative to other platforms with a base cost of $460.

1.2 Open Sourced Quadrupeds

Quadrupeds are not a new concept within robotics; various types have been created, ranging in size, actuation, and control. In the last twenty years, quadrupedal robots have substantially decreased in size due to technological improvements. Starting with LittleDog, developed by Boston Dynamics for quadruped research and to develop robust locomotion on different types of terrain [23]. As locomotion research within quadrupedal development improved, Boston Dynamics developed Big Dog, a larger quadruped robot with the goal of carrying heavier cargo while traversing through rough terrain [24]. Then, two other robots, HyQ and HyQMax, were developed by the Italian Institute of Technology (IIT) to improve quadrupedal locomotion research, such as running and jumping [25, 26]. All four of these robots utilize a bio-inspired quadruped approach with the knees symmetrical and angled backwards, whereas the ANYmal quadruped has its front and back knees pointed inwards. This robot uses a unique approach to the locomotion problem, which the authors explain, that this quadruped performs better dynamically in various environments [27]. MIT then developed the MIT Cheetah 1 and 2, which included powerful motors to achieve trotting on Cheetah 1 and then Cheetah 2, achieving autonomous high-speed bounding over obstacles [28, 29].

As discussed in the previous paragraph, various quadrupeds have been developed. With the addition of new manufacturing techniques and the reduction in component costs, there have been new developments in open-source quadrupeds targeting different research areas at an affordable price. A significant area for affordability has been research into AM quadrupeds. The benefit of AM is that it is cheaper, easier to build, redesign, or replace parts, and accessible to anyone with a 3D printer. As a result, there has been an influx of quadrupeds, aiming for affordability or enhanced capability.

When going for affordability, the aim is to remove machining cost by using 3D printed parts for the structural components. The primary cost of these robots is mainly in the motors. This can be seen with PADWQ, a fully 3D printed robot with powerful motors that make up a disproportional amount of the cost [22]. Even with a more affordable price than other robots, this cost may still be an obstacle to some research groups wanting to test SLAM capabilities on legged robots.

Another quadruped includes the Stanford Pupper, a capable quadruped that can trot, jump, and is a comparatively smaller size to PADWQ [21]. Stanford Pupper's goal is an affordable quadruped to complete locomotion research and other tasks. Before the Pupper, the Stanford Doggo utilized a unique leg design which allowed Doggo to jump and be used for dynamic locomotion research [30].

Whereas RealAnt is a great choice for researchers interested in testing reinforcement learning algorithms in a real environment [31]. With this robot, the cost is comparatively less than Pupper at $784, but a research group may need to modify it to fit their needs if it does not align with RealAnt's original purpose. But, if cost is a concern and the team has moderate experience with electronics and mechanical assembly, the base robot cost is $410, the lowest of the mentioned quadrupeds. Compared to the other quadruped robots, such as Pupper, Doggo, and PADWQ, RealAnt's features an arachnid-inspired leg configuration.

Another quadruped that utilizes this arachnid design is Charlotte. This quadruped was explicitly developed to explore SLAM applications. Presented within their work is a quadruped that achieves SLAM functionality better than the previously mentioned quadrupeds [32].

Overall, each open-source quadruped has positive and negative aspects in price, configuration, or goals. The final decision on which quadruped to use for any research group involves their objectives and financial situation. If they are researching locomotion and care about

the type of joint sensors and overall response, then Pupper, PADWQ, and Doggo are a great choice. Are they focusing on exploration, reinforcement learning, or SLAM? Then Pupper, RealAnt, PADWQ, and Charlotte are possible choices. If cost per robot is a factor, then this selection becomes limited with the high cost of PADWQ, removing it from the running. However, this cost is still considerably less expensive than quadrupeds, which Boston Dynamics, IIT, MIT, and ANYbotics originally developed. Open Source Quadrupeds are configurable in all design aspects, allowing research groups not to be limited in their scope.

1.3 Multi-Robot SLAM

SLAM can be broken down into various approaches depending on sensor configuration. For this case, the algorithms utilized will be Filter-based and Graph-based approaches since the specific sensor used is a 2D lidar. Filter-based approaches treat SLAM as a state estimation problem that utilizes filtering to handle robot changes over time and sensor input for mapping. Common filters used are the Extended Kalman filter [33, 34] and Rao-Blackwellize particle filter [35], which is the basis for GMapping. Graph-based approaches use a nodal structure with each node containing the robot pose and the robot sensor information at that position. Then, with the next node, a transformation must be performed between the two nodes using sequential odometry data or aligning robot observations at the prior and new node. The graph-based approach includes Karto [2] and SLAM toolbox [5].

As SLAM abilities have significantly advanced for individual robots, their application is now being extended to multi-robot systems, which is enhancing the infrastructure and effectiveness of these robotic networks. The applications for multi-robot SLAM include Search & Rescue [36] and agricultural [37], but are expanding rapidly into other areas. Using a multi-robotic system over a single robotic system is beneficial because more robots can complete

tasks collaboratively, saving on task completion time and increasing efficiency. To avoid confusing swarm robotics with multi-robotics, the primary differentiation is that swarm robots are commonly simple agents that achieve complex tasks together.

However, multi-robotic systems are more complex and can work together or independently depending on the environment or conditions. Multi-robot SLAM has made significant progress over the past decade, from enabling multiple robots in unknown positions to build a joint map, to allowing robots to merge maps based on various constraints [38, 39]. With multi-robotic systems, the amount of data processed and shared between agents is large. In Lazaro's work, they approach this issue by condensing this data to share it efficiently with each robot [40]. The importance of the prior work is a key problem that must be addressed to create robust multi-robotic systems. Recently, at MIT, a published work that had multi-robotic systems individually mapping large areas was presented. When contacting another robot, peer-to-peer communication was established, and the map data was transmitted to each robot. Once the new additional map was transmitted, a map merging algorithm was created to establish loop closure and create a robust map [6]. Overall, multi-robotics systems are becoming of interest to explore because of technological improvements and various capable platforms able to test algorithms experimentally to create robust frameworks.

1.4 Contributions

As a result, this thesis presents a low-cost, 3D-printed quadrupedal robot that is capable of SLAM. The specific contributions are the AM design, custom-designed electronics for the whole robot, and a simple communication and integration framework for multiple robots. These improvements allow for an easy-to-manufacture but still affordable and capable robot. Two of these robots can be built for the price of one of the least expensive, previously

presented robots known to the author, Stanford Pupper [21]. Using servo motors lowers the cost of each quadruped, with the tradeoffs being no joint sensing. This results in a lack of closed-loop control; however, this decreased control performance is not a problem specifically for SLAM platforms, as localization can account for any discrepancies in realized vs. desired motions. This robot also has expanded capabilities that can be explored as it has an integrated ROS framework, allowing for the interchangeability of algorithms and components. Overall, this robot is a cheaper, accessible platform for implementing and testing SLAM algorithms for research, education, and small-scale applications.

1.5 Outline

- **Chapter 2** is about the mechanical design of Squeaky. It reviews the improvements made on this robot, such as the navigational components and perception. It also includes a brief description of the legs and body.
- **Chapter 3** covers the electrical design of Squeaky. The focus is on the new items, such as the PCB, and changes to make the platform simpler to assemble.
- **Chapter 4** talks about the software stack, which includes an overview of the high-level and low-level systems and a brief description of the ROS visualization setup. Closing with the startup procedure for the high-level system needed for using this robot.
- **Chapter 5** presents how to assemble certain parts of this robot that may not seem as intuitive. It also shows a cost comparison between Squeaky and a few other open-source quadrupeds.
- **Chapter 6** presents the results of leg validation, single and multi-robot SLAM on Squeaky.

- **Chapter 7** concludes this work and presents possible avenues to pursue within future work.

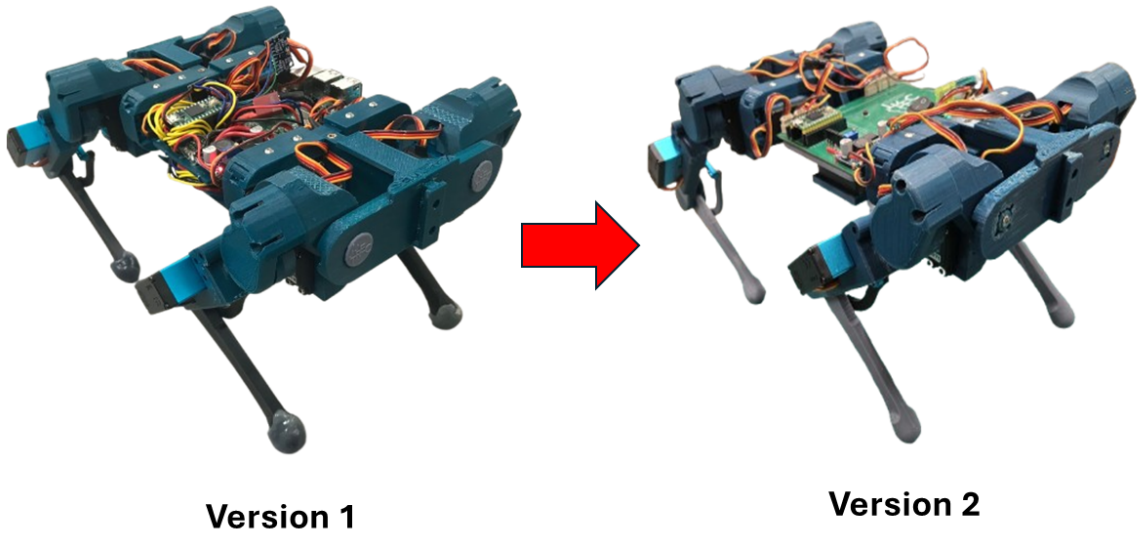

Figure 2.1: Figure Shows Version 1 and Version 2 of Squeaky.

Chapter 2

Mechanical Design

2.1 Previous Version

This section reviews the previous design iterations of Squeaky and is organized with an overview highlighting the changes of versions 1 and 2. Then, at the end of this section, it proceeds to a brief introduction of version 3, with more detailed information provided later in the chapter.

Understanding the evolution of the design is crucial for recognizing why specific changes were made. This context sheds light on the development process and helps research and education groups who wish to modify by clarifying the rationale behind previous decisions. This is why a detailed background is provided in this section.

The previous design of Squeaky was a smaller robot in terms of body length. It also had no perception sensors or condensed electronics as seen in Figure 2.1. The Teensy 4.0 took velocity commands from the Raspberry Pi 4, taking its inputs from an Xbox controller. Servos were the mode of actuation for this Squeaky, and are still used on the latest version.

The key difference between Squeaky V1.0 [41] and Squeaky V2.0 is the electronics as seen

Figure 2.2: Squeaky V3.0 or Navigational Squeaky.

in Figure 2.1. The cable routing was not condensed or organized, and there was no fixed placement for components other than proto-boards, which also were not fully secured. This type of routing leads to various electrical issues, including circuit shorts, incorrect sensor readings, and components falling off the chassis. The newer version addresses this issue using a full-fledged PCB designed for optimal power distribution, ease of assembly, and secured component placement to prevent electrical shorts.

Seen in Figure 2.1 is Squeaky V1.0, a proof of concept showing the feasibility of the quadruped; this version allowed for a baseline of the overall weight, locomotion capabilities, and cost. Based on what was learned from version 1, a list of ideas and improvements was made, such as securing the battery, PCB, condensing electrical cable routing, using liquid rubber on the feet, and simplifying the mechanical design for version 2.

The next iteration will be the focus of this thesis. To stay consistent between versions, this one will be referred to as the Navigational Squeaky or Squeaky V3.0, as seen in Figure 2.2. Squeaky V3.0 included a larger PCB with an NVIDIA Jetson Nano, perception sensors, and a higher current output buck converter. Due to the inclusion of the NVIDIA Jetson Nano, the chassis was lengthened, and 3D printed mounts were created to maintain the Center of Mass (COM) location in the center of the robot.

In conclusion, besides the initial iteration, each version has focused on optimizing the design and adding features. The first goal was to condense the electronic wiring into a more permanent PCB. This was done to make the design easier to use and assemble, aligning with its open-source nature. The following iterations simplified the part count to lower the robot's overall assembly time. Lastly, the current mechanical version included expanding Squeaky's capabilities to include an NVIDIA Jetson Nano and perception sensor, which required a few new parts to be designed. This latest version followed the over arching design principle and created low material cost AM parts that could be screwed onto already existing inserts and

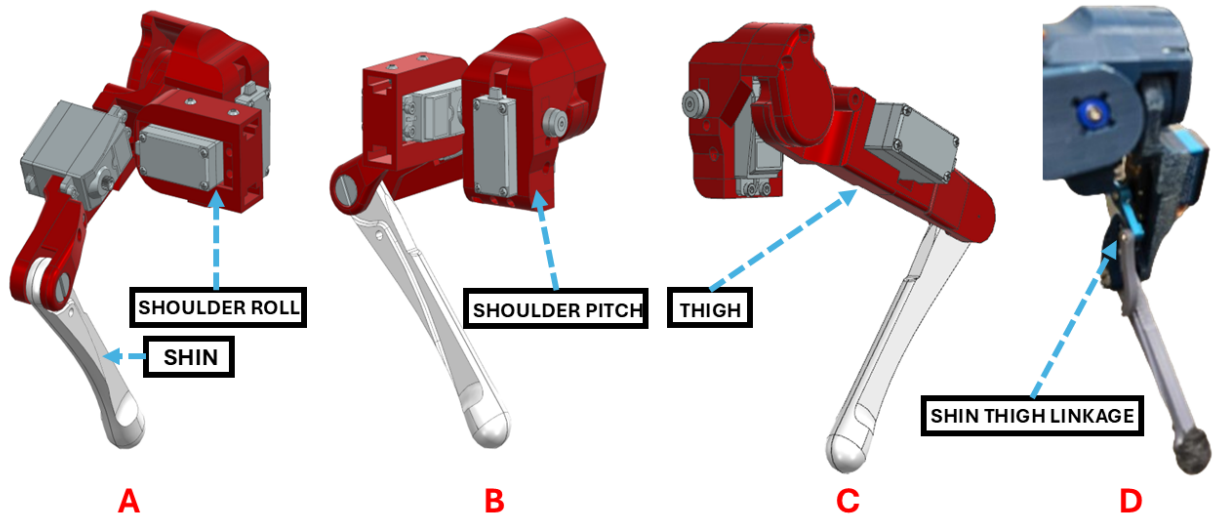

Figure 2.3: Legs labeled and shown isometrically from 3 different perspectives. Image D shows the connecting linkage for the servo and the shin.

brackets.

2.2 Leg Overview

The leg design of Squeaky remains unchanged from its original implementation in V1.0, as detailed in the prior conference paper [41]. It follows a biologically inspired yet simplified structure based on mammalian locomotion, consisting of four main components: the shoulder motor (A), shoulder socket (B), thigh (C), and shin (A), as illustrated in Figure 2.3.

Figure 2.3 provides multiple perspectives of the leg, isolated from the chassis. At the top of the assembly, the shoulder is broken into two rotational axes: shoulder roll and shoulder pitch. A dedicated servo motor actuates each axis, both embedded directly into the 3D-printed components. This embedded design allows for external mounting and a compact

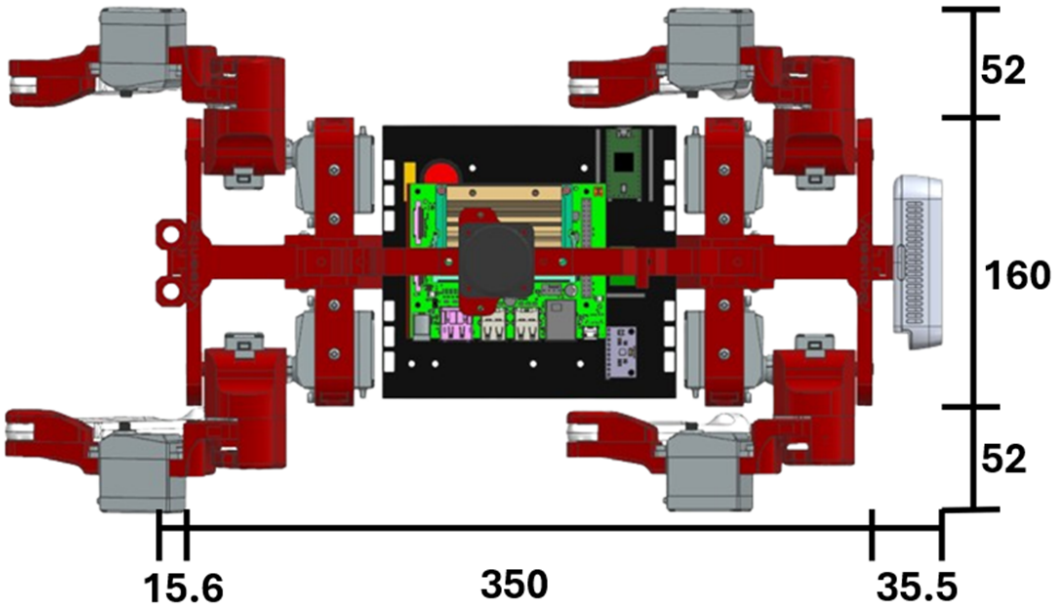

Figure 2.4: Current Squeaky Dimensions all in mm.

joint structure that interfaces cleanly with the robot body.

Below the shoulder lies the thigh, which connects the shoulder assembly to the shin and houses the servo responsible for actuating the shin. Notably, this joint does not follow a direct-drive configuration; a linkage transmits the torque to the shin, creating rotational movement about its axis.

The final component, the shin, is a single solid 3D-printed piece designed for structural stability. Its base is coated with liquid rubber to enhance traction and ground reaction forces during walking. The shin's geometry is optimized to endure repeated impacts and stresses during locomotion without premature material fatigue.

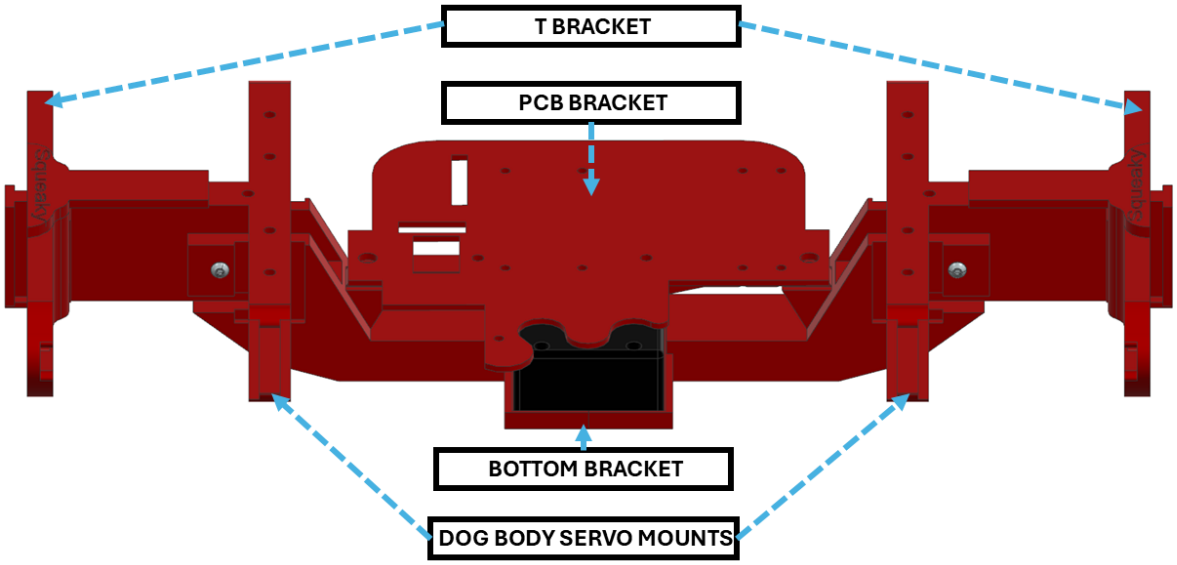

Figure 2.5: Isometric side view of Squeaky body with labeling of important structural components.

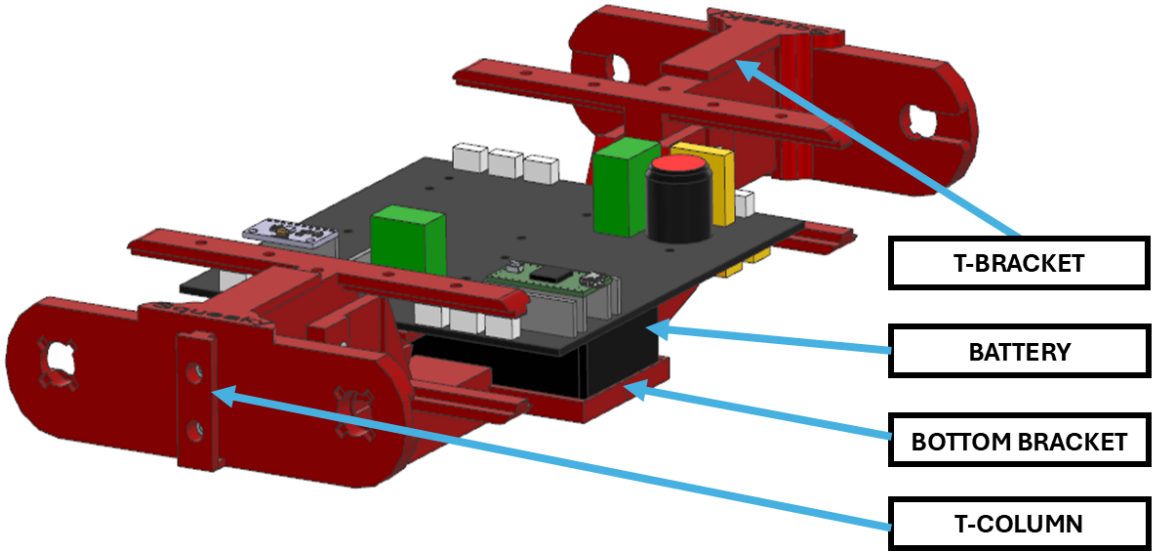

Figure 2.6: Isometric view of Squeaky's Body.

2.3 Body

The chassis has been widened and lengthened compared to Squeaky V1 and V2. Previously, it was 275 mm and has now been lengthened to 350 mm to accommodate an NVIDIA Jetson Nano, a larger buck converter, and a custom control board as seen in Figure 2.4. This was done with the initial intention that the NVIDIA Jetson Nano would screwed onto this board, with only a slight offset not directly to contact the control board. This was adjusted when the wires coming from the buck converter made that board placement impossible, and to adjust the CoM of Squeaky.

The body design is a simplified rectangular prism containing sections for 4 legs at each corner. The main body then includes the central PCB, which acts as a structural piece on the robot. It is also the core of the robot's handling of power distribution to motors and logic commands. The battery is placed under the PCB and secured with a bracket designed into the chassis, and then the PCB is placed on top, mounted to a printed board holder that keeps the battery in place, as seen in Figure 2.5. The battery is only accessible once the PCB is removed.

Squeaky has on its front/back a T column labeled in Figure 2.6 that allows for any new component to slide into it and locked in with screws when needed. These slide parts on the very front and rear of the robot were used because if a new test sensor wanted to be added, there would be a universal mount. Because of this T-column slide mechanism, parts can be quickly designed and printed, allowing for the easy interchange of new hardware.

This section outlines the primary modifications to the chassis, particularly the dimensional changes made to accommodate the NVIDIA Jetson Nano and supporting hardware. It also includes labeled diagrams for easier identification of critical body components. Special attention is given to the central PCB and battery configuration, forming the robot's functional

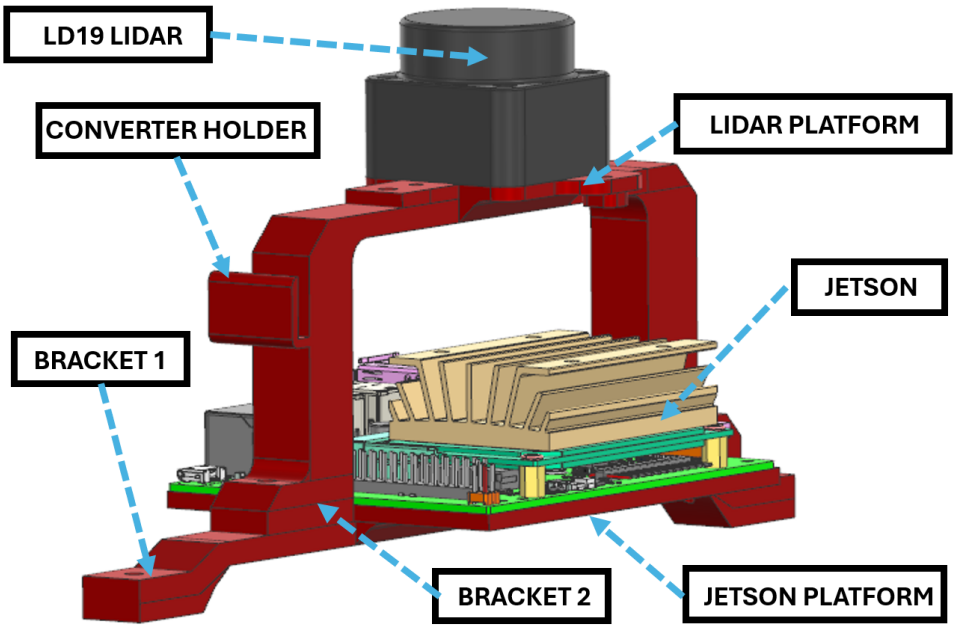

Figure 2.7: Navigational Design component, each part labeled.

core. Finally, the addition of front and rear T-columns is emphasized for their role in enabling quick, tool-free integration of new components, reinforcing the robot's modular and adaptable design philosophy.

2.4 Navigation Design

The Navigational Design section details the mechanical additions to Squeaky to support new perception sensors, including the NVIDIA Jetson Nano, Lidar, and depth camera. It begins with background information on these components and explains the rationale behind developing new AM parts to house and support them. The section then breaks down the Lidar's placement and mounting strategy, followed by positioning the depth cameras and antennas on the newly designed AM components.

This section highlights the integration of advanced perception tools and reinforces the chas-

sis's modularity. It demonstrates Squeaky's adaptability to a range of new components made possible by the flexibility of AM techniques. Ultimately, it showcases how these custom-designed parts enable rapid prototyping and seamless attachment of new hardware, supporting ongoing development and experimentation.

This perception variant of Squeaky includes a component for securing the NVIDIA Jetson Nano, which can be seen in Figure 2.7. Early designs had the NVIDIA Jetson Nano placed on the PCB as mentioned in Section 2.3, but had to be shifted up because of wire routing for the buck converter and CoM. The current NVIDIA Jetson Nano placement has space to connect to the GPIOs on the custom PCB to control the robot. Then, with the AM parts, a large bracket was designed and implemented to mount the 2D LD19 LDROBOT lidar above the robot chassis. It was shifted higher because a fan was affixed to the microprocessor to handle cooling. The fan was added because the microprocessor, during runtime, maintains a temperature around 20-21 °C on idle. Still, when running computationally heavy algorithms, it will exceed 39 °C+ and cause a thermal shutdown. The fan preserves the temperature around 31-32 °C when intensive algorithms such as SLAM are running.

The attachment for the lidar is comprised of two separate parts. These 2 parts are the Lidar platform and Bracket 2. The Lidar platform interfaces the LD06 Lidar to Bracket 2 as seen in Figure 2.7. The NVIDIA Jetson Nano also sits on a platform that interfaces with Bracket 1. These components were separated into 4 different AM-designed pieces to simplify the 3D printing by removing excessive supporting structure and shortening the print time. Bracket 2 includes an outcropping on one column labeled Converter Holder in Figure 2.7 to support the lidar converter board.

Squeaky is designed to be easily configurable because it can add and remove components as necessary. This was discussed in more detail within subsection 2.3 on the exact mechanism. In Squeaky V3.0, these T-Columns are currently used to secure a D435i camera on the front.

On the back are two antennas that give the NVIDIA Jetson Nano WIFI and Bluetooth. Lastly, the D435i camera has a 1/4-20 UNC thread mounting point, which secures the camera to the bridging part that slides into the T-column.

In conclusion, this section detailed the new perception-focused additions to Squeaky, emphasizing how easily the custom AM parts were designed and integrated into the chassis. It highlighted incorporating key components such as the Lidar and depth camera. It demonstrated how the T-columns continue to support modular attachments, facilitating the secure mounting of the depth camera and antennas for the NVIDIA Jetson Nano. These updates further underscore Squeaky's adaptable design and readiness for evolving research and development needs.

Chapter 3

Electrical Design

This chapter presents an overview of the electrical system that powers and coordinates Squeaky V3.0. The electrical design was intentionally kept simple to ensure ease of use for troubleshooting and modification. These were the key goals for this platform, which is intended to support a wide range of research and educational applications. By simplifying the layout and making all key components easily accessible, the system encourages users of varying experience levels to engage with the hardware without needing deep electrical engineering knowledge.

The chapter describes the custom–designed PCB, the central power distribution, and the signal routing hub. It then outlines the power supply system, including the battery configuration and protective features, and a breakdown of how components like the Teensy microcontroller, NVIDIA Jetson Nano, and servo motors are connected and communicate. Together, these sections document the robot's technical structure and demonstrate how the electrical design supports Squeaky's modularity, adaptability, and user-friendliness goals.

3.1 PCB Hub

This robot's straightforward electrical design facilitates the Squeaky platform's ease of use and modularity. Beginning with 12 servo motors that handle the locomotion, these motors are all connected to the custom-designed PCB that serves as the core of the quadruped.

The servo grouping can be seen in Figure 3.1 labeled as Servo Connection Grouping; each grouping represents one leg, containing a shoulder, thigh, and shin servo. The battery is placed under the board and fitted with a 3D printed bracket to secure it, as mentioned in Section 2.3. The PCB has a connector on the bottom that allows the battery to be directly connected to the board. A fuse was added between the rocker switch and the battery because the battery is always connected to the PCB once mounted in the chassis. The addition of the fuse serves as a measure to protect the board and a cut-off to deny power to the rest of the robot in the case of accidentally flipping the rocker switch, which will put the robot into standing position. This also serves as an excellent feature for securing and transporting the robot.

Power distribution on the PCB is handled with a 4-layer board. The top layer is for signals, 2nd layer is a ground plane, 3rd layer is a power plane for the servos, and 4th layer is another signal. Space on the signal layers that does not have routing is converted to a ground plane

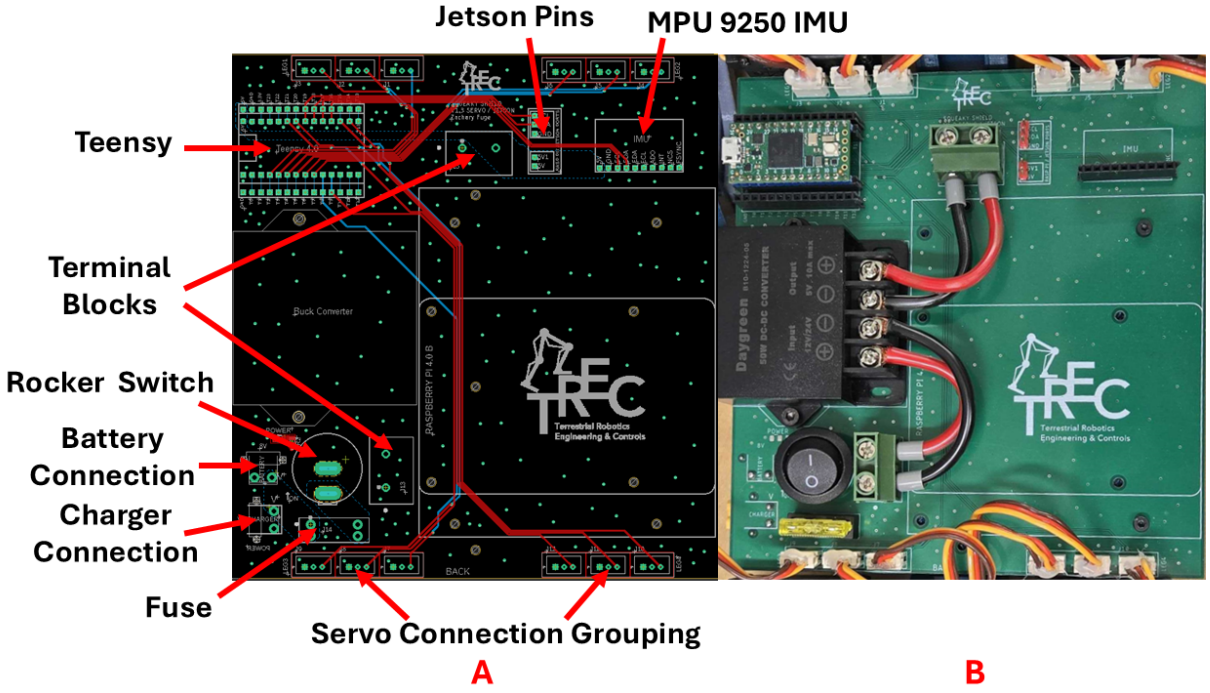

Figure 3.1: Image A shows the PCB in CAD, and image B shows the physical PCB

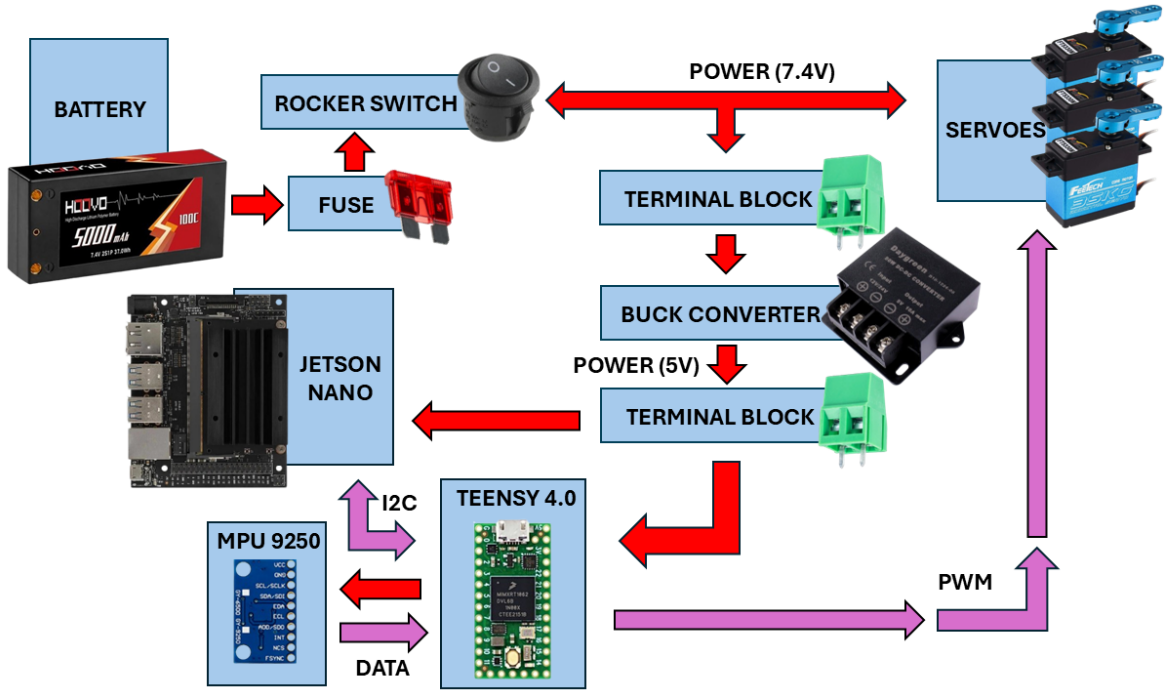

Figure 3.2: Simplified Diagram of Power and Data connections to each component

to eliminate possible noise between signal routes. Power routing on the PCB is shown in Figure 3.2. The power plane is connected directly to the battery after the fuse and rocker switch. In this case, a 7.4 V LiPo battery powers the servo motors. The buck converter transforms the 7.4V input to a 5V output for the logic circuits. To ensure no constraints in the current flow, a sectioned-off copper sheet dedicated to 5V power is placed on the bottom signal layer and powers all associated components that require this voltage.

In the previous Squeaky version [41], the smaller buck converter was capable of supplying only 2.5A. During initial testing when creating Squeaky V3.0, this low current output became unsuitable for this new design because of higher power demands. These higher current demands were primarily due to the NVIDIA Jetson Nano and the perception peripherals. Hence, the Squeaky V1.0/V2.0 buck converter was replaced with a 5V 10A output converter. This buck converter was overspecified compared to strictly required, since the robot only pulls around 6.5 Amp. 6 Amp is the max of what the NVIDIA Jetson Nano will pull if

all peripherals and a computationally heavy algorithm are running. The remaining 4 Amps are reserved for the remaining logic circuits or for the addition of other sensors. The logic circuit's current maximum demand is 200 mA, considering the Teensy 4.0 is running at 600 MHz and pulling 100 mA.

The new buck converter presented a difficulty because it did not have a DIP variant for a direct connection that could be soldered into the PCB. Hence, the PCB added two pairs of terminal blocks, whose specifications meet the current needs of the logic power of Squeaky. The wire gauge is 12 AWG and was increased to ensure no limits are reached when powering. In earlier tests, smaller-gauge wires had issues maintaining the needed power transmission, where components did not receive their power demands during more intensive algorithms, leading to complete system failure.

3.2 Power Supply

The battery on Squeaky V3.0 is a HOOVO 2S LiPo 7.4V, 5000mAh 100C. This provides a high discharge rate and can handle rapidly changing power demands. The battery is connected to a simple PCB circuit, allowing power to pass through a fuse and a rocker switch to turn on/off the robot. As the battery is hard to detach from the PCB once placed, another port is angled outward to connect the charger cables to the board. This allows the battery to be charged through the board and does not need removal. As seen in Section 2.3, a complete disassembly of many components would be required to remove the battery. Hence, these connections are set up so that the fuse can be removed and the battery can be directly charged without sending power into the rest of the robot. The battery also has a signal cable connected to the cells, and the charger requires this cable to be inserted to start charging. This hangs off the side of Squeaky V3.0 as seen in Figure 3.3.

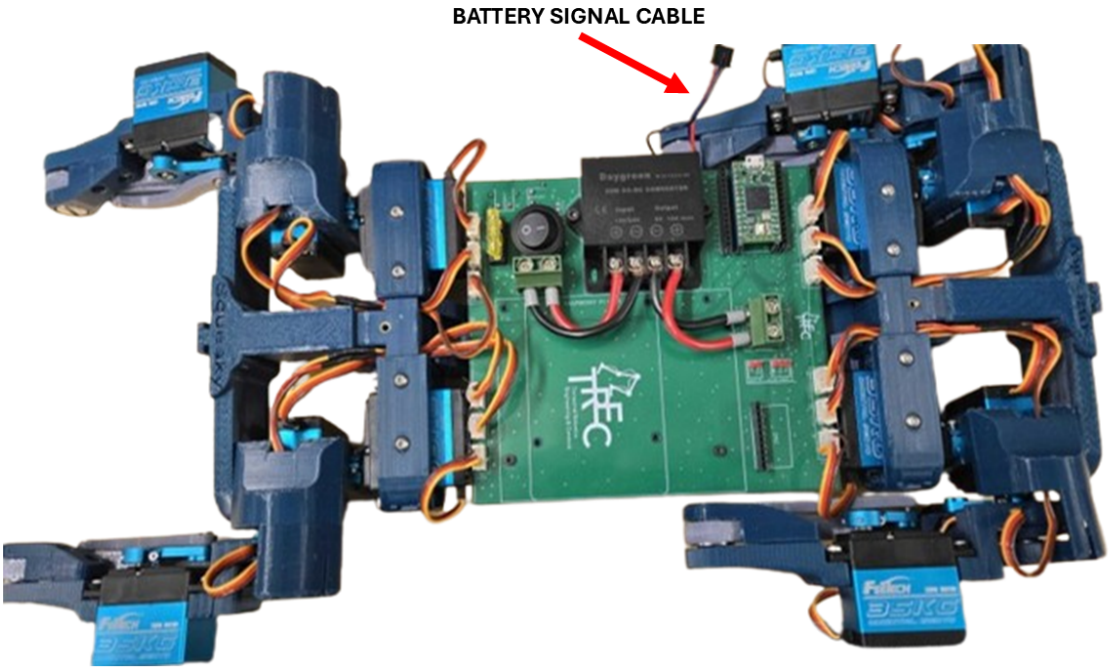

Figure 3.3: Signal Cable Labeled on Squeaky

3.3 PCB Connections

To command the servos, the PWM pins are connected to individual GPIO pins on a Teensy 4.0, which controls the locomotion of the entire robot. The custom PCB has a dedicated slot where the Teensy has been designed to be inserted. Once inserted, the appropriate pins connect the servo PWM lines to the required pins and the I2C communication pins for the NVIDIA Jetson Nano and IMU. This also includes the relevant power and ground pins. The Teensy connection over I2C to the NVIDIA Jetson Nano is where it receives commands. Hence, the NVIDIA Jetson Nano can be described as the ”brain” of the robot, as it controls everything other than the execution of the locomotion, which is what the Teensy role is. The NVIDIA Jetson Nano is also powered through the same battery as the rest of the robot, as there are 5 pins on the custom PCB, 2 for I2C (SCL and SDA), 2 for power, and 1 for ground as seen in Figure 3.1. A basic representation of the routing of the board is shown in

Figure 3.2, which showcases the power lines and simplified connections of data transmission to the NVIDIA Jetson Nano, IMU (MPU 9250), and servos.

In summary, the electrical system of Squeaky V3.0 was designed with accessibility, robustness, and expandability in mind. The custom PCB consolidates control and power management into a single platform, simplifying the internal wiring and supporting clean integration of components. The system ensures reliable operation even under demanding computational loads through clear separation of logic and motor power, careful current planning, and flexible connections for both the Teensy and NVIDIA Jetson Nano. This design supports Squeaky's role as a versatile and modifiable robotics platform for researchers and educators by providing a straightforward yet powerful electrical foundation.

Chapter 4

Software

This chapter presents an overview of the software architecture that governs Squeaky V3.0's functionality. The software is intentionally designed to be modular and accessible, enabling users to modify or expand the robot's capabilities with ease. Whether integrating new sensors, changing locomotion parameters, or adapting to different research goals, the system is structured to support seamless updates.

The motivation behind this design is to reduce the entry barrier for both researchers and educators by leveraging open-source tools, primarily to use ROS to handle control, visualization, and navigation tasks. This chapter breaks down the system into LL and High-Level (HL) components, clarifies their interactions, and provides step-by-step procedures for single and multi-robot operations. By outlining the software framework, it becomes easier for users to replicate, modify, and troubleshoot experiments across different deployments of Squeaky.

4.1 Low-Level System

This section breaks down the system framework, focusing on the LL system. A summary of this framework is shown in Figure 4.1, which can be divided into two parts: the LL system and the HL system. Each system has distinct tasks and priorities. This section will describe the LL system in detail and explain briefly the differences between the HL and LL systems. While Teensy servo calibrations will be mentioned, this will be discussed in more detail in

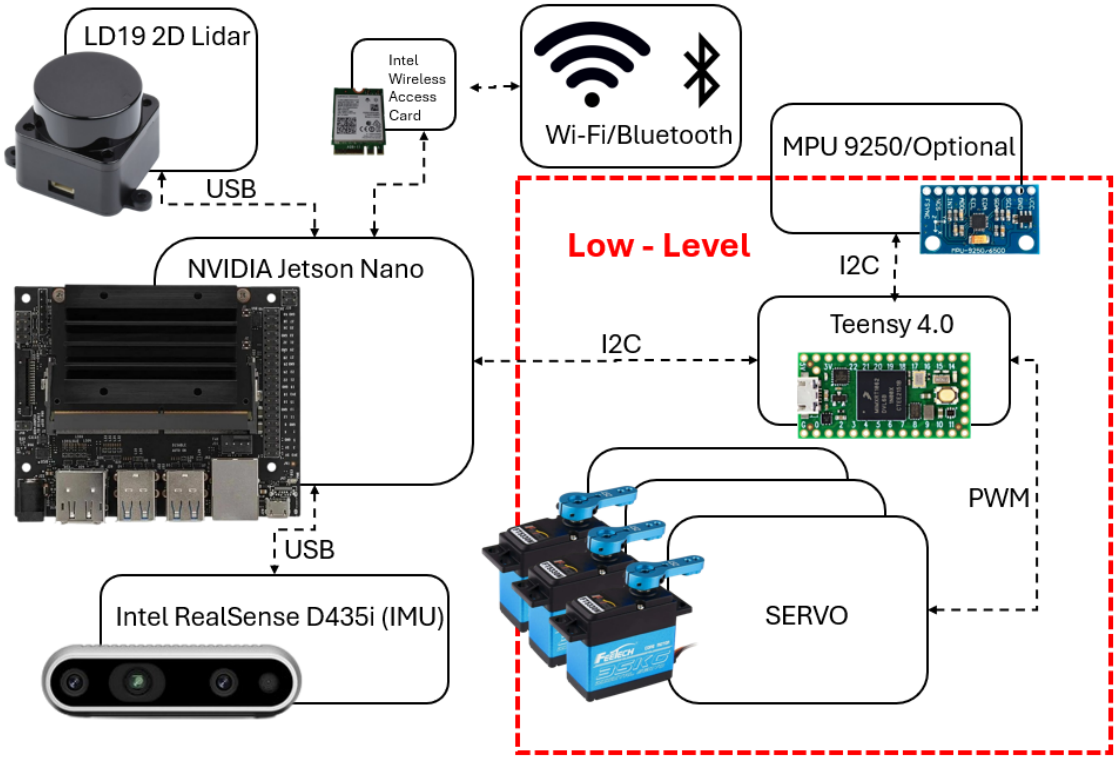

Figure 4.1: Overview of the system components for communication and control of Squeaky

Section 5.3. Lastly, the Arduino file in Appendix A.1 will be referenced, focusing on key code blocks and arrays.

The LL system is the Teensy 4.0 microcontroller, with its primary task of controlling the servo motors. Since these servos have a Range of Motion (ROM) of only 180 degrees, they must be calibrated by finding the zero value according to the servo and placing the servo horn in the correct position. Servos are absolute in their position, even though they are open-loop and provide no feedback. If calibrated in a known position, then the legs can be zeroed out and set to the desired position through the Teensy. An offset variable is also created within the LL to calibrate the servos to the final desired position. The calibration explanation and procedure are discussed in Section 5.3 in more detail. The reason for calibrating the servos in the Teensy is because of the ROS controller in the HL system. The HL contains a robot model that the real robot must mirror, especially since there is no feedback, leading to the

strict calibration procedures. The I2C line between the Teensy and the NVIDIA Jetson Nano is the connecting bridge between the HL and LL. The controller calculations are performed in the HL and output the desired angles that the servo motors must achieve. The desired angles of the motors are the only transmission over I2C between the NVIDIA Jetson Nano and Teensy.

The LL code includes serial calibration commands and an I2C interrupt with a few functions that parse the messages over the I2C bus. Then, a function takes the parsed data, sorts it into relevant arrays, applies the offset, and compares the min/max servo PWM before applying it to the servos. The void loop within the Teensy contains nothing but an if conditional that confirms that a flag has been triggered when the I2C interrupt occurs to execute the Leg motion. The robot will remain standing and await the next command if no command has been received.

Another essential matter is the Arduino file that is flashed to the Teensy of each Squeaky. Mainly since the file contains three arrays called offset, min, and max, which are tuned to each Squeaky's specific servos. For example, index 1 of the array is associated with the front left shoulder, 2: the front right thigh, 3: the front right shin, etc. Hence, updating these and labeling each file with the associated Squeaky is essential, as flashing the wrong code will give the robot the wrong set of min, max, and offsets. If this occurs, the robot will act erratically as it has the wrong tuning parameters.

4.2 High-Level System Sensors

The HL system performs various other critical tasks for experimentation, such as perception sensor input, IMU data processing/conversion, and many more tasks. Firstly, perception sensors are connected through USB on the NVIDIA Jetson Nano, allowing the robot to

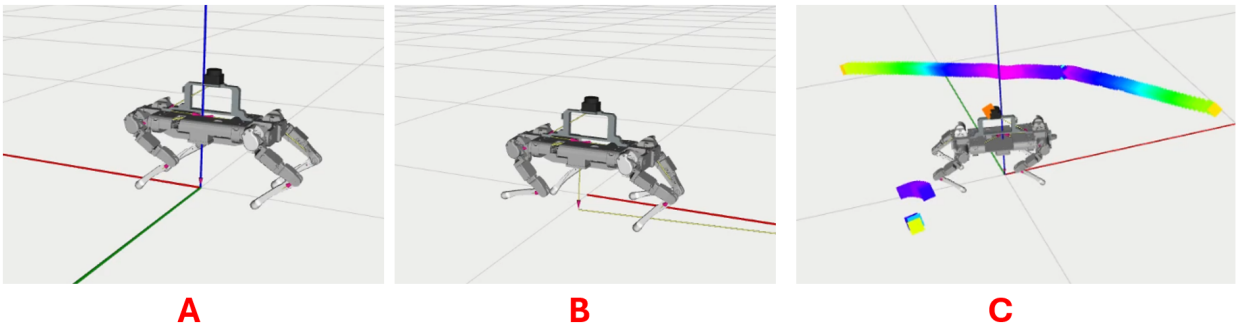

Figure 4.2: Comparison of Squeaky Standing (A), Walking (B), and RVIZ Output that shows lidar point array visualization(C).

perceive its surroundings. Driver nodes between the sensors and the NVIDIA Jetson Nano work for the data to be processed into the ROS framework, making it usable. In this case, the primary sensor is the 2D Innomaker LD19 lidar, which, as described in Section 2.4, is placed at the very apex of the robot. Another sensor includes the D435i Intel RealSense, a depth camera with an IMU. Currently, only the IMU is used to estimate the odometry needed for SLAM

4.3 ROS Visualization

A Universal Robot Description Format (URDF) is written to describe all the links and joints of the robot. It also describes sensor placement and the base link of the robot. All these are needed for ROS so that different nodes, such as ROS control, ROS visualization (RVIZ), and the navigational nodes, have a model of the physical robot. This virtual visualization will be used for the physical robot and also set up for the simulation version of the robot. Since Squeaky V3.0 has been designed in CAD software, its model can be converted into a URDF version using a tool called URDF exporter. This tool will create all the necessary joints, joint types, links, and axes of rotation for each joint and linkage.

Once exported, minor changes will need to be made to the file depending on specific use

cases. If planning to use this URDF model for simulation, such as in Gazebo, the collisions must be changed under each link to be simplified into boxes and spheres. Otherwise, a large increase in the necessary processing power will be required.

Another joint/link combination that needed to be written was to define a little box within the lidar at the expected location of the scans. This is primarily because the laser scan topic in ROS will be placed along the XY coordinate plane of the base link if not specified. This initially occurred when using the URDF for the first time without making the necessary adjustments. The specific code block is shown here for the joint/link configuration:

```
<!-- <joint name="laser" type="fixed">
  <origin rpy="0 0 0" xyz="0 0 0.027"/>
  <parent link="LD06_Lidar_Link"/>
  <child link="base_laser"/>
</joint>
<link name="base_laser">
  <inertial>
    <mass value="0"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <inertia ixx="0" ixy="0" ixz="0" iyy="0" iyz="0" izz="0"/>
  </inertial>
  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <box size="0.005 0.005 0.005"/>
    </geometry>
  </visual>
</link> -->
```

A critical component of this code block is that the joint name must be laser. This will associate the laser scan frame with the appropriate joint link configuration. Hence, RVIZ

will show the laser scan from the lidar on the correct plane as seen in Figure 4.2.

Lastly, when testing the laser frame, it may not be oriented as expected in RVIZ. For example, the real robot may scan a wall in front of it, but RVIZ shows the wall to the right side. As seen in the previous code block, the ”rpy” values may need to be adjusted under the joint configuration. As seen within this previously displayed code block, the ”rpy” values did not need to be adjusted.

The next addition will include adding the IMU joint/link configuration to be placed at the appropriate location. When working with the D435i software that the Intel RealSense team provided, it worked on x86 architectures without issue, but when ported to the NVIDIA Jetson Nano, issues occurred, such as incorrect rotation of the robot model compared to the real robot. This error was solved and is believed to have happened because of the extra transforms published from Intel's RealSense camera launch file. To fix it, the transforms were disabled within the hardware_multi launch file within the rs_camera launch parameter, where the argument publish_tf was set to false, as seen here:

```
<arg name="publish_tf" value="false"/>
```

Once disabled, the robot model mimics the rotation of the real robot. This was a critical error to fix as it would result in the mapping to fail since the odometry frame would be inverted from the real robot, causing each new scan to create an incorrect overlay on the previous iteration of the map.

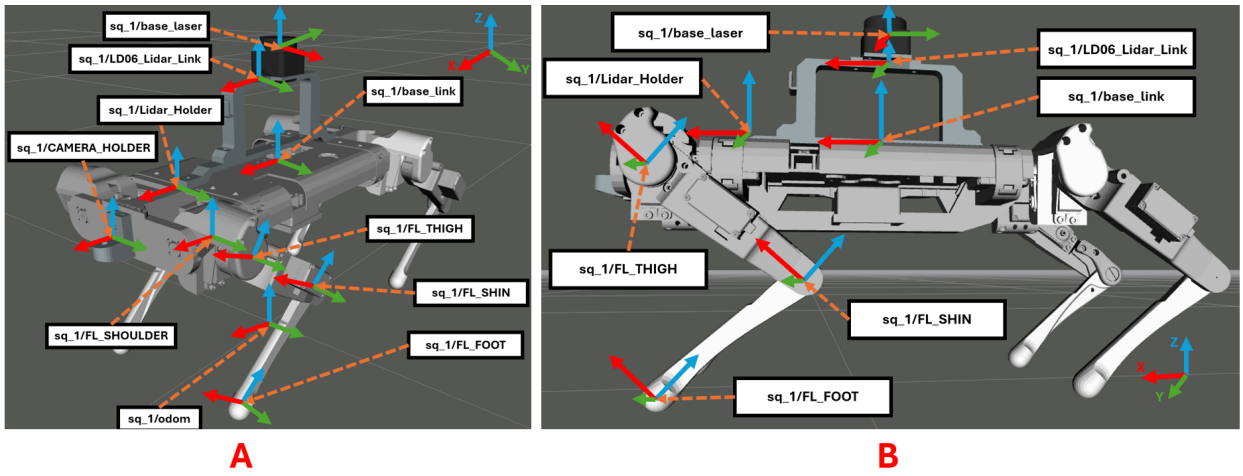

Figure 4.3: A Isometric (A) and Side View (B) of URDF Squeaky labeled with associated namespaced frames.

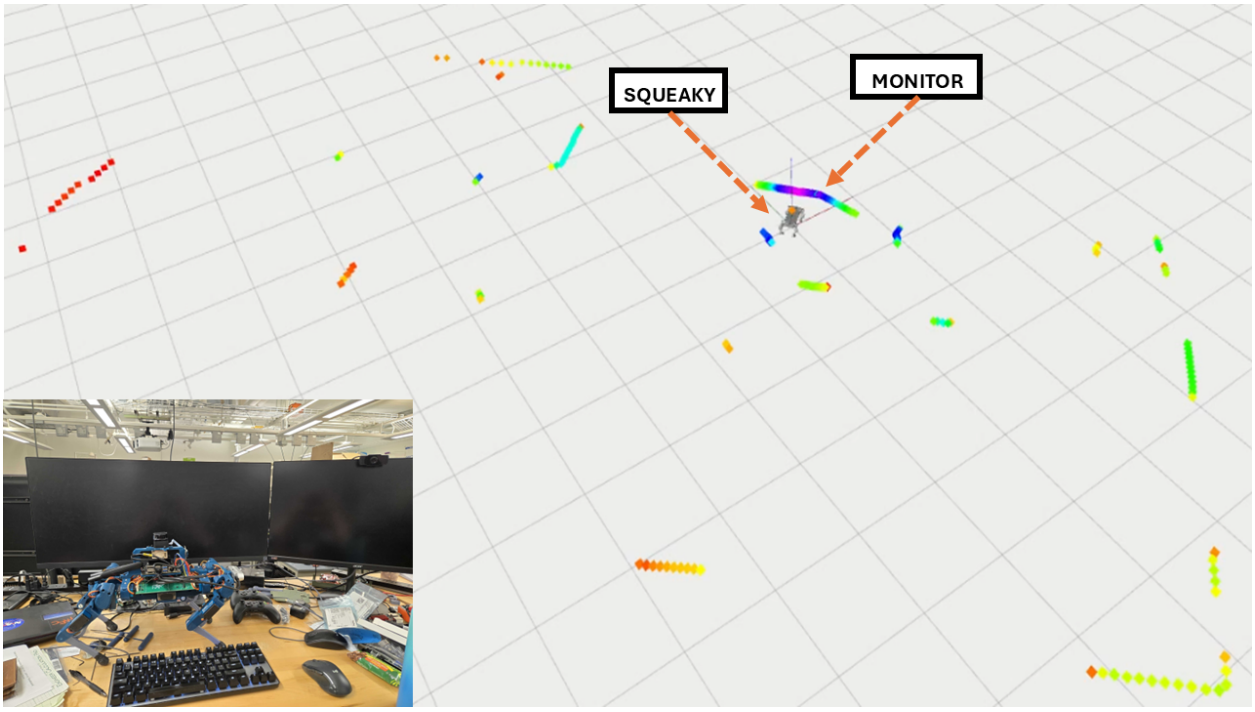

Figure 4.4: RVIZ is shown in the background with a picture of Squeaky showing the environments that the lidar is currently visualizing.

These changes mentioned were the main adjustments that needed to be made within the URDF on the public GitLab [1]. The robot URDF also contains commented-out sections that include using the D435i as a laser scanner rather than the lidar. Currently, only one of the

[1]This robot is open-sourced and available at the TREC gitlab: gitlab.com/trec-lab.

laser scanners, the D435i or the lidar, can be run simultaneously within the URDF to avoid ROS incompatibility errors. This also includes starting a Depth Camera to a laser scanner node to take the 3D point cloud that the D435i outputs and convert it to a laser scan.

For reference, Figure 4.3 shows the frames' location and orientation on an isometric and side view of Squeaky with the frames labeled. Only one leg is labeled to reduce clutter, but each leg is configured similarly. This visualization of Squeaky is shown from the RVIZ model that the HL expects.

For example, the inertia must be added and defined to have the URDF work with Gazebo, ROS simulation software. This robot is not set up with Gazebo and has only been tested in real environments. Because of this, the focus was on setting up RVIZ correctly to visualize and confirm that the ROS controller was working as expected and that sensors were broadcasting. A visualization of Squeaky URDF and sensor data can be seen within Figure 4.2 image C. A picture of Squeaky in the real world is in the corner of Figure 4.4 to compare the visualization of Squeaky's lidar scan. Since RVIZ can be set up to visualize SLAM algorithms from the sensor data, frames of reference can also be set up, including the robot's world frame and base frame, which is essential since the robot's odometry will be tracked throughout the mapping.

Lastly, an open-source package called CHVMP[2] is used to ensure that this robot interfaces correctly. Another package called CHVMP setup assistant[3] is used to create the necessary files. This is relatively intuitive in how it works, as a guide is within the README, and a GUI walks the user through the process. The output of this package is a group of files and the structure of the custom robot used to interface with CHVMP controllers, state estimation, and other nodes within the primary package.

[2]This framework is open-sourced and available on GitHub: github.com/chvmp/champ.
[3]This framework is open-sourced and available on github: github.com/chvmp/champ_setup_assistant.

4.4 High Level Overview

The NVIDIA Jetson Nano runs Ubuntu 18.04 and ROS Melodic. This version was chosen because, at the time, it had many fully integrated packages and was straightforward to implement, serving as a strong foundation for robotic development. This version was used over ROS2 because, when the project started, the ROS2 versions of the same packages were not fully updated or integrated. In this case, three primary packages are used, each with many dependencies: CHVMP, SLAM Toolbox, and M-Explore. CHVMP is an open-source quadruped integration framework that makes it smoother and faster to integrate new quadrupeds into full functionality. It also has features that include state estimation, ROS control, basic SLAM algorithms, move base for path planning, and name spacing for multiple robotic systems on the same ROS network.

ROS was also chosen as the main operating system for Squeaky V3.0 because of its modularity in switching algorithms. For example, the SLAM algorithm GMapping that was initially implemented in the CHVMP package can be easily removed and replaced with another. The new algorithm can then be immediately integrated for testing to see if it follows ROS conventions.

Hence, the SLAM algorithm chosen for this project and integrated was SLAM toolbox. It is a newer integration of SLAM than Gmapping. Implementation of SLAM toolbox required adjusting the namespacing primarily once this project expanded into more multi-robotic applications under one ROS network. Otherwise, initial integration once GMapping was replaced took little to no time. A larger amount of time was spent understanding this toolbox's new features that previous algorithms did not have.

Lastly, a node was written in ROS to handle the I2C bus between the microcontroller and the NVIDIA Jetson Nano. The primary function of this node, when activated:

1. Establish connection to Teensy
2. Record the controller joint states and package them into an array
3. Transmit desired angles to Teensy
4. repeat 2-3

The node does not receive data from the Teensy because there is no feedback from the motors or sensors.

Lastly, on TREC's public GitLab for Squeaky, under Software, there are three folders labeled High-level computer, Jetson, and Teensy. Focusing on the main computer and Jetson, the main computer squeaky_ros1 goes into the catkin workspace on the main computer. The squeaky_ros1 under the Jetson goes to the two different Squeaky robots' catkin workspace. The primary difference is that the Jetson squeaky_ros1 packages are adjusted with appropriate name-spacing. These can serve as an example of what needs to be adjusted if this is to be expanded to more robots.

4.5 Squeaky High Level Startup Procedure for Initial Test

Currently, on startup, four ROS launch files are utilized; in this case, the first two launch files are critical for overall robot control, while the other two are optional, depending on what the application is. The first launch file is for the D435i Camera node, LD19 lidar, and I2C transmitter node to the LL. The second launch file contains the overall robot system, which utilizes the CHVMP framework, an open-source GitHub for ROS robots in quadruped

control.[4] This launch will include many algorithms, such as a ROS controller configuration and state estimation for the robot.

The third node to launch is a Squeaky's teleoperation node that takes in a joystick command and outputs a command velocity to control the robot. Then, the fourth node will launch. Through this same package, GMapping will be started, and a simple 2D mapping of the environment can be made. This allows all sensor configurations to be tested to confirm that the robot is operating as expected. Afterwards, it can be replaced with more sophisticated SLAM algorithms depending on the sensors that are included on the robot and what the operator's goals are.

Previously, it was mentioned that the third and fourth nodes are optional. This means that they can be easily replaced. For example, the teleoperation node can be omitted if more autonomous algorithms are being tested and joystick commands are not needed. The fourth node, which handles mapping, can be replaced with other mapping algorithms as necessary.

The sophisticated SLAM algorithm used is SLAM toolbox. Instead of launching the Gmapping algorithm, an adjusted launch file is inside the public GIT for Squeaky under squeaky_nodes called "online_async_squeaky.launch". This will run the mapping for SLAM toolbox, and RVIZ may need to be enabled for visualization purposes when testing.

The D435i IMU is filtered, and the process is described as follows. First, it is received through a driver package by INTEL RealSense for ROS, which outputs a raw acceleration and gyration frame. Then, a Madgwick filter node gets the raw data and outputs an orientation. This is used in robotic odometry, which is required before using SLAM algorithms such as GMapping and the SLAM toolbox.

[4]This framework is open-sourced and available on github: github.com/chvmp/champ.

4.6 Squeaky High Level Startup Procedure for Multi-Robotics

To perform multi-robotics, ROS allows the master to be run over a WiFi network. This was utilized to collaborate with robots and merge each robot's internal map into a single merged map. This section will describe the startup procedure needed to start these systems.

1. Turn on the main computer, Squeaky 1, and Squeaky 2.

2. On the main computer, open up 10 terminals and organize them in groups of 2, 4, and 4. This is done as the two groupings of four will represent the two Squeakys, and the grouping of 2 will be run on the main computer.

3. SSH into each Squeaky robot. This will be performed in all 4 of the command terminals per robot.

4. Turn on the Xbox controllers, which should directly connect to each NVIDIA Jetson Nano.

5. In the Main computer terminal, export the ROS_MASTER_URI with the master computer IP address and 11311. Example: "export ROS_MASTER_URI=http://mia:11311/" where mia is my IP address.

6. Now use the same in step 5, but this time export the ROS MASTER URI into each Squeaky terminal. This will have the NVIDIA Jetson Nano look for the ROS master to be run from the main computer

7. On the first Squeaky terminal for each robot, give permission using chmod for the lidar USB bus and the I2C bus. This will give permission for the ROS to read/write

over this communication bus. Specifically, this will allow the NVIDIA Jetson Nano to communicate with the Teensy and Lidar.

8. Now launch the roscore in the 1st main computer command terminal.

9. To check that the Jetson can see the roscore on the main computer, use ”rostopic list” to display current topics. If topics appear, then the Jetson is connected with the ROS master on the main computer.

10. On the first Squeaky terminal, launch ”roslaunch squeaky_nodes hardware_multi.launch”. This launches the relevant D435i Camera, IMU filter, LD06 Driver, and hardware transmitter nodes.

11. On the second Squeaky terminal, launch ”roslaunch squeaky_config bringup_multi.launch”. This will launch all the main quadruped nodes that CHVMP provides for Squeaky.

12. On the third Squeaky terminal, launch ”roslaunch squeaky_teleop teleop.launch joy:=true”. This launches the node that talks with the Xbox controller and interfaces to give velocity commands to the ROS controller.

13. On the fourth Squeaky terminal, launch ”roslaunch squeaky_nodes online_async_multi.launch”. This will launch an asynchronous mapping mode for SLAM toolbox with the necessary parameters.

14. Repeat steps 10-13 for the other 4 Squeaky terminals for the second robot.

15. Squeaky 1 and 2 should be running and mapping at this step.

16. On the last command terminal that is for the main computer, now launch the map merging node. Use the command ”roslaunch squeaky_nodes multi_robot_merge.launch”. This will launch the merging node, and RVIZ will open on your main computer.

17. To begin testing, start controlling each robot and map the area. Once enough similar features are made, the map merging will combine each robot's individual map.

As directed, this is the startup procedure to perform experiments in multi-robot mapping for the Squeaky repository on TREC's public gitlab[5].

In conclusion, the software framework of Squeaky V3.0 has been built to balance technical depth with accessibility. From LL servo control and calibration via the Teensy microcontroller to HL tasks such as SLAM, visualization, and multi-robot coordination through ROS, each layer is designed to be transparent and modifiable. The startup procedures for both single and multi-robot setups illustrate the flexibility and scalability of the system. By combining a well-structured ROS environment with hardware integration, this software design ensures that Squeaky can serve as a robust and adaptable platform for continued development, research, and education.

[5]TREC's gitlab, Squeaky HL files: Software Files gitlab.

Chapter 5

Assembly & Cost

This chapter provides a comprehensive guide for assembling the Squeaky quadruped robot and understanding its cost structure. It begins with an overview of the public repository and a breakdown of how to set up the software environment, ensuring that users can successfully install and configure the necessary dependencies. Following this, the physical assembly process is covered in detail, including key hardware considerations such as servo wiring, leg construction, and integrating the PCB and sensors. Step-by-step instructions and annotated figures are provided to simplify the build process and support first-time users.

Next, a thorough calibration procedure is outlined to align the real robot's joint positions with the expectations of the HL software system. Given the actuators' open-loop nature, this calibration is essential for ensuring accurate and safe operation.

The chapter concludes with a cost analysis comparing Squeaky to other open-source quadrupeds such as Stanford Pupper and Doggo. A full bill of materials highlights how Squeaky remains cost-effective while maintaining the necessary capabilities for real-world SLAM, navigation, and multi-robot research. This chapter is designed to make the platform easy to build, calibrate, and deploy, supporting its broader goal as an accessible tool for robotics research and education.

5.1 Robot Repositories & Setup

This section provides a detailed breakdown of the Squeaky repository hosted on TREC's public GitLab.[1] It begins by outlining the structure of the repository, followed by step-by-step instructions for setting up the high-level configuration, installing required dependencies, and resolving common errors. The section concludes with an overview of the Arduino folders containing the Teensy code, explaining the file separation and its importance for hardware-specific calibration.

This section serves as a practical guide and is intended to make the Squeaky platform more accessible to a broader audience. Clear documentation and troubleshooting support reinforce the project's open-source goals and lower the entry barrier for new users. Ultimately, well-written and user-friendly setup instructions ensure that Squeaky can be quickly adopted and expanded upon by researchers and educators alike.

All files for the Squeaky robot have been consolidated into a single primary repository hosted on TREC's public GitLab. The repository comprises three main subfolders: Mechanical, Electrical, and Software. The Mechanical folder includes all CAD parts and assemblies for the robot, while the Electrical folder contains the Autodesk Eagle files for Squeaky's custom-designed PCB.

Lastly, the software folder has three subfolders: High Level Computer, Jetson, and Teensy. The High Level Computer and Jetson folders contain the squeaky_ros1 package, which includes several other subfolders, such as Squeaky's URDF, launch files, and custom ROS nodes. A key point is that the Jetson folder is divided into two subfolders for different robots. This has been configured as such because of the need to namespace the robot files for the shared ROS Master.

[1] This robot is open-sourced and available at the TREC gitlab: gitlab.com/trec-lab.

Navigating to the squeaky_nodes folder and then into the source folder will reveal an ROS node called hardware_listener.py. This node was developed to send joint commands from the high-level computer to the microcontroller. Specifically, it subscribes to the joint_group_pos_controller/command topic, processes, and packages the desired joint angles before sending them to the Teensy.

As previously stated, the Main Computer and Jetson folder only contains the squeaky_ros1 package. This means that it does not include external packages used, such as the D435i ROS wrapper, madgwick IMU filter, CHVMP framework, SLAM toolbox, and m-explore, which are a requirement to work with squeaky_ros1. When cloning the repositories from GitHub, confirm they are from the ROS Melodic branch. Before building the catkin workspace, the necessary dependencies must be installed. To complete this task, run the following command: "**rosdep install –from-paths src –ignore-src -r -y**" This will install the latest versions of all required packages. However, this may cause issues with the Intel RealSense D435i, as the newest firmware is incompatible with older versions of the RealSense library. To resolve this, the D435i firmware must be downgraded to version **5.13.0.50**, which is compatible with **Ubuntu 18.04**. After that, make sure to use version **v2.50.0** of the librealsense package and version **2.3.2** of the **RealSense ROS Wrapper**.

Once the repositories and dependencies are successfully configured, testing that the IMU from the D435i is working properly is important. To test the D435i, execute the command that Intel RealSense recommends below:

```
roslaunch realsense2_camera rs_camera.launch
enable_accel:=true enable_gyro:=true
unite_imu_method:=linear_interpolation
```

The previous command is a single line of instruction that will be pasted into the command terminal. Once entered, check that the topic **/camera/imu** shows data by echoing it within

the terminal. The data will be published in the terminal if everything works as expected.

In RViz, this can also be visually confirmed by adjusting the frame of reference to the odom frame. The IMU topic can then be easily visualized through a coordinate frame that rotates in the same direction as the camera. This method is valuable when troubleshooting the IMU if issues occur.

Returning to the Software folder, the Teensy subfolder contains Arduino code uploaded to the microcontrollers on the quadruped. The structure is also divided into two robot folders, similar to the organization mentioned in the Jetson folder. This is important because the Arduino files contain servo calibrations specific to each robot, which will be discussed in Section 5.3.

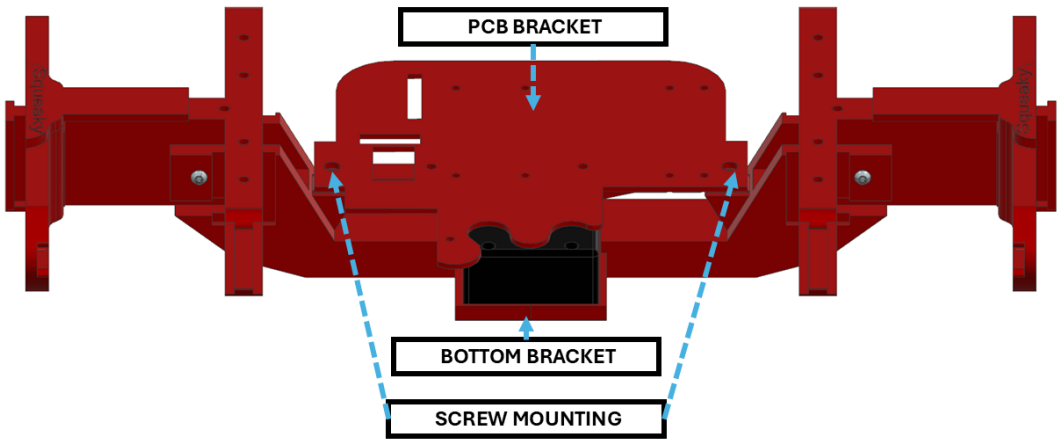

Figure 5.1: Main parts on Body labeled for assembly.

5.2 Assembly

Assembling Squeaky is considerably more straightforward compared to the previous quadrupeds mentioned in the Introduction, though some of the more intricate components may still pose challenges. Overall, the time required to build one robot, including 3D printing, is no more than a week. This also includes adding the necessary M3 heated inserts into each 3D-printed part using a soldering iron. This section will provide an overview of assembling the body and legs, as a first-time assembly of this robot may not be intuitive to some users.

Assembly begins with the robot body, starting with the bottom bracket that secures the battery. This bracket forms the structure's base, with the battery inserted into the cut-out holding area in the bottom bracket. Next, the PCB holding bracket is installed, which not only supports the main circuit board but also clamps the battery securely in place using two screw points that connect it to the bottom bracket. Before fully mounting the PCB, it's essential to connect the battery to the XT30 connector on the underside of the board, as this connection becomes challenging to access once the PCB is fully seated.

While working through the directions below, utilize Figure 5.1 as a reference. Beginning with the robot body:

1. Combine the bottom bracket with the battery in the designated slot.
2. Screw in the PCB bracket to the two screw points.
3. Attach the XT30 connector on the bottom of the PCB to the battery. It is angled towards the battery compared to the XT30, which is angled outwards.
4. Secure the PCB to the PCB bracket.

The following four pieces of the main body structure are straightforward to install, as their

design is mirrored on both sides. The T-shaped bracket, identifiable by the holes on its top surface, is slid into the gap along the sides of the main body and secured with screws from both sides and the bottom. Once in place, the front and rear plates are attached by sliding them over the exposed slots and securing them with screws from either the front or back, depending on which side is being assembled. With all six pieces installed, the main chassis structure is complete.

Before adding the servos to their 3D-printed part, the connectors that come out of the box do not attach to the Custom PCB. The specific connectors required are Molex Mini SPOX. Below are step-by-step directions for creating the necessary connections.

1. Remove the previous connectors.
2. For servos that are furthest from the PCB, it is recommended to modify the closest servo wire by cutting it to the shortest length that can reach the PCB.
3. Solder the shortened wire to the shin servo wire to extend it, ensuring the wires are long enough to reach the PCB.
4. Add the crimps to each wire.
5. Put the crimps into the Mini SPOX connectors.

It is recommended that the leg assembly be started from the thigh. This is because the shin is a single structural piece that doesn't require any additional components to be assembled until it is time to mount it to the thigh. The attachment point on the shin is at the rotational joint with the thigh and the servo horn that controls its motion. The following steps are for assembling the thigh.

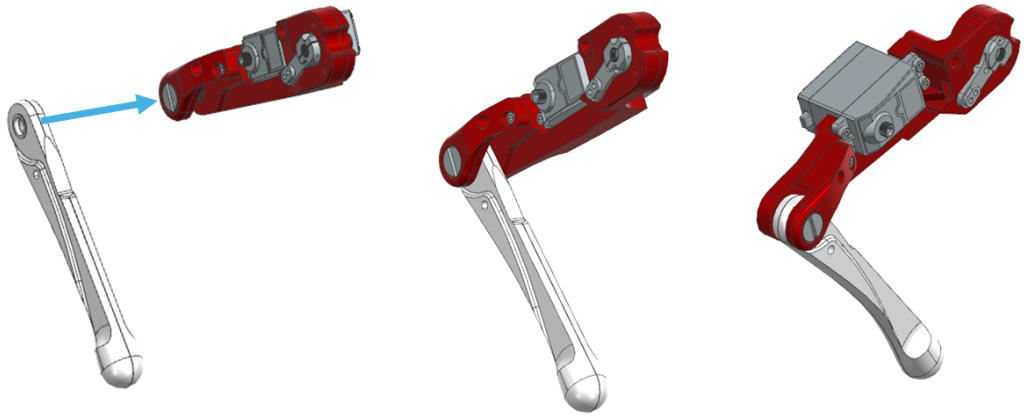

Figure 5.2: Shin's main connection to the thigh at the bolt.

Thigh Directions:

1. Secure the shin servo into the thigh as seen in Figure 5.2

2. Move to the top of the thigh where the shoulder servo is, then press in the servo horn for the servo that rotates the thigh. The other end of the horn should align with a hole which a M3 screw from the opposite side will screw the horn to the thigh.

3. Next to the horn where the servo screws in, a bearing will be placed to allow smooth rotation of this joint. This can be seen in image C in Figure 5.3 marked as Bearing Placement.

As seen within Figure 5.3, the thigh holds the servo that rotates the shin. This assembly starts at the rotating joint, takes a bearing placed on the thigh, then a bolt that will screw into a insert that holds a bearing from the shin to rotate smoothly.

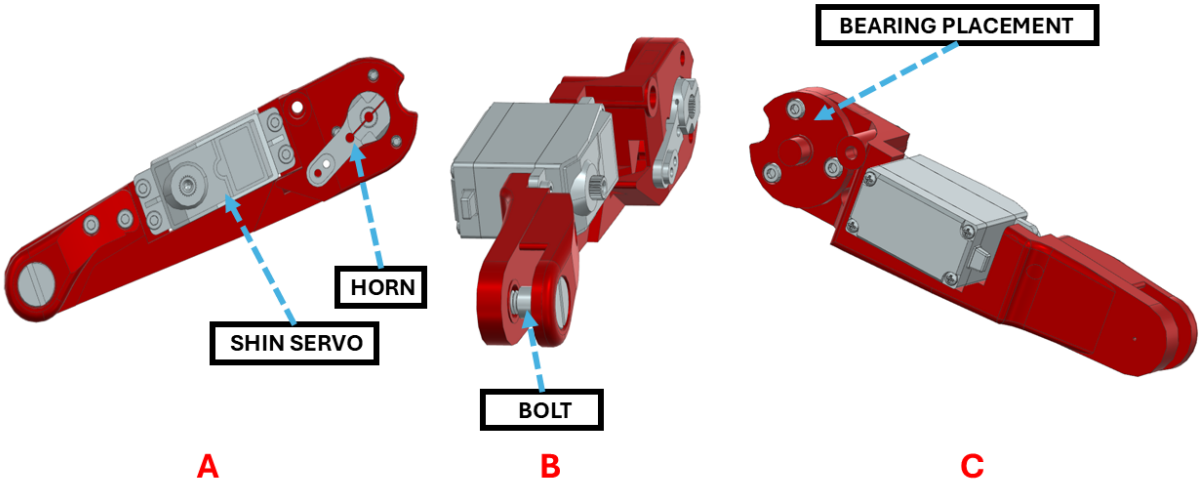

Figure 5.3: Labels on important components & component placements on the thigh.

Shoulder Directions:

1. Attach the servo that handles the rotation of the x-axis and will also connect with the servo horn that is press-fitted into the associated piece first, as seen in Figure 5.5. Figure 5.4 shows a closer visual of the two components to be connected. At this point, the assembly is fairly straightforward by building down, adding servos to necessary slots and associated bearings, then connecting the shoulder to the thigh servo as seen in Figure 5.6, enclose, and the leg is built.

The final connection for the leg is to slide it into place on the body, add an associated bearing, and a bolt to allow for rotation, which fixes it to the axis. This is seen in Figure 5.7 Then string through the wires in a small gap that leads to the PCB and servo slots as you push the shoulder into place, so before adding the screw to fix. Then, attach the servos to the associated connectors, and one leg is done.

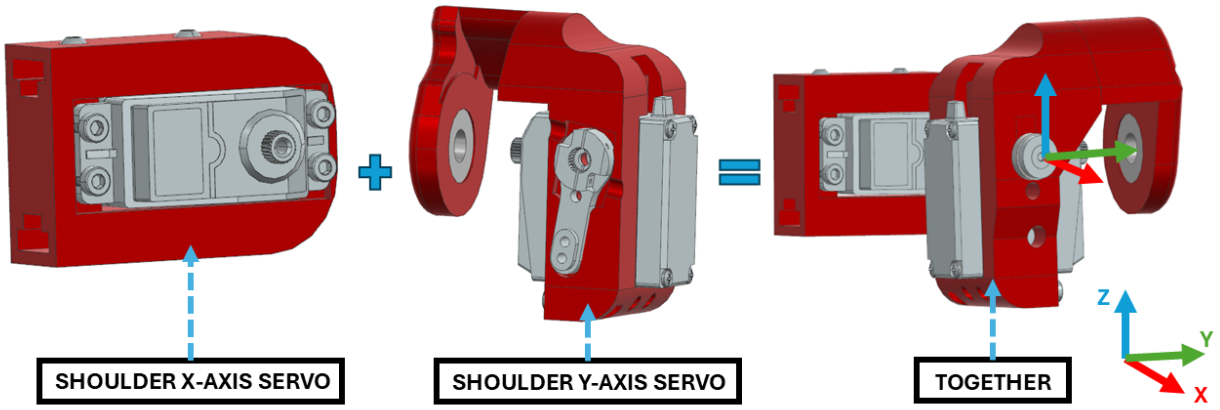

Figure 5.4: Labels on important components on the shoulder.

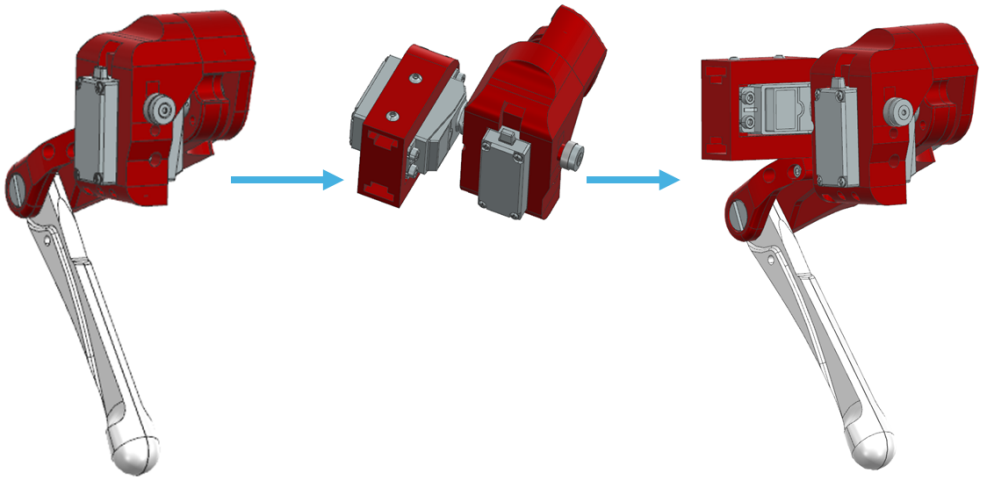

Figure 5.5: Both shoulder servos connected.

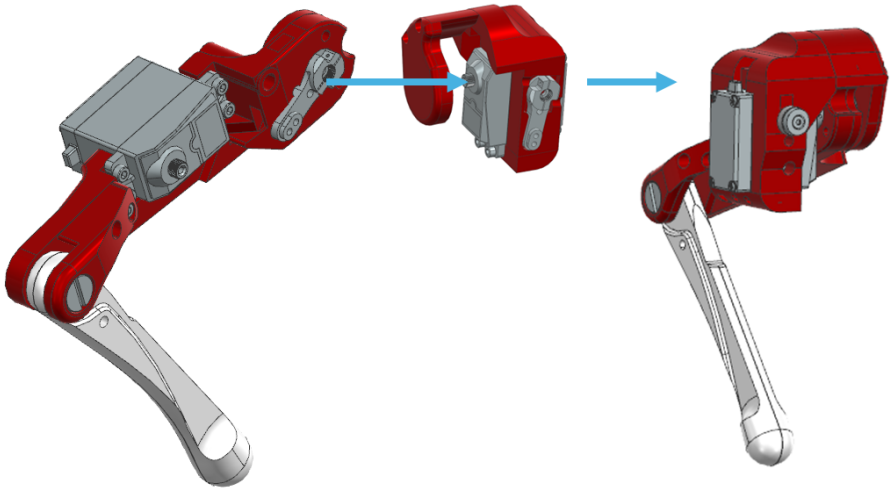

Figure 5.6: Thigh first connection to shoulder directly to servo, note that the other shoulder that handles x-axis rotation would also be attached. It is removed to see this connection visually.

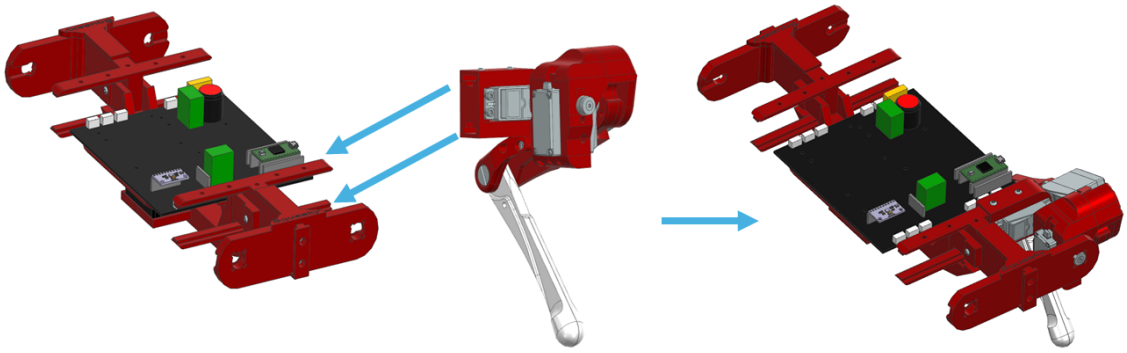

Figure 5.7: Leg connected to rest of robot body.

The rest of the legs are assembled the same way as previously described.

Lastly, these 3D-printed brackets will also use the M3 inserts to attach the navigational equipment and then be bolted in place. This assembly is straightforward.

5.3 Calibrations

This section outlines the calibration procedure for Squeaky, providing a step-by-step guide to properly align the servo positions with the expectations of the HL system. It begins with a detailed explanation of the setup required to perform the calibration safely and effectively. Following this, Method 1 is presented in a sequential format, while Method 2, which follows a similar approach but uses a different command set and configuration, is introduced as an alternative.

As discussed in Section 4.1, the motivation for this procedure stems from Squeaky's servos operating without feedback. As a result, the HL system assumes the LL system is achieving the commanded positions, even though it has no way to verify them. If the servos are not calibrated correctly, this mismatch can lead to movements that exceed the mechanical range of the joints, potentially damaging components. This risk is exceptionally high when the servos are rotated while unpowered and then suddenly return to their last known position upon activation. Proper calibration is therefore essential to ensure safe and accurate operation.

The leg must be disassembled to begin the calibration, and the horn must be removed from the servo. The horn's orientation is crucial; removing it lets the servo shaft be rotated to a known position electronically. Once this desired orientation is achieved, the horn is reattached, allowing Squeaky to have a full range of motion at each joint. The calibration is completed after assembly because of the need to ensure that the fits and orientations of

the motors are verified before calibrating for the range of motion.

When setting up to perform these calibrations, disconnect the wires from the buck converter and the terminal blocks. This is necessary because the Teensy will be connected directly to the laptop via USB for manual adjustments. The computer will power the microcontroller and allow the servos to be manually adjusted through serial communication. The specific commands are in the Teensy Arduino code provided in Appendix A.1.

Now, a brief explanation of the commands important for calibration will be provided. The first command is:

```
theta,servoPin,new_angle
```

Where "theta" is to call the command, "servoPin" to select a specific servo, and "new_angle" allows the user to change DesiredPWM_HL directly. DesiredPWM_HL is an array with preset servo values centered around 90 degrees or 45 degrees, depending on the servo location on the leg. This allows the user to change the servo's specific position when calibrating easily.

The second command is:

```
off,servoPin,angle_offset
```

In this command, "off" refers to the requested operation, "servoPin" selects the specific servo, and "angle_offset" defines the offset applied to the DesiredPWM_HL. DesiredPWM_HL is an array that holds the desired angles, which gets updated whenever the HL sends new values. This command is used to adjust the offsets through a range of values directly and then manually store them in the offset array when the real robot matches the model defined by the HL.

The purpose of the two commands explained earlier is to adjust specific arrays that store new calibration values within lines 63-72 of the "ros_squeaky.ino" file, shared in Teensy

Code Section A.1 of Appendix 1. Specifically, lines 65-67 define minPWM, maxPWM, and invertServo. For both minPWM and maxPWM, these two arrays define the minimum and maximum ranges that the servo can rotate to. These arrays should be set to 0, tailored to each specific servo during calibrations.

Before diving into the role of InvertServo, it's important first to introduce an essential formula:

```
MotorRotation = offset[i] + Desired_PWM_HL[i]
```

In this formula, MotorRotation is the array that holds the adjusted PWM values after applying each specific servo offset. The offset and Desired_PWM_HL were discussed earlier; to reiterate, Desired_PWM_HL represents the initial HL angles, and the offsets are applied to align the real robot with the HL model.

Building on this, the invertServo array consists of 12 elements, each containing either true or false. When an element in invertServo is true, it triggers a conditional check where the corresponding element in motorRotation is subtracted from 180 and recorded in an array called ServoRotation. Each of the 12 components in motorRotation goes through this check, either subtracted or unchanged. ServoRotation is the final array before the angles are applied to the servos.

With the background calculations for the different arrays described, the next step is to prepare the robot for calibration. This involves changing the pin assignments for all servos in the LL code to non-existent pins, except for the one currently being calibrated. This modification is made in the servoPin array on line 25 of ros_squeaky.ino, as shown in Appendix A.1. Another necessary setup of the servoPin array is that there are 12 elements, each is divided into leg groupings where leg 1 is array indices 0-2, leg 2 is array indices 3-5, and so forth. Below is the relevant code snippet:

```
const int servoPins[12] = {4, 3, 2, 5, 6, 7, 14, 9, 8, 15, 20, 21};
```

For example, when calibrating servo 1 (pin 4), the other pins in the array are changed to arbitrary values that exceed the Teensy PINs. The code will be re-uploaded to the Teensy as each servo is iterated through the calibration steps.

Before powering the servos with the rocker switch, ensure the cables from the buck converter to the LL hardware are disconnected. As mentioned earlier, power will be supplied by the USB connection to the Teensy for the logic circuits. This precaution will prevent potential issues arising from using two separate power sources. The following set of directions outlines the steps for iterating through this process:

Directions:

1. Confirm that minPWM, maxPWM, and offset are set to zero for the servo currently being calibrated,
2. Remove the servo horn from the servo, starting with the shin servo, and then turn the servo on.
3. Use the theta command in the Serial Monitor in the following format: "theta,servo_number,angle". This command calls the theta function, which is parsed and then sets the servo number (between 0-11) and a specific angle (between 0-180). For this case, starting from the shin, set theta to 10, which moves the servo to its zero position.
4. **Note:** This setting is specific to the shin, as the shin cannot fully extend to the expected high-level position. This can be adjusted in the offset array for these specific servos to account for this in the LL code.
5. Reattach the horn to the servo.

6. Test the rotation of the servo. If the servo rotates in the wrong direction when a command is entered and tries to exceed its range, this indicates that the invertServo setting should be adjusted.

7. Only proceed to step 8 if the servo rotated in the incorrect direction in step 6. Otherwise, proceed to step 9.

8. Remove the horn, set invertServo for that specific servo to true, then enter the theta command with the value 10 again. When invertServo is set to true, this will subtract 10 from 180. Reattach the horn in the same position as before, then test again. The servo should now rotate as expected, completing the final parts of calibration.

9. Now that the horn is placed within the correct range of motion for the leg, the next steps involve adjusting the 3 arrays within the LL code for the robot.

10. **Note:** This is the purpose of the min, max, and offset arrays. minPWM and maxPWM protect the robot from exceeding ranges that could damage the hardware. Since the high-level code does not handle these limits, they are encoded in the LL code to ensure the motor positions stay within safe bounds.

11. Using the theta command, move the robot to its minimum and maximum range. Record the specific theta values and add them to the minPWM and maxPWM arrays.

12. **Note:** The offset array is the most important, as it sets the reference point to ensure the LL code matches the HL model. The HL model expects the robot to have fully straightened legs, from shoulder to foot. However, the primary constraint of the real Squeaky robot is that it cannot fully extend its shin due to the limitations of the shin servo motor. An offset can be applied to match the expected configuration. This

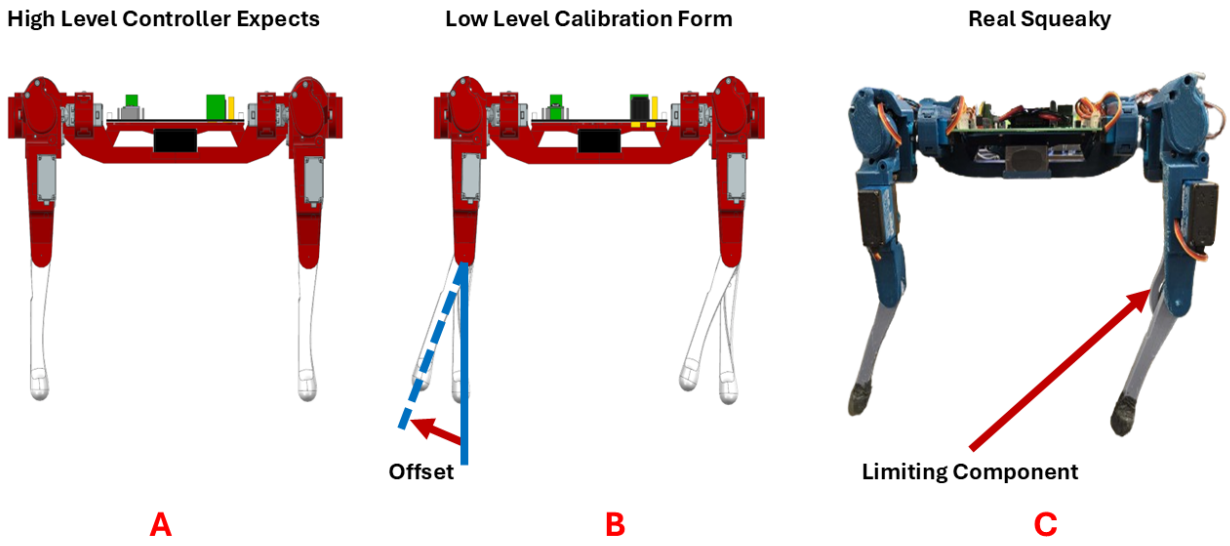

Figure 5.8: Image A shows the position that the high-level model expects, image B shows the offset needed to be made at LL, and Image C shows the limiting component.

offset will vary between each servo, but the theta for each joint should be set as close as possible to its minimum limit. Then, an offset is applied to adjust the distance the Squeaky leg can achieve to be fully straightened, as illustrated in Figure 5.8.

13. Use the "off" command and iterate through different offset values until it visually matches what the HL expects, as seen in Figure 5.8.

14. Record servo offset to the relevant offset array.

15. Repeat steps 2-14 until all servos have been calibrated and data recorded.

Another method to calibrate Squeaky is to position the robot so that each joint is orthogonal. An example of the orthogonal joint position for each part of the robot can be seen in Figures 5.9 and 5.10. This approach is more straightforward, relying on visual confirmation relative to the ground to obtain angle offsets. To facilitate this, the "off" command can be used,

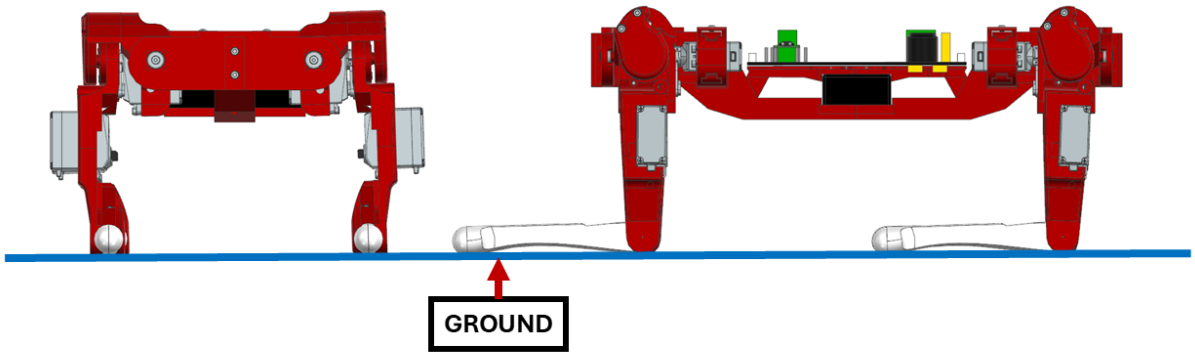

Figure 5.9: CAD version of Squeaky showing a front and side view with legs squared and at 90°.

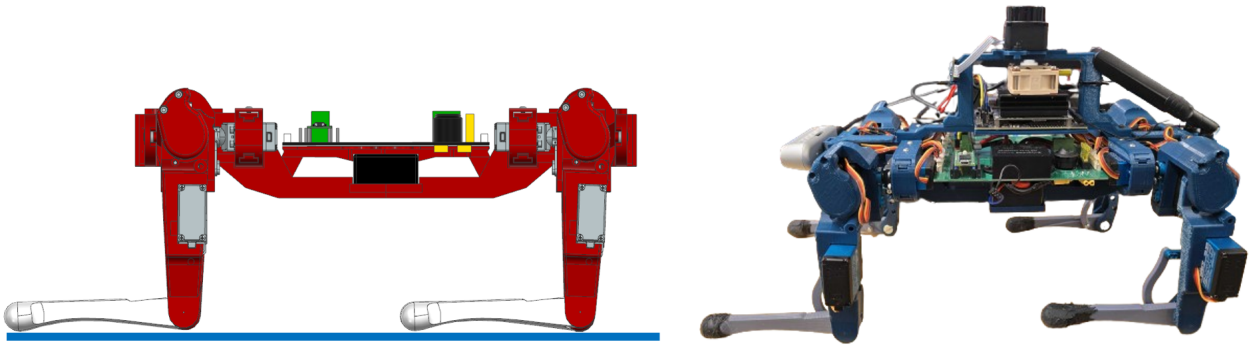

Figure 5.10: CAD and Real version of Squeaky with legs squared and at 90°.

which allows an offset to be applied to each servo individually. This makes making minor angle adjustments easier to square up the robot. As mentioned earlier, the offset variable is applied to the DesiredPWM_HL variable, which initially sets all servos to 90 degrees, except for the thigh servos, which are set to 45 degrees. Once the calibration is complete, the offset array will be updated with the obtained offset angles.

This section presents the calibration procedure for Squeaky, highlighting the critical importance of calibrating the physical robot with the expected configuration defined by the HL system. It began with the necessary setup steps and a detailed walkthrough of Method 1. Method 2 was introduced as an alternative approach, utilizing a different robot configuration, as illustrated in Figure 5.9 and Figure 5.10. Future iterations of this platform may incorporate motors with built-in feedback or more advanced features, which could significantly simplify the calibration process and improve overall ease of use.

5.4 Cost

One key point in the development of Squeaky was affordability. This section will break down the cost of the base platform and detail the cost to add perception. The cost of Squeaky

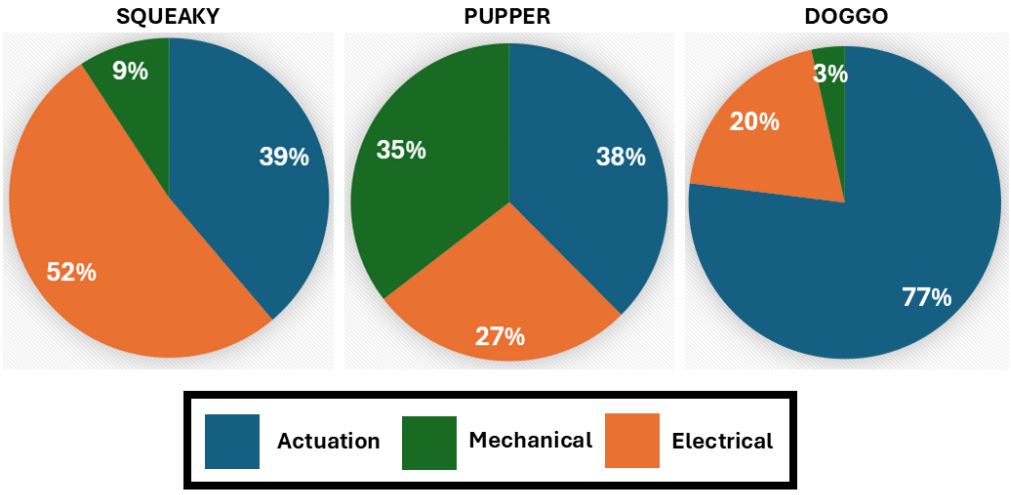

Figure 5.11: Pie Charts compare the distribution of cost between Squeaky, Pupper, and Doggo

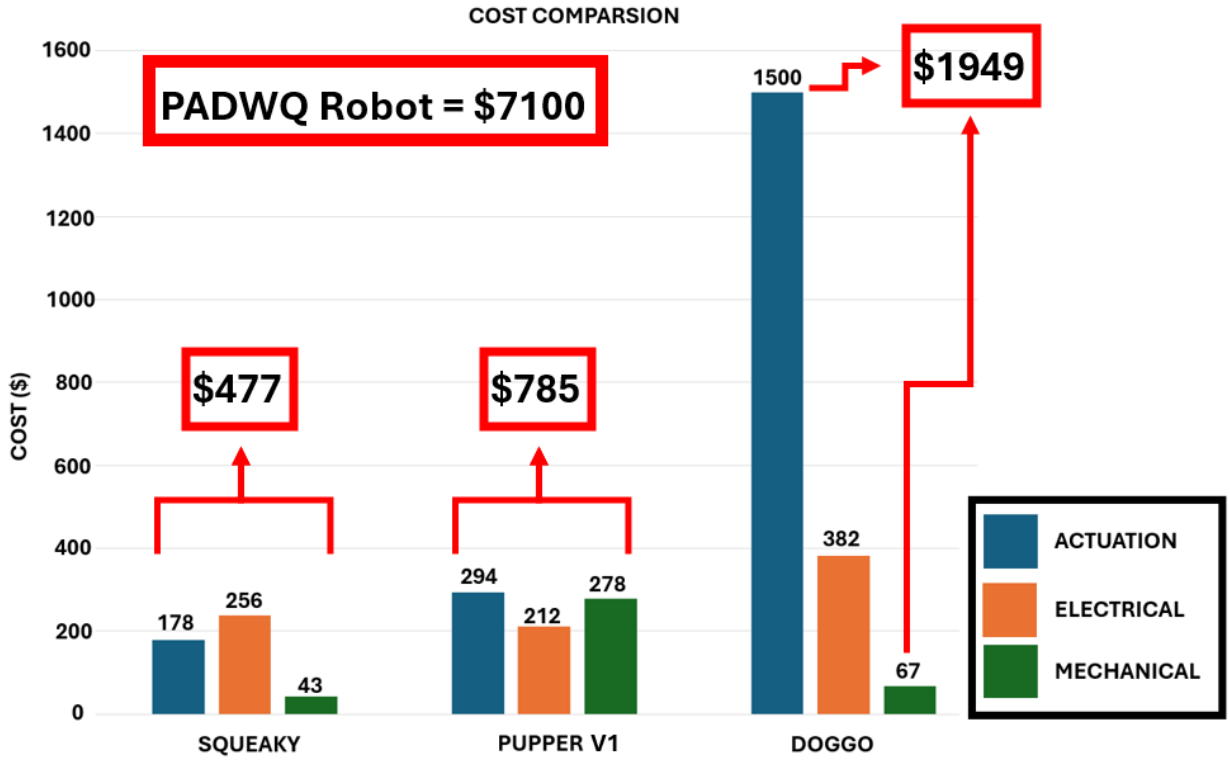

Figure 5.12: This bar graph compares the cost differences between Squeaky, Stanford Pupper V1, and Doggo and divides them into actuation, electrical, and mechanical. It also highlights PADWQ Overall Cost without perception sensors Cost.

will be compared with that of the Stanford Pupper V1 and Doggo, robots that inspired its development.

Keeping Squeaky affordable offers several distinct advantages. First, it makes the platform accessible to more research and educational groups. Second, it lowers the barrier to multi-robotic studies. Lastly, its configurability enables real-world experimentation with a variety of algorithms.

The cost allocation for Squeaky was divided into actuation, mechanical, and electrical, as shown in Figure 5.11. Figure 5.11 also compares the cost allocation between Squeaky, Stanford Pupper V1, and Doggo. Since Squeaky takes inspiration from Pupper V1, the actuation costs are closely aligned in percentage between these two robots. This is because they use

similar servos. However, the mechanical cost allocation is not proportional, as Squeaky uses an AM-centered approach for structural parts, while Stanford Pupper V1 employs subtractive manufacturing techniques.

When comparing the mechanical cost allocation for each robot, Squeaky and Doggo have similar costs because Doggo uses more AM parts for its chassis. When comparing Squeaky to the other robots, the electrical allocation accounts for more than half of the total cost. This is because Squeaky uses the NVIDIA Jetson Nano, Stanford Pupper V1 uses a Raspberry Pi 4, and Doggo uses a Raspberry Pi 5. The cost difference is that the Raspberry Pi 4 and 5 are priced between $40 and $80, whereas the NVIDIA Jetson Nano costs $140. Additionally, Squeaky incurs higher costs using a 5V 10A output buck converter than Pupper's 5V 3A output to provide for the NVIDIA Jetson Nano and other sensors to run experiments.

Figure 5.12 provides a clearer picture of the overall cost distribution for each section of the robots. The most obvious comparison between robots is the actuation costs, which comprise most of the expense. Starting with the most expensive actuation, the Stanford Doggo motors cost up to $1500. Next is the Stanford Pupper V1 at $294, followed by Squeaky at $178.

Description	Price Per Component (USD)	Quantity	Total Cost (USD)
35 kg Servo	14.87	12	178.38
PLA+ 1 spool	24.99	1	24.99
Bolts & Electronics (various)	35	N/A	35
LIPO Battery	28.99	1	28.99
Buck Converter	20	1	20
Jetson Nano	140	1	140
Jetson Fan	14.95	1	14.95
Teensy 4.0	23.42	1	23.42
Custom PCB	11.31	1	11.31
D435i (IMU ver.) Intel RealSense	334	1	334
2D Lidar LD19/STL-19P	85.99	1	85.99
		Total Cost Robot w/o Perception	**$ 477.04**
		Total Cost Robot with Perception	**$ 897.03**

BASE COST

PERCEPTION

Figure 5.13: Bill of Material of Squeaky with and without perception.

When making this comparison, it is crucial to consider the intended purpose of each robot. For example, Doggo's higher motor cost reflects its design for in-depth locomotion research, including running and jumping.

In contrast, Stanford Pupper V1 was created as an open-loop control robot for various testing applications, to prove that quadrupeds could be built at a lower cost. On the other hand, Squeaky was designed as an affordable experimental platform for testing real-world SLAM and navigational algorithms, focusing on researching multi-robotic applications. Closing the simulation-to-reality gap is crucial, as there are many variables to consider, such as hardware failures that are often not accounted for in simulations and other environmental factors.

Figure 5.13 shows a cost breakdown of Squeaky, including all components needed for one robot. As previously stated, most of the cost comes from the NVIDIA Jetson Nano and the servo motors for the base robot. When perception is factored in, the D435i Intel RealSense and the 2D LiDAR account for about half of Squeaky's total cost, amounting to $420 for the sensors.

This section compares Squeaky with relevant open-source quadrupeds to demonstrate that it is an affordable platform. A cost breakdown was provided, comparing three primary categories for each robot: actuation, mechanical, and electrical costs. A final bill of materials was also presented, showing the cost of the Squeaky base platform with and without perception. Future work may focus on further cost-cutting measures while still maximizing the platform's potential, ensuring it remains an excellent resource for research and education.

This chapter outlined the Squeaky robot's complete setup, assembly, calibration, and cost structure. It began by walking through the repository and software configuration to help users get started quickly and with minimal friction. The assembly section broke down the robot's physical build, including servo wiring, leg construction, and structural layout, with

visual guides for clarity. The calibration section addressed the importance of aligning the physical robot with its digital model in the HL system, providing two practical methods to ensure consistent, repeatable motion. Finally, the cost breakdown compares Squeaky with other open-source quadrupeds, demonstrating that it offers a highly capable and affordable robotics research and education platform. Moving forward, improvements such as smarter actuators and potential reductions in component costs could further enhance the platform's accessibility and functionality. This chapter is a foundation for building, operating, and scaling Squeaky as a single robot and as part of larger multi-robot systems.

Chapter 6

Results

This chapter presents the experimental results validating Squeaky's hardware and software performance. The chapter is divided into three primary sections: locomotion validation, SLAM performance using a single robot, and multi-robot SLAM capabilities. Each section is designed to showcase different aspects of the platform under real-world conditions.

The locomotion validation assesses Squeaky's gait and mobility across various terrains, identifying performance limitations caused by open-loop control and terrain-robot interactions. The second section focuses on implementing and evaluating two widely used SLAM algorithms: GMapping and SLAM Toolbox, demonstrating how the robot builds consistent maps in structured environments. Finally, the chapter explores the multi-robot SLAM capabilities made possible by Squeaky's modular framework and ROS-based architecture, including real-time map merging using unknown robot starting positions. Together, these results confirm that Squeaky is a capable, low-cost research platform for mobile robotics.

6.1 Locomotion Validation

This section will analyze the effectiveness of Squeaky's locomotion. It focuses on testing on four different terrains: turf, rough concrete flooring, slick concrete flooring, and a rubber mat. For each terrain, Squeaky's effectiveness is evaluated visually.

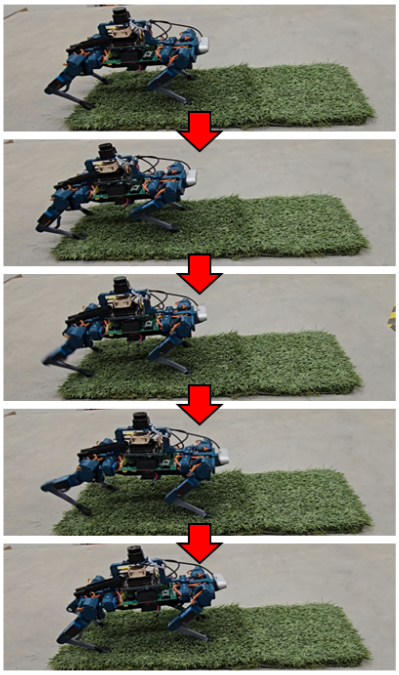

Figure 6.1: Snapshot of Squeaky transversing turf.

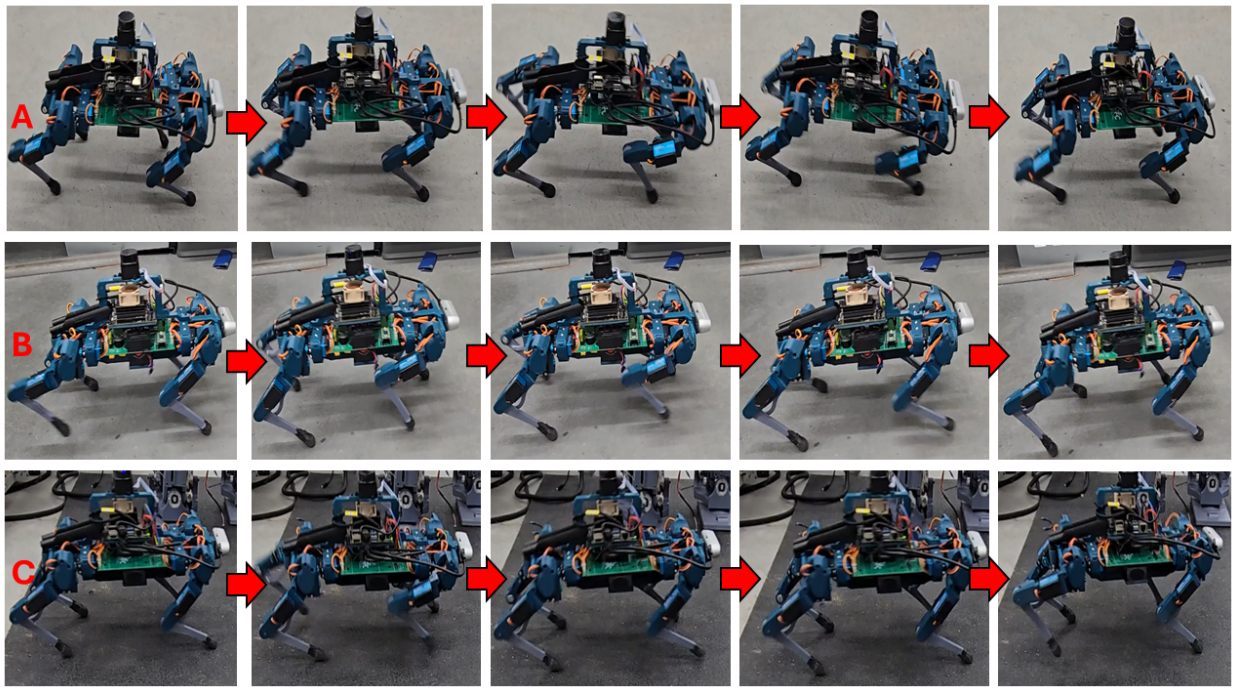

Figure 6.2: Snapshot of Squeaky transversing different terrains. Set A is rough concrete, Set B is smooth concrete, and Set C is the rubber mat.

During experiments on the turf, an increase in difficulty was witnessed for the robot as it attempted to proceed forward, as seen in Figure 6.1. Interestingly, the robot traverses the turf with little difficulty when moving in reverse. This seems to occur after further experiments and observations because the feet are caught within the turf. When the robot proceeds forward, it has a direct impact and the feet, especially the front feet, are hindered by the turf. When going in reverse, the shins are at an angle of 45°relative to the ground, allowing them to glide across the turf smoothly.

When tests were performed on rough surface concrete as seen in set A of Squeaking walking in Figure 6.2, in this case, the robot was able to walk on it with ease. The front right foot sometimes seems to drag, causing possible instability issues on the chassis. Similar results are seen in set B in Figure 6.2 where Squeaky was tested on the smooth concrete surface. The smooth and slick concrete surface allowed the robot to compensate for the issue of the foot dragging at various points.

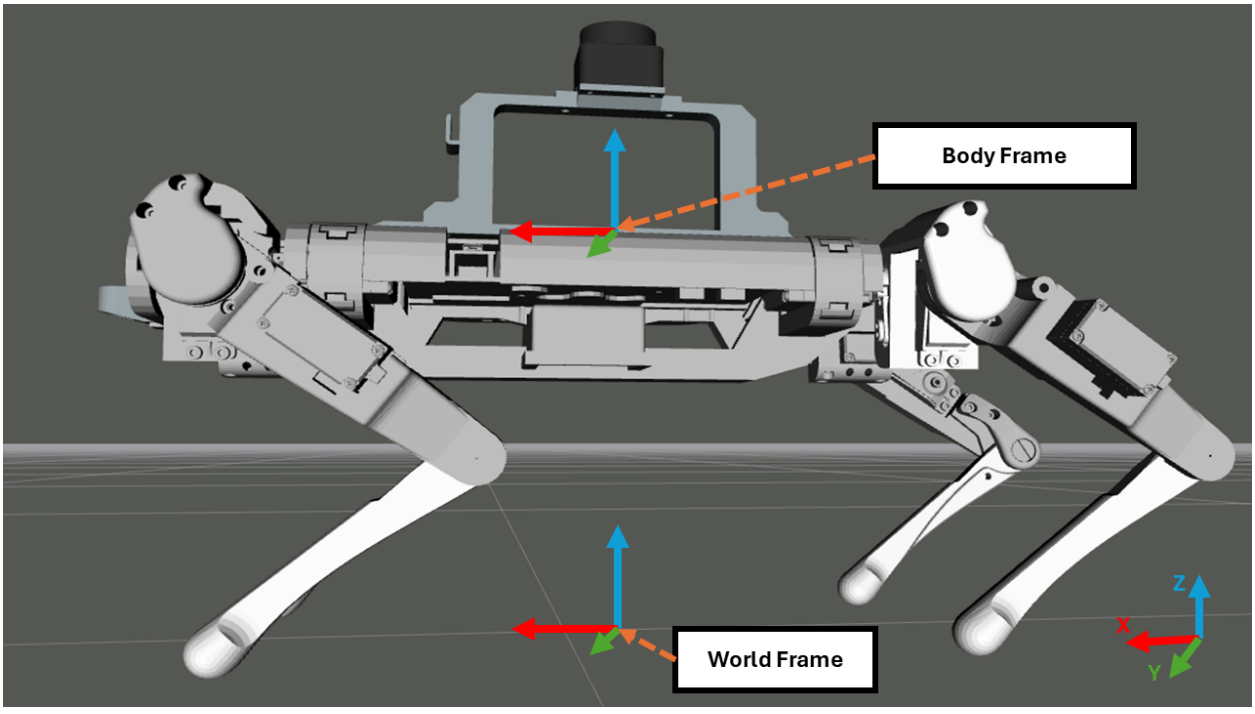

Figure 6.3: Displaying World Frame and Body Frame of the High-Level framework.

The robot on the rubber mat seen in set C in Figure 6.2 handles the material the best. Difficulties were observed when transversing from one terrain to another: concrete to the rubber map, especially because of the one-inch gap. This occurred primarily because of the robot's open-loop control, which currently does not have a way to account for terrain-level changes. Hence, the gait remains the same. After a few seconds, the robot clears the gap and proceeds onto the rubber map. Once all legs are on the same terrain, it proceeds along the map with little to no difficulty.

The test was performed for the rubber mat and both concrete variants by rotating around the body axis relative to the world frame. In these cases, it was to observe how well Squeaky could rotate on different terrains. To sum up these terrain tests, the robot struggles to rotate clockwise along the z-axis relative to the base of the robot. Figure 6.3 shows the body and world frame approximate location. It has been observed to happen as the robot shifts all its weight to the front right leg, causing that foot to remain in contact with the ground. Hence, the entire chassis struggles as the HL and LL do not have feedback and will continue the rotating gait. Interestingly, this is not seen when the robot is turning counterclockwise.

6.2 SLAM Overview On Squeaky

This robot's versatility is demonstrated by its ability to implement any SLAM package, only needing to adjust sensor configuration when necessary. If sensor configuration needs to be modified, this is a simple task, as these sensors can be replaced quickly and efficiently as long as driver nodes are written for the hardware to operate with ROS. The limiting factor is the SLAM package if it is an off-the-shelf version.

The specific package utilized on this platform is SLAM toolbox primarily because it is open-sourced and can work with multiple robotic systems. It also includes many other features

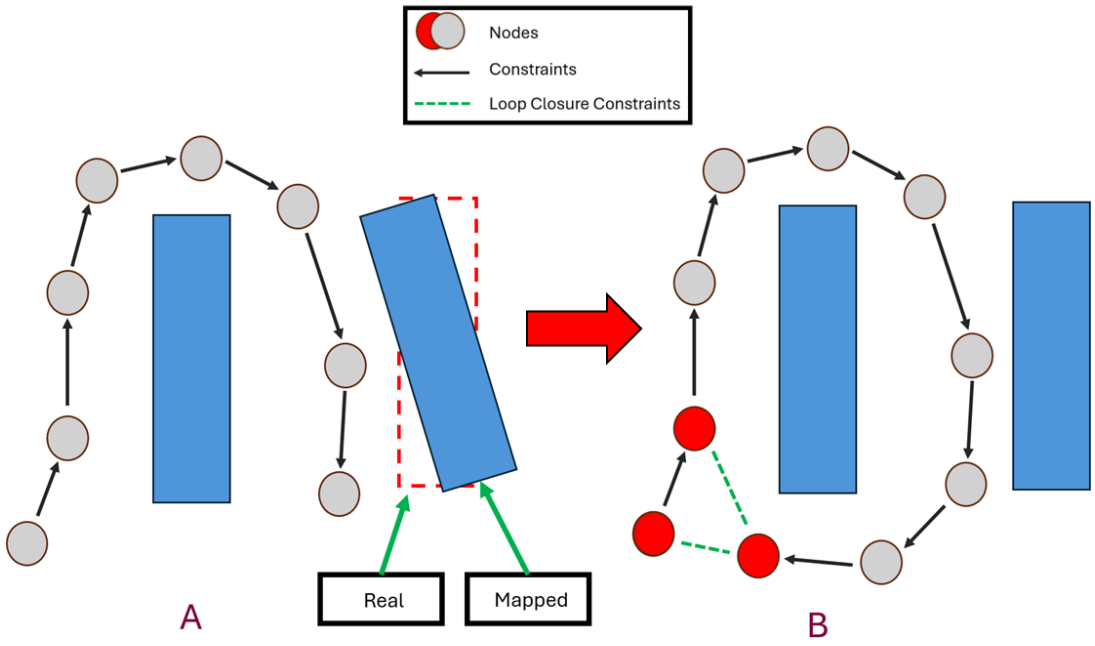

Figure 6.4: Image A is before loop closure, and Image B shows that when loop closure constraints are formed, the map is adjusted.

such as asynchronous and synchronous mapping modes. The ability to map an area, save it, load the saved pose-graph/map, and then continue to map on the previous version. This is just a glimpse of the features, and other potential features in this paper by Steve Macenski [5].

When comparing SLAM Toolbox to other types of SLAM, it is built around the graph-based SLAM approach, which addresses the SLAM problem by associating a robot's pose at each point in time with the measurements taken at that moment, linking them to a node in the graph. As the robot proceeds to map the area, a transformation between two nodes is identified, and either the robot's odometry is utilized or alignment between the two maps is. As the map gets larger and a robot may return to a prior position, these previous nodes can be compared with the new nodes, and a loop closure can occur. When this occurs, any map alignment issues that may have been happening can be adjusted to create a more reliable

map. Figure 6.4 shows an example of this in action, where the orange circle represents a pose at each point in time. The green arrows represent the transformation between each Pose. Image B shows a loop closure occurring when a relationship is formed between the nodes. What is most noticeable is the column realigning once this relationship is formed.

Another reason why SLAM toolbox was chosen was the need for the final map to be published on a namespace/map topic and with multiple Squeakys under the same ROS master, allowing a map merging node to be executed. This map merging node can work with n number of

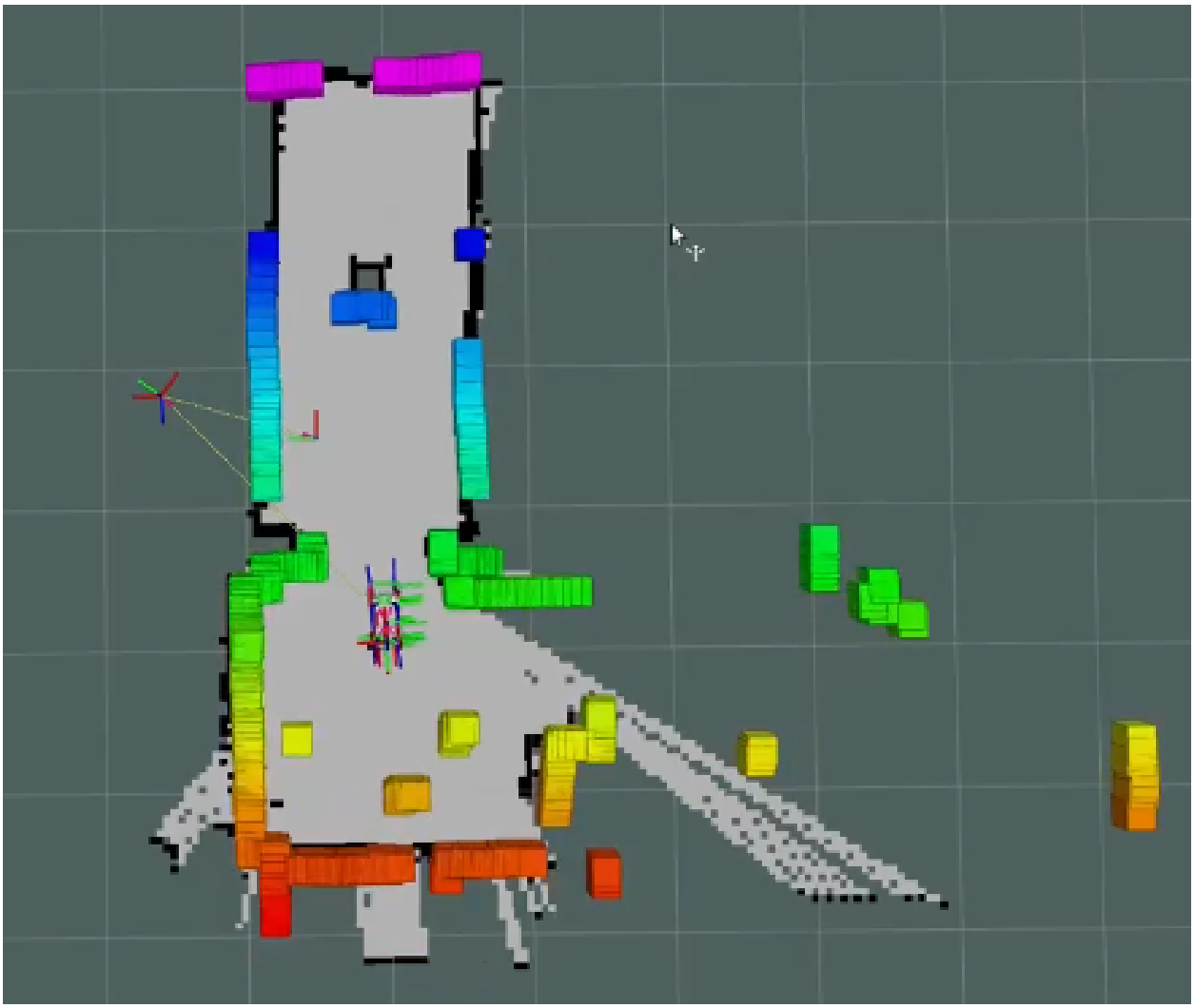

Figure 6.5: SLAM Map Generated with Gmapping.

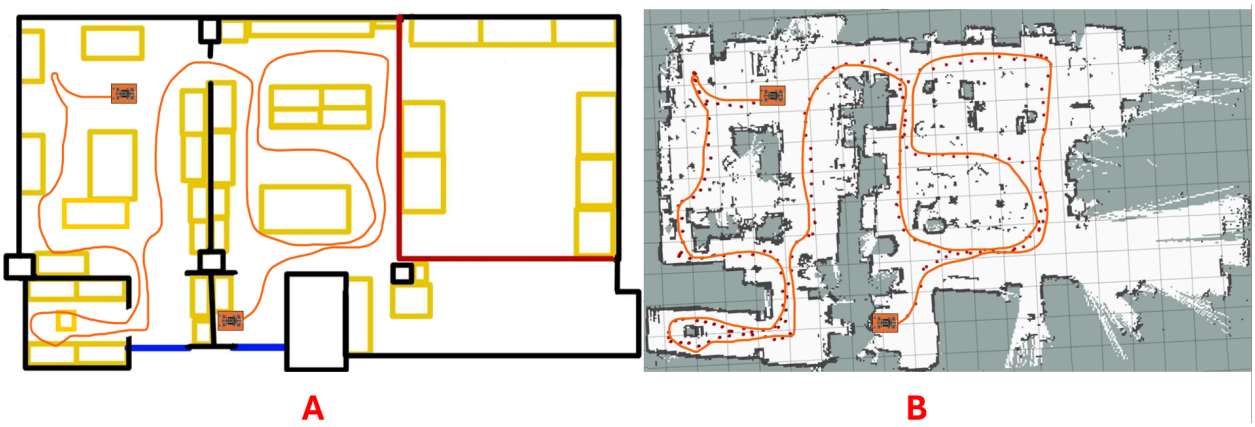

Figure 6.6: Image A shows a simplified building schematic of the lab space with yellow representing approximate furniture placement. The Orange line is Squeaky's path, which can be compared to the map Squeaky created in Image B.

robots /maps and outputs one merged map. This can be adjusted based on the mode used, as there are two: merging with known and unknown positions. As described in its name, the known position map merging generally works the best since the user can provide the initial position for each robot, allowing it to merge with little consequence. This is not always the case, and working with robots within a real environment, always knowing the exact position, can become rare on startup. In this case, these robots are in a real environment with an unknown position, hence the feature matching component of the algorithm is necessary for these maps to be merged. In this case, a large area must be mapped to match the two individual robot maps to be merged into one.

6.3 Single Robot SLAM

To demonstrate the capabilities of this platform, two different SLAM packages are implemented through ROS. First, GMapping was used, a basic algorithm that utilizes Rau-Blackwell particle filtering and scan matching to maintain robot localization. As seen in Figure 6.5, this implementation is shown in a straightforward rectangular office space with

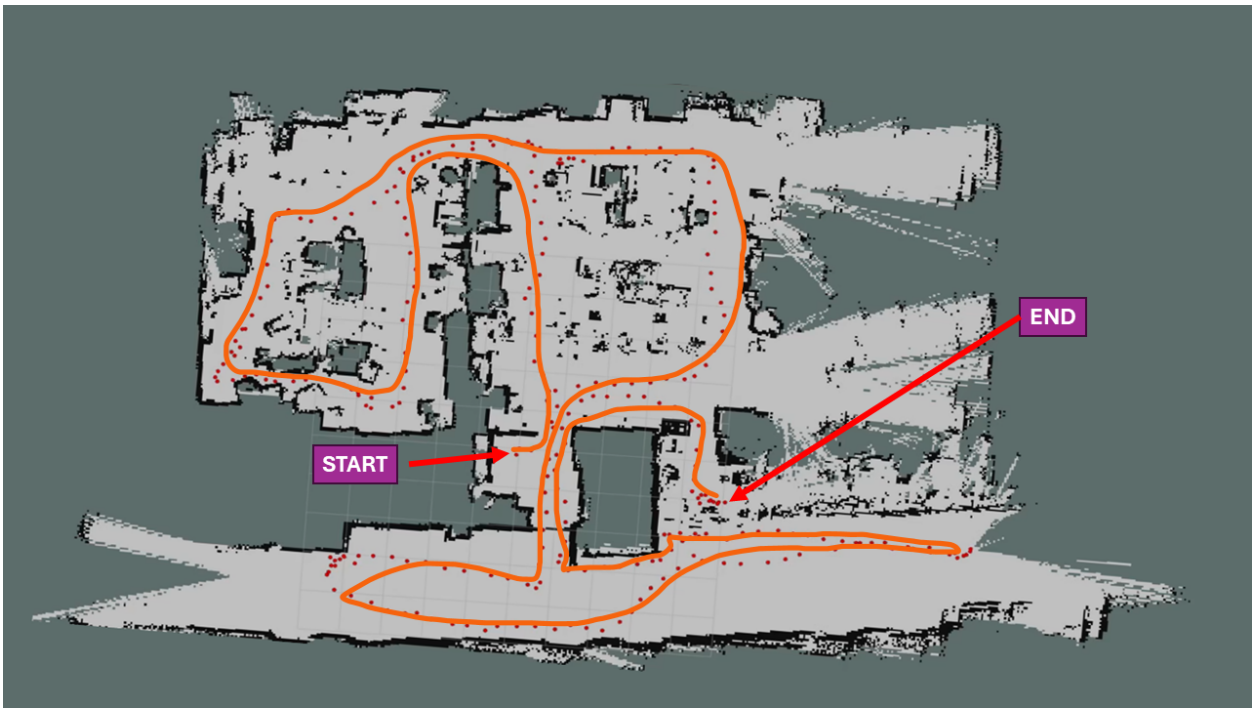

Figure 6.7: Single Squeaky mapping labspace including exterior hallway, 18 minutes to map this area.

an obstacle at the center. In Figure 6.5, the robot model based on the URDF is also implemented, showing the location and orientation of the robot. A key point regarding Gmapping is that it does not record the robot's path over time. In this particular figure, the robot mapping is being recorded. At the same time, the different points of the lidar are also displayed as a range of colors based on how reliable the current reading is. Gmapping does not record anything into the overall map to create a reliable map until a confidence level is reached.

SLAM toolbox is the second package used to show the modularity of the high-level framework of this robot. Instead of launching Gmapping, SLAM toolbox mapping algorithm is utilized, which opens many capabilities. Like Gmapping, SLAM Toolbox works only within 2D mapping but allows multi-robot implementation. Another key improvement is that the SLAM toolbox tracks the odometry over time. These markers are displayed as red dots recording the robot's position as it traverses the environment. This implementation is seen in Figure 6.6, which shows two images. Image A is a simplified layout of the lab space. It

also shows the pathing of Squeaky taken from the robot's internal map in Image B. Here, the red dots are partially visible under the Orange line drawn roughly over to give a clear understanding of the path taken. The path may seem erratic under initial observation, but Squeaky walked over some of the same paths to refine its internal map.

Another test was performed with SLAM toolbox and can be seen in Figure 6.7. The robot starts in the exact location and orientation as in Figure 6.6, but this time it maps for prolonged periods and expands the area it maps. To map this area, it took approximately 20 minutes and made a refined map, some issues that can be seen within the map are around where it ends and what can be seen to be the hallway, there is a glass wall which the lidar picks up partially, but also not entirely, causing a very erratic result at that location. What can also be observed is that there is one connection between the lab space and the hallway, causing the hallway to slightly bend.

6.4 Multi Robot SLAM

Since the robots start from an unknown initial position, two robots are placed in the same room and orientation. Each test was performed under these conditions to start the robots in the same space and collect many similar features in the beginning, as the package performing the merges is set to unknown positions and will merge these maps based on these similarities. This is seen with the path taken within the top of Figure 6.8. The red dot in each references the robot's previous location within the environment, and the path taken can be seen at the bottom of 6.8, and the merge map is the combination of both maps. The robot's path is removed for this combined map, as the node merger removes the pose graph. Once enough similarities are seen, the combined map is created and displayed in Figure 6.8.

The results presented in this chapter validate Squeaky's ability to function as a robust and

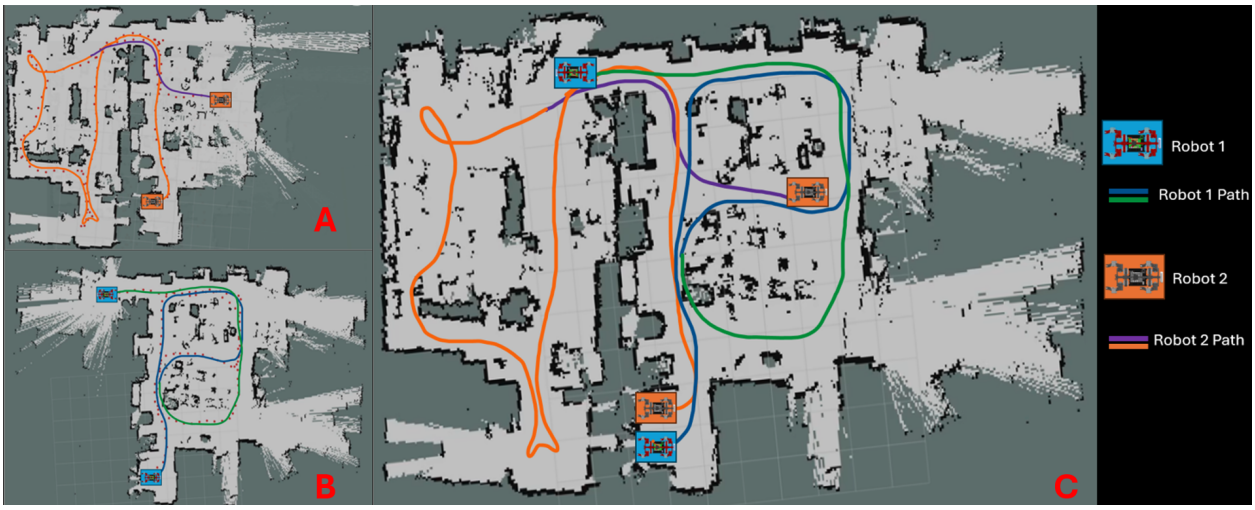

Figure 6.8: Multi Robot SLAM Map Generated Using SLAM Toolbox and Map Merging. Image A is the robot 1 map, Image B is the robot 2 map, and Image C is the merged map of 1 and 2.

adaptable research platform. Terrain testing showed that while the robot performs reliably on flat and moderately rough surfaces, its open-loop locomotion introduces challenges on soft or uneven ground. These insights offer a foundation for future work refining gait control and incorporating closed-loop feedback mechanisms.

Using GMapping and SLAM Toolbox on the SLAM side, both single-robot and multi-robot implementations were demonstrated. The experiments highlighted the platform's modularity and compatibility with standard ROS tools. SLAM Toolbox, in particular, enabled advanced features such as map saving, pose-graph refinement, and multi-robot map merging using unknown starting positions.

Overall, these experiments reinforce Squeaky's value as a research tool in mobile robotics, enabling experimentation in real-world scenarios involving localization, mapping, and coordination. Future work may enhance performance further through improved terrain adaptation, feedback-based locomotion, and tighter integration between SLAM and navigation.

Chapter 7

Conclusion

This final chapter brings together the full scope of this work, summarizing the motivations, contributions, and demonstrated results of the Squeaky platform. While numerous open-source quadrupeds exist, each with distinct design goals—from reinforcement learning to dynamic control, Squeaky targets a specific niche: affordable multi-robotic experimentation in real-world environments. Its modularity, combined with robust on-board computation via the NVIDIA Jetson Nano and support for various perception sensors, allows researchers to deploy more complex algorithms without sacrificing accessibility or scalability. We will end with a look at the future and avenues of future research for the Squeaky platform.

7.1 Conclusion

Many works have developed and designed their own open-sourced quadrupeds. Each quadruped's goal has been vastly different, as seen in their design. Some for machine learning, some for control testing in a real-world environment, and many more. Squeaky fills the void within the robotic community that focuses on more multi-robotic systems and swarm applications within real environments. The overall cost of the platform is affordable compared to other robots for what it provides, such as a NVIDIA Jetson Nano that allows more intensive algorithm testing, a variety of perception sensors, and the option to reconfigure the robot depending on the project's changing needs.

Throughout this work, I presented an overall design and implementation of SLAM into a biologically inspired and affordable quadruped. This paper reviewed the entire robot, ranging from the mechanical, electrical, and software design to assembly and testing with maps resulting from applying SLAM to the Squeaky platform. The results demonstrated the effectiveness of this system for multirobotic applications within real-world environments at an affordable cost.

7.2 Limitations

Squeaky is a capable robotic platform; however, its low overall cost does come with certain limitations. Notably, the servos used lack feedback sensors, restricting the system to open-loop control. This design choice, along with the use of low-cost servos, results in reduced locomotion performance, particularly when the robot carries heavier payloads. Despite these constraints, Squeaky remains an effective platform for research and educational applications, offering accessibility and functionality at a fraction of the cost of more advanced systems.

7.3 Future Work

Squeaky's current configuration, which includes a navigational design framework, ROS integration, perception sensors, a custom PCB hub, and an NVIDIA Jetson Nano, makes it a versatile platform with numerous avenues for future exploration. Potential applications range from machine learning for autonomous decision making to large-scale multi-robot systems operating in real-world environments. Modularity has been a core design principle, allowing users to adapt and upgrade components easily.

Several future directions are worth pursuing. One is the integration of vision-based SLAM

algorithms that fully utilize the capabilities of the D435i sensor. Another possibility is refining the multi-robot SLAM framework so that the merged map replaces the original, enabling persistent, shared environmental understanding. Additionally, since Squeaky is already embedded within the ROS ecosystem, migrating to ROS 2 is a natural next step. The existing infrastructure supports this transition, including the URDF and related robot files. Squeaky provides a robust and affordable baseline for testing SLAM and other robotic algorithms in real-world conditions.

Bibliography

[1] G. Grisetti, C. Stachniss, and W. Burgard, "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Apr. 2005, pp. 2432–2437, iSSN: 1050-4729. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1570477

[2] R. Vincent, B. Limketkai, and M. Eriksen, "Comparison of indoor robot localization techniques in the absence of GPS," *Proceedings of SPIE - The International Society for Optical Engineering*, Apr. 2010.

[3] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271–1278.

[4] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.

[5] S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, May 2021. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.02783

[6] Y. Chang, Y. Tian, J. P. How, and L. Carlone, "Kimera-Multi: a System for Distributed Multi-Robot Metric-Semantic Simultaneous Localization and Mapping," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*,

May 2021, pp. 11 210–11 218, iSSN: 2577-087X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9561090

[7] H. Lategahn, A. Geiger, and B. Kitt, "Visual SLAM for autonomous ground vehicles," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1732–1737, iSSN: 1050-4729. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5979711?casa_token=jvxj3UxQoHAAAAAA:RO4XP3GKazPF9C1a_m163c-zYjpYZJmY4_wHVAlNH9v21NcXP-PT0cTQ_MMwbu5SLUErqcXDbQ

[8] S. Yang, S. A. Scherer, X. Yi, and A. Zell, "Multi-camera visual SLAM for autonomous navigation of micro aerial vehicles," *Robotics and Autonomous Systems*, vol. 93, pp. 116–134, Jul. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889015302177

[9] S. Zheng, J. Wang, C. Rizos, W. Ding, and A. El-Mowafy, "Simultaneous Localization and Mapping (SLAM) for Autonomous Driving: Concept and Analysis," *Remote Sensing*, vol. 15, no. 4, p. 1156, Jan. 2023, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2072-4292/15/4/1156

[10] S. Halder, K. Afsari, J. Serdakowski, S. DeVito, and R. King, "Accuracy Estimation for Autonomous Navigation of a Quadruped Robot in Construction Progress Monitoring," pp. 1092–1100, May 2022, publisher: American Society of Civil Engineers. [Online]. Available: https://ascelibrary.org/doi/10.1061/9780784483893.134

[11] S. Halder, K. Afsari, J. Serdakowski, S. DeVito, M. Ensafi, and W. Thabet, "Real-Time and Remote Construction Progress Monitoring with a Quadruped Robot Using Augmented Reality," *Buildings*, vol. 12, no. 11, p. 2027, Nov. 2022, number:

11 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2075-5309/12/11/2027

[12] S. Halder and K. Afsari, "Robots in Inspection and Monitoring of Buildings and Infrastructure: A Systematic Review," *Applied Sciences*, vol. 13, no. 4, p. 2304, Jan. 2023, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/13/4/2304

[13] S. Halder, K. Afsari, E. Chiou, R. Patrick, and K. A. Hamed, "Construction inspection & monitoring with quadruped robots in future human-robot teaming: A preliminary study," *Journal of Building Engineering*, vol. 65, p. 105814, Apr. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352710222018204

[14] F. Shu, P. Lesur, Y. Xie, A. Pagani, and D. Stricker, "SLAM in the Field: An Evaluation of Monocular Mapping and Localization on Challenging Dynamic Agricultural Environment," 2021, pp. 1761–1771. [Online]. Available: https://openaccess.thecvf.com/content/WACV2021/html/Shu_SLAM_in_the_Field_An_Evaluation_of_Monocular_Mapping_and_WACV_2021_paper.html

[15] H. Ding, B. Zhang, J. Zhou, Y. Yan, G. Tian, and B. Gu, "Recent developments and applications of simultaneous localization and mapping in agriculture," *Journal of Field Robotics*, vol. 39, no. 6, pp. 956–983, 2022, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.22077. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.22077

[16] R. Islam, H. Habibullah, and T. Hossain, "AGRI-SLAM: a real-time stereo visual SLAM for agricultural environment," *Autonomous Robots*, vol. 47, no. 6, pp. 649–668, Aug. 2023. [Online]. Available: https://doi.org/10.1007/s10514-023-10110-y

[17] P. Beinschob and C. Reinke, "Graph SLAM based mapping for AGV localization in large-scale warehouses," in *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, Sep. 2015, pp. 245–248. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7312637?casa_token=B3wUJrI1UsUAAAAA:fp4MW7dEiecxmmnKmYLzHE26DjDMqsi2kVRyCdDz3TWeHEMxOfYiJijsz0R0iKTYVIEDJiV04w

[18] Y. Chen, Y. Wu, and H. Xing, "A complete solution for AGV SLAM integrated with navigation in modern warehouse environment," in *2017 Chinese Automation Congress (CAC)*, Oct. 2017, pp. 6418–6423. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8243934?casa_token=cMUo7QMLv3kAAAAA:A8oBUaLkQ-Zh52ZO53W6hFo76YqOTI5xc09qACJMsuA5UNygYkOP2KPkNuARWcoYXbHhsJ6_rg

[19] A. Moura, J. Antunes, A. Dias, A. Martins, and J. Almeida, "Graph-SLAM Approach for Indoor UAV Localization in Warehouse Logistics Applications," in *2021 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, Apr. 2021, pp. 4–11. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9429791?casa_token=lRDWvLpTzzwAAAAA:XcFJ44SbbtuABDGCHI0PkQX3GmqGEfQols0scqhv61Gfwhtd0h9Fg2Ctayp58o_EkpBEk5pWQA

[20] R. Keith and H. M. La, "Review of Autonomous Mobile Robots for the Warehouse Environment," Jun. 2024, arXiv:2406.08333 [cs]. [Online]. Available: http://arxiv.org/abs/2406.08333

[21] N. Kau and S. Bowers, *Stanford Pupper: A Low-Cost Agile Quadruped Robot for Benchmarking and Education*, Oct. 2021.

[22] J. Kim, T. Kang, D. Song, and S.-J. Yi, "Design and Control of a Open-Source, Low Cost, 3D Printed Dynamic Quadruped Robot," *Applied Sciences*, vol. 11, no. 9, p. 3762, Jan. 2021, number: 9 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/11/9/3762

[23] J. R. Rebula, P. D. Neuhaus, B. V. Bonnlander, M. J. Johnson, and J. E. Pratt, "A Controller for the LittleDog Quadruped Walking on Rough Terrain," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, Apr. 2007, pp. 1467–1473, iSSN: 1050-4729. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4209295?casa_token=ZmylCuCg6l4AAAAA:3k38p930oIk_7rvMCQJEViRliTdgkMqSQ3LT1YqBk4sbYPNJOz4ipfUWwiTHwCCij5P6O2h4Yw

[24] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter, "BigDog, the Rough-Terrain Quadruped Robot," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 10 822–10 825, Jan. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016407020

[25] C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, "Design of HyQ – a hydraulically and electrically actuated quadruped robot," *Proceedings of the Institution of Mechanical Engineers, Part I*, vol. 225, no. 6, pp. 831–849, Sep. 2011, publisher: IMECHE. [Online]. Available: https://doi.org/10.1177/0959651811402275

[26] C. Semini, V. Barasuol, J. Goldsmith, M. Frigerio, M. Focchi, Y. Gao, and D. G. Caldwell, "Design of the Hydraulically Actuated, Torque-Controlled Quadruped Robot HyQ2Max," *IEEE/ASME Transactions on Mechatronics*, vol. 22, no. 2, pp. 635–646, Apr. 2017, conference Name: IEEE/ASME Transactions on Mechatronics. [Online]. Available: https://ieeexplore.ieee.

org/abstract/document/7587429?casa_token=tB0ivGGZl7AAAAAA:BpFr5vbw_Rj2-x95gUqlV-dLn9w8zGrSMuvd23GlpcIHal18OnXhJI1Ij4KfUg5hG2xorpgUNg

[27] M. Hutter, C. Gehring, D. Jud, A. Lauber, C. D. Bellicoso, V. Tsounis, J. Hwangbo, K. Bodie, P. Fankhauser, M. Bloesch, R. Diethelm, S. Bachmann, A. Melzer, and M. Hoepflinger, "ANYmal - a highly mobile and dynamic quadrupedal robot," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 38–44, iSSN: 2153-0866. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7758092?casa_token=b1k2Euzb67cAAAAA:PA67g85a-kuYbaQ2YRl1DYImG1Lg3_2kwREIjYMBz4HkzWvIte03Ua12dL8uzaYLkAaxskOGOw

[28] D. J. Hyun, S. Seok, J. Lee, and S. Kim, "High speed trot-running: Implementation of a hierarchical controller using proprioceptive impedance control on the MIT Cheetah," *The International Journal of Robotics Research*, vol. 33, no. 11, pp. 1417–1445, Sep. 2014, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/0278364914532150

[29] H.-W. Park, P. M. Wensing, and S. Kim, "Jumping over obstacles with MIT Cheetah 2," *Robotics and Autonomous Systems*, vol. 136, p. 103703, Feb. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889020305431

[30] N. Kau, A. Schultz, N. Ferrante, and P. Slade, "Stanford Doggo: An Open-Source, Quasi-Direct-Drive Quadruped," in *2019 International Conference on Robotics and Automation (ICRA)*, May 2019, pp. 6309–6315, iSSN: 2577-087X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8794436

[31] R. Boney, J. Sainio, M. Kaivola, A. Solin, and J. Kannala, "RealAnt: An Open-Source Low-Cost Quadruped for Education and Research in Real-World

Reinforcement Learning," Jun. 2022, arXiv:2011.03085 [cs]. [Online]. Available: http://arxiv.org/abs/2011.03085

[32] F. García-Cárdenas, N. Soberón, O. E. Ramos, and R. Canahuire, "Charlotte: Low-cost Open-source Semi-Autonomous Quadruped Robot," in *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, Apr. 2020, pp. 281–286. [Online]. Available: https://ieeexplore.ieee.org/document/9096210

[33] A. Mourikis and S. Roumeliotis, "Analysis of positioning uncertainty in simultaneous localization and mapping (SLAM)," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 1, Sep. 2004, pp. 13–20 vol.1. [Online]. Available: https://ieeexplore.ieee.org/document/1389322

[34] S. Huang and G. Dissanayake, "Convergence and Consistency Analysis for Extended Kalman Filter Based SLAM," *IEEE Transactions on Robotics*, vol. 23, no. 5, pp. 1036–1049, Oct. 2007, conference Name: IEEE Transactions on Robotics. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4339532?casa_token=Z9aVPz8B388AAAAA:QKev1p2D7Ka9pW1NUYfbXzg1qEZPmDp1w2RCbArLZ4brHBoAwwrGOn0M5eu8O-DVW0kZff4K

[35] G. Grisetti, C. Stachniss, and W. Burgard, "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, Feb. 2007, conference Name: IEEE Transactions on Robotics. [Online]. Available: https://ieeexplore.ieee.org/document/4084563

[36] R. Dubé, A. Gawel, H. Sommer, J. Nieto, R. Siegwart, and C. Cadena, "An online multi-robot SLAM system for 3D LiDARs," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 1004–1011, iSSN: 2153-0866. [Online]. Available: https://

ieeexplore.ieee.org/abstract/document/8202268?casa_token=apwJeyP_yFwAAAAA:7zsR5Cn4UdcYBiSn1DKR0328d4B8RJqupXVzLl8g082CmRLB3GYpe-gVur8qlryqjclLbyYWEg

[37] C. Ju, J. Kim, J. Seol, and H. I. Son, "A review on multirobot systems in agriculture," *Computers and Electronics in Agriculture*, vol. 202, p. 107336, Nov. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169922006457

[38] S. Thrun and Y. Liu, "Multi-robot SLAM with Sparse Extended Information Filers," in *Robotics Research. The Eleventh International Symposium*, P. Dario and R. Chatila, Eds. Berlin, Heidelberg: Springer, 2005, pp. 254–266.

[39] X. S. Zhou and S. I. Roumeliotis, "Multi-robot SLAM with Unknown Initial Correspondence: The Robot Rendezvous Case," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2006, pp. 1785–1792, iSSN: 2153-0866. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4058636?casa_token=wJ806GHKM1MAAAAA:U7CPpjnq5QNHxHnp9cxNa3kjwgXAGVjfyKfOyp6VbshymlPblwH8NTXULGjtvIT3zvtrNo2xxg

[40] M. T. Lázaro, L. M. Paz, P. Piniés, J. A. Castellanos, and G. Grisetti, "Multi-robot SLAM using condensed measurements," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov. 2013, pp. 1069–1076, iSSN: 2153-0866. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6696483?casa_token=O8XmgsWmyHQAAAAA:7YuS47-sojSL9Gc94ewHQadbJW1jnrDVFdFBjMBzKjdQaDIqawkCZVWGxnaeUlimXlj9xa_pgw

[41] S. E. Schoedel, A. J. Fuge, B. Kalita, and A. Leonessa, "Development of an Affordable and Modular 3D Printed Quadruped Robot." American

Society of Mechanical Engineers Digital Collection, Feb. 2023. [Online]. Available: https://dx.doi.org/10.1115/IMECE2022-95700

Appendices

Appendix A

First Appendix

A.1 Teensy Code for 1 Robot

```
1
2 #include <i2c_driver_wire.h>
3 #include <Servo.h>
4
5 #define USING_JET
6 #define MAX_MESSAGE_BUFFER 64
7 #define MAX_CMD_BUFFER 64
8 //#define USING_IMU
9
10 #ifdef USING_IMU
11 #include "MPU9250.h"    // https://github.com/bolderflight/MPU9250
12 #include "Streaming.h"    // needed for the Serial output https://github.com/
       geneReeves/ArduinoStreaming
13 #include "SensorFusion.h"
14
15 SF fusion;
16
17 MPU9250 IMU(Wire1, 0x68);
18 int status;
19 #endif
20
```

```
21 #define NUM_SERVOS 12
22 // Servo Pins For SQUEAKY Shield V1.1
23 // pin grouping (Groups of 3 pins) -> (leg1 leg2 leg3 leg4)
24 const int servoPins[12] = {4, 3, 2,     5, 6, 7,     14, 9, 8,     15, 20,
       21};
25
26 // Create servo objects
27 Servo flhr; // front left hip roll = shoulder
28 Servo flhp; // front left hip pitch = leg
29 Servo fls;  // front left shin
30
31 Servo frhr;
32 Servo frhp;
33 Servo frs;
34
35 Servo blhr;
36 Servo blhp;
37 Servo bls;
38
39 Servo brhr;
40 Servo brhp;
41 Servo brs;
42
43 // array of servos:
44 // Servo Mapping, Utilized for SQUEAKY Shield V1.1
45 Servo servos[12] = {flhr, flhp, fls, frhr, frhp, frs, blhr, blhp, bls, brhr,
       brhp, brs};
46
47 // Global Variables
48
49 // Servo position array, will be updated when new poisition recieved from HL
```

```
50 int DesiredPWM_HL[12] = {90, 45, 90,     90, 45, 90,     90, 45, 90,    90,
       45, 90};
51
52 // Set global intialization for these two. in future may optimize and remove
       these arrays to simplify
53 int servoPWM_offset_LL[12] = {0, 0, 0,      0, 0, 0,         0, 0, 0,
       0, 0, 0};
54 int servoPWM_LL[12] = {0, 0, 0,      0, 0, 0,         0, 0, 0,        0, 0,
       0};
55
56 // Output shaft motor angles in degrees
57 float motorRotations[NUM_SERVOS] = {90, 90, 90,     90, 90, 90,     90, 90,
       90,     90, 90, 90};
58
59 // Servo SQUEAKY Shield V1.1 SQUEAKY
60 // Min and Max PWMS,
61 // Invert Servo
62 int minPWMs[NUM_SERVOS] = {0, 0 , 0,       0, 0, 15,       0, 0, 5,         0,
        0, 10};                                          // For v3
63 int maxPWMs[NUM_SERVOS] = {180, 140, 160,  120, 180, 150,   150, 130, 155,
       180, 130, 160};                       // For v3
64 bool invertServo[NUM_SERVOS] = {false, false, false,     false, true, true,
          true, false, false,     true, true, true}; // For v3 Zach's Army
       Green Squeaky
65
66 // Offsets to get theta = 0 for  each servo
67 // Note that offsets is Squeaky will be on it knees, and all joints will be 90
        degrees (visually) then
68 // use theta command for Serial to Calibrate
69 int offsets[NUM_SERVOS] = { 5, 15, -10,   -5, 5, -15,     0, 15, -20,    0,
       15, -10}; // For shield
```

```
70
71 // This will put motors into position when true
72 bool NewData = true;
73
74 // For serial communication
75 char message[MAX_MESSAGE_BUFFER] = {};
76 int messageIndex = 0;
77 String cmd[MAX_CMD_BUFFER] = {};
78 int cmdIndex = 0;
79
80 void setup() {
81
82   Serial.begin(115200);
83
84   Serial.println("Serial begin");
85
86   // pinMode(13, OUTPUT);
87
88  #ifdef USING_JET
89
90   Serial.println("before starting wire on 0x8");
91   // Join I2C bus as slave with address 8
92   Wire.begin(0x8);
93
94   // Call receiveEvent when data is received over I2C
95   Wire.onReceive(receiveEvent);
96
97   Serial.println("using raspberry pi");
98
99 #endif
100
```

```
101   // Assign all servo objects to appropriate pins
102   for (int i = 0; i < NUM_SERVOS; i++)
103   {
104     servos[i].attach(servoPins[i]);
105   }
106
107 }
108
109 void loop() {
110   // Currently Empty, Testing I2c Needed to make sure valid results
111
112   //  When new data is recieved, activate leg motion
113   if (NewData){
114     LegMotion();
115     NewData = false;
116   }
117
118     /*
119    * Used only for commands sent directly through serial from PC (not
          raspberry pi)
120    */
121   // Read incoming serial data
122   if (Serial.available() > 0)
123   {
124     readInput();
125     parseCmd();
126     Serial.println("Msg Sent");
127
128   }
129
130 }
```

```
131
132 /*
133  *  Called when data is received from master over I2C
134  */
135 void receiveEvent(int numBytes) {
136   // Feed controller input watchdog
137   // receiverWatchDog = millis();
138
139   // Set LED high while reading
140   // digitalWrite(ledPin, HIGH);
141
142   Serial.print("num bytes: ");Serial.println(numBytes);
143
144   String i2c_msg = "";
145
146   // IN high level offset is two, I wonder if that is the two bad bytes that
        sam was seeing? Zach
147   // For now, uncommenting SAM code to remove the two trash bytes
148   char throw_byte = Wire.read();
149   throw_byte = Wire.read();
150
151   int signal_theta[12];
152   int theta_index = 0;
153   NewData = true;
154
155   // Read inputs as shorts with MSB first
156   while (Wire.available()) {
157     // High Level Sending 1 byte of data, total of 12 data Bytes,
158
159     // Store Each Byte of data in each index
160     signal_theta[theta_index++] = Wire.read();
```

```
161     Serial.print("theta: "); Serial.println(signal_theta[theta_index-1]);
162   }
163
164   // Copy data from I2c to global variable
165   for (int i = 0; i < NUM_SERVOS; i++)
166   {
167     DesiredPWM_HL[i] = signal_theta[i];
168   }
169 }
170
171 // Function to take in theta and output Servo Motion
172 void LegMotion()    // Add an offset that is from calibrations that matches
      what high level expects for low level.
173 {
174
175   // will have recieve events set a booleon when new data has been sent (for
        now this will be in void loop, in future check if teensy has hardware
        interrupts, but arduino based interupt are not the best)
176
177   for (int i = 0; i < NUM_SERVOS; i++)
178   {
179     // Calculate Motor Rotations variable with offsets
180     motorRotations[i] = offsets[i] + DesiredPWM_HL[i];
181
182     // Invert pwms where necessary
183     if (invertServo[i] == true)
184     {
185       servoPWM_offset_LL[i] = 180 - motorRotations[i];
186     }
187     else
188     {
```

```
189         servoPWM_offset_LL[i] = motorRotations[i];
190       }
191
192       // Constrain pwms to given bounds
193       servoPWM_LL[i] = withinBounds(servoPWM_offset_LL[i], minPWMs[i], maxPWMs[i
              ], i);
194     }
195
196     // Apply pwms to each servo
197     for (int i = 0; i < NUM_SERVOS; i++)
198     {
199       servos[i].write(servoPWM_LL[i]);
200     }
201
202 }
203
204 int withinBounds(int pwmRaw, int pwmMin, int pwmMax, int motorIndex)
205 {
206     int pwmContained = pwmRaw;
207     if (pwmRaw < pwmMin)
208     {
209       pwmContained = pwmMin;
210 //      pwmRaw_exceeded_bounds[motorIndex] = true;
211     }
212     else if (pwmRaw > pwmMax)
213     {
214       pwmContained = pwmMax;
215 //      pwmRaw_exceeded_bounds[motorIndex] = true;
216     }
217
218     return pwmContained;
```

```
219 }
220
221 /*
222    readInput and parseIntoCmd are for making cmd string
223 */
224 void readInput()
225 {
226   // Fill message array with incoming chars
227   while (Serial.available() > 0)
228   {
229     message[messageIndex] = Serial.read();
230     messageIndex++;
231   }
232   parseIntoCmd();
233   messageIndex = 0;
234   //   if (debugging)
235   //   {
236   //     Serial.println(message);
237   //   }
238 }
239
240 void parseIntoCmd()
241 {
242   cmdIndex = 0;
243   int lasti = 0; // Saves most recent place after last string split
244   String messageStr(message); // Char array into string so can use substring
245   for (int i = 0; i < messageIndex; i++)
246   {
247     if (message[i] == ',') // Delineates command word
248     {
249       cmd[cmdIndex] = messageStr.substring(lasti, i);
```

```
250         cmdIndex++;
251         lasti = i + 1;
252       }
253       else if (i == messageIndex - 1) // Last command (no ending comma)
254       {
255         if (message[messageIndex] != ',') // Ignore any stray ending commas
256         {
257           cmd[cmdIndex] = messageStr.substring(lasti, i + 1);
258         }
259       }
260     }
261 }
262
263 /*
264    parseCmd determines what actions to take based on input
265 */
266 void parseCmd()
267 {
268   // if (cmd[0] == "theta")
269   // {
270   //   int servo = cmd[1].toInt();
271   //   float inputAngle = cmd[2].toFloat();
272   //   motorRotations[servo] = offsets[servo] + inputAngle;
273   //   moveToRotations();
274   // }
275
276   if (cmd[0] == "off") //off = offset
277   {
278     int servo = cmd[1].toInt();
279     float offset_angle = cmd[2].toFloat();
280
```

```
281       Serial.print("Previous Offset is: ");Serial.println(offsets[servo]);
282       // motorRotations[servo] = offset_angle[servo] + DesiredPWM_HL[servo];
283       // update offset from global during testing
284       offsets[servo] =  offset_angle;
285
286       // print Current Servo new offset that is desired
287       Serial.print("Servo: ");Serial.println(servo);
288       Serial.print("User Entered Offset is: ");Serial.println(offsets[servo]);
289
290       LegMotion();
291
292     }
293
294 }
```