Multi-Modal Locomotion Robot

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted by:
Andrew Euredjian
Ankur Gupta
Revant Mahajan
Dante Muzila

Date: May 6, 2021

Submitted to:
Professor Michael A. Gennert
Professor Mohammad Mahdi Agheli Hajiabadi
Professor Randy Clinton Paffenroth

Abstract

A variety of animals such as primates, dogs, and bears switch modes of locomotion between quadrupedalism and bipedalism to better complete certain tasks. However, very few robotic platforms can effectively combine the two forms of locomotion. A multi-modal robotic platform with such capabilities would provide additional adaptability in unstructured environments, broadening its potential applications. Therefore, we extended an existing quadrupedal platform with the capability to transition into a bipedal stance. In this project, we built a physical robot, developed an accompanying software stack with a reinforcement learning pipeline, implemented quadrupedal locomotion, and achieved stance transition in simulation. Our integrated hardware and software platform affords future roboticists the opportunity to test and develop more adaptable locomotion strategies and increase the functionality of robots more broadly.

Acknowledgments

We would like to thank Professor Gennert, Professor Agheli, and Professor Paffenroth for their for guidance throughout this project.

We also extend our gratitude towards the Open Dynamic Robot Initiative and ODrive community for their help.

Lastly, we want to acknowledge Abigail Payne for her time and support.

Table of Contents

List of Figures

List of Tables

1 Introduction

As we aim to develop robots with improved task completion capabilities, the environments in which robots operate are becoming increasingly complex. In order to maneuver through unstructured terrains, and complete their designated tasks, robots must possess adaptable and robust locomotion strategies. For roboticists, the challenge remains in developing robotic systems and corresponding locomotion strategies that can stably and efficiently traverse unstructured terrains while also maintaining task completion capabilities.

Today, with advances in mechanical design and control systems, roboticists are implementing a variety of walking gaits and locomotion techniques on two and four-legged robotic systems. Inspired by humans, bipedal robots such as NASA's Valkyrie and Boston Dynamics' ATLAS, move on two actuated limbs and have demonstrated the ability to walk, jump, and hop [1]. These robots are not only popular because of their multiple gaits, but also because their upper limbs provide manipulation and dexterity capabilities that other robots (e.g., quadrupeds) lack. Quadruped robots mimic the physical features and locomotion strategies of dogs, cats, and other four-legged animals [2]. Robots like Spot and BigDog from Boston Dynamics [3] and HyQ from the Italian Institute of Technology [4], have demonstrated stable walking on unstructured terrain alongside trotting, squatting, and jumping capabilities.
Even though bipeds and quadrupeds have demonstrated some impressive locomotion capabilities, certain disadvantages to mono-modal locomotion currently persist. Dynamic stability is an ever-present issue for bipedal robots and traversing uneven terrain remains a challenging task [5]Click here to enter text.. Quadrupedal robots are considered to be more stable than their bipedal counterparts since they generally possess a larger support polygon [6]Click here to enter text.; however, they lack the object manipulation abilities needed to complete complex tasks [5]Click here to enter text.. This usually limits quadrupeds to reconnaissance work [1]. Furthermore, while humanoid bipeds possess the dexterity required for complex tasks, their movement is typically more energy-intensive than quadrupeds, making them comparatively inefficient walkers over long distances [7].

Just as bipedal and quadrupedal robots are biologically inspired, we can once again look towards nature in an attempt to overcome the current shortcomings of mono-modal robot systems. In the natural world there are a variety of animals who benefit from the ability to move through multiple locomotion types or achieve multiple walking gaits. Primates are perhaps the most well-known example of multi-modal locomotion. Primates can seamlessly transition between quadrupedal and bipedal walking, as well as swinging and climbing. Employing multiple gaits allows primates to move efficiently in their terrestrial-arboreal environment and adapt to different surroundings.

Unfortunately, in robotics, very few platforms can transition between multiple locomotion types. Of the multi-modal locomotion robots that have been developed, a variety of strategies have been explored, some mimicking nature and others taking a more artificial approach [8]. Among the different multi-modal locomotion strategies, a robotic platform that can stably convert between a quadrupedal and bipedal stance has the greatest potential for real-world

application. A robot with this capability would provide beneficiaries with the advantages that both quadrupedal and bipedal locomotion afford. This serves as motivation for developing a multi-modal robot that can efficiently walk long distances or traverse uneven terrain in quadruped mode while converting to biped mode to complete tasks where dexterity and object manipulation are required.

The goal of this project is to create a robotic platform capable of multi-modal transition and locomotion. This platform is intended to be used for research and development purposes; therefore, it has been designed with future usability in mind. Our platform extends an existing open-source quadruped design, swaps out all custom electronics with off-the-shelf (OTS) variants, and offers a seamless simulation-to-reality testing experience. Our simulation stack also features a built-in pipeline for Reinforcement Learning (RL) experiments. Our final system demonstrates full quadrupedal capabilities as well as bipedal standing in simulation. An easily extendible multi-modal robotic platform will decrease the steep barrier of entry into multi-modal robotics and offer opportunities to create robots that navigate a diverse range of environments.

2 Background

Over the years roboticists have developed various multi-modal robots in an attempt to create more effective and adaptable locomotion strategies. This section provides background on the field of multi-modal robotics and details the state-of-the-art in quad-bi robotics and control systems. We also discuss the Solo8, an open-source quadruped robot, and provide insight into the platform's role in our project. Lastly, we discuss Reinforcement Learning (RL) for robot controls and explain the basics of RL.

2.1 Overview of Multi-Modal Robotics

In the field of multi-modal robotics, several platforms exist that implement different types of multi-modal locomotion. Reconfigurable robotic systems achieve multi-modal locomotion through the simple premise of connectable mechatronic modules that can alter their shape and function to changing environments [9]. Russo et al. [10] developed Scout, a cubic-shaped reconfigurable robot that can perform inchworm locomotion in a three-module configuration and quadrupedal locomotion using five modules (one for the body and four for the legs). Other systems have modules that can be configured to form a snake or spider-like shape for crossing uneven terrain, as well as forming a ball or wheel for quick movement on flat surfaces [10]. Robots like the PENS-FlyCrawl, a disaster-zone reconnaissance robot developed by Kuswadi et al. [11], can fly and crawl using a bi-copter mechanism for aerial propulsion and two 360-degree rotating legs for terrestrial crawling. Legged-wheeled locomotion is another form of multi-modal locomotion, which consists of wheels mounted on legs to move, either by using its wheels, by stepping, or both [12]. The Halluc II of the Chiba Institute of Technology is an eight-legged robot equipped with driven wheels at the end of each appendage. The Halluc II can switch between wheel-cruising and leg-walking, offering increased mobility performance [5]. Work has also been done in the field of posture conversion and quadrupedal-to-bipedal (quad-bi) locomotion.

2.2 Quad-Bi Locomotion

Despite being a relatively niche field of robotics, quad-bi locomotion has been implemented on several robots. These robots range in design and capabilities. The following section details current multi-modal platforms in the field of quad-bi locomotion and describes the various control techniques used for stance conversion and movement. The foot design of certain multi-modal robots is also discussed.

2.2.1 State of the Art

There are a handful of robotic platforms that can change their posture and perform quad-bi locomotion. These platforms implement multi-modalism in different ways and to varying extents. As part of the DARPA Robotics Challenge, Carnegie Mellon University developed

CHIMP, a humanoid-inspired robot that moves on motorized tracks. CHIMP can move and stand bipedally on its rear legs, as well as change its posture to drive on all four tracks. In bipedal mode, CHIMP can use its high degree-of-freedom arms, while transitioning to its quadrupedal posture to stably move over uneven terrain [13]. WAREC-1 is a novel four-limbed robot that was designed to exhibit versatility in locomotion styles. The robot's mechanically identical legs and three degrees-of-freedom hip/shoulder joints enable WAREC-1 to stand on two or four limbs, transition between quadrupedal and bipedal walking, and perform vertical ladder climbing [14].

Quad-bi locomotion has also been achieved through designs that implement biomimicry. The Gorilla Robot III [15] and the hominid-robot Charlie [16] were designed to mimic the physical appearance and locomotion strategies of gorillas and chimpanzees, respectively. Both robots can perform bipedal and quadrupedal walking, as well as change their posture. Additionally, researchers have implemented quadrupedal locomotion on humanoid robots. Huang et al. [17] implemented quad-bi locomotion on the BHR-6, a traditional humanoid platform. In simulation, the researchers were able to develop a motion planning method that allowed the BHR-6 to dynamically transition between a hand-knee crawling gait and a bipedal walking gait. Yoon and Kim [18] approached the problem of humanoid multi-modal locomotion differently. By studying the differences in pelvis length between bipedal humans and quadrupedal anthropoids, they designed a humanoid robot with an adaptive pelvis mechanism that allows for effective quadrupedal and bipedal gaits. The robot can successfully perform bipedal and quadrupedal walking; however, the robot can not transition between the two locomotion types on its own.

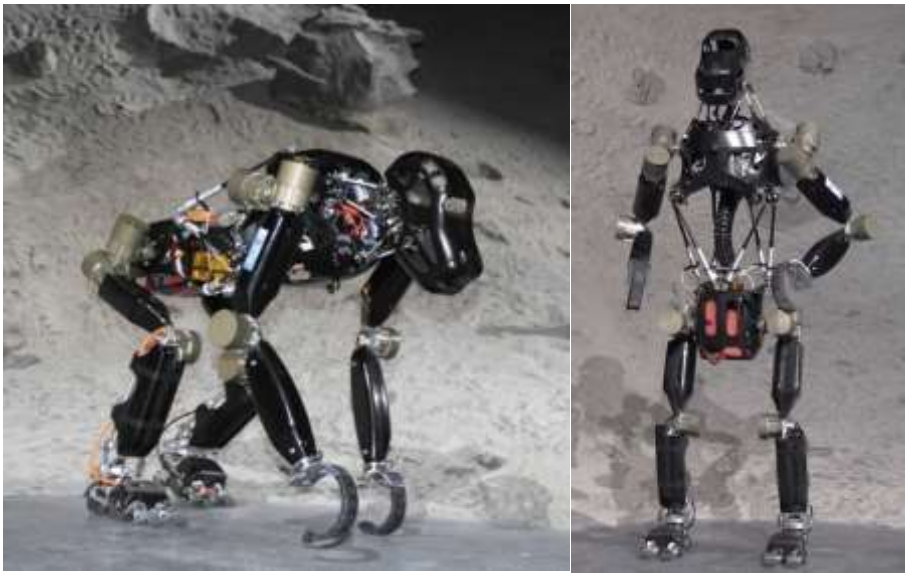


Figure 1: Charlie in quadrupedal posture (left) and bipedal posture (right) [16]

2.2.2 Control and Stance Transition

The challenge with multi-modal locomotion is developing a path planning and control algorithm that can transition between and realize both locomotion forms. To achieve a transition gait on the

BHR-6, Huang et al. [17] proposed and verified in simulation a path-generation method based on kinematic primitives of crawling and walking. The transition gait is constructed with a combination of motion primitives, created by performing a kinematic primitive analysis on obtained joint trajectories, and a polynomial interpolation of varying parameters. Adapted from the path planning scheme for the humanoid ASIMO, Kamioka et al. [19] developed a new algorithm with intermediate transitions for bipedal and quadrupedal locomotion. The algorithm is based on the linear time-variant inverted pendulum model, which is used to determine the base Zero Moment Point trajectory and calculate the trajectory at the center of gravity. Using this control method, a biped test robot demonstrated multi-modal locomotion with gait transitions.

Several control algorithms for quad-bi locomotion use Central Pattern Generators (CPGs), a biologically inspired control method that generates and controls periodic motion (ie. walking) [20]. In robotics, CPGs are a network of oscillators that act as a controller by generating rhythmic joint trajectories and providing stability properties [21]. Aoi and Tsuchiya [22] proposed and verified a control system that obtained a smooth gait transition for a robot in simulation. The control scheme implemented a CPG that created nominal joint trajectories by the phases of nonlinear oscillators. The mapping of the joint trajectories is continuously and gradually changed as the robot converts from a quadrupedal to a bipedal gait. Aoi, Tsuchiya, and others modified this control system to implement multi-modalism on a physical bipedal robot. Using the same oscillator network model, they incorporated a phase resetting mechanism to increase the robustness of the gait transition. During experiments, the bipedal test robot was able to stably transition between quadrupedal and bipedal gaits [23]. However, it is difficult for robots to generate stable walking by CPG only [20]. Asa, Ishumua, and Wada tried to solve this issue by pairing CPG with an independent posture controller. In simulation, the researchers used the bifurcation phenomenon to realize adaptive transition behavior depending on the gradient of a slope. Bifurcation of the potential function for the transition gave the robot the ability to switch from a control law for biped walking to another for quadruped walking, based on bifurcation parameters (ie. the gradient of the slope) [20].

2.2.3 Foot Design

Most quadruped robots are equipped with single-point-contact feet (point feet) because it simplifies the robot's design and control requirements. Unlike bipeds who have a considerably smaller support polygon, quadrupeds can utilize point feet because stability is provided by their large support polygon. Most bipeds are designed with some type of flat foot, whether it be passive or actuated, to increase the robot's contact area with the ground [24].

From the literature, all legged multi-modal robots at the very least have active feet, with a few having more complex flexible-active feet. Active feet are actuated by a servo or motor and provide at least one degree of freedom at the ankle. Flexible-active feet have three degrees of freedom, with an active joint at the toes and two passive, spring torsioned, joints at the heel and toes [24]. Fondahl et al [25], writes that while passive feet are sufficient for effective quadrupedal walking, it hinders stable and efficient locomotion in bipedal walking. For this reason, the previously discussed Gorilla Robot III [15] and BHR-6 [17] have active feet with

two-degrees of freedom and Charlie [16] has flexible-active feet. Charlie's feet are multi-point contact feet with three degrees of freedom at the ankle. The ankle joint, toe actuators, and flexible heel imitate the damping and Windlass mechanism of human feet [26]. This provides stable locomotion in both locomotion types and allows for smoother transition between quadrupedal and bipedal locomotion.

2.3 Solo8

The Solo8 is an open-source torque-controlled quadruped robot system developed by the Max-Planck Institute for Intelligent Systems in collaboration with New York University's Tandon School of Engineering [27]. As part of the Open Dynamic Robot Initiative (ODRI), the Solo8 was developed to enhance robot locomotion research through its lightweight and modular design. Through experiments, the Solo8 has demonstrated a diverse range of quadrupedal locomotion capabilities. The robot can jump 0.65m (twice its leg length), balance on moving platforms, and walk over uneven surfaces [27].

For our project, we chose to extend this platform because we believed the Solo8's modular and lightweight design would aid us. The robot is easy to transport, work with, and modify. However, an even greater benefit of the Solo8 is its open-source nature. All the Solo8's mechanical and electrical hardware blueprints in addition to software are open-source under the BSD-3-clause license. The robot's bill of materials and assembly instructions, along with STL and CAD files for 3D printing and part modification, are available on the Open Dynamic Robot Initiative's GitHub repository. Given the inherent time restriction of a nine-month-long project, using the Solo8's design as our starting point made achieving our project's goals more feasible.

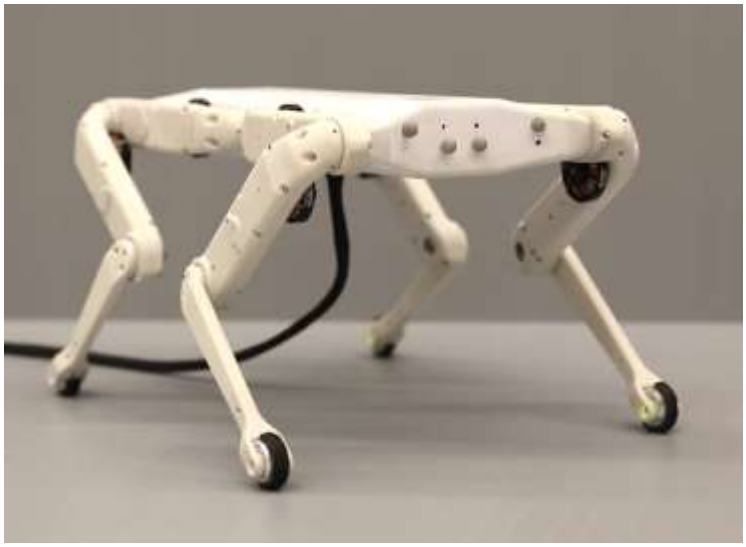
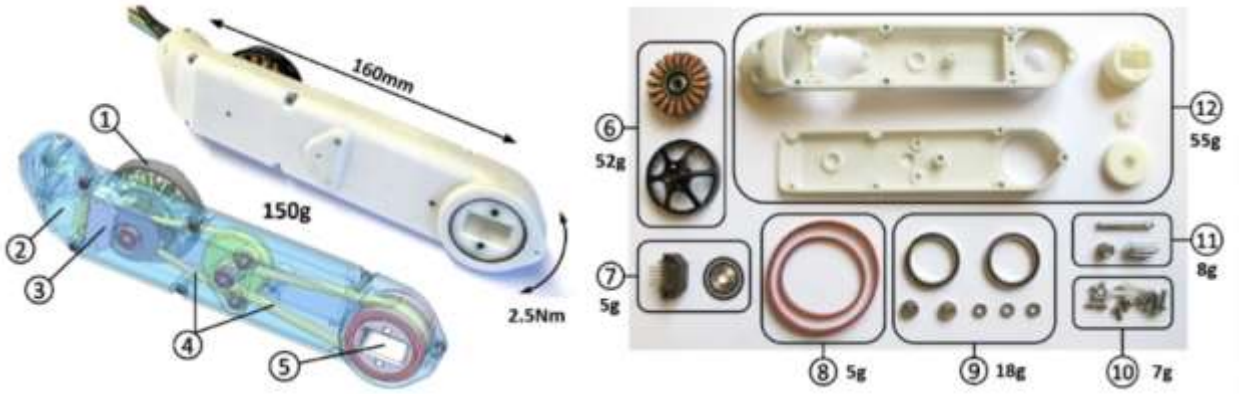


Figure 2: Solo8 [27]

2.3.1 Mechanical Design

The Solo8 consists of four identical legs and a 3D printed body structure. Like many quadrupeds, the Solo8 has point feet. The 8-DoF robot has 2-DoF in each leg and multi-revolution capable joints. An individual leg is composed of two identical brushless actuator modules and a lower leg. The actuator modules consist of a brushless motor (T-Motor Antigravity 4004, 300KV), a high-resolution optical encoder with an index pulse and a 5000 pulse-per-revolution code wheel mounted to the motor shaft. Additionally, the actuator modules have a 9:1 dual-stage timing belt transmission that allows for impedance and force control at the joints. All the actuator modules' components are housed within a lightweight, 3D printed shell. Except for the motor shafts and pulleys, which are machined from stock material, all the modules' components are either 3D printed or off-the-shelf parts. In total, the Solo8 weighs 2.2kg and has a standing hip height of 24cm, a body length of 42cm and a width of 33cm [27].



(a) Actuator Module (b) Parts Overview

Figure 3. Brushless actuator module (a) assembled and (b) parts. BDLC Motor (1), two-part 3D printed shell structure (2), high resolution encoder (3), timing belts (4), and output pulley (5). Brushless motor (6), optical encoder (7), timing belts (8), bearings (9), fasteners (10), machined parts (11), and 3D printed parts (12). [27]

2.3.2 Software

Currently, the documentation for the Solo8's software is minimal and everything we learned about the software was through communications with the original creators on the ODRI discourse forum (https://odri.discourse.group/). On the forum we were told that the Solo8's software stack is based on the dynamic graph concept to generate dynamically stable walking gaits. The steps involved in executing an algorithm are as follows.

1. Prototype and implement an algorithm in simulation (PyBullet) using Python
2. Translate the algorithm into C++ to execute on the real robot
3. Embed the algorithm in a dynamic graph to be executed
4. Execute it on the real robot.

Upon pursuing more research on the dynamic graph concept, we could not gather more information. Since our only means of gathering information was by communicating with the Solo8 developers, the team decided to pursue other avenues for time purposes. Since we were having difficulty getting basic information about the software during the planning phase of the project, we were skeptical about the feasibility of using their undocumented software for our project timeline. Therefore, we decided to create our own software stack. This decision is explained more in Section 3.1.

2.3.3 Electronics

The Solo8's electronics are contained within the body structure and the robot is externally powered and remotely controlled by a PC. The motors are controlled with off-the-shelf TI micro-controller evaluation boards and custom TI motor driver electronics. The micro-controllers are equipped with two BLDC booster cards that are capable of Field Oriented Control (FOC) and can execute dual motor torque control at 10 kHz. The motor driver electronics were miniaturized by a volume factor of ten to reduce the electronics footprint. The resulting MPI Micro-Driver electronics are open-source and consist of a Texas Instruments micro-controller (TMS320F28069 M) and two brushless motor driver chips (DRV8305) on a single six-layer printed circuit board. The board includes a JTAG port for programming, CAN and SPI ports for control communications, and operates at motor voltages up to 40V [27].

2.4 Reinforcement Learning for Robot Control

In traditional robot control algorithms, a roboticist orchestrates a movement by changing motor values (positions, torques, etc.) to perform a task. However, the human designed solution is not always the most efficient way to complete the given task.

Reinforcement Learning (RL) is an active field of research that aims to remove human bias when solving a task. RL methods use Machine Learning (ML), a technique used to infer the behavior of an arbitrary function, to "teach" robots to complete tasks by themselves. These algorithms only require information about the environment (observations, how well it is performing, etc.), heavily reducing the amount of human bias in the algorithm design process. This section offers a working introduction to ML, how it is used within RL, and concludes with the algorithms chosen by this team.

2.4.1 A Gentle Introduction to Machine Learning

In computer science, machine learning is a class of algorithms that aims to discover relationships in data without any prior information on the data's domain. These algorithms can be broadly described as *supervised* and *unsupervised*, based on the nature of the data. Unsupervised machine learning algorithms find hidden patterns in data without the need for any human intervention. Common unsupervised machine learning problems include clustering and dimensionality reduction. In contrast, supervised machine learning algorithms require a *ground*

truth, or prior knowledge of a desired output value for a given input. If the desired output is discrete, then the machine learning task is described as *classification*. Otherwise, if the desired output is continuous, the task is described as *regression*. The scope of this project only involves supervised machine learning models aimed at solving regression tasks.

2.4.1.1 Neural Networks

Artificial neural networks (ANNs) are a type of machine learning technique that mimic the structure of the human brain to learn the solution to a task [28]. ANNs can be broken down into their atomic units called artificial neurons, or neurons for short. Each neuron receives an input signal and sends an output signal to its connected neurons, which receive the signal as an input [28].

In most neural networks, neurons are split into groups called *layers*. In feedforward networks, layers are connected sequentially, where each neuron's output from a given layer is every neuron's input in the next layer. Note that in feedforward neural networks, no cycles exist between any two neurons. Neural networks with cycles between neurons are called *recurrent neural networks* and have shown great success in representing sequence-dependent data [29]. However, recurrent neural networks are outside the scope of this project.

Therefore, the input to any given neuron is simply a linear combination of the outputs of the previous layer with a vector of weights (**w**) and biases (**b**). An *activation function (σ)* is then applied to this combination, and this result is the output of the neuron. Hence, the output (*z)* of a neuron in layer l is expressed as:

$$z_i^{(l)} = \sigma\left(\sum_{j=1}^{n^{(l-1)}} \mathbf{w}_j^{(l-1)} \boldsymbol{z}_j^{(l-1)} + \mathbf{b}_j^{(l-1)}\right) \tag{1}$$

where:

- $n^{(l-1)}$ is the number of neurons in layer $l-1$
- $\mathbf{z}_{\mathrm{j}}^{(l-1)}$ is the output of the *i*th neuron in layer $l-1$
- $\mathbf{w}_j^{(l-1)}$ is the weight associated with $\mathbf{o}_{\mathbf{j}}^{(l-1)}$
- $\boldsymbol{b}_j^{(l-1)}$ is the bias weight associated with $\mathbf{o}_{\mathrm{j}}^{(l-1)}$
- σ is the activation function

Then, we can define $\mathbf{z}^{(l)}$ to be the vector output for a layer l as:

$$\boldsymbol{z}^{(l)} = \begin{bmatrix} z_1^{(l)} & z_2^{(l)} & \dots & z_n^{(l)} \end{bmatrix}^{\mathrm{T}} \tag{2}$$

Therefore, in vector notation, this entire operation per neuron layer can be expressed as:

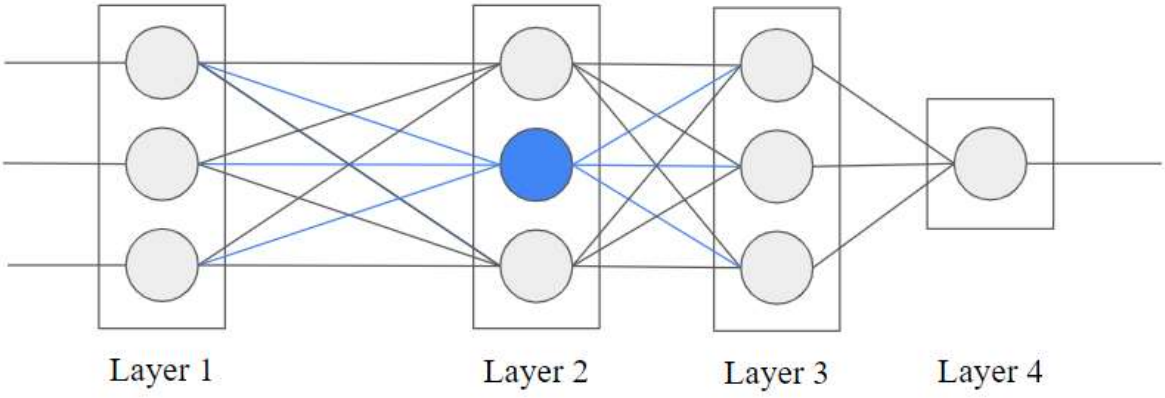


Figure 4. A four-layer, feed forward neural network. Observe that the input to highlighted neuron in the second layer is just a linear combination of layer one's outputs, as per (1). Note that the inputs to layer one considered the inputs to the entire network and that the output of layer 4, or $\boldsymbol{z}^{(4)}$, is the output of the entire network.

$$\boldsymbol{z}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}) \tag{3}$$

In this case, $\mathbf{W}^{(l)}$ is an $m \times n$ matrix, where m is the number of neurons in layer l (the current layer) and n is the number of neurons in layer $l-1$. One particularity to note is that σ, even though it is given a vector input, is applied element-wise. Additionally, one requirement of σ is that it be differentiable—this is elaborated upon in Section 2.4.1.2.

Thus, for a feedforward neural network with L layers, $\mathbf{z}^{(l)}$ is the input to $\mathbf{z}^{(l+1)}$ and $\mathbf{z}^{(L)}$ is the final output of the network. A simple example of a neural network with four layers can be seen in Figure 4. Formally, we express the final output $\mathbf{z}^{(L)}$ of the network as a function of a single data point $\mathbf{x}$:

$$\mathbf{z}^{(L)} = \sigma^{(L)}(\mathbf{W}^{(L)} \cdots (\sigma^{(2)}\left(\mathbf{W}^{(2)}\sigma^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) \cdots + \mathbf{b}^{(L)} \tag{4}$$

Note that $\mathbf{x} \in \mathbb{R}^{n^{(1)}}$, where $n^{(1)}$ is the number of neurons in the input layer. Additionally, the activation function σ can differ between layers, so $\sigma^{(l)}$ refers to layer l's activation function.

2.4.1.2 Objective Functions & Learning

As mentioned in Section 2.4., since we are interested in building a regressor, we sample a response variable $\mathbf{y}$ which is the output of the ground-truth function $f(\mathbf{x})$. However, the ground-truth function is unknown; therefore, supervised machine learning aims to best approximate $f(\mathbf{x})$ when only given $\mathbf{x}$ and $\mathbf{y}$.

By stacking layers with different amounts of neurons, neural networks can effectively transform an input vector $\mathbf{x}$ into any desired dimensional space. Therefore, neural networks can be viewed as a function on arbitrary dimensions:

$$\hat{\mathbf{y}} = \hat{f}(\mathbf{x}) : \mathbb{R}^{\mathrm{n}} \rightarrow \mathbb{R}^{m} \tag{5}$$

where

- $\hat{f}(\mathbf{x})$ is the neural network attempting to approximate the ground truth function $f(\mathbf{x})$
- $\hat{\mathbf{y}}$ is the guess of the neural network. Note that $\hat{\mathbf{y}} = \mathbf{z}^{(L)}$
- n is the dimensionality of $\mathbf{x}$. Observe that this value is the same as the number of neurons in layer $l = 1$
- m is the dimensionality of the output, or $\hat{\mathbf{y}}$. This is also the same number of neurons in layer $l = L$.

Once dimensioned correctly, the performance of a neural network is evaluated via an *objective function*. The neural network is then iteratively optimized with respect to the objective function to reduce the amount of error in the approximations [30].

In regression tasks, one of the most popular objective functions is Mean Squared Error (MSE), simply defined as

$$MSE = \frac{1}{2p}\sum_{i=1}^{p} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|_2 \tag{6}$$

Recall that $\|\hat{\mathbf{y}}_i - \mathbf{y}_i\|_2$ is the l2-norm, or the Euclidean distance, between the ground truth $\mathbf{y}_i$ and the network's approximation $\hat{\mathbf{y}}_i$.

Observe that MSE in (6) is uniformly differentiable. Additionally, as σ is required to be differentiable—initially stated in Section 2.4.1.1—notice that $\hat{\mathbf{y}}$ is comprised of purely differentiable operators: function composition, multiplication, and addition. Therefore, $\hat{\mathbf{y}}$ is differentiable.

By the chain rule, we know that $\frac{\partial MSE}{\partial \hat{\mathbf{y}}}$ exists. Therefore, for any tunable weight w encased in either a weight matrix $\mathbf{W}$ or a bias weight vector $\mathbf{b}$, we can compute $\frac{\partial MSE}{\partial w}$ as

$$\frac{\partial MSE}{\partial w} = \frac{\partial MSE}{\partial \hat{\mathbf{y}}}\frac{\partial \hat{\mathbf{y}}}{\partial w} \tag{7}$$

Recall that for a vector $\mathbf{v}$, and a function f, the gradient of f w.r.t. $\mathbf{v}$ is defined as:

$$\nabla_{\mathbf{v}} f = \begin{bmatrix} \frac{df}{d\mathbf{v}_1} & \frac{df}{d\mathbf{v}_2} & \dots & \frac{df}{d\mathbf{v}_n} \end{bmatrix}^{\mathrm{T}} \tag{8}$$

where n is the number of elements in $\mathbf{v}$.

Then, by (7), (8) and the chain rule, for any layer l in a neural network, $\nabla_{\mathbf{W}^{(l)}} f$ and $\nabla_{\mathbf{b}^{(l)}} f$ can be computed. Recall that the gradient is a vector pointing in the direction of greatest change [31].

Using these weight gradients, the neural network can iteratively "learn" to reduce the error in its approximations. Since MSE is the error between the ground truth $\mathbf{y}_i$ and the network's guess $\hat{\mathbf{y}}_i$, $\nabla_{\mathbf{v}} MSE$ is the partial derivatives of MSE with respect to $\mathbf{v}$ to *increase* the error most quickly.

However, since the goal of learning is to *decrease* the error, $\nabla_{\mathbf{v}} MSE$ can simply be negated to find the direction of quickest error decrease. Then, a small update can be made to all the weight parameters:

$$\begin{aligned}
\mathbf{W}^{(1)} &= \mathbf{W}^{(1)} - \alpha \nabla_{\mathbf{W}^{(1)}} MSE \\
\mathbf{b}^{(1)} &= \mathbf{b}^{(1)} - \alpha \nabla_{\mathbf{b}^{(1)}} MSE \\
\mathbf{W}^{(2)} &= \mathbf{W}^{(2)} - \alpha \nabla_{\mathbf{W}^{(2)}} MSE \\
\mathbf{b}^{(2)} &= \mathbf{b}^{(2)} - \alpha \nabla_{\mathbf{b}^{(2)}} MSE \\
&\vdots \\
\mathbf{W}^{(L)} &= \mathbf{W}^{(L)} - \alpha \nabla_{\mathbf{W}^{(L)}} MSE \\
\mathbf{b}^{(L)} &= \mathbf{b}^{(L)} - \alpha \nabla_{\mathbf{b}^{(L)}} MSE
\end{aligned} \tag{9}$$

Note that the gradients' updates are being subtracted—this is to *reduce* the error, as mentioned previously. Additionally, as $\nabla_{\mathbf{v}} MSE$ is simply the *direction* of greatest change, it is multiplied by a small value α, known as the *learning rate*, to ensure that the gradient update does not overshoot back and forth during training.

Efficiently computing these gradient updates is known as *backpropagation* [30] and is outside the scope of this project. However, most deep learning libraries such as Pytorch [32] and Google's TensorFlow [33] have backpropagation efficiently implemented. Using backpropagation and gradients to iteratively reduce loss is an algorithm known as *gradient descent*. While there have been improvements to gradient descent—such as ADAM, which dynamically adjusts the learning rate [34]—the idea of using the gradient to reduce the loss on an arbitrary function is the key intuition to making neural networks learn.

2.4.1.2 Choosing Activation and Objective Functions

In neural networks, activation functions typically dictate *how* the network will learn while objective functions dictate *what* the network will learn. In the original neural network, the Heaviside step function was used, which directly models the all-or-none firing nature of biological neurons [35]:

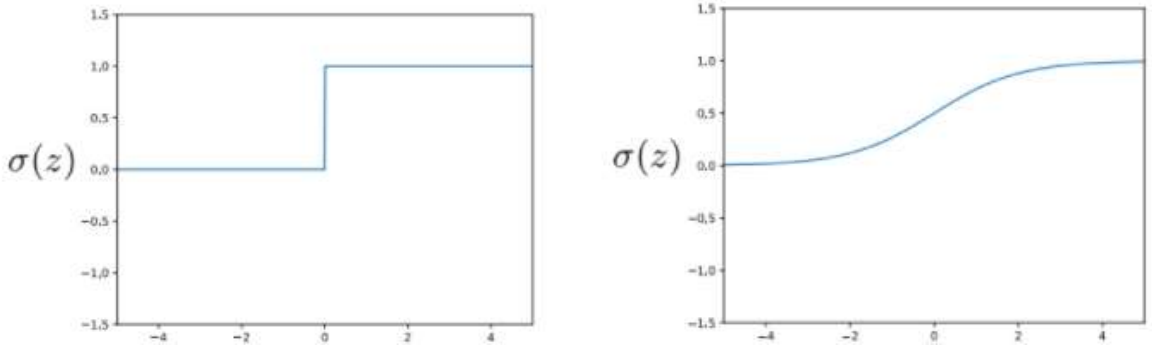


Figure 5. On the left, the Heaviside function as defined in (10). On the right, the logistic sigmoid, as stated in (11).

$$\sigma(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} \tag{10}$$

A visualization of this function can be seen in Figure 5. However, the gradient is 0 over almost the entire domain. Considering how important of a role the gradient plays in the learning process, this was largely replaced by the logistic sigmoid, which can be seen in Figure 5:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{11}$$

Recall from (4) that the final output of a multi-layered neural network is a large composition of the activation functions. Additionally, recall the chain rule for a function $f(g(x))$:

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx} \tag{12}$$

Therefore, for neural networks that have many layers, the computations for the final layers' gradients must undergo tens, if not hundreds of multiplications. Observe that at large magnitudes within the domain, the logistic sigmoid's gradient is near zero. A common error that can arise while training large networks is that the final layers' gradients undergo many near-zero multiplications. This can cause floating-point inconsistencies and end up with a gradient of 0, preventing any future learning. This is often referred to as the *vanishing gradient problem* [36].

Rectified Linear Units (ReLU) attempt to solve this problem by forcing a fixed value for the derivative of all positive values. ReLU, seen in Figure 6 is defined below:

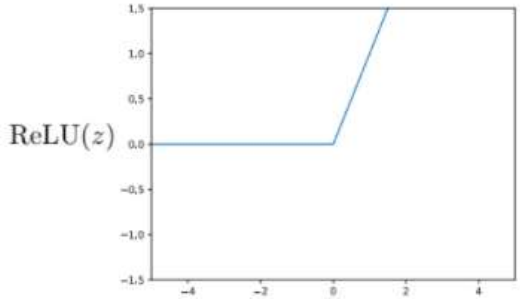


Figure 6. The ReLU activation function, as stated in (13)

$$ReLU(z) = \max\{0, z\} \tag{13}$$

Here, the derivative of ReLU is always either 1 or 0. Even though ReLU does have a zero derivative, having a fixed derivative for all positive domain values is enough to "restart" a neuron with a zero gradient [36]. The ReLU activation function has shown great empirical success in training modern neural networks and was selected for this project.

In contrast, objective functions control *what* the network learns. In the preceding example, MSE was used as the objective function of choice; however, the only requirements for a loss function (an objective function that is minimized instead of maximized) is that it is differentiable and always positive. MSE is the simple Euclidean distance between two vectors, so it fits the criteria for an objective function that aims to minimize l2-error. Many advanced applications of machine learning often involve directly modifying the objective function to get the desired behavior from the network.

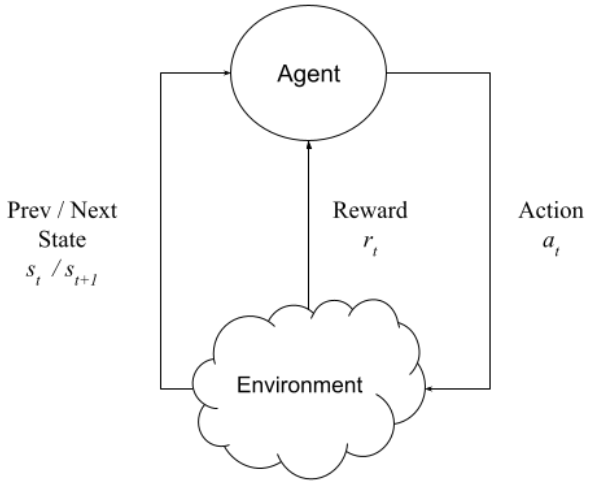


Figure 7. The RL representation of an agent-environment interaction.

2.4.2 Reinforcement Learning

Reinforcement Learning (RL) uses ML to teach autonomous agents how to optimally navigate an environment. RL differs from traditional supervised learning problems in that the ground truth labels are usually discovered by the agent during exploration. This section offers an overview of RL and the techniques used in our experiments.

2.4.2.1 Agents & Environments: The RL Problem Formulation

In RL problems, an agent evaluates the state s_t of the environment at timestep t and performs an action a_t. The agent's decision-making process is called its *policy*. Typically, an agent's policy is represented by a probability distribution $\pi(a_t|s_t)$. The environment responds—sometimes non-deterministically; imagine a car on any icy road—with a new state s_{t+1}. During training, the environment also gives the agent a reward r_t based on the agent's previous action. This cycle can be seen in Figure 7.

The agent continues this cycle for a fixed number of timesteps T or until the environment terminates it, such as in a video game. This collection of tuples, $((s_1, a_1, r_1), (s_2, a_2, r_2), \ldots (s_T, a_T, r_T))$ that describe the agent's interaction with the environment is called a *trajectory* or an *episode*.

However, some states are more desirable than others. For example, an agent driving under the speed limit is probably performing better than an agent skidding out of control—assuming the agent's task is to travel safely. The *value* of a state is its expected discounted sum of rewards over time. Mathematically, the discounted sum of rewards starting at state s_t is expressed as:

$$r(s_t) = \sum_{j=t}^{T} \gamma^{j-t} r_j \tag{14}$$

Where $t \in [1, T] \cap \mathbb{Z}$ and $\gamma \in [0, 1)$. As gamma is strictly less than one, observe that rewards later in the trajectory have less of an effect on the value of the state. As more trajectories are computed, the value, $V(s_t)$, converges to $\mathbb{E}[r(s_t)]$, the expected reward starting at state s_t. Thus, the value function $V(s)$ effectively ranks the state space of the environment.

As the expected sum of rewards is computed per trajectory, it tends to have a large sampling variance. To reduce the variance and help assist training, the *advantage* of a state is typically used rather than the value or the discounted sum of rewards. The advantage for a state s_t is defined as

$$\begin{aligned} A_t &= r(s_t) - V(s_t) \\ &= \sum_{j=t}^{T} \gamma^{j-t} r_j - \mathbb{E}\left[\sum_{j=t}^{T} \gamma^{j-t} r_j\right] \end{aligned} \tag{15}$$

The advantage function simply subtracts the expected rewards from the collected rewards—effectively making A_t unbiased.

2.4.2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) was initially introduced in 2017 by OpenAI as an RL algorithm for continuous control [37]. Since 2017, iterations have been made to PPO to optimize it for data sampling efficiency as well as runtime improvements on NVidia CUDA-enabled Graphics Processing Units (GPUs). PPO was designed to overcome the shortcomings of previous RL algorithms; therefore, it heavily resembles algorithms such as Advantage Actor Critic [38] and Trust Region Policy Optimization [39]. Unfortunately, a full history of RL techniques is outside the scope of this paper and a working explanation for PPO will be offered instead in this section.

As mentioned in Section 2.4.2, RL's primary difference against traditional supervised learning is that the agent is actively interacting with the environment to gather data. Thus, traditional training measures such as MSE can cause aggressive gradient updates which make the environment irrecoverable. For example, imagine a robot attempting to walk up a flight of stairs. Even one misstep could cause the robot to tumble all the way down—resulting in a new environment that the network has never trained for. One of the primary challenges in RL is effectively limiting agent exploration with minimal consequences during training.

When designing PPO, OpenAI intended for it to "perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune" [37]. As such, PPO begins by replacing the policy with a neural network that optimizes a very intuitive objective function:

$$maximize\ \mathbb{E}[A_t] \tag{16}$$

Note that the policy network π_θ is a neural network that inputs s_t and outputs a distribution for the action a_t to be sampled from. In practice, this usually means that π_θ outputs a mean and a standard deviation—but most ML libraries have their own canonical implementation of statistical distributions [32], [33].

Here, PPO's objective is very straightforward: optimize the policy network π_θ such that the expected advantage is increased. In practice, this involves collecting a set of trajectories under a policy π_θ, computing the advantages, and using that data to train π_θ. Note that to prevent bias, after every iteration of updating π_θ, a new set of trajectories will need to be collected.

This simple objective function is known to cause relatively large training updates and be unstable while training policy networks [39]. To help mitigate this, a policy ratio is applied to A_t, yielding:

$$maximize\ \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t\right] \tag{17}$$

Here, a copy of the policy network, $\pi_{\theta_{old}}$ is saved between iterations. By maintaining $\pi_{\theta_{old}}$, we can not only get the probability of the current policy choosing an action, $\pi_\theta(a_t|s_t)$, but also the probability of the previous policy, $\pi_{\theta_{old}}(a_t|s_t)$.

By introducing the probability ratio, A_t effectively gets amplified when the current π_θ is more likely to perform an action and vice versa. This only works because we know from (15) that A_t is unbiased and that less-optimal actions correspond to negative A_t values.

However, observe that $\pi_\theta(a_t|s_t)$ is just a single point in the action space; this objective function does not consider the *distribution* of all possible actions. PPO addresses this by punishing large differences between distributions:

$$maximize\ \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t\right] - \beta\mathbb{E}\left[KL(\pi_\theta, \pi_{\theta_{old}})\right] \tag{18}$$

Where $KL(\pi_\theta, \pi_{\theta_{old}})$, is the Kullback-Leibler (KL) divergence for the π_θ and $\pi_{\theta_{old}}$ distributions evaluated at state s_t. Recall that the KL-divergence measures the difference between two data distributions P and Q:

$$KL(P, Q) = \mathbb{E}_x\left[\log\frac{P(x)}{Q(x)}\right] \tag{19}$$

and that larger values of KL correspond to a greater difference between distributions [40].

This KL-penalty is then multiplied by a variable β that controls the weight of the KL-penalty. In implementation, β is dynamically adjusted. Another hyperparameter is typically introduced, δ, which is the target KL-divergence per iteration. However, if $KL\left(\pi_\theta, \pi_{\theta_{old}}\right) > 1.5\delta$, that means that the agent is making too drastic of an update and β is simply doubled to punish that for the next iteration. If $KL\left(\pi_\theta, \pi_{\theta_{old}}\right) < \frac{\delta}{1.5}$, then the converse is true and β is simply halved as a result. The decision to multiply and divide δ by 1.5 is arbitrary but is recommended by the original authors [37]. By automatically adjusting β, PPO effectively allows the researcher to control the exploration per iteration via δ. Finally, note that the penalty is being *subtracted* because the overall objective function is being *maximized*. The entire algorithm and training process can be seen in Figure 8.

The combination of a policy ratio and a KL penalty has shown great empirical success in RL applications. Even as late as 2019, PPO has shown the most consistent convergence among top RL algorithms [41]. As such, PPO is one of the top algorithms of choice for researchers exploring new problems; it allows them to focus on how the environment should reward the agent rather than having to worry about the nitty gritty details of optimization.

OpenAI later silently released PPO2, a CUDA GPU-accelerated version of PPO. PPO2 was shown to run 3x faster than PPO on OpenAI's Atari environments [37] and was the

algorithm chosen in this project. Using PPO, Google DeepMind was able to achieve quadrupedal standing on a quadruped; however, their quadruped has 12 DoFs compared to the Solo8's 8 [42]. Since we were only considering 8 DoFs in our problem, we were hopeful that we would be able to successfully use PPO in our project.

Input: initial policy parameters θ_0, initial KL penalty β_0, target KL-divergence δ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta||\theta_k)$$

  by taking K steps of minibatch SGD (via Adam)
  if $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \geq 1.5\delta$ **then**
    $\beta_{k+1} = 2\beta_k$
  else if $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \leq \delta/1.5$ **then**
    $\beta_{k+1} = \beta_k/2$
  end if
end for

Figure 8. The PPO with Adaptive KL Penalty algorithm from the original paper [37]. Note that $\mathcal{L}_{\theta_k}$ is simply OpenAI's shorthand for (17) and that $\bar{D}_{KL}(\theta||\theta_k)$ is the KL-divergence between π_θ and $\pi_{\theta_{old}}$, as explained in (19). Observe that at the end of every iteration, β is adjusted to maintain a target KL-divergence.

3 Design Decisions

While the Solo8 provided a good starting point for the project, the platform required modifications for us to achieve our goals. As discussed in Section 2.3 the software stack was convoluted and required multiple code re-writes to go between the simulation environment and real robot. The robot's electronics where custom built and lightly documented, making it difficult to replicate them and learn how to use them. The robot's point feet were also not conducive with bipedal standing as they provided a small bipedal support polygon. To overcome these shortcomings, we modified robot's software architecture, electronics, and mechanical design to varying degrees. This section details our design decisions, how we made them, and provides justifications for them.

3.1 Software Stack

As mentioned in Section 2.3.2, the Solo8's software stack had various constraints that made it difficult for us to implement the software in our project. Being new to the platform, we were unfamiliar with the ODRI's specific tools and libraries. This problem was compounded by a severe lack of documentation regarding the Solo8's software stack and slow response time on the ODRI discourse forum. Since this was our primary means of communicating with the Solo8 developers, we were skeptical of being able to address any unforeseen issues and bugs with the software in a timely manner. Therefore, we decided to develop our own software stack for this project because we felt more confident in our ability to create a custom stack that would meet our project's goals and timeline. We decided to use standard tools and libraries, like ROS and Arduino, because we were familiar with them, and they provided better access to community support.

On the ODRI forum, we discovered that the original Solo8 has a multi-language software stack for simulation and the robot. Algorithms must be written in Python to be tested in simulation. When porting over to the robot, these algorithms are then converted to C++. We thought that this conversion step from Python to C++ could potentially induce bugs that would be difficult to troubleshoot on the real robot. As such, we decided to create a single language software stack and implemented it in Python. By doing so, we could improve the simulation-to-robot pipeline and make it easier to use.

We also decided to split the robot control between two controllers. The Raspberry Pi is responsible for all higher-level planning and control (quadrupedal trajectory, bipedal trajectory) and the Arduino is responsible for all lower-level control (controlling motors, communicating with sensors). Our decision to use this hardware is further explained in Section 3.2. We decided to use ROS2 Dashing on our Raspberry Pi. ROS has a lot of benefits in terms of already implemented standard tools and libraries for various standard hardware. Originally, we were planning to use ROS Melodic instead of ROS2 due to our familiarity with it and its arguably larger presence in the field. Later, we switched to ROS2 given its native support for Python 3 and full availability of features and libraries that we thought might be useful for this project. ROS2 also has a growing community with heavy active development; so, we predict that ROS2

will only get better over time. Furthermore, using ROS2 ensures compatibility with sensors like Lidars, depth cameras, etc. for future use cases.

Based on the above-mentioned decisions, we created a software architecture that can be found in Appendix A. A concise version of the software architecture diagram can be found in Figure 9. The software architecture can be broken down into three primary components: path generation and execution, simulation, and Reinforcement Learning (RL). We also decided to structure our code so that the same program would run in both the simulation as well as the real robot; doing this increased our confidence when porting our code. For the real robot, we divided the software stack into two parts, one part responsible for all the actuating, sensing, and lower-level implementations, and the other part responsible for higher-level decision making. We decided to use the ROS2 framework for the higher-level system. The motion to be executed--implemented on ROS2 --is divided into two parts: path generation and trajectory executor (denoted by 'dyn_stabalizing' in (Appendix A). The logic separation allows the user to independently define the path to be executed. The path generator also needs to handle the case when the robot gets stuck in an unrecoverable state during the execution of the trajectory by producing a new path. The trajectory executor converts the path into a trajectory. It is also responsible for executing this trajectory, making minor adjustments to keep the robot on the predefined path as needed, and terminating the trajectory if the robot goes into an unrecoverable state. An example of an unrecoverable state would be the robot falling over. In order to execute the trajectory, the trajectory executor on the Raspberry Pi communicates with the Arduino over a serial UART connection. The Arduino is responsible for controlling the motors, reading data from the sensors, and storing the current state of the robot. Furthermore, at the end of every control loop cycle, the trajectory executor requests the current state of the robot from the Arduino. Using this information, any deviations from the goal trajectory will be calculated. To successfully complete the trajectory a best effort is made to employ the corrections. If the robot goes into an unrecoverable state, the trajectory executor lets the path generator know. It is now the path generator's responsibility to define the correct behavior to handle such a case. By handling unforeseen terminations this way, we give users the option to define their own error handling, leading to more robust and tweakable solutions. We use the industry standard serial communication practices of preamble and checksum verification for serial communication to ensure data packet validity.

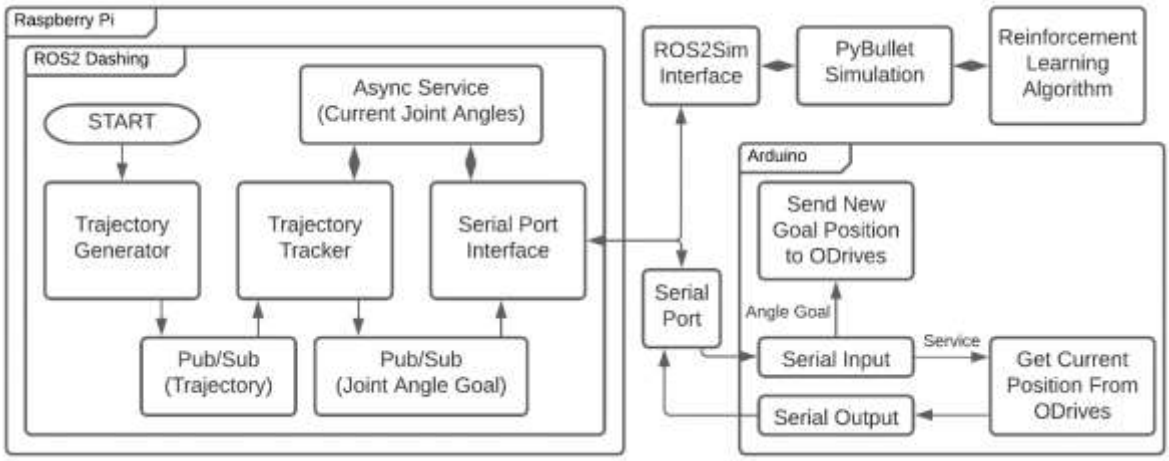


Figure 9: Diagram of Software Stack (Concise)

A similar process is used to control the robot in simulation. In this case, both ROS2 and the simulator run on the same platform and all communication is done via software serial ports. Since we had a need for both software and real serial communication, we decided to create an abstract communication node for ROS2. This way, the appropriate mode of communication is dynamic to the platform: serial for the real robot and software serial for simulation. By doing so, no user change is required to switch mode of communication. This was in line with our vision of creating a seamless transition between the simulation and the real robot. We created a wrapper that acts as an intermediary between ROS2 and the simulator with the sole purpose of providing seamless communication between the two. For our simulator, we decided to use PyBullet, an open-source physics simulator prevalent within the RL community. While there is not much associated documentation, the Solo8 creators did confirm using PyBullet as their simulator, giving us confidence in its abilities. To abstract all our problem-specific knowledge we created a wrapper for PyBullet simulator. In accordance with the open-source nature of our project, we designed our wrapper to be modular, well-tested, and conformant to domain standards--specifically the OpenAI Gym API.

Abstracted out, our problem shares quite a few similarities with many other (solved) problems in Reinforcement Learning: a continuously controlled agent is being given a set of observations and tries to optimize a continuous reward. As such, many experts in the field have released open-source implementations of the top RL algorithms. Because these implementations are highly optimized and verified to be correct, we decided to use these off-the-shelf solutions as much as possible. This is where the well-thought design of the simulation wrapper comes into play. As we already conforming to the field's golden standard, OpenAI's Gym API, our simulated Solo8 environment is immediately compatible with both stable-baselines and Tensorflow Agents- the two most actively maintained implementations of RL algorithms available. With these libraries, we now have access to the top continuous-control learning algorithms (PPO, TRPO, DDPG).

We also wrote an autotrainer framework to automate the entire hyperparameter tuning step for Reinforcement Learning. This autotrainer heavily leveraged the Weights & Biases

(W&B) service. W&B is an online machine learning monitoring tool. However, they also offer a "sweeps" feature, where W&B coordinates your training servers to tune your model. W&B also uses a probabilistic model to intelligently tune the hyperparameters, rather than the industry-standard random search. Additionally, as we already have access to implementations of various RL models, we designed the W&B hyperparameter search to not only tell us what the best parameters for a given algorithm are, but also what the highest performing algorithm is. This functionality is reduced to one command: *wandb agent*, which can be run on any computer to start intelligently training.

To ensure that we did not accumulate too much technical debt near the end of the project, we decided to diligently follow proper coding practices. Every change that was pushed into the master branch of our code repository was reviewed by at least one other team member. Additionally, we followed PEP standards and stylistic formats for Python for easy code readability. Throughout the project, we were very stringent about making sure our code was tested and reliable.

3.2 Electronics Changes

Instead of using the Solo8's custom electronics, we decided to use off-the-shelf (OTS) hardware. The only electronic components that we decided to keep from the original Solo8 were the motors (T-Motor's antigravity 4004 KV300) and encoders (Broadcom AEDT-9810-Z00). By using the antigravity motors, we saved ourselves the time of having to redesign the actuator modules to accommodate a new motor. Reasoning for this decision is give in Section 3.3. The decision to use OTS hardware also went hand-in-hand with our decision to implement a custom software stack. We believed that OTS electronics would better match our software requirements, since our unfamiliarity with the Solo8's custom electronics would make porting our software stack onto the custom boards a challenge. The replacement electronics consists of three major components: a single board computer, an Arduino microcontroller, and brushless motor controllers. The team researched various off-the-shelf electronics to determine the best fit for each component. We specifically focused on OTS parts due to their wider community support.

3.2.1 Single Board Computer

When looking at single board computers (SBCs), the team's criteria for selection included price, community support, connectivity, performance, and applicability to our project. The four contenders included the Raspberry Pi 4B, the Jetson Nano Dev Kit, the LattePanda, and the BeagleBone. Other SBCs were considered but were similar enough to these boards in all categories while being less powerful that they did not make the cut. The chosen SBC needed to interface with an Arduino microcontroller through a serial connection and run ROS on Ubuntu 18.04. This section covers the pros/cons of each of the four SBCs mentioned and explains which board(s) our team decided to use.

BeagleBone

The BeagleBone has a few different variants that specialize in different robotics topics. For example, the BeagleBone AI is geared towards more computationally heavy tasks while sacrificing performance in other areas outside the processor, such as RAM (only 1GB). Additionally, the BeagleBone product line generally limits options for connectivity by having a single serial port in some cases and none in others. On the positive side, the BeagleBone has considerable community support. The prices for these boards range from ~$30 to ~$150.

LattePanda

The LattePanda also comes in a wide range of variants. The prices of these boards range from ~$100 to ~$900. The LattePanda is an SBC with a processor capable of running Windows 10 as well as having an Arduino coprocessor on the same board with the appropriate pinout headers. Although the board has this integration between a more powerful than average processor and Arduino coprocessor, the Arduino coprocessor is based on a Leonardo. Leonardo is one of the smaller Arduino boards that lacks pinouts. Although we did not foresee needing too many pinouts, we wanted to have as many as possible to cover unforeseen future needs. Also, the combination of the x86 processor and Arduino coprocessor into one board brings about potential new complications or issues that could be avoided by having separate boards. Additionally, these boards are preloaded with Windows 10. Since we decided to use ROS, we would have had to replace the default OS with Ubuntu anyways. Traditionally, such a procedure is straightforward on x86 based systems. However, due to the unique configuration of this board, such a change could also bring about unwanted complications. Although the LattePanda is a very intriguing board, the uncertainties associated with the board's construction disqualified it for use in our project as our team needed to minimize as many potential issues as possible.

Jetson Nano Dev Board

The Jetson Nano Dev Board has a couple of different variants that have similar features. The Nano takes the general form and configuration of a Raspberry Pi while adding a NVidia GPU to make the board more capable in AI applications. NVidia also provides a custom Linux image to use on the Nano based on Ubuntu 18.04 and preloaded with various packages/libraries commonly used in AI. In terms of pinout and connectivity, the Nano mimics the Raspberry Pi 3B. It carries most of the same external device connections while also adding a Display Port and barrel jack power input. Its pinout is also largely the same as the Pi's, consisting of interfaces for various communications protocols (I2C, I2S, SPI, UART). However, due to this board being relatively new in the market, it does not have a very large support community compared to other boards. In some instances, support forums for the Pi can be applied to the Nano, but there is no guarantee of compatibility. The price of the Jetson Nano Dev Board is ~$100.

Raspberry Pi 4B

The Raspberry Pi has been around for several years and has had multiple evolutions, the most recent being the 4B model with options of 1, 2, 4, or 8GB RAM. The Pi is one of, if not the most popular SBC in the robotics community with an exceptionally large support community. The Raspberry Pi Foundation provides a custom image for the Pi called "Raspbian" based on Debian. Canonical also recently started providing official images of Ubuntu server and desktop for the Pi. The Pi 4B offers a variety of different pinouts and connectivity options. It carries the standard external device connections such as USB, HDMI, and network interfaces. The Pi 4B also sports a 40 pin GPIO header that provides access to various communications protocols (I2C, SPI, UART). The price of the Raspberry Pi 4B ranges from $35 to $75.

Verdict

After considering the four options discussed, our team opted to move forward with the Raspberry Pi 4B. Table 1 provides a summary of how well each SBC aligned with the criteria our team set.

Table 1 – Board evaluation with the leftmost criteria being most important and rightmost being least important.

	Community Support	Connectivity / Pinout	ROS Compatibility	Cost
Raspberry Pi 4B	High	High	High	Cheap
Jetson Nano	Med	High	High	Mid-range
LattePanda	Low	Med	High	Expensive
BeagleBone	Med	Low	Med	Mid-range

The most important reason for our decision was the overwhelming size of the community for the Pi due to its popularity among roboticists. When compared to the other three SBCs, the Pi was an obvious choice. The BeagleBone does not have as large a community as the Pi and none of the variants are as versatile in terms of pinout or connectivity. The LattePanda is an interesting concept, and its main processor is potentially more powerful/faster than the Pi's depending on the variant; however, the smaller support community and uncertainties associated with the board's unusual configuration makes it difficult to choose. Its Arduino coprocessor is also based on the Leonardo, which is lacking in pinouts and connectivity. The Jetson Nano was a strong contender but considering that the chosen compute module does not perform any sort of heavy-duty AI work, it did not make sense to spend more on what is essentially a Pi 3B with a NVidia GPU when the GPU would not be used. Additionally, the support community is not as large as that of the Pi's.

3.2.2 Arduino

In the robotics community, Arduino microcontrollers are ubiquitous for their many variants and different applications. Boards such as the UNO and Leonardo are considered "starter" boards

while the Mega and Due are examples of boards for more advanced uses. Features common to most, if not all Arduino boards include digital I/O, analog I/O, and communications interfaces (I2C, SPI, UART). From the official Arduino lineup, our team chose to use the Due since it has the most versatile and diverse pinout of any official Arduino board. It also boasts the fastest processor the official Arduino lineup has to offer at 84 MHz. Before making this decision, we recognized that there are unofficial Arduino variants that could offer better performance than the Due. However, we felt that the Due would reduce potential complications in the future. Based on future issues that will be further discussed in Section 4.4, we ultimately decided to switch to a Teensy 4.1

3.2.3 Brushless Motor Controller

When searching for motor controllers, our team needed to confirm that the selected motor controllers had smart features such as built-in PID position control. In the market, we found a few different motor controllers that provided these features. The first was the RoboClaw motor controller by BasicMicro. The feature set of the RoboClaw fit all of our team's requirements, however, all their current motor controllers are designed for brushed motors only. BasicMicro has revealed that they are working on a brushless variant of their motor controller, but it would not make it to market in time to be used for this project. Next, we came across a brushless motor controller by Embention designed for use with drones called the Veronte GIM3. Once again, the GIM3 provided the features that our team was looking for as well as much more. Apart from integrated PID control, the GIM3 has IP67 waterproofing, embedded data recording, telemetry on motor health, and regenerative braking. While these features are attractive, they are unnecessary for our project and would not be used. The additional features also drive the cost of the GIM3 to a price our team could not afford: 550 EUR per unit. Finally, our team considered the ODrive by ODriveRobotics. The ODrive is an open-source brushless motor controller capable of running two motors at once. In line with our requirements, it provides onboard PID control as part of the package and offers a variety of running modes. To enable onboard PID control, the motors and their corresponding encoders connect directly to the ODrive at a designated header. From there, the specifications of the encoders being used, such as pulses per revolution (PPR) are programmed into the ODrive. The ODrive also supports a variety of protocols for communication with a microcontroller or other external controllers including CAN, I2C, and UART. Given the relatively low cost ($149) and the support it has in terms of both community and documentation, we decided to move forward with the ODrive.

3.2.4 Supporting Electronics

With all the core electronic components chosen, our team worked to find the supporting electronics for power delivery. At the basic level, this consisted of a power supply, a circuit breaker, a power distribution block (PDU), and DC-to-DC converters. Additionally, we needed a CAN breakout board to enable the CAN functionality on our Arduino Due.

When looking for power supplies, the primary consideration was the amount of current we expected to draw. Since the ODrive, Arduino, and Raspberry Pi all draw a small amount of current, in the order of milliamps, the motors were the deciding factor on the amount of current and voltage the power supply would need to output. To reduce cost since power supplies tend to be expensive, we assumed that the max current draw would be in the scenario where the robot is transitioning to standing on two legs and almost exclusively using 4 of the 8 motors. With the motor operating natively at 24V, and the motor's 9 amps continuous current draw rating in mind, we chose the 24V 40A RSP-1000-24 power supply from Jameco Electronics.

After selecting the power supply, choosing the rest of the power delivery electronic components became trivial. Power distribution blocks are easily found at large electronics suppliers such as McMaster-Carr, which is where we purchased ours from (https://www.mcmaster.com/9290T11/). Circuit breakers and DC-to-DC converters are common enough that they can be found at both electronics suppliers and general online retail websites. In our case, we purchased these components from Amazon (Surface Mount 60A Circuit Breaker, 24V to 12V DC Converter, 24V to 5V DC Converter). Choosing a circuit breaker is straightforward as the only specification to look for is the current draw at which the breaker pops. Since we were expecting to draw a maximum of around 40A, our circuit breaker was rated for 60A to deal with any potential spikes in current draw that could be caused by a short circuit for example. For DC-to-DC converters, the required specification is defined by the voltage of the power supply (24V) and the input voltage range of the Due (7-12V), Raspi (5V), and servos at the ankle joints (8.4V). Since the current draw of the Due and Raspi are trivial, just about any DC converter would work such as the ones we purchased from Amazon. The servos, however, draw around 2A each at their stall torque. We found DC converters capable of supplying this amount of current on Aliexpress (24V to 8.4V DC Converter).

Lastly, we needed to get a CAN transceiver to enable the CAN ports on the Due. Although the Due already has CAN TX and RX pins broken out to a header, these pins do not directly translate to the high and low signals CAN uses for carrying data along a wire. As such, a CAN transceiver is required as an interface between the differential signal carried on the wire and the CAN port on the Due. The ODrive has a built-in CAN transceiver so an external breakout board is not required on the ODrive's side. There are a few CAN transceiver boards commonly used in the Arduino community that breakout different transceiver chips, with the most popular one being the MCP2515 chip. However, in our case we chose a breakout board by CopperHill Technologies based on the SN65HVD230 chip since its operating voltage is the same as the Due (3.3V).

3.2.5 Wiring

When wiring together all the electronic components, we used four different types of connectors. Ring terminals of different sizes were used for all power related connections since these components use screw terminals. Dupont cables were used as data wires between the Raspi, Due, and ODrives as well as feeding power to the Raspi. Molex connectors were used exclusively for connecting the encoders to the ODrives and the Anderson connectors were used to connect the motors to the ODrives. Figure 10 shows the complete wiring diagram for the robot including all electronic components.

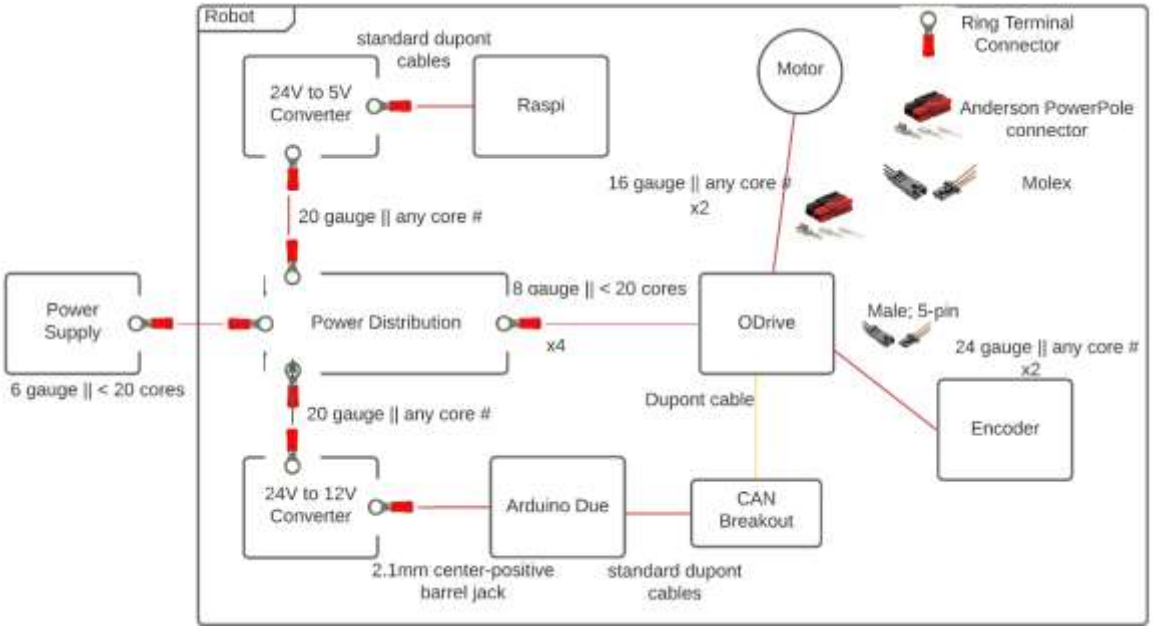


Figure 10: Wiring diagram for robot

3.3 Robot Analysis

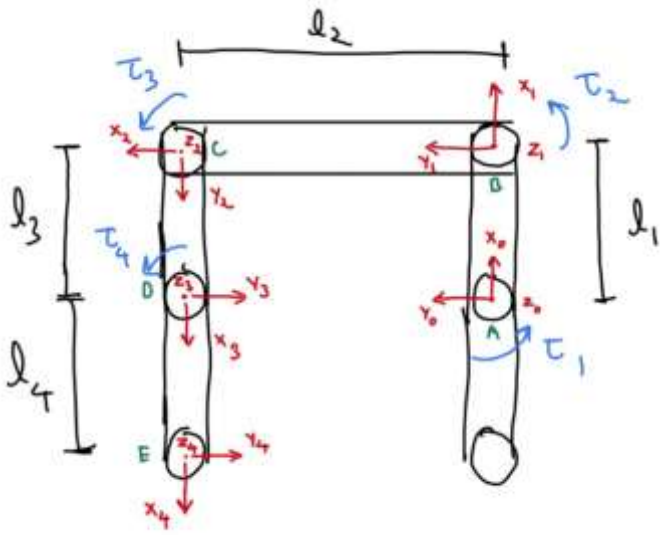


Figure 11: Initial torque analysis notation

To evaluate the feasibility of the Solo8 undergoing stance transition, we conducted a preliminary analysis at the beginning of the project. For this, we did a simple torque analysis by considering the robot as an open chain manipulator (attached in Appendix B). This simplification is valid in our case because we are calculating the torques when the robot undergoes stance transition, during which it essentially acts as an open chain manipulator. We calculated the worst-case torque requirements for when the robot would lift its front limbs in order to undergo stance transition. The resultant torques, presented at the end of Appendix B, are the torque requirements for the end effectors on each body link. Torque requirements for the motors were calculated by accounting for the gear reduction. Unfortunately, we misinterpreted the gear reduction to be 81:1 back in A term. In reality, it is a much smaller value of 9:1. After accounting for the gear reduction, albeit the wrong value, the torques translate to the following based on notation from Figure 11:

$$\tau_1 = 0.0291Nm$$
$$\tau_2 = 0.1375Nm$$
$$\tau_3 = 0.005Nm$$
$$\tau_4 = 0Nm$$

We compared these torques to the motors' stall torque and ensured they were within the stall torque limit of 0.34 Nm (this value was discovered after contacting a sales representative from T-Motors). After confirmation, we decided to use these motors for our modified robot. As mentioned, our calculations were erroneous due to the wrong gear reduction, so this result was invalid. We later rectified and overcame this mistake, and this is discussed in Section 4.2.

3.4 Mechanical Modifications

While the Solo8 provided a strong starting point, certain modifications to the Solo8's design were required to achieve the project's goals. Firstly, in order to achieve stable bipedal standing, the robot's bipedal support polygon had to be increased. We decided to do this by creating a flat foot on the rear legs. The robot's body structure also had to be modified to support OTS electronics. This section details our decisions regarding the robot's fabrication and part acquisitions.

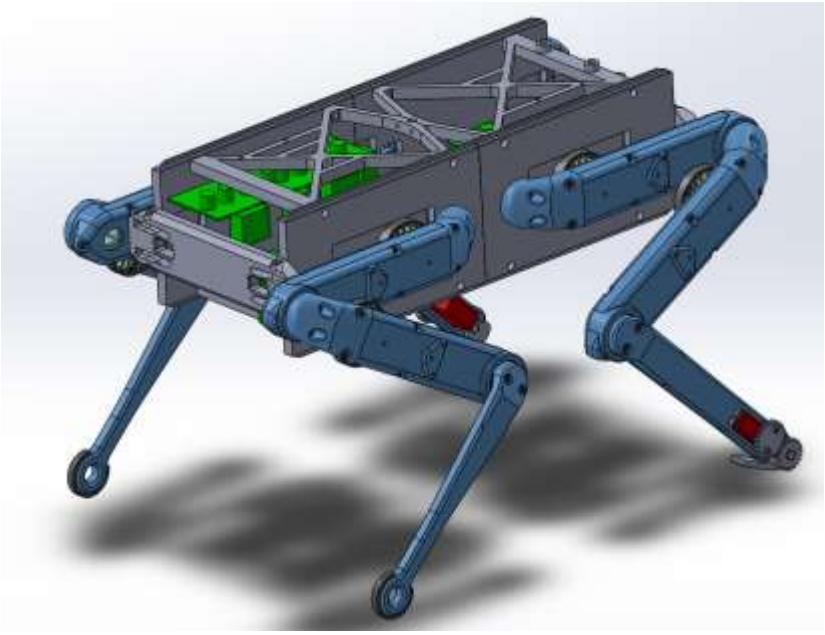


Figure 12: Modified Solo8 CAD model

3.4.1 Flat Foot

The original Solo8 was designed with conventional point feet. While point feet are effective for quadrupedal walking, they are very ineffective for bipedal locomotion because they provide a small support polygon. The team decided that the Solo8's rear feet required a redesign to increase the robot's bipedal support polygon and improve bipedal stability.

At the beginning of the design process, the team created a list of requirements that outlined the foot design's basic operations and brainstormed potential solutions. The full list of requirements can be found in Appendix C. The most important requirement that we identified was that the foot needed to support both quadrupedal and bipedal locomotion, while also increasing the robot's bipedal support polygon. To be effective in both locomotion types, the team concluded that the rear feet needed to convert between a point and flat foot. This way the robot would be capable of accomplishing quad-to-bi stance transition.

Using these requirements as a starting point we sketched different preliminary designs that involved a flat foot rotating around the Solo8's original point feet. Early in the design process, we decided that the flat foot should be actuated by a servo because it affords the greatest locomotion versatility. Unlike a servo, a latch or a spring-loaded system would only allow the flat foot to be deployed once and it would not be able to convert back to a point foot for quadrupedal locomotion. This would limit the Solo8 to a single stance conversion, whereas a servo-actuated foot affords continuous multi-modalism. Preliminary designs considered connecting the servo and flat foot with a four-bar linkage or a gear drive as shown in Figure 13. We ultimately decided to pursue a gear drive for a few reasons. Firstly, the design is more compact. More importantly, the gear drive provides a greater range of motion and allows us to keep the foot flat parallel to the ground as the angle of the robot's leg changes.

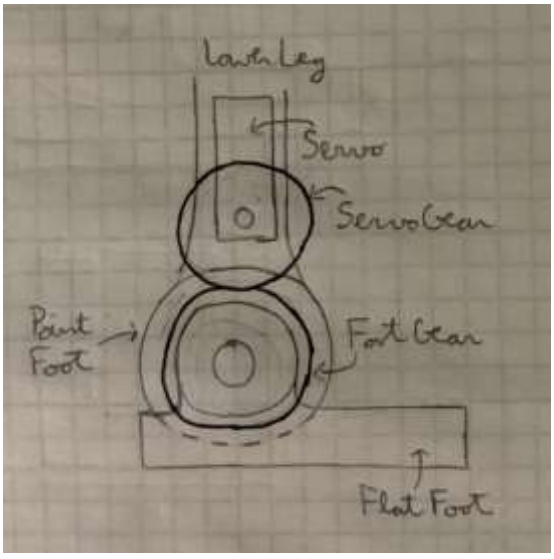


Figure 13: Four bar linkage design (left) and gear drive design (right)

The flat foot went through three major design iterations: a flat foot with a heel, the dog-inspired foot, and the J-foot. Figure 14 shows CAD models for the three different designs. The J-foot design, named after its resemblance to a backward J, was determined to be best suited for our application because it allows for the easiest point-to-flat foot conversion. For the first two designs, the flat foot sits beneath the point foot. Consequently, point-to-flat conversion can only occur if the robot's rear leg is elevated off the ground. This would not afford a smooth transition between quadruped and biped mode. The J-foot overcomes this flaw with its co-radial design. The flat foot's heel and lower leg's point foot are co-radial so that the flat foot can rotate while the point foot remains in contact with the ground. This provides a smooth point-to-flat conversion that does not interfere with the robot's stance transition or either locomotion type.

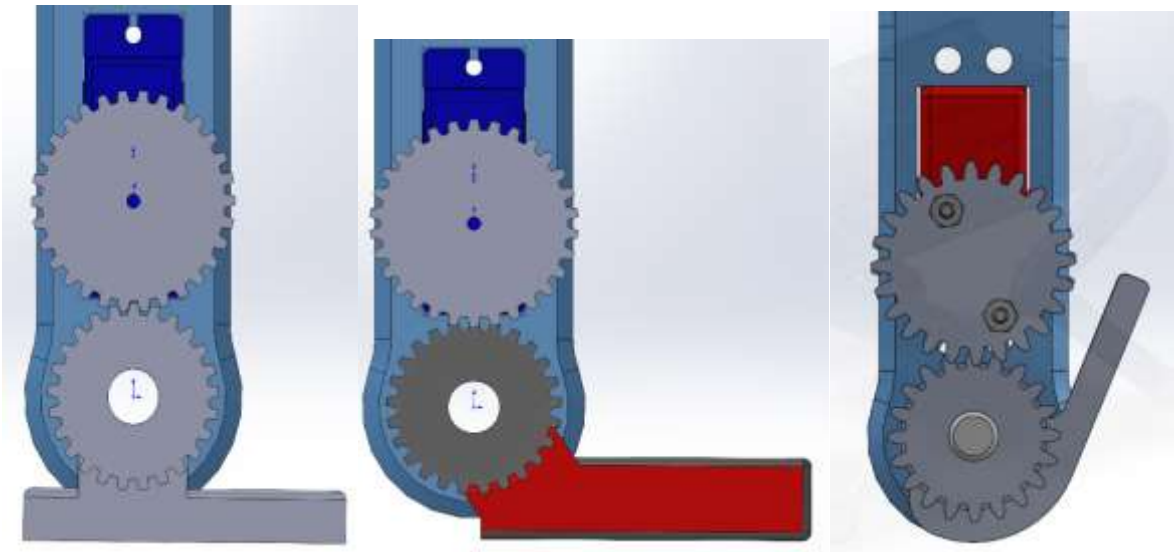


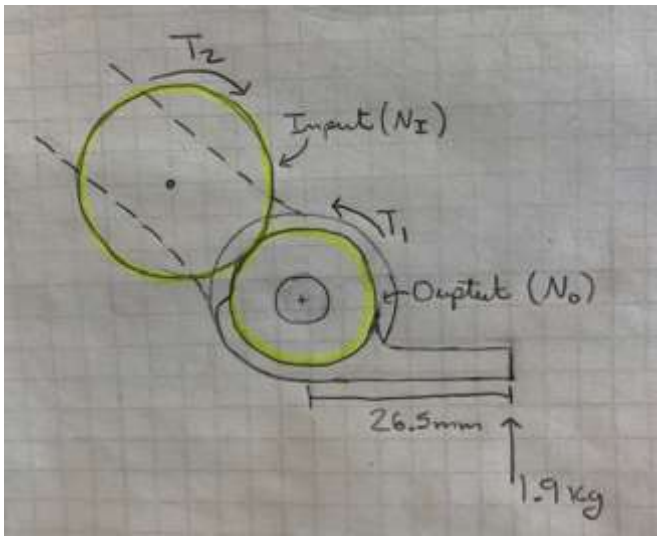
Figure 14: Flat foot with heal (left), dog foot (center), J-foot (right)

The foot mechanism was prototyped with both the foot and 20-tooth gear as separate parts and as one component. During the prototyping phase, we realized that fastening the foot and gear with screws added unnecessary complexity to the assembly process. After juxtaposing the two, we decided to incorporate the foot and gear into a single part because of how small the parts are. This also helped to slightly reduce the number of components in the assembly.

The flat foot provides 560mm^2 of surface contact and is actuated by a DS150CLHV micro servo with a 5:6 gear ratio (24-tooth servo gear and 20-tooth foot gear). An important factor in choosing the DS150CLHV micro servo over other models was that it is small enough to be embedded inside the lower leg. This meant that we did not have to make drastic modifications to the rear lower legs to support the foot design. Our decision to use the DS150CLHV micro servo was also informed by a simple torque analysis that we conducted on the foot and servo shaft. The analysis assumed a "worst case scenario" where the robot's full weight is on the front tip of the flat foot.

Table 2 – Foot torque analysis

Assumptions and Criteria:
1. All of the robot's weight is acting on the foot's front tip 2. The robot weighs 3.1kg 3. 5:6 gear ratio



$3.1kg * 9.81m/s^2 = 30.41N$

$T_1 = 30.41N * 0.0265m = .78Nm$

$T_2 = T_1 * (N_I/N_O) = 0.78Nm * (6/5) = 0.93Nm$

From this analysis, the torque acting on the servo was determined to be ~0.93Nm. Based on this result, we chose the DS150CLHV micro servo because it produces 150oz-in (1.06Nm), exceeding the worst-case-scenario torque requirement. Even though 150oz-in does not provide a large safety margin, the team felt that the "worst-case-scenario" that the analysis was based on was unlikely to occur. It is important to note that the torque analysis in Table 2 has some shortcomings, as it does not account for the torques exerted on the servo by the front limbs during stance transition.

Figure 15: Lower leg and foot prototype

3.4.2 Body Structure and Rear Legs

The Solo8's original body was designed to house custom boards and hardware that are much smaller than the PDU and ODrives that we used. As a result, we redesigned the Solo8's body to accommodate the robot's larger OTS hardware. Since the ODrives are fragile, expensive, and crucial to the robot's operation, we decided to place the four ODrives inside the body structure because it maximized their protection. The ODrives' placement is staggered between the top and bottom body braces to provide accessibility for wiring. To fit the ODrives in the body cavity, the side plates were widened from 26.5mm to 47.5mm. The top and bottom body braces were also modified by adding new mounting brackets for the ODrives, power converters, and Raspi. The ODrives and 24V/5V converter are screwed to the braces while the other electronics are zip tied. Since the PDU was the heaviest component, we placed it on the rear in an attempt to improve the robot's standing stability by using the PDU's weight to shift the robot's COM backwards and lower.

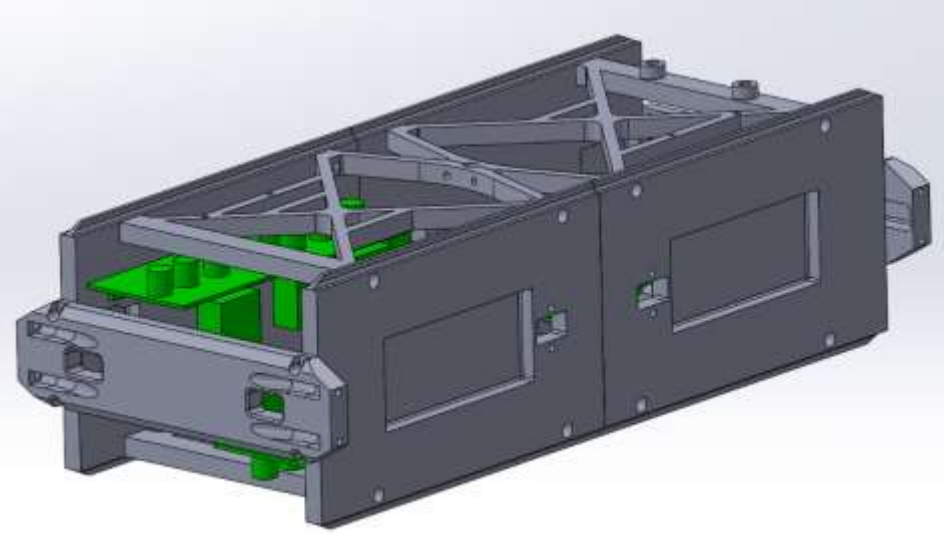


Figure 16: Modified body structure

The rear lower legs were also modified so that the micro servo could be embedded inside the leg structure. A recess was created at the bottom of the leg so the servo could be secured. The recess was made 1mm larger than the servo's width to provide some tolerancing. The rear lower legs were also made 12mm wider to improve the leg's structural strength. The added width provides additional material around the servo so that the leg's structural integrity is not compromised by the servo's recess.

3.4.3 Fabrication and Hardware Acquisition

Even though the ODRI GitHub repository "Open Robot Actuator Hardware" page has recommendations and instructions for part fabrication and acquisition, the team still made decisions regarding the sourcing and fabrication of certain parts. We decided to outsource the machined parts because we believed this would be most cost-effective, considering the numerous hours we would need to spend machining the parts in Washburn, not to mention the cost of machining errors. We got quotes from various machine shops and ordered the parts from the cheapest option, KVC Engineering. Additionally, someone on the ODRI forum recommended KVC and affirmed that KVC could machine the parts to the required tolerances. We faced a similar decision with the robot's codewheels and encoders, where we had the option between purchasing the parts separately or buying an encoder kit which included the codewheel. The encoder kit cost less; however, the codewheel from the kit had an aluminum flange that needed to be lathed off for the codewheel to be compatible with the actuator module. Once again, we reached out to the ODRI forum and learned that others had issues lathing the aluminum flange because fixturing the codewheel to the lathe was difficult. As a result, people regularly damaged or completely broke the codewheel while machining. This information helped us make the decision to purchase custom codewheels from PWB Encoders GmbH.

The Solo8 developers specify that the body parts and actuator module shells should be printed with PC-ABS on a FDM printer and that the 3D printed pulleys and encoder mounts should be printed with the Accura Xtreme filament on an SLA printer. We printed the body parts and actuator module shells in PLA on a Prusa MK3s+ printer. We opted for PLA because it was readily available to us and we thought it would provide similar performance to PC-ABS. While most of the robot was printed in-house, the team decided to outsource the 3D printed pulleys and encoder mounts. We outsourced these parts because we did not have access to an SLA printer and the parts' small, detailed features could not be replicated on the MK3s+. We conducted several test prints on the MK3s+ and we could not print the parts to a satisfactory level of detail. Even though outsourcing these parts came at a cost of $460, it allowed us to properly print them on an SLA printer. This gave us confidence that the parts would function properly in the actuator module.

4 Implementation

While we intended to implement the design decisions discussed in Section 3, practical constraints and unforeseen challenges forced us to change course in certain areas. This section details our actual implementations and how they deviated from our initial decisions. After overcoming numerous challenges, the team successfully validated our software stack via RL experiments, built and tested the robot, and developed a quadrupedal walking gait and stance transition trajectory in simulation.

4.1 Reinforcement Learning Experiments

The development of the OpenAI gym wrapper was completed before the robot was fully built. As such, we validated our RL pipeline by first solving known RL problems and then teaching the Solo8 to stand quadrupedally.

4.1.1 Inverted Pendulum & Lessons Learnt

In Section 3.1, we discussed how our pipeline was built around running RL experiments as efficiently as possible. Supporting features such as automatic deployment, hyperparameter tuning, and OTS RL implementations removed much of the boilerplate work involved in setting up the RL experiments.

To debug the pipeline, we first trained an agent to solve a RL problem with a known solution. For our test problem, we chose the **Pendulum-v0** environment that is included in the base OpenAI gym package. In **Pendulum-v0**, the agent's goal is to apply a rotational force to a pendulum to keep it upright. When the pendulum is fully upright, the agent is given a reward of 1 and 0 otherwise. Screenshots of this environment can be seen in Figure 17.

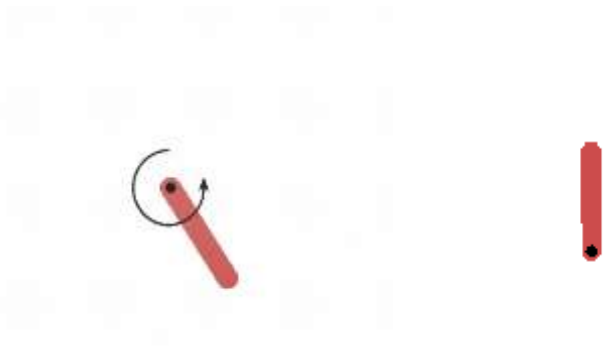


Figure 17. Open AI Gym's **Pendulum-v0** environment. Here, the agent applies torques to a pendulum with one rotational point. On the left, the agent's torque is being visualized by an arrow representing direction and magnitude. On the right is a properly trained agent. This agent is actually applying torques to balance the pendulum, but they are so minute that it is not visible in the screenshots.

The **Pendulum-v0** environment was specifically chosen because both the environment's states and the agent's actions are continuous. Recall from Section 2.4.2.1, that in RL, the agent is trying to learn a policy function π that outputs an action a_t given an environment state s_t. When running RL experiments with our robot, we expect the environment state to be the robot's sensor inputs and the agent's actions to either be motor torques or joint positions. Regardless, we know that both s_t and a_t are continuous. By choosing **Pendulum-v0**, we can be certain that the resolution of our state and action spaces between problems are the same—albeit the spaces associated with our robot will have more dimensions simply due to having more sensors and motors to control. Additionally, as **Pendulum-v0**'s default reward function is known to converge, a properly trained **Pendulum-v0** agent would imply that the entire pipeline works.

For the model training, we used the PPO2 implementation from stable-baselines. As stable-baselines is OpenAI Gym-compatible and known to work correctly, it transferred well to our robot's RL experiments once the pipeline was validated. While the results for our pipeline validation experiments can be found in Section 5.1.1, we learned several important lessons that were implemented into the pipeline for our Solo experiments.

4.1.1.1 Normalized State & Action Spaces

When initially training the **Pendulum-v0** agent, we encountered severe issues with the PPO2 model diverging. Upon further investigation, it was due to the gradient updates ∇--defined in (8)—being too large. This is because PPO2 assumes that the state and action spaces are normal when being sampled. However, as the state space of **Pendulum-v0** is valued between 0 and 360—the current angle of the pendulum—this caused PPO2 to sample values greater than 1.

As per [43], this is a known shortage of PPO2 and the recommended solution is to ensure that the state and action spaces are normalized to be in $[-1, 1]$. As such, we modified our pipeline to dynamically normalize the state and action spaces. For uniformly-distributed values, such as the angle of the pendulum, this was done via a simple scaling operation:

$$\tilde{\mathbf{x}} = -2 * \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})} - 1 \tag{20}$$

As each dimension of $\mathbf{x}$ can have different domains, min and max—in fact, all operations—are applied element-wise in (20). Observe that this will map all values in $[\min(\mathrm{x}), \max(\mathrm{x})]$ to $\tilde{\mathbf{x}} \in [-1, 1]$. For nonuniformly distributed values, such as sensors tracking velocity, standardization was used:

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{21}$$

Note that just in (20), all operations in (21) are applied element-wise. In this case, $\boldsymbol{\mu}$ is a vector of averages, with the *i*th value of $\boldsymbol{\mu}$ corresponds to the average of the *i*th values of $\mathbf{x}$. Similarly, $\boldsymbol{\sigma}$ is a vector of $\mathbf{x}$'s standard deviations. In our implementation, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ were dynamically calculated as states are collected. All actions outputted from the networks were

valued in $[-1, 1]$ and were scaled back to their original ranges before being sent to the environment.

4.1.1.2 Environment Vectorization

Recall from Section 2.4.2.2, that PPO trains based off trajectories for the current iteration of the policy π. This means that trajectories cannot be reused during iterations. To increase our runtime training performance, we vectorized our environment to run on multiple CPU cores at the same time. In our experiments, our trajectory terminates after T timesteps; however, our vectorized environment wrapper can also handle non-standard trajectories with premature terminations. In practice, we found that vectorizing over 12 CPU cores yields a near 10x decrease in training time; heavily increasing the effectiveness of our training pipeline.

4.1.2 Solo 8 Quadrupedal Standing

Once we had our pipeline validated, we wanted to run some proof-of-concept experiments on the Solo8 model. We chose quadrupedal standing as a baseline task due to its intuitive nature and because it would allow us to focus on having PPO control the gym_solo environment. Unfortunately, our version of the Solo8 model was in heavy development, so we used ODRI's model of the unmodified Solo8 as a placeholder. Note that the only difference between the two models is that our final version of the modified Solo8 included an ankle joint. However, the ankle joint is intended to be inactive during quadrupedal mode; therefore, excluding it has minimal consequences.

When designing our environment, we wanted to mimic realistic conditions as much as possible. We only allowed the agent to observe its motor encoder position and velocity values. As in traditional robot control, the agent outputs either motor torque or joint position values. Note that since our pipeline automatically normalizes actions, as mentioned in Section 4.1.1.1, switching the meaning of the agent's output has minimal effects on its training.

With the environment properly configured, we focused our attention to shaping the reward function. As mentioned in Section 2.4.2.2, PPO's goal is to choose the best robot control to optimize the given reward. However, these rewards need to be designed very strategically. For example, simply rewarding the agent 1 for quadrupedal standing and 0 otherwise, similar to the inverted pendulum in Section 4.1.1, will not work. In the inverted pendulum, the agent will often randomly hit the "goal" state, so it can learn which actions most often yield that behavior. However, for more complicated tasks such as quadrupedal standing, randomly achieving the goal state is nearly impossible. Under the simple reward function, this would mean that the agent is essentially getting no meaningful feedback from the environment—making any form of learning impossible. Instead, the bulk of the experimentation is done in "shaping" the reward function; specifically, determining the best way to incrementally reward the agent to the end goal. Our specific variants and results are explored in Section 5.1.

4.2 Revised Robot Analysis & Build

As discussed in Section 3.3, our initial analysis erroneously considered the gear reduction to be 81:1 instead of 9:1. While fixing this error, we also realized that we never accounted for the fact that in a 2D representation of the robot, two actuators are supplying the required torques. Thus, the actuator module torque requirements in Appendix B had to be divided by two. After rectifying these mistakes, we got the following torque requirements for the motors.

$$\tau_1 = 0.1311Nm$$
$$\tau_2 = 0.6189Nm$$
$$\tau_3 = 0.0261Nm$$
$$\tau_4 = 0Nm$$

We observed that τ_2 was almost twice the stall torque of the motor. Based on these updated torque values, we realized that our robot could not perform every case of transitioning to a bipedal stance. We had to take a conservative approach for finding a transitioning trajectory that fell under our torque constraints.

We also updated this analysis to account for the ankle joints. Instead of four joints, we now had five joints to consider. The updated torque analysis notation diagram is show in Figure 18.

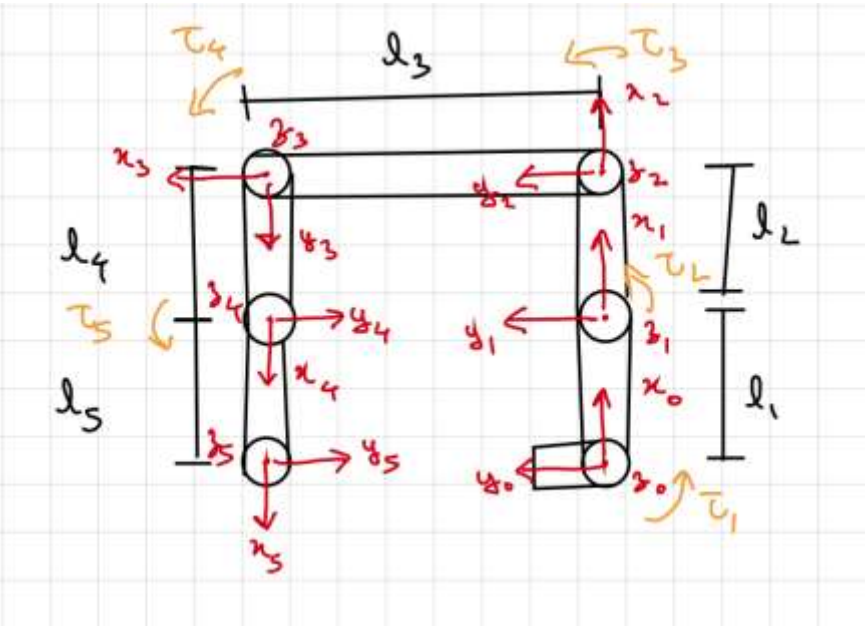


Figure 18: Updated torque analysis notation

We created a MATLAB script to calculate the torque requirements based on a robot's stance. Using this, we manually found a stance and the corresponding torque values that fulfilled our constraints. This is elucidated more in Section 4.10.

The robot was built following the assembly instructions found on the ODRI GitHub repository "Open Robot Actuator Hardware" page. For testing and documentation purposes, each actuator module was labeled 1 through 8. The Figure below shows how the modules were labeled.

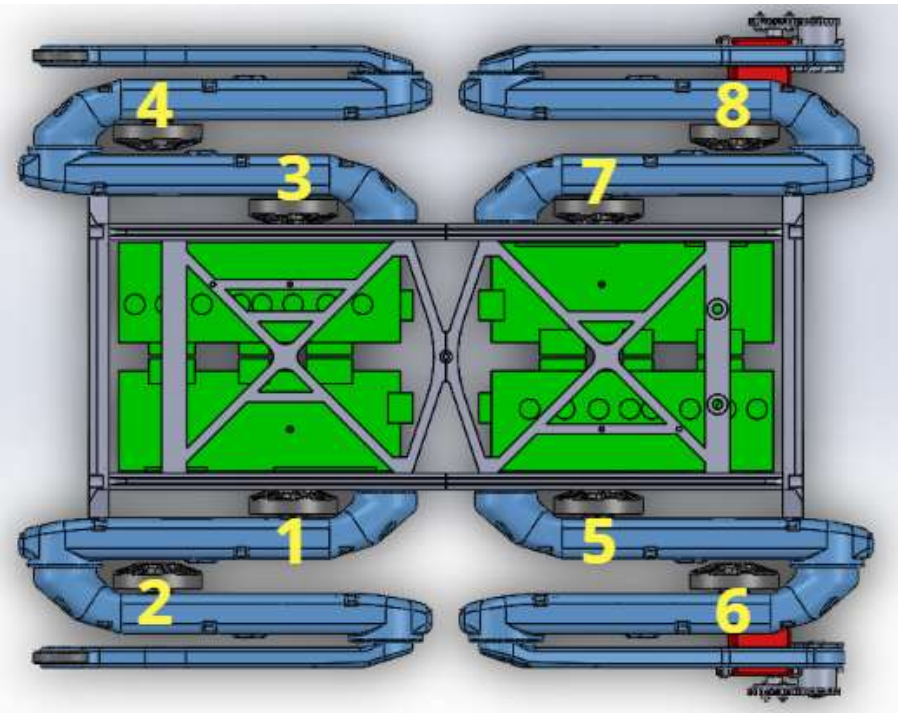


Figure 19: Module labels

Module 1: Front Left Hip Module
Module 2: Front Left Upper Leg Module
Module 3: Front Right Hip Module
Module 4: Front Right Upper Leg Module
Module 5: Back Left Hip Module
Module 6: Back Left Upper Leg Module
Module 7: Back Right Hip Module
Module 8: Back Right Upper Leg Module

The actuator modules were built with the exact same components as the original Solo8 except for one part: the 6mm wide, 201 tooth AT3 GEN III Synchroflex® Timing Belts in the dual stage timing belt transmission. When we were purchasing parts for the actuator modules there was a global shortage of 6mm wide, 201 tooth AT3 GEN III Synchroflex® Timing Belts and we could not source them. Instead, we purchased "standard" AT3 Synchroflex® Timing Belts. The polyurethane material in the standard belts is not as tough as the GEN III version but the Solo8 developers told us on the ODRI forum that it should function the same. For all the

parts and materials used to build the robot, a comprehensive bill of materials can be found in Appendix D.

When assembling the actuator modules, we sometimes encountered clearance issues between the codewheels and encoders. This was caused by the codewheel sitting too high on the encoder mount and resulted in the codewheel's upper surface rubbing against the encoder. If the misalignment was not corrected, the codewheel would get scratched and stop functioning properly. We encountered this issue on actuator modules 1, 2, and 7. We solve the issue by removing the brass ring that acts as a spacer between the aluminum pulley and motor. This lowered the codewheel enough that it correctly aligned with the encoder.

Figure 20: Encoder assembly (left) and brass right (right)

For the foot mechanism, the flat feet and servo gears were 3D printed with PLA. Figure 21 shows an exploded view of the foot mechanism components. Due to 3D printer tolerances, the shaft hole in the lower leg and foot were made slightly larger than the ¼" shaft diameter. The servo gear is screwed to a round servo arm that is fastened to the servo shaft. The ¼" shaft was cut to size and using a Dremel, rings were cut into the shaft ends for the retaining rings.

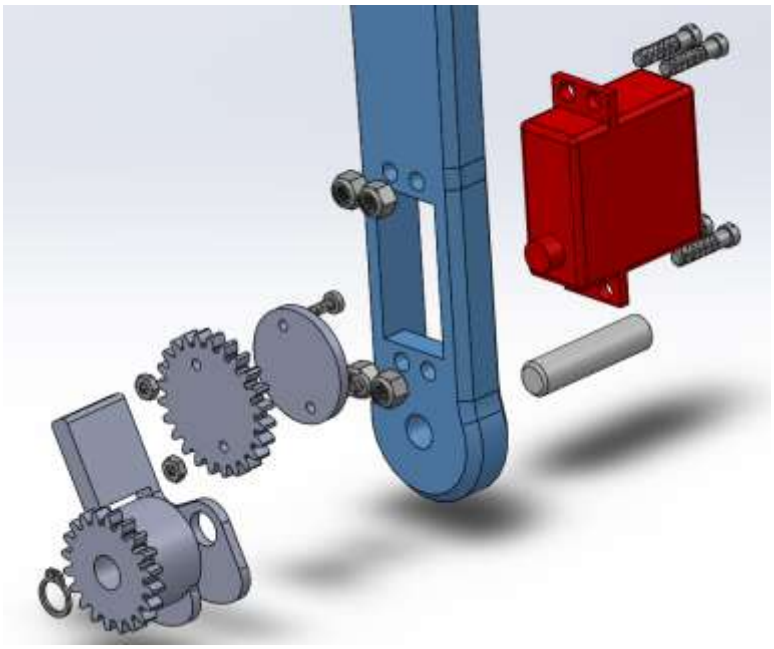


Figure 21: Flat foot assembly exploded view

4.3 ODrive Tuning

ODrives are very powerful motor controllers, but to harness their full potential, they need to be tuned for the specific motors they are driving. Thankfully, ODrives are relatively well documented and have an amazing support community. While referencing the documentation, we created a basic configuration for the ODrives. We tested the basic configuration by moving the motor to a specific set point. During testing we observed an interesting error where the motors only moved after immediate calibration, but not after a reboot. After discussing with the ODrive community and diving deeper into the documentation, we realized that noise on the encoder's index channel could be causing the issue. To rectify the problem, we soldered a 47 nF capacitor between the index pin (Z) and ground (GND) on both ODrive axes. After installing the capacitors, our tests were successful both after immediate calibration and reboot. This validated that the ODrives, motors, and encoders were functioning properly.

Figure 22: 47nF capacitors soldered to both ODrive axes

During additional testing, we observed that the motor would not move to its desired setpoints. This occurred because the ODrives were not yet tuned for our motors. We followed the ODrive tuning instructions to tune our system. It is important to note that the ODrive controller does not implement a regular PID loop. Instead, they use a cascading Position, Velocity and Current Controller. After tuning, we got the following PID tuning graph in Figure 23 for all our actuator modules.

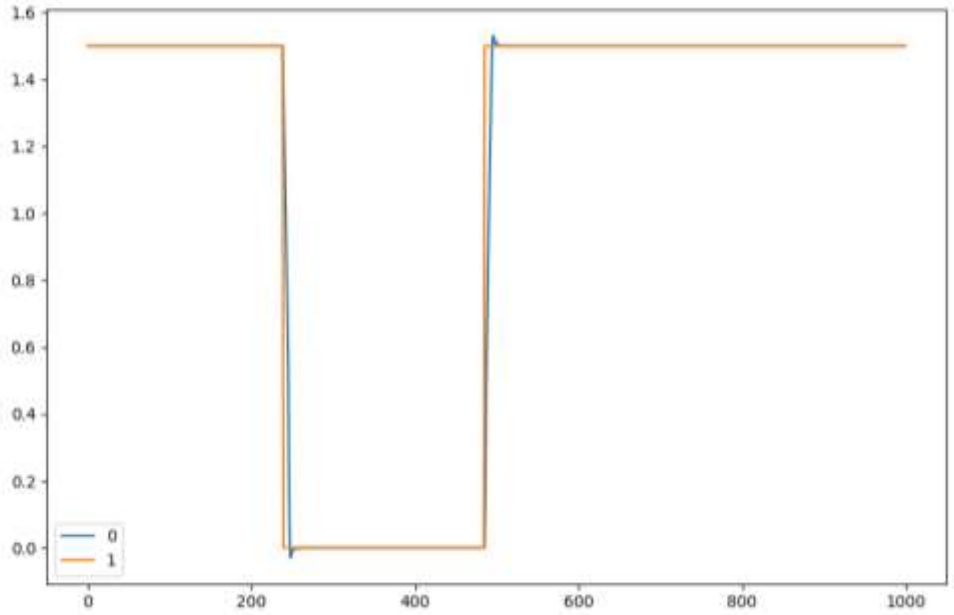


Figure 23: PID tuning graph. The y-axis corresponds to the revolution of the motor and x-axis corresponds to the time.

Another problem we encountered was the need for index calibration every time we booted-up the ODrives. Currently, ODrives require that incremental index encoders search for the index pulse at bootup to calibrate. The only way to avoid this was by implementing a custom firmware on the ODrives. Given our time constraints, we decided against this option because of our unfamiliarity with the ODrive stack. Instead, we decided to implement a mechanical workaround for the index calibration. We fixed the direction of the index calibration to move the legs away from the center of the body during it. We then assembled the legs in the robot home position at the index position of the encoders. This is explained more in Section 4.6. The robot legs are moved a little bit towards the center of the robot from the home position. When the robot is powered on from this position, the robot legs move into the home position (Figure 27). After tuning and calibration were completed, our actuator modules were ready to be integrated with the software stack.

4.4 Pivot from Arduino Due to Teensy 4.1

As was mentioned in Section 3.2.2, our team's original plan was to connect the ODrives to the Due through CAN bus. However, while we were developing code to implement the CAN protocol, issues arose that prevented us from successfully establishing two-way communication between the ODrives and Due. For reasons we were unable to figure out, the Due would consistently receive heartbeat messages from the ODrive over CAN but when the Due sent a message to an ODrive, the ODrive would not receive the message. For about two weeks, we rigorously tested the validity of the code, debugged the signals on the wire, and discussed potential solutions with the ODrive community. Ultimately, we were unable to resolve the issue and chose to pivot instead of spending more time trying to solve the problem. There were two other communication protocols to choose from that the ODrive supported: I2C and UART. We chose to use UART because it had better documentation and it was the safer bet since to enable I2C, we would have to physically modify the ODrives by de-soldering the CAN transceiver from the PCB. After some quick testing, the UART protocol proved to work perfectly in code on both hardware and software serial ports.

Despite its success in code implementation, the switch to serial UART produced a new issue on the hardware end. Our robot uses four ODrives to drive all the motors and the Arduino Due has four hardware serial ports. One of the ports is pre-configured to work with a console on a host computer connected via USB. Another port is consumed by the serial connection to the Raspberry Pi. This leaves just two hardware serial ports available, which is not enough to control the robot's four ODrives because each ODrive needs a dedicated serial port connection. After researching potential solutions, our team came up with 4 options:

1. Completely cut out the Due and use four of the Raspberry Pi's six hardware serial ports to run the ODrives directly. This would also require that the Raspberry Pi run a control loop for the motors separate from the existing ROS nodes.

2. Get a second Due, such that each Due controls two ODrives, and synchronize their control loops over I2C.
3. Continue using a single Due and add more serial ports using a software serial library. A software serial library creates additional serial ports by bit-banging digital pins.
4. Swap out the Due with a Teensy 4.1 that has eight hardware serial ports.

We ultimately chose to move forward with Option 4: Swap out the Due with a Teensy. We chose this route due to concerns we had with the other three options and because of the simplicity of transitioning to the Teensy. With Option 1, our concern was that since ROS is not a real-time system, there could be timing issues with the movements we want to send to the ODrives. Option 2 was a feasible option, but when compared to Option 4, Option 2 becomes inefficient, both in terms of the number of electronic components needed and code that needs to be written. Option 3 was arguably the closest contender with Option 4, but its largest drawback was the fact that software serial ports have a history of instability and could cause issues down the line. With these thoughts in mind, Option 4 was the best choice to move forward with because there were no perceptible drawbacks. Transitioning over to the Teensy proved to be a trivial task, and the Teensy provided a boost in speed when comparing its 600Mhz processor to the Due's 84Mhz processor.

4.5 Updated Wiring

A consequence of switching the Due with the Teensy 4.1 was that our wiring needed to be changed to accommodate the Teensy. Both the Teensy and the Due have an operating voltage of 3.3V, however their input voltage requirements are different. The Due has an input range of 7-12V while the Teensy requires a 5V input source. Luckily, the Raspberry Pi also requires a 5V input, so we were able to use the same DC converter for both the Raspberry Pi and the Teensy. This also allowed us to completely remove the 24V to 12V DC converter. The only other change we needed to make was wiring between the Teensy and the ODrives. Since we were moving from CAN to serial UART, we were able to cut out the CAN transceiver breakout board and wire the ODrives directly to the Teensy's hardware serial ports. Figure 24 shows an updated version of the wiring diagram after the change.

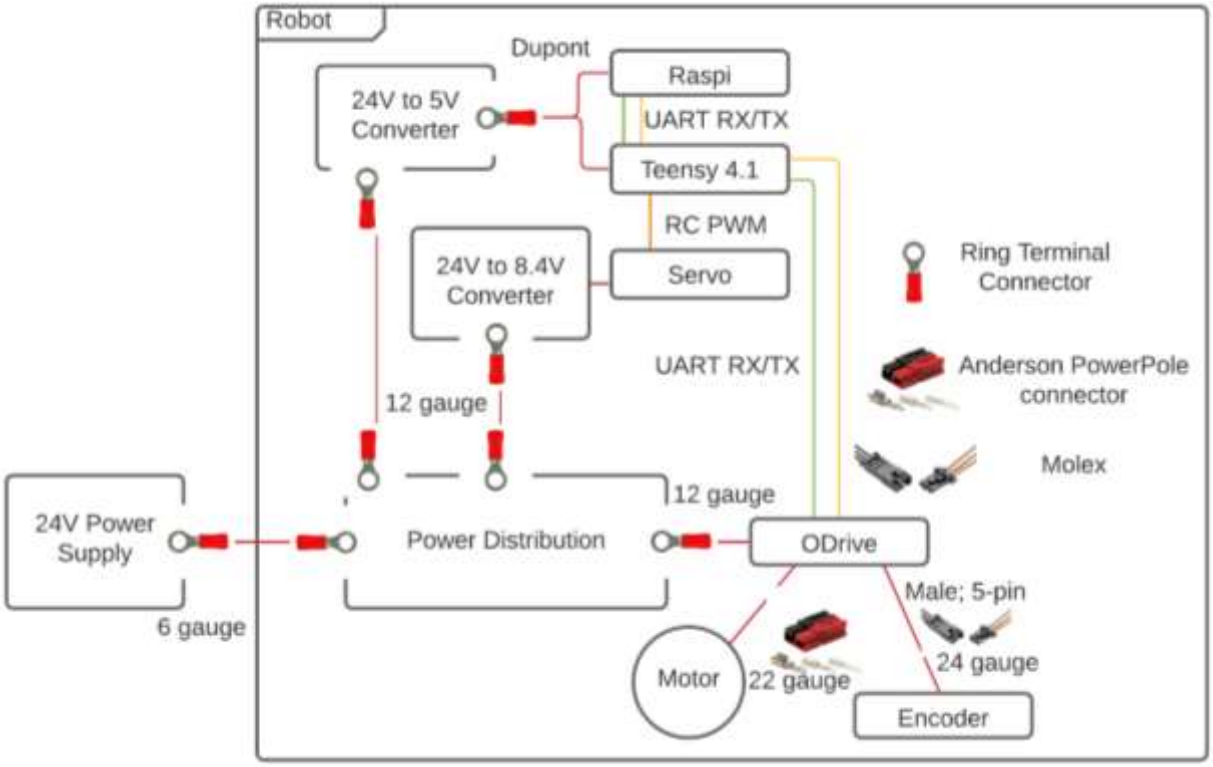


Figure 24: Wiring diagram with Teensy replacement

4.6 Quadrupedal Home Position

The Solo8's quadrupedal home position is the position where all the legs are straight and perpendicular to the ground. The home position is important because it acts as a zeroing point from which other leg configurations can be set. However, setting the home position was complicated by the actuator modules' design and the incremental encoders. As described in Section 2.3.1, each actuator module has an output pulley to which the subsequent appendage (actuator module or lower leg) is attached too. As shown in Figure 25, when the robot is in its home position, the output pulley's horizontal axis should be aligned with the hip module's horizontal axis. For the upper leg module, the output pulley's horizontal axis should be perpendicular to the module's horizontal axis (Figure 26).

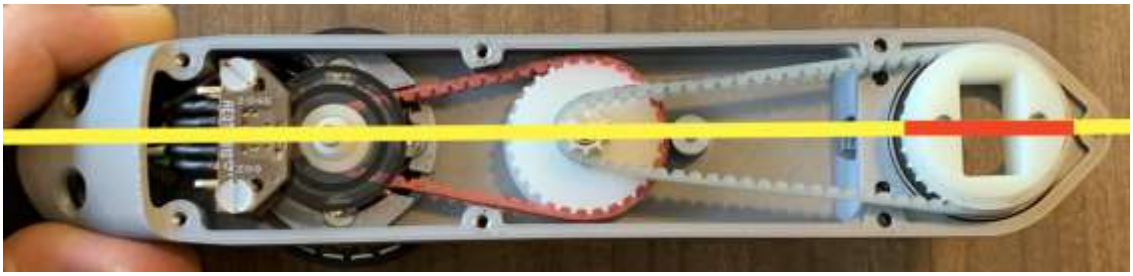


Figure 25: Hip module's horizontal axis (yellow line) aligned with the pulley's horizontal axis (red line)

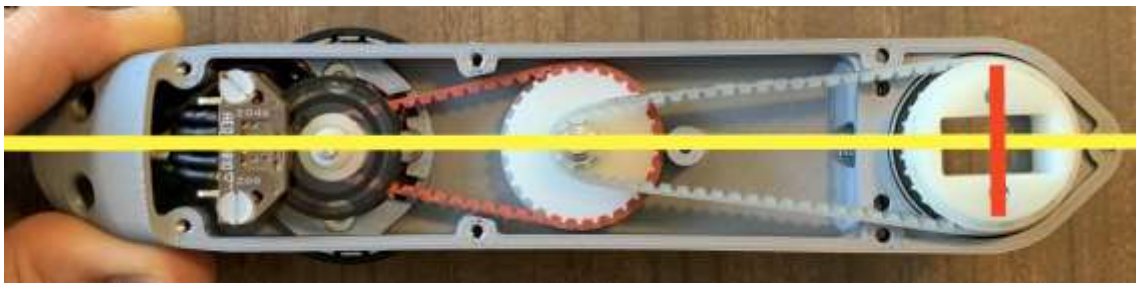


Figure 26: Upper leg module's horizontal axis (yellow line) perpendicular to the pulley's horizontal axis (red line)

By nature of the 9:1 dual-stage timing belt transmission, one rotation of the pulley (equivalent to one rotation of the connected appendage) is equal to nine rotations of the motor shaft (i.e. nine rotations of the codewheel since the codewheel is directly attached to the motor shaft). When the robot is powered on, the motors automatically move to the codewheels' index position. However, the codewheel's index position does not necessary correspond with the robot's home position because one rotation of the codewheel is $1/9^{th}$ a rotation of an appendage. Based on how the output pulleys and 201 tooth timing belts are assembled with the center pulleys, the connected appendages can theoretically be in nine different positions. Therefore, for the codewheel's index position to correspond with the robot's quadrupedal home position, the output pulleys need to be assembled in the correct orientation according to the codewheel's positioning. Therefore, the output pulleys had to be positioned so that the quadrupedal home position corresponded with the codewheel's index position. We did this by assembling the output pulleys and modules when the module's motor and encoder were powered on and in the codewheel's index position. Since the motor was powered on, it provided enough resistance so that the codewheel stayed stationary while we connected the output pulley to the timing belt transmission. The output pulleys were assembled along the axis orientations described before.

Unfortunately, due to physical constraints in the timing belt transmission we could not position the output pulleys so that their axes perfectly aligned with the modules' axes. Instead, we aligned the axes as close as possible and accounted for the unique angle offset of each leg in the Arduino. To do this we powered the robot on and let the legs calibrate to the codewheels' index positions. Next, using the Raspi we moved the modules until the connected appendage was vertical. Each modules' offset angle was put into the Arduino code and the programmed quadrupedal home position was set. From then on, every time a home position command was sent from the Raspi, the robot went into the following position.

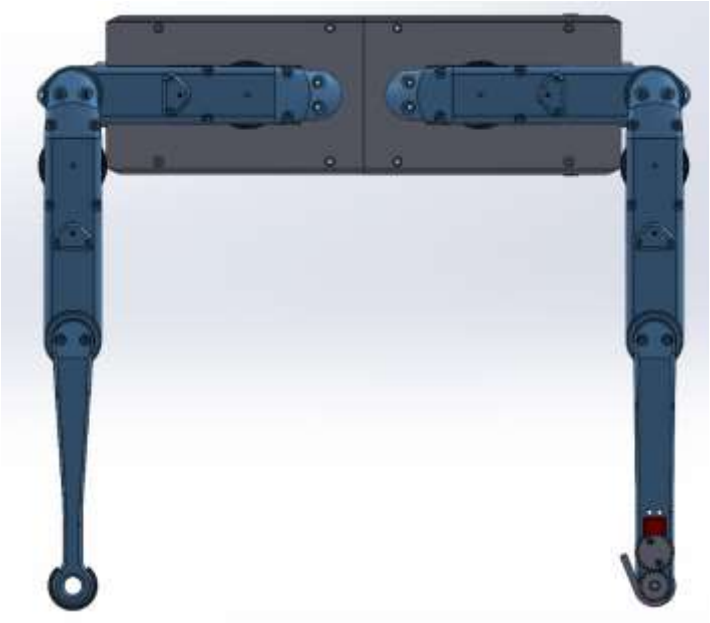


Figure 27: Home position

4.7 Quadrupedal Standing

After the legs were tuned and the quadrupedal home position was calibrated, we first implement quadrupedal standing on the robot. We tested quadrupedal standing by setting-up the robot in its standing position and placing it on the ground. However, our first attempt at quadrupedal standing was unsuccessful as the robot could not support its own weight. At this moment the team realized that we had made a great oversight as we never verified that the robot could support the weight of the OTS electronics. At this time, the robot weighed 3.1kg compared to the original Solo8's weight of 2.2kg. The weight increase was due to our off-the-shelf hardware which was much larger and heavier than the original Solo8's custom electronics. We made the mistake of assuming that the robot could support an additional 0.9kg and never conducted an analysis like we did for bipedal standing.

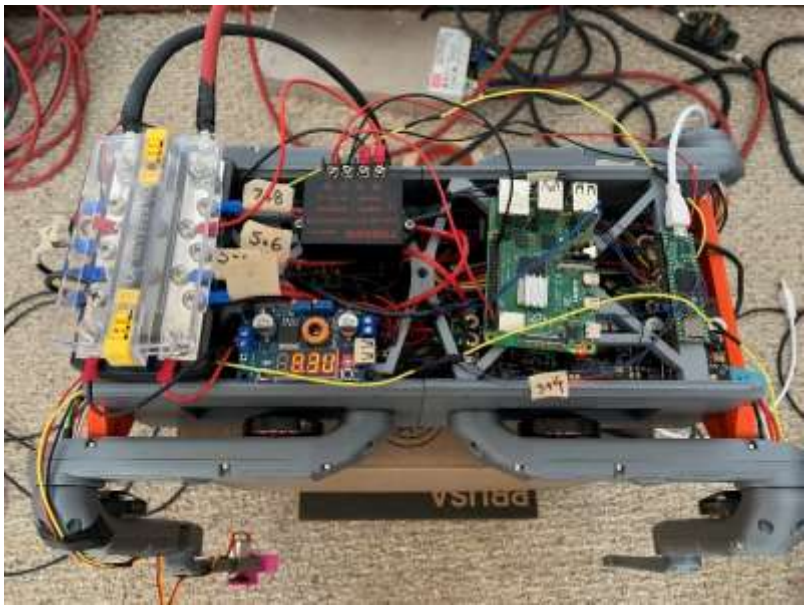


Figure 28: Robot's original electronics with PDU that was too heavy.

Another oversight was that we did not consider how the configuration of the legs would impact the robot's ability to support its own weight. For our initial tests, the rear legs were positioned outside the body as shown in Figure 29. By positioning the legs outside the body, we moved them farther from the robot's COM, thus increasing the amount of torque exerted on the rear leg motors. This problem was compounded by the fact that our off-the-shelf PDU, the robot's heaviest hardware component, was placed on the robot's rear. This further increased the weight that the rear leg motors had to support and as a result the rear legs struggled to support the robot's weight. When the robot tried to support its own weight, the robot's knee joints would bend and the motors on Module 6 and 8 began to overheat.

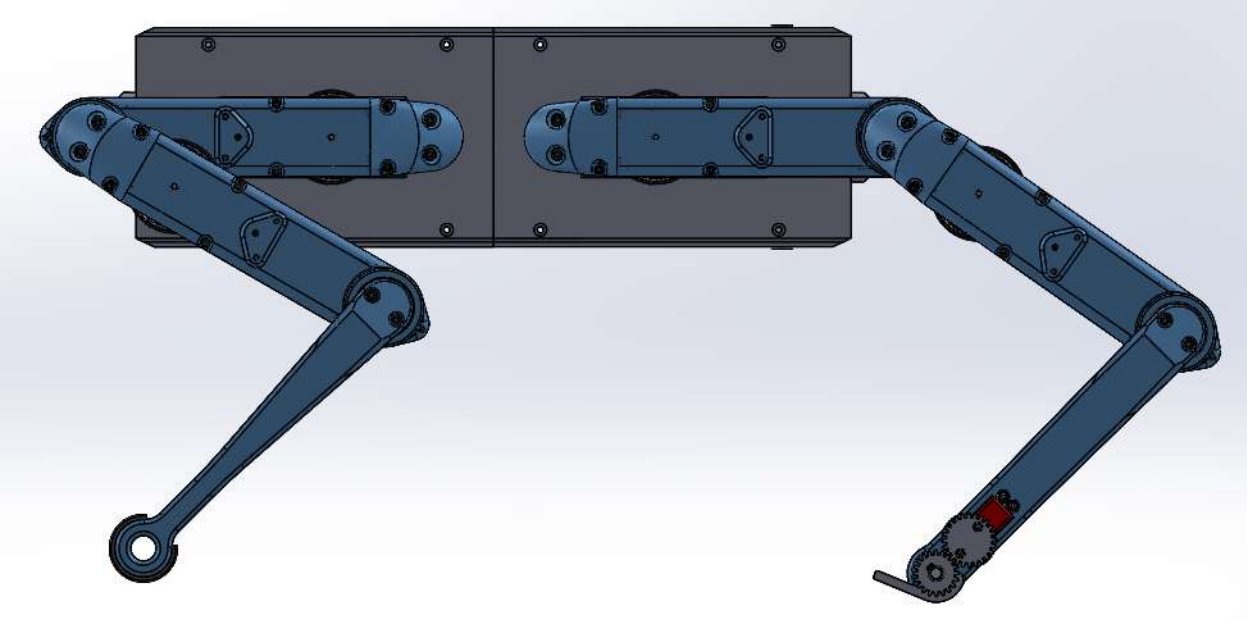


Figure 29: Initial quadrupedal standing leg configuration

Based on our observations the team concluded that the robot's weight had to be reduced. After discussing possible solutions, the team concluded that that the only way to reduce weight was to condense the wiring inside the robot and remove as much hardware as possible. When we first assembled the robot, we did not shorten the lengths of the motor and encoder wires. Instead, we bundled the wires with zip ties, creating a 'rat's nest' of wires inside the robot. To give a sense of how much extra wiring was in the robot's body, Figure 30 shows the wires we removed in order to reduce the robot's weight.

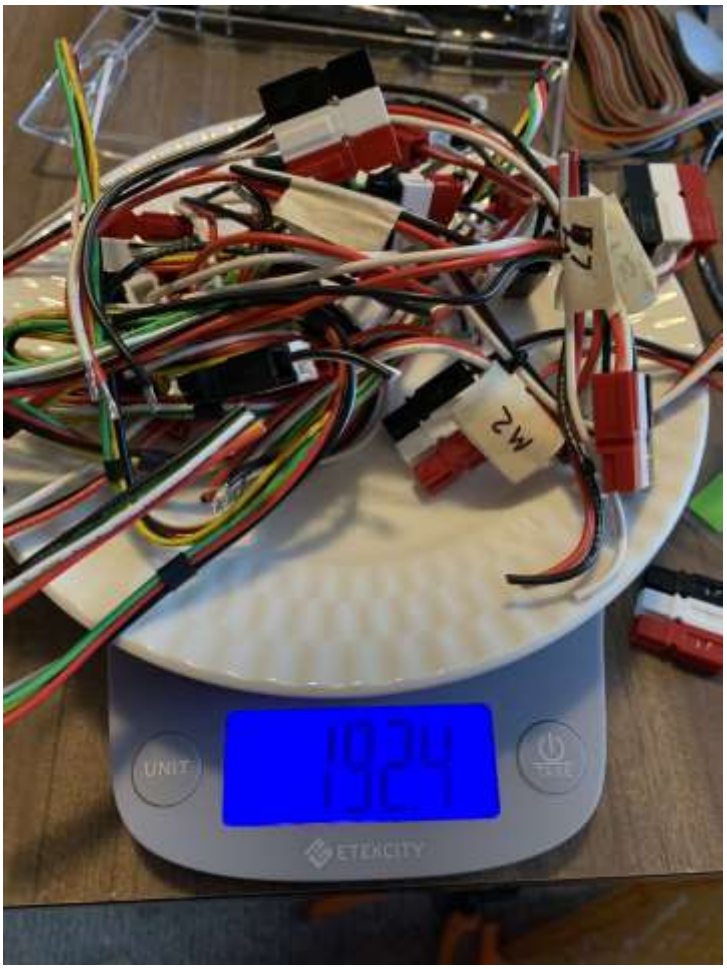


Figure 30: Removed wiring weighing about 190 grams

As seen in Figure 31, we removed roughly 0.2kg of weight by shortening the motor and encoder wires to their optimal lengths. We also removed the Anderson Power Poles that connected the motors to the ODrives and simply fastened the motor wires to the ODrive screw terminals.

Figure 31: Wire reduction

To reduce the weight of the hardware, we evaluated each piece of hardware individually by considering its weight and functionality. We concluded that we could significantly reduce weight if we replaced the off-the-shelf PDU with a lighter, home-made solution. Our solution was to make a PDU out of two ¼" x 1-¾" bolts. The 24V power source's wire ends were secured to the robot and each bolt was directly connected to the positive and negative ends, respectively. The bolts created a PDU from which the ODrives and power converters were connected to. After shortening the wires inside the body and modifying the PDU, we successfully reduced the robot's weight to 2.65kg.

Table 3 – Weight of original hardware

Hardware	Quantity	Weight (kg)
Power Distribution Module	1	0.25
24V/5V Power Converter	1	0.069
24V/8.4V Power Converter	1	0.0286
Teensy	1	0.007
Raspi	1	0.045
Odrive	4	0.088
Total Weight		.76

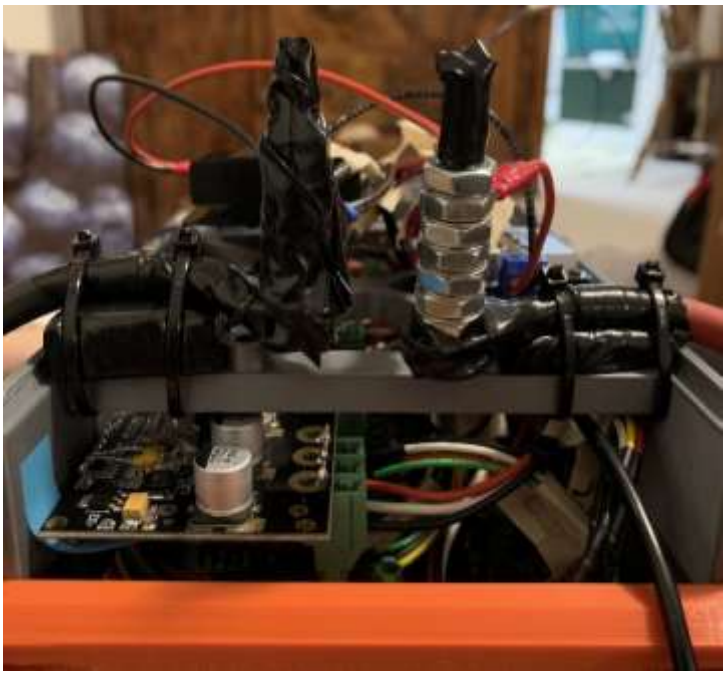


Figure 32: Home-made PDU

We also changed the configuration of the rear legs so that they were inside the robot's body. This reduced the torque on the rear motors. After these changes were implemented, quadrupedal standing was retested, and successfully performed.

Figure 33: Robot standing after reducing the weight of wiring and implementing the new leg configuration.

4.8 Flat Foot Testing

To verify that the feet mechanisms functioned properly and were integrated into the robot's controls, we tested the foot's ability to convert between a flat and point foot. We started with the foot on the back right leg. After wiring the servo to the 24V/8.4V power converter and Teensy, we disassembled the flat foot to initially test the servo. This was done for safety reasons because it was our first time operating the servo and we did not want to damage the servo by over-rotating the flat foot into the lower leg. After verifying that our code operated the servo as intended, we set the servo to 0 degrees and reassembled the flat foot in the point foot configuration. By doing this we protected the servo from over-rotating the flat foot, as the point foot configuration corresponds with the servo's physical limit. Next, we successfully tested the point-to-flat conversion on the right foot and then used the same procedure to test the left foot.

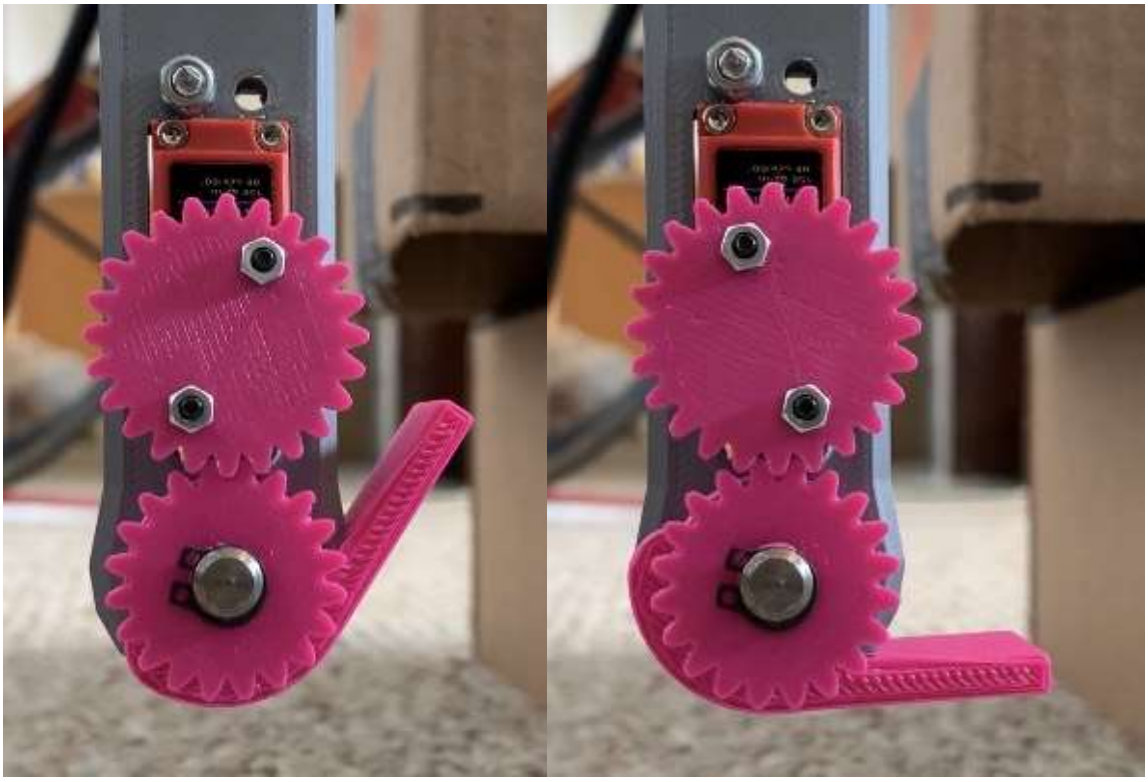


Figure 34: Point foot configuration (left) and flat foot configuration (right)

4.9 Quadrupedal Walking

We implemented a modified symmetric wave gait (an example is shown in Figure 35) for our robot because of its inherent stable nature. Since the Solo8 has 8DoFs, we do not have many ways to change the support polygon for the robot as it walks. By using a symmetric wave gait, we ensured that the robot always had three legs on the ground, giving it a relatively wider

support polygon, and making walking more stable. Furthermore, only having one foot off the ground at a time also reduces the torque load on the rest of the motors.

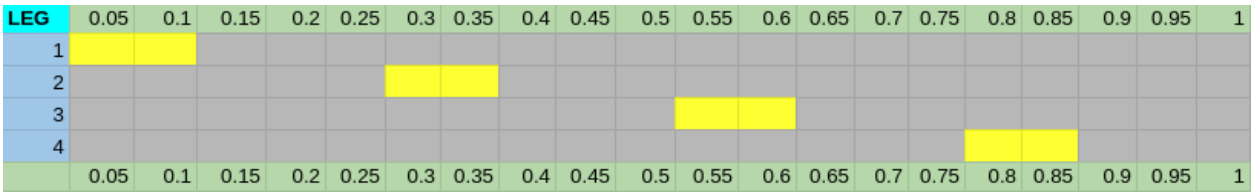


Figure 35: An example of a symmetrical walking gait. Yellow boxed indicate the transfer phase (foot is off the ground) and the grey boxes indicate non transfer phase (foot is on the ground). In our implementation, leg 1 is the front-right, leg 2 is back-right, leg 3 is front-left, and leg 4 is back-left.

We decided to generate our trajectory based on the following parameters after extensive testing in simulation and on the real robot:

- velocity of the robot $v = 0.15m/s$
- stride length of each foot $L = 0.15m$
- duty factor $\beta = 0.9$
- cycle_time $T = 1s$

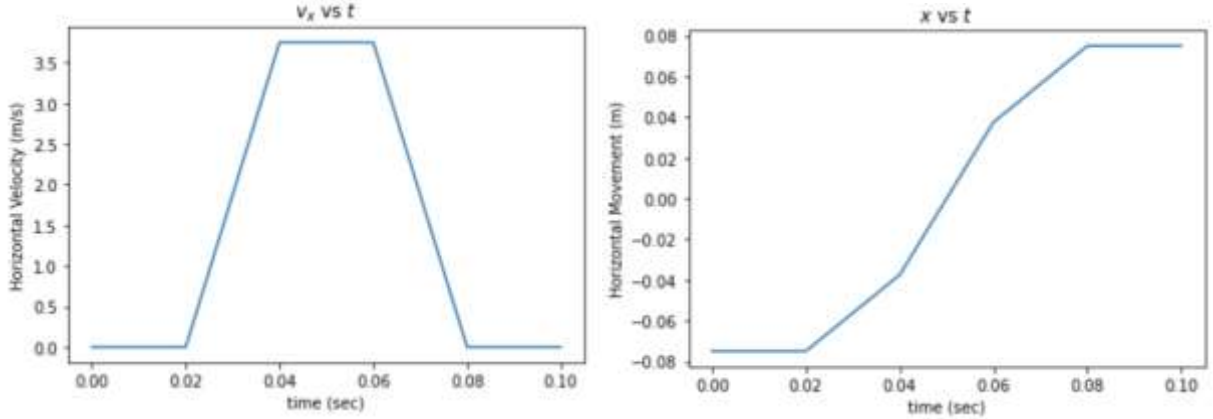
Based on these parameters, we calculated the time in transfer phase for each leg to be $T_l = T(1-\beta) = 0.1\ seconds$. We generated six intermediate points for the transfer phase

1. Begin lifting the transfer foot up
2. Begin moving the transfer foot forward while slowing down the leg lifting
3. Move the leg horizontally with the foot suspended in the air
4. Begin putting the foot down
5. Stop moving the leg horizontally
6. Place the foot down

Based on the stride length, we calculated the max horizontal velocity in Figure 36 (a). Since the area under the velocity curve is the displacement, we simply broke down our curve into basic shapes to compute the area: $(0.06 + 0.02) * \frac{v_x}{2} = 0.15$.

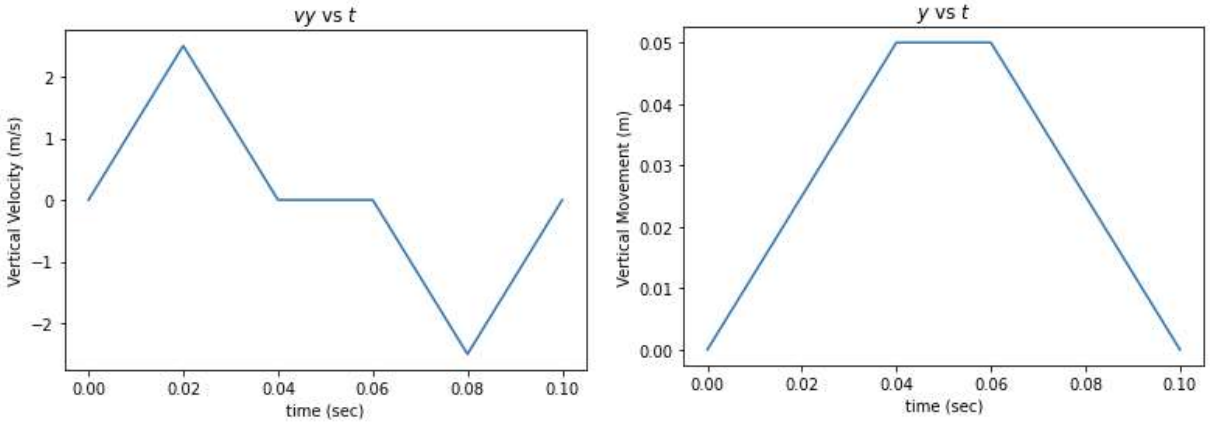
Therefore, each leg must travel at a max horizontal velocity of 3.75 m/s. We used Figure 36 (a) and took the derivative over $\Delta_t = 0.02s$ to calculate the horizontal movement graph in Figure 36 (b). Based on experimentation, we decided to lift the robot's leg 0.05 m vertically during the transfer phase to account for inconsistencies on the ground surface. Each leg is displaced to this vertical distance in the first half of the transfer phase and then brought back to the ground position. This is shown in Figure 37 (b). Based on Figure 37 (a), Figure 37 (b) is calculated by graphing the integral. By combining the graphs in Figure 36 (b) and Figure 37 (b), we can get the graph in Figure 37 (a), the vertical vs horizontal displacement of the leg for the transfer phase with respect to the ground. This is then converted into vertical vs horizontal displacement with respect to the body by subtracting the global position of the origin/hip joint, as shown in Figure 38 (b). The points represented in this graph are the actual values used to move

the leg during the transfer phase. Using inverse kinematics, we calculated the joint angles for each leg for all timesteps during the transfer and non-transfer phases.



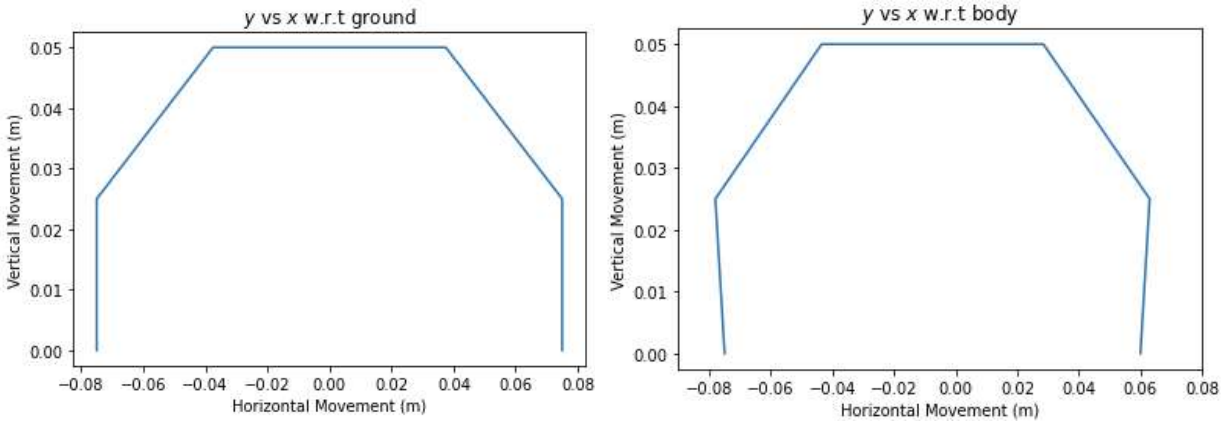
(a) (b)

Figure 36: Horizontal velocity and horizontal displacement of each leg in transfer phase



(a) (b)

Figure 37: Vertical velocity and vertical displacement of each leg in transfer phase



(a) (b)

Figure 38: Vertical displacement vs horizontal displacement with respect to ground and body

As seen in Figure 38 (b), at the beginning of transfer phase, the foot position starts at 0.075*m* behind the joint origin and ends 0.06m in front of the joint origin at the end of the transfer phase. For the non-transfer phase, we simply linearly interpolated between the start and end positions of the transfer phase. Once we had the leg location, we computed the joint angles via inverse kinematics.

While testing the above-mentioned gait on the real robot, the team observed that the robot's rear legs were not completely leaving the ground during its transfer phase. We attributed this observation to the fact that the robot's center of mass was shifted towards the rear side of the body. This essentially caused the robot to tip slightly backwards whenever it was trying to lift the rear legs. To account for this, we decided to tilt the robot forward during walking. This was done by keeping the height of the front legs lower as compared to the height of the rear legs. Based on simple geometric calculations shown below, we calculated the Δ_x and Δ_y for both front and back legs and added them to each point of the leg cycle. These values are visualized in **Error! Reference source not found.**.

$$\alpha = \tan^{-1}\left(\frac{\Delta_h}{l_{body}}\right)$$
$$\Delta_{h_f} = l_f \sin(\alpha)$$
$$\Delta_{x_f} = \frac{\Delta_{h_f}}{\cos(\alpha)}$$
$$\Delta_{y_f} = \Delta_{h_f} \tan(\alpha)$$
$$\Delta_{h_b} = l_b \sin(\alpha)$$
$$\Delta_{x_b} = \frac{\Delta_{h_b}}{\cos(\alpha)}$$
$$\Delta_{y_b} = \Delta_{h_b} \tan(\alpha)$$

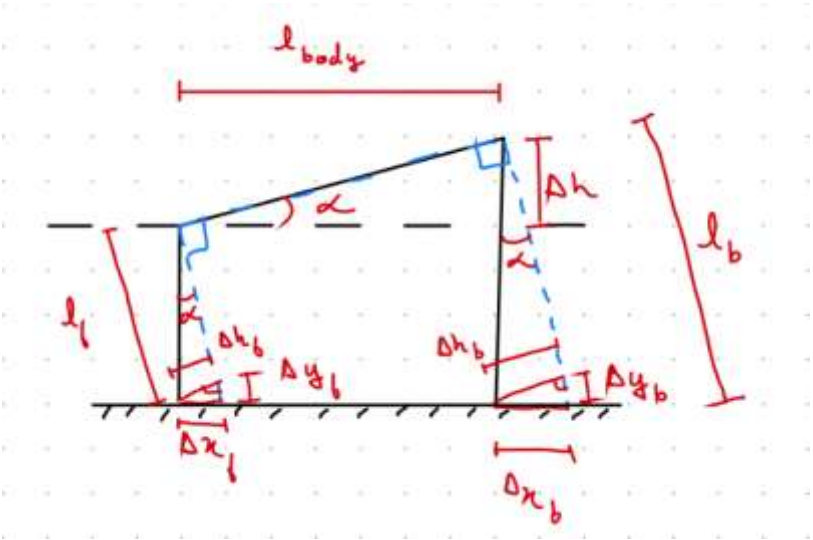


Figure 39. Labels and reference frames for the robot with a forward tilt.

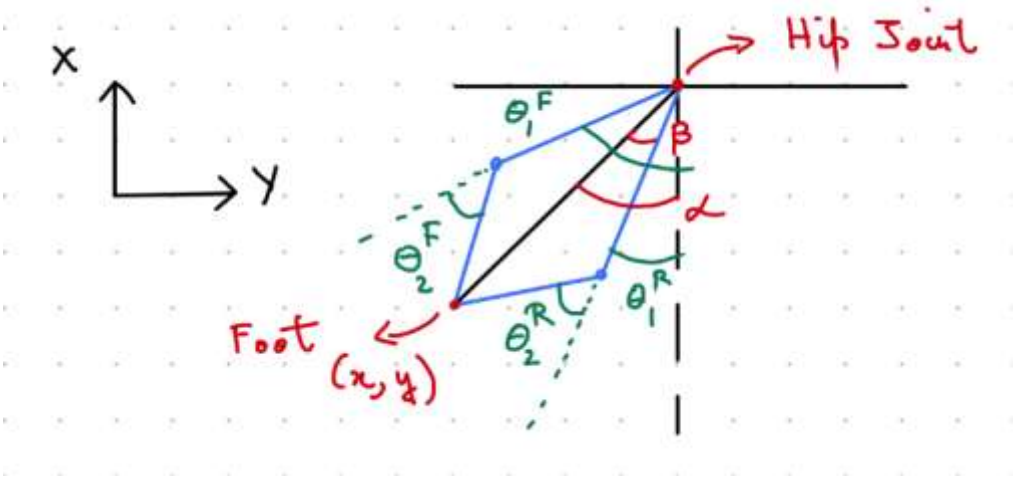


Figure 40. Labels and reference frame for performing inverse kinematics on our Solo's legs.

After accounting for Δ_x and Δ_y for the front and back legs, we converted the position of the foot into joint angles using inverse kinematics. Each leg can be abstracted to a 2 DoF robot arm. Given the foot (x, y) values according to the axis in **Error! Reference source not found.**, we can calculate the following:

$$\alpha = \tan^{-1}\left(\frac{x}{y}\right)$$

Note that our tangent term is $\frac{x}{y}$ instead of $\frac{y}{x}$; this is because of how our axes are defined.

$$\beta = \cos^{-1}\left(\frac{L_1^2 + x^2 + y^2 - L_2^2}{2L_1\sqrt{x^2 + y^2}}\right)$$

With our β term, we can then compute the joint angles:

$$\theta_1^F = \alpha - \beta$$
$$\theta_2^F = \cos^{-1}\left(\frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$
$$\theta_1^R = \alpha + \beta$$
$$\theta_2^R = -\cos^{-1}\left(\frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$

These values can be visualized in Figure 40. θ_1 and θ_2 are then used to move the respective legs to the desired position.

4.10 Stance Transitioning

For this project, we decided to implement the stance transition in a static manner. To do so, we always kept the body's center of mass within the support polygon. We first calculated the robot's center of mass experimentally. It was done this way because our real robot did not perfectly match our simulation model. We calculated the center of mass by balancing the robot on a suspended rod. The center of mass is shown in Figure 41 and Figure 42 as the green circle.

As the next step, we found an initial quadrupedal starting position to transition from. Again, the important constraint was keeping the center of mass within the support polygon. While finding the position, we also wanted to be sure that all the torques on the joints were under the 3.4 Nm stall torque limit after gear reduction. Based on the starting position, we then found a reasonable bipedal position. There were two criteria for deciding this bipedal position. First was that the robot had to somewhat resemble a bipedal robot when in this stance. Second, was that the robot should have minimal movement of hip and knee joints while getting to this position from the initial quadrupedal position. Our major inspiration came from the way humans stand up when they are in squatting position and have their front limbs touching the ground. After some experimentation, involving a manual search for a reasonable starting and ending position, we found the poses in Figure 41 and Figure 42 with the respective torques based of the notation in Figure 18. They satisfied all our criteria.

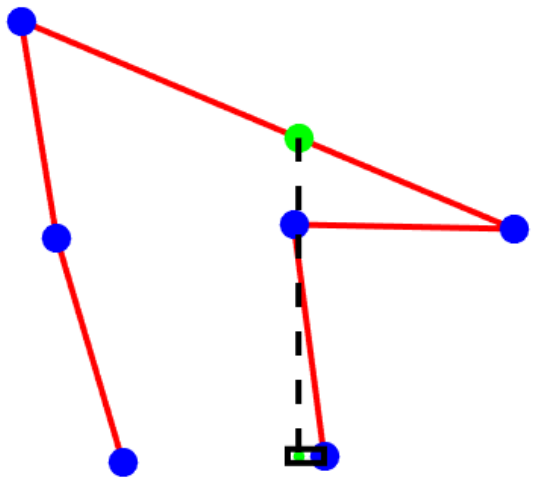


Figure 41. The stance transition starting position. In this position, the motor torque values are:

$$\tau_1 = 0.82385Nm$$
$$\tau_2 = 0.48836Nm$$
$$\tau_3 = 2.9057Nm$$
$$\tau_4 = -0.050753Nm$$
$$\tau_5 = -0.014884Nm$$

We generated a stance transitioning trajectory based on these starting and ending positions. A quintic polynomial was used to generate this trajectory for each of the joints. We preferred the quintic polynomial over its cubic counterpart because it generates smoother transition motions. We generated our trajectory as the following:

Quintic Trajectory:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

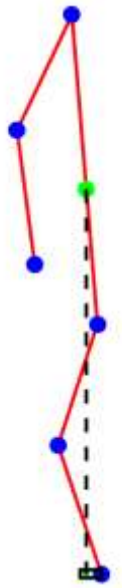


Figure 42. The robot in its stable standing stance.

In this position, the joint torque values are:

$\tau_1 = 0.43189 Nm$
$\tau_2 = -0.39103 Nm$
$\tau_3 = 0.3507 Nm$
$\tau_4 = 0.088422 Nm$
$\tau_5 = -0.0067781 Nm$

Constraint Equations:

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 + a_4 t_0^4 + a_5 t_0^5$$
$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 + 4a_4 t_0^3 + 5a_5 t_0^4$$
$$\alpha_0 = 2a_2 + 6a_3 t_0 + 12a_4 t_0^2 + 20a_5 t_0^3$$
$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 + a_4 t_f^4 + a_5 t_f^5$$
$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 + 4a_4 t_f^3 + 5a_5 t_f^4$$
$$\alpha_f = 2a_2 + 6a_3 t_f + 12a_4 t_f^2 + 20a_5 t_f^3$$

In matrix form:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2t_0 & 3t_0^2 & 4t_0^3 & 5t_0^4 \\ 0 & 0 & 2 & 6t_0 & 12t_0^2 & 20t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ a_0 \\ q_f \\ v_f \\ \alpha_f \end{bmatrix}$$

The quintic polynomials were generated for each of the five joints with the following parameters using a MATLAB script:

- Q_0: starting joint position for the respective joint.
- Q_f: was the ending joint position for the respective joint
- v_0: initial velocity of 0 rad/sec
- v_f: final velocity of 0 rad/sec
- α_0: initial acceleration of 0 rad/sec^2
- α_1: final acceleration of 0 rad/sec^2
- T_0: Starting time of 0 seconds for the transition trajectory
- T_f: ending time of 3 seconds for the transition trajectory

Based on the quintic polynomials, we generated 50 intermediate points. Time to reach a successive waypoint was calculated to be $\frac{3\ seconds}{50} = 0.006\ seconds$. The number of intermediate points was chosen based on experimentation in our simulation. The trajectory along with the time for each way point was then exported as a simple csv file from MATLAB. Our simulation node read the trajectory from this csv and executed it in the simulation environment.

5 Results and Discussion

5.1 Reinforcement Learning

As discussed in Section 4.1.1, we first validated our RL pipeline on Open AI Gym's **Pendulum-v0** environment. Once we had achieved a stable pipeline, we then trained the Solo8 to stand quadrupedally. In this section, we describe our experiments and associated results. Note that in all experiments, the policy network was a 4-layer neural network with an input layer, 2 layers of 64 neurons each, and the output layer. We experimented by adding up to 10 layers between 16 to 256 neurons each but did not notice any nontrivial results. We chose our specific architecture because it was recommended by the authors of stable-baselines and used it to evaluate all our results.

5.1.1 Pendulum-v0

Recall from Section 4.1.1 that **Pendulum-v0** was primarily experimented upon to fix any problems with the RL pipeline. As **Pendulum-v0** has a known solution, there were no modifications done to the states, actions, or rewards—other than those outlined in Section 4.1.1.1. Our autotrainer makes is so that the only hyperparameter we had to tune was the training duration. One issue that we ran into was that we were not training for long enough. As seen in

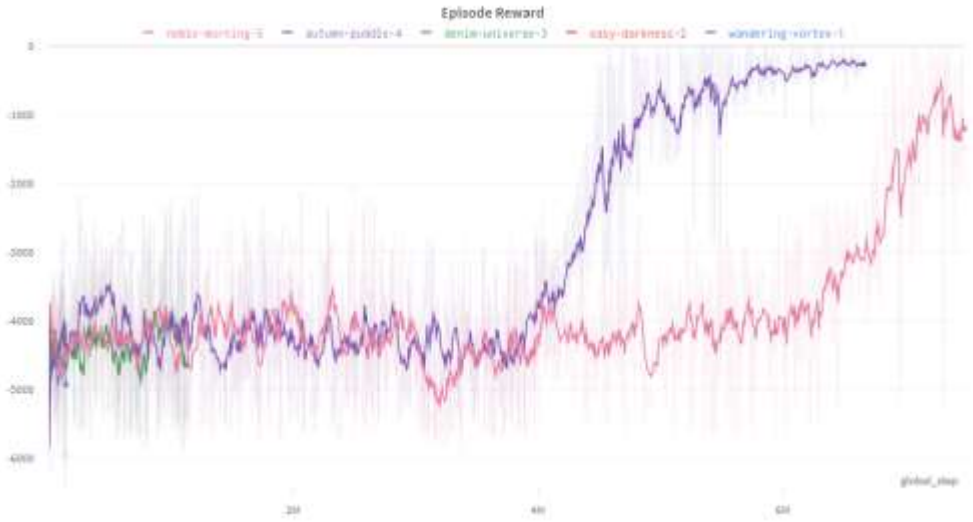


Figure 43. The testing episode reward during the model training. Observe that the major convergences occur between timesteps 4 million and 6 million.

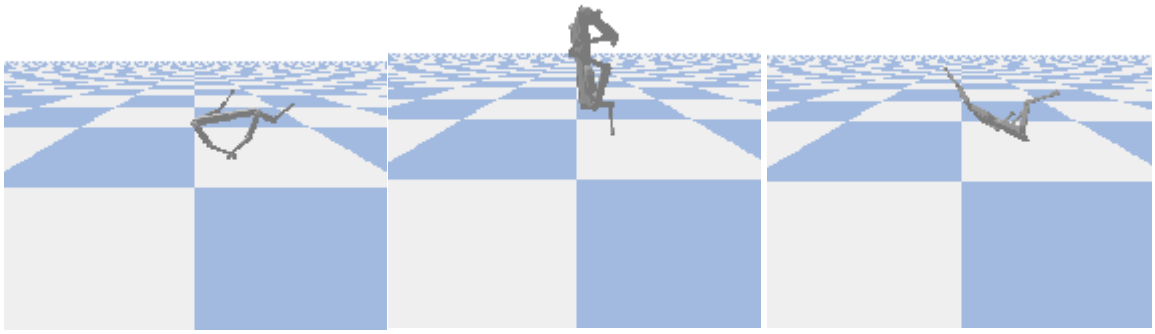


Figure 44. The agent attempting quadrupedal standing with the Naïve Height reward (22). In an episode, the agent randomly moves its limbs, causing it to bounce around. As the height of the torso is usually $\in [0,\ 2 * 0.337]$, the agent is still getting a positive reward, even if it is doing undesirable behavior. The choice for 0.337 as our limit is elaborated upon in Section 5.1.2.1.

Figure 43, the agent seemed to converge between 4 million and 6 million timesteps—when we started, we were prematurely stopping the training at 1 million timesteps.

With 10,000 training episodes at 750 timesteps each, we were able to successfully train an agent to balance the pendulum upright. Our best agent was able to balance the pendulum within 30 timesteps, equating to less than a tenth of a second in real time.

5.1.2 Quadrupedal Standing

With our pipeline validated, we focused on teaching our Solo8 to stand quadrupedally. We wanted to give the agent a total of 5 seconds real time to stand up, so all experiments were run with 2,000 steps, spaced 0.025 seconds apart. Additionally, the Solo8 began each episode with all legs folded and the torso resting on the ground. This behavior can be seen in Figure 44.

5.1.2.1 Naive Height Reward

Our first experiment was simply to reward the agent for maintaining its torso a certain distance off the ground. We empirically found that the torso sits 0.337 meters off the ground during quadrupedal standing, so we set that as our target height. Then, our reward function simply interpolated the location of the torso:

$$r(s_t) = 1 - clamp\left(\frac{|0.337 - h_t|}{h_t}, 0, 1\right) \tag{22}$$

where

- h_t is the height of the torso at timestep t
- $clamp(v, lower, upper)$ returns v if $lower \leq v \leq upper$, $lower$ if $v \leq lower$ and $upper$ if $v \geq upper$

Thus, $r(s_t) \in [0, 1]$ s.t. $r(s_t) = 1$ when the height is at the target. $r(s_t) = 0$ when the absolute difference between the target and torso height is more than 0.337, the target height.

With this reward function, we found that the agent tends to "jump" to the target height. In other words, the agent randomly actuates all its motors with the hopes that a random movement will launch the robot off the ground to the correct height. However, this behavior is undesirable as the agent has no stability at the target height and does not actually achieve quadrupedal standing.

5.1.2.2 Flat Torso Reward

The most apparent problem that we saw with (22) was that the agent was actuating its joints too sporadically. To combat this, we introduced a new *flat torso* term:

$$r(s_t) = 1 - \frac{\left(clamp\left(\frac{|0.337 - h_t|}{h_t}, 0, 1\right) + \left(\frac{\Delta\theta_x}{2 * \pi} + \frac{\Delta\theta_y}{2 * \pi}\right)\right)}{2} \tag{23}$$

note that:

- $\Delta\theta_x$ is the absolute deviation from the x axis and $\Delta\theta_y$ with the y axis, respectively
- $\Delta\theta_x$ and $\Delta\theta_y$ are divided by π as the maximum absolute deviation for an angle is valued $\in (0, pi)$, assuming the angle is reported in radians. These values are then divided by 2 to ensure that the total value adds up to 1.

Similar to (22), $r(s_t) \in [0, 1]$ in (23). However, with this reward function, we found that the agent often "tips" over and gets stuck in an upside-down position, as shown in Figure 45. Intuitively, this behavior makes sense. As the robot is stable on its back, it is achieving a perfect flat torso reward, at the expense of the height reward. At the end of the episode, the agent tries to get some height again by actuating its joints, but by that point the agent has already reached an

Figure 45. The agent trying to stand quadrupedally using (23) as the reward function. The agent tended to "jump" up at the beginning to achieve the height reward but ends up tipping over onto its back—achieving a perfect "flat" reward. Near the end of the episode, the agent tries to move its legs to lift it up, but that is not enough to achieve quadrupedal standing.

unrecoverable state. So, even though the torso might have been more stable, the agent still could not stand quadrupedally.

5.1.2.3 Stability Reward

When training their quadruped, Google DeepMind also ran into issues with their agent consistently entering an unrecoverable state during training episodes. They combated this by penalizing large changes in the robot's horizontal location and joint positions [42]. We implemented a similar reward function:

$$r(s_t) = 1 - \left(\frac{\left(clamp\left(\frac{|0.337 - h_t|}{h_t}, 0, 1 \right) + \left(\frac{\Delta\theta_x}{2 * \pi} + \frac{\Delta\theta_y}{2 * \pi} \right) \right)}{2} \right) \frac{v_h}{l_h} \frac{v_\theta}{l_\theta} \tag{24}$$

where

- v_h is the horizontal velocity of the agent
- v_θ is the average joint velocity
- l_h and l_θ are scaling factors on the horizontal and joint velocities. These values were determined experimentally

Google DeepMind found success in multiplying by their stability reward rather than adding it as the effect distributes to both subrewards (the torso flatness and height rewards, respectively) [42]. As seen in Figure 46, during episodes the agent "jumps" up and lands in a quadrupedal standing stance. However, instead of dynamically moving its legs, the agent simply locks them into a stable position. Unfortunately, this "stable position" does not reach the target height. Therefore, while the robot is standing quadrupedally, it did not perform the desired behavior but rather found a shortcut to a locally optimal solution.

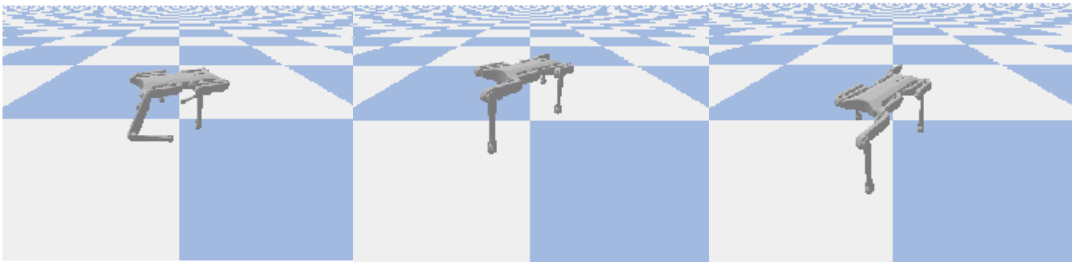


Figure 46. The agent standing quadrupedally by optimizing (24). The agent jumps up, locks its legs, and simply stands on them. In this configuration, the agent's height is only 0.3, when the target height is 0.337. However, the agent is incredibly stable as there are no actions performed once the agent is standing. For this agent, $l_h = 3.215m$ and $l_\theta = 10\, rad/s$.

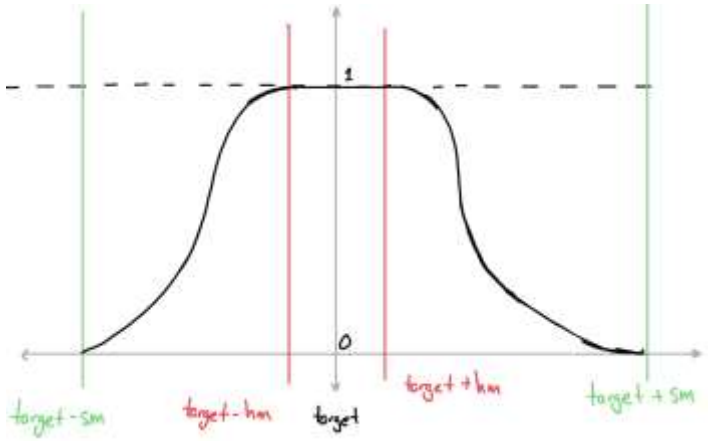


Figure 47. A visual representation of our custom Gaussian tolerance function. Observe how the reward function returns 1while x is within hm of the $target$.

5.1.2.4 Gaussian Tolerances

In the previous experiments, we see that the agent optimizes some subrewards while "sacrificing" others. This can be seen in Figure 46, where the robot achieves perfect small control, horizontal velocity, and torso flatness rewards—but a suboptimal height reward. Our intuition behind this behavior was that there was "no tolerance" in the rewards.

Since all of our subrewards are linearly interpolated, the only time the agent can get a reward of 1 is to achieve perfect quadrupedal standing. However, that is highly unrealistic, and the agent can be considered to be successfully standing quadrupedally even if there are slight deviations in the orientation and joint control.

To allow for toleranced rewards, we wrote a custom Gaussian interpolator:

$$g(x, target, sm, hm) = \begin{cases} 1, & if\ \text{abs}(x - target) < hm \\ \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\left(\frac{x \pm hm}{sm}\right) - (target \pm hm)}{-2 * \log(0.01)}\right)}, & if\ hm \leq \text{abs}(x - target) \leq sm \\ 0, & if\ \text{abs}(x - target) > sm \end{cases} \quad (25)$$

Where hm is the *hard margin* of the tolerance and sm is the *soft margin* of the tolerance. The behavior of the function (visualized in Figure 47) is as follows:

- If the difference between x and $target$ is less than *hard margin*, yield a reward of 1 (perfect reward)
- If the difference between *x* and *target* is more than *soft margin*, yield a reward of 0

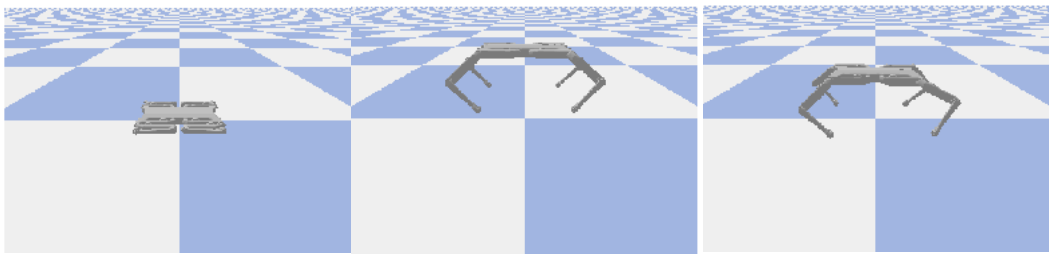


Figure 48. Quadrupedal standing results using (26) as the reward function. Here, the agent jumps up, lands on four feet, and performs leg adjustments to stay stable.

- If the difference is between *hard margin* and *soft margin*, then interpolate based off a Gaussian centered at $target + hm$ and returns 0.01 when $x = sm$. Note that even though this calculation is based off the general formula for a PDF of a Gaussian distribution [44], it is scaled as per Section 4.1.1.1 to be in [-1, 1].
- Note that hm is either added or subtracted from x and/or $target$ depending on if $x \leq target$ or $x \geq target$

With this new tolerance function, we rewrote (24) as:

$$r(s_t) = 1 - \left(\frac{clamp(g(h_t, 0.337, sm_h, hm_h), 0, 1) + g(\Delta\theta_x, 0, sm_x, l_x) + g(\Delta\theta_y, 0, sm_y, l_y)}{3} \right) \times g(v_h, 0, sm_v, l_v) g(v_\theta, 0, sm_{sc}, l_{sc}) \quad (26)$$

where:

- sm_h and hm_h are the soft and hard margins for the torso height reward, respectively
- sm_x is the soft margin for the torso rotation rewards, respectively. Note that $sm_x = sm_y$ in practice for symmetry.
- sm_v is the soft margin for the horizontal velocity reward
- sm_{sc} is the soft margin for the joint velocity reward

Using (26), we were able to achieve successful quadrupedal standing. As shown in Figure 48, the agent successfully brings the torso to the desired height and maintains it via dynamically adjusting its legs. By allowing tolerances in our reward function, we were able to train the agent to perform desired behavior, instead of settling for a local maximum.

Our final model was automatically tuned by our RL pipeline, as explained in Section 3.1, and the parameters for our best agent are as follows:

Parameter	**Value**
Episodes	12000
sm_h	0.1636
hm_h	0.007876

sm_x	4.33
l_x	0.06703
sm_y	4.33
l_y	0.06703
sm_v	3.168
l_v	0.9127
sm_{sc}	0
l_{sc}	10.808

5.2 Quadrupedal Walking Tests

The quadrupedal symmetric wave gait was tested on multiple surfaces to evaluate the gait's performance in different environments. These tests were all performed on relatively flat surfaces and produced varying results. This section describes the walking tests we conducted and gives the results of those tests.

Carpet

The gait was first tested on carpet because this was the flooring of the room where we built and set-up the robot. The robot successfully performed quadrupedal walking without any issues. Due to the size of the room and length of the power cord, the robot could only walk a maximum distance of roughly seven feet.

Figure 49: Robot performing walking gait on carpet.

Plywood

Tests were also conducted on a 4’x 2’ sheet of plywood. During multiple tests, the robot successfully walked; however, we routinely observed instances where the gait skipped steps as a result of the robot slipping on the plywood. The team attributed the skipping motion to the robot slipping and then regaining traction on the surface. During these tests, the robot walked roughly 3’ because it was limited by the size of the plywood.

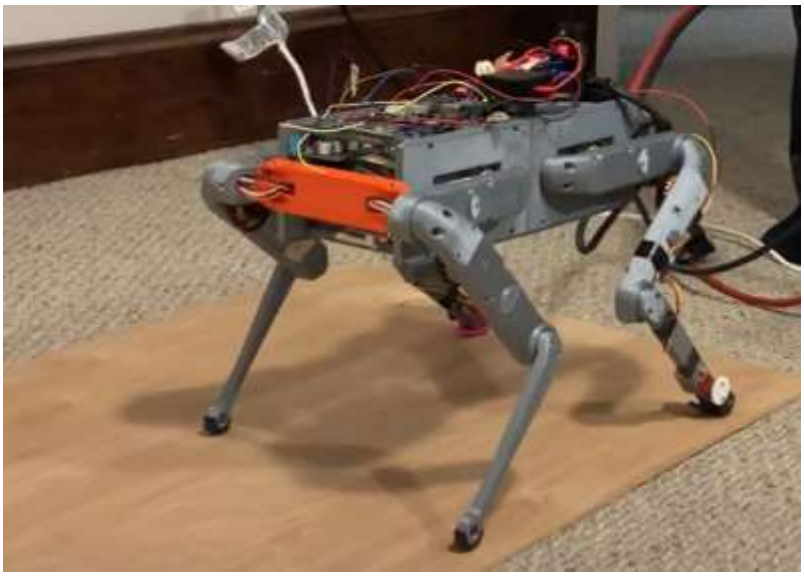


Figure 50: Gait test on plywood board

Brick Path and Concrete

We tested the gait outside, on the brick and concrete paths around the Quad. Our observations and results were similar to those of the plywood test: the robot successfully walked on both surfaces, but the gait would occasionally skip. During the instances where the gait skipped, it was clear that the robot had momentarily lost traction, just as we observed during the tests on the plywood surface. We also tested the gait on a section of the path where the surface changed from concrete to brick. In this case, the gait was able to transition between the surfaces, but we did observe it skip.

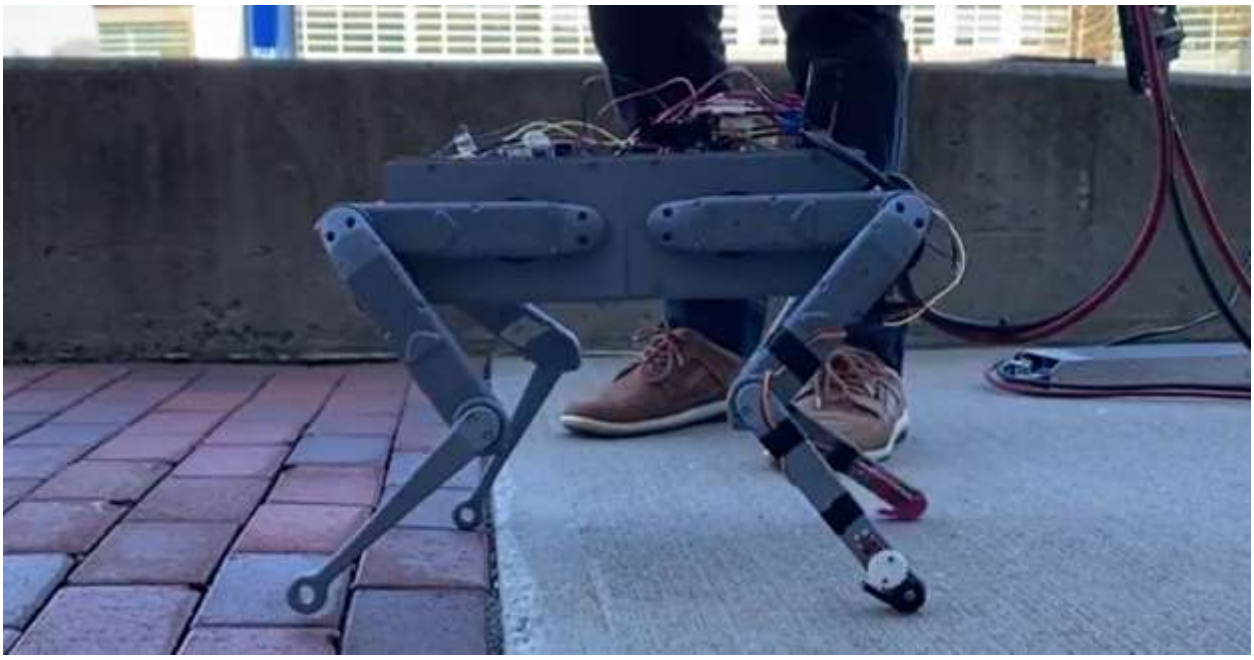


Figure 51: Gait transitioning from concrete surface to brick path.

Grass

On grass, the walking gait was unsuccessful. The robot could not produce any forward motion and instead skipped in place. Fearing we would damage the robot by conducting further tests, we stopped and only performed a single test on grass. The team believes that the skipping in place was a more severe case of the phenomenon we observed during tests on the plywood and concrete & brick paths. A combination of poor traction and a slightly uneven surface interfered with the gait and prevented forward motion.

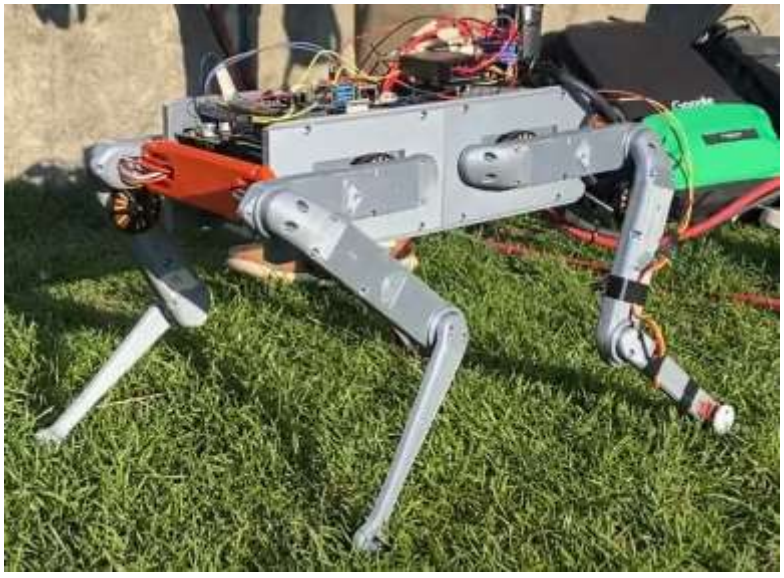


Figure 52: Gait test on grass

Foam Matt

We tested the gait on interlocking foam matts. These tests were successful, and the phenomenon of the gait skipping was never observed. From our observations, the gait appeared to function the smoothest when on the foam matts.

Cobblestone

Our last test was on the cobblestone path near the Fountain. Our tests on cobblestone were brief because the robot broke during testing. The robot performed quadrupedal walking during the first test, but while we were setting up the robot for a second test, the front left leg's hip actuator module (Module 1) broke. While the robot was positioned in its prepared walking stance, the motor overheated, melting the PLA module shell from which the motor is secured to. A combination of the leg supporting the robot's weight and the motor's heat deforming the melting plastic, caused the motor to break free from the shell. At this moment the module had been compromised to a point where it could not support the robot's weight and the robot subsequently collapsed.

Figure 53: Robot on cobblestone path moments before the motor overheated.

We are confident that the motor did not overheat because of current overload. Our reasoning is that the ODrives limit the motors' received current to 10A. If the ODrive receives a greater current than the set limit, the ODrive automatically powers off. Therefore, if too much current was being delivered to the ODrive and motor, the ODrive would have shut-down before the motor could become hot enough to melt the actuator module shell. This leads us to believe that the motor overheated for another reason. We believe the culprit was the uneven surface of the cobblestone. When we placed the robot on the ground, we think that the front left leg was positioned in a way where the leg contacted the ground at a point slightly below the other legs. This imbalanced the robot's weight distribution and put more weight on the front left leg. Module 1 had to support this added weight, and this caused the motor to heat up. The extent of the damage to Module 1 is detailed in Section 5.2.1.

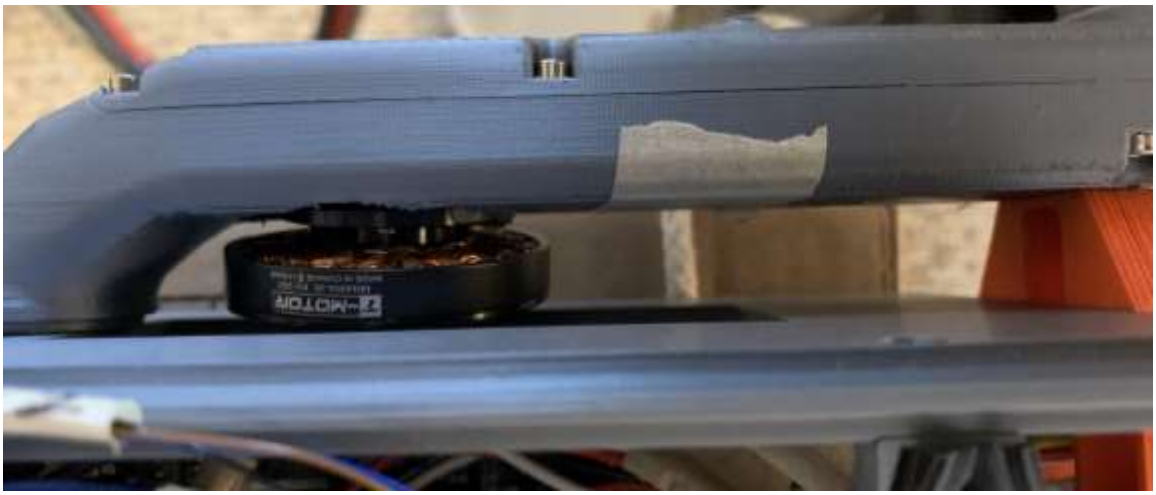


Figure 54: Motor melted the PLA and broke off its mounting.

5.2.1 Damage to Module 1

After breaking Module 1, we disassembled the front left leg and assessed the damage to the module. Once the module was disassembled, we determined that the damage was much less than initially feared. The timing belts were in working condition, the motor shaft was not bent, and the codewheel and encoder were not visibly damaged. The motor's stator had some melted plastic on its coils which was manually cleaned off. The most damage was to the actuator module's shell base. The PLA around the motor mount was severely deformed as the plastic had melted and then solidified once cool. Some PLA was fused to the motor and the shell had to be cut apart to free the motor from the plastic.

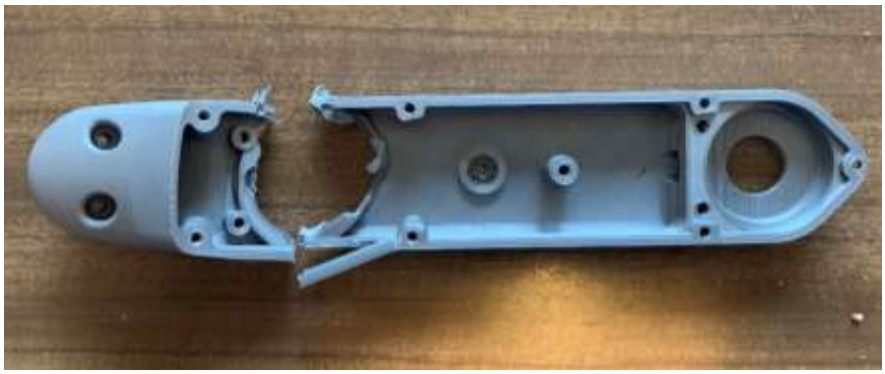


Figure 55: We had to cut the shell apart to free the motor.

We 3D printed a new module shell base to replace the broken one and reassembled the module. The module's motor and encoder were tested, and the module was determined to function properly. Interestingly, the occurrence of the module shell melting might have been avoided if we used PC-ABS, like the Solo8 developers use, rather than PLA. The antigravity motors are designed for drones, and as such they rely on air cooling for temperature control.

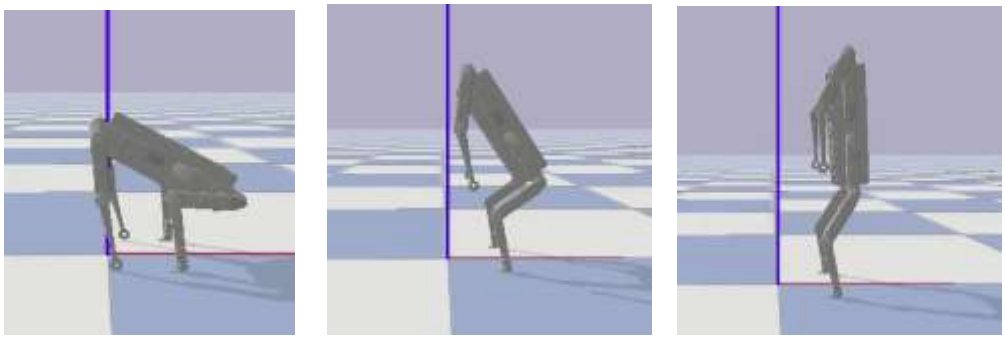
However, on the robot, the motors do not receive much air flow since the robot and motors do not move very fast compared to a drone. This allows the motors to overheat easier, especially when they are stationary and holding a load above their stall torque, as we theorize happened when Module 1 broke. PC-ABS' melting point is roughly 100°F greater than PLA, which might have withstood the heat from the motor overheating. In this way, our design choice to use PLA over PC-ABS may have played a role in Module 1 breaking and is an important note for future design iterations on the robot.

5.2.2 Walking Gait Discussion

Our tests results show that the quadrupedal symmetric wave gait was successfully implemented on the robot. While Module 1 broke during testing, we believe that the module breaking was unrelated to the gait since the module broke while the robot was stationary. The robot was unable to walk on grass, but it performed its walking gait on carpet, plywood, foam matts, concrete, and the brick and cobblestone surfaces. However, our tests show that the gait is far from ideal, as minor surface imperfections can be problematic for the gait. The robot's inability to walk on uneven surfaces (i.e., grass), demonstrates the need for a more adaptable control system that can dynamically respond to its environment. The tests also show that the robot sometimes slips on smooth, low traction surfaces, or surfaces like concrete and brick paths that have small imperfections. The symmetric wave gait is a good first step as it validates the robot's quadrupedal capabilities and achieves the team's goal of implementing quadrupedal locomotion on the robot. However, more work is needed to increase the robot's functionality and develop a walking gait that can maneuver through unstructured environments.

5.3 Stance Transition

We were successfully able to demonstrate stable bipedal standing in simulation. The standing trajectory gave us consistent results. While we were looking forward to implementing the transition trajectory on the real robot, we were unable to do so. This was due to lot of technical challenges that occupied our time, leaving us unable to spend time on our stretch goal of implementing the standing trajectory on the real robot.



(a) (b) (c)

Figure 56. Our Solo8 performing a stance transition in simulation. (a) is the starting position, (b) is a intermediate position and (c) is the ending position.

6 Recommendations and Future Work

Since our project was intended to be the first step in developing a multi-modal robotic platform, there is plenty of future work to be done to improve the Solo8's quadrupedal and bipedal capabilities. Some of this future work involves implementing stance transition on the real robot, improving the robot's control system, or making further mechanical modifications to the robot's design. This work is intended to further develop the robot's multi-modal abilities and improve the robot's locomotion strategies.

6.1 Implementing Multi-Modalism

While the team achieved stance transition in simulation, we did not have enough time to implement the transition on the real robot. Right now, the Solo8 only has the theoretical capability to transition into a bipedal stance and stand stably. With more time, future teams can test the stance transition on the real robot, implement stable bipedal standing in real life, and develop a bipedal walking gait in simulation. This work would dramatically improve the Solo8's multi-modal abilities. Work can also be done to optimize the quadrupedal walking gait. We implemented a quadrupedal symmetric wave gait because it was the simplest gait to develop in the time that we had. While the gait works, it was not compared to other gaits to optimize quadrupedal locomotion. In the future, a team can develop and implement additional quadrupedal walking gaits to improve its quadrupedal abilities.

6.2 Control System Improvements

Another way to improve the robot's capabilities is to implement dynamic control. Currently, the Solo8 can only walk on flat surfaces because its control system has no sensing capabilities and cannot adapt to its environment. A dynamic control system would provide intelligent motion and increase the robot's functionality. As part of a dynamic control system, future teams can also implement autonomous motion by adding a camera or other sensors to the

robot. With the ability to make decisions on its own, the Solo8 would have adaptability in unstructured environments and could decide when and where to change its stance or locomotion type.

6.3 Design Modifications

Larger design modifications, like increasing the Solo8's degrees of freedom or designing manipulators for the front limbs, could also be pursued. The robot's 10 degrees of freedom is enough to accomplish basic bipedal locomotion; however, mobility could be improved if an additional joint were created at the hip. The ODRI has a 12DoF version of the Solo8, with hip joints, and our robot could be modified with the ODRI's design. The front limbs can also be modified to provide additional functionality in bipedal mode. Right now, the front limbs have no task completion capabilities as they only have 2DoFs and no manipulators. Future teams can design and implement manipulators that allow the Solo8 to perform tasks when in its bipedal stance.

More work can also be done to optimize the flat feet. Even though the current flat foot design increases the robot's bipedal support polygon, the support polygon is relatively small compared to the robot's body. In its current iteration, the robot might have trouble remaining stable when the simulated stance transition is attempted on the real robot. A larger foot would improve stability, making bipedal standing easier for the robot and bipedal locomotion more feasible. An in-depth analysis of the foot could be conducted to determine the optimal foot length that provides the greatest support polygon without creating a moment arm that produces too much torque on the servo. There are also stronger micro-servos available on the market that could be implemented with minor changes to the design. To address the slipping issue we observed during the quadrupedal gait tests, future teams can prototype different foot materials that provide greater traction.

If a camera or other sensors were added to improve the robot's control system, new mountings might need to be designed and incorporated into the existing body structure. The body structure can also be changed if future teams decide that design changes are required to better support new sensing hardware.

6.4 Reinforcement Learning

Since our pipeline is already developed, future teams have the tools to conduct more experiments. We have already achieved quadrupedal standing via RL on ODRI's original Solo8. Next steps in quadrupedal standing include re-training an agent using our updated Solo8 model. Additionally, due to the seamless nature of our sim-to-reality pipeline, all models can be run on the real robot as well—opening up interesting questions on how well theory translates to practice.

Furthermore, as our RL pipeline uses OTS implementations, it is compatible with the state-of-the-art algorithms. In our project, we used PPO2 to train our agent, but a potential venue for research would be comparing the efficacy between PPO2 and other upcoming RL algorithms.

Finally, we only explored quadrupedal standing to show that our pipeline works and can be used to achieve intelligent behavior. Past quadrupedal standing, there is room to explore quadrupedal walking as well as tasks such as fetching. Since our robot also supports bipedal locomotion, agents can be trained to perform stance transitions and bipedal tasks. With the tools we have developed, we hope to lower the barrier to explore these interesting problems—and hopefully get to see our physical robot run these algorithms.

7 Conclusion

We intended our project to be a stepping-stone in the development of a robot capable of quadrupedalism and bipedalism. Despite COVID and time restrictions, we believe that we have created a novel multi-modal platform. We have built a physical robot, developed an accompanying software stack with a reinforcement learning pipeline, and implemented quadrupedal locomotion on the real robot as well as achieved stance transition in simulation. We are excited about the possibilities that our platform affords to future roboticists. Using locomotion strategies developed on our robot, we hope to inform and inspire a new era of multi-modal robotics research here at WPI and beyond.

References

[1] N. B. Ignell, N. Rasmusson, and J. Matsson, “An overview of legged and wheeled robotic locomotion,” 2012. Accessed: Sep. 21, 2020. [Online].

[2] Y. Zhong, R. Wang, H. Feng, and Y. Chen, “Analysis and research of quadruped robot’s legs: A comprehensive review,” *International Journal of Advanced Robotic Systems*, vol. 16, no. 3, p. 172988141984414, May 2019, doi: 10.1177/1729881419844148.

[3] E. Guizzo, “By leaps and bounds: An exclusive look at how Boston dynamics is redefining robot agility,” *IEEE Spectrum*, vol. 56, no. 12, pp. 34–39, Dec. 2019, doi: 10.1109/MSPEC.2019.8913831.

[4] 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: Journal of Systems and Control Engineering*, vol. 225, no. 6, pp. 831–849, Aug. 2011, doi: 10.1177/0959651811402275.

[5] S. Dubey, “Robot Locomotion-A Review Design and Development of Pine Needle Collector Robot View project.” [Online]. Available: http://www.ripublication.com.

[6] S. Böttcher, “Principles of robot locomotion.” Accessed: May 05, 2021. [Online].

[7] N. Kashiri *et al.*, “An overview on principles for energy efficient robot locomotion,” *Frontiers Robotics AI*, vol. 5, no. DEC. Frontiers Media S.A., p. 129, Dec. 01, 2018, doi: 10.3389/frobt.2018.00129.

[8] S. Mintchev and D. Floreano, “Adaptive morphology: A design principle for multimodal and multifunctional robots,” *IEEE Robotics and Automation Magazine*, vol. 23, no. 3, pp. 42–54, Sep. 2016, doi: 10.1109/MRA.2016.2580593.

[9] W.-M. Shen *et al.*, “Multimode locomotion via SuperBot reconfigurable robots,” vol. 20, pp. 165–177, 2006, doi: 10.1007/s10514-006-6475-7.

[10] S. Russo *et al.*, “Design of a robotic module for autonomous exploration and multimode locomotion,” *IEEE/ASME Transactions on Mechatronics*, vol. 18, no. 6, pp. 1757–1766, 2013, doi: 10.1109/TMECH.2012.2212449.

[11] S. Kuswadi, M. N. Tamara, D. A. Sahanas, G. I. Islami, and S. Nugroho, “Adaptive morphology-based design of multi-locomotion flying and crawling robot ‘PENS-FlyCrawl,’” in *2016 International Conference on Knowledge Creation and Intelligent Computing, KCIC 2016*, Mar. 2017, pp. 80–87, doi: 10.1109/KCIC.2016.7883629.

[12] F. Ois Michaud *et al.*, “Multi-Modal Locomotion Robotic Platform Using Leg-Track-Wheel Articulations *,” 2005. [Online]. Available: http://www.gel.usherb.ca/laborius.

[13] A. Stentz *et al.*, “CHIMP, the CMU Highly Intelligent Mobile Platform,” *Journal of Field Robotics*, vol. 32, no. 2, pp. 209–228, Mar. 2015, doi: 10.1002/rob.21569.

[14] K. Hashimoto *et al.*, “WAREC-1 - A four-limbed robot having high locomotion ability with versatility in locomotion styles,” in *SSRR 2017 - 15th IEEE International Symposium on Safety,*

Security and Rescue Robotics, Conference, Oct. 2017, pp. 172–178, doi: 10.1109/SSRR.2017.8088159.

[15] T. Kobayashi, T. Aoyama, M. Sobajima, K. Sekiyama, and T. Fukuda, "Locomotion selection strategy for multi-locomotion robot based on stability and efficiency," in *IEEE International Conference on Intelligent Robots and Systems*, 2013, pp. 2616–2621, doi: 10.1109/IROS.2013.6696725.

[16] D. Kuehn, M. Schilling, T. Stark, M. Zenzes, and F. Kirchner, "System Design and Testing of the Hominid Robot Charlie," *Journal of Field Robotics*, vol. 34, no. 4, pp. 666–703, Jun. 2017, doi: 10.1002/rob.21662.

[17] Y. Huang, Q. Li, A. Ming, Y. Liu, Y. Liu, and Q. Huang, "Dynamic Gait Transition of a Humanoid Robot from Hand-Knee Crawling to Bipedal Walking based on Kinematic Primitives," in *Proceedings of IEEE Workshop on Advanced Robotics and its Social Impacts, ARSO*, Oct. 2019, vol. 2019-Octob, pp. 240–245, doi: 10.1109/ARSO46408.2019.8948746.

[18] B. Yoon and S. Kim, "Note: Reconfigurable pelvis mechanism for efficient multi-locomotion: Biped and quadruped walking," *Rev. Sci. Instrum*, vol. 88, p. 126104, 2017, doi: 10.1063/1.4990544.

[19] T. Kamioka, T. Watabe, M. Kanazawa, H. Kaneko, and T. Yoshiike, "Dynamic gait transition between bipedal and quadrupedal locomotion," in *IEEE International Conference on Intelligent Robots and Systems*, Dec. 2015, vol. 2015-Decem, pp. 2195–2201, doi: 10.1109/IROS.2015.7353671.

[20] K. Asa, K. Ishimura, and M. Wada, "Behavior transition between biped and quadruped walking by using bifurcation," *Robotics and Autonomous Systems*, vol. 57, no. 2, pp. 155–160, Feb. 2009, doi: 10.1016/j.robot.2008.04.005.

[21] S. Gay, J. Santos-Victor, and A. Ijspeert, "Learning robot gait stability using neural networks as sensory feedback function for Central Pattern Generators," in *IEEE International Conference on Intelligent Robots and Systems*, 2013, pp. 194–201, doi: 10.1109/IROS.2013.6696353.

[22] S. Aoi and K. Tsuchiya, "Transition from quadrupedal to bipedal locomotion," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 2005, pp. 3419–3424, doi: 10.1109/IROS.2005.1545499.

[23] S. Aoi, Y. Egi, R. Sugimoto, T. Yamashita, S. Fujiki, and K. Tsuchiya, "Functional roles of phase resetting in the gait transition of a Biped Robot from quadrupedal to bipedal locomotion," *IEEE Transactions on Robotics*, vol. 28, no. 6, pp. 1244–1259, 2012, doi: 10.1109/TRO.2012.2205489.

[24] F. B. Ouezdou, S. Alfayad, and B. Almasri, "Comparison of several kinds of feet for humanoid robot," in *Proceedings of 2005 5th IEEE-RAS International Conference on Humanoid Robots*, 2005, vol. 2005, pp. 123–128, doi: 10.1109/ICHR.2005.1573556.

[25] K. Fondahl *et al.*, "An adaptive sensor foot for a bipedal and quadrupedal robot," in *Proceedings of the IEEE RAS and EMBS International Conference on Biomedical Robotics and Biomechatronics*, 2012, pp. 270–275, doi: 10.1109/BioRob.2012.6290735.

[26] R. Kaslin, H. Kolvenbach, L. Paez, K. Lika, and M. Hutter, “Towards a Passive Adaptive Planar Foot with Ground Orientation and Contact Force Sensing for Legged Robots,” in *IEEE International Conference on Intelligent Robots and Systems*, Dec. 2018, pp. 2707–2714, doi: 10.1109/IROS.2018.8593875.

[27] F. Grimminger *et al.*, “An Open Torque-Controlled Modular Robot Architecture for Legged Locomotion Research,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3650–3657, Apr. 2020, doi: 10.1109/LRA.2020.2976639.

[28] C. H. Dagli, *Artificial Neural Networks for Intelligent Manufacturing*. GBR: Chapman & Hall, Ltd., 1994.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, 1986, doi: 10.1038/323533a0.

[30] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[31] “Gradient -- from Wolfram MathWorld.” https://mathworld.wolfram.com/Gradient.html (accessed May 02, 2021).

[32] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[33] Mart\’{\i}n~Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.” 2015, [Online]. Available: http://tensorflow.org/.

[34] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” Dec. 2015, Accessed: May 02, 2021. [Online]. Available: https://arxiv.org/abs/1412.6980v9.

[35] F. Rosenblatt, “The Perceptron: A Perceiving and Recognizing Automaton,” Buffalo, N.Y., Jan. 1957.

[36] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines.” Accessed: May 02, 2021. [Online].

[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv*. arXiv, Jul. 19, 2017, Accessed: May 02, 2021. [Online]. Available: https://arxiv.org/abs/1707.06347v2.

[38] V. Mnih *et al.*, “Asynchronous Methods for Deep Reinforcement Learning,” *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2850–2869, Feb. 2016, Accessed: May 02, 2021. [Online]. Available: http://arxiv.org/abs/1602.01783.

[39] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust Region Policy Optimization,” *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, Feb. 2015, Accessed: May 02, 2021. [Online]. Available: http://arxiv.org/abs/1502.05477.

[40] “Kullback-Leibler Divergence Explained — Count Bayesie.” https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained (accessed May 03, 2021).

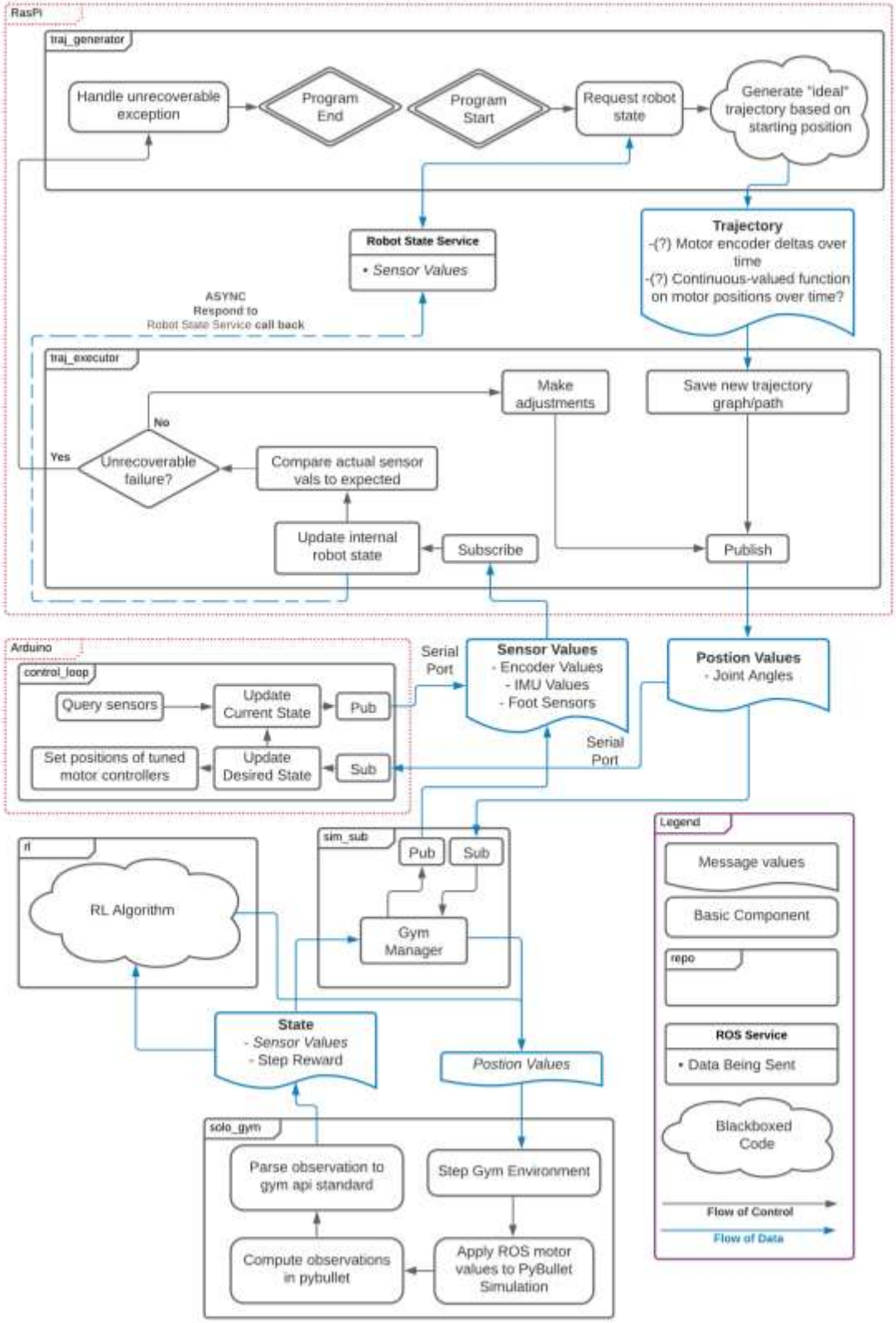
[41] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep Reinforcement Learning that Matters,” *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 3207–3214, Sep. 2017, Accessed: May 03, 2021. [Online]. Available: http://arxiv.org/abs/1709.06560.

[42] Y. Tassa *et al.*, “DeepMind Control Suite,” *arXiv*, Jan. 2018, Accessed: May 03, 2021. [Online]. Available: http://arxiv.org/abs/1801.00690.

[43] P.-W. Chou, D. Maturana, and S. Scherer, “Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution,” 2017. Accessed: May 04, 2021. [Online].

[44] “1.3.6.6.1. Normal Distribution.” https://www.itl.nist.gov/div898/handbook/eda/section3/eda3661.htm (accessed May 05, 2021).

Appendix A: Software Architecture Diagram



Appendix B: Torque Analysis

Torque Analysis

Multi-Modal Robot Locomotion
Major Qualifying Project

QB-20 [gr-qp20-mqp@wpi.edu]

October 16, 2020

1 Introduction

Before a transition gait can be implemented on the Solo8, the feasibility of such a task must first be determined. One part of the feasibility study is determining the required motor torques for stance conversion. This information will not only give us an understanding of the torques involved in stance conversion, but also help the team make informed decisions on motor specifications. The following document shows the combined dynamic and static analysis that was performed on the Solo8.

2 Assumptions

- All of the these calculations are for the Solo 8 quadruped robot.
- No friction is assumed.
- All the link weights are assumed as point masses and placed for the most conservative estimate.

3 Methodology

For the purpose of this analysis, the quadruped robot was considered as an open chain manipulator. This open chain manipulator has four joints. The Home position for the robot was determined to be as shown in figure 1. In the home position, reference frames were assigned to each joint of the robot according to the DH parameter rules. Four different cases were identified for calculating the total torque needed to hold the robot in the figure 1 position. To derive the torque values, the team created a general form of the torque equation by combining the static and dynamic torque equations. This was done because the static analysis only considers the end-effector's mass and neglects the masses of the other joints. Using the general torque equation, more conservative torque values are calculated. First we found the static torques by calculating the DH parameters for each of the four cases. In each case, the mass of the last link was placed at the presumed end-effector. The Jacobian (J(q)) was calculated for each of the cases using Matlab. The Jacobian along with the force vector were then used to calculate the static torques using equation 1.

$$\tau = J^T(q) F_{end} \tag{1}$$

Torques based on dynamic analysis were calculated using equation 2. By modifying a Matlab script that was provided by Professor Agheli, we calculated the torques for each of the cases. However, since the robot is being considered as motionless, the acceleration and velocity terms are considered to be zero and we are left with only the gravity term. When calculating the torque for all the cases during dynamic analysis, the end-effector's mass was not considered. Only masses of the other joints were considered. This was done because the end-effector's mass is already considered in the static analysis.

$$\tau = M\ddot{\theta} + C\dot{\theta} + G \tag{2}$$

To create the general equation below, we summed the static and dynamic torques. The torque values obtained are in the general form with respect to theta, angular velocity, and angular acceleration. By inputting these values, torques for any orientation can be obtained. The torque values are calculated for the home position below across four different cases. These cases vary based on the position of the presumed end-effector.

$$\tau = M\ddot{\theta} + C\dot{\theta} + G + J^T(q)F_{end} \tag{3}$$

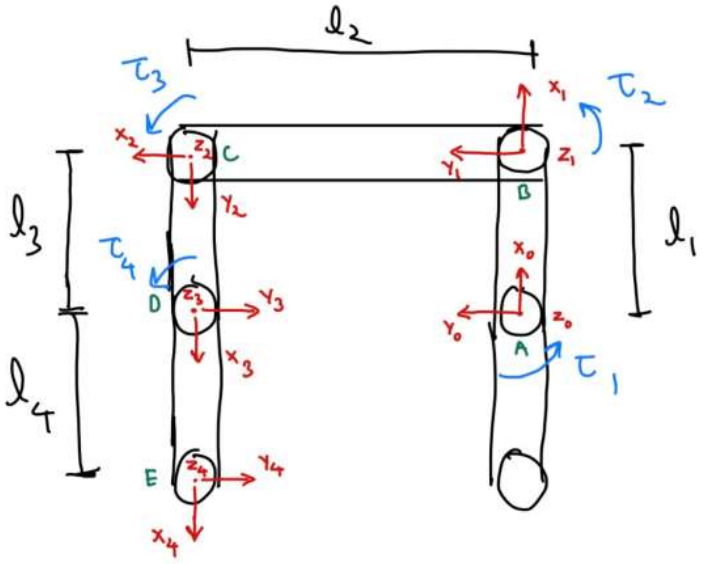


Figure 1: Solo8 in home position

3.1 Case 1

Case 1 is depicted by Figure 2. The end-effector was assumed to be at point B and was applying a force of -0.3*g N along the x axis according to the 0th reference frame. The g refers to gravity in this case. Based on this configuration, the DH parameters were calculated, as show in Table

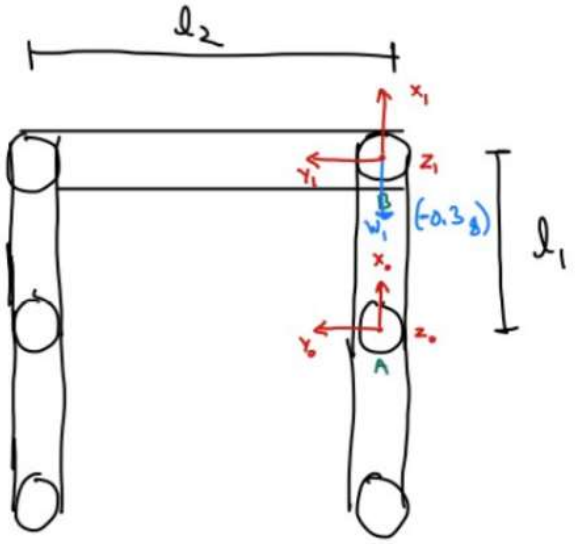


Figure 2: Reference frames and force for Case 1

1. Based on the DH parameters, the Jacobian is calculated. Using equation 1 and the Jacobian, torque for joint A is calculated. For Case 1, the dynamic analysis was not conducted because the end-effector is the only mass present on the robot in this configuration and is already considered in the static analysis.

i	a	α	d	θ
1	L_1	0	0	θ_1

Table 1: DH Parameters for case 1

3.2 Case 2

Case 2 is depicted by Figure 3. The end-effector was assumed to be at point C and was applying a force of -2.2*g N along the x axis according to the 0th reference frame. The g refers to gravity. Based on this configuration, the DH parameters were calculated, as show in Table 2. Based on the DH parameters, the Jacobian is calculated. Using equation 1 and the Jacobian, torques for joint A,B are calculated. For the dynamic analysis equation 2, the mass at point B (0.3 Kg) was considered.

3.3 Case 3

Case 3 is depicted by Figure 4. The end-effector was assumed to be at point D and was applying a force of -0.3*g N along the x axis according to the 0th reference frame. The g refers to gravity

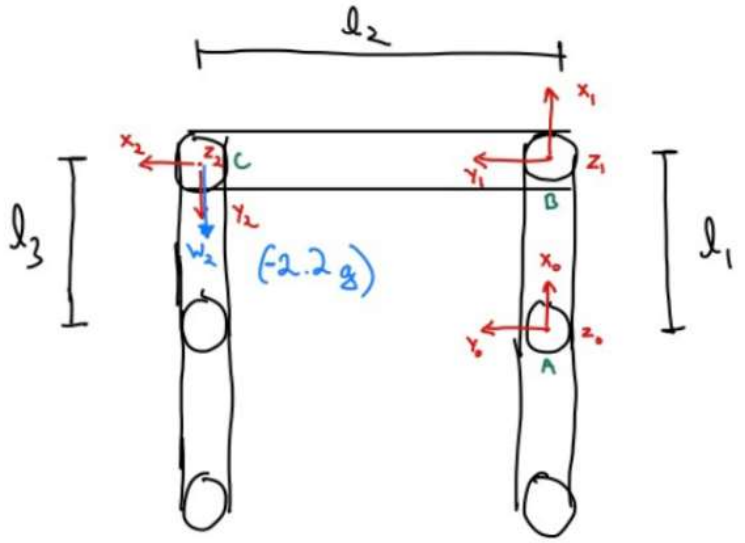


Figure 3: Reference frames and force for Case 2

i	a	α	d	θ
1	L_1	0	0	θ_1
2	L_2	0	0	$\theta_2 + \pi/2$

Table 2: DH Parameters for case 2

in this case. Based on this configuration, the DH parameters were calculated, as show in Table 3. Based on the DH parameters, the Jacobian is calculated. Using equation 1 and the Jacobian, torques for joint A,B,C are calculated. For the dynamic analysis equation 2, the mass at point B (0.3 Kg) and point C (2.2 Kg) was considered.

i	a	α	d	θ
1	L_1	0	0	θ_1
2	L_2	0	0	$\theta_2 + \pi/2$
3	L_3	0	0	$\theta_3 + \pi/2$

Table 3: DH Parameters for case 3

3.4 Case 4

Case 4 is depicted by Figure 5 . The end-effector was assumed to be at point E and was applying a force of -0.3*g N along the x axis according to the 0th reference frame. The g refers to gravity in this case. Based on this configuration, the DH parameters were calculated, as show in Table 4. Based on the DH parameters, the Jacobian is calculated. Using equation 1 and the Jacobian,

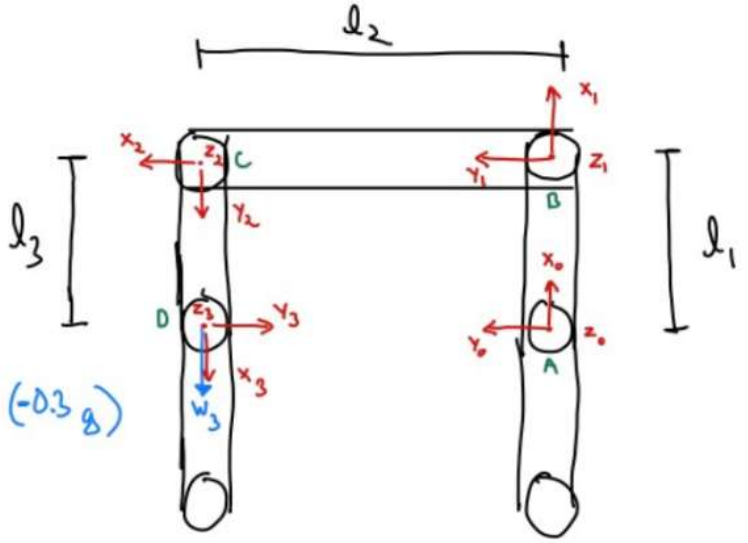


Figure 4: Reference frames and force for Case 3

torques for joint A,B,C,D are calculated. For the dynamic analysis equation 2, the mass at point B (0.3 Kg), point C (2.2 Kg), and point D (0.3 Kg) was considered.

i	a	α	d	θ
1	L_1	0	0	θ_1
2	L_2	0	0	$\theta_2 + \pi/2$
3	L_3	0	0	$\theta_3 + \pi/2$
4	L_4	0	0	θ_4

Table 4: DH Parameters for case 4

4 Results

The following results for torques were obtained by running the respective case.m script for all the cases in Matlab with q = [0,0,0,0].

4.1 Case 1

4.1.1 Static Torque

$$[\tau_1] = [0] \tag{4}$$

4.1.2 Dynamic Torque

$$[\tau_1] = [0] \tag{5}$$

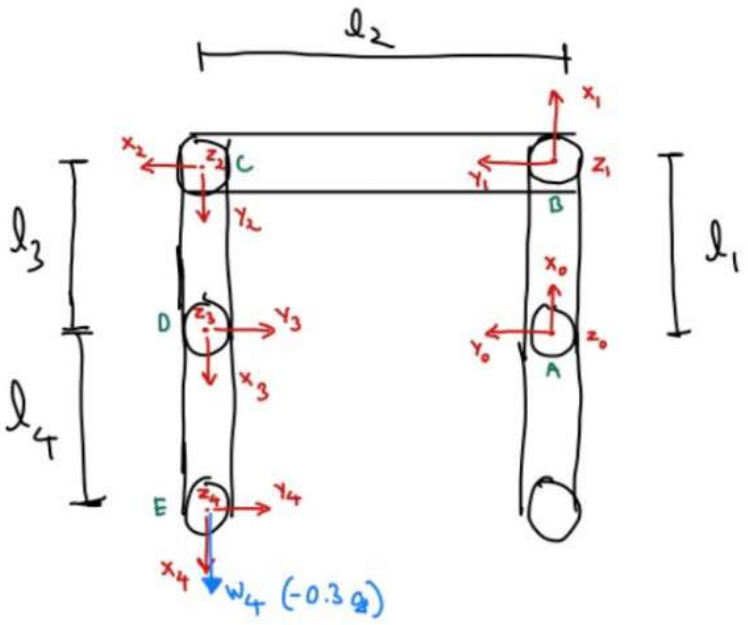


Figure 5: Reference frames and force for Case 4

4.1.3 Total Torque

$$[\tau_1] = [0] \tag{6}$$

4.2 Case 2

4.2.1 Static Torque

$$\begin{bmatrix}\tau_1 \\ \tau_2\end{bmatrix} = \begin{bmatrix}8.39Nm \\ 8.39Nm\end{bmatrix} \tag{7}$$

4.2.2 Dynamic Torque

$$[\tau_1] = [-0.47Nm] \tag{8}$$

4.2.3 Total Torque

$$\begin{bmatrix}\tau_1 \\ \tau_2\end{bmatrix} = \begin{bmatrix}7.92Nm \\ 8.39Nm\end{bmatrix} \tag{9}$$

4.3 Case 3

4.3.1 Static Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} 1.14Nm \\ 1.14Nm \\ 0Nm \end{bmatrix} \tag{10}$$

4.3.2 Dynamic Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} = \begin{bmatrix} -3.92Nm \\ 0Nm \end{bmatrix} \tag{11}$$

4.3.3 Total Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} -2.78Nm \\ 1.14Nm \\ 0Nm \end{bmatrix} \tag{12}$$

4.4 Case 4

4.4.1 Static Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{bmatrix} = \begin{bmatrix} 1.14Nm \\ 1.14Nm \\ 0Nm \\ 0Nm \end{bmatrix} \tag{13}$$

4.4.2 Dynamic Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} -3.92Nm \\ 0.47Nm \\ 0.47Nm \end{bmatrix} \tag{14}$$

4.4.3 Total Torque

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{bmatrix} = \begin{bmatrix} -2.78Nm \\ 1.61Nm \\ 0.47Nm \\ 0Nm \end{bmatrix} \tag{15}$$

5 Conclusion

Total torques are added from all four cases. The result is shown in equation 16

$$\begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{bmatrix} = \begin{bmatrix} 2.36Nm \\ 11.14Nm \\ 0.47Nm \\ 0Nm \end{bmatrix} \tag{16}$$

These torque values will inform our decisions regarding motor specification. This decision will be made in the upcoming week

Appendix C: Foot Requirements and Specifications

Rear Foot Requirements

Increases the ground surface contact area
- Some sort of flat foot

Converts from point to flat foot
- Some type of mechanism or linkage
- One-time latch release or servo operated
    - One-time latch release limits the robot's ability to convert back to quad mode
        - Could be spring loaded
        - The release could be a servo or it's somehow released by the changing angle of the leg as the robot beings to stand
    - Servo actuated increases complexity but gives the design more robustness (foot can convert back to a point foot for quad locomotion)

An axle for the toe to rotate around when converting from point to flat foot
Toe rotates around the point foot and creates a flush surface with the bottom of the point foot

Degrees of Freedom?
- If the foot is going to transform, then there needs to be at least 1 rotational DOF
- The toe/flat foot does not require any DOFs
    - The robot can stand with a rigid foot (other biped robots have rigid feet)
    - Rigid feet aren't optimal for walking but we're are not trying to implement bipedal walking in this project

Dimensions?
- Creates a large enough support polygon to prevent some angle of pitch
    - Need to determine this angle

Should the flat foot have ground contact infront or behind the robot's frontal plane (both?)?
- Choosing between infront/behind robot's frontal plane is influenced by the configuration of the legs during bipedal standing
- Each toe must rotate across a different angle depending on the configuration of the lower leg
- Both:
    - Provides more pitch stability
    - Each toe is its own part
    - Either requires two servos or the motion of one toe needs to be connected to the motion of the other
        - Integrate gear teeth into the toe design to create the desired motion
        - Offset the gear teeth on either toe so that the back toe only rotates at the end of the front toe's rotation
            - Need to determine the number of teeth

        - Small teeth probably can't be 3D printed so the gear hub needs to be large enough for the determined number of teeth
    - Design is most robust for the bipedal standing leg configuration

Materials?

- 3D printed is best because it's cheap, lightweight, and allows for easier design iteration

Servo

- Where/how does the servo attach?
    - Imbedded in the lower leg
- Needs to be small enough to fit in the leg
- Needs to produce enough torque to move the toes
    - Need to determine the servo specifications

Servo-foot connection

- Two bar linkage
- Flexible connection points between links and foot
    - Reduces the risk of damaging the servo
    - Gives the toe some flexibility
- Gears
    - provides controlled motion
    - compact

Lower leg design

- Needs to be wide enough so the servo can be imbedded into the leg body
- Could design a completely new lower leg that better compliments the flat foot
    - Keep the connection point between the lower leg and actuator module
    - Widen the leg for the servo if needed
    - Create an axle hole at the center of the point foot

Parts in the whole assembly

- Lower leg (includes point foot)
- Front toe
- Back toe
- gear
- Servo

Appendix D: Bill of Materials

Actuator Module					
Part Name	**Description**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
Motor	T-Motor Antigravity 4004 300kv	8	T-Motor	72.95	583.6
Encoder	Encoder Broadcom AEDT-9810-Z00	8	Mouser Electronics	27.9	223.2
Codewheel	625cpr, ID 7mm / OD 25,56mm	8	PWB Encoders	30.1	240.8
First Stage Timing Belt	Timing Belt Conti Synchroflex AT3 GEN III, width: 4mm/ length: 150mm/ 50 teeth	8	Belting Online	5.94	47.52
Second Stage Timing Belt	Timing Belt Conti Synchroflex AT3, width: 6mm/ length: 201mm/ 67 teeth	8	Belting Online	8.25	66
Output Shaft Bearing	6705-2RS 25x32x4mm	16	123Bearing	6.5	104
Motor Shaft and Center Shaft Bearing	MR84 Mini Ball Bearing 4x8x2mm Open	24	Bearings Direct	6.59	158.16
Timing Belt Tensioner Bearing	683-ZZ-ZEN 3x7x3mm	16	123Bearing	2.64	42.24
Timing Belt Washers	M2.5 Screw Size, ID 2,7mm, OD 5mm, 98689A111	16	McMaster-Carr	0.03	0.48
Motor Shaft	4mm x 3.99mm Stainless Steel Rod	8	KVC Engineering	24	192
Motor Pulley	AT3 T10 Aluminium 7075 Pulley	8	KVC Engineering	48	384
Center Pulley	AT3 T10 Aluminium 7075 Pulley	8	KVC Engineering	83	664
			Actuator Module Total		**2706**
Fasteners					
Part Name	**Description**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
Socket Head Cap Screw	M3x8 - Stainless Steel, 91292A112	28	McMaster-Carr	0.05	1.4
Socket Head Cap Screw	M3x12 - Stainless Steel, 91292A114	16	McMaster-Carr	0.05	0.8
Socket Head Cap Screw	M3x16 - Stainless Steel, 91292A115	16	McMaster-Carr	0.06	0.96
Socket Head Cap Screw	M2.5x6 - Stainless Steel, 91292A010	40	McMaster-Carr	0.05	2

Socket Head Cap Screw	M2.5x10 - Stainless Steel, 91292A014	23	McMaster-Carr	0.06	1.38
Flat Head Screw	M3x5 - Stainless Steel, 92125A125	32	McMaster-Carr	0.04	1.28
Flat Head Screw	M3x10 - Stainless Steel, 92125A130	16	McMaster-Carr	0.06	0.96
Flat Head Screw	M3x16 - Nylon, 92929A246	16	McMaster-Carr	0.1	1.6
Helicoil Threaded Inserts	M3x4.5 Helicoil, 91732A647	28	McMaster-Carr	0.71	19.88
Helicoil Threaded Inserts	M3x6 Helicoil, 91732A773	16	McMaster-Carr	0.9	14.4
Helicoil Threaded Inserts	M2.5x3.8 Helicoil, 91732A767	64	McMaster-Carr	0.5	32
Narrow Cheese Head Slotted Screw	M3x12 - Zinc Plated, 90657A107	4	McMaster-Carr	0.023	0.092
Low Profile Socket Head Screw	M2x8 - Alloy Steel, 93070A277	4	McMaster-Carr	1.75	7
				Fasteners Total	**76.66**
3D Printed Parts					
Part Name	**Material**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
Body Structure Brace Part 1	PLA	2	NA	0.49	0.98
Body Structure Brace Part 2	PLA	2	NA	0.48	0.96
Body Scucture Side Part 1	PLA	2	NA	1.26	2.52
Body Scucture Side Part 2	PLA	2	NA	1.26	2.52
Body Structure Front/Back	PLA	2	NA	0.5	1
Hip Module Shell Base	PLA	4	NA	1.12	4.48
Hip Module Shell Cover	PLA	4	NA	0.46	1.84
Upper Leg Module Shell Base	PLA	4	NA	1.13	4.52
Upper Leg Module Shell Cover	PLA	4	NA	0.46	1.84
Front Lower Leg	PLA	2	NA	0.48	0.96
Rear Lower Leg	PLA	2	NA	0.58	1.16
Codewheel Mount	Acurra Xtreme White 200	8	3D Hubs	10.78	86.24

Transmission Pulley AT3 T30 Center	Acurra Xtreme White 201	8	3D Hubs	20.26	162.08
Transmission Pulley AT3 T30 Output	Acurra Xtreme White 202	8	3D Hubs	23.16	185.28
Timing Belt Tensioner Roller 10mm	PLA	16	NA	0.01	0.16
Servo Gear	PLA	2	NA	0.02	0.04
Right Flat Foot	PLA	2	NA	0.1	0.2
Left Flat Foot	PLA	2	NA	0.1	0.2
				3D Printed Parts Total	**456.98**
Servo-Actuated Flat Foot					
Part Name	**Description**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
Servo	150oz-in Micro, CLS DS150CLHV	2	ProModler	54.99	109.98
Shaft	1/4"x1.5", 1327K133	2	McMaster-Carr	2.5	5
Retaining Ring	1/4" OD, 97633A130	4	McMaster-Carr	0.08	0.32
Hex Nut	M2x0.4mm thread - Zinc Plated, 90591A265	4	McMaster-Carr	0.02	0.08
Nylon-Insert Locknut	M3x0.5 thread - Steel, 90576A102	4	McMaster-Carr	0.04	0.16
Round Servo Arm	21mm OD, PDRS107	2	ProModler	1.99	3.98
				Servo-Actuated Flat Foot Total	**119.52**
Electronics					
Part Name	**Description**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
24V Power Supply	Part no. 1585717	1	Jameco	230	230
Circuit Breaker	Blue Sea Systems 285-Series 60A	1	Amazon	40.66	40.66
24V to 5V Power Converter	EPBWOPT	1	Amazon	10.79	10.79
ODrive	ODrive V3.6, 24V	4	ODrive	129	516
24V to 8.4V Power Converter	XL4015 5A DC Buck Step Down Power Converter Voltage Current & LED Voltmeter USB	1	ebay	6.95	6.95
47nF Capacitor	Bojack Ceramic Capacitor Kit	8	Amazon	0.01	0.08
Rasberry Pi		1		35	35
Teensy		1		19.95	19.95

Power Distribution Bolt	1/4in x 1-3/4in	2	Home Depot	1.99	3.98
Hex Nuts	1/4in	8	Home Depot	0.35	2.8
				Electronics Total	**866.21**
Wiring					
Part Name	**Description**	**Quantity**	**Distributor**	**Unit Cost ($)**	**Total Cost ($)**
Encoder Wires	EX Electronics Express - Hook Up Wire Kit, 22 Guage (6 Different colors), 25 ft long	1	Amazon	20.99	20.99
Motor Wires	300V AC, 20 Guage, 25 ft long 8054T14	3	McMaster-Carr	3.98	11.94
ODrive Wires	12 Guage, 25 ft long, Black Stranded CU THHN Wire 22964185	2	Home Depot	12.21	24.42
Power Supply Wires	6 Guage, 10 ft long Copper Flexible Cable Wire 307832073	2	Home Depot	20.35	40.7
Ring Terminals	Qiback Insulating Wiring Terminals Connectors Assortment Kit	1	Amazon	24.99	24.99
Dupont Pin Connectors	Proster Dupont Pin Connectors Kit	1	Amazon	24.99	24.99
Dupont Jumper Wires	Ribbon Cable Kit	1	Amazon	9.99	9.99
				Wiring Total	**158.02**

System	**Cost ($)**
Actuator Module	2706
Fasteners	76.66
3D Printed Parts	456.98
Servo Actuated Flat Foot	119.52
Electronics	866.21
Wiring	158.02
Total Cost ($)	4383.39