CS225A: Experimental Robotics
Stanford University

Final Project Report: Doggo

June 26, 2019

Victor Zhang, Simon Le Cleac'h, Laura Lee, Brian Jackson, Taylor Howell

June 2019

Contents

1 Introduction

Doggo [1] is a robotic quadruped platform designed and built by the *Stanford Robotics Club*. The aim of this robot is to be a low-cost research platform with comparable capabilities to more expensive commercial systems (e.g., Ghost Robotics' Minitaur). To date, Doggo has performed a number of simple behaviors including: walking gaits, a high jump, and a backflip. The control used for each of these demonstrations relied on position control and was performed open loop.

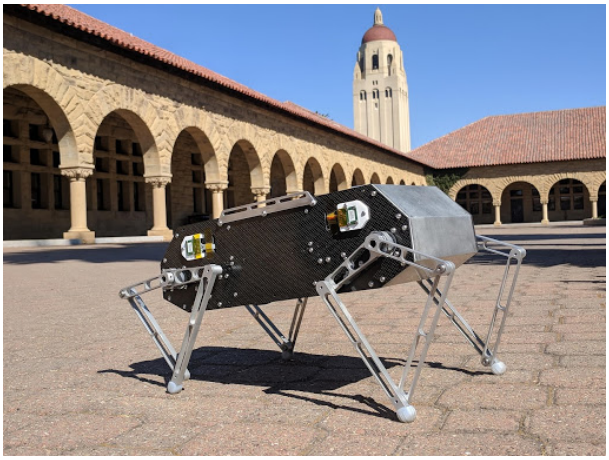


Figure 1: Robotic quadruped: Doggo

Stanford's *Robotic Exploration Lab* is exploring Doggo as a test platform for testing contact-implicit control on more dynamic tasks such as: walking on two legs, climbing stairs, and performing gymnastics-like maneuvers. For such applications, torque control is necessary to achieve closed-loop control or implementing a model-predictive control scheme. A higher level control from an onboard computer, instead of a microcontroller, is necessary to perform optimization routines and process motion capture data.

The purpose of this project is two-fold: **1. demonstrate Doggo balancing on its two back legs** and **2. advance the development of Doggo as a research platform.** To achieve these objectives we modeled Doggo at various levels of fidelity (e.g., from acrobot with correct inertia properties to a full model with closed-linkage legs and contact points). PD controllers were used to maintain desired leg configurations and upright balancing. A Linear Quadratic Regulator (LQR) controller was designed and tested to maintain body configuration. A discrete time Linear Time Invariant (LTI) Kalman filter was designed to provide state estimation, specifically for noisy angular velocity measurements. A simulator was implemented to test the model, controllers, and state estimator with a discrete controller and physical motor deadband. Radio communication was implemented for high data rate streaming of motion capture data. A C++‘ communication library was developed between the onboard ODROID computer and ODrive motor controllers. Lastly, experiments were performed to test the control stack for the balancing task.

We were able to accomplish the simplified task of balancing Doggo on half of the leg mechanism with torque control using configuration-space PD control using both motion capture data from radio and encoder data. Completion of the two objectives was not entirely successful as the team encountered many unexpected issues, primarily related to hardware. The motor drivers were particularly problematic and aspects of the mechanical design were found to be insufficient to achieve the task of balancing on complete legs. The team has recommended a number of specific improvements to the platform throughout the report to enable it to be useful as a research platform. **We conclude that the platform in its current state is not useful for research purposes and that the mechanical design is not capable of balancing on its back legs or demonstrating more dynamic behaviors.**

2 Final implementation

Here we detail our attempted solution and some of the critical infrastructure we built up to achieve our result. This section is organized by subsystem, including hardware, simulation and modeling, control and state estimation, and experiments.

2.1 Hardware

Doggo's mechanical hardware was built during Winter Quarter in collaboration between the *Stanford Robotics Club* and the *Robotic Exporation Lab*. At the beginning of the quarter the robot had been tested just a single time using the Stanford Robotics Club open loop walking gait, which ended up falling over. While the hardware was mostly complete, we made some key improvements to the hardware and the software required to control and communicate with the robot over the Spring Quarter.

2.1.1 Onboard Computation

At the beginning of the quarter, the Doggo robot was equipped with a simple TEENSY microcontroller that sent position commands to the ODrive motor controllers, running modified firmware. The TEENSY was limited to communicating with the motor controllers using UART, and was unable to easily integrate with the radio streamed MoCap data. We decided to replace the TEENSY with an ODroid XU4 to allow more flexibility and computation power for the current project and future research directions. The ODroid is running Ubuntu Mate 1.12.1. A high-level schematic of the electronics used in the project is given in Figure 2. It should be noted that since we replaced the onboard computer used by the *Stanford Robotics Club* we could no longer use the same code they used to control the robot.

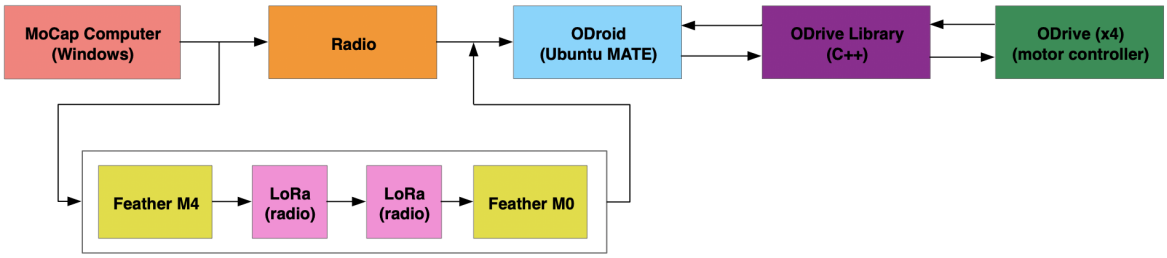


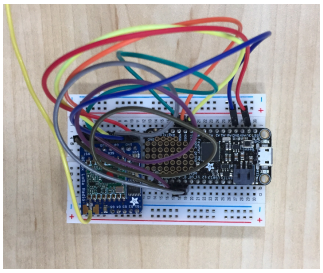
Figure 2: Schematic of electronics and communication protocols used in the project

2.1.2 Radio

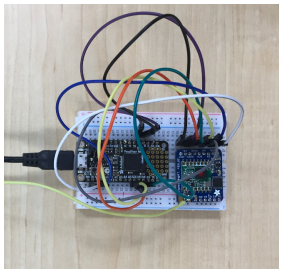
The Stanford Flight Room's streaming of motion capture data occurs over Wifi at about 100 Hz rate and 50 ms of latency which likely would not have been sufficient for the low-level control required for balancing Doggo. We built the transmitter and receiver hardware pictured in Figure 3b. The transmitter and receiver use a Feather M4/M0 microcontroller connected to a LoRa Radio. The Feather microcontrollers were connected to the MoCap computer and ODroid via USB and communicated over serial.

In addition to building the physical hardware, we wrote a C# program on the MoCap computer to interface with the *Optitrack* Motive software. This application uses the NatNetSDK to continually read tracked rigid body data and writes to the transmitting Feather microcontroller. Each packet of rigid body information consists of 32 bytes of information: 4 bytes of overhead with rigid body information, 12 bytes of world position data, and 16 bytes of rigid body quaternion data.

Experimental testing of this system indicates that the data streams at approximately 33kHz with approximately 1-2% of packets dropped. An unexpected challenge of the radio communication was the static present in the flight room. Due to the foam flooring in the flight room, static easily builds up in the room and this has been shown to interfere with the radio receiver. This static results in corrupted data packets. This was found to be fixed by resetting the microcontroller at the start of each run.



(a) Radio transmitter



(b) Radio receiver

2.1.3 ODrive Communication

Our final solution included a custom C++ library in order to interface between the ODroid onboard computer and the ODrive motor controllers, included in the project repository. It should also be noted that we reflashed the ODrive motor controllers back to the default firmware (instead of the Stanford Robotics Club's custom firmware). More details about this decision are provided below in Section 3.1.

2.1.4 Onboard Circuitry

We made a few minor tweaks to the onboard hardware in addition to those mentioned previously, namely an improved (more reliable) kill switch and power relay.

2.1.5 Motor Characterization

Doggo does not have direct torque control, which necessitates using open loop current control instead. In order to convert from torque to current, we characterized the motor torque/current curve, seen in Figure 4. We found that the current deadband was 0.0241A and the conversion factor is 39.5388 A/Nm.

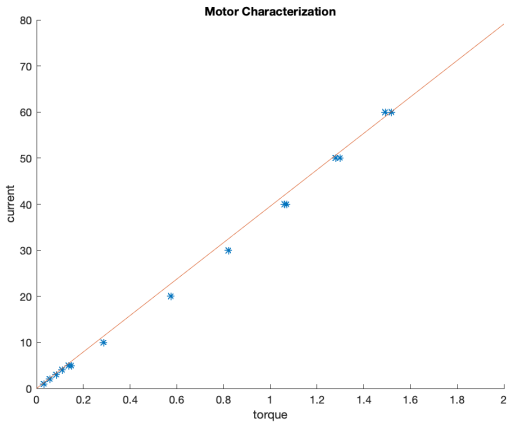


Figure 4: Motor torque-current curve.

2.2 Modeling and Simulation

Prior to this project, no dynamics model—even simplified—existed for Doggo. We developed a number of models ranging in fidelity from an acrobot (i.e., underactuated double pendulum) with correct inertial properties to a complete Doggo model with each leg modeled as a four-bar linkage and experiencing contact forces from the ground. Our final solution where we balanced on half of the legs didn't use a dynamics model (since it used just a simple PD controller), and our attempted solution at balancing with the full leg used the model shown in Figure 5b.

2.2.1 Model Parameters

The model parameters used for our models are included in the URDF file(s) we wrote for the project, which are included in the project repository. The geometry and inertial properties for each link were taken from CAD models.

2.2.2 Simplified Model

The first model of Doggo used for the LQR controller was a double and triple pendulum model. In this model Doggo was represented by doubling the leg mass and inertia properties and using this as the first and third links with the body as the middle link. This 3 DOF model consists of three parameters: ϕ, θ, and ψ, where ϕ is the angle between the ground and the first link, θ is the angle between the first leg and the body, and ψ is the angle between the body and the top leg. During testing this model was simplified further down to a double pendulum because we only had use of half of the motors, which we used to control the bottom two legs.

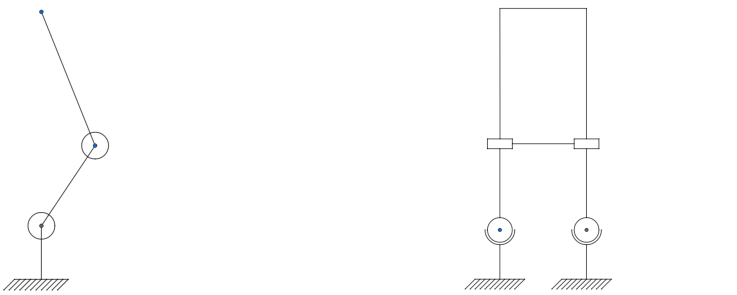
This double pendulum model was used for the PD balance using the half leg mechanisms. However, when the full legs were assembled, this model presented problems in that it could not successfully correct for twist between the two legs.

Our final solution to balancing Doggo on its full hind legs modeled the robot as two inverted pendulums connected to the body. This 3 DOF model is parameterized by three angles θ_1, θ_2 and ϕ where θ_1 is the angle between the left leg and the body, θ_2 is the angle between the right leg and the body, and ϕ is the angle between the body and the vertical world axis. While this model is actually not physically realizable (since the feet should be modeled as ball joints instead of pin joints), it captures the general behavior to at least first order, which is all we needed for the LQR controller. In this model we assumed massless legs (which is fairly accurate given how light the legs are compared to the body). The forward kinematics for the body center of mass are:

$$\begin{aligned} x &= \ell \sin(\psi) + h \sin(\phi) \\ y &= 0 \\ z &= \ell \cos(\psi) + h \cos(\phi) \\ \omega_x &= \frac{\ell}{w}(\dot{\theta}_1 - \dot{\theta}_2) \\ \omega_y &= \dot{\phi} \\ \omega_z &= 0 \end{aligned} \tag{1}$$

where ℓ is the length of the leg, $2w$ is the width of the body, h is distance to the center of mass of the body from the leg joint, and $\psi = \phi - \frac{1}{2}(\theta_1 + \theta_2)$.

Given the forward kinematics, we computed the dynamics using the first and second derivatives of the forward kinematics and the Euler-Lagrange equation.



(a) Double pendulum (b) Double Pendulum with Twist

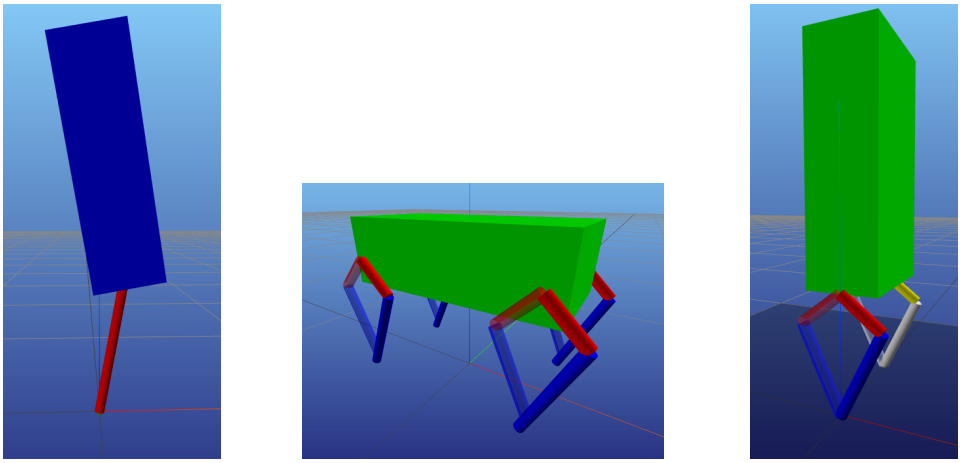
Figure 5: Simplified models of Doggo

2.2.3 Simulation Model

We used `RigidBodyDynamics.jl` [2] and `MeshCatMechanisms.jl` [3] for our simulation and visualization. RigidBodyDynamics allows for URDF parsing so we developed a few different models of varying degrees of fidelity. Our most basic simulation model was just the simple

double pendulum model. For this model we assumed some given angle between the upper links of the legs and used a nonlinear solver to solve for the combined inertial properties of the four bar mechanism, and then doubled the values to capture the mass properties of both legs. This model worked very consistently in our simulations and we never really had issues getting it to balance.

After attending ICRA 2019 and getting some valuable information from the authors of these packages, we were able to construct models that captured the closed-bar linkages as well as contact, which was the most physically realistic simulation we developed. We also made a slightly simplified version of this model where we pinned the feet to the ground and removed the arms.



(a) Double pendulum (b) Full model with contact (c) Pinned without arms

Figure 6: Simulation models of Doggo

2.3 Control and State Estimation

The overall control architecture is given in Figure 7. At a basic level, we used the sensor information from the encoders and the motion capture system to feed into a state estimator that returned the states for our simplified model (see Section 2.2.2). These estimated states were fed into a controller that returned 4 motor torques. Prior to being sent to the robot, these motor torques were passed through a "safety controller" that checked for torque limits, current limits, and leg angle limits.

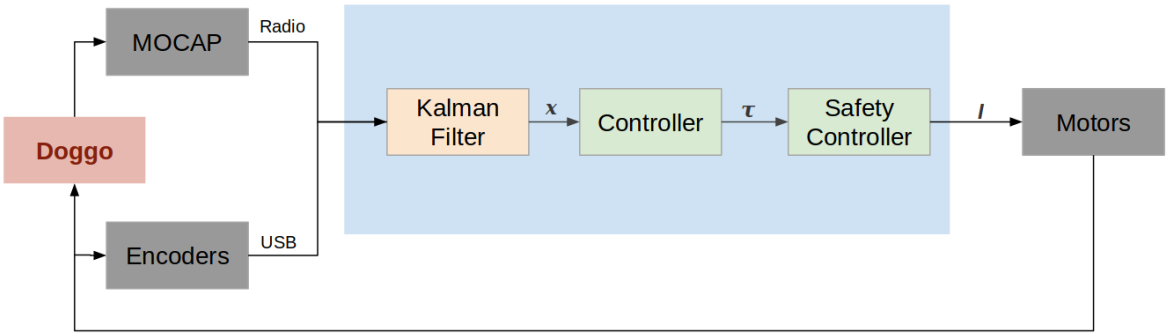


Figure 7: Control architecture

2.3.1 Leg Configuration Controller

Our initial approach was to model the robot as a simple double pendulum (See Figure 5a). This model made the assumption that both bottom legs acted as a single rigid body. To make this assumption valid we used high-gain PD control to keep make maintain the height of each leg separately, and make the legs track each other.

These simple control laws can be written down as:

$$\tau_{len} = \begin{bmatrix} K_{p,len}\alpha_1 + K_{d,len}\dot{\gamma}_1 \\ -K_{p,len}\alpha_1 - K_{d,len}\dot{\gamma}_1 \\ K_{p,len}\alpha_2 + K_{d,len}\dot{\gamma}_2 \\ -K_{p,len}\alpha_2 - K_{d,len}\dot{\gamma}_2 \end{bmatrix}, \quad \tau_{\text{twist}} = \begin{bmatrix} -K_{p,\text{twist}}\beta - K_{d,\text{twist}}\dot{\beta}_1 \\ 0 \\ -K_{p,\text{twist}}\beta - K_{d,\text{twist}}\dot{\beta}_2 \\ 0 \end{bmatrix} \tag{2}$$

where $\alpha_i = (\gamma_{i2} - \gamma_{i1})$ is the difference in the two encoder values (radians) of each leg, $\beta = \gamma_{21} - \gamma_{11}$ is the angular difference between axis 1 of each leg, and $\gamma_{ij}$is the encoder value for the jth axis on leg i. The proportional and derivative gains for the length and twist controllers were (0.85, 0.01) and (0.6, 0.005), respectively. A video of the two PD controllers can be found here: PD Controller.

2.3.2 Half-leg balance

To achieve the balancing we demonstrated in video we used the leg configuration controller described in the previous section along with a simple PD controller to drive the angle of the body with respect to the world to zero:

$$\tau_{\text{body}} = -K_p\phi - K_d\dot{\phi} \tag{3}$$

where ϕ is the angle between the body and the world vertical, which is provided by the motion capture system. This controller used gains of: $K_p = 0.4$ and $K_d = 0.002$.

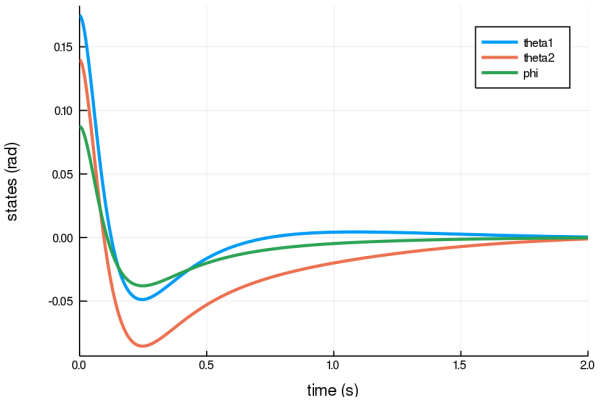


Figure 8: Response of the simplified model to initial conditions

2.3.3 Full Body Balance

In our first attempt to achieve full body balancing we used the leg configuration controller described previously along with an LQR controller for the simple acrobot model. However, this did not work very well so we instead used the model described in Section 2.2.2 and used an LQR controller for the whole model after physically pinning the legs together and only controlling one of the two motors on each leg.

To calculate the LQR feedback gain matrix K we linearized the dynamics model we created about the equilibrium position of $x_{eq} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$ to get a linear model with matrices

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 125 & 125 & -33 & 0 & 0 & 0 \\ 125 & 125 & -33 & 0 & 0 & 0 \\ 64 & 64 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 41 & 39 \\ 39 & 41 \\ 22 & 22 \end{bmatrix}$$

We picked $Q = \text{Diagonal}(\begin{bmatrix} 1000 & 1 & 1 & 1 & 1 & 1 \end{bmatrix})$ and $R = \text{Diagonal}(\begin{bmatrix} 100 & 100 \end{bmatrix})$ and calculated the feedback gain matrix using a continuous time Riccati Equation to get

$$K = \begin{bmatrix} 85 & 84 & -289 & 19 & 18 & -67 \\ 106 & 109 & -368 & 23 & 25 & -86 \end{bmatrix},$$

The response of this controller without any sensor noise can be seen in Figure 8

2.3.4 Kalman Filter

The sensor measurements of the system consists of the MoCap data of Doggo's body angle with respect to the world, four encoder positions, and four encoder angular velocities. The MoCap data and encoders both have noise in their sensor measurements. Due to the available sensor measurements and their respective noise, a Kalman filter was necessary to both estimate the state of the system as well as filter the noise.

To get the discrete time dynamic matrices, we used the above continuous time matrices and approximated the discrete time dynamics by:

$$\begin{aligned} A_t &= e^{A\delta t} \\ B_t &= B\delta t \end{aligned} \tag{4}$$

With the given measurements and state, we have the following measurement model:

$$y = \begin{bmatrix} \phi \\ \theta_1 \\ \theta_2 \\ \dot{\theta_1} \\ \dot{\theta_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} x \tag{5}$$

The process noise was estimated by weighting the uncertain states more highly. Due to uncertainty in the dynamics between our simplified model and the physical robot, we chose uncertainty weights to be:

$$Q_t = \text{Diagonal}(\begin{bmatrix} 0.00001 & 0.00001 & 0.00001 & 0.0001 & 0.0001 & 0.0001 \end{bmatrix}) \tag{6}$$

The measurement noise was characterized using logged experimental sensor data:

$$R_t = \text{Diagonal}(\begin{bmatrix} 0.0016 & 0.001 & 0.001 & 0.01 & 0.01 \end{bmatrix}) \tag{7}$$

Using the above model, we can estimate the state at the next time step by propagating through the dynamics:

$$\begin{aligned} \hat{x}_{t|t-1} &= A_t x_{t-1|t-1} + B_t u_t \\ \Sigma_{t|t-1} &= A_t \Sigma_{t-|t-1} A_t^T + Q_t \end{aligned} \tag{8}$$

The state can then be updated using the most recent sensor measurements:

$$\begin{aligned} S &= C\Sigma_{t|t-1}C^T + R_t \\ K &= \Sigma_{t|t-1}C^T S^{-1} \\ x_{t|t} &= \hat{x}_{t|t-1} + K(y_t - C\hat{x}_{t|t-1}) \\ \Sigma_{t|t} &= (I - KC)\Sigma_{t|t-1} \end{aligned} \tag{9}$$

The state estimator was tested in simulation in the LQR controller with the results seen in Figure 9.

3 Challenges

We encountered a significant number of challenges in this project, both expected and unexpected. The following sections briefly outline several of the challenges we faced as well as our attempt to address them.

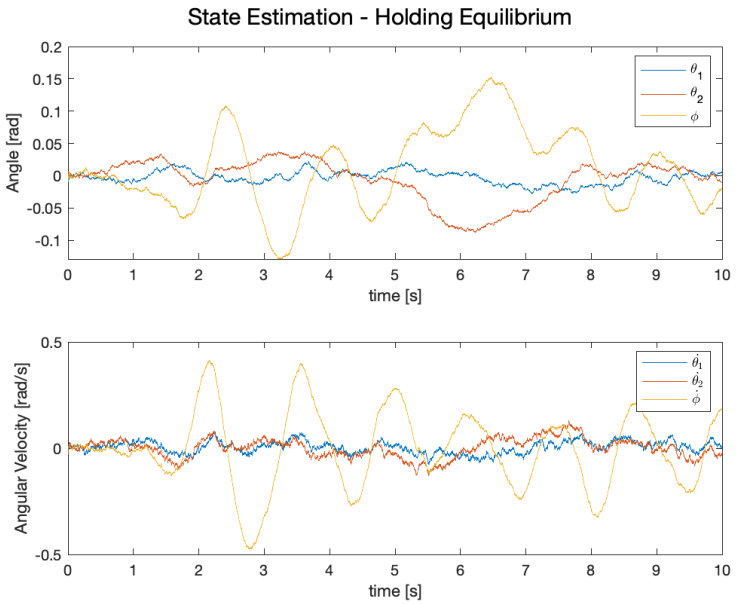


Figure 9: Simulation of Kalman filter with LQR controller holding equilibrium.

3.1 ODrive motor controllers

Nearly all of our challenges arose from the ODrive motor controllers. These motor controllers were selected by the S*tanford Robotics Club* because they are open source, relatively inexpensive, not too large, and meet the relatively high voltage and current requirements of the DC brushless motors.

One of the earliest challenges was getting the ODrives to communicate with the ODroid at all. We ended up having to recompile and re-flash the stock firmware onto the ODrives to get communication working, since the Robotics Club had installed custom firmware that altered the USB communication interface.

After getting the communication issues sorted out, we burned out a few ODrives and 2 motors while just trying to control the motors using the Python API. We believe the motors burned out due to failed ODrive circuitry since the motors burned out on the same axis on the same ODrive. It should be noted that the ODrives burned out without us sending any commands to the motors: they were supposed to be holding their current position.

When we finally got the motor controllers working with the ODroid via Python, we found we could not set very high gains on the PD controllers without the controllers going unstable. After diagnosis we found the controller was running at less than 100 Hz, so we were very likely running into the Nyquist frequency. So we developed a C++ interface with the microcontrollers on the ODrive motor controllers to read and write to a few critical registries we need to send current commands. Our custom library was based off of a previous community written library for ODrive v3.4 and earlier. However, we use ODrive v3.5 boards, which utilize a completely new communication protocol. In order to characterize this new communication protocol, we reverse engineered the Python API using *WireShark* to sniff the packets sent between the computer and the boards. This sped up the controller to about 200-300 Hz.

Our lab plans on continuing investigation into this issue and figuring out a way to speed up the communication between the ODroid and ODrive motor controllers. It appears that our ODrive library has hit a known time limit with communication to the motor controllers. Each packet sent between the ODrive and ODroid takes approximately $500\mu s$ and we need to send and receive at least 20 packets to read encoder values and send current commands. In order to increase the communication rate, we will need to change the firmware running on the ODrives,

and possibly the communication protocol running on the motor controllers.

3.2 Unstable PD Controllers

Our initial approach to this problem was to model the robot as a double pendulum, which rested on the assumption that the bottom legs acted like a single rigid body. We operated under this assumption for nearly the entire project, since it seemed very reasonable to implement the leg configuration controller described in Section 2.3.1 with high gains. However, as mentioned in the previous section, we could not set the gains high enough to get reasonably rigid behavior from the legs. It took our group a while to diagnose the communication speed issues mentioned in the previous section, as well as figure out how to reliably send current commands, since the direction convention of the ODrives was unclear and often swapped from the encoder direction and inconsistent between motors. We eventually found good ways of characterizing these directions by sending test commands to each motor.

We were also getting very spiky velocity data from the encoders, which caused the legs to go unstable whenever there was a large spike in encoder velocity. We addressed this issue by writing the Kalman filter described in Section 2.3.4 and using outlier detection and elimination to remove extraneous velocity readings. This made a dramatic difference and made the system behave much more reliably, although this solution came very late in the quarter since we had originally assumed the encoders would be giving us good velocity estimates.

In the end, we found that we could not get the PD gains high enough to really justify our modeling assumption so switched from the simple double pendulum model (Figure 5a) to the one described in Section 2.2.2 (Figure 5b) so that we could remove the rigid body assumption and allow the LQR controller to relax the separation requirements between the poles for the balancing and leg configuration controllers by modeling the joint dynamics.

3.3 Drivetrain

We also believe the quality of the hardware used in the drivetrain is significantly lacking. When trying to run the LQR controller (or even basic PD controllers) one of the legs would oscillate extremely quickly. This could be due to poor encoder data as mentioned previously, but also could be due to large amounts of backlash in the drivetrain. We had to remove some components fairly early on since the runout between concentric axes was so bad it was causing enormous amounts of friction when spinning the motors. We believe the drivetrain is not precise enough to get the fairly fine motor commands required to balance a very top-heavy double pendulum.

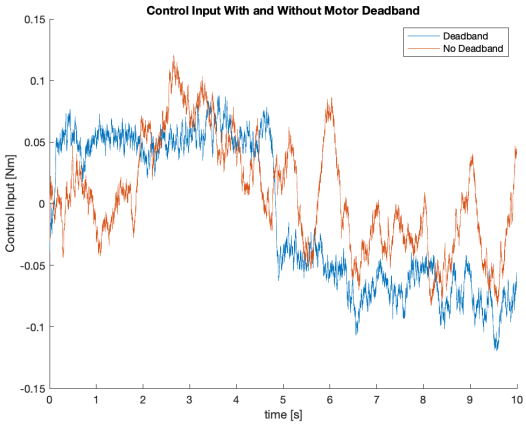


Figure 10: LQR control input simulated with and without control input deadband.

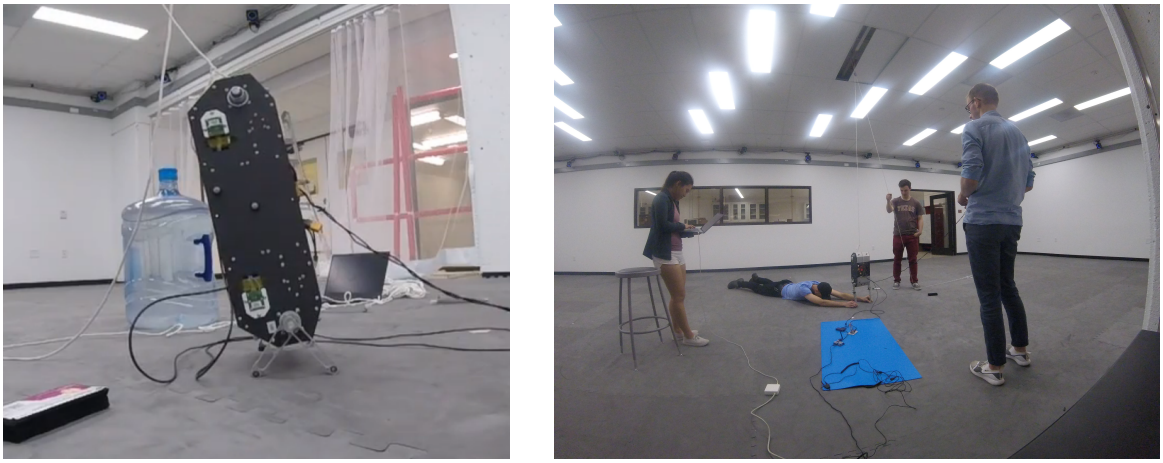
To better understand the effects of the drivetrain backlash, we simulated the controller with a deadband on the control input, as seen in Figure 10. These results matched what we saw with the large oscillations about zero in the control.

3.4 Lacking Simulation Options

We decided to not use SAI2 as our simulation engine since it didn't seem to support the closed-loop linkages used for the legs. We decided to build upon the infrastructure in our lab which is all written in Julia [4], and leverage the excellent packages `RigidBodyDynamics.jl` [2] and

`MechCatMechanisms.jl` [3] developed by students in Russ Tedrake's lab at MIT. However, it wasn't until fairly late in the quarter (after ICRA 2019) that we figured out how to actually use these packages to model closed loop mechanisms and contact. Prior to this we spent a decent amount of time developing an analytical solution for the dynamics of this linkage, which we ended up not using in our final solution. It was also a significant learning curve trying to figure out how to use these packages to get realistic simulations that included the whole control loop. In hindsight maybe we should have looked more into using SAI2 as our simulator.

4 Results and Conclusion



(a) Balancing experiment

(b) Twist acrobot experiment

Figure 11: Experimental results

While this project didn't meet the goals we set out to fulfill, our group learned *a lot* about experimental robotics. We had to develop an enormous amount of infrastructure during the course of the project, everything from a good simulation platform to basic communication between hardware. We learned very well how all the pieces need to come together to get a working robot. One of the most valuable lessons we learned was that of starting with a good simulation. We put this off for probably way too long and we would have benefited greatly from doing this much earlier. We also learned it is important to characterize all parts of your system before you begin trying to do anything with it. We probably would have saved a lot of time by really digging into the communication rates early on, as well as looking at the raw output from our sensors. We also learned that sampling rates are very important. A huge portion of our time was spent on getting our communication rates fast enough, both over radio as well as between the ODroid and ODrive motor controllers. We also learned the value of having reliable, quality hardware to do closed-loop torque/current control. The hardware really got in the way of this project and a huge majority of our time was spent on seemingly trivial aspects of just getting the hardware to work the way we expected it should.

Last but not least, we learned that things get much more complex when you move to hardware, and things that are seemingly simple in simulation can be extremely trickly to implement in hardware. Russ Tedrake, a professor at MIT, said about their double pendulum hardware: "it was like the dragon we needed to slay in the lab for a number of years." This type of highly-nonlinear and under-actuated system is difficult to control in practice, especially with inexpensive, uncharacterized hardware such as the Doggo robot.

5 Videos

PD Controller: https://drive.google.com/file/d/1JNbgxBfiRLaNEut2EqkxRCJLq4ERB$_Z$d/view?usp = sharing
PD Balance using MoCap: https://drive.google.com/file/d/1Np3gOS0$_G$DamimV4Aa88jd8kyPCxmbVc/view?usp = sharing
LQR Attempt: https://drive.google.com/file/d/1183ysgfH40LPutPpHulOGaDKDbEG$_6$TV/view?usp = sharing
Doggo Falls: https://drive.google.com/file/d/1egZzoxAYkJmfQSdEAVa1p12rYxCWifr$_/$view?usp = sharing

References

[1] N. Kau, A. Schultz, N. Ferrante, and P. Slade, "Stanford doggo: An open-source, quasi-direct-drive quadruped," *arXiv preprint arXiv:1905.04254*, 2019.

[2] T. Koolen and contributors, *RigidBodyDynamics.jl*, 2016.

[3] R. Deits and contributors, *MeshCatMechansisms.jl*, 2016.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.