Washington University in St. Louis

Washington University Open Scholarship

Mechanical Engineering and Materials Science Independent Study

Mechanical Engineering & Materials Science

5-18-2021

Quadruped Pupper Robotics: Dynamics and Control

Daniel Shen
Washington University in St. Louis

Isaac Sasser
Washington University in St. Louis

Aaron Manuel
Washington University in St. Louis

Tom Kang
Washington University in St. Louis

Kenny Huang
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/mems500

Recommended Citation

Shen, Daniel; Sasser, Isaac; Manuel, Aaron; Kang, Tom; and Huang, Kenny, "Quadruped Pupper Robotics: Dynamics and Control" (2021). *Mechanical Engineering and Materials Science Independent Study*. 150.
https://openscholarship.wustl.edu/mems500/150

This Final Report is brought to you for free and open access by the Mechanical Engineering & Materials Science at Washington University Open Scholarship. It has been accepted for inclusion in Mechanical Engineering and Materials Science Independent Study by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

Washington University in St. Louis

JAMES MCKELVEY SCHOOL OF ENGINEERING

Quadruped Pupper Robotics: Dynamics and Control

Independent Study Faculty Sponsor: Louis Woodhams

Independent Study Sponsor: Stuart Bowers

Independent Study Sponsor: Lee Redden

Report Submission Date: Wednesday, May 12, 2021

We hereby certify that the lab report herein is our original academic work, completed in accordance with the McKelvey School of Engineering and Undergraduate Student academic integrity policies, and submitted to fulfill the requirements of this assignment:

Isaac Sasser

Aaron Manuel

Daniel Shen

Kenny Huang

Tom Kang

Contents

Contents 1

List of Figures 1

List of Tables 3

ABSTRACT 4

INTRODUCTION 4

THEORY 5

METHODS 13

RESULTS & DISCUSSION 19

CONCLUSION 33

References 34

Appendix A Robot Startup Guide 35

Appendix B Software Installation Guides 36

Appendix C Hardware Build Notes 40

Appendix D Recommended Reading and Helpful Resources 41

Appendix E Simulator Guide 42

List of Figures

1 Transformation matrix definition. 5

2 Glossary for Transformation Matrix [4]. 6

3 3D Transformation Matrix Example [5]. 7

4 Transformation Matrix ${}^{0}_{1}T$ schematic. . . . 8
5 Transformation Matrix ${}^{0}_{1}T$ result. . . . 9
6 Transformation Matrix ${}^{1}_{2}T$ result. . . . 10
7 Simulation result for the quadruped model without the leg loading feedback at constant acceleration [6]. . . . 11
8 Simulation results for the quadruped model with the leg loading feedback during acceleration [6]. . . . 11
9 Diagram of the rhythmic activities of the flexor half-centers of the four legs and three types of postures in the stance phase of the two legs in phase of each diagonal pair during trotting [6]. . . . 12
10 The nine emerged gaits that result from the possible combination of two elements out of the pair [6]. . . . 12
11 Block diagram for the control of the Robot. . . . 17
12 Circuit diagram of the power drawing elements in the PCB. . . . 18
13 Flow chart that shows the CAN connections on the board. . . . 18
14 Picture of the activated robot standing (front). . . . 20
15 Picture of the activated robot standing (side). . . . 20
16 Picture showing the keyboard connection to the robot. . . . 21
17 Picture of the robot tilting upward. . . . 22
18 Picture of the robot tilting downward. . . . 22
19 Picture of the robot rolling left. . . . 23
20 Picture of the robot rolling right. . . . 23
21 Controller Modes 25
22 Mappings Between Modes 26
23 Robot Postures at Different Zs 27
a $z = -0.02$ 27
b $z = -0.05$ 27
c $z = -0.09$ 27
d $z = -0.15$ 27

e $z = -0.22$ . . . 27

24 Complete Motion of A Normal Hop . . . 28

a REST . . . 28

b HOP . . . 28

c FINISHHOP . . . 28

d Falling . . . 28

e Landing . . . 28

25 Robot Postures at Different Xs . . . 29

a $x = -0.20$ . . . 29

b $x = -0.05$ . . . 29

c $x = +0.05$ . . . 29

26 HOP Mode Parameters Summary . . . 30

27 FINISHHOP Mode Parameters Summary . . . 31

28 BALANCE Mode Parameters Summary . . . 31

29 Complete Motion of A Jump . . . 32

a Rest (HOP) . . . 32

b Pushing (FINISHHOP) . . . 32

c Jumping (FINISHHOP) . . . 32

d Falling (FINISHHOP) . . . 32

e Landing (FINISHHOP) . . . 32

List of Tables

1 Table of tools and equipment used to construct the robot. . . . 15

2 List of challenges and solutions discovered during the build process. . . . 16

ABSTRACT

The purpose of this project is to provide insights on the Pupper Robot, from Hands-On Robotics (handsonrobotics.org), for future studies and research. The Hands-On Robotics (HOR) team aims to provide robotics kits and educational curricula to explore agile locomotion, motor control, and AI for community colleges and high schools. We worked with the HOR team in this project to help them better achieve their goals. The main objectives of this project include: 1. Build the robot and analyze the dynamical behaviors of the robot. 2. Investigate the robot control from both hardware and software perspectives. 3. Design a new gait for the Pupper Robot. 4. Create an implementation guide for future groups, documenting knowledge we have learned during the project. By the end of this project, we achieved the following: A. Built a fully functioning robot. B. Investigated the theoretical underpinnings of quadruped robots, including inverse kinematics and gait generation theories. C. Understood and reflected on the control structure of the robot. D. Implemented a new jumping gait which allows the robot to leap forward and land on balance. E. Composed detailed guides on robot building instructions, controller files installation, simulator installation, and simulator modifications.

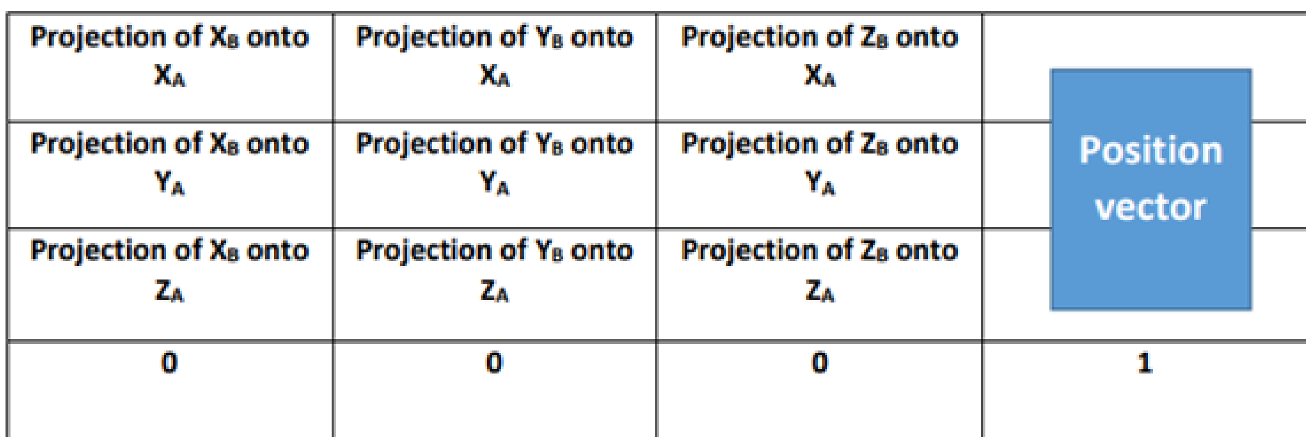
INTRODUCTION

This project was focused on the construction and testing of a quadruped robot. Quadruped robotics offer a unique ability to experience and learn a number of interesting topics such as control systems, gait control and analysis, and a number of hardware and software applications such as circuits and coding. As a result, the quadruped robot is an interesting educational possibility; however, often the advances in this area are very localized and many variables change at the same time. This makes it challenging to isolate advances or find a performance benchmark. Additionally, quadruped robots are often very expensive. The Stanford Pupper was designed to be a low-cost open source option for quadruped testing, promoting fast iteration and information sharing [1].

The goal of this project was to build the robot and analyze the dynamics of the robot as well as robot control. This involved analyzing gait control as well as developing new gaits for the robot. Furthermore, to promote information sharing on this topic, step-by-step guides to building the robot as well as troubleshooting common problems that were encountered during our work are also found within this report.

THEORY

Quadruped Robot Kinematics. Quadruped robots are very complex in structure and are harder to control than wheeled and crawler robots [2]. In order to achieve and maintain a certain posture, each of the four legs of the robot is often performing a kinematic behavior that is different from the other legs. Kinematic models are very important tools that help with the stability analysis and trajectory planning of the robot. The basic idea of the kinematic model is to describe the motion of a system of linked bodies. In this application, the kinematic model helps define the motion of our robot, namely the motion of each joint, leg, and the main body. For a quadruped robot, there is a main body with 4 legs. With 3 joint motors on each leg, the total degrees of freedom is 12. Although the legs and the body are connected, it would be difficult to model them as a whole system because the motor on each joint can provide rotary movement only subject to the particular joint. For example, if we are trying to model our robot in a 3D coordinate system, and we set the hip joint of leg 1 to be our origin, then the 3 other legs' motion are still undefined because their movements are centered around their own joints. In fact, since there are 12 joints, each of them is connected to a part of the leg, so all parts are actually doing some movements subjective to only their connected joints. Technically, we describe each of them to be in their own frames when modeling. To properly model the movement, from one inertial frame to another, we need a tool called the transformation matrix, as defined in Fig. 1.



Projection of X_B onto X_A	Projection of Y_B onto X_A	Projection of Z_B onto X_A	Position vector
Projection of X_B onto Y_A	Projection of Y_B onto Y_A	Projection of Z_B onto Y_A	
Projection of X_B onto Z_A	Projection of Y_B onto Z_A	Projection of Z_B onto Z_A	
0	0	0	1

Figure 1 Transformation matrix definition.

This form of transformation matrix shown in Fig. 1 is called the affine transformation matrix.

The affine transformation matrix can represent any linear transformation [3]. For any n dimensional object, the affine transformation matrix will be an (n+1) by (n+1) matrix. Thus, the matrix in Fig. 1 is used to transform an object between two 3D frames. The glossary for transformation matrix configurations in 2D is shown in Fig. 2.

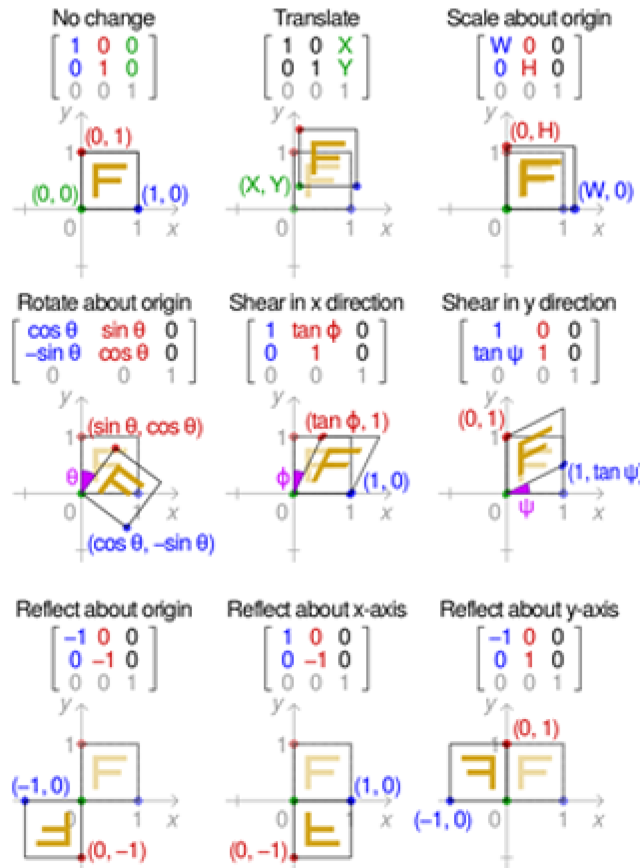
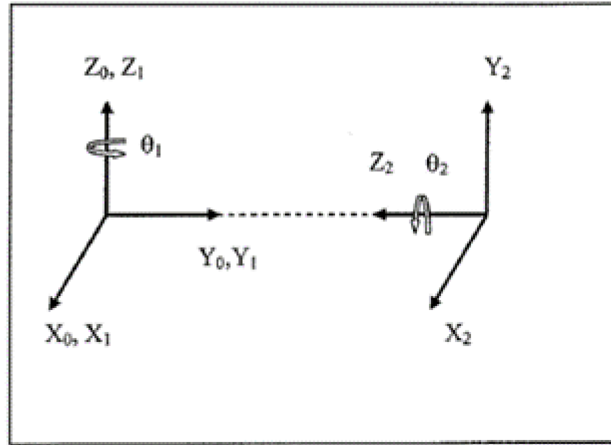


Figure 2 Glossary for Transformation Matrix [4].

Figure 3 is an example of 3D transformation matrices. This example is important because this concept is essential in order to understand the inverse kinematics of the robot.



$$
{}^{0}_{1}T = \begin{bmatrix} Cos(\theta_1) & -Sin(\theta_1) & 0 & 0 \\ Sin(\theta_1) & Cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{1}_{2}T = \begin{bmatrix} Cos(\theta_2) & -Sin(\theta_2) & 0 & 0 \\ 0 & 0 & -1 & L \\ Sin(\theta_2) & Cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Figure 3 3D Transformation Matrix Example [5].

In this example, we have 3 frames: FRAME0 is the normal Cartesian coordinates in 3D; FRAME1 shares the same origin and Z-axis with FRAME0, but with some rotation around the Z axis; FRAME2 is quite different than the first two frames, and the z-axis of FRAME2 is inline with y-axis of FRAME0. It's obvious that FRAME0 and FRAME2 can be used to model a part of the robot leg with two joints. ${}^{0}_{1}T$ is the matrix that transforms objects in FRAME1 to objects in FRAME0. Similarly, ${}^{1}_{2}T$ is the matrix that transforms objects in FRAME2 to objects in FRAME1. And by matrix manipulation, we can also have a matrix transform objects in FRAME2 to FRAME0, that matrix ${}^{0}_{2}T$ can be calculated by ${}^{0}_{1}T \cdot {}^{1}_{2}T$ (matrix product). The form of the transformation matrices are shown in Fig. 1 [5]. We will break down the two matrices by parts and discuss their compositions.

${}^0_1T$ **Matrix**:

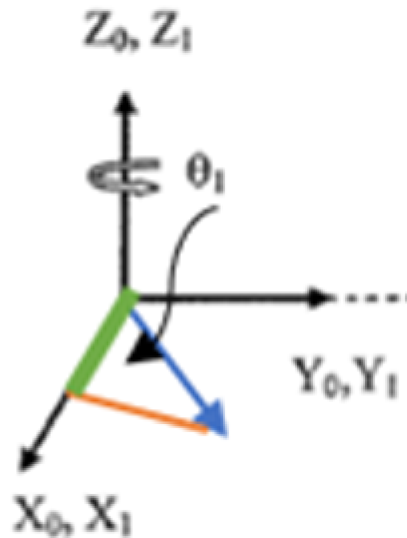


Figure 4 Transformation Matrix ${}^0_1T$ schematic.

We can first conclude that the position vector should be [0, 0, 0] because FRAME0 and FRAME1 share the same origin. In other words, the distance between the two origin (of FRAME0 and FRAME1) is zero, so the position vector is the zero vector.

Since FRAME1 uses the same z-axis as FRAME0, we can also expect the third column and third row to be [0, 0, 1] (Fig. 5). No matter what θ is, the x and y axis are always going to be perpendicular to z-axis, so the red vectors are always zero. The Z-axis of the two frames is identical, so the purple number is going to be 1, independent from θ.

To find the projection of the x and y axis, we can degrade the system to a 2D system. Similar to the "Rotate about origin" case in Fig. 2, the xy plane of this example is also a rotating model. As shown Fig. 4, consider the blue arrow being the x-axis of FRAME1. We can see how it is rotating around the z-axis for a degree of θ from the x-axis of FRAME0. The projection of X_1 on X_0 then, is the green vector marked in Fig. 4. Note that the orange segment and the green segment should be perpendicular. Since the green part can be found by cos(θ) times the blue vector, we have determined the first entry of the matrix to be cos(θ). All other entries can be found using the same logic. And the results are marked green.

The last row is always [0, 0, 0, 1] for an affine transformation matrix by convention. The result

for ${}^{0}_{1}T$ is shown in Fig. 5.

$$
{}^{0}_{1}T = \begin{bmatrix} Cos(\theta_1) & -Sin(\theta_1) & 0 & 0 \\ Sin(\theta_1) & Cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Figure 5 Transformation Matrix ${}^{0}_{1}T$ result.

${}^{1}_{2}T$ **Matrix**:

we can first conclude that the position vector should be [0, L, 0] where L is the distance between two origins. Note that the L is in the second place because the origin of FRAME2 is "departed" from the origin of FRAME1 in the y-axis (of FRAME1) direction.

Note that the z-axis of FRAME2 is inline with y-axis of FRAME1, but in opposite direction, we have the purple being -1. Since the xy plane is perpendicular to the z-axis, we can conclude that the xy plane of FRAME2 is also perpendicular to the y-axis of FRAME1. Similarly, since the xz-plane is perpendicular to y-axis, we can also state that the xz plane of FRAME1 is perpendicular to z-axis of FRAME2. Hence we have those red vectors being 0.

The green vectors can be found using the same idea that we discussed in previous matrix. Note that the direction of θ in this matrix is different from in the previous previous example.

The last row is always [0, 0, 0, 1] for an affine transformation matrix by convention. The result for ${}^{0}_{1}T$ is shown in Fig. 6.

$$
{}^{1}_{2}T = \begin{bmatrix} Cos(\theta_2) & -Sin(\theta_2) & 0 & 0 \\ 0 & 0 & -1 & L \\ Sin(\theta_2) & Cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Figure 6 Transformation Matrix ${}^{1}_{2}T$ result.

This example provides a holistic view of how to transform different inertial frames into one base frame through transformation matrix.

Gait Generation. Gaits are the patterns of movement that describe how animals control their leg swings and stance. Animals select their gaits based on speed. Studies show that the reason of gait change includes energetic, durable, biomechanical, environmental and morphometric considerations. In the study done by Yasuhiro eta, they used a central pattern generator(CPG) as a control system to determine the motion of the swing and stance of the quadruped robots' legs [6]. Initially, the CPG network was always hard-wired and the lateral neighboring CPG models were mutually and inhibitorily coupled. This method produced the most basic gait-trot in which the diagonal pairs of legs move in-phase and the other pair of legs move out of phase. The concept of trot can be showed in the following figure. The footfalls describe the duration of stance for each leg and it is controlled by the extensor. The footfall data shows the coupled diagonal legs' movement for the trot gait. The swing phase of the gait is controled by the robot's flexor which can lift the feet. The body pitch tilt is effected by speed of the robot.

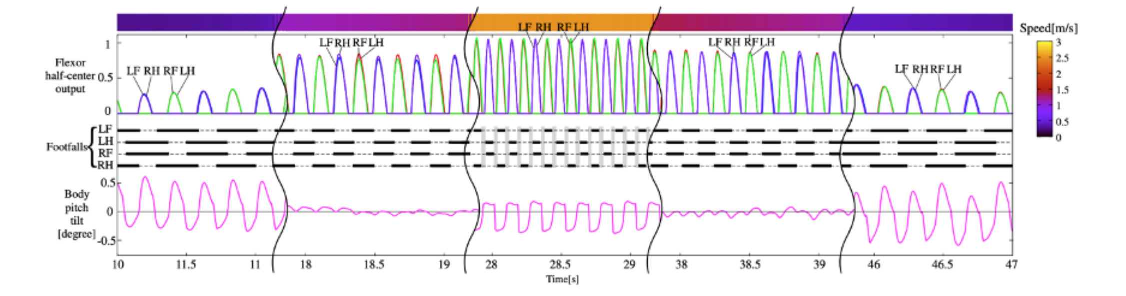


Figure 7 Simulation result for the quadruped model without the leg loading feedback at constant acceleration [6].

Based on the CPG system, the foot loading sensor is equipped at the ankle area for each leg which feeds back the foot loading. The flexor, which controls the swing and the extensor which controls stance mutually inhibits one another. The load-sensitive receptors in the ankle extensor motors inhibit the flexor. While the leg is loaded, the extensor is excited because of the inhibition of the flexor which results in prolongation of stance duration and prevention of initiation of the swing phase. Based on the foot loading feedback, each step would be adjusted according to body tilt which generated many different gaits. The following figure shows a constantly accelerated robot with different gaits generated.

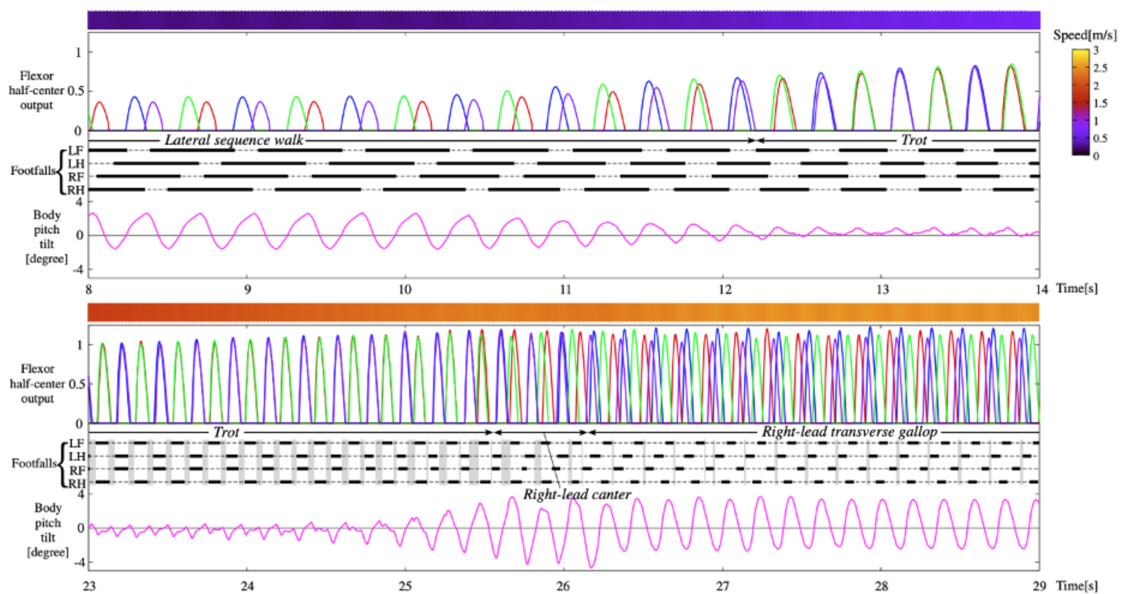


Figure 8 Simulation results for the quadruped model with the leg loading feedback during acceleration [6].

The principle of gait generation is related to the diagonal pair of legs moving out of phase. The following picture shows the different directions of out-phasing causing different body tilt. The green curve is the trot which is when the diagonal pairs are in phase.

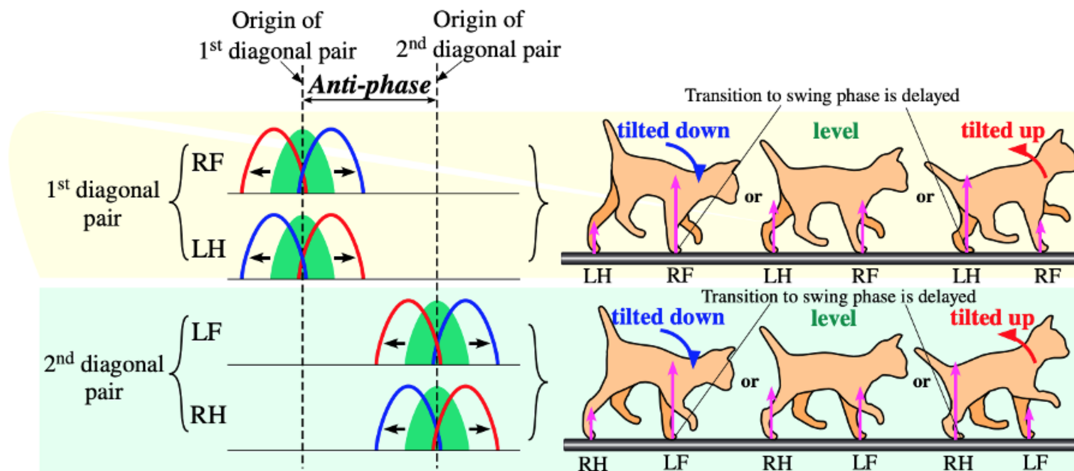


Figure 9 Diagram of the rhythmic activities of the flexor half-centers of the four legs and three types of postures in the stance phase of the two legs in phase of each diagonal pair during trotting [6].

Each pair of diagonal legs can cause different body posture, which means 9 combinations of gait can be generated based on this principle. The following figure shows the combination of diagonal pair phasing condition with its corresponding gaits' name.

1st diagonal pair of RF and LH	2nd diagonal pair of LF and RH	Emerged Gait
tilted up (RF to LH)	tilted up (LF to RH)	L-walk
	tilted down (RH to LF)	RT-gallop
	level (LF = RH)	R-canter
ilted down (LH to RF)	tilted up (LF to RH)	LT-gallop
	tilted down (RH to LF)	D-walk
	level (LF = RH)	unusual canter
level (RF = LH)	tilted up (LF to RH)	L-canter
	tilted down (RH to LF)	unusual canter
	level (LF = RH)	trot

Figure 10 The nine emerged gaits that result from the possible combination of two elements out of the pair [6].

METHODS

Software Control Implementation.

Note: This document refers to the "Stanford Quadruped" controller repository.

From the control perspective, the Stanford Pupper can be broken into three sections: joystick interface, controller, and motor interface. In the joystick interface section, the command is given to the robot through a PS4 joystick or keyboard joystick emulator. The joystick interface reads the signal sent from the joystick. The controller part includes a gait scheduler to plan the gait for the motion. The gait is achieved by two helper methods: swing and stance. The swing method makes the leg move to the next location and the stance method makes the leg stay on the ground. Through the combination of "swing" and "stance", a specific gait can be implemented. The "swing" and "stance" methods are controlled by the foot coordinates in the body frame, and an inverse kinematics method is used to convert the Cartesian coordinate of the desired location to a set of joint angles. This array of joint angles is passed to the motor interface. The motor interface reads the joint angles array from the controller section and converts it to PWM (Pulse-width modulation) duty cycle and sends the signals to the motor controllers (Teensy board).

The joystick interface (JoystickInterface.py) reads the signal from the PS4 joystick. In this method, both discrete and continuous commands can be read and passed to the controller. Discrete commands include activation and deactivation, as well as changing mode between resting, trotting, and hopping. Continuous commands include moving in x and y direction and changing pitch angle, height, and rolling. The output of this function is a command object that records x and y velocity, yaw rate, height, pitch, roll, and which gait mode the robot is executing.

The main controller (Controller.py) reads the command object passed from the joystick interface and uses the swing and stance method to carry out the movement. Three helper methods used by the main controller are gait controller (Gaits.py), swing controller (SwingLegController.py), and stance controller (StanceController.py). The gait controller breaks the continuous movement to phases and defines the motion as a cyclic motion for each individual leg. The swing controller simulates the movement of a specific leg. It calculates the desired Cartesian coordinates from the command object and returns the final foot location. The stance controller simulates the contact between a specific leg

and the ground. In stance mode, the leg is not moving if the reference frame is the ground. However, the leg moves backward if the reference frame is the robot. For example, if in the stance phase the feet move backwards at -0.4m/s (to achieve a body velocity of +0.4 m/s) and the stance phase is 0.5 seconds long, then the feet will have moved backwards -0.20 m. Inside the main controller, three modes, including trot, hop, and rest, are defined. The main controller calls the gait controller to determine which legs are moving and which legs are staying, and calls the gait controller and the swing controller to calculate the coordinates of the next foot location. Inverse kinematic (Kinematics.py) is used to convert the coordinates into a set of joint angles. These joint angles and the coordinates will be passed to the motor interface.

The motor interface (HardwareInterface.py) passes commands from the main controller to individual motors thorough the motor controller, Teensy board. This is achieved by the communication through serial port. The serial port is a two-way communication channel between the Teensy board and the main controller's Raspberry Pi board. The Teensy board sends error messages and feedback to the Raspberry Pi board while the Raspberry Pi board sends commands to Teensy board to control the motors. The parameters that can be sent to the motor includes the activation/deactivation command, PD control parameters (K_p and K_d), and maximum current. The main controller can send two types of command to the motor: the actuator position (joint angles) and Cartesian position (Cartesian coordinates). The PD control inside the motor controller (Teensy board) moves the leg to the set of desired joint angles or the desired Cartesian coordinates.

The main method (run_djipupper.py) is a while true loop that keeps reading from the joystick interface. The user can activate or deactivate the robot here. If the robot is in the activated mode, the main method will repeatedly read from the joystick, run the controller to find the desired joint angles and cartesian coordinates, and pass the command to the motor interface until a keyboard interruption is entered to stop the while true loop.

The objects that are used to record data are Config, State, and Command. These objects are mainly used to store data and to pass along data between different phases and methods. The Config object sets the maximum value and geometric constraints. The State object records the parameters and behavior mode at the current state. The Command object records the desired motion received from the joystick. The Config object is invariant while the State and Command objects change when the

robot moves and when a command is given to the robot, respectively.

Hardware and Construction.

Initial Setup and Preparation. This section contains information about how the team prepared for the build. Upon arrival the robot kit would ideally contain all the hardware needed for assembly. However, in our case there were some hardware pieces missing. In addition, some required assembly tools may also come with the kit; for example, our kit included a crimping set but not a soldering iron.

The team gathered the tools required to complete the build. Some of these were gathered at the start, while others were acquired as they were needed. The list of tools that were used is found in Table 1.

Table 1 Table of tools and equipment used to construct the robot.

Tool	**Additional Information**
Needle Nose Pliers	
Wire Cutter	
Wire Stripper	
Crimping Set	If not provided
Soldering Iron	Make sure it's strong enough (>60 W).
Solder	
Heat Shrink Tubing	For splicing wires
Metric and Imperial Allen Wrenches	
Extra Wire	
Multimeter	

After the kit arrived, the first step was itemizing the kit. The kit was compared against the bill of materials (BOM) to determine if any items were missing. In our case, we were missing 4 "clamping D-hubs," 2 "left lower links," and 1 "left hip," along with some M3x8 flat head screws.

Build Process. Once the initial steps were complete, the hardware itemized and the tools collected, the team began the build process. There were two steps that should be done before beginning the build process: shortening the controller wires and crimping the JST connectors onto the controller wires. The wires were shorted to about 6 cm by cutting them and splicing them back together. The cleanest way to do this was to desolder the connection on the controller, shorten the wire and re-solder the wires. Each of the controllers came with an additional heat shrink for this purpose. However, this was not the approach taken as the team's soldering iron at the time was under-powered for the task. Therefore, the wires were cut in the middle and shortened then spliced back together.

During the build process the team ran into a number of challenges. These challenges and the solutions taken are listed in Table 2

Table 2 List of challenges and solutions discovered during the build process.

Challenge	**Solution**
Couldn't melt solder	More powerful soldering iron
Shorting of the PCB	Use a multimeter to check for shorts before powering on the robot
Wire Crimping	Develop crimping guide
Screws not biting the Nylon shrouds	Really push on the screws for the first few turns
Legs over or under rotating	Screw orientation matters. Make sure to check the guide for D-hub screw location
Teensy not working when plugged into the shield	Teensy shield may be shorting the Teensy. Check all pins with multimeter
Power the Teensy when running a Raspberry Pi	Run a 5V regulator to the Pi and power the Teensy through the USB on the Pi.

Software Setup. Figure 11 shows a block diagram of the control flow for the robot.

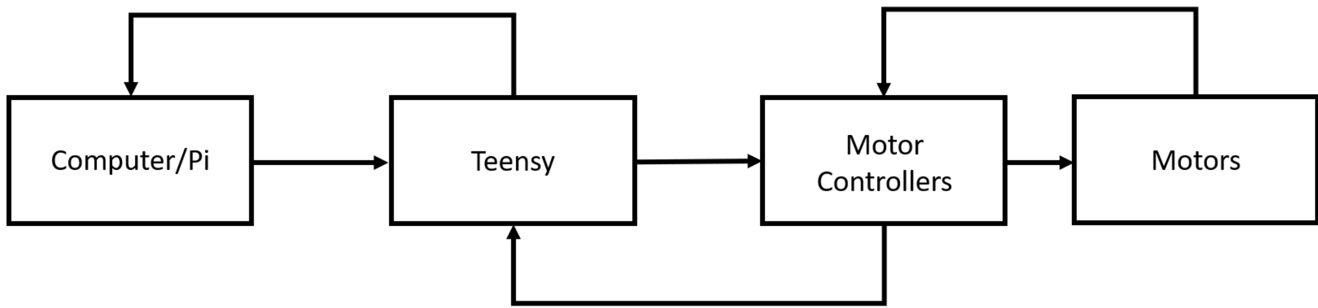


Figure 11 Block diagram for the control of the Robot.

As shown in the figure, the commands from the computer are sent to the Teensy, from the Teensy to the motor controllers, and finally from the motor controllers to the motors. Additionally, the computer receives feedback from the Teensy about the motors.

Both the Pi and the Teensy were loaded with software developed specifically for the control of the DJI Pupper robot. The Teensy was "flashed" with code designed to interpret the signals sent by the Pi and provide useful information back to the Pi. The Raspberry Pi was loaded with two code repositories: the Pupper control code and the joystick emulator. The control code contained the main functions necessary to control the robot, while the joystick emulator allowed the use of a standard keyboard as controller, by translating certain key inputs to equivalent buttons on a PS4 controller.

Software installation proved to be more difficult and time-consuming than expected, presenting several unforeseen problems and setbacks. While solving these issues, we developed a detailed and comprehensive software setup guide to supplement the current Pupper build documentation. This new guide expanded on areas left vague in the original instructions, hopefully enabling a smoother installation experience for those without prior robotics experience. Software setup guides and a few general build tips can be found in the Appendix of this paper.

Circuitry. The foundation of the robot is a printed circuit board (PCB). This board is the core for how information and power flow throughout the robot. Figure 12 shows a circuit diagram of a rough recreation of the power elements in the PCB.

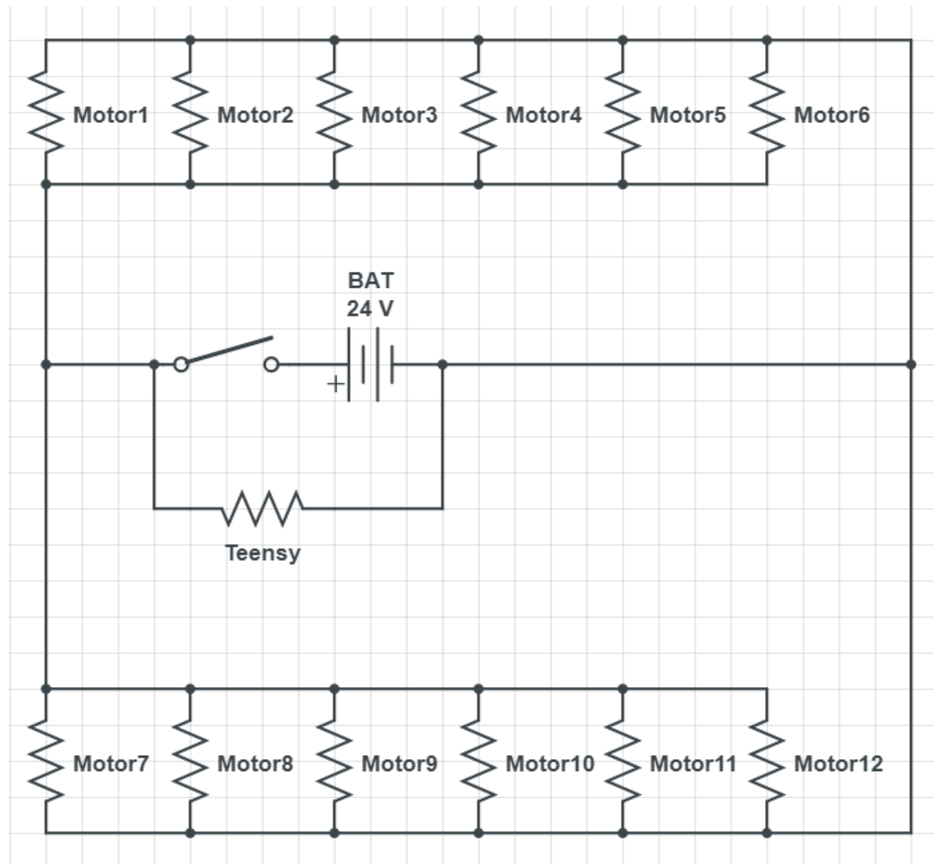


Figure 12 Circuit diagram of the power drawing elements in the PCB.

In this case, the power outputs are modeled as resistors. Each of the power plugs runs in parallel to the battery such that each receives 24 V. Additionally, the Teensy also receives 24 V, which must be stepped down to 5 V with a regulator. However, the Teensy was not connected to the battery and is drawing its power from the Raspberry Pi, thus the 5V regulator is not necessary. After fixing the short on the Teensy shield, plugging the Teensy into the battery caused the Teensy to stop working. As a result, the spare Teensy was used and not connected to the battery but powered through the Raspberry Pi.

The diagram for the CAN communication in Fig 13.

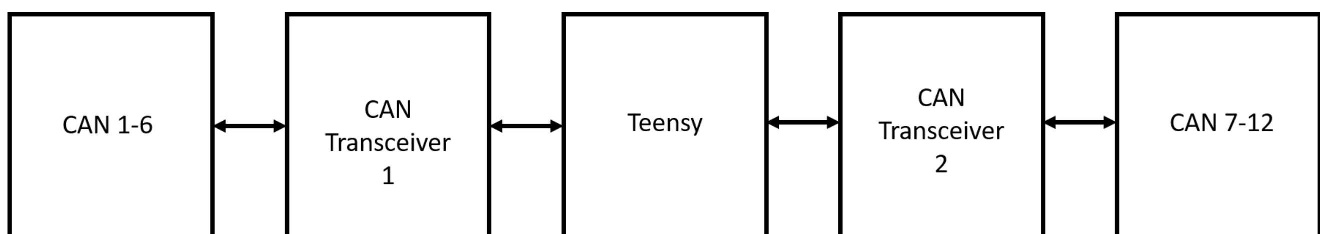


Figure 13 Flow chart that shows the CAN connections on the board.

On each side of the robot the CAN connections run in parallel. Each side connects to a central CAN transceiver at the center of the PCB, both of which then connect to the Teensy and send feedback and receive commands from the Teensy.

RESULTS & DISCUSSION

Physical Construction of the Robot. The construction of the robot was ultimately successful, though not exactly in the expected way. Since a mistake in the manufacturing of the Teensy shield led to the Teensy being unable to receive commands while drawing power from the battery, we improvised by chaining power to the Teensy through the Pi. In the completed robot, the battery was connected to the 5V regulator, which regulated power to the Pi; the Teensy was powered through its USB connection with the Pi. Though not what the creators of Pupper intended, this way of powering the hardware works just fine.

In the final stages of construction, the robot was configured to enable it to go mobile. This involved attaching the battery and the Pi to the robot's back. The battery was secured using electrical tape, while the Pi was encased in a 3d-printed Raspberry Pi 4 case, which was then fastened to the robot's back using one of the PCB screws. Photos of the standing robot are shown below in Figs. 14 and 15.

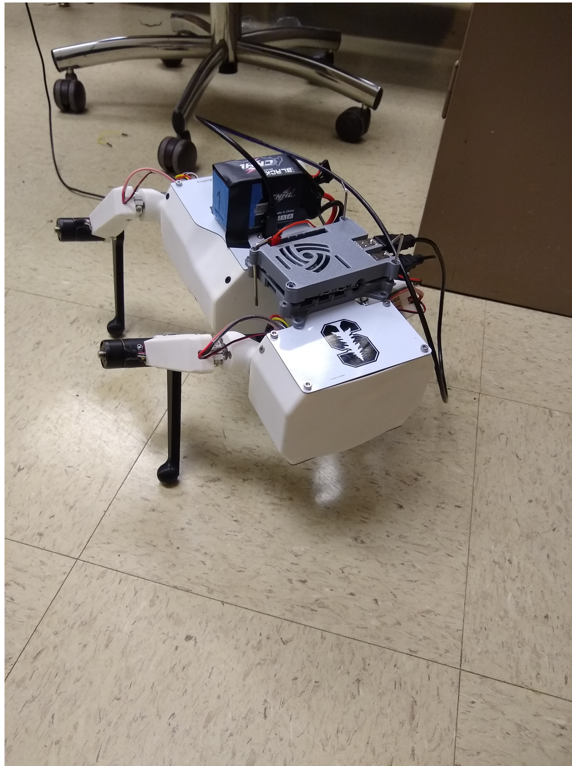


Figure 14 Picture of the activated robot standing (front).

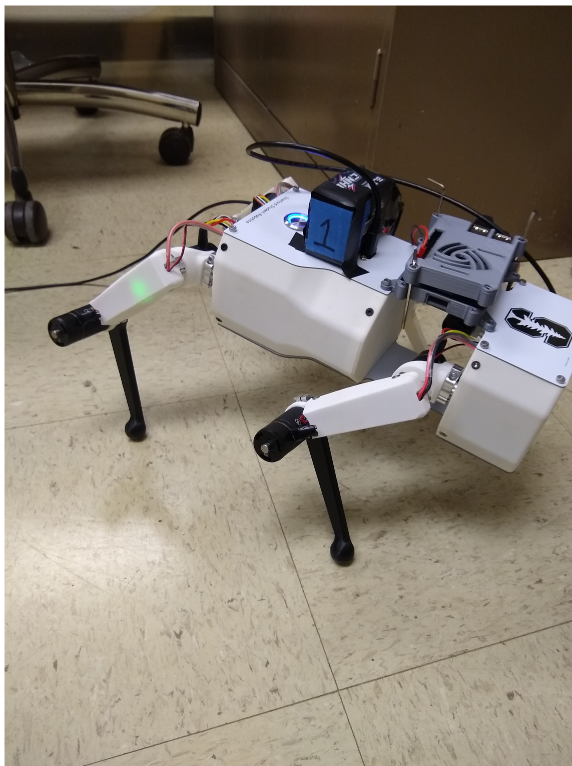


Figure 15 Picture of the activated robot standing (side).

These methods of securing the battery and the Pi work for brief stints of movement, but the energetic shaking of the robot during its trotting gait causes the battery and Pi to come loose somewhat quickly. Future teams should find a more permanent solution. The battery should have enough room to rest inside the robot's main compartment if the wires are packed in just right, but we were unable to find an appropriate configuration to allow for this. As for the Pi, a dedicated case should be designed with holes aligned to the locations of all 4 PCB screws at the robot midsection. This would allow the Pi to be secured using four screws instead of just one, preventing it from rotating during operation of the robot.

The finished robot was controlled using a traditional USB keyboard rather than the intended PS4 controller. While connected to the Pi via USB, the PS4 controller touchpad and mouse feature worked as expected. However, none of the controller buttons registered on the Pi, without or without the keyboard joystick interface running. This problem should be investigated in future work. Figure 16 below shows the keyboard-to-robot connection.

Figure 16 Picture showing the keyboard connection to the robot.

With the keyboard controller, we achieved all the out-of-the-box functionality of Pupper.

Figures 17, 18, 19, and 20 show the robot's tilting and rolling functions in action.

Figure 17 Picture of the robot tilting upward.

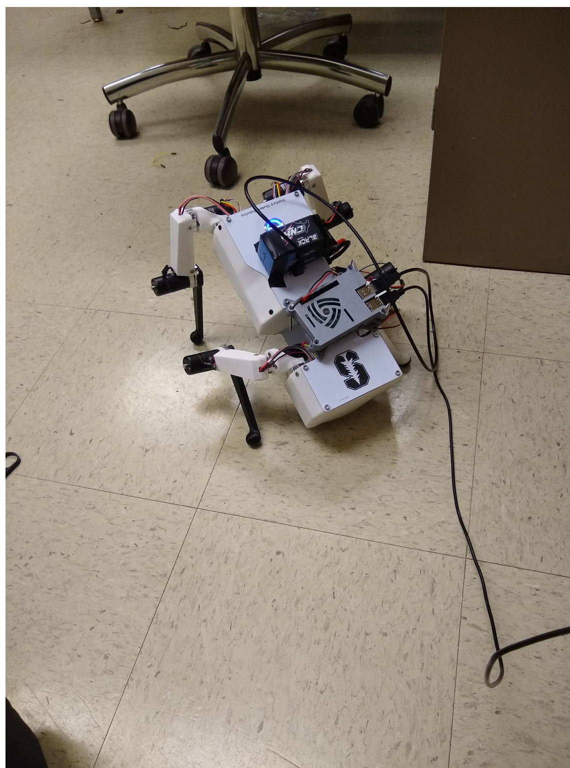


Figure 18 Picture of the robot tilting downward.

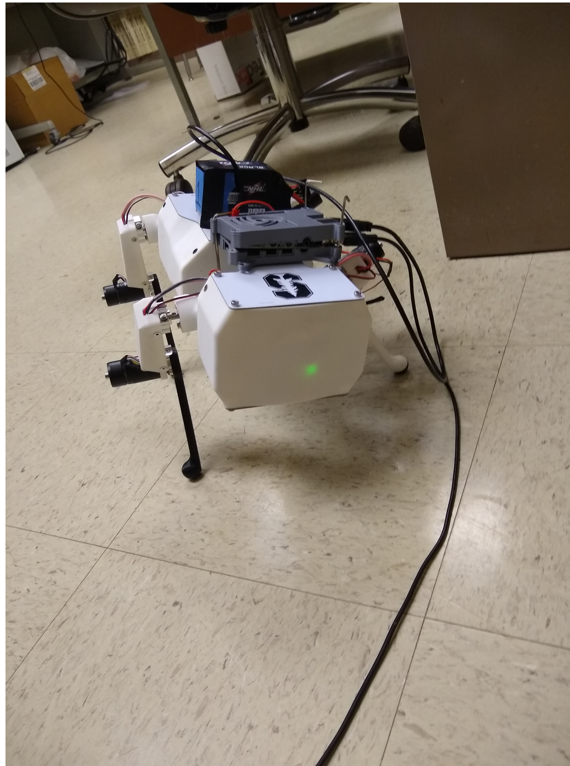


Figure 19 Picture of the robot rolling left.

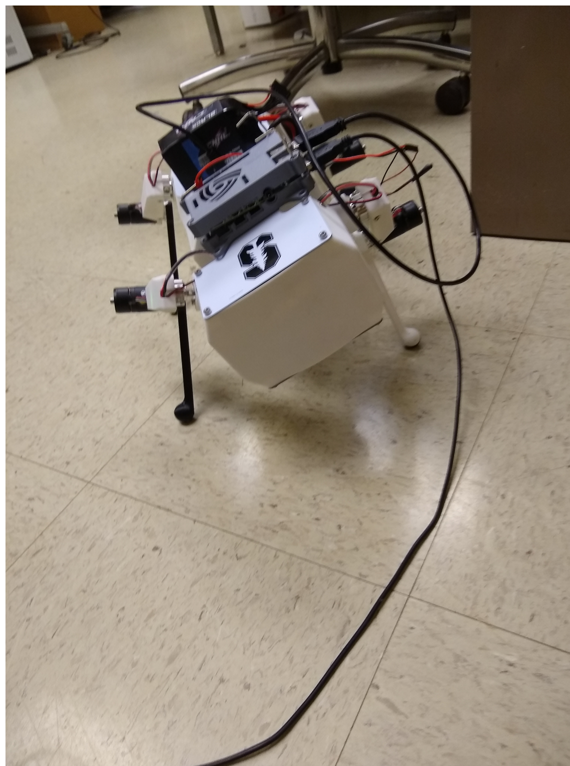


Figure 20 Picture of the robot rolling right.

We also achieved walking functionality, which is difficult to show in a still image. For video of our Pupper walking, check this Box folder.

We found several unforeseen issues related to controlling the robot. One minor issue is the tendency of the robot to shift and turn even while trotting in place. By design, the robot does not walk unless the "trot" mode is activated, in which the robot constantly marches by lifting and lowering its feet. However, the robot can move unintentionally while supposedly trotting in place, slowly rotating or backing up. This could be due to weight imbalances in the robot causing it to shift without the feet firmly planted, and is probably an unavoidable issue. Still, it is important that the controller be aware that the robot never really trots "in place."

Another more substantial problem is the robot's tendency to reset incorrectly when the legs are extended (this is shown a several of the Box videos linked above). When the legs near a 180 degree angle, they have a tendency to reset backwards, with the "knee" of the robot pointing to the front rather than the rear. This can and will cause fatal errors when the robot tries to move, such as the front and rear legs clashing with each other or a simple loss of balance, leading to the robot toppling over. Therefore, whenever one of the robot's legs resets backward, the problem must be immediately resolved. This can be done by deactivating the robot completely and re-running the control code with the zeroing option on, but this is an enormous pain. A quicker solution is to use the "raise/lower" movement option to make the robot stand up as high as possible, with legs nearly straight vertical. Then, slowly tap the button to lower the robot while pushing the robot's knee backward, forcing the leg to reset to the correct orientation. If multiple legs have bent wrong, this may have to be done multiple times, once for each leg. Nevertheless, it is a much faster solution than completely resetting the robot.

It seems as though this problem should be preventable with a change to the robot's code. Future groups may want to try implementing a section of code that prevents the knee motors from ever angling the leg links at greater than 180 degrees (or less than 0, if taking the zeroed state of the motors as the frame of reference). This feature would save a lot of time and headache with the robot's operation, as the legs have a tendency to turn backward often.

Development of New "Jumping" Gait.

As mentioned in a previous section, the main controller has several modes that it can switch between to achieve different gait. These modes are defined in the <state.py> file.

```
class BehaviorState(Enum):
    DEACTIVATED = -1
    REST = 0
    TROT = 1
    HOP = 2
    FINISHHOP = 3
```

Figure 21 Controller Modes

In these modes, DEACTIVATED mode is assigned to -1, while mode 0-3 indicating that the robot is in an ACTIVATED stage. Mode 0 corresponds to the REST mode, where the robot is ACTIVATED but not performing any actions. Mode 1 corresponds to the TROT mode, where the robot will be stepping on the same spot initially. Only when the controller receives changes from the joystick interface, and the foot location (for the next state) of the robot is changed, will the robot start walking. Both mode 2 and 3 together composed the HOP gait. The HOP gait is a gait where the robot will jump vertically. Mode 2 corresponds to the movement before the hop, where the knees are bent, and body is lowered. Mode 3 make the robot's legs push against the floor to produce a counter force upward. There are two main parts about gait controls. One is to switch between the modes, and the other is to design the mode so that it performs the desired tasks. We are going to introduce how Pupper achieves these in the next several paragraphs.

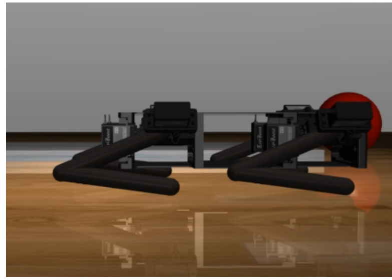
To enable the transitioning between different modes, the Pupper controller <controller.py> uses dictionaries Figure (22). These dictionaries allow a 1:1 mapping between modes. If a mapping does not show up in the dictionary, then the transitioning is disabled. For example, the only mode that can transit into DEACTIVATED mode is the REST mode, meaning that it is only possible to put the robot into rest before deactivating it.

```
self.hop_transition_mapping = {
    BehaviorState.REST: BehaviorState.HOP,
    BehaviorState.HOP: BehaviorState.FINISHHOP,
    BehaviorState.FINISHHOP: BehaviorState.REST,
    BehaviorState.TROT: BehaviorState.HOP,
}
self.trot_transition_mapping = {
    BehaviorState.REST: BehaviorState.TROT,
    BehaviorState.TROT: BehaviorState.REST,
    BehaviorState.HOP: BehaviorState.TROT,
    BehaviorState.FINISHHOP: BehaviorState.TROT,
}
self.activate_transition_mapping = {
    BehaviorState.DEACTIVATED: BehaviorState.REST,
    BehaviorState.REST: BehaviorState.DEACTIVATED,
}
```

Figure 22 Mappings Between Modes

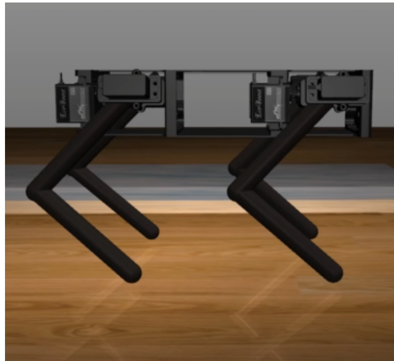
For the purpose of this project, we will focus mainly on investigating the hop gait design of the robot. As mentioned above, the hop gait is composed of two modes: HOP mode and FINISHHOP mode. The basic idea of this gait is straight forward: 1) Since the hop gait only generates a vertical movement, there is no net change for the foot location on x(forward) or y(side) directions. 2) For the HOP mode, knees are bent, body is lowered, thus the distance between the body and foot is short. 3) For the FINISHHOP mode, the legs are pushed, so the distance between the body and foot is long. Based on these rationales, the implementation of these two modes is simple: In HOP mode, copy the default foot locations, apply a net change in the z direction of magnitude $|Z_\alpha|$. In FINISHHOP mode, copy the default foot locations, apply a net change in the z direction of magnitude $|Z_\alpha|$. And $|Z_\beta| < |Z_\beta|$. The use of the absolute value is because all Zs should be negative since the foot is below the body. A gallery of different Z values and their correspondent robot posture is shown in Figure (23a).

(a) $z = -0.02$



(b) $z = -0.05$

(c) $z = -0.09$

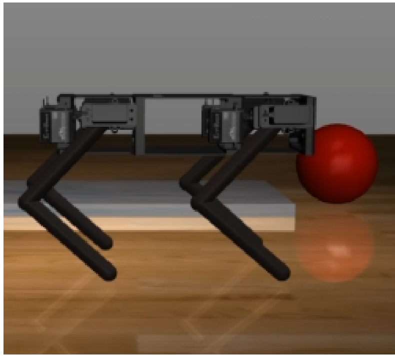


(d) $z = -0.15$

(e) $z = -0.22$

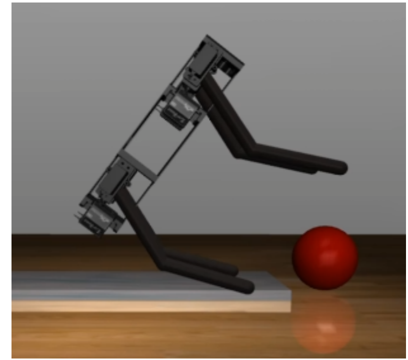
Figure 23 Robot Postures at Different Zs

The default hop gait uses (0.09) for HOP mode and (0.22) for FINISHHOP mode. Upon our investigation, we figured that (0.22) is the max that the legs can stretch, and (0.09) is a good parameter for HOP mode. Anything above (0.09) will cause the robot not to leap as high, and figures below (0.09) have almost the same effects as. A gallery of the complete motion of a hop is shown in Figure (24a). Note that the robot leaps the moment that it enters FINISHHOP mode and it will stay in the FINISHHOP mode even after landing because no other instructions were given.

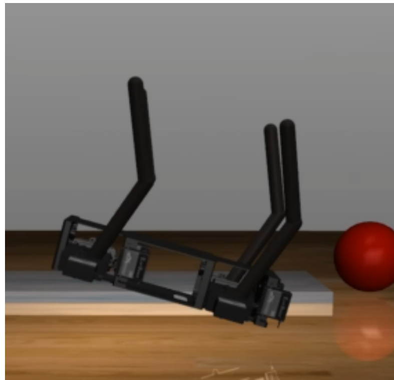


(a) REST

(b) HOP



(c) FINISHHOP



(d) Falling

(e) Landing

Figure 24 Complete Motion of A Normal Hop

Jump Gait Design.

In this project, we managed to design a jump gait. For the jump gait, we want the robot to achieve the following: 1) Jump forward. 2) Retain balance with feet on the ground after jumping. Based on the previous Hop gait analysis, we determined the modifications that need to be done in order to achieve our goal: 1) Add changes to the x direction so the robot can be pushed forward. 2) Add another mode after the FINISHHOP so that the legs can contract and provide balance again.

To implement the first modification, we investigated the effects of changes on x direction. To better observe the results, we set the two front legs to maintain a default posture, and experiment on the two rear legs:

(a) $x = -0.20$

(b) $x = -0.05$

(c) $x = +0.05$

Figure 25 Robot Postures at Different Xs

We can observe that the changes in the x direction can be positive or negative. If a reference line was drawn at the location of the hip motors, a positive change makes the foot lands before the line, and a negative change makes the foot lands after the line. Also, the range on the positive side is significantly shorter than the negative side. At +0.05, the knee of the robot is touching the floor to reach the desired foot location. Since the jump gait needs a push in the forward direction, we can conclude that we need a negative change in the x direction.

To implement the third mode, we first need to define it in the <state.py> file. In our case, we named it BALANCE and assigned it to mode 4. To note, since the joystick interface for the simulator is prewritten and unchangeable, we do not have a way to swap between the normal hop gait and the new jump gait. Therefore, in order to implement the new gait, we overwrite the original hop gait,

so that the HOP and FINISHHOP mode are now part of the jump gait. The loop between the three modes should follow this sequence: HOP -> FINISHHOP -> BALANCE. The logics are: bend knees, prepare for jumping -> spring off legs to push against floor -> contract legs to provide balance before landing.

Finally, we need to modify/build the three modes so that each of them performs their designated tasks. For the HOP mode, we mimicked the posture of a frog, where the two front legs are expanded, and the two rear legs are contracted. With this posture, the shoulder of the robot is higher than its hip, making its head to point into an up and forward direction. Based on previous analysis, we summarized the desired parameters for HOP mode:

HOP	Purpose	Parameter
Front legs x direction	Provide balance	Positive, Average
Front legs z direction	Expand	Negative, |Large|
Rear legs x direction	/	Negative, |Small|
Rear legs z direction	Contract, Lowering Hip	Negative, |Small|

Figure 26 HOP Mode Parameters Summary

One thing to note in this table is the choice for the rear legs x direction parameter. Our initial thought is that it should be a small positive figure. However, after multiple trials, we discovered that with the other three parameters settled, any positive number for this parameter will result in the knees of the rear legs to touch the floor. This is not a typical situation where the robot's knees are contacting with the floor. Therefore, we decide to assign the rear legs x direction parameter to a small negative value. The magnitude of this value is dependent on the value of rear legs z direction parameter. In our case, with rear legs z direction parameter being (-0.11), a value of (-0.01) for rear legs x direction is sufficient.

For the FINISHHOP mode, the main purpose is to push hard against the floor. From previous analysis, the max expansion that the legs can have to the negative direction is about (-0.2). However, we should not assign all parameters to this max value. One key constraint here is that the front legs should be pushing less hard than the rear legs. This constraint is a conclusion from observations to the original HOP gait. In the original gait, as the front and rear legs are pushing with the same scale, the robot will always flip over. In order to prevent this behavior, we decided to design the parameters

so that the rear legs contribute to most of the pushing power in both directions. A summary of desired parameters for FINISHHOP mode is shown here:

FINISHHOP	Purpose	Parameter
Front legs x direction	Pushing Forward	Negative, |Large|, Smaller than RearX
Front legs z direction	Pushing Upward	Negative, |Large|, Smaller than RearZ
Rear legs x direction	Pushing Forward	Negative, |Large|
Rear legs z direction	Pushing Upward	Negative, |Large|

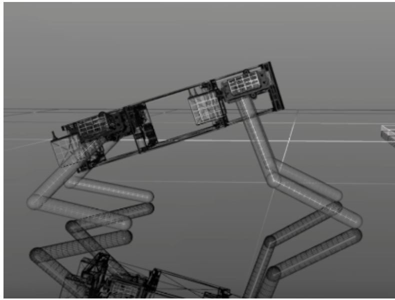
Figure 27 FINISHHOP Mode Parameters Summary

For the BALANCE mode, the main purpose is to provide balance for landing. In order to provide balance, an important constraint for this mode is that the front legs and the rear legs should be fairly separated. The BALANCE mode should also lower the body of the robot to provide extra balance. Additionally, we should design the BALANCE MODE so that its transitioning to the HOP mode is fairly smooth. A summary of desired parameters for FINISHHOP mode is shown here:

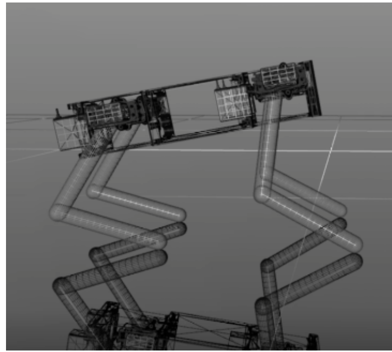
	Purpose	Parameter
Front legs x direction	Balancing	Similar to HOP mode
Front legs z direction	Lowering Body	Negative, |Small|
Rear legs x direction	Balancing	Similar to HOP mode
Rear legs z direction	Lowering Body	Negative, |Small|

Figure 28 BALANCE Mode Parameters Summary

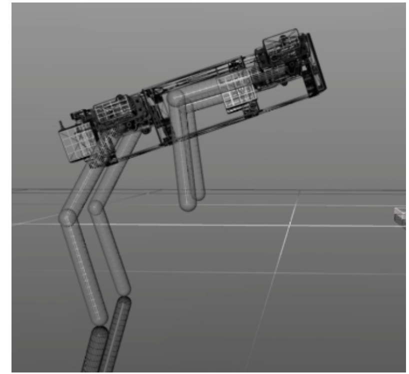
A gallery of the complete motion of a jump is shown in Figure(29a-29e). Although the pictures cannot fully show the motions during the whole process, it is still observable that the robot jumped forward and maintained balance after landing. Thus we can conclude that the jump gait is successfully implemented.



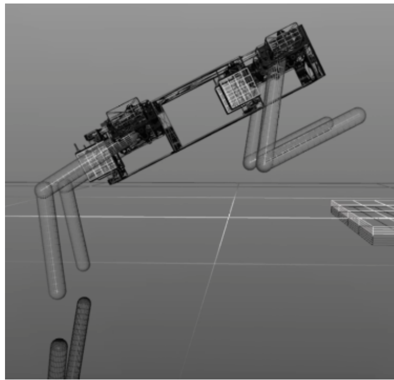
(a) Rest (HOP)



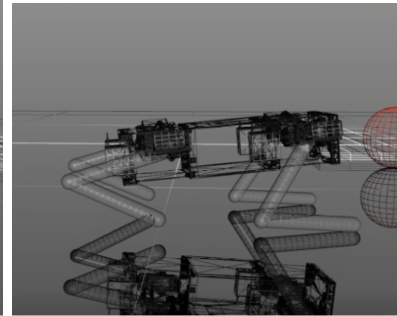
(b) Pushing (FINISHHOP)



(c) Jumping (FINISHHOP)



(d) Falling (FINISHHOP)



(e) Landing (FINISHHOP)

Figure 29 Complete Motion of A Jump

CONCLUSION

The quadruped robot built in the project was the Stanford Pupper. In this project, we built the actual quadruped Stanford Pupper robot and designed a new gait for robot to hop forward. Detailed instructions for the robot assembly, Raspberry Pi 4's set up, and software set up are created for future student use. The robot was built and tested to for standing, walking and tilting in different directions. The team was also able to design a new gait "jumping" which includes 4 states: rest, pushing, jumping, and landing.

For the future study, many improvements to the robot can be done based on this study. Firstly, the jump gait simulated by this project can be tested on the robot. In addition, current feedback from the motor can be used to track the force from each leg and use the control system described in the gait generation to generate different gaits and compare with theories. In addition, the performance of the PD controller can be evaluated by comparing the theoretical food location and actual foot location. Last but not least, computer vision module and trained AI can also be added to the robot software to control robot travel through obstacles or to chase a specific object.

References

[1] Kau, N. and Bowers, S., "Stanford Pupper: A Low-Cost Agile Quadruped Robot for Benchmarking and Education," Tech. rep.

[2] Muhammed Arif Sen, V. B. and Kalyoncu, M., "Inverse Kinematic Analysis Of A Quadruped Robot," Tech. rep.

[3] Katsumi, N., "Affine differential geometry : geometry of affine immersions," Tech. rep.

[4] "2D affine transformation matrix," accessed May. 8, 2021, https://commons.wikimedia.org/wiki/File:2D_affine_transformation_matrix.svg

[5] Craig, J. J., 2005, *Introduction to Robotics*, 3rd ed., Pearson Education, Upper Saddle River , NJ.

[6] Yasuhiro Fukuoka, Y. H. and Fukui, T., "A simple rule for quadrupedal gait generation determined by leg loading feedback: a modeling study," accessed May. 14, 2021, https://www.nature.com/articles/srep08169

A Robot Startup Guide

Background: Follow this procedure each time you start up the robot. Unless you are extremely familiar with the process and confident in the precision of your keystrokes, it is recommended to do this while the Raspberry Pi is connected to a monitor via its micro HDMI port. Once the bot is active and responsive to commands, the monitor can be disconnected without issue.

(1) Press the robot power button and wait a few seconds. You should hear all the motor controllers beep and the Raspberry Pi will boot up automatically.

(2) The Pi is currently set to boot to desktop. Open the command terminal by clicking the terminal icon on the taskbar, or using the keyboard shortcut Ctrl+Alt+T.

(3) Run the command:

sudo ifconfig eth0 10.0.0.52 netmask 255.255.255.0

Note: this command may become unnecessary if a good method can be found to set the ethernet IP on startup.

Note: "eth0" is the name of our Raspberry Pi ethernet port. The original guide used "en1" instead. Run the command "sudo ifconfig" to see information on the available wired and wireless connections. Look for a connection labeled "ethernet".

(4) Run the commands:

cd PupperKeyboardController

python3 keyboard_joystick.py

(5) A separate window should open up and wait for keyboard input, while the terminal will begin to print status information.

(6) Open a new terminal with the taskbar or Ctrl+Alt+T.

(7) Run the commands:

cd StanfordQuadruped

python3 run_djipupper.py

Note: To re-zero the motors, run "python3 run_djipupper.py –zero". This should be done each time the Pi is booted up, but not necessarily for each activation of the "run_djipupper.py" program.

(8) The terminal will say “Waiting for L1 to activate robot.” Navigate to the keyboard controller window using the mouse or (keyboard shortcut unknown). Once the keyboard controller window is active, the robot is now ready to use and will receive commands.

    Note: When in doubt, press L1/q to deactivate the bot. The motors can move abruptly and forcefully, so unless you’re very comfortable and familiar with the commands you’re sending, keep a finger on the deactivation button.

(9) Command list (PS4 controller—keyboard):

    Activate/deactivate (L1—q)

    Trot/rest (R1—e)

    Move forward/back and left/right (left joystick—wasd)

    Tilt up/down and yaw left/right (right joystick—arrow keys)

    Translate up/down and roll sideways (D-pad—ijkl)

    Note: To move the robot, first press R1/e to trot, then left joystick/wasd to move. The robot can lift, roll, tilt, etc. without trotting, but it must be trotting to walk around.

B Software Installation Guides

Teensy. Background: the Teensy board is responsible for the fine control of each motor and takes commands from the Pi, which handles high-level control. Since it does not receive commands directly from the user and has no interface to interact with, the Teensy must be pre-loaded (or "flashed") with code via the following process.

Note: The process for flashing the Teensy was successfully completed on a Windows computer. According to the directions provided by the original Pupper team, it should work with exactly the same commands and programs on Mac. However, we were unable to flash the Teensy using the Pi itself due to permissions issues, and so we cannot recommend using Linux for this process.

(1) Install Git for your operating system: https://git-scm.com/downloads

(2) Open a command line. This was done with the Windows command line (Windows+R, type "cmd"), but it should also work in Windows PowerShell and the Mac terminal.

(3) Navigate to the directory where you want your project files to be located. The command will be something like:

cd C:\Users\YourName\Documents\PupperProject

(4) Clone the "DJIPupperTests" folder from Github.

git clone https://github.com/Nate711/DJIPupperTests.git

(5) Navigate into the "DJIPupperTests" folder you just created:

cd DJIPupperTests

(6) Run the command:

git submodule update –init

Note: if the terminal doesn't show various files being updated, check that you're in the right directory. You can use "dir" (Windows) or "ls" (Linux) to check the contents of the current directory, and "cd" (Windows) or "pwd" (Linux) to check the file path.

(7) Install Visual Studio Code: https://code.visualstudio.com/Download.

Note: Even if the Teensy is not being flashed using the Raspberry Pi, it is still handy to install VSCode on the Pi as well because the "monitor" feature can be used to monitor the Teensy's debug information. To install VSCode on the Pi, use the command line to run:

sudo apt update

sudo apt install code

Run VSCode on the Raspberry Pi by typing "code" into the Pi terminal.

(8) Open VSCode and install the PlatformIO extension. To do this, click on the icon with four boxes on the left toolbar labeled "Extensions" and search for PlatformIO. Locate the extension, install, then restart VSCode.

(9) Connect the Teensy to the computer using a microUSB cable. The Teensy should register the connection by blinking its LED several times.

Note: it is also recommended to install the "Python" and "C/C++" extensions for general use while you're at it.

(10) Follow the steps in this video: https://knowledge.autodesk.com/community/screencast/cbf5a477-08e8-4b54-aa1b-aeffc3e5aa3d.

Note: the Github repository has been updated several times over the course of this project. Some versions may have "ArduinoJson" and "BasicLinearAlgebra" already installed; don't worry if this is the case.

Note: you can click "Upload" first to check for errors, then "Monitor" to print information from the Teensy to the terminal. The aptly-named "Upload and Monitor" option does both with one click.

(11) When the code is first uploaded and monitored, the VSCode terminal will print debug information that indicates the positions of each motor connected to the Teensy.

Note: depending on the code version you clone, the debug info may print very fast and/or be unreadable. To print human-readable debug information at a comprehensible pace, you can edit the code in "main.cpp" using the VSCode interface. Open "main.cpp" and change PRINT_DELAY on line 11 to 1000 (this makes the debug info print once every second). On line 182, comment out "drive.PrintMsgPackStatus(options);" by typing "//" in front of it. On line 180, uncomment "drive.PrintStatus(options);" by deleting the "//". This will change the debug info to a list of numbers representing motor positions and velocities. Once the information-printing code is changed to your liking, re-upload and re-monitor to check that the debug information now prints correctly.

Raspberry Pi (Keyboard Controller). Background: the keyboard controller allows control of the robot using a keyboard rather than a PS4 controller. This comes with trade-offs; the controller is less bulky and provides more intuitive control, but the keyboard allows for command line setup and robot control with one device.

Note: for ease of access, it is recommended to install the code in the folder "/home/pi". This is the folder that the terminal will open to by default, and it has a special short name: "~". If your terminal is not already in the home folder, run the command "cd ~" before following these steps.

(1) Clone the joystick emulator code from the Github page:

git clone https://github.com/stanfordroboticsclub/PupperKeyboardController.git

(2) Python 3 should already be installed on the Pi. If not, run:

sudo apt update

sudo apt install python3

(3) Make sure pip is installed for Python 3:

sudo apt install python3-pip

(4) Install PyGame:

    pip3 install pygame

(5) Install UDPComms:

    git clone https://github.com/stanfordroboticsclub/UDPComms.git

    sudo bash UDPComms/install.sh

Raspberry Pi (Control Code). Background: the Pi control code handles high-level control of the robot. Unlike the Teensy, the Pi receives commands directly from the user while the robot is operating.

Note: for ease of access, it is recommended to install the code in the folder "/home/pi". This is the folder that the terminal will open to by default, and it has a special short name: "~". If your terminal is not already in the home folder, run the command "cd ~" before following these steps.

(1) Clone the control code Github repository:

    git clone https://github.com/stanfordroboticsclub/StanfordQuadruped.git

(2) Go to the StanfordQuadruped folder:

    cd StanfordQuadruped

(3) Checkout the "dji" branch:

    git checkout dji

(4) Connect the Teensy to the Pi and find out the Teensy's tty device name. In the /dev folder of the Raspberry Pi, there is a long list of folders labeled "tty[something]", where [something] is a unique combination of letters and numbers. Whenever you plug the Teensy into the Raspberry Pi, a new tty folder will be created. Find the name of this folder. The Pupper project doc recommends running the command "ls /dev | grep tty.usbmodem", but this method did not work for us because our tty folder name did not include the text "usbmodem". We found the folder name using the windowed interface rather than the command line. We navigated to the /dev folder, then plugged and unplugged the Teensy while scrolling through the folder's contents. Eventually, we noticed a folder popping in and out of existence as the Teensy was plugged and unplugged. For us, the folder was named "ttyACM0".

(5) Once the Teensy's tty folder name is known, open the Python code file

" /StanfordQuadruped/djipupper/IndividualConfig.py" with the Pi's "Text Editor" program. At the bottom of the file, a line will read "SERIAL_PORT = "/dev/tty.usbmodem73090601" # nathan's". Edit this line with the name of your tty folder; for us, the line now reads: SERIAL_PORT = "/dev/ttyACM0". Save and exit the text editor.

(6) Install the "transforms3d" python package. This package is necessary to run the control code Python file:
pip3 install transforms3d

C Hardware Build Notes

Before Doing Anything.

(1) Make a copy of the official bill of materials (BOM) and add a column representing the number of each part that you have. Take a full inventory of the kit, recording the number of each part in your own BOM. This will quickly show which parts you're missing, if any.
Note: The kit may have some issues, such as missing/badly printed parts. For example, we did not receive enough clamping D-hubs or M3x8 flathead screws. Also, we received 4 right lower legs and 3 right hips with 1 left hip. The lower legs are supposed to have small holes to attach the rubber feet. However, the legs we received with the kit had misprinted, filling the holes and making it impossible to attach the rubber feet. Expect issues like this, and try to identify them as soon as possible.

(2) Label each bag of hardware with its actual name! The screws come in bags with an ID number that's listed in the BOM, but the actual build pictures use the name of the part. For example, "92125A128" is an M3x8 flat head screw. Write the part name on the bag so you don't have to constantly cross-reference part names/numbers with the BOM. This will save a LOT of time and prevent potential part mix-ups.

Other Notes.

(1) It can be difficult to keep track of which parts correspond to the right and left legs. We recommend using a labeling system during assembly. We used tape to mark off different sections of the table for "front left", "rear left", "front right", and "rear right", which allowed us to sort

the parts without putting any marks on them.

(2) In general, the pictures provided on the Pupper product page are more useful than the build video when assembling the robot. The video should be used to get an idea of the general thing you should be doing next, while the step-by-step assembly should be done referencing the project page pictures.

(3) Being able to print replacement 3d-printed parts yourself is very useful, as several parts have a tendency to break during assembly, or to simply be misprinted to begin with. Of note are the upper leg shrouds, which are thin and can snap while being screwed on. And as mentioned before, the lower legs may not mesh with the rubber feet correctly.

(4) At 28:58 in the build video it shows how to attach the golden threaded inserts. Note that the tool that should be used to attach these inserts is a soldering iron! The inserts are meant to heat up and melt the plastic around them. It may take a few seconds for the heat to begin melting the plastic. Press the soldering iron firmly and steadily against the top of the insert until it sinks in.

(5) Screw orientation matters! The purpose of adding one longer/cap head screw on some of the leg pieces is to restrict the leg range of motion. If these screws are put in the wrong place, the legs will not be able to move where they're supposed to move.

(6) The motor-controller pairs need to be calibrated and the controller IDs need to be set according to the instructions on the project page. Unfortunately, the controller IDs can be difficult to keep straight. Make sure to keep the motor-controller pairs organized or labeled.

D Recommended Reading and Helpful Resources

There are many resources which were helpful to our project but which were not directly used to provide any content in the report. Rather than citing these as references, we direct readers to them here.

(1) Raspberry Pi setup instructions.

(2) A good explanation of many basic linux commands for controlling the Raspberry Pi. See especially: pwd, cd, ls, locate, sudo, man, and the "Bonus Tips and Tricks" section.

(3) A basic guide to Git and Github, explaining cloning.

(4) Read more about VSCode.

(5) Read more about PyGame.

E Simulator Guide

Please See next page for guides on simulator installations and modifications.

Simulator Guide

This documentation serves as a **supplement** to the original instructions provided. Specifically, this documentation aims to help **WINDOWS** users because most the instructions are oriented around UNIX systems. Note that some of the steps shown in this document are different from what are shown in the original instructions. Please FOLLOW THE ORIGINAL INSTRUCTIONS FIRST. Only refer to the guides in this logbook when you are having trouble with the original instructions.

Here is a glossary of syntax that we used in this document:

<anything> specifies a file. For example: <Atom.exe>

{anything} specifies a folder. For example: {New Folder}

[anything] specifies the path/folder. For example: [C:\Users\abcd\Desktop]

~ specifies something that's user-particular. For example, we will represent our ip address or username by ~.

Part I: Simulator Installations/Setup Guide

Original Instructions on:

<https://github.com/Mark-Bowers/PupperSim.jl/blob/master/README.md> provided by Mark Bowers.

- Step 1: Install MuJoCo and obtain a license
  - Follow the instructions from MuJoCo website to correctly install all necessary MuJoCo files.
  - For the license file, it should be a file called <mjkey.txt>.
  - First copy the mjkey file to the {bin} folder where you installed the MuJoCo. On our PC, it's [~\.mujoco\mujoco200_win64\bin].
  - Once done, open the <simulate.exe>. A window should popup. Don't close it.
  - Then go to the parent folder and find the {model} folder. Open it up and then input (drag) any file with type <.xml> to the <simulate> window.
  - If it correctly opens up with no warning, then your license is valid, and you are good to proceed.

- Step 2: Install Jullia
  - Follow EXACTLY as shown in the original instruction.
  - To note, julia is both case-sensitive and space-sensitive. Pay close attention to the Caps, spacing, and even the period(.) at the end of the line.

- Step 3: Set up the license and the path.
  - This step corresponds to the "set the environment variable MUJOCO_KEY_PATH to point to its location." part in the original instruction. Note that in our guide we've changed the order of the steps slightly.
  - In WINDOWS search bar, search "Edit environment variables for your account"

- ➢ In the pop up window, click on “New” on the top half of the window. Notice that you won’t be able to change anything from the bottom half, so all our guides in this section will be operating on the top half.
- ➢ In the new popup, enter “MUJOCO_KEY_PATH” for the variable name.
- ➢ For variable value, click on “Browse file”, then navigate to your <mjkey.txt> and select. Make sure you select the file itself but not just the path to it. The variable value should be set to something like [~\mjkey.txt].

❖ Step 4: Run Simulator From julia

➢ This step corresponds to the “Usage” part of the original instruction. Follow the EXACT instructions.

➢ If the simulator window shows up, and you can see the Pupper on your screen, you are good to go. Feel free to jump to Step 6 on some of the basic controls of the Pupper in the simulator.

➢ If an error shows up saying something about invalid license, don’t panic, go to the next step.

❖ Step 5: Fix the license path

- ➢ Copy the <mjkey.txt> file. NOT the path to it, but the file itself.
- ➢ Paste one copy to the user’s root folder. For example, if your logged in account to the WINDOWS is called “user”, then the folder should be at [C:\Users\user]. To check if this is the right folder, open it up and you should see a {.julia} and a {.mujoco} folder. Paste the <mjkey.txt> file into this {user} folder.
- ➢ Now find the {bin} folder in the {.mujoco} folder. The path should be something like [C:\Users\~\.mujoco\mujoco200_win64\bin]. Make a copy of the <mjkey.txt> here inside the {bin} folder. If this is the initial location where you stored the <mjkey.txt> you don’t need to replace it.
- ➢ Now go back to the user’s root folder {user} and open the {.julia} folder. In the {.julia} folder, find the {iUMEQ} folder WITHIN the {PupperSim} folder. In our case, the path to it is: [C:\Users\~\.julia\packages\PupperSim\iUmEQ]. Paste a copy of the <mjkey.txt> in this folder.

- ➢ Technically you should have 3 or 4 copies of the <mjkey.txt> file at different locations. That is okay. The ultimate purpose is for MuJoCo to find the license.
- ➢ Finally, go back to the environment variable set up page again. (Search bar -> Edit environment variables for your account). Select the MUJOCO_KEY_PATH variable that you set up in step 3 and click on "Edit". Change the variable value to [~\PupperSim\iUmEQ\mjkey.txt].
- ➢ Go through step 4 again. Now the simulator should be working.

❖ Step 6: Simulator Controls (brief)

- ➢ The best way to control the simulator is by a PS4 controller.
- ➢ One can use DS4Windows or x360ce to install necessary drivers and set up the connection between the PS4 controller and the PC.
- ➢ Technically you should connect the controller first before running the simulator.
- ➢ The starting screen of the simulator should be a Pupper standing on the ground with four legs straight.
- ➢ To start time (make simulator play), press "SPACE" on your keyboard.
- ➢ To make the Pupper start walking, press "R1" from the controller.
- ➢ Use the left analog on the controller for moving and the right analog for turning.
- ➢ If you want to restart the simulator, press "BACKSPACE" on your keyboard

Part II: Implementing Changes in Simulator

There are no original instructions for this online. Please follow the steps clearly as stated. The main purpose of this part is to make whatever change you want with the robot and be able to see its effect in the simulator.

- Step 1: Locate julia executable file
  - If you don't have a shortcut for julia.exe and always open from the executable directly, go to Step 2.
  - Right click on your julia shortcut. For my case, it says <Julia 1.5.3> on my desktop.
  - In the drop down menu, click on "open file location"

- Step 2: Create a PATH to the julia.exe on WINDOWS
  - Copy the path to the <julia.exe> file. In my case, it is:

    [C:\Users\~\AppData\Local\Programs\Julia 1.5.3\bin]
  - Right click on your julia shortcut. For my case, it says <Julia 1.5.3> on my desktop.
  - In WINDOWS search bar, search "Edit environment variables for your account"
  - In the pop up window, select "Path" on the top half of the window, then click on "edit". In the new pop up window, click on "new", then paste you previously copied path (path to julie.exe), and click "ok".
  - Open WINDOWS command prompt <cmd>. Enter "julia".
  - If it runs with no error and you see the "Julia" symbol in <cmd>, proceed.

- Step 3: Develop a local version of the package
  - Here is the rational behind developing local instances of those packages. In Part I, we installed packages from GitHub to julia. However, julia does not expect us to make changes to those preexisting packages. Therefore, we need to develop our own local packages, make change there, and tell julia to run our version of the packages directly.

- Open julia from <cmd>.
- Type "dev https://github.com/Mark-Bowers/QuadrupedController.jl". This develops a local instance of the QuadrupedController package.
- Type "dev https://github.com/Mark-Bowers/PupperSim.jl". This develops a local instance of the PupperSim package.
- Close down the cmd once the installation process is finished.
- Open the {.julia} folder. Again, mine locates at [C:\Users\~\.julia].
- Open {dev} folder inside {.julia}
- It would be time saving if you add this folder {dev} to quick access. Everything that you want to edit will be in this folder.
- In {dev}, you should see two folders. One named {QuadrupedController}, the other named {PupperSim}. If you see them, this step is completed.
- !!IMPORTANT NOTE!! You can also see these two folders in the {packages} folder. Like explained earlier, julia does not expect things in the {packages} folder to be changed. All changes should be made only to things in the {dev} folder. Keep this in mind.

❖ Step 4: Modify the Manifest file

- Find the <Manifest.toml> file in the {.julia} folder. In my case, it is in:

  [C:\Users\~\.julia\environments\v1.5].

  It is also likely that there is <Manifest.toml> file in the:

  […\dev\QuadrupedController] or […\dev\PupperSim] folders.
- Make the following steps to ALL the <Manifest.toml> files
- Make a copy in case something goes wrong.
- Open <Manifest.toml>. If you don't have an editor to edit it, just open it with the Notepad.
- It should say: "# This file is machine-generated - editing it directly is not advised" on the top. Ignore that for now.
- Find the "[[QuadrupedController]]" chuck, make the following edits:

- [[QuadrupedController]]

  deps = ["Conda", "PyCall", "StaticArrays"]

  path = "/users/~/.julia/dev/QuadrupedController/"

  uuid = "0a0771de-1099-46b6-8518-8474b76bc44f"

  version = "0.1.0"

- It’s likely that you don’t need to make any change at all. Then just leave it as it is.

❖ Step 4: Make a change that you want

- Note that the simulator provides only a mean to simulate the robot. Changing how the robot works has nothing to do with julia or MuJuCo. In this part, we will provide a simple example of how to change one parameter of the robot and see the effect in simulator. Do not refer to this guide on how to make changes to the robot. This guide is only about how to implement a change, but not the rational or logic behind it.
- In our case, we’d like to change the walking speed of the robot.
- Open the {QuarupedController} folder in the {dev} folder. Again, DO NOT change anything in the {packages} folder.
- The parameter that we are looking for is in the <config.py> file.
- Change “max x velocity” to 0.2. This should slow down the robot.
- Save your changes.

❖ Step 5: Implement a change

- Locate the path to the developed {PupperSim} folder. In my case, it is:

  [C:\Users\~\.julia\dev\PupperSim]

  Copy this directory.

- Open <cmd>
- Navigate to the {PupperSim} folder. To achieve this, type:

  “cd C:\Users\~\.julia\dev\PupperSim” (“cd” then paste the path)

- Now you should see the running directory changed to the {PupperSim} folder
- Run julia from this directory. (just type "julia" then enter)
- Wait until julia is loaded.
- Enter package mode by press "]". You should see something like:

```
(@v1.5) pkg>
```

- Enter command "activate ." This forces the use of the local <Manifest.toml> file. Note the SPACE and PERIOD after the phrase "activate".
- Now you should see:

```
(PupperSim) pkg>
```

- Press BACKSPACE to return to julia REPL.
- Run "Using PupperSim" (Same as how you run simulator in PART I). In general, if your "Using PupperSim" command finish running in 1 sec, it's likely something went wrong and your new changes were not implemented. Normal running time for us is somewhere between 10-30s.
- Run "PupperSim.simulate()" (Same as how you run simulator in PART I).
- You should be able to see the changes applied in simulator. In our case, the robot is now moving slower than before.
- Every time you want to make a new change, you need to CLOSE the <cmd> and REPEAT step 4&5 again.