6-10-2022

Foundations of the Pupper Quadruped Robot with Preliminary Work on an End Effector

Liana Tilton
Washington University in St. Louis

Clement Siu
Washington University in St. Louis

Jack Nanez
Washington University in St. Louis

Max Saltrelli
Washington University in St. Louis

Lila Dickstein
Washington University in St. Louis

See next page for additional authors

Recommended Citation

Tilton, Liana; Siu, Clement; Nanez, Jack; Saltrelli, Max; Dickstein, Lila; and Tang, Joshua, "Foundations of the Pupper Quadruped Robot with Preliminary Work on an End Effector" (2022). *Mechanical Engineering and Materials Science Independent Study*. 195.
https://openscholarship.wustl.edu/mems500/195

Authors
Liana Tilton, Clement Siu, Jack Nanez, Max Saltrelli, Lila Dickstein, and Joshua Tang

Washington University in St. Louis
JAMES MCKELVEY SCHOOL OF ENGINEERING

Foundations of the Pupper Quadruped Robot with Preliminary Work on an End Effector

Lila Dickstein[1], Jack Nanez[1], Max Saltrelli[1], Clement Siu[2], Joshua Tang[2], Liana Tilton[1]

Independent Study Spring 2022

Faculty Advisors: Louis Woodhams[1] and Roger Chamberlain[2]

[1] Mechanical Engineering and Materials Science Department, Washington University in St. Louis
[2] Computer Science and Engineering Department, Washington University in St. Louis

TABLE OF CONTENTS

INTRODUCTION

Quadruped robots serve as bio-inspired systems that present design and control challenges. They cover a large number of engineering topics and provide countless applications in the field of robotics.[1] Studying a quadruped robot allows us to focus on the concepts of forward and inverse kinematics, working with systems of bidirectional motors, and the application and analysis of various gaits. This Independent Study explored the basics of building, controlling, and simulating a quadruped robot and was an effort in collaboration with Hands-On Robotics.[2] Following the Hands-on Robotics and Stanford Robotics Independent Study online curriculum, we assembled a Pupper quadruped robot. The Stanford Robotics Independent Study curriculum consisted of seven labs, each focusing on a different robotics topic that built off each other, culminating with full robot assembly and implementation of the robot simulator. It should be noted that Lab 5 was not published at the time of our work. The three motors on each leg enabled 12 degrees of freedom per leg which provided us with a difficult yet educational challenge, particularly with motor calibration and connecting each of the four limbs to mimic quadruped locomotion. The software and hardware challenges of working on a quadruped robot offered substantial insight into the concepts that go into the locomotion of more advanced robots.

This report is divided into six primary sections: forward kinematics, quadruped inverse kinematics, robot assembly, robot simulator, Pupper calibration and initialization, and robotic arm end effector attachment. The first five sections roughly correspond to Labs 3, 4, 6, and 7 of the Hands-on Robotics Curriculum. The Pupper calibration and initialization section discusses the challenges we faced with achieving a consistent initialization sequence. Finally, we discuss the preliminary work done designing and building a robotic arm end effector.

FORWARD KINEMATICS

In Lab 3, we learned about the basics of forward kinematics and how to apply it to a leg of the Pupper. Each Pupper leg contains 3 servos and our mission for the lab was to calculate the x, y, and z cartesian coordinates of the leg given the angles the 3 servos are set to. We were also given the lengths of the parts making up the Pupper leg. With this information we figured out the 3 equations, one for each cartesian coordinate. After obtaining the equations, we coded the equations in python so that we were able to move a Pupper leg around and see the position that the tip of the leg was at in real time.

Equations for the forward kinematics calculations:

Given that:

l_1 = length of hip part

l_2 = length of part connecting hip to the leg's end piece

l_3 = length of leg's end piece

θ_1 = angle of the hip servo

θ_2 = angle of servo controlling the leg's middle piece

θ_3 = angle of servo controlling the leg's end piece

The equations for each cartesian coordinate was:

$x =- l_2 sin(\theta_2) - l_3 sin(\theta_2 + \theta_3)$ = forward coordinate

$y = l_1 cos(\theta_1) + l * sin(\theta_1)$ = vertical coordinate

$z = l_1 sin(\theta_1) - l * cos(\theta_1)$ = lateral coordinate

With l being a equation factored out of the right sides of y and z

$l = l_2 cos(\theta_2) + l_3 cos(\theta_2 + \theta_3)$

QUADRUPED INVERSE KINEMATICS

In Lab 4, we learned about inverse kinematics. The difference between inverse and forward kinematics is that for inverse kinematics we are given the cartesian coordinates. With the cartesian coordinates, we need to find the angles for each leg and corresponding servo that will put the tip of the leg in that coordinate. To achieve this, we used gradient descent with an objective function of the euclidean distance between the forward kinematics calculated position with our guessed angles and the true position we want to get to. Thus requiring the derivative of the forwards kinematics and thus a jacobian matrix. Once we converted the math into Python code, we pushed the angles calculated to the Teensy. We were then able to give the leg coordinates and the leg would automatically go to our input coordinates.

The equation for general gradient descent (slightly modified for this problem) is as follows:

$$\theta_{k+1} = \theta_{k} - \lambda\nabla_{\theta}C(\theta)$$

θ_{k} is a vector of the angles of the geared motors on the legs. We define our forward kinematics equation as $f(\theta)$. For some θ, we get the resulting cartesian coordinates. The gradient descent equation effectively finds a new value of θ_{k} based on a constant multiplied by the derivative of some objective function $C(\theta)$. We can manipulate the constant λ of the gradient descent equation so that the equation results takes an appropriate step, in other words we aim to bring the value of $C(\theta)$ lower, thus gradient "descent". Therefore, if we design our objective function to tell us the error between guessed angles for geared motors, thereby corresponding cartesian position, and the true cartesian position, we can continuously iterate and guess new angles that lower the error. In short, imagine this equation descends down a valley in search of the very bottom and the valley is formed by the objective function. If the objective function defines error between an angle and the target, we have found the closest angles to get to a specific cartesian position.

Our objective function is:

$$C(\theta) = \frac{1}{2}||f(\theta) - r||^{2}$$

Our objective is to simply find some set of angles or θ that will minimize the distance between our target cartesian location or r. Both the outputs of $f(\theta)$ and r will be 3x1 vectors that are cartesian coordinates. The line brackets are simply representations of finding the euclidean distance for the $f(\theta) - r$. The use of squaring and multiplying by a half is to simply make it easier to take the derivative of the objective function. This equation is effectively how good our current angles are, the closer to zero the better.
The gradient of the objective function is as follows:

$$\nabla_{\theta}C(\theta) = (\nabla_{\theta}f(\theta))^{T}(f(\theta) - r)$$

With this solved, we can now substitute into the general gradient descent equation. Resulting in a final equation:

$$\theta_{k+1} = \theta_k - \lambda(\nabla_\theta f(\theta_k))^T (f(\theta_k) - r)$$

θ_{k+1} is a 3x1 matrix which represents our next guess, θ_k is the current guess. These matrices are the angles for the three servos that make up a leg. λ is the learning rate which is a constant, this simply influences the effective increase/decrease of the new guess. $\nabla_\theta f(\theta)$ is a 3x3 matrix of the jacobian of our $f(\theta)$ function, which is the forward kinematics equation. Finally, $f(\theta) - r$ is the error, where r is the 3x1 matrix containing the cartesian coordinates we want to get to and $f(\theta)$ is the 3x1 matrix containing the cartesian coordinates of our current guess. We then matrix multiply the Jacobian with the error and then scalar multiply that with the learning rate which gives us the change between the new angle and current angle. This change demonstrates the effects of the learning rate. If the learning rate is high then we don't need to repeat as many times as we move greater distances between angles, but we might overshoot the correct angles. If the learning rate is lower then it might take too long to get a set of correct angles. We are satisfied with our approximation when $||\theta_{k+1} - \theta_k||^2$ is small enough. The two main areas we had to adjust were the learning rate and deciding when $||\theta_{k+1} - \theta_k||^2$ was small enough. We played around with the learning rate and convergence value until we found ones that deduced acceptable angles reasonably fast. We ended up using a learning rate of 10 and a convergence value of .000001.

Code for calculating the jacobian, everything else is essentially the same as shown above. This is just the code representation of the jacobian matrix for the derivative of the forward kinematics equation.

```
BLA::Matrix<3,3> jacobian(const BLA::Matrix<3> &joint_angles, const KinematicsConfig &config){
 BLA::Matrix<3,3> res;
 // fx row
 res(0,0) = 0;
 res(0,1) = -1*config.l2*cos(joint_angles(1))-config.l3*cos(joint_angles(1)+joint_angles(2));
 res(0,2) = -1*config.l3*cos(joint_angles(1)+joint_angles(2));


 // fy row
```

```
  res(1,0) = -1*config.l1*sin(joint_angles(0)) +
config.l2*cos(joint_angles(1))*cos(joint_angles(0))+config.l3*cos(joint_angles(1)+joint_angles(2))*cos(joint_angles(0));
  res(1,1) =
-1*config.l2*sin(joint_angles(1))*sin(joint_angles(0))-config.l3*sin(joint_angles(1)+joint_angles(2))*sin(joint_angles(0));
  res(1,2) = -1*config.l3*sin(joint_angles(1)+joint_angles(2))*sin(joint_angles(1));

  // fz row
  res(2,0) =
config.l1*cos(joint_angles(0))+config.l2*sin(joint_angles(0))*cos(joint_angles(1))+config.l3*sin(joint_angles(0))*cos(joi
nt_angles(1)+joint_angles(2));
  res(2,1) =
config.l2*cos(joint_angles(0))*sin(joint_angles(1))+config.l3*cos(joint_angles(0))*sin(joint_angles(1)+joint_angles(2));
  res(2,2) = config.l3*cos(joint_angles(0))*sin(joint_angles(1)+joint_angles(2));

  return res;
}
```

The lab was helpful in teaching us how to get the equations for inverse kinematics. We were able to apply these equations to the Pupper leg and see how changing the equations affected the leg movement and performance. In the future, we would like to learn and implement other methods of inverse kinematics for different numbers of degrees of freedom. Additionally, we would like to improve performance by solving the optimal learning rate and minimizing convergence values.

ROBOT ASSEMBLY

In Lab 6, the remainder of the Pupper was assembled. See the Appendix for a full Bill of Materials. Two right legs and two left legs were assembled along with the bottom printed circuit board (PCB) with a Teensy 4.0. Both sets of legs were attached to a mount that was connected to the PCB. Each leg had 3 motors, two at the "hip" and one at the "knee", that were all set to their own respective motor IDs. This was done by plugging in the motors to the PCB and using the motor controllers to assign motor IDs based on their location in the Pupper. The inertial measurement unit (IMU) and Raspberry Pi (RasPi) microcontroller were then attached to the PCB. An IMU is a device that uses gyroscopes to measure inertial properties like force, angular acceleration, and orientation of the body. The Teensy 4.0 was attached to the RasPi through a

USB A to USB micro cable. The RasPi was connected to power using the 5V and GND pins. The IMU, if used, would be connected directly to the RasPi with a ribbon cable. A RasPi camera was also connected to the Ra Pi and was situated at the front of the Pupper build. After completing the electronics, the top PCB connected the power switch to the bottom PCB and battery source.

To connect the Pupper to the remote transmitter a receiver was required. A Frisky USB receiver was attached to the Pupper's RasPi. A BetaFPV Transmitter was then used as the remote controller to control the Pupper over wifi. The pairing sequence is described in Appendix 1. The receiver was then attached to the Pupper. An image of the fully assembled Pupper can be found below.

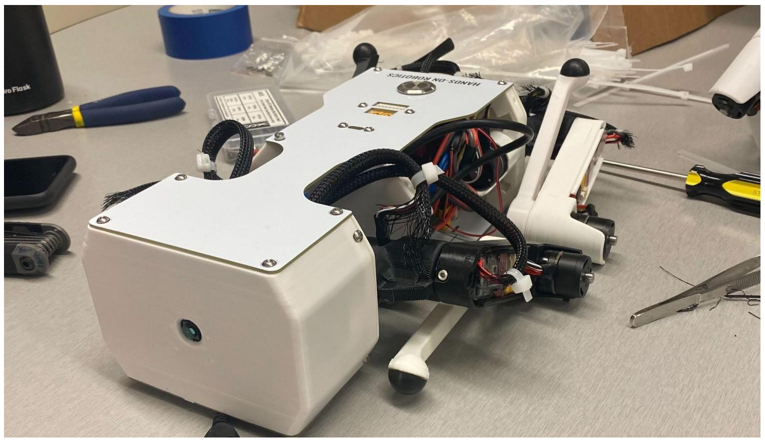


Figure 1. Fully Assembled Pupper Quadruped Robot

One common issue we ran into with the hardware was the hip of the Pupper fracturing. Often after extended use, the hip would either snap in half or would crack to a point where it would impede the Pupper's motion. We were able to 3D print more pieces each time it broke; however, the constant replacements became very time-consuming, so finding a way to make the hips more durable is definitely an improvement we'd like to work on in the future. The CAD file for the hip can be found in Appendix A2. A solution we have considered is using SLS Nylon printing instead of FDM to improve durability of the printed layer boundaries.

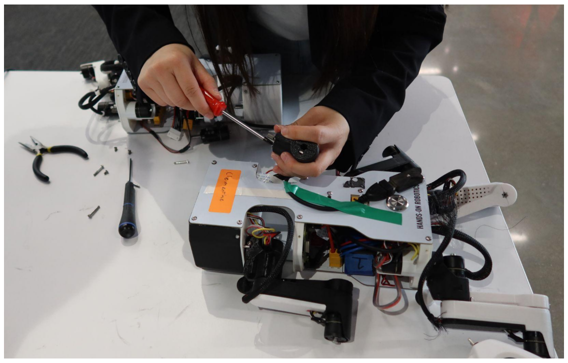


Figure 2. Replacing a Fractured Hip

We also encountered several issues when connecting the RasPi and the controllers which we ultimately determined were due to electrical connection problems produced by broken ribbon cables. To address this, we replaced the ribbon cables with thicker angled cables.

ROBOT SIMULATOR

Lab 7 was a short introductory lab for getting the simulator of the Pupper working. The main feature of this lab was the simulator which simulated the policies and control systems that can be implemented onto the actual robot. Additionally it was possible to actually walk around with said policies and controlling robot as one would with a controller. It should be noted that the controlling was done using a keyboard instead of a typical joystick controller, thus not an entirely one to one recreation. In terms of the actual policies to control the Pupper robot, there is a machine learning component that is addressed in lab 5. However, most changes to the control systems of the robot were done within the Config.py file with the various parameters highlighted in the lab.

We attempted this lab with both a Mac OS and a Windows OS. The Mac OS user encountered some issues and was not able to resolve them before focusing on another project. The Windows OS user was able to successfully run the simulator but required extra steps as highlighted below:

1. Install WSL2 or windows subsystem for linux

    a. Instructions can be found here: https://docs.microsoft.com/en-us/windows/wsl/install
    b. The specific successful distribution used was Ubuntu-20.04
    c. WSL must be enabled when doing simulator stuff like running installs and such, thus running the command *wsl* in the command prompt is necessary before doing other things
2. Install miniconda
    a. We are now following the steps on the Puppersim github or https://github.com/jietan/Puppersim
    b. This particular link had a short helpful guide on how to install miniconda through command prompt: https://educe-ubc.github.io/conda.html
3. Follow “Conda setup” on the Puppersim github
4. Setup some way to get program displayed outputs from WSL
    a. In short, if a program shows something on the screen, we need a way to see it as otherwise our only method of interfacing is the CMD prompt.
    b. This answer from stack overflow highlights everything needed: https://stackoverflow.com/a/66645230
        i. Disable VPNs, they may cause an issue
        ii. export LIBGL_ALWAYS_INDIRECT=0 may fix some issues if problems are occurring
5. Make sure X-server is running and you are using wsl before going on to the next steps
6. Follow “Getting the code ready” section on the simulator github
    a. If running into problems check below steps
7. To actually simulate and control the robot, follow the steps “Simulating the heuristic controller”
    a. This also needs to be cloned: https://github.com/stanfordroboticsclub/PupperKeyboardController
    b. Some dependencies are not installed correctly, so manually pip installing them may be required
        i. We observed these dependencies needed to be installed: matplotlib, gcc, quadprog, when trying to run run_djiPupper_sim

As a whole, this lab was successful. We were able to achieve a running simulator and the Config.py changed the properties of the control system of the Pupper. Thus at this point in time, it should be sufficient enough to start testing various changes and parameters and observing how they would influence the Pupper's gait. Eventually even testing those parameters on the actual Pupper in a real setting, hopefully improving the gait. We also inadvertently learned about the properties of the simulator, linux, and other various technical areas as we worked on setting up the simulator and getting it running. This should be a solid foundation for further work involving building and customizing the Pupper's gaits.

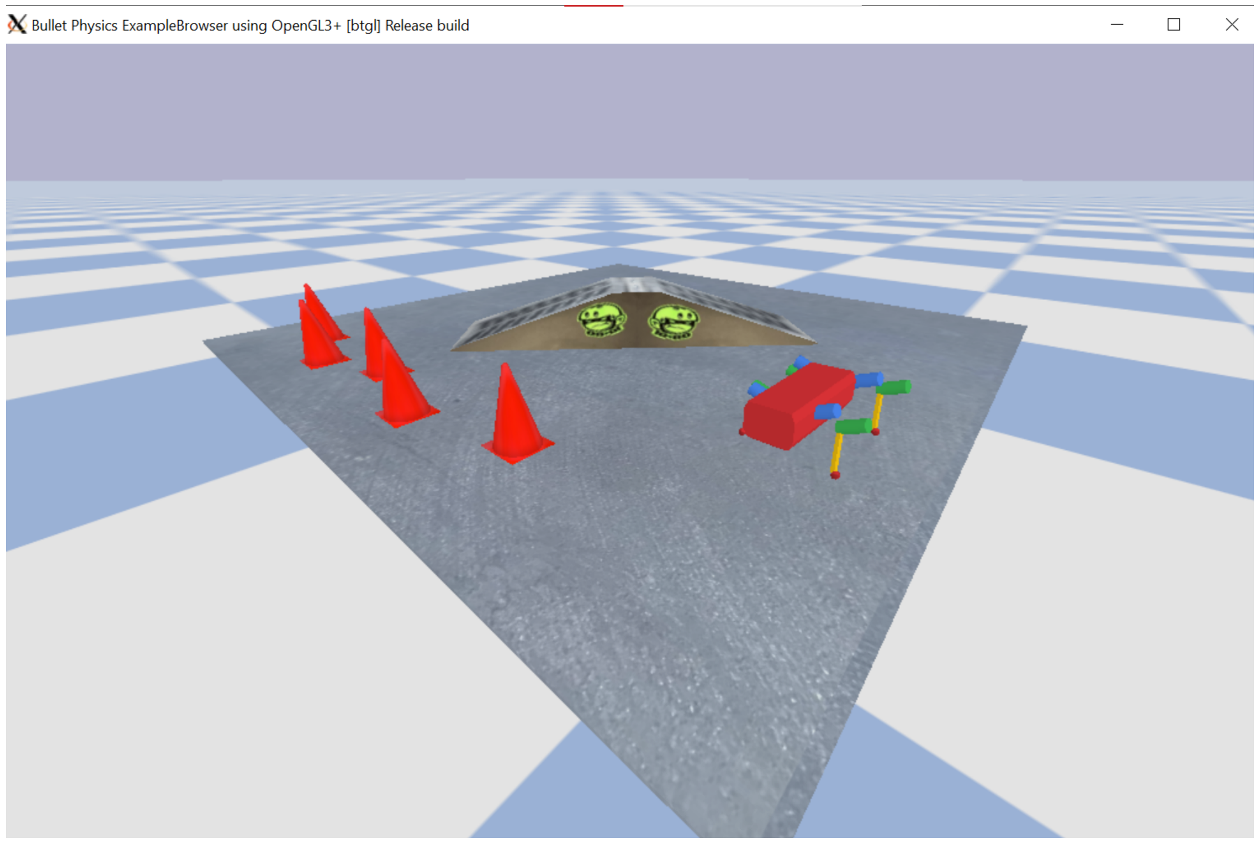

Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build

Figure 3. Simulator using PyBullet to recreate the Pupper robot and some additional obstacles

As for future work, some high-priority items would be getting the setup of the simulator working on Mac computers. Afterwards, double-checking all setup instructions and such are clear and reproducible in other computers and environments. Otherwise, two other important tasks are properly experimenting with the config variables and working with the machine learning component of the simulator.

The config variables would be interesting to test. First, it would be important to properly config that we are able to influence the Puppers gait. Second, learning how the variables influence the gaits would help understand the Pupper’s control systems and control systems as a whole.

The machine learning portion of the Pupper uses reinforcement learning to retrain the Puppers gait. Lab 5 was written to teach this but was only made available shortly before writing this report, so this lab and feature weren’t able to be explored. But this presents new opportunities to truly develop the Pupper’s gait and make the Pupper’s performance and build unique to WashU engineers.

PUPPER CALIBRATION AND INITIALIZATION

Pupper follows a motor calibration and initialization procedure each time it is switched to power on, independent of the type of power source (on-board battery or external power supply). Calibration works by rotating the motors until they encounter a physical stop, either a pin or another stable part of the leg. This location where the motor encounters a physical stop can be used as a consistent calibration point as it is a structural point of reference that is unlikely to change. When turning on the power, the Pupper will rotate the hip motors first until stopped. Next the knee motors will rotate and find their stopping point, finally the ankle motors will rotate and stop. Once all motors have been calibrated, the Pupper will be still with motors and legs locked in a specific position. After connecting the controller, the Pupper can initialize and stand up on all four legs.

We observed that the initialization procedure was highly inconsistent. Running a four trial experiment where the ground surface and starting position were held constant across the trials, we observed the following behavior.

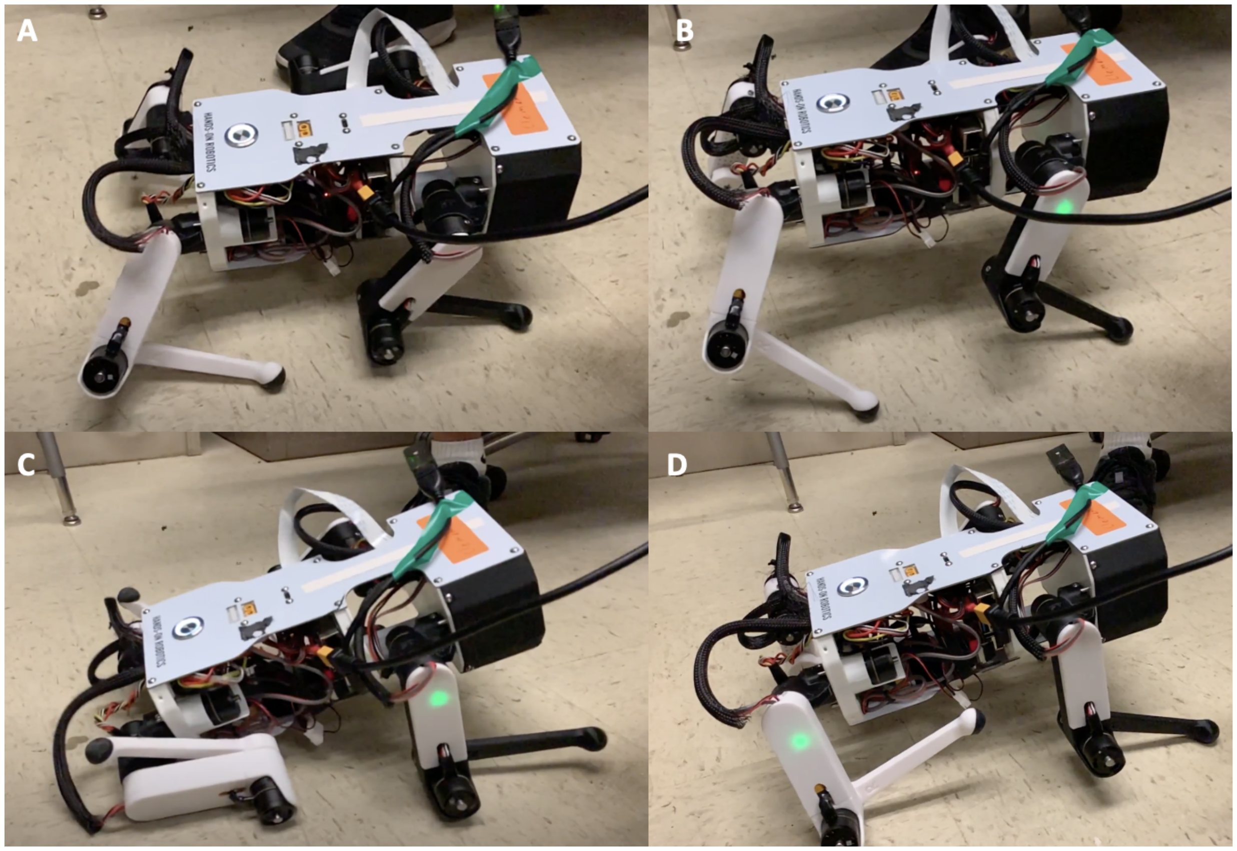


Figure 4. Two Tests of the Initialization Protocol. A) test 1 during initialization. B) test 1 end position. C) test 2 during initialization. D) test 2 end position.

We observe that a leg may initialize lower or higher than the other legs, leading to the Pupper tilting. This resulted in issues with walking where the entire leg scrapes the ground versus just the foot, or other issues such as falling over because the Pupper is no longer balanced. Other times, the leg initialized so poorly that it was impossible to achieve walking. As a consequence of this, experiments with the robot walking became impossible due to this inconsistency.

For troubleshooting why this issue was occurring, a few steps were taken. First, firmware programs for both of the Raspberry Pi and Teensy were double checked and redownloaded. Calibration did not change between successful initializations and unsuccessful initializations. Next, the back left hip and right hip motors were swapped out as they were observed to deviate most from the other motor movements. We did not observe any apparent effects. The third step involved editing a portion of the code. It was proposed by the Hands-on Robotics team that the code was not compatible with the Inertial measurement unit on board the Pupper and our issue could be solved by removing a specific line in the code that updated the IMU. This third step was not tested due to timeline constraints; however, it would be a good next troubleshooting step.

ROBOTIC ARM END EFFECTOR ATTACHMENT

As an expansion to the Pupper quadruped robot, we designed and built a multi-joint, five-degree-of-freedom robotic arm with the goal of eventually integrating it onto the Pupper's. The robotic arm base, bicep, and forearm were designed using SolidWorks and then 3D printed with polylactic acid (PLA) on a Original Prusa i3 MK3(S/S+) and LulzBot TAZ Workhorse. At each joint was a MG996R Metal Gear Torque Digital Servo motor and the gripper was controlled via two SG90 Micro servo motors. All motors were controlled using an Arduino Uno microcontroller and programmed in the Arduino IDE.

Our next steps fall into three categories 1) mechanically attach the arm onto the Pupper 2) link the arm with the existing Pupper electronics 3) alter the software controls. We would first replace the base and design a new base plate that is compatible with the Pupper's top plate and allows us to securely mount the arm onto the Pupper's back. Next, we would do proper research and electronic and circuit analysis to determine the most ideal mechanism for connecting these components. Adding this robotic arm would also add load to the overall system and we would finally address tuning the software control theory to account for the added load and torque produced by the arm movements.

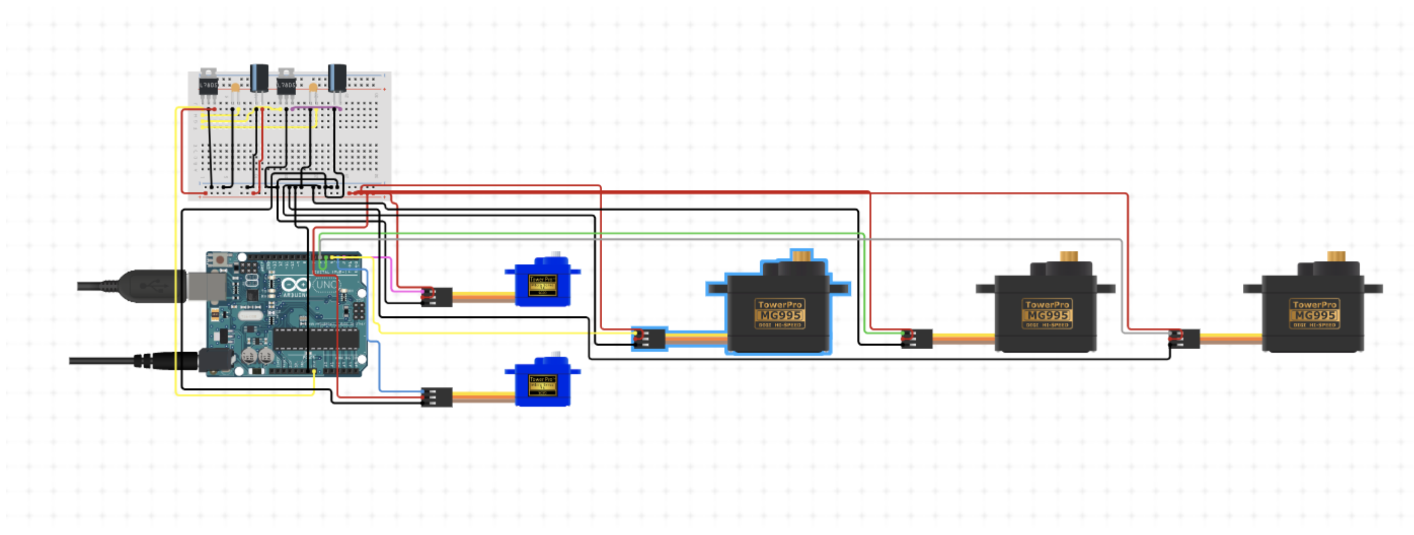


Figure 5. Robotic Arm Circuit Diagram

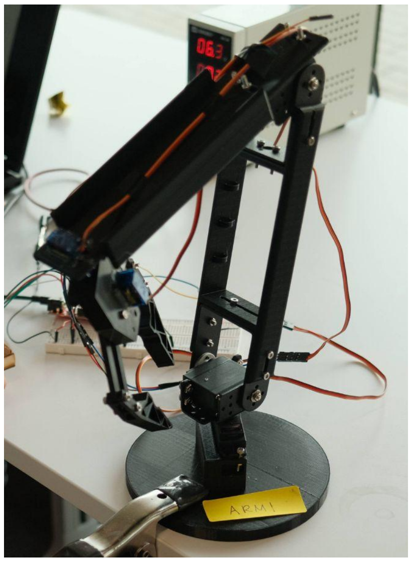


Figure 6. Robotic Arm Final Prototype

Table of Challenges

Challenge	Solution	Takeaway
Hip part had a high fracture rate	Print new hip part using via Prusa FDM printing with PLA	Switching to more robust printing, such as SLA or SLS, would be ideal
Transmitter to Receiver had faulty connections	Replace ribbon cables with thick, angled cables	Ribbon cables can be very unreliable and break connections easily
The initialization process was highly inconsistent	Attempted: standardize the process by maintaining a consistent initial Pupper position and surface level. Proposed but not tested: remove potentially interfering lines of code that update the IMU	A consistent initialization setup procedure is necessary for any robotic system. As systems get more complex, one must pay close attention to how one part of the system could influence the others.

CONCLUSION

The purpose of this project was to work through the Hands-On Robotics curriculum and provide feedback to their team about potential avenues for improvement. Their mission is to make robotics more accessible to students by providing a quadruped kit with a curriculum that can aid in learning various topics such as robotic gait design, motor control, robotic simulation, and more.[3] Through learning from the Hands-on Robotics Pupper Labs, we were able to familiarize ourselves with important forward and inverse kinematic principles, the assembly of the complex Pupper quadruped system, and the various parameters of a robotic simulator. We also began designing and building a robotic arm end effector to be mounted onto the Pupper but were not able to integrate it in time. We discovered some steps that could be taken to improve the overall Pupper design, as well as the build process. To address the robot failing during locomotion, we would print the hips more robustly. To address connection issues, we would replace the delicate ribbon cables with thicker cables. To further expand work with the simulator, we could utilize machine learning theories to improve Pupper gaits or create new gaits. As a whole, this project was extremely helpful as a first step toward understanding robotics at a professional and holistic level. Additionally, it is a solid foundation for continuing the journey to building greater and better robots.

REFERENCES

[1] Raibert, Marc H., “Legged Robots”, *Communications of the ACM.* Volume 29. No 6.

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

[3] *Hands-on Robotics*. https://handsonrobotics.org/

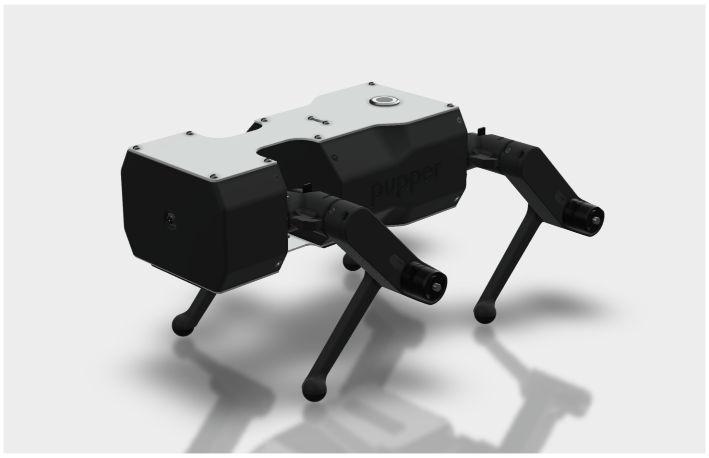
APPENDIX

A1: Pupper Bill of Materials

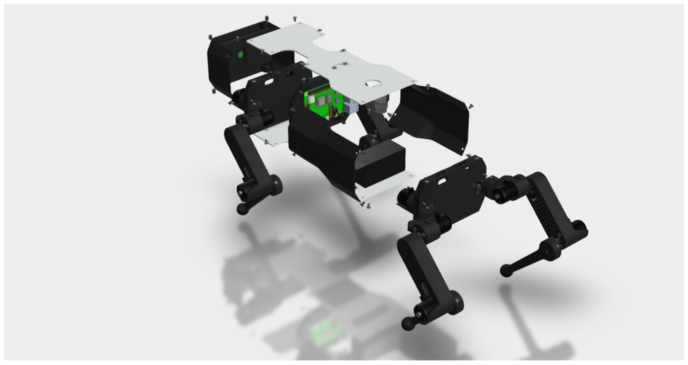
The table below shows the essential electronic hardware for the Pupper and the 3D printers used.

Part	Quantity	Purpose
DJI C610 Motor Controller	12	Attachment of PCB to Motor
DJI M2006 Gear Motor	12	Joint Motors
Top and Bottom PCB	1	Connecting Hardware and Power
1000mah 6s Lipo battery	2	Battery
Teensy 4.0	1	Arduino on PCB
Raspberry Pi 4	1	Attached to Teensy, PCB, Pi Cam, Transmitter
32GB MicroSD Card	1	Memory
Pi Camera v2	1	Pupper Head Camera
FRSKY XRS-SIM	1	Remote Control Receiver
BetaFPV Transmitter	1	Remote Control Transmitter
Ultimaker S3	1	3D Printer
Prusa i3 MK3	1	3D Printer

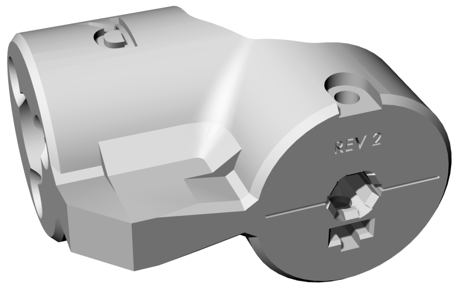
A2: CAD Photos of Full Robot and Hip



Pupper Assembly



Exploded View



Hip CAD File

A3: Transmitter and Receiver Pairing Sequence

To bind the RC receiver to the controller, the receiver was connected to a computer while holding the button down. The controller was then turned on for 5 seconds until a double vibration and its LED turned green. The transmitter's left joystick was then moved until it turned blue. The BIND button on the back of the controller was pressed where the controller blinks blue and red alternatively. The controller was then bound once the receiver turned a solid green color.

A4: Assigning Motor IDs

Each motor controller was assigned a unique ID number. To assign the motor ID, the button on the side of the motor controller was sequentially clicked a specific number of times. For example if we wanted to assign the number 3 as the ID for a motor, we would click the motor 4 times; if we wanted the number 5 as the ID, we would click the motor 6 times.

A5: Motor Calibration

Proper calibration of each motor was critical to the overall performance of Pupper. To calibrate a motor, the following procedure was followed:

a. Isolate the motor to calibrate and ensure the motor shaft doesn't have any load.
b. Position the motor so the shaft points vertically upwards. This should make the shaft move without any resistance from gravity or anything else, thus calibration should always be consistent.
c. Hold the button on the motor controller until the shaft starts moving ever so slightly, this means the shaft is beginning calibration.
d. Let calibration complete.