REPRODUCING CANINE GAIT IN A ROBOTIC QUADRUPED

Riley Karp
Thesis advised by Professor Caitrin Eaton
Colby College Department of Computer Science

As robots develop the capability to perform more complicated tasks, stable locomotion will become a basic required element of all mobile robots. While walking robots may have various numbers of legs, four legs provide a reasonably stable platform without requiring excessive amounts of power and resources that would be necessary to control robots with more limbs. It is useful to base the gait patterns of these robotic quadrupeds on existing quadrupeds in nature, such as dogs. This project explores three different joint control algorithms in simulation using Python3.7 and on a robotic dog controlled by an Arduino Uno. The three algorithms include an inverse kinematics solution based on a predetermined foot trajectory, a forward kinematic solution based on joint angles collected from biological data, and a waypoint solution, which was found to be optimal for this project. The robot described in this paper was created specifically as a tool to explore different motor control strategies for reproducing canine gait patterns in robotic limbs; however, these motor control concepts are applicable to more complicated limbed robots as well.

INTRODUCTION

Walking robots are becoming increasingly popular among roboticists as mobility becomes a more valuable aspect in advanced robots. Although bipedal robots are most humanoid, and multipedal robots are often most stable, quadrupedal robots provide more stability than a biped, while decreasing power consumption that would be required by a robot with more limbs, and therefore more joints and actuators. There are many quadrupedal animals existing in nature that may be used as a basis for gait patterns in robotic quadrupeds. One simple and commonly used quadrupedal example is a dog.

There have been multiple robots created based on dogs, and each of these robots has its own unique purpose. For example, BigDog created by Boston Dynamics is the size of a large dog and was their first rough terrain robot. The BigDog project was originally funded by DARPA and was created to be used as a tool among active duty military personnel to traverse through rough terrain with an animal-like walking pattern, while carrying supplies. BigDog was turned down by the military because its gasoline engine and hydraulic actuators created too much noise to be stealthy enough in military environments.[1]

Boston Dynamics learned from BigDog and developed a new pair of quadrupeds named Spot and SpotMini. These robots are electrically powered and are quieter than BigDog, but still maintain some of the stability in rough terrain that was developed in BigDog. Spot operates both indoors and outdoors, while SpotMini is meant to help inside homes and offices by opening doors and using its arm to carry things and perform more advanced tasks.[2], [3] Although these robots are

stable and may be useful in carrying relatively large loads, they do not resemble dogs (**Figure 1**) and would not be very practical as a personal canine-like companion.

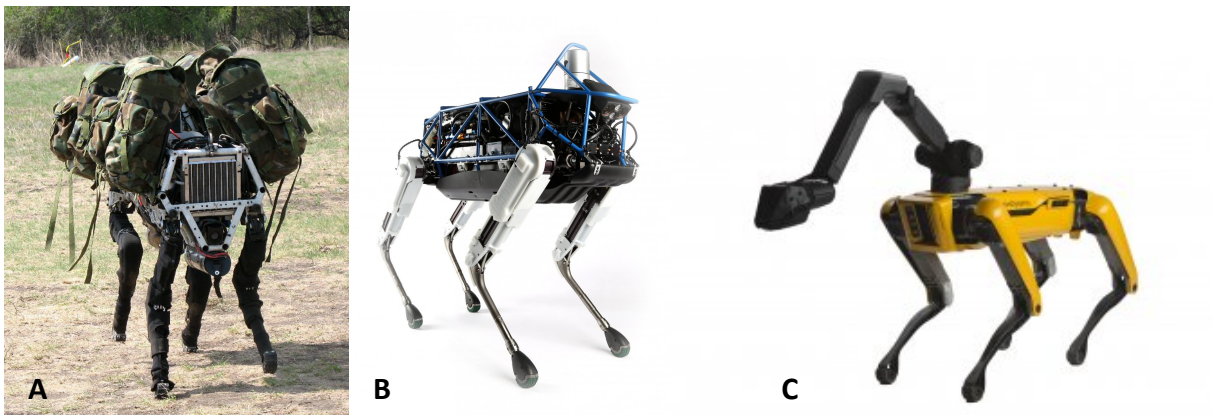

Figure 1. Boston Dynamics quadrupeds. A) BigDog, B) Spot, C) SpotMini

Sony created a robotic dog named AIBO, which means "companion" in Japanese and is also an acronym for Artificial Intelligence roBOt. AIBO was created to act as a canine-like companion and it uses artificial intelligence to learn different behaviors based on its environment and its owner's actions. This cute robotic dog is so life-like that it can express emotions using its eyes, ears, and body language (see **Figure 2**). It has sensors that allow it to react to touch, vocal commands, and visual cues just like a real dog. While many technological components allow AIBO to behave like a realistic canine companion, its 22 degrees of freedom require complicated precise control algorithms to recreate life-like motions.[4] For newer roboticists interested in creating their own companion, a simpler robotic quadruped would be a better starting point.

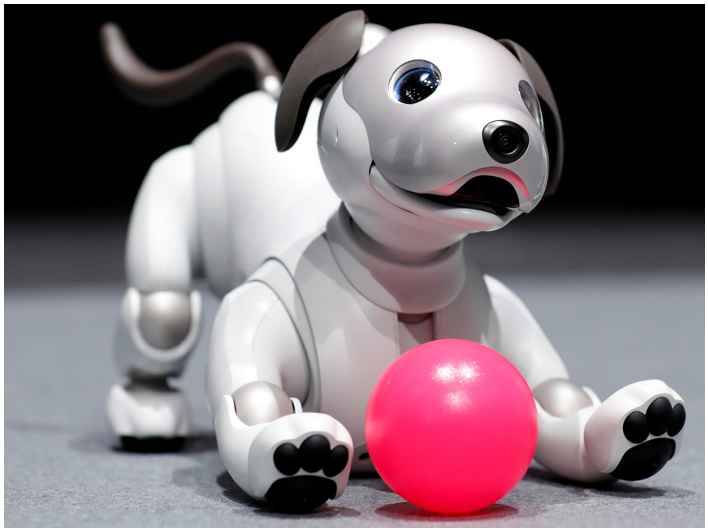

Figure 2. Sony AIBO playing with its ball toy.

The Nybble cat created by roboticist Ronzhong Li is clearly not a dog, but it is a good example of a simple open source robotic quadruped. Nybble is the lightest and fastest robotic cat that walks, and it includes two degrees of freedom in each of its four limbs, which are easily controlled by an Arduino board. This open source cat was designed to be a fun way for new programmers to become

familiar with robotics.[5] Nybble's lightweight plywood body (seen in **Figure 3**) and the simple servo motor control with an Arduino, inspired the design of the robotic dog created for this thesis project.

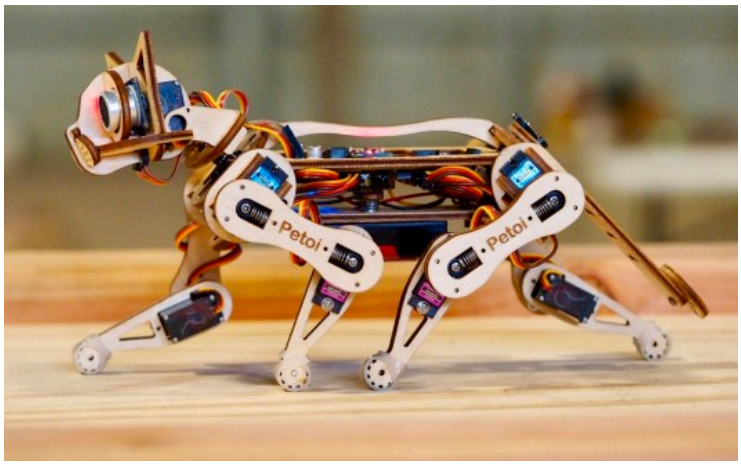

Figure 3. Nybble cat designed from laser cut plywood pieces and 2 servo motors in each leg.

The robotic dog created for this project combines the simplicity exemplified by the Nybble cat with the realistic companionship achieved by the Sony AIBO dog. Three degrees of freedom in each of the dog's four legs allow for a more realistic canine gait, while the light plywood body and servo motor joints allow for simple motor control using and Arduino board. Although this dog was intended to be used as a tool to explore different motor control strategies for reproducing canine gait patterns in robotic limbs, these motor control concepts are applicable to more complicated limbed robots as well. The three motor control strategies explored here are an inverse kinematics solution, a forward kinematics solution, and a waypoint solution. This paper will discuss key aspects of the design of the robotic dog created for this project, the methodology and results of the different motor control strategies that were attempted in simulation and in practice, and finally improvements and applications of these findings to locomotive robots in general.

METHODS

Robotic Dog Design

The dog created for this project consisted of body parts laser cut from 1/8" plywood, three 180^{o} micro servo motors in each of the four legs, an Arduino Uno with an Atmega328p chip and a 12-bit servo shield to control the motors, and a 5V power source to power the motors. The body parts were designed using FreeCAD v0.17, and laser cut using a Full Spectrum laser cutter. CAD models of each of the body parts can be seen in **Figure 4**. The plywood parts were connected with wood glue, and the motors were attached to the legs and body using the motor horns and screws that came with each motor. A full list of parts can be found in **Table 1**.

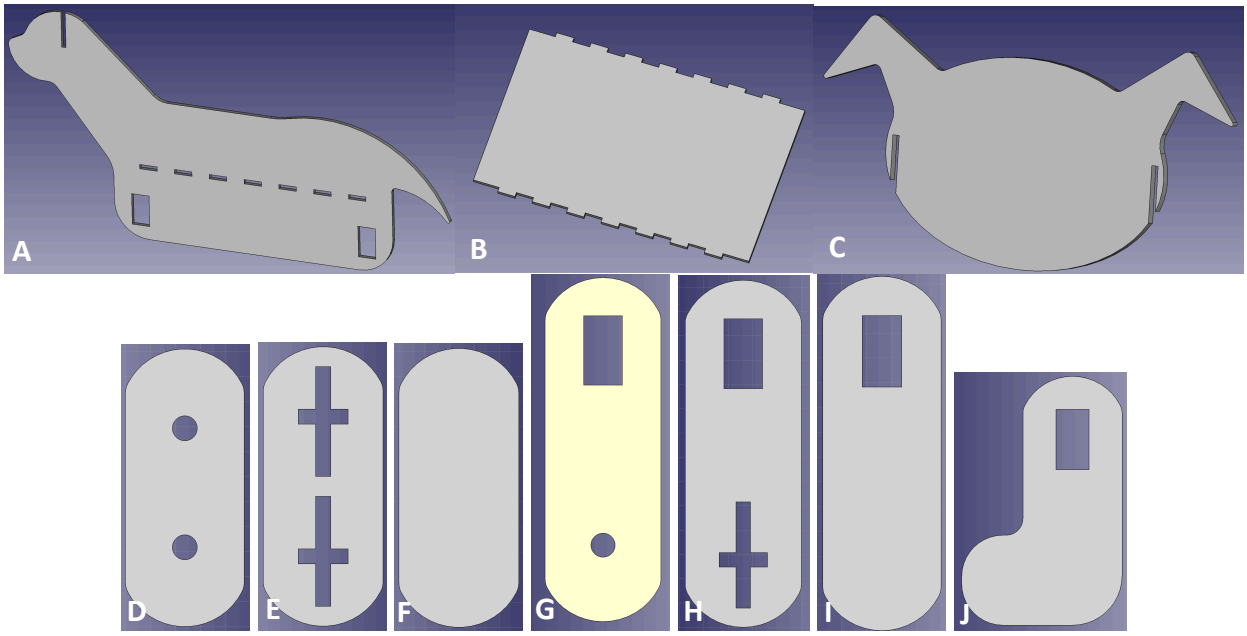

Figure 4. Body parts that were designed in FreeCAD and laser cut into 1/8" plywood. A) Body side (2 copies required for full dog), B) Body base, C) Head, D-F) 3 layers combined to form one leg segment connecting hip to knee motor, G-I) 3 layers combined to form one leg segment connecting knee to ankle motor, J) one of three identical layers of one segment that connects the ankle motor to the foot

Table 1. Parts List

Part	Notes
1/8" plywood (~4.5 sq ft)	For body parts and legs
Wood glue	To attach wooden body parts
High power, high torque, metal gear micro servo motors (x12)	3 joints in each of the 4 legs
Arduino Uno	Send PWM signals to the servo motors
16 Channel, 12-bit PWM/servo motor shield	Control all 12 motors with one Arduino Uno
5V, 10A switching power supply	Power the servo motors
Female DC power adapter	Connect the power supply to the motor shield
Insulated copper wires	Connect the power supply to the motor shield
Full Spectrum Laser Cutter	Cut the body parts into plywood
Computer with the Arduino IDE and FreeCAD installed	Write and run code to control the motors, and design the body parts to be laser cut
Googley eyes (optional)	Add a face to the dog

The parts listed above were combined to create the completed robotic dog, which can be seen in **Figure 5** below. The Arduino Uno with the motor shield attached was secured with screws to the underside of the dog's base, and the 12 motors were then connected, along with the power source and the data cable.

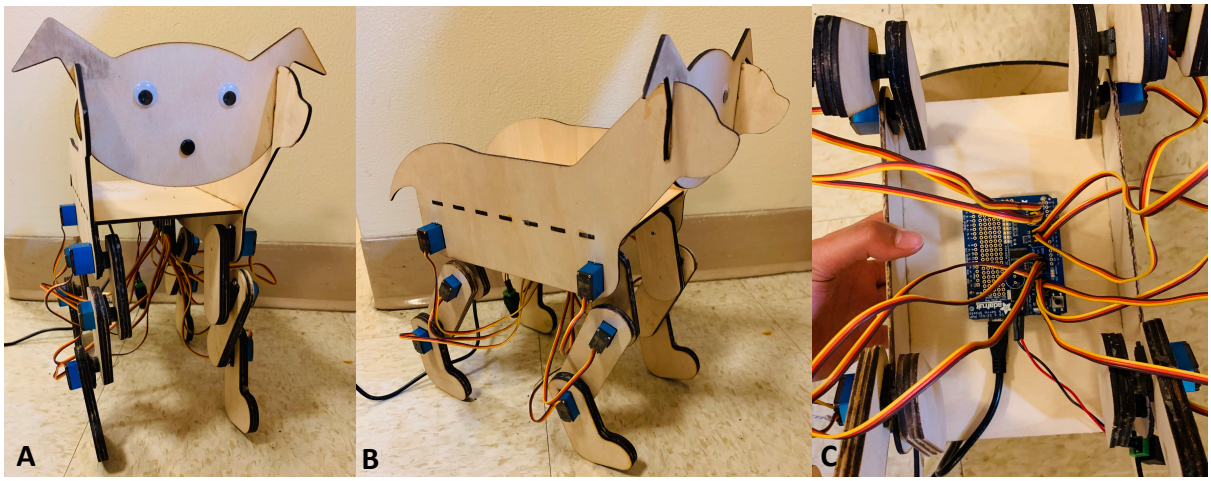

Figure 5. Completely assembled robotic dog. A) Front view, B) Side view, C) Bottom view

Kinematics of Limb Control

In order to make the robotic dog walk, it was easiest to first focus on controlling the movement of one limb. There are a couple different ways to control the limb motion. The first is controlling the end effector position and using inverse kinematics to calculate the joint angles that will get the end effector to the desired position. The second option is controlling the joint angles and calculating the resulting end effector position using forward kinematics. Versions of both options were applied to the dog in this project. There are also a couple different ways to perform the calculations to convert between joint angles and end effector positions, including the trigonometric approach and the jacobian approach.

Trigonometric Approach

The trigonometric approach for calculating joint angles and end effector positions involves creating imaginary triangles at each of the joints. For inverse kinematics calculations, a joint angle can be calculated given an end effector position using the inverse tan equation (**Eqn. 1**) if the triangle formed by the limb is a right triangle (**Figure 6A**). If the triangle formed by the limb is not a right triangle (**Figure 6B**), then the law of cosines may be used to calculate joint angles based on known limb lengths and positions (**Eqn. 2-4**).

(Eqn. 1) $a = \tan^{-1}(y/x)$

(Eqn. 2) $a^2 = b^2 + c^2 - 2bc*\mathbf{cos}(A)$

(Eqn. 3) $b^2 = a^2 + c^2 - 2ac*\mathbf{cos}(B)$

(Eqn. 4) $c^2 = a^2 + b^2 - 2ab*\mathbf{cos}(C)$

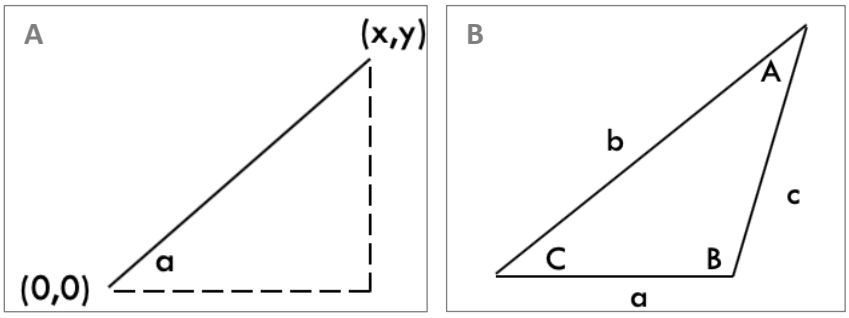

Figure 6. Triangles used for the trigonometric approach to inverse kinematics. A) Right triangle used with **Eqn. 1**, B) Scalene triangle used with **Eqn. 2-4**

For forward kinematics calculations, the end effector position can be calculated given a joint angle *a*, and known limb length, *r*. Given an angle, the end effector's x-position can be calculated using **Eqn. 5**, and the y-position can be calculated using **Eqn. 6**. The imaginary triangle formed by the limb and joint in this instance can be seen in **Figure 7**.

(Eqn. 5) $x = r*\mathbf{cos}(a)$

(Eqn. 6) $y = r*\mathbf{sin}(a)$

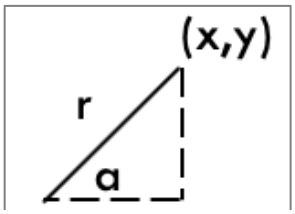

Figure 7. Triangle used for the trigonometric approach to forward kinematics.

The trigonometric approach to kinematics worked well for precisely calculating the desired joint angles and end effector positions; however, it required many tedious calculations that had to be repeated for each of the three joints in each leg. Furthermore, one set of calculations for a leg only resulted in joint angles or an end effector position for one single leg configuration. Since a walking pattern requires the leg to move through multiple configurations, performing these trigonometric calculations for every joint in every configuration would have required many hours of work. Additionally, with so many calculations, there was a lot of room for error if one single negative sign or decimal point was out of place in one of the many calculations. After performing some of these calculations, it was quickly decided that the jacobian approach, described below, would be superior to the trigonometric one.

Jacobian Approach

The jacobian approach to kinematics requires multiple matrix calculations in order to obtain the jacobian matrix; however, the same jacobian can then be used to conduct kinematics calculations for all joints in the limb, and for any limb configuration. There are three steps to finding the jacobian matrix before it can be used in kinematics calculations. First, the Denavit-Hartenberg Parameters (DH parameters) for the limb must be determined. For this project, a version of DH parameters called the modified DH parameters were used.[6] These parameters define how the segments of the limbs are organized in relation to each other based on the angles and distances

between the segments. Since each leg of the robotic dog in this project is the same, the DH parameters are the same for each leg. These parameters can be seen in **Table 2**.

Table 2. Modified Denavit-Hartenberg parameters for this robotic dog's legs

Joint	α_{i-1}	a_{i-1}	d_i	θ_i
1. hip	0	0	0	θ_1
2. knee	0	1.5"	0	θ_2
3. ankle	0	2.5"	-0.75"	θ_3
4. toe	0	2.75"	-0.75"	0

The next step in finding the jacobian is plugging each row of the DH parameters table into the transformation matrix equation (**Eqn. 7**) to determine the orientation and position of each joint's frame with respect to the frame of the preceding joint. For example, transformation matrix ${}^{0}_{1}T$ describes the orientation and position of the hip in the world's frame. The overall transformation matrix, ${}^{0}_{4}T$, which describes the orientation and position of the toe in the world's frame can be found by multiplying each of the four individual transformation matrices together in order, as described by **Eqn. 8**. The first three entries in the last column of the final transformation matrix are the x, y, and z positions of the end effector in terms of the world frame's coordinates.

(Eqn. 7)
$$ {}^{i-1}_{i}T = \begin{bmatrix} cos\theta & -sin\theta & 0 & a \\ sin\theta * cos\alpha & cos\theta * cos\alpha & -sin\alpha & -d * sin\alpha \\ sin\theta * sin\alpha & cos\theta * sin\alpha & cos\alpha & d * cos\alpha \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

(Eqn. 8)
$$ {}^{0}_{4}T = {}^{0}_{1}T * {}^{1}_{2}T * {}^{2}_{3}T * {}^{3}_{4}T $$

The final step in calculating the jacobian is finding the partial derivatives of the positional terms of the final transformation matrix, with respect to each of the three angle variables. The form of the jacobian matrix can be seen in **Eqn. 9**. All of these matrix calculations were done manually and then confirmed using the numpy and sympy packages for Python3.7.

(Eqn. 9)
$$ J = \begin{bmatrix} \frac{d}{d\theta 1}x & \frac{d}{d\theta 2}x & \frac{d}{d\theta 3}x \\ \frac{d}{d\theta 1}y & \frac{d}{d\theta 2}y & \frac{d}{d\theta 3}y \\ \frac{d}{d\theta 1}z & \frac{d}{d\theta 2}z & \frac{d}{d\theta 3}z \end{bmatrix} $$

Once the jacobian, J, was found for the limbs of this specific robotic dog, the matrix could then be used in inverse kinematics calculations to determine changes in joint angles given changes in end effector position (**Eqn. 10**), and in forward kinematics calculations to determine changes in end effector position given a set of joint angles (**Eqn. 11**).

(Eqn. 10) $\Delta angles = J^{-1} * \Delta position$

(Eqn. 11) $\Delta position = J * \Delta angles$

The jacobian was used to create three different control algorithms for creating limb motion. The first was an inverse kinematics solution that found joint angles based on a predetermined triangular foot trajectory. The second algorithm was a forward kinematics solution that controlled the limb using joint angles taken from biological beagle data.[7], [8] The final solution was a waypoint solution that combined information from the previous two solutions to create an optimal algorithm. The details and behavior of each algorithm are discussed in the results section.

Limb Control in Simulation

Each of the three limb control algorithms were simulated using Python3.7. The Python libraries numpy and sympy were used for kinematics calculations, and matplotlib was used to visualize the movement of the limb. The general control flow for the inverse kinematics simulation started with a target position being chosen from a predetermined foot trajectory. The position was multiplied by the inverse jacobian matrix to get the resulting joint angles that would put the end effector at the desired position. The limb segments were plotted at those angles using matplotlib, and then the end effector position was recalculated, and the process was repeated. For the forward kinematics simulations, the general control flow began by chosing the desired joint angles based on a predetermined pattern, and then simply finished by drawing the limb segments at those angles. Then the next set of joint angles were found, and the drawing process was repeated.

Servo Motor Control

The joints in the robotic dog were 180° micro servo motors whose angular positions were controlled by pulse width modulated (PWM) signals. An Arduino Uno with a 16MHz Atmega328p microcontroller chip was used to control the motors; however, a servo motor shield was required to be placed on top of the Arduino Uno since the atmega328p did not have twelve timers that were needed to control the twelve motors. The 16 channel, 12-bit PWM servo motor shield used in this robot communicated with the Arduino Uno via I2C protocol. The Arduino IDE was used to program the microcontroller using C++. The motor shield's library was used to set the frequency of the PWM signals to 60Hz, and to set the duty cycle of each of the signals to control the position of the servo motors attached to the motor shield. The duty cycle was set between 4% and 12%, where a 4% duty cycle corresponded to the motor being at its minimum of 0°, and a 12% duty cycle corresponded to the motor being at its maximum around 180°. To control the robotic limb based on a predetermined foot trajectory, a target position was chosen, and the inverse jacobian was used to calculate the joint angles that would get the foot to the desired position, just like in the simulation. These joint angles then had to each be converted to pulse lengths that would create a PWM signals with the proper duty cycles to move the servo motors to the calculated joint angles. Finally, the PWM signal was sent to the motor driver, which then moved the limb to the desired angular position. The target foot position was updated, and the process repeated itself until power was no longer supplied to the system. For the forward kinematics control algorithms, the predetermined set of joint angles was used to choose the limb configuration in each iteration, instead of using a predetermined foot trajectory. This eliminated the jacobian calculation step that was required in the inverse kinematics control algorithm. The angles were simply converted to PWM signals that were sent to the motors via the motor driver, and the process was repeated with updated target joint angles.

RESULTS

Inverse Kinematics Solution

The inverse kinematics limb control algorithm used a series of target positions that followed a predetermined foot trajectory. A triangular foot trajectory with rounded corners was chosen because this shape was shown to be common among walking quadrupeds.[9] The simulated foot trajectory pattern can be seen in **Figure 8**. For each time step, if the actual end effector position was more than 0.4" from the target position, but still getting closer, then the limb configuration was updated using the jacobian. To update the limb configuration, first the jacobian was calculated based on the current joint angles. Next, the inverse jacobian was calculated using numpy. The end effector velocities were then calculated to be 10% of the distance between the current and target end effector positions. The inverse jacobian was right multiplied by the end effector velocities to find the amount by which each joint angle should be adjusted to progress the end effector towards the desired target position. Finally, the new joint angles and new current end effector positions were calculated. If the foot was within 0.4" of the target position, then the target position was updated to the next position in the predetermined foot trajectory. The entire process was repeated until the user ended the simulation.

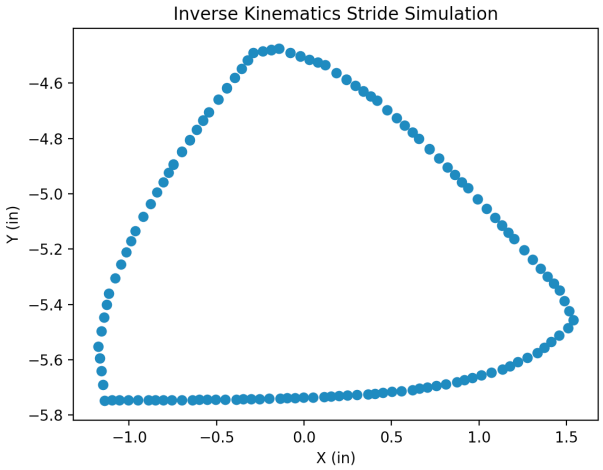

Figure 8. Foot trajectory of the limb in the inverse kinematics simulation.

This inverse kinematics solution seemed to work well in simulation; however, the Arduino Uno microcontroller was unable to handle all of the required calculations. Furthermore, it was difficult to find a way to perform matrix multiplication and matrix inversion to perform these calculations on the Arduino. Instead, the inverse jacobian was found using sympy for Python3.7, and was hard coded into the Arduino Uno as separate functions for each term of the matrix. Linear combinations of these terms were created to virtually calculate each matrix multiplication output by hand, since matrix calculations could not easily be completed on the Arduino. This process, along with creating the PWM signals required to drive the motors, was too computationally expensive for the 16MHz microcontroller to handle, resulting in the lack of motion in the motors. To ameliorate this problem, a forward kinematics solution was developed to eliminate the matrix inversion and multiplication steps that were required in the inverse kinematics solution.

Forward Kinematics Solution

The forward kinematics limb control algorithm used joint angles based on biological data collected from beagles[7], [8]. Data points were visually extracted from the reported plots of joint angle vs. stride duration for the hip, knee, and ankle joints of beagles (**Figure 9**). Numpy and matplotlib for Python3.7 were then used to fit a polynomial to the data for each of the three joints. In all three cases, a sixth order polynomial fit the best. A float counter that remained between 0 and 1 was used to keep track of the stride phase, and this phase was used as input into the joint angle polynomial functions, which produced the angles at which to draw the limb segments at each time step of the simulation.

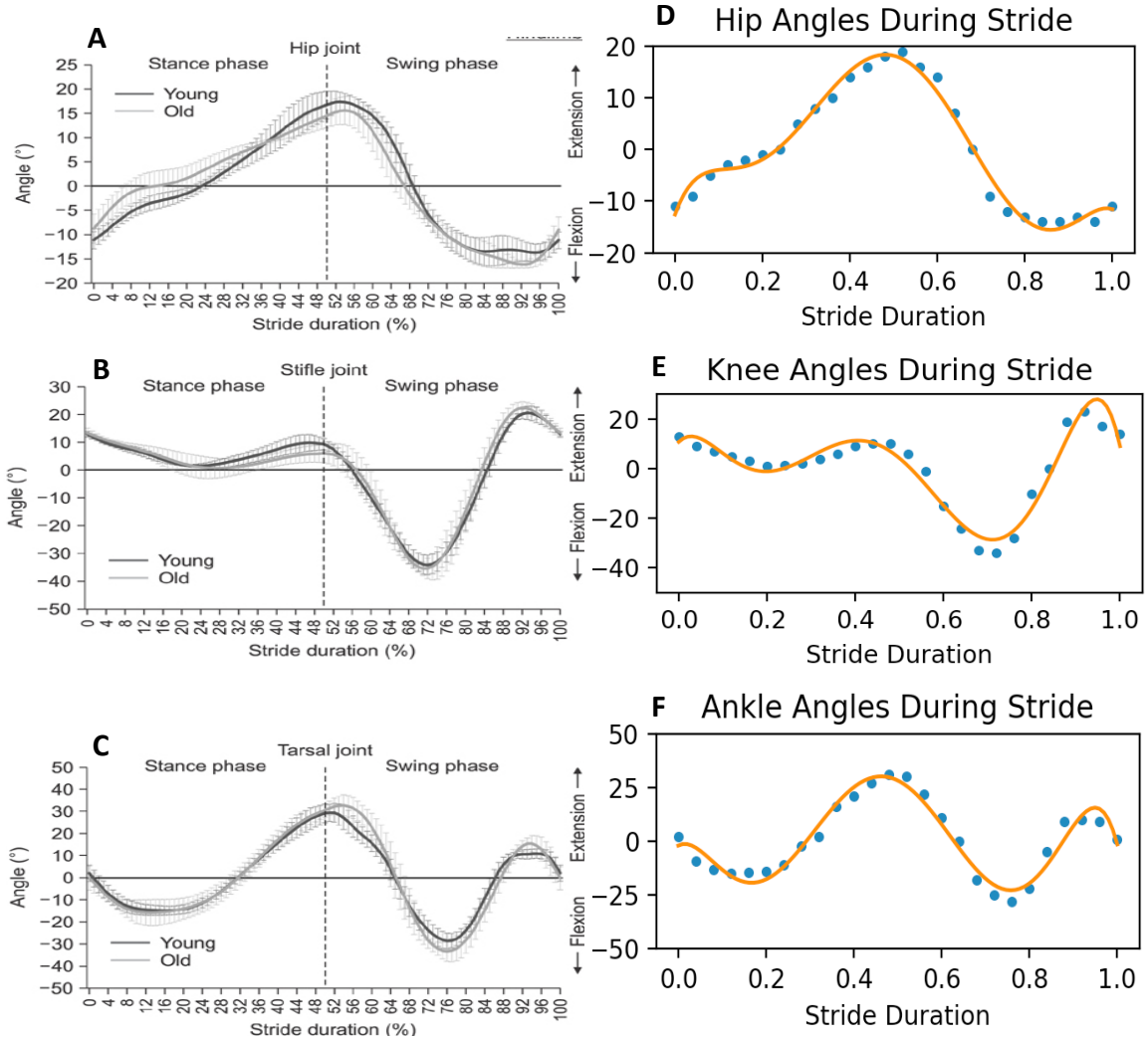

Figure 9. Joint angles vs. stride duration. A-C) Experimental beagle data[7], [8], D-F) Extracted data fitted with sixth order polynomials for each joint

The simulated foot trajectory that resulted from using these biologically determined angles as input was parabolic (**Figure 10**). A similar trajectory occurred when the forward kinematics control algorithm was used on the Arduino to control the physical robotic limb.

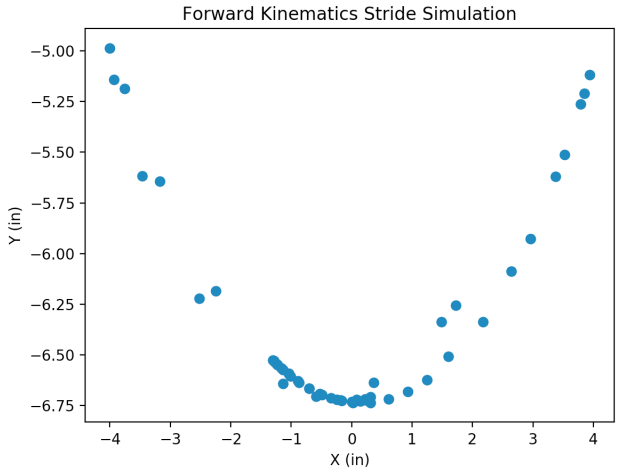

Figure 10. Foot trajectory of the limb in the forward kinematics simulation.

Once this control algorithm was implemented on the Arduino Uno to control the robotic limb, it was especially clear that this stride pattern would not produce a smooth walking gait when applied to all four limbs. The leg swung back and forth along almost the same path, which did not leave enough space between the foot and the ground for the foot to be lifted and moved forward to take a step. Additionally, the foot was spending a large portion of its stride in the "swing" phase, where it is supposed to be in the air, and a short period of time in the "stance" phase, where the foot is touching the ground. This is because the researchers who collected beagle angle data normalized their data points by time, so that exactly 50% of the stride duration was comprised of joint angles in the swing phase, and 50% was from joint angles in the stride phase.[7], [8] It was difficult to unnormalize the beagle data, and it was unclear exactly how long each section of the stride should take, which led to the development of the waypoint solution for limb control.

Waypoint Solution

The waypoint solution for limb control combined parts of the inverse kinematics and forward kinematics approaches. By examining the end effector positions that resulted from using the biological angles used in the forward kinematics solution, some reasonable target foot positions were determined, and a foot trajectory was created using thirteen discrete waypoints. Six points were chosen to be in stance phase, while five were chosen to be in swing phase. The remaining two points were at the ends where the foot transitioned between the two phases. These thirteen target end effector positions were then plugged into the inverse kinematics simulation to find the joint angles that would put the end effector at the desired positions. These calculated angles were stored in three lists, one for each of the three joints, containing thirteen angles each. The foot trajectory that resulted from drawing the limb segments at each of the thirteen waypoint configurations can be seen in **Figure 11**. This control algorithm and the thirteen discrete angular positions created a smooth stride pattern when implemented on the Arduino Uno to control the physical robotic limb.

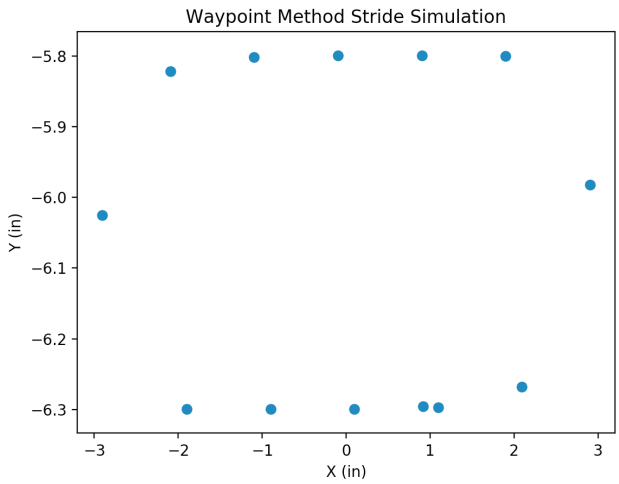

Figure 11. Foot trajectory of the limb in the waypoint method simulation.

Walking Algorithm

Once the limb control algorithm was working properly on each of the four robotic limbs, the next step was to make sure the limbs were working together to propel the body forward effectively. It was determined by experimentation that the most stable walking pattern for this robot allowed only one foot to be off the ground at any given time. The order in which each foot was lifted off the ground was determined by biological data collected from live dogs.[10], [11] In the stance pattern determined by the dog research (**Figure 12**), the right hind and right front feet are simultaneously off the ground for a short period of time. This simultaneity was eliminated for the robotic dog in this project to maintain a balanced walking platform. However, the basic stepping pattern was the same where the lift hind foot was lifted first, followed by the left front foot, then the right hind foot, and finally the right front foot.

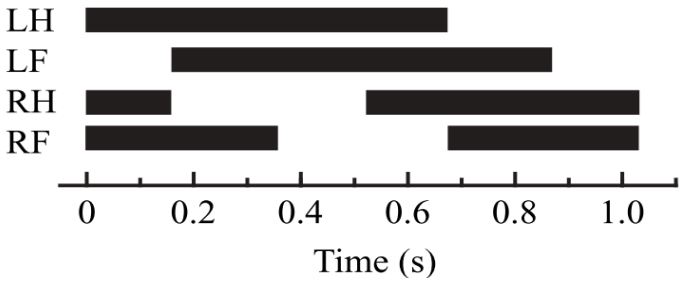

Figure 12. Foot swing pattern based on biological dog data[10]

The walking algorithm used for the robotic dog in this project is a fairly simple infinite loop that continues until power is no longer supplied to the motors or the microcontroller. First, the joint angles are initialized to the rest position, which corresponds to the first of the thirteen waypoints in the lists of joint angles. The target angles are initialized to the second waypoint in the lists of angles. Then the infinite loop starts, which slowly increments the angles of each joint by a small amount until the joints are within about 4.6° of their target angular position. At this point, the target angles are updated to be the next waypoint in the lists of waypoints. The stride state is also updated, which determines which of the feet is currently off the ground. This process continues as long as

power is supplied to both the microcontroller and the motors. The general layout of the walking algorithm can be seen below:

```
Initialize joint angles to the first element in angles lists
Initialize target angles to the second element in angles lists
while(true):
    for each joint:
        Increment current angle by 5% of angular distance to target angle
        Set PWM to move joint to new angle
    if all joints are close enough to target angles:
        Update stride state
        Update target angles
```

DISCUSSION

The final motor control algorithm that created the most realistic canine-like gait pattern in this robotic dog was the waypoint method, which used waypoints based on biological data. This method reduced the computational power required by the inverse kinematics solution, and created a smoother step pattern than the forward kinematics approach which directly mimicked the joint angles seen in live dogs. Live dogs have legs with tendons and muscles that are significantly different from the wooden limbs and servo motor joints that created this simplified robotic dog, which may have contributed to the unusual step pattern created by the biologically-based forward kinematics solution. The inverse kinematics solution may have been successful given a more powerful microcontroller. Furthermore, the inverse kinematics would be beneficial in more complicated robots, especially ones with sensors that provide feedback to the robot about its environment. Using inverse kinematics would allow the robot to calculate its next move on the fly based on external feedback, instead of having its movements restricted to a set of fixed waypoints or biologically-inspired joint angles. Future iterations of this project would include adding sensors to the robot to provide feedback about the robot's surroundings, and adding a stronger microcontroller that would enable the implementation of the inverse kinematics solution on the robot instead of just in simulation. The motor control strategies explored in this project have useful applications in other canine-like robots and advanced quadrupeds, and can be adjusted for multipedal robotic locomotion as well.

ACKNOWLEDGEMENTS

This work would not have been possible without the guidance of Professor Caitrin Eaton and the support and resources allocated by Professor Bruce Maxwell and the Colby Computer Science Department. Tim Stonesifer and the Muleworks lab staff were also invaluable resources while constructing the physical robotic dog.

REFERENCES

[1] Boston Dynamics, "Big Dog," 2019. [Online]. Available: https://www.bostondynamics.com/bigdog.

[2] Boston Dynamics, "Spot," 2019. [Online]. Available: https://www.bostondynamics.com/spot-classic.

[3] Boston Dynamics, "SpotMini," 2019. [Online]. Available: https://www.bostondynamics.com/spot.

[4] SONY, "Aibo," 2019. [Online]. Available: https://us.aibo.com.

[5] R. Li, "Introducing Nybble," 2018. [Online]. Available: https://www.petoi.com.

[6] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Pearson Education, 2009.

[7] J. Abdelhadi, P. Wefstaedt, I. Nolte, and N. Schilling, "Fore-Aft Ground Force Adaptations to Induced Forelimb Lameness in Walking and Trotting Dogs," *PLoS One*, vol. 7, no. 12, pp. 1–8, 2012.

[8] M. Lorke *et al*., "Comparative kinematic gait analysis in young and old Beagle dogs," *J. Vet. Sci*., vol. 18, no. 4, p. 521, 2017.

[9] W. Vagle, "The ABC's of Walkers, Big and Small." [Online]. Available: https://www.diywalkers.com/walker-abcs.html.

[10] T. M. Griffin, "Biomechanics of quadrupedal walking: how do four-legged animals achieve inverted pendulum-like movements?," *J. Exp. Biol*., vol. 207, no. 20, pp. 3545–3558, 2004.

[11] A. S. Aristizabal Escobar, A. N. A. de Souza, A. C. B. de Campos Fonseca Pinto, and J. M. Matera, "Kinetic gait analysis in English Bulldogs.," *Acta Vet. Scand*., vol. 59, no. 1, p. 77, Nov. 2017.