UNIVERSITY OF CENTRAL FLORIDA

Implementation of Quadrupedal Gait For the Unitree Go1 Robot

EEL 5669 Autonomous Robotic Systems

Fall 2024 Class

12/09/2024

Jarett Artman, Eric Frankle, Brandon Gross, Steven Keller, John Piergallini, Andrea Styles

ABSTRACT

Our team implemented and tested different walking gaits for the unitree Go1 quadruped robot. We developed control systems for walking, trotting, cantering, and galloping gaits using ROS (Robot Operating System) and custom trajectory planning. The project focused on creating smooth transitions between gaits while maintaining gait stability through velocity scaling and phase synchronization. We successfully implemented walking and trotting gaits in both simulation and physical testing, with the robot achieving stable forward motion at speeds up to 0.5 m/s. While full gallop implementation proved challenging due to stability issues, we established foundational control architecture for all four gaits. Some of our key achievements include ROS control integration and a velocity scaling system that allowed for smooth speed transitions. Testing revealed the importance of proper startup sequences and tuning of parameters for optimizing performance. This work demonstrates the feasibility of implementing natural inspired gaits on the Go1 platform.

I. BACKGROUND

The Unitree Go1 robot is a quadruped four-legged robot created by Unitree Robotics that resembles a dog [1]. The robot consists of a main body and four sets of identical legs. Each leg consists of three links and motors: a body motor, a thigh motor, and a knee motor. The body and thigh motors have 23.70 N*m of torque each while the knee motors have 35.55 N*m of torque each. The robot weighs 12 kg (~26.45 lb) and is approximately 645 x 280 x 400 mm in size when standing [2]. The Go1 is capable of carrying a payload of up to 5 kg (~11.02 lb) and can reach a top speed of 3.7 m/s (~12.14 ft/s). It has a 6,000 mAh battery capable of lasting up to 2.5 hours of use. The robot contains a variety of sensors including LIDAR, ultrasonic, IMU, and motor encoders. The Go1 is run off a combination of Jetson Nano or Jetson NX processors.

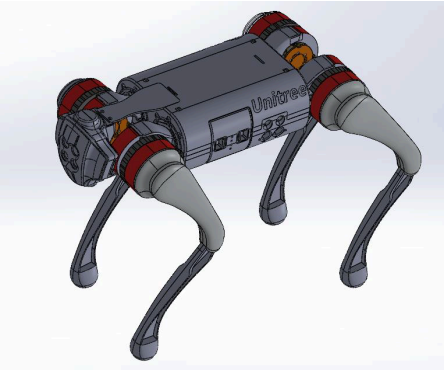

Figure 1: Unitree Go1 CAD Model

Quadrupedal robots such as the Unitree Go 1, are designed to mimic the movement of four legged animals. This enables them to navigate a variety of terrains with stability and control. These robots are built to perform different gaits, each optimized for varying speeds and terrains. The gaits include

crawling, trotting and galloping, which are all inspired by the natural movement of animals.

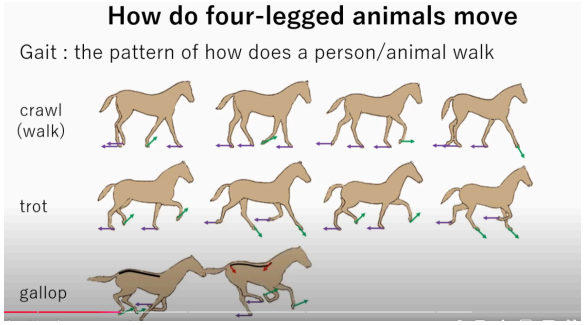

Figure 2: Three main quadrupedal gaits [7]

The crawl gait offers the most stability at the cost of speed. In the crawl gait the robot moves one leg at a time while keeping the other three legs grounded. This ensures that there are always three points of contact with the ground which reduces instability and helps maintain balance. This gait is very useful for navigating rough or uneven terrain, where maintaining stability is more important than speed.

The trot gait is faster and more efficient than the crawl. In the trot, diagonal pairs of legs move together, allowing the robot to maintain balance while increasing its speed. This gait creates a stable pattern where the robot alternates between two diagonal legs being on the ground at all times, which provides support and stability while allowing for faster forward motion. The trot is commonly used for traversing moderate terrain or when the robot needs to move quickly without losing too much of its stability.

The gallop gait is the fastest but least stable of the three gaits. During the gallop the front and rear legs swap states. So when the rear is in swing, the front is in contact. While this gait allows for the highest speed, it sacrifices stability and energy efficiency, making it best suited for situations where speed is crucial and the terrain is less complex.

By utilizing these different gaits, quadrupedal robots can adapt to a range of environments and tasks, this ensures they are both versatile and efficient when operating.

II. CRAWL GAIT

Crawling is the slowest gait, as it trades speed for stability. It gains this stability by moving one leg up and forward while the other 3 on the ground move backwards. It repeats this cycle for each leg. This keeps at least 3 points of contact on the floor at all times. This reduces any instability caused by the motion of the individual leg by keeping the center of gravity in between the 3 legs on the floor. Due to the low speed of this gait the inertial force from the leg is generally negligible.

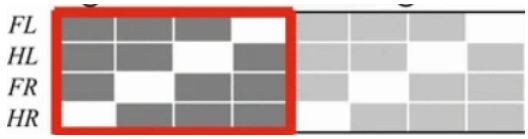

Figure 3: Crawl gait contact phases, gray is load bearing, white is swing phase [8]

III. TROT GAIT

The trotting gait is one of the most common and efficient motion patterns observed in robots and animals. In a trot, diagonal pairs of legs move in sync. The front left leg will move in tandem with the back right leg, while the front right will move with the back left. This coordinated set of movements creates a stable pattern that minimizes the oscillation of the center of mass. Which in turn improves energy efficiency and stability through various terrains.

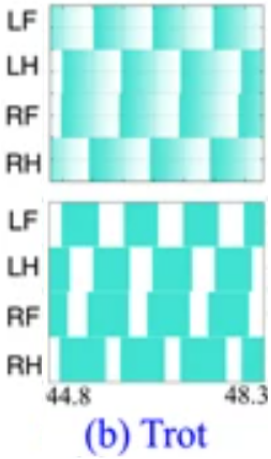

Figure 4: Trot gait contact phases, blue is load bearing, white is swing phase [9]

IV. CANTER GAIT

The canter gait is used to transition between trotting and galloping. As such its speed is in-between

trotting and galloping as well. It alternates through all the legs being on the ground so that 3 legs are typically in the air at a time followed by 3 legs being on the ground.

In figure 5, the top diagram shows the motion of the leg with color being forward and white being backwards. The diagram below shows at which points weight is placed on the legs. Color shows weight is acting on the leg. From that we see that one leg is typically on the ground with 3 being in the air.

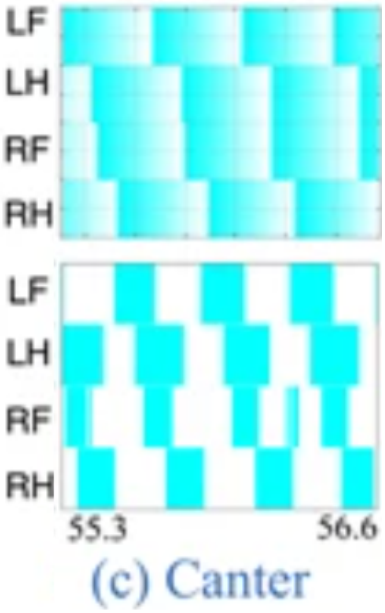

Figure 5: Trot gait contact phases, blue is load bearing, white is swing phase [9]

V. GALLOP GAIT

Galloping is the least efficient and least stable method of locomotion until the maximum speed. It considerably increases the stress and forces felt on the quadruped. Despite these flaws, it is the fastest as well. This speed and instability comes from the fact that the legs switch between front and rear legs jumping forward. It also deals with greater impacts due to essentially jumping which can cause slipping or sudden forces upon landing.

It works by first getting to a speed to carry momentum forward. Then it lifts the rear hind legs and brings them forward. While this is happening, the front legs are moving backwards on the ground continuing forward momentum. Then the back legs hit the ground and move backwards while the front legs are lifted and moved forward again.

When the back or front legs hit the ground while moving backwards the quadruped uses this time to jump, which lifts that side of the quadruped up. This is what leads to being able to move the legs forward to reset for the next impact. This is also what increases the strain on the quadruped.

In order to increase stability the quadruped slightly offsets the front and hind legs in order to keep the center of gravity centered. This prevents the quadruped from falling over sideways during landings. This is similar to how the trot moves diagonal legs for the same purpose.

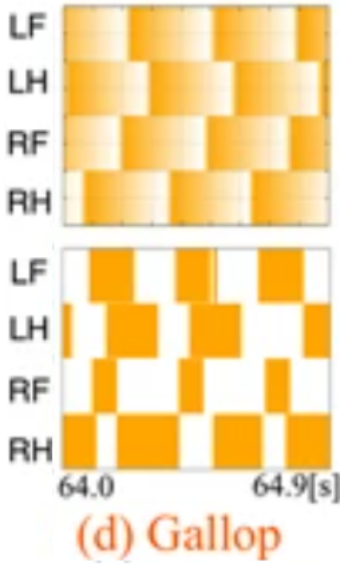

Figure 6: Gallop gait contact phases, orange is load bearing, white is swing phase [9]

The speed of galloping is linearly affected by changing the frequency of the cycle. This changes the length of the stride as well. This increased stride length reduces the ability to turn quickly due to only being able to change the direction when most of the feet are on the ground.

VI. KINEMATICS

The dimensions for the robot were obtained from a CAD model found online [3]. The size of the robot matched up with the size provided by the manufacturer, so the model seems dimensionally

reliable. The following dimensions were retrieved from the CAD model.

d1 = 60.1 mm

d2 = 81 mm

a2 = 213 mm

a3 = 212.272 mm

r = 22.921 mm (radius of spherical foot)

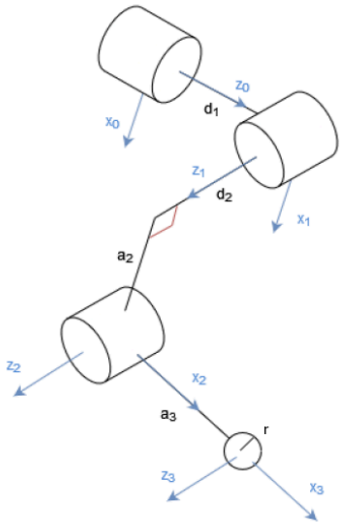

Figure 7: Leg diagram with assigned coordinate frames

The DH table for the leg is the same as for the RRR manipulator, but the shoulder offset introduces a non-zero value for d2.

Table 1: Robot Leg DH Table

Link	a_i	α_i	d_i	θ_i
1	0	90^o	d_1	θ_1^*
2	a_2	0	d_2	θ_2^*
3	a_3	0	0	θ_3^*

The forward kinematics for the above DH table are shown in the following homogeneous transformation matrix.

$$T_3^0 = \begin{bmatrix} c_1c_{23} & -c_1s_{23} & s_1 & d_2s_1 + a_2c_1c_2 + a_3c_1c_{23} \\ s_1c_{23} & -s_1s_{23} & -c_1 & -d_2c_1 + a_2s_1c_2 + a_3s_1c_{23} \\ s_{23} & c_{23} & 0 & d_1 + a_2s_2 + a_3s_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the inverse kinematics, each leg has 3 DOF's and is similar in nature to a RRR manipulator with a shoulder offset. The world coordinate system is defined such that z0 is up, x0 is pointed towards the head of the robot, and y0 is pointing normal to the right side of the robot. Since the leg has 3 DOF's, we define the target location as (xc, yc, zc) relative to the base of the body revolute joint. First, to find theta 1, the perspective becomes perpendicular to the axis of the shaft of the body revolute joint.

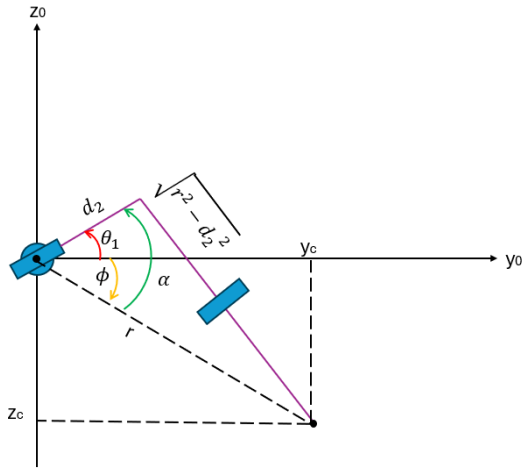

Figure 8: 2D view of the leg perpendicular to the body joint shaft

$$\phi = atan2(y_c, z_c)$$

$$r^2 = y_c^2 + z_c^2$$

$$\alpha = atan2(d_2, \sqrt{r^2 - d_2^2})$$

$$\theta_1 = \phi + \alpha$$

To find theta 2 and theta 3, the perspective becomes perpendicular to the plane containing the thigh and calf, essentially becoming a 2-link planar manipulator.

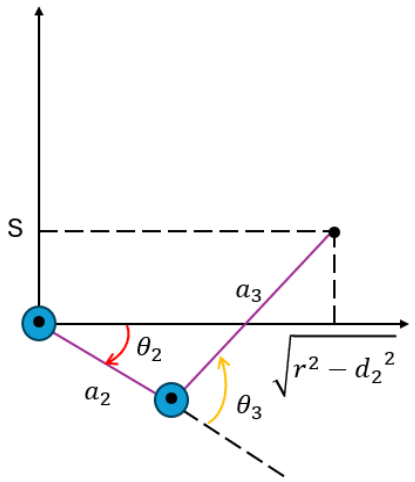

Figure 9: 2D view of the robot leg perpendicular to knee joint shaft

$$s = x_c - d_1$$

$$\theta_3 = acos\left(\frac{r^2 - d_2{}^2 + s^2 - a_2{}^2 - a_3{}^2}{2a_2a_3}\right)$$

$$\theta_2 = atan2(\sqrt{r^2 - d_2{}^2}, s) - atan2(a_2 + a_3cos(\theta_3), a_3sin(\theta_3))$$

As such, the three joint angles can be determined given the input coordinate location in 3D space. It is of importance to note that zc is located at the center of the spherical foot. To reach the ground or go from the ground to the center of the spherical foot, simply add or subtract respectively the spherical foot radius of 22.921 mm from the z direction.

The above equations are given with respect to the hip/body joint base. To move from the robot centroid to the zeroeth frame of each leg, one simply translates in the x and y direction, with the signs changing based upon translating to the front/rear and left/right side. The distances are given by $l_x = 128mm$ and $l_y = 46.75mm$ to move from the robot centroid to the front right base joint.

VII. FOOT TRAJECTORY

To facilitate the walking motion of the Go1 robot, several foot path curves were created based on the available literature [4] [5] [6]. All of the created profiles were designed using NoLimits 2 Roller Coaster Simulation software and are continuous Non-uniform Rational B-splines (or NURBS). The following figures show the original proposed foot paths and associated radius combs.

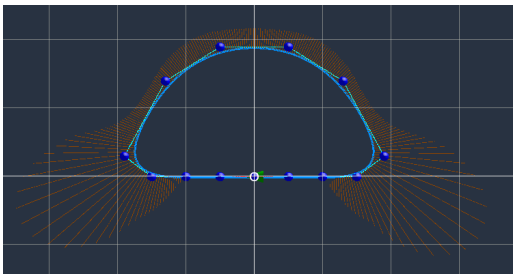

Figure 10: Symmetric Hump Path

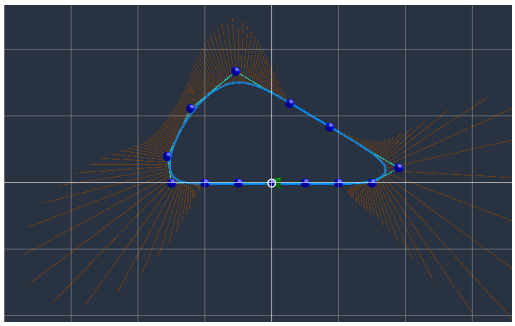

Figure 11: Forward Reach Path

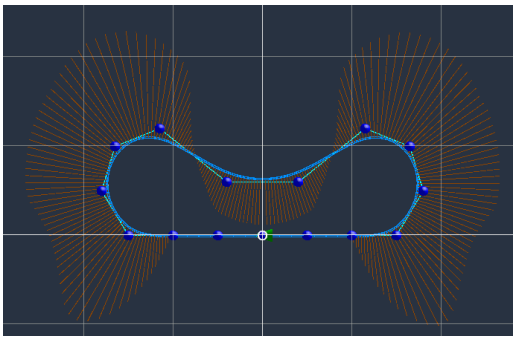

Figure 12: Center Dip Path

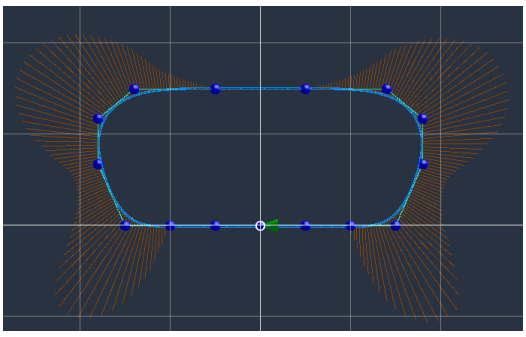

Figure 13: Oval Path

Out of the above proposed foot paths, the Symmetric Hump path was selected for its overall simplicity and its similarity to the cycloidal trajectory already utilized by the Go1. With this profile selected, the curve was time-parameterized by dividing the path into four 0.25s sections, creating an overall base cycle time of 1.0s that can be scaled if needed.

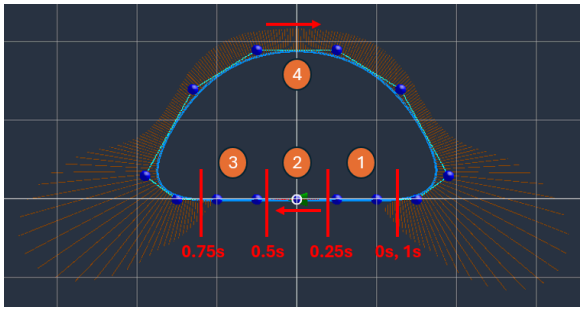

Figure 14: Symmetric Hump Time Divisions

With the time steps set up, the velocity for each direction was calculated using the central difference method with an order of accuracy of $O(\Delta x^2)$. The position and velocity trajectory graphs are presented below, with the position graph also accounting for the default leg height of the robot (vertical distance from robot centroid to foot).

$$f'(x) \simeq \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}$$

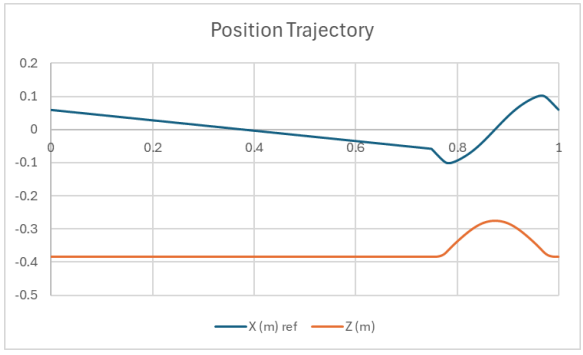

Figure 15: Position Trajectory curves for 1s cycle

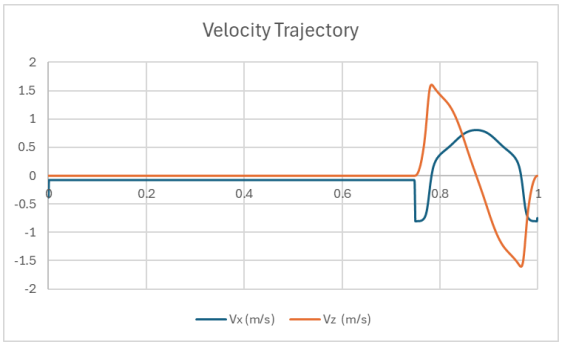

Figure 16: Velocity Trajectory curves for 1s cycle

In order to move the robot at different forward velocities, both the forward distance per cycle and cycle time are scaled proportionally based on the desired velocity. First the velocity scale s is calculated.

$$s = v_d / v_o \; (s > 0)$$

$$v_o = default\ contact\ speed = 0.07843\ m/s$$

$$x_s = distance\ scale = \sqrt{s}$$

$$t_s = time\ scale = 1/\sqrt{s}$$

The above equations are valid because $s = x_s / t_s$, thus the overall velocity scaling factor is always preserved. For the case where travel in the opposite direction is desired, simply use the absolute value of the desired velocity and reverse the sign of the trajectory x coordinates.

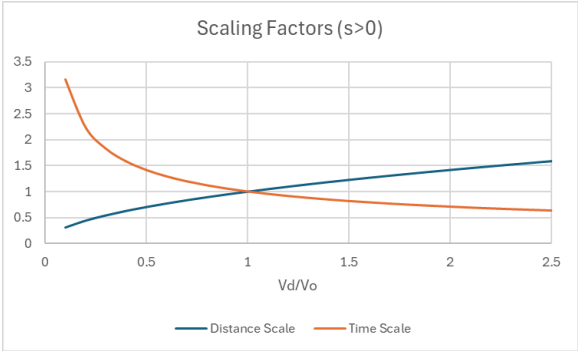

Figure 17: Velocity Scaling Factors

Using the above scaling rules, and with the default contact speed $v_o = 0.1569\ m/s$, the default cycle distance of $\Delta x_o = 204.52\ mm$, and the maximum cycle distance of $\Delta x_{max} = 361.28\ mm$ (based on

leg geometry), $x_{s,max} = 1.766$. It follows that $s_{max} = x^2_{s,max} \approx 3.12$. Therefore, the maximum desired linear velocity is $3.12v_o$, or approximately 0.5 m/s. Once this point is reached, the x scaling is saturated and only the time can be scaled further to achieve greater speeds according to the following relationship:

$$x_s = \sqrt{s_{max}}$$

$$t_s = \sqrt{s_{max}}/s$$

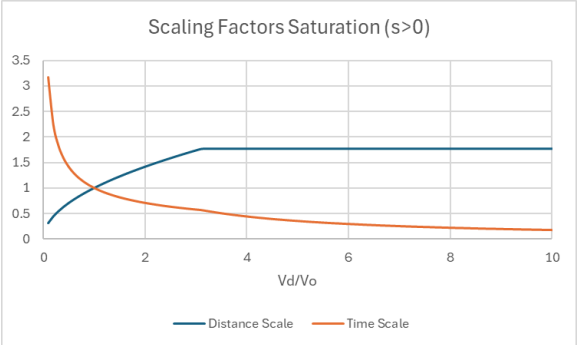

Figure 18: Velocity Scaling Factors with x_s saturation

In practice, a s_{max} should be chosen lower than the calculated 3.12, such as 2.5, in order to provide a safety factor against reaching any potential singularity points.

VIII. CODING

The Robot Operating System, or ROS for short, is a pseudo-operating system for your robot. It provides all the expected functions of an operating system as well as providing tools and libraries for working with code in various ways across multiple computers [10]. ROS is not intended to replace the standard operating system but to work in tandem to control the robot. It provides a means for computers to communicate with each other through multiple processes and is a very robust system having many of its packages fully debugged and stable for each major distribution [10]. The modular nature of a ROS system allows each subsystem to be split apart and tested individually, sometimes even replacing physical hardware with a simulator, such as Gazebo.

ROS is available on many different operating systems and can compile code in a few different languages. For this project, we use ROS Noetic, as this is a requirement of the Unitree GO1.

Unitree Robotics provides a few GitHub repositories for use with the Go1. The URDF file for the physical robot is part of the unitree_ros distribution. This one also includes the robot and joint controller and the descriptions necessary to launch the robot in Gazebo [11]. The unitree_ros_to_real packages can be used to control our Go1 in the real world, it allows for both low- and high-level control. Allowing us to individually command all joints and control the overall movement [12]. The unitree_legged_sdk can be used for basic communication between a PC and the onboard controller [13]. It also has capabilities to build either python or C++ code by modifying the way the package is called to build. The last of the useful SDKs that are provided is the unitree_guide. This is a simple example that creates and builds a controller from the main.cpp in unitree_guide/src as the node junior_ctrl. [14]. Unitree notes that the guide is "a basic quadruped robot controller for beginners [14]." It is not perfect and can be made by tweaking the controller.

Three packages are developed to accomplish the tasks of the project: ucf_go1_bringup, ucf_go1_control, and ucf_go1_util. The bringup package provides the launch and configuration files necessary to run the project. The launch files provide an easy way to start up the robot and run the necessary components. The configuration files contain tunable parameters. This package also provides modified xacro files that remove the simulation bits of the go1_description package and modify the transmissions used for ros_control. Each package in the repository provides additional readmes and code documentation for future reference. Nearly all of the project uses C++14 as its primary programming language. This is used due to the available libraries and performance provided by the language compared to Python for low latency applications.

The ucf_go1_control package provides controller software for the project. Ros_control [18] is used extensively to accomplish the goals of the project. Ros control provides abstractions and

implementations for control of joints and actuators. In this project, we use both the joint_trajectory_controller and the joint_state controller. The joint_state_controller simply broadcasts joint states read from hardware on the topic */joint_states*. The joint trajectory controller provides a multi-axis trajectory controller that ingests reference trajectories with timestamped positions and velocities and outputs a joint velocity for each joint. Section IX goes into greater detail on the control scheme. In order to use this controller with the Unitree hardware, we developed a robot hardware interface. This software wraps the provided Unitree UDP transport software to control joints based on the commands from the ros_control controllers. It also reads data from the UDP transport to states that are read by controllers such as the joint_state_controller.

This approach is chosen due to the ability to swap robot hardware without modifying any upstream controller interfaces. For example, the same controller scheme can theoretically be used in simulation as on the real robot. In practice, however, Gazebo simulation does not provide all of the same interfaces as the real robot, and the controller tuning parameters diverge due to imperfect modeling. Swapping between these configurations is as easy as using a launch argument to select simulation or real robot parameters.

The reference trajectory is acquired by loading a csv profile that is available to the ros package. The reference trajectory generation is discussed in section VII. A simple cycloidal trajectory is also provided in our main control. This was used as a placeholder while the final profiles were being developed. Additional profiles can be added simply by adding a csv file in the appropriate directory with the right format. A launch argument is provided to select the desired profile when running the robot.

During early development, the ucf_go1_util package was created to provide simple utilities for use throughout the project. Only a simple function generator that published sinusoidal trajectory commands was developed.

Through early testing, we concluded that the easiest way to connect to and run our code on the Go1 is a wireless connection through the onboard WiFi chip. Each robot has its own individual network, and connecting allows a computer to communicate with the Go1's Raspberry Pi, which acts as a connection to the main switch, and subsequently the main control unit. Once connected, running launch files from the command line produces output on the physical robot. All of the software we wrote runs on a remote laptop and only the final commands are sent over UDP to the robot controller. This proved to be somewhat unreliable. Occasionally, we would lose connection to the robot or receive messages out of order which would cause errors in the control scheme. We recommend deploying the hardware interface, at minimum, to the robot itself to be more robust to wireless communications on performance and safety critical commands in the future.

ROS 1 is utilized in this project. There are no further releases of ROS 1 and the operating systems that ROS 1 can be released on are at the end of life this year. This means that unaddressed software security issues may arise as time goes on. ROS 2 is recommended for future project work.

IX. CONTROL SYSTEM

We designed two main systems to control the robot. There is a main body controller that works within ROS, and the hardware interface that talks with UDP to send and receive messages from the robot. The main control loop is shown in Figure 19.

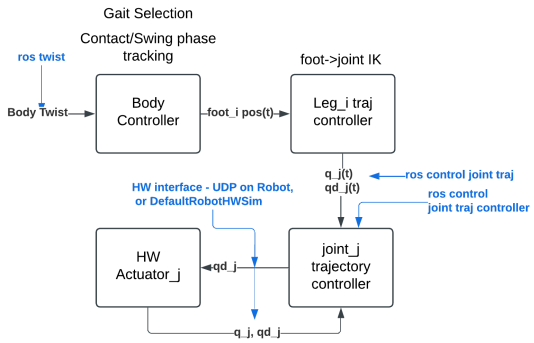

Figure 19: Robot Control Loop

The body controller is a custom node for ROS consisting of two submodules, the first of which is the main body controller. This module will read a desired body twist and gait type. The trajectories are generated via a controller using desired robot

velocity, scaling factors, and phase. It creates the reference foot trajectory by using the precalculated foot profile and determining where in the path the foot needs to go with the phase logic discussed in section X.

The outputs from the body controller are then used in the second submodule, the leg trajectory controller. This controller reads the current foot position and velocity from the hardware interface, and the desired foot position and velocity from the body controller. It breaks up each foot trajectory into individual joint trajectories, using the inverse kinematics handled by the Unitree ROS libraries. These joint trajectories will output a desired position, and velocity and send them to the joint trajectory controller.

The joint trajectory controller reads in the desired joint trajectories from the body controller and using a feed forward PID controller for each individual joint, calculates the velocities to send the commands to the hardware interface. As referenced before, the joint trajectory controller receives a reference trajectory of positions and velocities in joint space for each joint in a single message. The message has a reference trajectory for a configurable amount of lookahead steps. That is, the joint trajectory controller does not receive just one timestep, but rather 100 timesteps as configured. These trajectories are calculated in the body controller at a configurable rate that we configured to 20 Hz by default. This means that the joint state controller will receive updated trajectories while it is still completing its previously received trajectory. This command batching is done to provide the controller with sufficient information to perform a smooth cubic interpolation of the trajectory to ultimately generate a desirable output. If the previous trajectory is not complete (which in general it is not), then another level of interpolation is performed to smooth between the two separate trajectories [19]. The trajectory control utilizes a feed-forward velocity to improve tracking and perhaps most importantly, slow the foot down as it comes down to the ground.

The actuator reads these commands and writes them to the UDP to actually command the robot to move. The actuator also reads messages, usually sensor data, from the robot via the UDP and distributes the sensor readings to the various modules that require real time data.

The hardware interface works in conjunction with the Unitree UDP library to interpret the messages being sent from the robot and the body controller. The hardware controller was designed to be able to utilize the strong capabilities built into ROS control that the built in interface from Unitree bypasses. Our interface has two submodules, the trajectory controller, and the hardware actuator.

For each joint, there are 6 parameters that can be tuned. There are standard P, I, and D control parameters, feed forward gain, and Kp and Kd parameters. The Kp and Kd provided by the Go1 motor control commands essentially act as spring and damper constants respectively for the joint. This compliance on the low level joints is critical to avoid jerky motions while stepping. However, increased compliance also worsened tracking during the swing phase. For our implementation, we were unable to tune these parameters contextually based on the gait phase. We used the following parameter values:

- Kp - 60 for front joints, 90 for rear joints
- Kd - 2 for front joints, 4 for rear joints
- P - 20 for all joints
- I - 0 for all joints
- D - 0 for all joints
- FF - 1 for all joints

An additional robot-wide parameter is provided in Unitree's safe power protect functionality. This function interprets joint commands and states to determine if the robot can safely execute the command without causing power problems. Initially we utilized a power protect value of 1, representing a maximum of 10% of the robots power being used. By the final gait after confidence was built, we allowed up to 70% of the robot's power to be utilized. This is critical to the success of the gallop gait in particular. Without increasing these values, the robot would simply stall itself out attempting the commands we needed to send to maintain a velocity required for the gait.

X. TRANSITION PHASES

Our implementation handles gait transitions through a state machine driven approach that manages both inter-gait and intra-gait phase transitions. Each gait cycle is divided into contact and swing phases, with

smooth transitions managed through our trajectory generation system. For the trotting gait, diagonal pairs of legs maintain synchronized phase relationships while the walking gait sequences through individual leg movements while maintaining tripod stability.

The phase transitions are governed by timing. We implemented a phase scaling system that adjusts based on the desired velocity, see section VII Foot Trajectory for further details. This scaling approach ensures smooth transitions between different movement speeds while maintaining gait stability. The contact phase velocity remains constant during ground contact, which helps maintain consistent force application and stability during the stance phase.

For each gait, we developed specific phase transition triggers:

1. Walk Gait: Phase transitions occur with a 25% offset between each leg, ensuring three legs maintain ground contact at all times. The transition sequence follows a diagonal-forward pattern to maintain stability.
2. Trot Gait: Diagonal leg pairs transition simultaneously, with a 50% phase offset between pairs. This creates the characteristic trotting motion while maintaining balance through alternating diagonal support.
3. Canter Gait: Implements a three-beat rhythm with transitions occurring at 30% and 70% of the gait cycle, serving as an intermediary between trot and gallop gaits.
4. Gallop Gait: Some feature brief suspension phases where all legs transition simultaneously, requiring careful timing of phase transitions to maintain forward momentum and stability. Our implementation however did not include the total swing phase, instead switching between front and rear legs with a slight offset. This follows the gaits we see in [15] and [16]. This is due to instability associated with zero floor contact.

Each phase transition is controlled through our state machine implementation, which monitors foot timing to determine appropriate transition points. This adaptive approach allows for more robust gait execution across varying conditions.

XI. FORCE FEEDBACK

As part of a method to reduce stomping and increase stability we investigated using the foot force sensors built into the Go1. This has been seen in use across many other quadrupedal robots. Namely the MIT cheetah [17] and in Unitree's own gait implementation. This is typically used to command state transition from swing phase to contact phase, eliminating the stomp by halting Z motion upon contact detection.

Our implementation changed the current phase when contact was detected. For trotting when two diagonal feet contacted, the phase was set to when the next pair of diagonal feet initiate the swing phase. Unfortunately, due to our trajectory controller following one single trajectory this ended up making our system worse in the simulation. The reason being that the sudden discontinuity would cause the position feedback to move even faster toward the current desired phase position. This increased the stomp and instability.

Although our implementation did not show good results, we still believe foot force feedback is important for future improvement of gait stability. One potential fix could be splitting the trajectory into independent swing state and contact phase trajectories. A way that could work without major refactoring is somehow setting the current position to read that it is at the desired location immediately after contact. Another could be to scale the speed of the gait slower for a brief interval to reduce stomping. Final method, which can't be simulated, would be to use the contact detection to reduce damping and spring constants upon contact.

One last note is that documentation on foot force sensors is limited. In our repo in the branch force-feedback shows the topic to subscribe to for feet force sensors with gazebo.

XII. RESULTS

Our implementation achieved several key objectives in both simulation and physical testing:

1. Gait Implementation:

- Successfully implemented all gaits in simulation
- Achieved stable trotting motion with diagonal leg synchronization
- Demonstrated phase transition stability between different gaits
- Validated scaling methods for varying movement speeds

2. Control System Performance:
   - ROS control integration successfully managed joint trajectories
   - State machine effectively handled gait transitions
   - UDP communication provided somewhat reliable robot control
3. Physical Testing:
   - Successfully assembled and validated test frame for safe robot testing
   - Implemented safety harness system for initial gait testing
   - Verified wireless control functionality
   - Confirmed low-level control mode operation
4. Velocity Control:
   - Implemented and validated velocity scaling system
   - Achieved smooth transitions between different speeds
   - Maintained stability during velocity changes
   - Successfully integrated feed-forward velocity control

Specific performance metrics include:

- Base cycle time: 1.0s nominal, adjustable through scaling
- Contact phase velocity: 0.1569 m/s during stance phase
- Forward distance per cycle: ~150 mm nominal
- Vertical clearance: 100mm during swing phase

Challenges and Limitations:

1. Initial stability issues during ground contact, particularly with faster gaits
2. Some trajectory tracking deviations during high-speed transitions
3. Gazebo simulation instabilities requiring careful initialization
4. Phase synchronization challenges during gait transitions

The results demonstrate the viability of our control architecture, particularly for walking and trotting gaits. While full gallop implementation remains challenging, the foundation for all four gaits has been established through our control system. The test frame and safety harness system proved essential for safe development and testing of the gaits, allowing for incremental validation of each component.

One significant finding during our testing is the controller button sequence to reliably connect and control the robot. The proper button sequence to press once the robot has booted up is:

1. L2+A to lock the joints
2. L2+A to enter prone state
3. L2+B to enter damping mode
4. L1+L2+Start to enter non-damping mode

We originally did not know about step 4, and this would cause audible oscillatory noise in the motors, severe instability, and random shutdowns. Once we started using the proper startup button procedure, the robot was much more reliable and had fewer errors.

XIII. DELIVERABLES:

Video documentation of our simulation robot results is found here:

https://drive.google.com/drive/folders/1UGBqQyxoG8vi57br3RaBHxZ80xq9eXSE?usp=drive_link

Video documentation of our real robot results is found here:
https://drive.google.com/file/d/1CHIlxI9Bcnb6RX3CXhHz8RmDmnhOAcRR/view?usp=drive_link

All code from our custom controller and hardware interface can be found here:
https://github.com/JohnPerg/5669-Project-2

Link for trajectories CSV:
https://docs.google.com/spreadsheets/d/1L4tp1TW9-Rawej7ie0i9doYsdvRQfib6/edit?usp=sharing&ouid=108278850878848175394&rtpof=true&sd=true

Link for sampled plots and ros bagged joint data for joint trajectory execution: https://drive.google.com/drive/folders/1g9p9fJzsxgBMBtrqn2PVUHVWmOE8AxRX?usp=sharing

XIV. CONCLUSION:

Through our implementation of velocity control, we were able to successfully generate each of the desired gaits and have the Go1 navigate forward while remaining upright, as desired. Through extensive testing, we were able to tune our code parameters for each gait and each joint individually. Velocity based joint control with position and velocity reference trajectory tracking is shown to be an excellent control scheme for quadrupedal motion, particularly in combination with impedance control on the low level joints. Given more time, we would look deeper into using the foot force sensors in order to achieve zero ground contact during the gallop gait and using balance control and compensation to achieve steadier gaits. We would deploy parts of our software onto the robot instead of on the laptop to address communication issues which would certainly be required to implement force feedback effectively. We would also incorporate additional profiles and phase timing to gallop more effectively. Additional body feedback could be incorporated as an additional layer in the control scheme with a balance controller. Finally, turning, sidestepping, and backstepping can be incorporated for a more complete solution.

XV. CONTRIBUTIONS:

Jarett Artman - gait research, coding documentation, repository identification, research paper development, ROS integration research

Eric Frankle - kinematics calculations, foot trajectory generation, velocity scaling factor, test rig assembly, parameter tuning, report writing.

Brandon Gross - project lead, controller design and development, simulation integration, hardware interface development, software architecture, launch, and configuration, coding documentation.

Steven Keller - gait research, phase state modulator development, foot force feedback research /development, and aided lab testing

John Piergallini - ros discovery, unitree repository discovery, robot interface/connection, time scaling implementation, testing and debugging, control documentation.

Andrea Styles - kinematics calculations, robot interface/connection, testing and debugging, result documentation.

XVI. CITATIONS:

[1] “Unitree Go1.” UnitreeRobotics, shop.unitree.com/products/unitreeyushutechnologydog-artificial-intelligence-companion-bionic-companion-intelligent-robot-go1-quadruped-robot-dog.

[2] “Unitree Robotics Go1 EDU Plus.” Wevolver.com, 2023, https://www.wevolver.com/specs/unitree-robotics-go1-edu-plus.

[3] “Unitree Go1.” INDRO ROBOTICS, 2024, indrorobotics.notion.site/Unitree-Go1-100ed1191b604d86b679398471d9e730

[4] Zeng, X., Zhang, S., Zhang, H., Li, X., Zhou, H., & Fu, Y. (2019). Leg Trajectory Planning for Quadruped Robots with High-Speed Trot Gait. Applied Sciences, 9(7), 1508. https://doi.org/10.3390/app9071508

[5] Wang, Mingfeng & Ceccarelli, Marco & Carbone, Giuseppe. (2016). A feasibility study on the design and walking operation of a biped locomotor via dynamic simulation. Frontiers of Mechanical Engineering. 11. https://doi.org/10.1007/s11465-016-0391-0

[6] Huang, S., Zhang, X. (2021). Biologically Inspired Planning and Optimization of Foot Trajectory of a Quadruped Robot. In: Liu, XJ., Nie, Z., Yu, J., Xie, F., Song, R. (eds) Intelligent Robotics and Applications. ICIRA 2021. Lecture Notes in Computer Science(), vol 13016. Springer, Cham. https://doi.org/10.1007/978-3-030-89092-6_18

[7] Mania, R. Quadruped Robot Basics. YouTube, n.d., www.youtube.com/watch?v=O_2swSMecB4.

[8] Kim, D., et al. "Optimization-Based Control for a Quadruped Robot with Nonlinear Constraints." Journal of Intelligent & Robotic Systems, 2023, link.springer.com/article/10.1007/s11370-015-0173-2
.

[9] Hess, M., et al. “Exploring Gait Transitions in a Quadruped Robot.” Scientific Reports, vol. 7, 2017, www.nature.com/articles/s41598-017-00348-9.

[10] J. M. O'Kane, A Gentle Introduction to ROS, 2013, https://jokane.net/agitr/

[11] Unitree Robotics, "GitHub - unitreerobotics/unitree_ros," [Online]. Available: https://github.com/unitreerobotics/unitree_ros?tab=readme-ov-file.

[12] Unitree Robotics, "GitHub - unitreerobotics/unitree_ros_to_real," [Online]. Available: https://github.com/unitreerobotics/unitree_ros_to_real.

[13] Unitree Robotics, "GitHub - unitreerobotics/unitree_legged_sdk," [Online]. Available: https://github.com/unitreerobotics/unitree_legged_sdk

[14] UniTree, "Unitree Guide," [Online]. Available: https://github.com/unitreerobotics/unitree_guide.

[15] Danilov, Vladimir & Diane, Sekou. (2023). CPG-Based Gait Generator for a Quadruped Robot with Sidewalk and Turning Operations. https://doi.org/10.1007/978-3-031-15226-9_27 .

[16] Choi J. (2021). Multi-Phase Joint-Angle Trajectory Generation Inspired by Dog Motion for Control of Quadruped Robot. *Sensors (Basel, Switzerland)*, *21*(19), 6366. https://doi.org/10.3390/s21196366

[17] Lee, J. (2013). Hierarchical controller for highly dynamic locomotion utilizing pattern modulation and impedance control : implementation on the MIT Cheetah robot . http://hdl.handle.net/1721.1/85490

[18] Chitta et al, (2017). ros_control: A generic and simple control framework for ROS, Journal of Open Source Software, 2(20), 456, doi:10.21105/joss.00456 https://joss.theoj.org/papers/10.21105/joss.00456

[19] Rodriguez, A. (2013). ros wiki: Joint trajectory controller, Understanding Trajectory Replacement. https://wiki.ros.org/joint_trajectory_controller/UnderstandingTrajectoryReplacement