Software and Control Design for the MIT Cheetah Quadruped Robots

by

Jared Di Carlo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 27, 2020

Certified by . . . .
Sangbae Kim
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Master of Engineering Thesis Committee

Software and Control Design for the MIT Cheetah Quadruped Robots

by

Jared Di Carlo

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis documents the development and implementation of software and controllers for the MIT Mini Cheetah and MIT Cheetah 3 robots. The open source software I developed is designed to provide a framework for other research groups to use the Mini Cheetah platform and is currently being used by seven other groups from around the world. The controllers I developed for this thesis are provided as example code for these groups, and can be used to make the robot walk, run, and do a backflip. The locomotion controller utilizes a simplified model and convex optimization to stabilize complex gaits online, allowing it to control complex, fully 3D gaits with flight periods, such as galloping. The backflip is accomplished through offline trajectory optimization with an accurate dynamic model and was the first backflip done on a quadruped robot.

Thesis Supervisor: Sangbae Kim
Title: Associate Professor of Mechanical Engineering

Acknowledgments

I would like to thank everybody who has helped me to complete this research. In particular, Sangbae Kim, my advisor, for supporting all of my work, and teaching me about legged robots. I have learned so much from discussions with him.

Ben Katz, for his amazing Mini Cheetah robot, for his many hours of helping me develop and debug the controllers, software, and electronics, and also for fixing Cheetah 3, each of the many times I broke it.

Patrick Wensing, for teaching me controls and dynamics, and for all of his advice and suggestions when I was first implementing these controllers.

Donghyun Kim, for all the work he has done to make giving out the Mini Cheetahs and their software go so smoothly, and for his contributions to the simulation and control framework software.

Alex Hattori, for his help in building the Mini Cheetahs, running experiments, and helping debug.

Everybody in the Biomimetics Lab. Working in this lab has been incredibly inspiring, and I will miss our weekly lab meeting discussions.

Contents

1 Background and Motivation **15**

1.1 Mini Cheetah Hardware 15

1.2 Cheetah 3 Hardware 17

1.3 Mini Cheetah Software Package for Research 18

1.3.1 Mini Cheetah Software for Demos 19

2 Modeling Quadruped Robots **21**

2.1 Simulation Model 21

2.1.1 Dynamics Model 21

2.1.2 Dynamics Simulation 25

2.1.3 Dynamics Simulation Tests 25

2.2 Contact Model 26

2.3 Actuator Model 27

2.4 Low-Level Controller Model 28

2.4.1 Mini Cheetah 28

2.4.2 Cheetah 3 30

3 Software **31**

3.1 Organization and Design 31

3.2 Third-Party Libraries 34

3.3 Linux on the Robot 35

3.4 Software Organization 35

3.5 Robot Framework 36

3.5.1 Leg Controller . . . 36
3.5.2 IMU . . . 37
3.5.3 Remote Controller . . . 38
3.5.4 Dynamics Model . . . 38
3.6 Simulator and User Interface . . . 38
3.6.1 Control Parameters . . . 40
3.6.2 Terrain Description File . . . 41
3.6.3 Visualizer . . . 42
3.6.4 Connection to Hardware . . . 44
3.7 Error Handling . . . 45
3.7.1 Robot Control Code Exceptions and Crashes . . . 46
3.7.2 Simulator Internal Error . . . 47
3.7.3 Timeout . . . 47
3.8 Communication between Simulator and Robot Framework . . . 48
3.8.1 Performance of Communication . . . 49
3.9 Communication between UI and Robot Hardware . . . 51

4 Cheetah Locomotion Controller 53

4.1 Gait Definition . . . 54
4.2 Prior Work . . . 56
4.3 Coordinate Systems and Reference Trajectories . . . 56
4.3.1 Global Coordinate System . . . 57
4.3.2 Body Coordinate System . . . 57
4.3.3 Yaw Coordinate System . . . 57
4.3.4 Hip Coordinate System . . . 58
4.3.5 Reference Trajectory . . . 58
4.4 Predictive Control . . . 59
4.4.1 Separate Swing Leg Control . . . 60
4.4.2 Single Rigid Body “Space Potato” Model . . . 61
4.4.3 Convex Optimization . . . 62

4.5 MPC Formulation . . . 68
4.5.1 QP Formulation . . . 70
4.6 Results on Mini Cheetah . . . 72
4.7 Results on Cheetah 3 . . . 73

5 Backflip **79**
5.1 Related Work . . . 79
5.2 Approach . . . 79
5.3 Model . . . 80
5.4 Optimization . . . 82
5.5 Results . . . 82
5.6 Extensions . . . 83

A Dynamics with Rotors **87**
A.1 System Description . . . 87
A.2 Algorithm Derivation . . . 89
A.2.1 Kinematics and Contact Points . . . 89
A.2.2 Mass Matrix . . . 89

B Mini Cheetah Software Package **97**

List of Figures

1-1 The MIT Mini Cheetah Robot (prototype version) 16
1-2 An MIT Mini Cheetah Robot (final version) being powered on for the first time. 17
1-3 Nine MIT Mini Cheetah Robots (final versions) being tested outside. 17
1-4 The MIT Cheetah 3 Robot (old version) 18
1-5 The Mini Cheetah does a backflip on "The Tonight Show Starring Jimmy Fallon" 19

2-1 Kinematic tree of the Cheetah model 23
2-2 Simulation time-step computation time histogram for forward dynamics calculation 26
2-3 Simulation contact points for Mini Cheetah. There are 8 points in the corners of the body's bounding box, a point on each knee, and a point on each foot. 27
2-4 Simulation time-step computation time histogram for forward dynamics and contact solve 28
2-5 Mini Cheetah simulated torque speed curve 29

3-1 64 Cheetah 3 Robots are simulated at 1.5x real time simultaneously on a 18-core CPU 34
3-2 Simulator Control Panel 41
3-3 Terrain of stairs, plane, and box 43
3-4 Visualization of 25 Cheetah 3 robots 44
3-5 Debugging visualization of swing leg controller. 44

3-6 Debugging interface for Mini Cheetah 45
3-7 Actuator test setup for a single Mini Cheetah 45
3-8 Error message when control code throws an exception 47
3-9 Synchronization performance 50
3-10 Synchronization performance 51

4-1 The footstep pattern for a galloping gait without flight 55
4-2 Simulated Mini Cheetah with a reference trajectory generated for a forward velocity command. The red dots are samples on the reference trajectory 59
4-3 Data from state estimator during high speed trotting on Mini Cheetah 72
4-4 Data from state estimator during high speed in place turning on Mini Cheetah 73
4-5 Several frames of the MIT Cheetah 3 Robot galloping at 2.5 m/s . . . 74
4-6 MPC result and orientation 75
4-7 Leg data from galloping 76
4-8 Velocity control of galloping to 3 m/s 76
4-9 Cheetah 3 climbing stairs while trotting 77

5-1 Frames from optimized trajectory 82
5-2 Joint torque and position tracking error during a backflip 83
5-3 Mini Cheetah doing a backflip outside 84
5-4 Cheetah 3 Simulated Backflip 84
5-5 Cheetah 3 Jump 85

List of Tables

3.1 Low level leg control capabilities of the MIT Cheetah 3 and MIT Mini Cheetah robots 37

Chapter 1

Background and Motivation

I have developed a simulator, control framework, and several new controllers for two different MIT Cheetah robots, the Mini Cheetah and Cheetah 3. The three goals of my work are to take advantage of the unique and highly dynamic Cheetah hardware with high-speed legged locomotion and acrobatic behaviors, provide a software package that others can use to do research with Mini Cheetah, and to create a robust "demo mode" for Mini Cheetah which is reliable enough to perform live demos when run by people who are untrained with the robot.

1.1 Mini Cheetah Hardware

The Mini Cheetah [11, 10] is a four legged robot developed by Benjamin Katz from the MIT Biomimetic Robotics Lab. Compared to other quadruped robots, Mini Cheetah is low cost, highly reliable, easy to use, and very powerful. Mini Cheetah is a torque-controlled 12 degree-of-freedom (DoF) quadruped robot. The prototype robot is shown in Figure 1-1, and final version is shown in Figure 1-3. Mini Cheetah has an onboard IMU, computer, and battery, which allow it to run completely untethered and without external sensing. The robot is well suited to dynamic behaviors because of its low cost, reliability, ease-of-use, and power.

Because of Mini Cheetah's low cost and easy repairability, risky and unproven controllers can be tested on hardware without fear of damaging expensive and hard-

Figure 1-1: The MIT Mini Cheetah Robot (prototype version)

to-replace hardware. Taking these extra risks decreases the amount of time required to deploy new controllers on hardware.

Mini Cheetah is also very reliable. It has survived falling over on any material, failing to land a backflip, failing to land a jump, and falling off the edge of the treadmill. If the robot detects it has fallen over, it will self right, stand up, and reset all controllers so the experiment can be continued without having to touch the robot or restart software.

These characteristics allow us to be ambitious in our control design. As an example, when the first Mini Cheetah was built, the first behavior we attempted was a backflip, which was successful on the first attempt. Because the hardware is so capable, the software quickly becomes the bottleneck for developing and tuning new controllers. It is important that the software allows the controllers and parameters to be tweaked and tested on the robot or simulation without having to wait a long time for the software to rebuild, or the simulator to run. I developed an interactive simulator which allows the user to tune parameters and control the simulated robot in real time, without needing to restart code or controllers between different attempts.

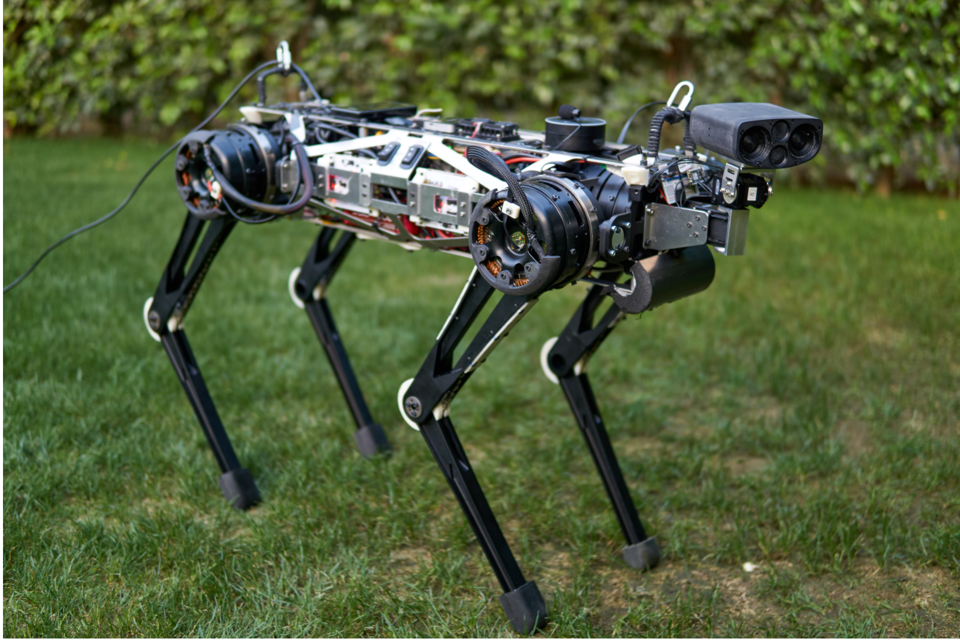
Figure 1-2: An MIT Mini Cheetah Robot (final version) being powered on for the first time.

Figure 1-3: Nine MIT Mini Cheetah Robots (final versions) being tested outside.

1.2 Cheetah 3 Hardware

The software can also run on MIT Cheetah 3 [4], a 45 kg quadruped robot shown in Figure 1-4. Like Mini Cheetah, all control, power, and sensing are onboard the robot, so it can operate untethered. At the time of writing, Cheetah 3 is being upgraded to a new computer, electronics, and body, so I can only present simulation and older experimental results. Cheetah 3 is a significantly larger quadruped, so it can handle larger obstacles and disturbances compared to Mini Cheetah. Due to its increased size, it is much more dangerous and fragile than Mini Cheetah, and is more time consuming to prepare and perform experiments. I expect controllers to first be developed an verified on Mini Cheetah, which is easier to use, then transferred to Cheetah 3, which is larger and as a result more capable.

Figure 1-4: The MIT Cheetah 3 Robot (old version)



1.3 Mini Cheetah Software Package for Research

The open source Mini-Cheetah software package contains a simulator, control framework, and set of example controllers for other research groups to use.

Currently, there are 11 Mini-Cheetah robots which have been built in the MIT Biomimetics Lab, and there will be more Mini Cheetahs built in the near future. These robots will be distributed to research groups around the world, who will use the open source control framework and simulator. Additionally, all of the controllers developed at MIT will be provided as an example. It is important that the software is easy-to-use and well documented, as some research groups may have little experience with programming in C++. The software is open source and available on GitHub.

A second goal of open sourcing the software is to allow others to replicate our results. Several robot enthusiasts have carefully analyzed pictures from [10] and have created their own robots which are similar to Mini Cheetah. The YouTube user "hongtao Zhang" has one of these robots and used our open-source control software on it. He demonstrates our locomotion controller in his videos "Kick the dog when it walks." [26] (he kicks a robot, not an actual dog) and "Quadruped robot new video"

[27]. Several other groups in China also have similar robots and are in the process of getting our software up and running.

1.3.1 Mini Cheetah Software for Demos

Although Mini Cheetah was designed mainly for legged-locomotion research, it is often used for demonstrations.

Mini Cheetah has given many demos, including “The Tonight Show Starring Jimmy Fallon”, shown in Figure 1-5, the “MIT Stephen A. Schwarzman College of Computing Celebration”, the “DEVIEW 2019” conference in South Korea, and the “AI for Robotics" workshop in France. At some of these demos, none of the lab members could be present, so the robot should be easy to set up and control for people who have not used it before. For the purpose of doing demos, the robot must be easy to start, easy to operate, reliable, and able to quickly recover if there is a fall. The software does not require an external computer to start, and the robot can be initialized and controlled entirely from a handheld R/C radio made for model aircraft.

Figure 1-5: The Mini Cheetah does a backflip on “The Tonight Show Starring Jimmy Fallon”

Chapter 2

Modeling Quadruped Robots

It is important to have an accurate dynamics model of the robot, for both control and simulation. The two Cheetah robots have an identical structure, so their models only differ in the mass properties, lengths, locations, and gear ratios. For rigid body models and dynamics, I use the spatial vector algebra conventions and algorithms described in [5].

2.1 Simulation Model

The simulation model is designed to be as accurate as possible, while still allowing simulation to happen at a reasonable speed. All of the simplified models discussed later in this thesis derive their parameters from the simulation model. The simulation model has three major components: dynamics, contact, and actuators.

2.1.1 Dynamics Model

Cheetah is modeled with a rigid body dynamics model. Previous work has shown that the dynamics of actuators and gearboxes are significant for legged locomotion [25], so I added rigid bodies to the model which correspond to the rotor inside of the actuator. Both Mini Cheetah and Cheetah 3 use actuator modules which combine an electric motor and planetary gearbox. Mini Cheetah's actuators have a 6:1 gear

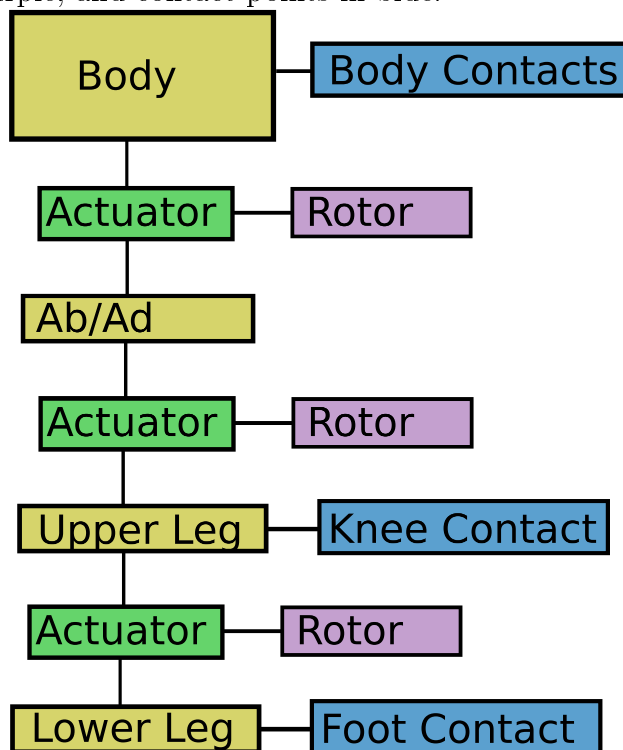
reduction, so the rotor will spin 6 times faster than the output of the actuator rotor. The knee has an additional belt reduction for a total reduction of 9.33. The inertia of these rotors contributes significantly to the dynamics of the legs because the inertia of the rotor felt at the output is multiplied by the square of the gear ratio. Modeling the actuators as a pure torque source would result in a model which can achieve higher joint accelerations than the actual robot, which would be an issue for highly dynamic behavior.

The complete model consists of 25 rigid bodies, with 12 actuated degrees of freedom, 6 unactuated degrees of freedom, and 12 constraints. It has a 36-dimensional state, represented with 37 state variables. The model includes the base, all the links, and the rotor inside of the actuator.

The 25 rigid bodies consist of the robot's main body, 4 abduction/adduction (ab/ad) links, 4 hip links, 4 knee links, and 12 acuator rotors. The 12 actuated degrees of freedom are the 12 links, which are rotary joints. The 6 unactuated degrees of freedom are the position and orientation of the base. The 12 constraints are gearing constraints which each satisfy $N\dot{\mathbf{q}}_{\text{link}} = \dot{\mathbf{q}}_{\text{rotor}}$ where N is the gear ratio, to couple the rotation of the actuator's rotor to the rotation of the appropriate link. All the link joints are rotary joints.

The 37 state variables consist of 12 link positions, written as $\mathbf{q} \in \mathbb{R}^{12}$; 12 link velocities, written as $\dot{\mathbf{q}} \in \mathbb{R}^{12}$; the Cartesian position of the base, written as $\mathbf{p} \in \mathbb{R}^3$; the Cartesian velocity of the base, written as $\mathbf{v} \in \mathbb{R}^3$; the orientation of the base, represented as the quaternion $\mathbf{q}_{\text{base}}$; and angular velocity of the base in the base's coordinate system, written as $\omega \in \mathbb{R}^3$. The kinematic tree for the body and a single leg is shown in Figure 2-1.

Figure 2-1: Kinematic tree of the Cheetah model with a single leg. All four legs have the same configuration. Bodies are shown in yellow, actuated degrees of freedom in green, rotors in purple, and contact points in blue.



Including the Rotor

Including the rotor adds an addition 12 rigid bodies and 12 constraints to the dynamics model. There are two general approaches to including their effect.

The first is to model them as separate rigid bodies, add gearing constraints, and use an algorithm for solving constrained rigid body dynamics. This approach will drastically increase the computational requirements of simulation, and may suffer from accuracy and convergence issues with constraints unless it is run with a very small time step. In general, constrained dynamics algorithms have $O(n_{\text{bodies}}^2)$ complexity [5]. Adding rotors will increase the rigid body count from 13 to 25, which will slow down the dynamics calculation significantly. This approach involves adding state variables for each rotor, increasing the size of the mass matrix and Jacobians. The state is no longer expressed in minimal coordinates, so all controllers and estimators using this formulation will need to be aware of the state constraints. Working in non-

minimal coordinates will complicate all algorithms which use dynamic information and constrained dynamics sometimes have convergence and accuracy issues, so I did not pick this approach.

A second approach is to use rigid body dynamics algorithms which can explicitly solve constrained dynamics in minimal coordinates. This means that we do not add additional variables to the state vector for the position of the rotors. This is possible because if we know the position of the joint, we can use the gearing constraint to compute the position of the rotor. The dynamics algorithms must then be modified to include the effect of the rotors.

While there is no general algorithm for explicitly computing constrained dynamics, the actuator rotor constraints are simple enough that deriving a closed form solution to the constrained dynamics which can add the effects of the rotors, without increasing the number of rigid bodies in the model, is possible. Even though we are simulating a system with 25 rigid bodies, the algorithm effectively works on a system with 13 rigid bodies, and additionally adds the effects of rotors, using $O(n_{\text{rotors}})$ additional time from running the unconstrained algorithm in the 13 rigid body system. This approach can be extended to all the dynamics algorithms used, meaning we can compute the minimal coordinate mass matrix and Jacobians. From the perspective of the controller, the effect of the rotor is lumped into the link's degree of freedom, and it does not have to consider the extra rotor bodies or gearing constraints. This massively simplifies control design. Patrick Wensing, a previous member in the lab, implemented this algorithm in MATLAB for forward dynamics. I reimplemented this in C++, and developed algorithms to compute the mass matrix, Coriolis torque, and gravity torques in minimal coordinates, including rotors.

With this approach, our dynamics can be written in exactly the same form as for the system without rotors:

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) = \sum_k \mathbf{J}_k^\top(\mathbf{q})\mathbf{f}_k + \mathbf{S}\mathbf{u} \tag{2.1}$$

where $\mathbf{H}$ is the mass matrix, $\mathbf{q}$ is a vector of generalized positions, $\mathbf{C}$ is a *vector* of Coriolis and gravity generalized forces, $\mathbf{J}_k$ is the spatial Jacobian of the body containing k-th foot, expressed at the foot and in the world coordinate system, $\mathbf{f}_k$ is the spatial force on the body containing k-th foot, expressed at the foot and in the world coordinate system, $\mathbf{S}$ selects actuated coordinates, and $\mathbf{u}$ is the control input. The dynamics software can efficiently compute $\mathbf{H}, \mathbf{C}, \mathbf{J}_k$ with the effect of rotors.

This algorithm is derived in [9]. While the derivation is clear and mathematically rigorous, it does not give an intuitive physical explanation of how the addition of rotors affects the system dynamics, nor how to determine how much of an effect is caused by the rotors. In Appendix A, I have provided an alternative derivation which approaches the problem by first computing the dynamics without rotors, then computes the additional effect of the rotors. The advantage to this approach is that it allows the addition of the rotors to be divided into five separate effects. For many systems, many of those effects are negligible and can be omitted to simplify dynamics.

2.1.2 Dynamics Simulation

For simulation, I implemented Featherstone's algorithm [5], which efficiently solves equation (2.1) for $\ddot{\mathbf{q}}$ when given $\mathbf{f}_k, \mathbf{u}, \dot{\mathbf{q}}$, and $\mathbf{q}$. To include the effect of rotors in Featherstone's algorithm, I used the modification discussed in [9].

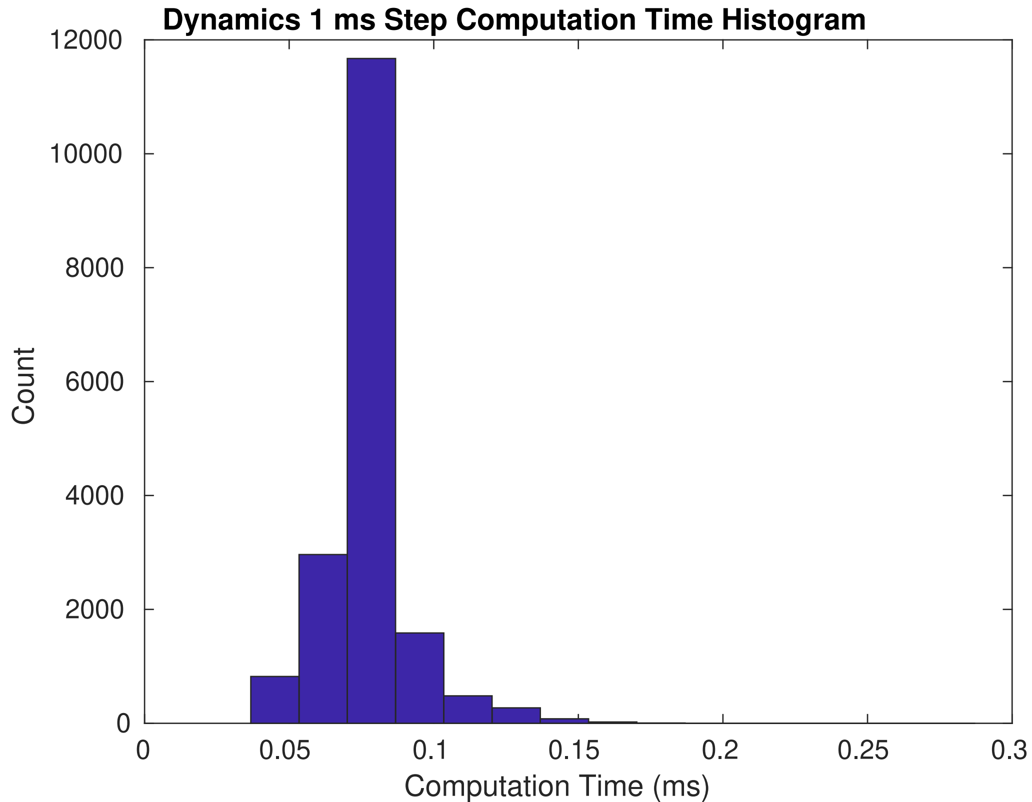
2.1.3 Dynamics Simulation Tests

To verify the simulation is correct, I initialized the robot with random initial conditions, and computing forward dynamics with two techniques: Featherstones algorithm, and solving equation (2.1). These were verified to be the same. At each time-step, I calculated the total kinetic energy of the system, both body-by-body, and with the mass matrix, using $E = \frac{1}{2}\dot{\mathbf{q}}^\top \mathbf{H}\dot{\mathbf{q}}$. These were also verified to be the same, and the total energy of the system was verified to be conserved.

To evaluate the performance of the dynamics algorithm, I measured how long it took to simulate forward a single 1 ms time-step during locomotion. Figure 2-2

shows a histogram for simulation calculation time. On average it took 0.077 ms to compute dynamics, which means the simulator will use 8% of CPU time to do forward dynamics.

Figure 2-2: Simulation time-step computation time histogram for forward dynamics calculation

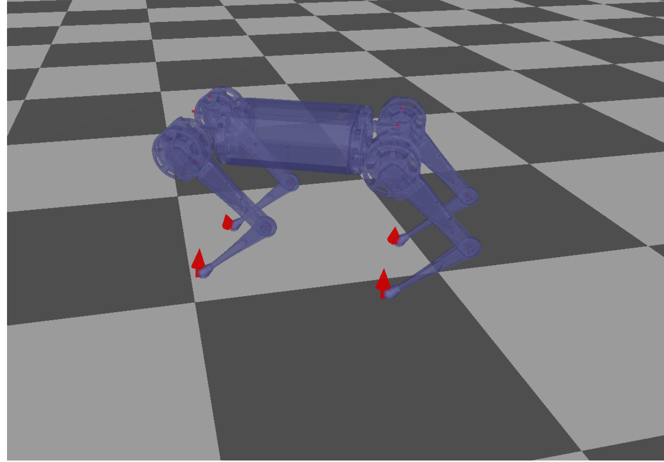


2.2 Contact Model

I implemented a simple contact system, which can detect contact between points and planes, and points and rectangular boxes. The terrain is modeled as a collection of planes and boxes, and the robot contains a number of ground contact points on various bodies shown in Figure 2-3.

I implemented a contact model described in [3], which implements the ground contact as a very stiff nonlinear spring and damper system. This approach was used in a previous Cheetah 3 simulator, and was very simple to implement. Unfortunately,

Figure 2-3: Simulation contact points for Mini Cheetah. There are 8 points in the corners of the body's bounding box, a point on each knee, and a point on each foot.



it requires very small time-step to be both stable and stiff enough to prevent the robot from sinking into the ground. For Mini Cheetah, it requires a time-step of 0.1ms to have acceptable performance.

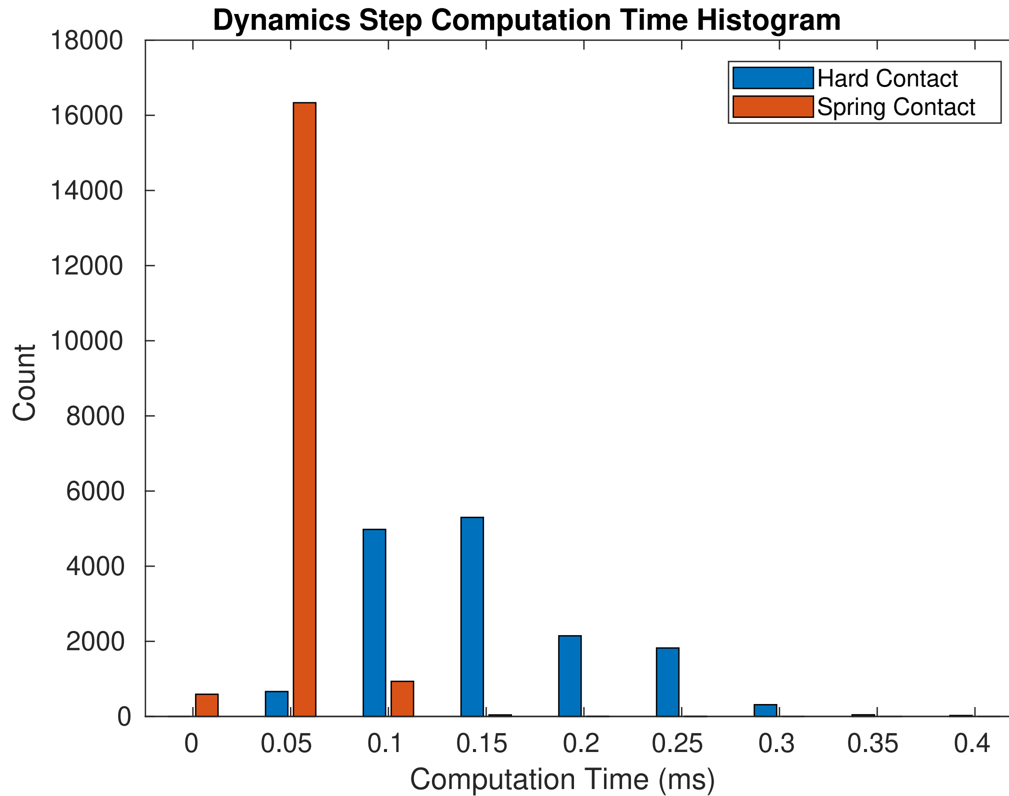
The simulator software was extended by Donghyun Kim, who wrote a more sophisticated contact solver based on an iterative approach to solve hard contact dynamics, as described in [23]. This contact model is slightly less accurate and more time consuming to solve, but can be used with a shorter simulation time step. I use this as the default contact model because it is accurate enough, and allows us to run our simulation with a 1 ms time-step. If more accurate dynamics are needed, the model from [3] can be enabled. At this point, we have not needed to use the more accurate contact model.

As shown in Figure 2-4, the spring contact is faster than the hard contact model. However, the spring contact must be simulated with a 10 times smaller time-step, so it ends up being slower in the end.

2.3 Actuator Model

In addition to the dynamics added by the rotors, I also consider the torque limits of the actuator. There are two limits for actuator torque:

Figure 2-4: Simulation time-step computation time histogram for forward dynamics and contact solve



- The absolute maximum allowed torque. This is 18 Nm on Mini Cheetah, which is described in more detail in [11].
- The maximum current the motor driver is able to achieve, limited by the battery voltage available and the back EMF of the motor. This will limit the motor's torque at high speed, and effectively sets maximum speed of the motor.

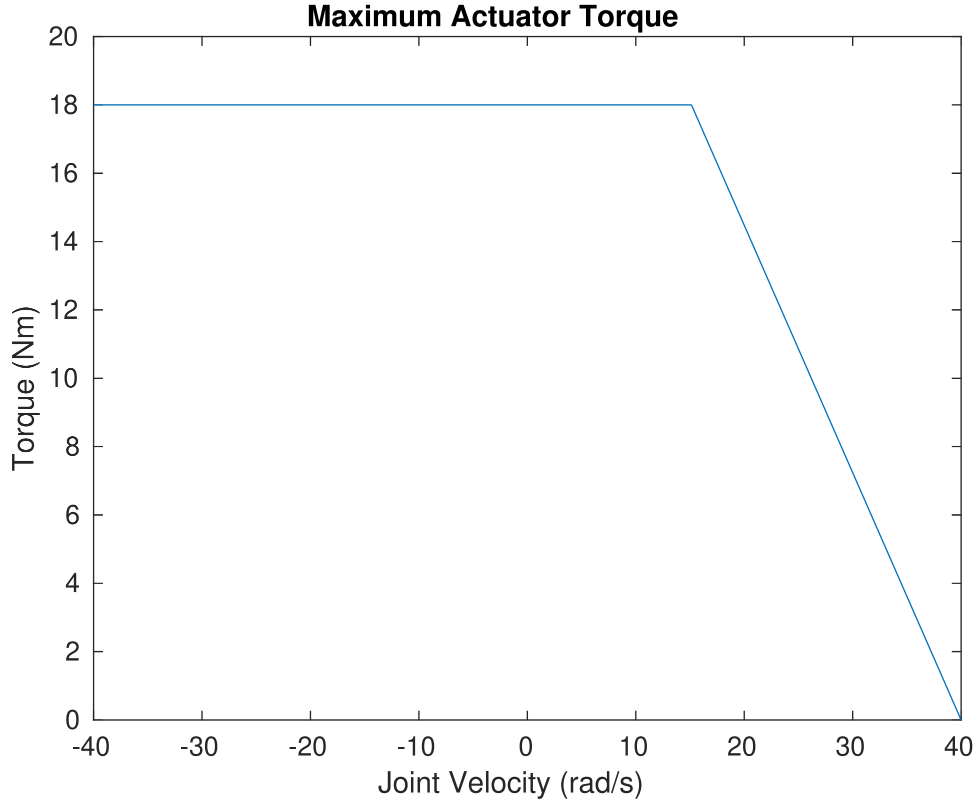
Together, these limits create the torque speed curve in Figure 2-5.

2.4 Low-Level Controller Model

2.4.1 Mini Cheetah

The actuators on Mini Cheetah have an internal microcontroller which can run a PD position and velocity control loop. This loop runs at 40 kHz, and determines the

Figure 2-5: Mini Cheetah simulated torque speed curve



motor torque setpoint with

$$\tau = \tau_{\text{ff}} + K_p(q_{\text{des}} - q) + K_d(\dot{q}_{\text{des}} - \dot{q}) \tag{2.2}$$

where τ is the final torque, τ_{ff} is the commanded torque, K_p is the commanded proportional gain, q and $\dot{q}$ are the measured position and velocity of the joint, and q_{des} and $\dot{q}_{\text{des}}$ are the command position and velocity of the joint. On the actual robot, this torque is used to determine the current setpoint for the motor. The closed loop bandwidth of the current loop is 1 kHz, and the simulator steps at 1 kHz, so I do not bother to simulate current dynamics.

This low-level controller is simulated by my simulator, but it cannot be simulated at 40 kHz because the dynamics time-step is either 1 or 10 kHz, depending on the contact model. Running this low-level controller simulation slower than on the actual robot means that the performance of the simulator may be slightly worse than the

actual robot. However, the robot's performance is limited somewhat by the finite resolution of the encoder causing noise in the estimated joint velocity. There is only a difference between simulation and hardware when the position control gains are extremely high and close to unstable, which is not something we use.

2.4.2 Cheetah 3

Cheetah 3 has a similar low-level controller, but it runs on all three motors in a leg instead of only a single motor. It uses the following control law:

$$\boldsymbol{\tau} = \boldsymbol{\tau}_{\text{ff}} + \mathbf{K}_{p,j}(\mathbf{q}_{\text{des}} - \mathbf{q}) + \mathbf{K}_{d,j}(\dot{\mathbf{q}}_{\text{des}} - \dot{\mathbf{q}}) + \mathbf{J}^{\top}[\mathbf{f}_{\text{ff}} + (\mathbf{K}_{p,c}(\mathbf{p}_{\text{des}} - \mathbf{p}) + \mathbf{K}_{d,c}(\dot{\mathbf{p}}_{\text{des}} - \dot{\mathbf{p}}))] \tag{2.3}$$

where $\boldsymbol{\tau}$ is the vector of motor torques for the leg, $\boldsymbol{\tau}_{\text{ff}}$ is the vector of commanded torques, $K_{p,j}$ and $K_{d,j}$ are diagonal proportional and derivative gain matrices for joint position control, $\mathbf{q}$ and $\mathbf{q}_{\text{des}}$ are the measured position and desired position of the joints, $\mathbf{J}$ is the foot Jacobian, $K_{p,c}$ and $K_{d,c}$ are diagonal proportional and derivative gain matrices for foot position control, $\mathbf{f}_{\text{ff}}$ is the command foot ground force, $\mathbf{p}$ and $\mathbf{p}_{\text{des}}$ are the actual and desired Cartesian position of the foot.

The Cheetah 3 low-level controller runs at either 5 kHz or 15 kHz, depending on the revision of the electronics. The simulator runs the Cheetah 3 low-level control simulation at 1 kHz when the simulator runs at 1 kHz, and at 5 kHz when the simulator runs at 10 kHz. Like with Mini Cheetah, the torque control is assumed to be perfect.

Chapter 3

Software

The Mini Cheetah software is developed in C++11. I developed the common libraries, robot control framework and simulator framework, and worked with two others to implement the dynamics and contact simulation. The robot control code was written by myself and three other members of the Biomimetics Robotics Lab. Other research groups are developing their own control code which they will use with our simulator and framework.

The software runs on both Mac OS and Linux. The robot itself runs Linux.

3.1 Organization and Design

Before starting the Mini-Cheetah software, I had previously worked on the implementation of the Cheetah 3 software, which was written in MATLAB/Simulink and used the Embedded Coder toolbox to generate code for the robot. I made the decision to switch from MATLAB/Simulink to C++.

We had original choose MATLAB/Simulink over C++ for the following reasons:

- **Familiarity** The Biomimetics Lab mostly has mechanical engineering students. MATLAB is often taught in undergraduate mechanical engineering, but C++ is not. MATLAB is also an easier language to learn and has fewer opportunities to create hard-to-find bugs.

- **Useful Built-in Tools** MATLAB can easily manipulate matrices and vectors, and the various toolboxes have well-written implementations of commonly used functions.

However, during my time working on the Cheetah 3 software, we discovered a number of disadvantages compared to C++.

- **Pricing** The cost of purchasing all of the toolboxes used in the MATLAB-based Cheetah 3 software is over $30,000, many times the cost of Mini Cheetah. It is unreasonable to expect any of our collaborators to spend this much to run our software. This is many times the cost of the entire robot. MIT provides these toolboxes for free, but most Mini Cheetahs will be used outside of MIT, where MATLAB is not free. The C++ compiler and all libraries we use are completely free.

- **Compile Times** It takes at least 5 minutes to rebuild the Cheetah 3 MATLAB/Simulink software after making a single change. During a roughly 10 hour day of working on Mini Cheetah software, I rebuilt the control code 215 times. If it took 5 minutes for each build, the 215 rebuilds would require would require 18 hours of build time. Rebuilding the C++ Mini Cheetah software takes anywhere from 5 to 30 seconds to rebuild, depending on how much code is changed.

- **Bugs** The MATLAB and Simulink Coder products often generate code which crashes or computes something different from the original MATLAB code due to bugs in the code generation product. Debugging this is extremely time consuming and our bug reports have not been addressed for years. Our lab has encountered no C++ compiler bugs.

- **Version Differences and Compatibility** Each year MATLAB is updated, the Cheetah 3 software requires changes to build without errors. Several people attempted to get Cheetah 3 software to build in MATLAB 2019 and they gave up after a few days because the MATLAB code generation tool would simply

crash. Any C++11 compiler will be able to compile our Mini Cheetah code. We have tested it on both ‘gcc’ and ‘clang’, the two most popular open-source C++ compilers, and it works on any version from 2013 (the first C++11 compatible version) or newer.

- **Performance** The performance of certain functions in MATLAB Coder is very poor. For example, computing the pseudo-inverse of a 3x3 matrix in MATLAB takes more time than the rest of the control code combined. Determining which functions are slow is time consuming, as MATLAB embedded coder has no profiling tools. There are many free tools which can profile C++ code.

- **Functionality** Many of the more advanced language features of MATLAB that would be useful for writing higher level state machine code does not work with embedded coder. I believe that C++'s templates, classes, and polymorphism features are better suited for this type of code than.

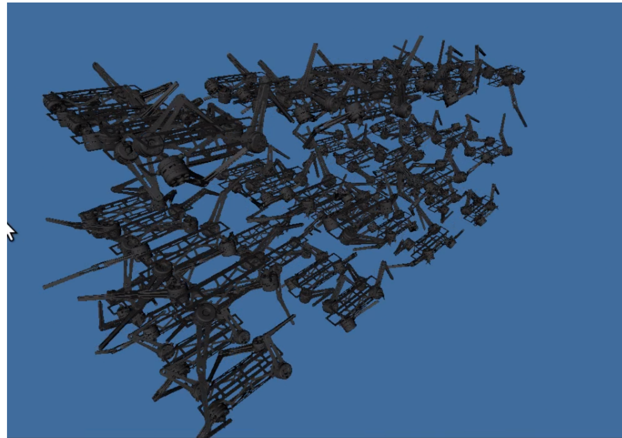
- **Third-Party Libraries** Calling C code from MATLAB coder is possible, but is error prone and time-consuming to set up. Interfacing with C++ code from MATLAB coder is technically possible, but does not allow you to call member functions and does not do name mangling correctly, so it is not useable in practice. This means that all third-party libraries we use need to have C wrappers written for them. Calling C/C++ library code from C++ is very straightforward.

In the end, I decided to switch to C++. The final motivation was performance. Before we built Mini Cheetah, we were worried that we would need to run the Mini Cheetah control code faster than the Cheetah 3 control code, as the robot is physically much smaller and has faster time constants. However, Cheetah 3's computer could not run the code any faster than 1 kHz, and Mini Cheetah was going to have a much slower computer.

An additional goal of the new simulator was for it to be able to run many simulations in parallel and faster than real time for data collection or machine learning.

While the current version of the simulator has no option for doing this, the code is designed so that it is possible to create multiple Simulation threads and have them all report to a common visualizer, or run without a visualizer. In Figure 3-1, there are 64 simulated Cheetah 3's all reporting to the same visualizer.

Figure 3-1: 64 Cheetah 3 Robots are simulated at 1.5x real time simultaneously on a 18-core CPU



3.2 Third-Party Libraries

I used a number of open-source libraries:

- **Eigen**[7] is a linear algebra template library. It is used for matrix and vector operations.
- **Lightweight Communication and Marshalling (LCM)**[8] is used to send data over the network between the Mini Cheetah and the control software. The control software sends commands such as gains and user input, and the robot sends back data to be logged or visualized for debugging.
- **qpOASES**[6] is a active-set quadratic programming solver suitable for smaller, dense problems
- **MIP** is a communication library for the protocol used by Microstrain IMUs
- **yaml** is a parser for YAML format configuration files

- **VectorNav - VN** is a library for communicating with a VectorNav IMU
- **Qt** is a UI framework and is also used to get the input from a USB game controller
- **OpenGL** is a graphics API for GPU accelerated 3D graphics
- **GLUT** is a collection of tools for OpenGL used to creating windows and getting mouse and keyboard input
- **Google Test** is a unit-testing framework
- **SOEM** is an EtherCAT Master communication library.

3.3 Linux on the Robot

In order to ensure real-time operation on the robot, I built a version of the Linux kernel which incorporates the PREEMPT RT patch. This drastically improves the timing accuracy of the system by making both the kernel itself and interrupt handlers fully preemptable. When running a 1 kHz loop for 5 minutes without the patch, the average timing error was 0.5 milliseconds, with a maximum of 19 milliseconds. With the patch, this dropped to 0.02 milliseconds and a maximum of 0.1 milliseconds.

Additionally, the Mini Cheetah's kernel contains an additional patch to add a driver for its Serial Peripheral Interface (SPI) connection to the legs. I used the `spidev` API from Linux to access this device.

The Cheetah 3 leg controllers are connected over EtherCAT. I used SOEM to send and receive data from the leg controllers.

3.4 Software Organization

The software is split into several components. There is a common library which contains utility functions used by other components. The Simulator contains a dynamics simulator, 3D visualizer, and a user interface for modifying the terrain and tuning

gains. The Robot Framework contains a library of functions to interface robot control code with both the simulator and the actual hardware. The User Controller code contains robot control software and relies on the Robot Framework library. Groups who use the MIT Mini Cheetah are expected to write their own User Controller code, but may use ours as an example or starting point.

An important feature of my software is that the same User Control executable can run on both the simulator and on the robot. The user can pick if the control code should connect to the simulator, or to the robot hardware with a command line flag at run time. This way, there will be fewer bugs that only happen in hardware. A large amount of time is lost to debugging, and I have found that using the same executable file to run on both the robot and the simulator has reduced the number of bugs. After the code is run in the simulator, the exact executable can be copied directly the robot.

3.5 Robot Framework

The robot framework is a library which allows users to write programs which interact with the robot hardware or simulator. A program written using this framework will accept a command-line argument at runtime to pick between Cheetah 3 and Mini Cheetah, as well as to pick between simulation and robot hardware. The Robot Framework will call the users control code at the appropriate rate and is responsible for setting up and running all interfaces to simulator and hardware.

3.5.1 Leg Controller

The Robot Framework allows access to a "Leg Controller" for each leg. This is designed to abstract away the differences between Cheetah 3 and Mini Cheetah, as well as differences between simulation and hardware. The Leg Controller will internally take care of communication with the "low-level" controller on that robot. The computer running Linux and the control software is referred to as the "high-level control".

Mini Cheetah and Cheetah 3 both have low-level feedback controllers that are

	Mini Cheetah	Cheetah 3
High Level to Low Level Rate	500 Hz	1 kHz
Low Level Rate	40 kHz	15 kHz
Torque Command	Yes	Yes
Force Command	No	Yes
Joint PD	Yes	Yes*
Foot PD	No	Yes
Max Torque	No	Yes
Encoder Zero	No	Yes

Table 3.1: Low level leg control capabilities of the MIT Cheetah 3 and MIT Mini Cheetah robots

separate from the main computer which can run higher frequency feedback control than is possible on the computer. The Mini Cheetah low-level controllers do feedback per joint, and the Cheetah 3 controllers do feedback per leg, so they have different possible commands, as shown in table 3.1.

Note that the Joint PD for Cheetah 3 is a new feature. It has only been tested on a single motor on an in-progress redesign of the Cheetah 3 body. The current Cheetah 3 shown in this thesis does not have this.

The leg controller allows users to command torques, foot forces, foot position control, and joint position control. If possible, it will pass the gains and setpoints down to the lower level control. Otherwise, it will run the feedback controller on the main computer. It determines the current joint angles, velocities, foot position, foot velocity, foot Jacobian, and leg link lengths.

Additionally, Cheetah 3 has an option to limit the maximum motor torque and to set the encoder offset. We have not found a need for motor torque limits on Mini Cheetah, and Mini Cheetah uses absolute encoders, so these options only work on Cheetah 3 and do nothing for Mini Cheetah.

3.5.2 IMU

The robot framework can read IMU data into a common format from three different kinds of IMUs: VectorNav VN-200 (Mini Cheetah prototype, New Cheetah 3), LORD Microstrain IMUs (Mini Cheetah), and KVH IMUs (Old Cheetah 3). In simulation

mode, this data is pulled from the simulator instead. The simulator adds noise to the IMU, and samples at the appropriate rate for the IMU being simulated. All IMUs report their acceleration, angular rate, and orientation.

3.5.3 Remote Controller

The robot framework can read commands from a Logitech F310 game controller, which is provided from the user interface, in both simulation and when running on the robot. It can also read commands over SBUS, a protocol used by R/C receivers. This allows us to control Mini Cheetah with an R/C transmitter commonly used for R/C cars or airplanes.

3.5.4 Dynamics Model

Finally, the robot framework allows the user to easily run our state estimator, and get a full dynamics model of the robot, including forward kinematics, mass matrix, Coriolis torques, gravity torques, Jacobians, and other information. There is also the ability to get the maximum allowable torques, velocities, leg lengths, and other robot-specific information.

3.6 Simulator and User Interface

The simulator is a user interface and 3D visualizer which controls the dynamics simulator for Mini Cheetah. Currently it only supports running a single simulation at a time, but the code is designed to be easily extendable to run many in parallel for data collection or machine learning. It can also run the simulation faster at a given rate, or as fast as possible. Assuming the computer is fast enough to achieve the desired simulation rate, there is a feedback system which ensure that the simulator time matches real time and does not slowly lag behind over time. This seems like a straightforward feature, but most dynamic simulators I tested, and my initial implementation, ran at around 98% real time, even if the computer was fast enough to run

at real-time. This is an issue if the simulation will ever need to communicate with other software which expects to be running in real time. In my simulator and other implementations, this timing error was caused by a few reasons:

Error Tracking

If at one point, the simulation runs slower than real time, the simulator needs to try to catch up to the correct time. This allows the simulator to keep up with real time even if there are random frames which are slower than real time. In my implementation, this feature will disable itself if the desired simulation time is too far in front of the actual simulation time, and will reset itself if the desired simulation speed is modified. For instance, if the user picks a desired sim rate of 100x real time, the simulator may only run at around 5x real time because the computer is not fast enough, and the desired simulation time will get very far ahead of the actual simulation time. If the user waits here for a long time, and he then sets the desired simulation rate to 1x real time, he probably expects the simulator to return to real time, rather than attempted to catch up. By detecting if the computer is too slow for the desired rate and temporarily disabling this feature, the simulator will be more responsive to changes in desired simulation speed.

Sleep Accuracy

Even if the simulator detects that it is running behind, it may not be able to catch up if it relies on a timed sleep function like `usleep`. In practice, using `usleep` as a delay will cause your thread to sleep for slightly more than the requested amount. A sleep which does not overshoot can be accomplished by calling `usleep` for half the time remaining, until the time remaining is very short, then just spinning in a loop until the correct time is reached.

Timestep Counting

In most implementation, all the simulation for an entire graphics visualization frame is run at the same time. This is because it is not efficient to sleep after every single

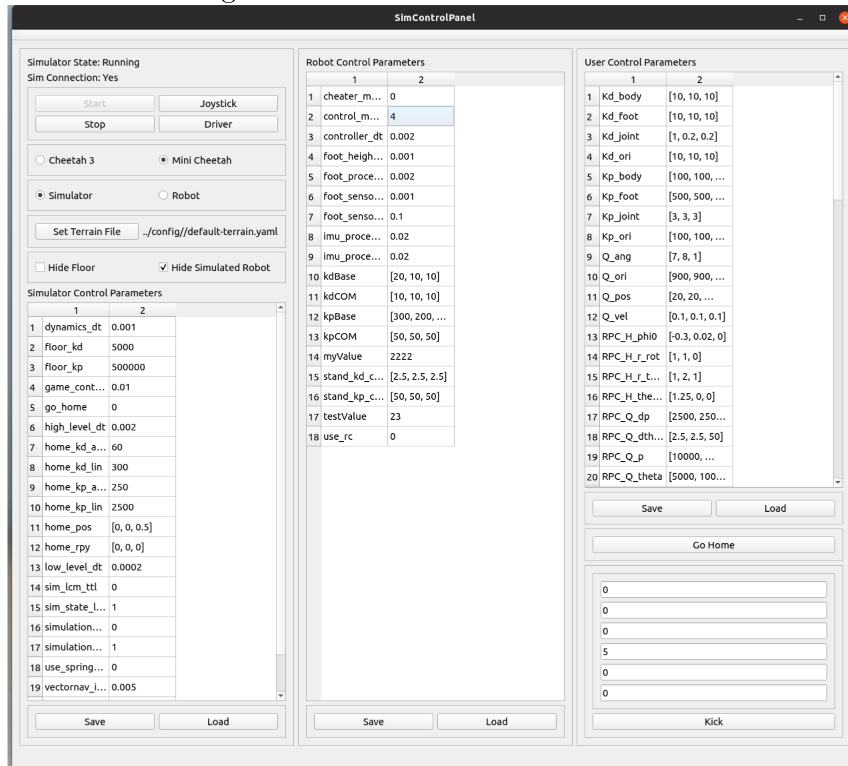
simulation time-step. It is better to run all the simulation required for a single graphics frame as fast as possible, then wait for the frame to be displayed until simulating the next frame. Doing this requires the simulator to track how many time-steps must run per frame. However, this number is not typically an integer, and it is only possible to run an integer number of time-steps. In order to prevent rounding error from accumulating, it is necessary to track the "fractional time-steps" that are not run. Over the course of multiple graphics frames, these fractional steps will add up to a full time-step, which should be added to a graphics frame.

To test these techniques, I ran a simulation of the robot trotting for around 60 hours and recorded the real time and simulation time of each simulation time-step. All of the over 200 million time-steps occurred within 1 graphics frame (16.667 ms) of the correct real time. Without the combination of these three techniques, this did not work.

3.6.1 Control Parameters

Additionally, the simulator has a user interface for changing parameters for the robot control code and controlling the simulator, as shown in Figure 3-2. Parameters can be integers, floating point numbers, or vectors of 3 integers or floating point numbers. There are groups of parameters for the simulator, robot framework, and user control code. The simulator parameters control settings like the friction, simulation rate, and contact dynamics. The robot framework settings allow you to adjust the high-level control rate, maximum torques, and enabling the Cheater Mode state estimator. The user control parameters are defined by the user, but they typically allow the user to select a control mode or tune controller gains. The simulator GUI does not need to be recompiled if new parameters are added to the user control code, or if a different user controller with a different set of gains is used. All parameters can be saved and loaded to a file. The names, types, and values of all parameters are sent to the robot code before it initializes, so that they may be used in the constructor of the user's controller. The framework will check that all parameters are sent, and that all of the types are correct.

Figure 3-2: Simulator Control Panel



3.6.2 Terrain Description File

The user can specific simulation terrain through a text file containing a list of objects. Currently objects can either be planes, boxes, or stairs. The size, friction, position, orientation, color, and transparency of objects can be specified. In the case of stairs, there are additional parameters for the step size and number of stairs. An example terrain can be seen in Figure 3-3. The terrain file is listed below:

```
# default-terrain file
ground-plane:
    type: "infinite-plane"
    mu: 0.5
    height: 0.0
    graphicsSize: [20,20] # how large the plane appears
    checkers: [40,40]     # how many checkers are on the plane

box-1:
```

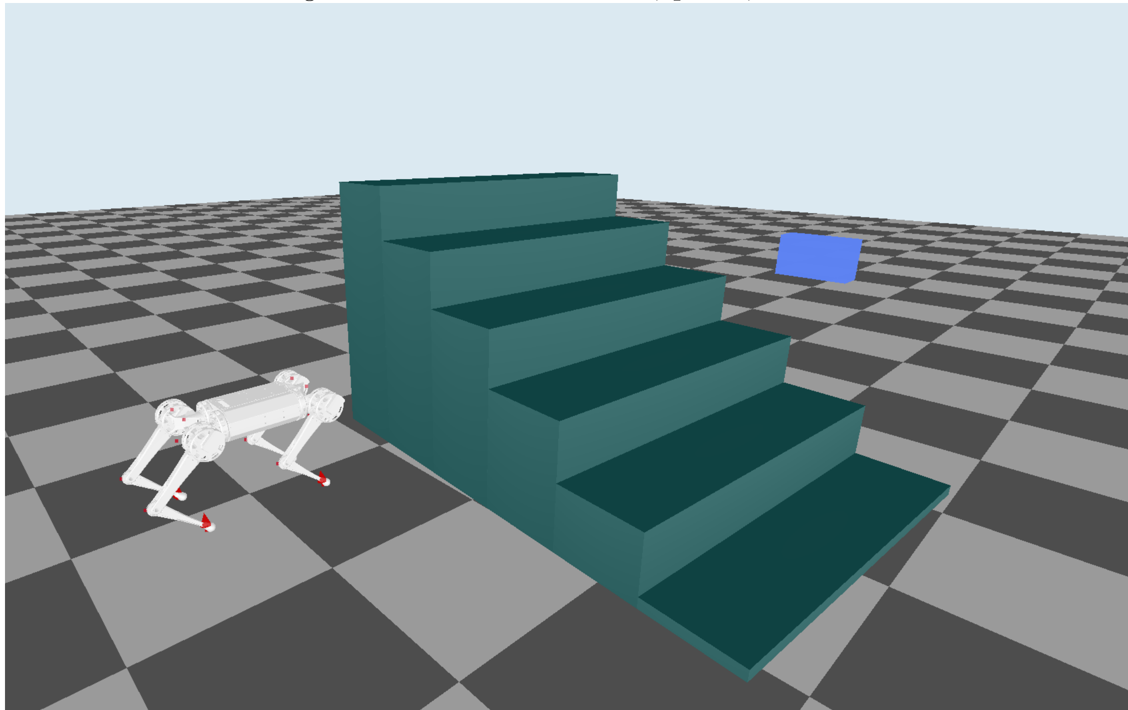
```
    type: "box"
    mu: 0.7
    depth: 0.2
    width: 0.5
    height: .3
    position:  [4.5, 0, 0.23]
    orientation: [0,.0,0]  # roll pitch yaw
    transparent: 1

stairs-1:
    type: "stairs"
    steps: 8
    rise: 0.1778 # 7 inches
    run: 0.2794 # 11 inches
    width: 1.2
    mu: .7
    position: [1,-2,-0.5]
    orientation: [.0, .0, 1.5]
    transparent: 0
```

3.6.3 Visualizer

The 3D visualizer uses OpenGL. I wrote a very simple shader which uses Phong shading [17] for lighting, and implemented a simple draw list which can draw meshes with either a solid color, or per vertex color. Meshes are loaded through an "obj" file reader I wrote. To improve efficiency, it is designed so that each model is uploaded to the GPU only once, and the geometry data is reused for each time the prototype is drawn. I also simplified the meshes of the robot in Blender to reduce the triangle count to around 100,000 for the entire robot. This allows the visualizer to run at 60 fps all of the lab computers, including older hardware. The previous MATLAB visualizer could not run at 60 fps on any of the lab member computers. I believe that

Figure 3-3: Terrain of stairs, plane, and box



being able to run the visualizer at 60 fps always is a huge advantage to understanding how the robot is behaving. Figure 3-4 shows the visualizer displaying 25 Cheetah 3 models and running at 60 fps, while all are being simulated in real time on an 18-core machine.

The visualizer can display both the simulator state and the state estimate of the robot. Additionally, the user may add debugging visualizations which can include arrows, lines, paths, cubes, and spheres. Figure 3-5 shows an example of the debugging visualization I created for the swing leg controller. The red arrows indicate ground reaction force direction and magnitude, the green balls indicate the actual foot position from forward kinematics, the red balls indicate the desired foot position, the green path indicates the desired leg trajectory, and the yellow ball indicates the desired touchdown location. The blue balls indicate the position where the stance feet touched down.

Figure 3-4: Visualization of 25 Cheetah 3 robots

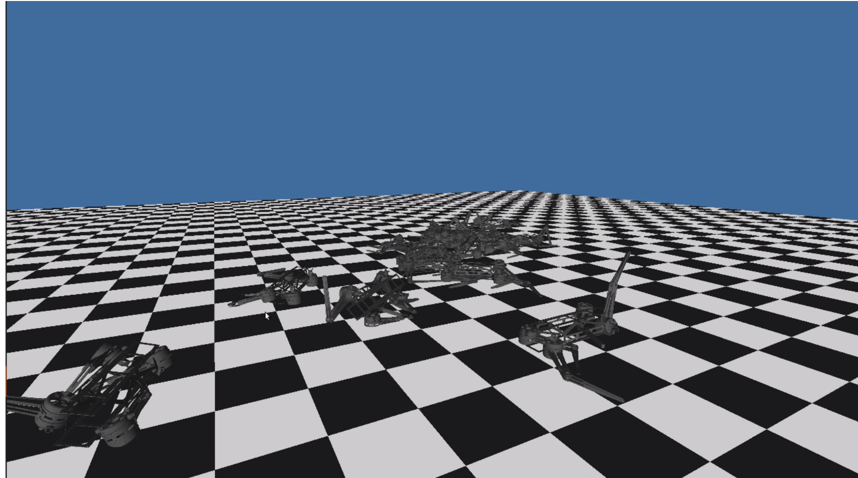
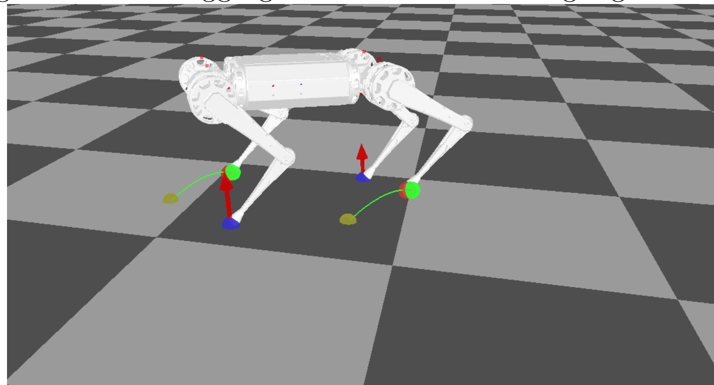


Figure 3-5: Debugging visualization of swing leg controller.



3.6.4 Connection to Hardware

The simulation UI can also be used when running an experiment on the real robot. The parameters can be tuned, and the robot can be visualized. Additionally, there is a debug mode for the Mini Cheetah hardware which allows motor controller status and SPI communication status to be inspected, and motor control commands to be manually specified. Figure 3-6 shows this user interface. This was used when assembling the 10 Mini Cheetahs in an actuator module test setup shown in Figure 3-7. This allowed us to check all the electronics before assembling the entire robot.

Figure 3-6: Debugging interface for Mini Cheetah

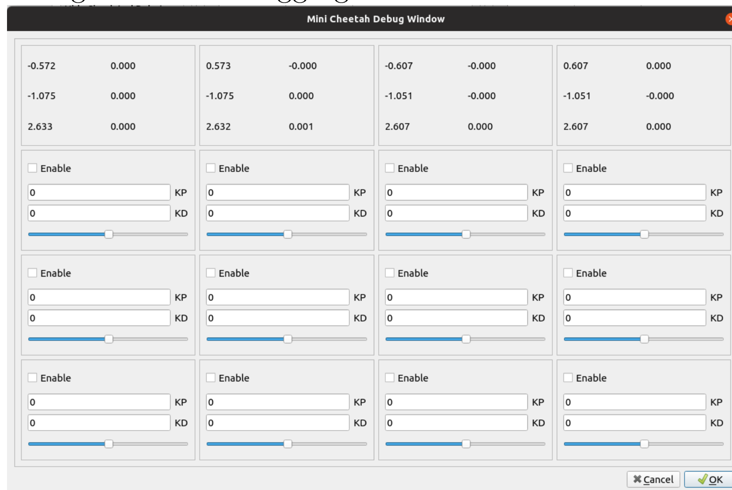


Figure 3-7: Actuator test setup for a single Mini Cheetah

3.7 Error Handling

It is important for the user interface to be robust to errors or bugs in the user code. The user interface should not crash when the robot control code has an error so the user does not lose their settings, and so they can observe the last simulator state before the crash. It should also display a useful error message to help the user figure

out what went wrong. The previous MATLAB-based simulator would often crash if the robot control code had a bug, which made debugging extremely challenging, as it was very hard to tell if the bug was in the simulator or the robot control code.

The Simulator itself has four states: Stopped, Starting, Started, and Error. The default state is Stopped, where there is no simulator thread running.

The Starting state creates the simulator thread, starts communicating with the robot code, and verifies that the set of user parameters given to the simulator matches the set of user parameters the robot code is expecting. If there is a mismatch or other issue, it goes into the Error state. After receiving and checking all parameters, the robot code framework will call the constructor of the user's robot controller.

The Started state is reached only after the communication between the two programs has been successfully started and the robot controller's constructor has succeeded. If there is an error in either, it will enter the Error state.

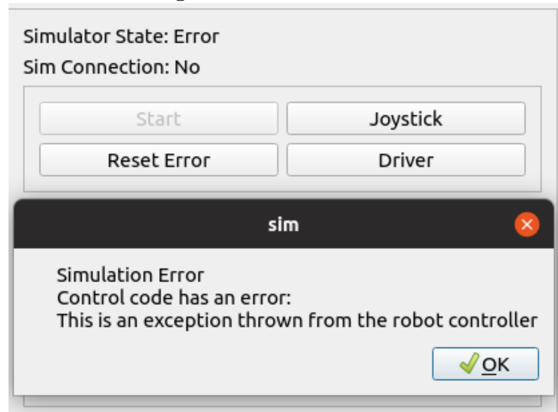
The "Error" state in the simulator allows you to inspect the last state of the simulation, and also allows you to reset the simulation. A "Reset" will clean up all resources allocated by the simulation thread. This recovery can be done without restarting the simulator UI, so the user does not lose his settings.

3.7.1 Robot Control Code Exceptions and Crashes

If the robot controller throws an exception, the robot control framework will catch it and send its message to the simulator over shared memory, which will go into the "Error" state and display the text message of the exception, as shown in Figure 3-8.

If the robot controller code crashes due to a segmentation fault, a divide-by-zero error, out-of-memory error, or similar, a POSIX signal handler installed by the robot control framework will inform the simulator of the reason so it can display an error message. Additionally, the `backtrace` function provided by `glibc` is used to generate a stack trace which is displayed so the user can figure out where the bug is located.

Figure 3-8: Error message when control code throws an exception



3.7.2 Simulator Internal Error

It is possible for the dynamics simulator or shared memory connection to have an error that is completely unrelated to the 3D visualizer or UI. This typically happens if a terrain configuration file is invalid, or if the robot control code has a bug which causes it to write garbage data into the shared memory. Most commonly, this occurs when the simulator encounters an invalid floating point number ("NaN"), caused by either a "NaN" coming from robot code, or a bug in contact dynamics. When this occurs, the simulator thread can throw an exception and be terminated, but the UI will remain open. The UI can still recover from this without restarting by cleaning up the crashed simulator thread and starting a new one.

3.7.3 Timeout

One case which is hard to handle correctly is when the robot code gets stuck in a loop and never responds to the simulator. In this case, the simulator will detect that the robot code is unresponsive, and will display an error message prompting the user to check on the robot code to see what has happened. This is probably the hardest error for a user to debug, but it does not happen often - most of the time the robot code will crash rather than get stuck in a loop.

One simple but extremely helpful feature is to have the robot program kill itself when the simulation ends. This way, you do not accidentally leave a robot program

running in the background, sending messages and causing issues. Forgetting to kill old robot programs had cost us hours of debugging and confusion on the old software, but has not happened yet on the new software because of this feature.

3.8 Communication between Simulator and Robot Framework

The simulator and robot programs must communicate and synchronize with each other. On each time step, the simulator will need to provide the robot code with the joint encoder positions and velocities, changes to control gains, inputs from the USB game controller, the simulated IMU readings, and also a “Cheater State”, containing the full simulated state of the robot. The purpose of the “Cheater State” is to help users debug their state estimator code when running in simulation, by allowing them to compare their state estimate to the actual simulation state.

The robot will send back the joint commands, as well as optional debug visualization data, which can be used to draw arrows, lines, paths, cubes, spheres, and other robots in the 3D visualizer.

Previously, in the MATLAB-based simulator, this communication was done with LCM. This uses POSIX sockets and the operating system's networking stack to send data. We found that this implementation was highly inefficient and would reach 100% CPU utilization when sending 2 kB packets at 2 kHz (4 MB/sec). This is an issue because the new simulator should be able to run faster than this, and has messages that are around 1000 times larger (2 to 3 MB), due to the addition of debugging data.

In the new simulator, I redesigned this so the two programs communicate using POSIX shared memory, and use POSIX unnamed semaphores for synchronization. After developing on Linux, I found than in Mac OS the operating system does not support POSIX unnamed semaphores, so I wrote an implementation using POSIX condition variables instead. On Linux, I have found that the condition variable approach is slower, so it continues to use semaphores, and there is a compile time option

to switch between the two.

Part of the reason the new approach is so much faster is that the data to be sent is never copied - it is written directly to memory, and read by the other program. With LCM, the memory is copied around a number of times, and must go through the operating system's networking stack. Additionally, the implementation of the synchronization primitives on Linux is very efficient.

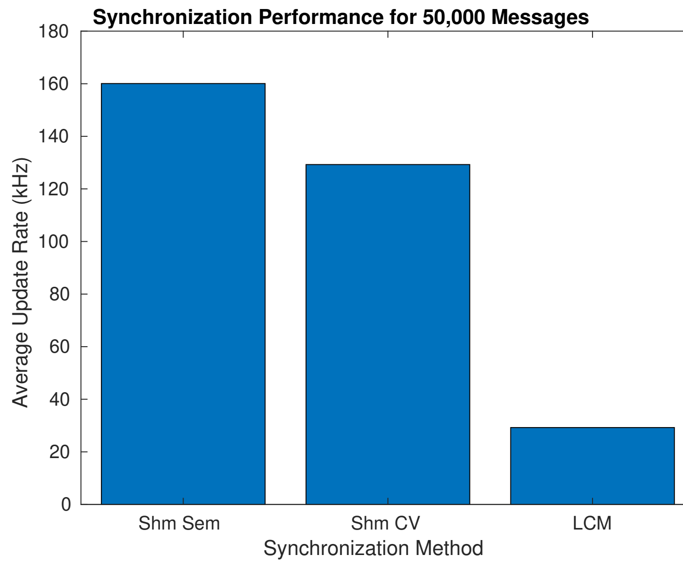
3.8.1 Performance of Communication

I measured the performance of the simulator/robot code communication in two ways. The first is the measure the overhead of synchronization, and the second is to determine the largest amount of data that can be sent while running 1000 time-steps per second in real time.

Synchronization

The robot code and simulator must synchronize so that they take turns running. To measure how efficient this is, I set up an empty loop for both the robot and simulator. When it is the robot's turn to run, it simply verifies that the time-step is correct, then reports to the simulator that it is finished and what time-step it is on. The simulator does the same. I then timed how long it takes to do 50,000 iterations of this to determine how efficient the synchronization is. The results are shown in Figure 3-9. The “Shm Sem" category is the shared memory semaphore, the “Shm CV” is the shared memory condition variable, and the “LCM” category is LCM. Shared memory has much higher performance than LCM in this case, but both are more than fast enough to run faster than real-time. Note that LCM would only be able to run roughly 3x real time if the simulator is stepping with 10 kHz real time, which may be an issue.

Figure 3-9: Synchronization performance

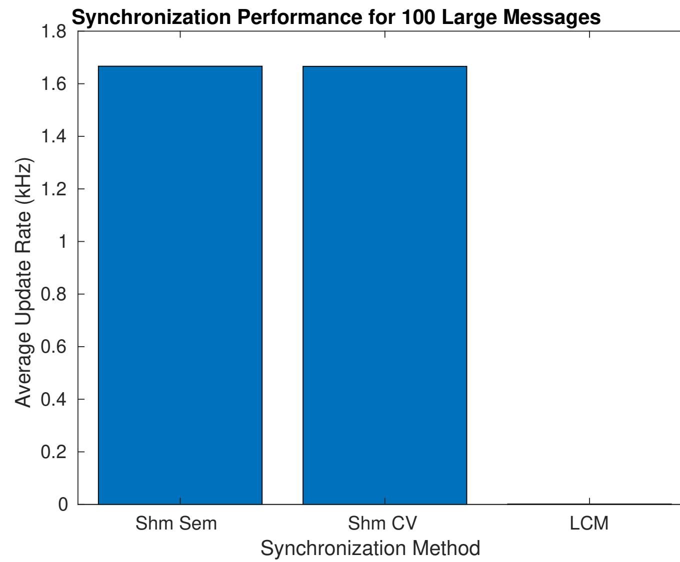


Sending 2 MB Messages

This test is much more useful, and measures how fast 2 MB messages can be sent. By comparing these results to the synchronization part, we can tell how much of the time is spent on synchronization overhead versus actually sending the data. For this test, the robot and simulator will write data into a 2 MB buffer. The robot will write all 1's and the simulator will write all 0's, and they will verify each other. The 2 MB message size was chosen because this is the size of the robot to simulator message, which can contain a lot of data for the debugging visualization.

In this test LCM's performance is very poor compared to shared memory, as shown in Figure 3-10. Both of the shared memory methods were able to achieve above 1.5 kHz, but LCM was only able to achieve 1.6 Hz. This is a difference of almost 1000x times.

Figure 3-10: Synchronization performance



3.9 Communication between UI and Robot Hardware

When running the robot code on hardware, it is still desirable to have communication back to the 3D visualizer to view the robot's estimated state, and also to the UI to tune gains and receive commands from the Logitech game controller. To accomplish this, I used LCM. Because LCM is much slower, I have to remove a lot of the data that is sent back, and send some data less frequently.

Chapter 4

Cheetah Locomotion Controller

The goal of my Cheetah Locomotion controller was to allow the robot to perform arbitrary gaits. In particular, I was interested in galloping. It is not clear if a galloping gait provides an advantage over other gaits for the Cheetah robot, but at the time, there were no other documented locomotion controller which supported galloping, and a controller general enough to handle galloping would also likely extend to most other desirable dynamic gaits, such as bounding, pronking, trot-running, and more complicated gaits designed to cross specific terrain or obstacles.

Dynamic gaits like galloping are challenging for a few reasons. First, it requires the robot's body to deviate from being level - the body will roll, pitch, and yaw significantly during galloping, which requires a fully 3D orientation model. Secondly, it requires a controller or a plan which anticipates the upcoming changes in contact sequence. For example, before the flight phase of pronking, the controller should cause the robot's base to have an upward velocity so the body does not fall too low during the flight phase. When the front legs are on the ground during bounding, the controller should pitch the robot back, so that when the back legs are on the ground, they can apply enough force to fight gravity without causing the robot's pitch angle to be too far forward. Without knowledge of the upcoming stepping pattern, it is impossible to effectively stabilize these types of gaits. Imagine trying to run, but not knowing which foot you will step with next or when - it would be very challenging.

Another challenge occurs when only one or two feet are on the ground. At these

instances, the dynamics of the body are uncontrollable. With two feet on the ground, you cannot apply a torque to tilt the body on the axis that is parallel to the line between the two feet, and when one foot is on the ground, you have 3 control inputs, but the body has 6 states, so it is very uncontrollable.

A final challenge of locomotion is the constraints imposed by contact with the ground. The robot cannot use its feet to pull itself closer to the ground, and must apply forces to the ground that do not cause the feet to slip, limiting the wrench which can be applied to the body even further. Working around these constraints is the key to stable locomotion.

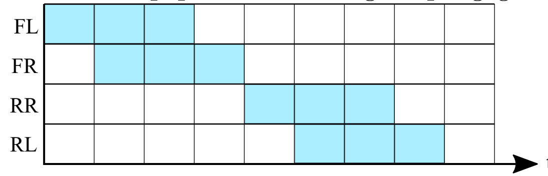
These unique challenges mean that traditional control methods are typically not applicable to legged locomotion. From a classical control perspective, the changing contact sequence looks like the model and dimension of the control inputs is changing over time. One alternative approach is to use hand-written, physics-based heuristics which capture properties of the gait. A second is to use a more general approach which is not designed with the specific physics of legged locomotion in mind. I chose to use a more general approach, but made modeling decisions based on knowledge of locomotion heuristics. Using a more general approach has the advantage of working on a different kinds of gaits, and can be used in the future with higher level planners which plan out irregular gaits for crossing challenging terrain.

4.1 Gait Definition

I define a gait as a sequence of contact states. In my implementation for Mini Cheetah, the contact states are defined at evenly spaced time steps, typically spaced around 30 milliseconds apart. This time-step was chosen because it is fine enough for me to accurately specify the timing of a gait, but coarse enough to have a reasonable number of contact states in a plan. Figure 4-1 shows an example of a contact sequence for a galloping gait without flight

The contact state has a flag for each of the four feet indicating if it is in swing or in flight. I use $\mathbf{c}[i] \in \{0, 1\}^4$ to indicate the contact state of all four legs at time-step

Figure 4-1: The footstep pattern for a galloping gait without flight



i, where a 1 indicates contact, and a 0 indicates swing.

Most commonly used gaits are periodic, meaning the sequence of ground contacts repeats. It seems desirable that the robot's motion during these gaits is periodic, ideally with the same period as the gait, although it is not clear if there is a practical benefit.

The periodic gaits I consider are trotting, trot-running (sometimes called flying-trot), hopping (sometimes called pronking), bounding, pacing, galloping, and a three-legged gait. One feature of all these gaits is that the period is relatively short - the robot completes one cycle in anywhere from 200 to 500 milliseconds. Each gait cycle is divided into 10 time-steps. For these gaits, I took the approach that I would always include at least one full cycle of the gait in my controller's plan. If I did not consider a full cycle, I would be unable to enforce a periodicity constraint on the motion of the body, if desired.

However, there may be some gaits which are not periodic, or have a period which is too long to fully consider in the plan due to computational limits. In the future, our lab may use a high-level planner to design an irregular gait to cross a specific obstacle. For these gaits, it does not make sense to think about a "cycle". To test this type of gait, I implemented a trot-like gait, but modified it so the frequency of the pairs of legs is not matched. The front left/back right pair of legs steps at $\frac{9}{13}$ of the frequency of the rear legs, leading to a gait period of approximately 35 seconds. For these gaits, I considered only the next 10 time-steps, or roughly 250 milliseconds. I believe this 10-step plan is a reasonable length because it is long enough for the locomotion controller to have time to anticipate contact sequences like in the periodic gaits, but is not unreasonably long for a high-level planner to generate. Current work with terrain detection in our lab considers the terrain 1 meter in front of the robot,

while traveling at around 0.25 meters per second. This means that the planner would be able to plan for the next 250 milliseconds.

By allowing gaits which are designed on-the-fly, I cannot use any precomputation to determine a nominal trajectory - the controller must generate these trajectories on the fly.

4.2 Prior Work

Some of the most impressive dynamic locomotion controllers are surprisingly simple. A good example is the work presented in [21] and [20], both published in 1986. Due to computational limits of the time, these controllers all used very simple physics-based heuristics. In *Running on Four Legs As Though They Were One* [20], an approach for dynamic quadruped running is given, but it does not extend to galloping due to the complicated footfall patterns, and the requirement for the robot's body to deviate from level. It can only work on a trotting gait, where the body can remain level.

A more recent example is the bounding controller used on Cheetah 2 [16], but this uses a planar model, and I could not generalize this approach to more complicated gaits in 3D, with arbitrary contact sequence timing.

Simulations of galloping robots exist, such as [12], but they are not fast enough to run in real time, and as a result have not run on hardware.

The only prior results of controlled galloping on hardware is on Cheetah 1 [15], which ran attached to a rail, eliminating roll, yaw, and lateral dynamics, and the Boston Dynamics WildCat robot [1]. The Cheetah 1 controller cannot be extended to 3D, and while the WildCat galloping video shows that their controller can stabilize galloping very well, there are no publications which explain how it works.

4.3 Coordinate Systems and Reference Trajectories

My locomotion controller uses a number of coordinate systems to simplify modeling:

4.3.1 Global Coordinate System

This system is established by the state estimator and is a fixed inertial system. The z-axis points up and z-zero is the ground. Roll and pitch are zero when the robot is upright, and yaw-zero is the yaw orientation of the robot when it is powered on. The yaw is aligned so that the robot is facing x positive when yaw is zero.

The coordinate system is used for predictive controller's dynamics because it is the only inertial coordinate system. The state estimator determines the robot's position in this coordinate system directly. Mini Cheetah and Cheetah 3 both use the state estimator developed by Patrick Wensing [4]. This estimator combines the IMU data with leg kinematics to determine the robot's position and velocity.

4.3.2 Body Coordinate System

The body coordinate system's origin is at the geometric center of the Mini Cheetah's body. This is very close to the center of mass of the Mini Cheetah. The orientation of the body coordinate system matches the orientation of the Mini Cheetah body.

This coordinate frame is used for dynamics calculations for feed forward in swing leg control.

4.3.3 Yaw Coordinate System

The body coordinate system's x and y zero is at the geometric center of the Mini Cheetah's body. The z zero is the ground. This is very close to the center of mass of the Mini Cheetah. The yaw of the yaw coordinate system matches the orientation of the Mini Cheetah body, but roll and pitch match the orientation of the global coordinate system.

This coordinate system is used to plan out footsteps and future robot trajectory, as it is always rotated so the Mini Cheetah is pointing forward, and has the ground at $z = 0$.

4.3.4 Hip Coordinate System

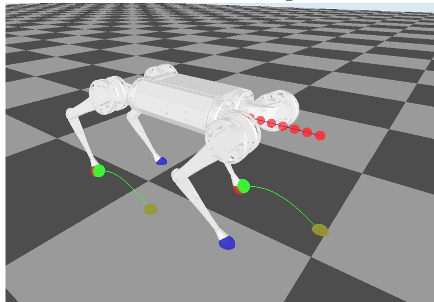
The Hip Coordinate System is like the Body Coordinate System, but shifted so the origin is at one of the hips of the Mini Cheetah. The “hip" is that point at which the leg attaches to the body.

This coordinate system is used to issue commands to the legs at a low level. The legs do not know the position or orientation of the robot, or where they are mounted on the robot, so they must receive a command relative to the point at which the leg is mounted to the robot.

4.3.5 Reference Trajectory

My controller's goal is to track a reference trajectory over a predetermined reference trajectory. The desired robot behavior is used to construct the reference trajectory, as shown in Figure 4.3.5. In my application, the reference trajectories are simple and only contain non-zero xy-velocity, xy-position, z position, yaw, and yaw rate. All parameters are commanded directly by the robot operator except for yaw and xy-position, which are determined by integrating the appropriate velocities. The other states (roll, pitch, roll rate, pitch rate, and z-velocity) are always set to 0. It is recomputed on every iteration of the controller, and is in the global frame. There are sanity checks in the reference trajectory. Because the reference trajectory is in the global coordinate system, if the robot is pushed away from its initial location and there is no control input from the operator, it will attempt to go back to where it started. However, if the robot is pushed very far away from the origin, the reference trajectory may contain an impossibly large velocity on the path back. To prevent this from happening, I detect excessively large velocities and adjust the reference trajectory so that it is always reasonable for the robot. As a result, I can tie a rope to the robot and drag it many meters, and the reference trajectory will move the robot toward the origin at a reasonable velocity.

Figure 4-2: Simulated Mini Cheetah with a reference trajectory generated for a forward velocity command. The red dots are samples on the reference trajectory



4.4 Predictive Control

Predictive control approaches are suitable for this type of control problem because they can anticipate the periods of flight or underactuation and use the reference trajectory. A predictive controller will consider the dynamics over a finite time horizon, then determine the optimal control inputs. By considering a time-varying dynamics model that accounts for the changing contact states over the horizon, a predictive controller can plan a trajectory for the robot to take.

Model predictive controllers are typically implemented by building and solving an optimization problem at each time-step. The optimization finds the state and control input trajectories which minimize tracking error, subject to dynamics and control input constraints.

$$\min_{\mathbf{x},\mathbf{u}} l(\mathbf{x}) \tag{4.1}$$

$$\text{subject to} \quad \mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \tag{4.2}$$

$$\mathbf{g}(\mathbf{u}_i) \leq \mathbf{0} \tag{4.3}$$

where the function $l(\mathbf{x})$ describes the tracking error, the function $\mathbf{f}(\mathbf{x}_i, \mathbf{u}_i)$ describes the the dynamics of the system, and the function $\mathbf{g}(\mathbf{u}_i) \leq \mathbf{0}$ describes the limits of the control inputs, including effects like maximum torques, velocities, and friction limits.

Model predictive control (MPC) is typically computationally intensive, so this

controller was designed to simplify the model and problem as much as possible. The Mini Cheetah platform has a much slower computer than most quadruped robots have. It uses the UP Board [10], which is roughly 3x slower at running my controller than the computer in Cheetah 3 [4]. Cheetah 3's computer is based on an 8 year old high end laptop CPU from 2011. The following subsections will describe how I formulated this problem to be computationally feasible.

4.4.1 Separate Swing Leg Control

The first simplification I made to the controller was to use a separate controller for legs which are not touching the ground. Swinging the legs forward to take the next step has a very small impact on the dynamics of the robot, as the mass of the legs is very small compared to the body. The legs weigh approximately 10% of the total mass, for both Cheetah 3 and Mini Cheetah. This means that the predictive controller does not need to consider the movement of swing legs while it is controlling the motion of the body.

To determine where the legs should swing, I used a heuristic inspired by simple geometry and controllers presented in [21]. I attempt to keep the time-averaged contact location of feet to be directly underneath the center of mass, which is the center of Mini Cheetah. To do this, each foot steps so that its time averaged position will be directly underneath the hip. When the robot is traveling forward at velocity $\mathbf{v}$, the foot swill be placed at

$$\mathbf{p} = \mathbf{p}_{\text{hip}} + \mathbf{v}\frac{t_{\text{stance}}}{2} \tag{4.4}$$

to compensate, where $\mathbf{p}_{\text{hip}}$ is the hip position, and t_{stance} is the amount of time the foot will be on the ground for.

A similar technique is used to take into account the yaw rate of the robot. When the robot yaws at a rate of ω, the foot placement is additionally modified by

$$\Delta\mathbf{p} = \frac{t_{\text{stance}}}{2}||\mathbf{p}_{\text{hip}}|| \begin{bmatrix}\cos(\omega) & \sin(\omega)\end{bmatrix} \tag{4.5}$$

Finally, to help balance the robot when disturbed, the foot position is modified based on the current velocity tracking error:

$$\Delta \mathbf{p} = k \mathbf{v}_{\text{error}} \tag{4.6}$$

If the robot is pushed in a direction, it will step toward that direction. This is based on the capture point concept [19], which adjusts the foot placement to allow the robot to change velocity without having the body orientation change. This approach is commonly used in humanoid robots. I set $k = 0.03$ experimentally.

The swing leg controller was originally developed by Patrick Wensing with MATLAB, and I ported it to C++. It plans a trajectory using a cubic Bezier spline in the global coordinate system, then uses position feedback in the hip coordinate system in the form of

$$\boldsymbol{\tau} = \mathbf{J}^\top (\mathbf{K}_p(\mathbf{p}_{\text{des}} - \mathbf{p}) + \mathbf{K}_d(\mathbf{v}_{\text{des}} - \mathbf{v})) \tag{4.7}$$

where $\boldsymbol{\tau}$ is the joint torque, $\mathbf{J}$ is the foot contact Jacobian in the hip coordinate system, $\mathbf{K}_p$ is a diagonal gain matrix, $\mathbf{p}_{\text{des}}$ and $\mathbf{p}$ are the desired and actual foot positions in the hip frame, and $\mathbf{v}_{\text{des}}$ and $\mathbf{v}$ are the desired and actual foot velocities in the hip frame.

This controller is described in more detail in [4].

4.4.2 Single Rigid Body "Space Potato" Model

The next simplification is to reduce the robot to a single rigid body, like a potato floating in space. This model exists in the global coordinate frame. The mass and inertia lump include the mass of the motors and legs.

The legs which are in contact with the ground are modeled as "force thrusters", located at the point of contact. The controller will attempt to determine what forces these "thrusters" should produce, and an additional controller will compute the joint

torques required to achieve these forces.

The states are the position $\mathbf{p}$, linear velocity $\mathbf{v}$, orientation $\mathbf{R}$, and angular velocity $\boldsymbol{\omega}$ of the single body, and the control inputs are the x, y, and z forces applied by each thruster, as well as the position of each thruster. All quantities are in the global coordinate system.

In my implementation, I used Jacobian transpose force control, and set $\boldsymbol{\tau} = \mathbf{J}^\top f$. This force control occurs in the hip frame, so the forces must be transformed first.

This approximation is reasonable because the legs are light (10% of the total mass), and because the robot is mechanically designed so that Jacobian transpose force control will be accurate and high bandwidth.

With this formulation of ground contact forces, it is challenging to describe the force limits, as they depend on both the configuration of the leg, and the velocity of all joints in the leg. I take a conservative approach and limit the maximum z-force of the leg to 150 Newtons, which can always be achieved in a running posture. It may be possible to produce more force in certain configurations, but this controller will never take advantage of this.

On the other hand, because the force is expressed in the global coordinate system, it is easy to add a no-slip constraint:

$$\sqrt{f_x^2 + f_y^2} \leq f_z \mu \tag{4.8}$$

4.4.3 Convex Optimization

From this point, the model is modified to allow the optimization problem to be expressed as a convex quadratic program. This form of optimization can be solved to a global minimum reliably, using third-party solvers. The cost function must be quadratic, and the equality and inequality constraints much be linear. A quadratic program can be written as

$$\min_{\mathbf{x}} \mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{g}^\top \mathbf{x} \tag{4.9}$$

$$\text{subject to} \qquad \mathbf{A}\mathbf{x} \leq \mathbf{0} \tag{4.10}$$

In order to get the problem into this form, I need to be able to write the dynamics in the linear form:

$$\mathbf{x}[i+1] = \mathbf{A}[i]\mathbf{x}[i] + \mathbf{B}[i]\mathbf{u}[i] + \mathbf{b}[i] \tag{4.11}$$

Where $\mathbf{x}[i]$ is the state at time step i, and $\mathbf{u}[i]$ is the command at time step i. Note that it is possible for $\mathbf{A}$ and $\mathbf{B}$ to be time varying, but they cannot depend on $\mathbf{u}$ or $\mathbf{x}$.

Position and Velocity Dynamics

First, we will look at the position dynamics:

$$\dot{\mathbf{p}} = \mathbf{v} \tag{4.12}$$

$$\dot{\mathbf{v}} = \frac{\sum_{i=1}^{n} \mathbf{f}_i}{m} - \mathbf{g} \tag{4.13}$$

where n is the number of feet on the ground, $\mathbf{f}_i$ is the force applied by each foot, m is the mass of the robot, and $\mathbf{g}$ is the acceleration of gravity. These dynamics are already linear, and can be included in (4.11) with no change.

Orientation Dynamics

Next is the orientation dynamics, which are nonlinear:

$$\dot{\mathbf{R}} = [\boldsymbol{\omega}]_\times \mathbf{R} \tag{4.14}$$

where $[\boldsymbol{\omega}]_\times$ is the 3×3 skew-symmetric matrix which satisfies $[\boldsymbol{\omega}]_\times \mathbf{x} = \boldsymbol{\omega} \times \mathbf{x}$.

To approximate orientation dynamics, I switched from a rotation matrix $\mathbf{R}$ to Euler angles $[\phi\theta\psi]$ for roll, pitch and yaw respectively. Additionally, I introduced an additional variable ψ_r, which is the reference yaw determined from the desired robot trajectory. These angles satisfy

$$\mathbf{R} = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\phi) \tag{4.15}$$

where $\mathbf{R}_n(\alpha)$ represents a positive rotation of α around the n-axis.

Note that this convention has a singularity at a pitch angle of 90 degrees, corresponding to the robot pointing straight up. We will use a small angle approximation for roll and pitch, as the robot's body does not deviate significantly from level during normal locomotion, so the singularity can safely be ignored. Note that we do not use any small angle approximations for yaw, as they robot must be able to rotate to any yaw angle. This order of Euler angles will not encounter a singularity under these conditions. To determine the orientation of the robot from these angles, first yaw the robot, then pitch, along the *robot's* y axis, then roll the robot along the *robot's* x axis.

To determine the angular rate from the Euler angle rates, consider the contribution of the roll, pitch, and yaw rates separately:

$$\boldsymbol{\omega} = \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + \mathbf{R}_z \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \mathbf{R}_z \mathbf{R}_y \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{4.16}$$

$$= \begin{bmatrix} \cos(\theta)\cos(\psi) & -\sin(\psi) & 0 \\ \cos(\theta)\cos(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.17}$$

This can be inverted if the robot is not pointed vertically ($\cos(\theta) \neq 0$):

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \cos(\psi)/\cos(\theta) & \sin(\psi)/\cos(\theta) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ \cos(\psi)\tan(\theta) & \sin(\psi)\tan(\theta) & 1 \end{bmatrix} \boldsymbol{\omega} \tag{4.18}$$

From here, we can make an approximation assuming roll and pitch are small (ψ and θ), and replace yaw with reference yaw to make this linear:

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \approx \begin{bmatrix} \cos(\psi_r) & \sin(\psi_r) & 0 \\ -\sin(\psi_r) & \cos(\psi_r) & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{\omega} \tag{4.19}$$

which is equivalent to

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \approx \mathbf{R}_z(\psi_r)\boldsymbol{\omega} \tag{4.20}$$

With this approximation, the orientation dynamics can be written in a linear time-varying form.

Note that the actual yaw angle of the robot must closely match the desired yaw angle for this to be an accurate model. As we will see later, yaw tracking works well,

so this is a reasonable approximation.

Angular Dynamics

Finally we simplify the angular dynamics:

$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{I}\boldsymbol{\omega}) = \sum_{i=1}^{n} \mathbf{r}_i \times \mathbf{f}_i \tag{4.21}$$

The first approximation we make is the the inertia. This equation is expressed in the global frame, so the inertia of the body must be transformed:

$$\mathbf{I} = \mathbf{R}\mathbf{I}_{\text{body}}\mathbf{R}^\top \tag{4.22}$$

where $\mathbf{I}_{\text{body}}$ is the constant inertia of the single rigid body model of the robot. I use a similar approximation and assume roll and pitch are small, and yaw will closely track the desired yaw:

$$\mathbf{I} \approx \mathbf{R}_z(\psi_r)\mathbf{I}_{\text{body}}\mathbf{R}_z(\psi_r)^\top \tag{4.23}$$

The next approximation we make is with the left hand side:

$$\frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{I}\boldsymbol{\omega}) = \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) \tag{4.24}$$

$$\approx \mathbf{I}\dot{\boldsymbol{\omega}} \tag{4.25}$$

The eliminated $\boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega})$ is likely very small - it corresponds to the gyroscope precession and nutation forces acting on the body, which are not significant. These forces are proportional to the product of angular velocities along different principal axes. While it is possible to experience large yaw velocities, the pitch and roll velocities are typically not very large, so this approximation is reasonable.

The final approximation we make is to the right hand side. The cross product of $\mathbf{r}$

and $\mathbf{f}$ involves the product of two state variables, so it must be approximated. Because the reference trajectory of the robot is known and we already have a controller for determining foot step location, I can determine the approximate location of the feet at each time-step along the trajectory. Therefore, this can be replaced with:

$$\sum_{i=1}^{n} \mathbf{r}_i \times \mathbf{f}_i \approx \sum_{i=1}^{n} \mathbf{r}_{\text{ref},i} \times \mathbf{f}_i \tag{4.26}$$

Combined Dynamics

The approximated orientation dynamics and translational dynamics can be combined into the following form:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\hat{\mathbf{\Theta}}\\ \hat{\mathbf{p}}\\ \hat{\boldsymbol{\omega}}\\ \hat{\dot{\mathbf{p}}}\end{bmatrix} = \begin{bmatrix}\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{R}_z(\psi) & \mathbf{0}_3\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{1}_3\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3\end{bmatrix}\begin{bmatrix}\hat{\mathbf{\Theta}}\\ \hat{\mathbf{p}}\\ \hat{\boldsymbol{\omega}}\\ \hat{\dot{\mathbf{p}}}\end{bmatrix} + \begin{bmatrix}\mathbf{0}_3 & \dots & \mathbf{0}_3\\ \mathbf{0}_3 & \dots & \mathbf{0}_3\\ \hat{\mathbf{I}}^{-1}[\mathbf{r}_1]_\times & \dots & \hat{\mathbf{I}}^{-1}[\mathbf{r}_n]_\times\\ \mathbf{1}_3/m & \dots & \mathbf{1}_3/m\end{bmatrix}\begin{bmatrix}\mathbf{f}_1\\ \vdots\\ \mathbf{f}_n\end{bmatrix} + \begin{bmatrix}0\\ 0\\ 0\\ \mathbf{g}\end{bmatrix} \tag{4.27}$$

Equation (4.27) can be rewritten with an additional gravity state to put the dynamics into the familiar state-space form:

$$\dot{\mathbf{x}}(t) = \mathbf{A}_c(\psi)\mathbf{x}(t) + \mathbf{B}_c(\mathbf{r}_1, ..., \mathbf{r}_n, \psi)\mathbf{u}(t) \tag{4.28}$$

where $\mathbf{A}_c \in \mathbb{R}^{13\times 13}$ and $\mathbf{B}_c \in \mathbb{R}^{13\times 3n}$. With these simplifications, $\mathbf{A}$ and $\mathbf{B}$ matrices depend only on yaw and footstep locations. Because these can be computed ahead of time, the dynamics become linear time-varying, which is suitable for expressing the problem as a quadratic program.

Discrete Time Dynamics

For each point n in the reference trajectory, an approximate $\hat{\mathbf{B}}_c[n] \in \mathbb{R}^{13\times 3n}$ and $\hat{\mathbf{A}}_c[n] \in \mathbb{R}^{13\times 13}$ matrix are computed from the $\mathbf{B}_c(\mathbf{r}_1, ..., \mathbf{r}_n, \psi)$ and $\mathbf{A}_c(\psi)$ matrix defined in (4.28) using desired values of ψ and $\mathbf{r}_i$ from the reference trajectory and foot placement controller. These matrices are converted to a zero order hold discrete time model using the state transition matrix of the extended linear system:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\mathbf{x}\\ \mathbf{u}\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \mathbf{u}\end{bmatrix} \tag{4.29}$$

The dynamics can now be written as a discrete linear time varying system

$$\mathbf{x}[n+1] = \hat{\mathbf{A}}\mathbf{x}[n] + \hat{\mathbf{B}}[n]\mathbf{u}[n] \tag{4.30}$$

The approximation in (4.30) is only accurate if the robot is able to follow the reference trajectory. Large deviations from the reference trajectory will result in $\hat{\mathbf{B}}[n]$ being inaccurate. However, for the first time step, $\hat{\mathbf{B}}[n]$ is calculated from the current robot state, and will always be correct. If, at any point, the robot is disturbed from following the reference trajectory, the next iteration of the MPC, which happens at most 40 ms after the disturbance, will recompute the reference trajectory based on the disturbed robot state, allowing it compensate for a disturbance.

4.5 MPC Formulation

Desired ground reaction forces are found with a discrete-time finite-horizon model predictive controller. Because we consider ground reaction forces instead of joint torques, the predictive controller does not need to be aware of the configuration or

kinematics of the leg. The MPC optimization can be written as

$$\min_{\mathbf{x},\mathbf{u}} \quad \sum_{i=0}^{k-1} ||\mathbf{x}_{i+1} - \mathbf{x}_{i+1,\mathrm{ref}}||_{\mathbf{Q}_i} + ||\mathbf{u}_i||_{\mathbf{R}_i} \tag{4.31}$$

$$\text{subject to} \quad \mathbf{x}_{i+1} = \mathbf{A}_i\mathbf{x}_i + \mathbf{B}_i\mathbf{u}_i, i = 0 \dots k-1 \tag{4.32}$$

$$\underline{\mathbf{c}}_i \leq \mathbf{C}_i\mathbf{u}_i \leq \overline{\mathbf{c}}_i, i = 0 \dots k-1 \tag{4.33}$$

$$\mathbf{D}_i\mathbf{u}_i = 0, i = 0 \dots k-1 \tag{4.34}$$

where $\mathbf{x}_i$ is the system state at time step i, $\mathbf{u}_i$ is the control input at time step i, $\mathbf{Q}_i$ and $\mathbf{R}_i$ are diagonal positive semi-definite matrices of weights, $\mathbf{A}_i$ and $\mathbf{B}_i$ represent the discrete time system dynamics, $\mathbf{C}_i, \underline{\mathbf{c}}_i$, and $\overline{\mathbf{c}}_i$ represent inequality constraints on the control input, and $\mathbf{D}_i$ is a matrix which selects forces corresponding with feet not in contact with the ground at time-step i. The notation $||\mathbf{a}||_\mathbf{S}$ is used to indicate the weighted norm $\mathbf{a}^\top\mathbf{S}\mathbf{a}$. The controller in this form attempts to find a sequence of control inputs that will guide the system along the trajectory $\mathbf{x}_{\mathrm{ref}}$, trading off tracking accuracy for control effort, while obeying constraints. In the event that the system cannot exactly track the reference trajectory, which is often the case due to uncontrollable dynamics during periods of underactuation, or due to constraints, the predictive controller finds the best solution, in the least squares sense, over the prediction horizon. The optimization over the horizon while taking into account future constraints enables the predictive controller to plan ahead for periods of flight and regulate the states of the body during a gait when the body is always underactuated, such as bounding.

Force Constraints

The equality constraint in (4.34) is used to set all forces from feet off the ground to zero, enforcing the desired gait. The inequality constraint in (4.33) is used to set the

following 10 inequality constraints for each foot on the ground

$$f_{\min} \leq f_z \leq f_{\max} \tag{4.35}$$

$$-\mu f_z \leq \pm f_x \leq \mu f_z \tag{4.36}$$

$$-\mu f_z \leq \pm f_y \leq \mu f_z \tag{4.37}$$

These constraints limit the minimum and maximum z-force as well as a square pyramid approximation of the friction cone.

4.5.1 QP Formulation

The optimization problem in (4.31) is reformulated to reduce problem size. I found that reducing the problem size as much as possible and using a dense solver was many times faster than using a sparse solver. With a dense solver, it is important to reduce the number of variables, so the states are removed from optimization variables, and the dynamics are included in the cost function. This formulation also allows us to eliminate trivial optimization variables that are constrained to zero by (4.34) so that we are only optimizing forces for feet which are on the ground.

The condensed formulation allows the dynamics to be written as

$$\mathbf{X} = \mathbf{A}_{\text{qp}}\mathbf{x}_0 + \mathbf{B}_{\text{qp}}\mathbf{U} \tag{4.38}$$

where $\mathbf{X} \in \mathbb{R}^{13k}$ is the vector of all states during the prediction horizon and $\mathbf{U} \in \mathbb{R}^{3nk}$ is the vector of all control inputs during the prediction horizon.

Given a starting state $\mathbf{x}_0 = \mathbf{x}[n_0]$ and sequence of control inputs $\mathbf{u}[n_0] \dots \mathbf{u}[n_0 + k - 1]$, the $n_0 + k$th state can be found with

$$\mathbf{x}[n_0 + k] = \mathbf{D}_k\mathbf{x}[n_0] + \mathbf{D}_{k-1}\mathbf{B}_0\mathbf{u}[n_0] + \mathbf{D}_{k-2}\mathbf{B}_1[n_0 + 1] + \cdots + \mathbf{B}_{k-1}\mathbf{u}[n_0 + k - 1] \tag{4.39}$$

where $\mathbf{D}_k = \mathbf{A}_0 \dots \mathbf{A}_{k-1}$.

This can be used to find $\mathbf{A}_{\text{qp}}$ and $\mathbf{B}_{\text{qp}}$:

$$
\begin{bmatrix} \mathbf{x}[n_0+1] \\ \mathbf{x}[n_0+2] \\ \vdots \\ \mathbf{x}[n_0+k] \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_k \end{bmatrix}}_{\mathbf{A}_{\text{qp}}} \mathbf{x}_0 + \underbrace{\begin{bmatrix} \mathbf{B}_0 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{D}_1\mathbf{B}_0 & \mathbf{B}_1 & \dots & \mathbf{B}0 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}_k\mathbf{B}_0 & \mathbf{D}_{k-1}\mathbf{B}_1 & \dots & \mathbf{B}_{k-1} \end{bmatrix}}_{\mathbf{B}_{\text{qp}}} \begin{bmatrix} \mathbf{u}[n_0] \\ \mathbf{u}[n_0+1] \\ \vdots \\ \mathbf{u}[n_0+k-1] \end{bmatrix} \tag{4.40}
$$

The objective function which minimizes the weighted least-squares deviation from the reference trajectory and the weighted force magnitude is:

$$
J(\mathbf{U}) = ||\mathbf{A}_{\text{qp}}\mathbf{x}_0 + \mathbf{B}_{\text{qp}}\mathbf{U} - \mathbf{x}_{\text{ref}}||_{\mathbf{L}} + ||\mathbf{U}||_{\mathbf{K}} \tag{4.41}
$$

where $\mathbf{L} \in \mathbb{R}^{13k \times 13k}$ is a diagonal matrix of weights for state deviations, $\mathbf{K} \in \mathbb{R}^{3nk \times 3nk}$ is a diagonal matrix of weights for force magnitude, and $\mathbf{U}$ and $\mathbf{X}$ are the vectors of all control inputs and states during the prediction horizon. On the robot, we chose an equal weighting of forces $\mathbf{K} = \alpha \mathbf{1}_{3nk}$. The problem can be rewritten as:

$$
\min_{\mathbf{U}} \quad \frac{1}{2}\mathbf{U}^\top \mathbf{H} \mathbf{U} + \mathbf{U}^\top \mathbf{g} \tag{4.42}
$$

$$
\text{s. t.} \quad \underline{\mathbf{c}} \leq \mathbf{C}\mathbf{U} \leq \overline{\mathbf{c}} \tag{4.43}
$$

where $\mathbf{C}$ is the constraint matrix and

$$
\mathbf{H} = 2(\mathbf{B}_{\text{qp}}^\top \mathbf{L} \mathbf{B}_{\text{qp}} + \mathbf{K}) \tag{4.44}
$$

$$
\mathbf{g} = 2\mathbf{B}_{\text{qp}}^\top \mathbf{L}(\mathbf{A}_{\text{qp}}\mathbf{x}_0 - \mathbf{y}) \tag{4.45}
$$

The desired ground reaction forces are then the first $3n$ elements of $\mathbf{U}$. Notice that $\mathbf{H} \in \mathbb{R}^{3nk \times 3nk}$ and $\mathbf{g} \in \mathbb{R}^{3nk \times 1}$ both have no size dependence on the number of states, only number of feet n and horizon length k.

4.6 Results on Mini Cheetah

The controller was used on the Mini Cheetah to do trotting, trot-running, pronking, and bounding. With a trotting gait, the robot was able to achieve a maximum forward velocity of 2.45 m/s, a maximum lateral velocity of 1 m/s, and a maximum turning rate of 4.6 rad/s. These speeds are significantly faster than any other robot of similar size, and are faster than most larger quadrupeds. The only robots which are faster are the much larger MIT Cheetah 3 [4], which reached 3.25 m/s; MIT Cheetah 2 [16], which reached 6.4 m/s; and the Boston Dynamics WildCat robot, which reached approximately 9 m/s.

A video showing locomotion with this controller can be found in [14]. Data from the state estimator is shown in Figure 4-3 and 4-4. Both of these test were done with the robot completely untethered in the hallway.

Figure 4-3: Data from state estimator during high speed trotting on Mini Cheetah

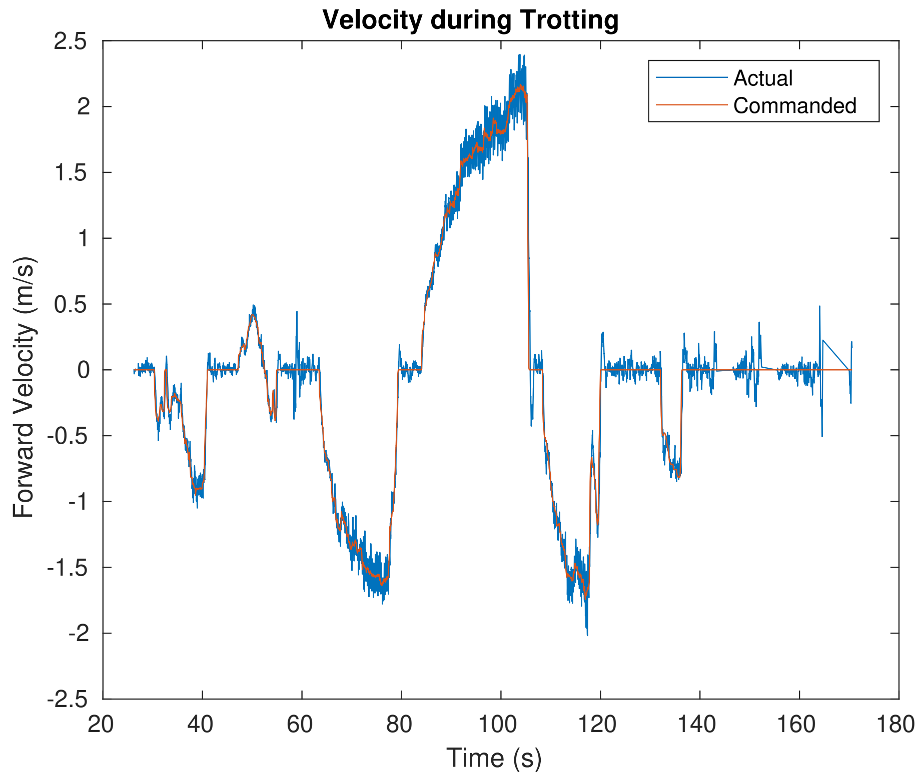
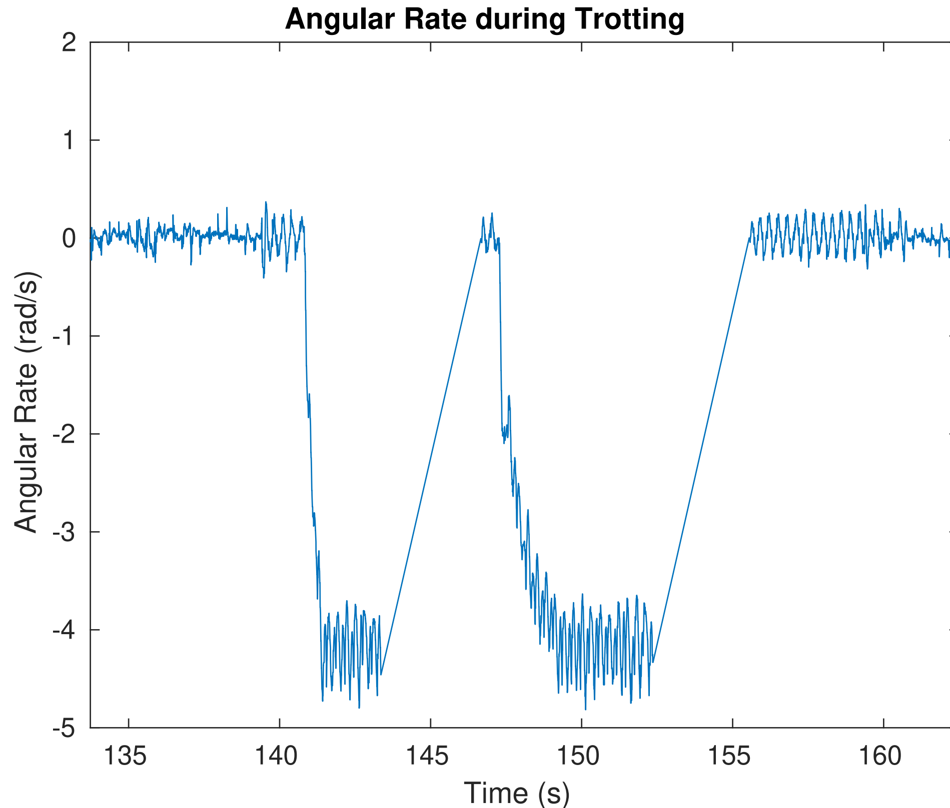


Figure 4-4: Data from state estimator during high speed in place turning on Mini Cheetah



4.7 Results on Cheetah 3

This controller was also tested on the MIT Cheetah 3 robot. Using a galloping gait, the robot was able to reach speeds of 3.0 m/s. Figure 4-5 shows several frames of the robot galloping. In Figures 4-6, 4-7 and 4-8, the MPC result, orientation, torques, and velocities are shown.

More gaits can be seen in `https://www.youtube.com/watch?v=q6zxCvCxhic`.

Additionally, the robot was able to blindly climb stairs covered in debris while trotting with this controller. The stairs shown in Figure 4-9 was climbed in under 6 seconds.

Figure 4-5: Several frames of the MIT Cheetah 3 Robot galloping at 2.5 m/s

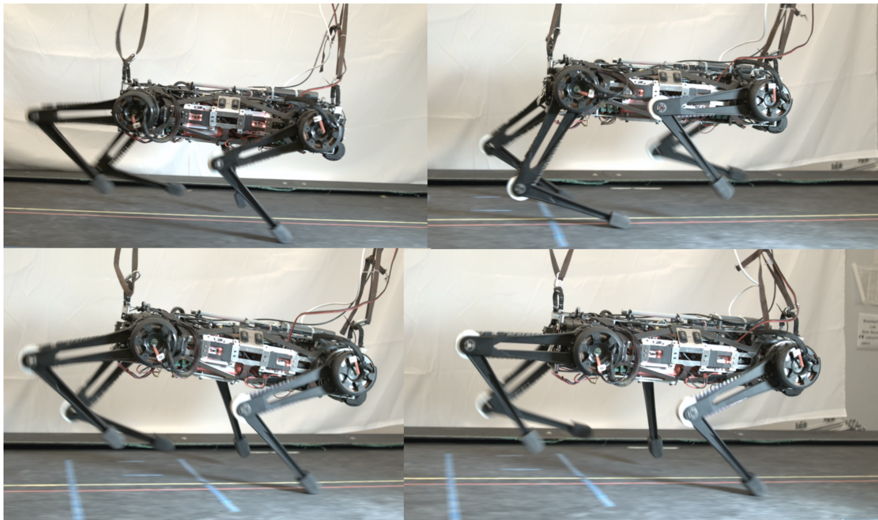


Figure 4-6: MPC result and orientation

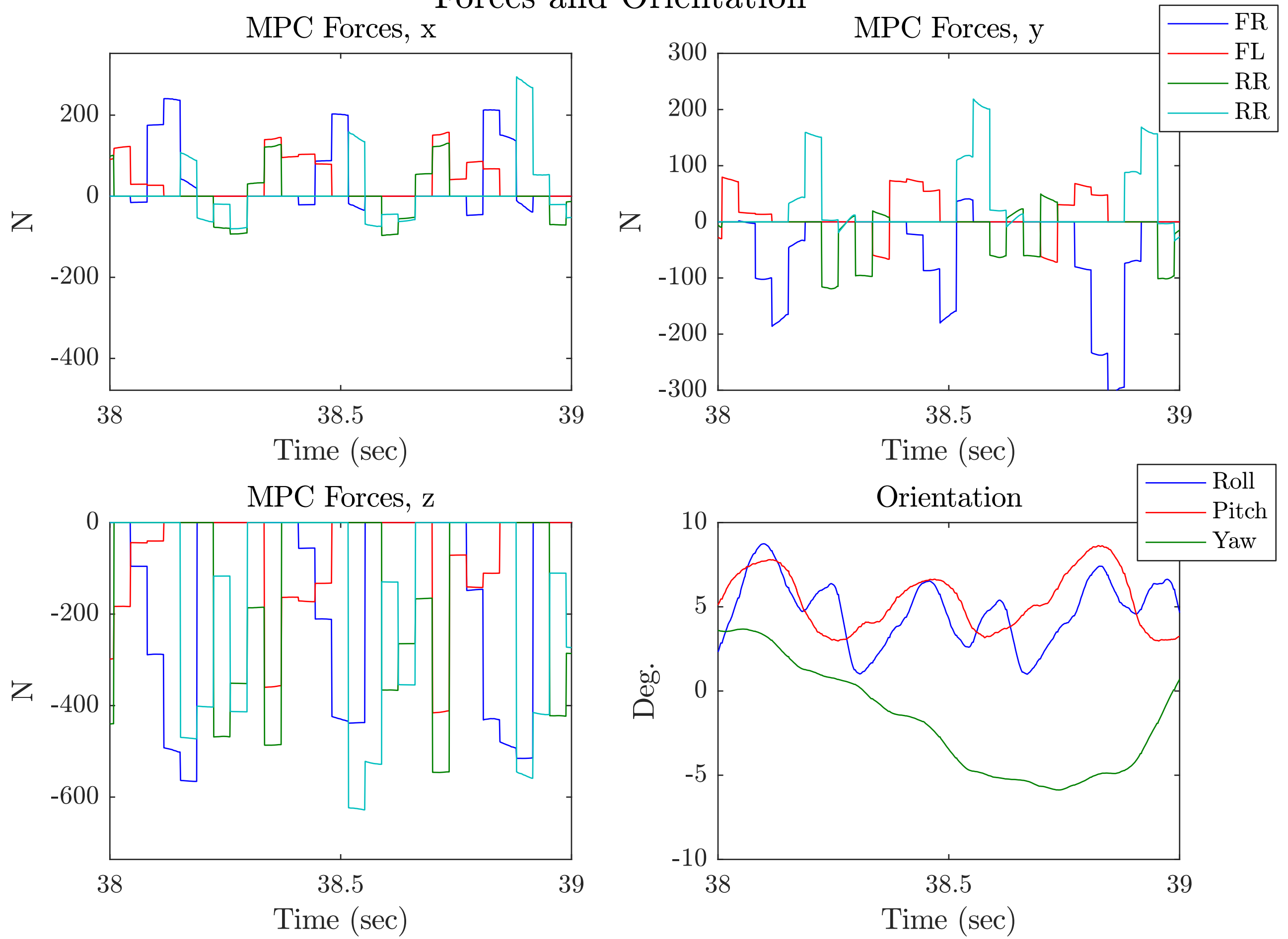


Figure 4-7: Leg data from galloping

Leg Data

Knee Torques

Hip Torques

FR
FL
RR
RR

Nm

200
0
-200

38 38.2 38.4 38.6

Time (sec)

Knee Velocity

Hip Velocity

rad/sec

10
0
-10

Knee Position

Hip Position

degrees

150
100
50

100
0
-100

Figure 4-8: Velocity control of galloping to 3 m/s

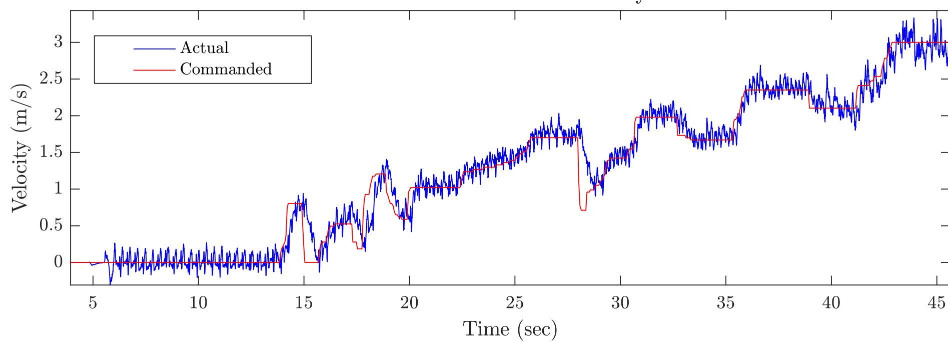
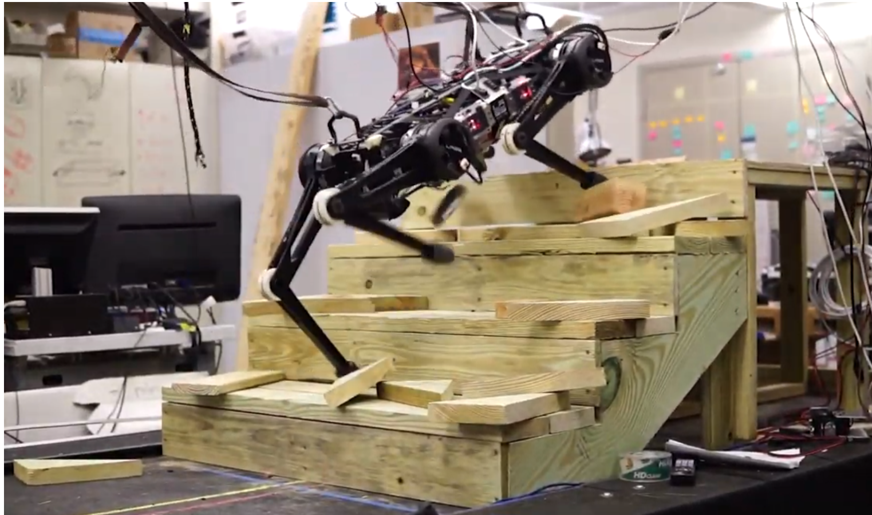


Figure 4-9: Cheetah 3 climbing stairs while trotting



Chapter 5

Backflip

5.1 Related Work

I was inspired to try a flip on Mini Cheetah by an older result from 1992 [18], where the "3D-Biped" robot developed by the Leg Lab did a somersault from a running gait. The largest controls challenge of the somersault is to have the robot land at the correct angle, as the biped walking controller was very sensitive to disturbances in pitch. This robot used a simple heuristic controller which adjusted the tucking of the legs to modify the rotation rate so that it would land at the correct angle.

Since 1992, the only other legged robot to do a flip is the Boston Dynamics Atlas biped robot, but there is no published information about its controller.

Prior to the Mini Cheetah, there have been no quadruped robots which have done flips. However, after Mini Cheetah's backflip, the company Unitree Robotics has recently done a backflip with their AlienGo quadruped [22].

5.2 Approach

I took the approach of planning the backflip trajectory ahead of time, then playing it back on the robot. I used a highly accurate model for the planned trajectory. There are two main reasons I took this approach.

First, with a quadruped like Mini Cheetah, it is hard to use the legs to effect

the rotation rate, as they do not have a significant mass. When comparing two trajectories, one which held the legs still in the air, and one which pumped the legs at maximum torque, the landing angle only differed by 7 degrees. It is important that the robot takes off with an accurate pitch velocity so that it can land at the correct angle. By planning the trajectory ahead of time, I can use a highly accurate, but computationally slow model which includes the full multi-body dynamics of the robot.

In order to perform a backflip, Mini Cheetah must use close to the maximum torque of its actuators. In order to design trajectories which approach this limit, I formulated the problem in terms of the joint torques and velocities. This requires using the full dynamics of the robot and actuator model, which is too slow to plan in real-time.

5.3 Model

The model I used is a planar version of the model discussed in 2.1.1 and includes the effects of rotors using the algorithms from A. This planar model removes the ab/ad degrees of freedom, the ab/ad rotors, and the ab/ad links. The mass of the ab/ad links are added to the body. The front two legs are combined into a single leg, and the rear two legs are combined into a single leg. In total, there are 5 rigid bodies, not including 4 rotors. The robot's base cannot move laterally, and cannot roll or yaw. With this model, there are seven degrees of freedom: three for the floating base, and two for each of the two legs.

The trajectory was planned using a nonlinear optimization. I used CasADi [2] to set up the nonlinear optimization problem and IPOPT [24] to solve it.

The optimization attempts to find joint and body positions $\mathbf{q}$, joint and body velocities $\dot{\mathbf{q}}$, joint and body accelerations $\ddot{\mathbf{q}}$, and ground reaction forces $\mathbf{f}_k$ at each time-step.

The total duration and contact sequence is determined ahead of time by hand. I based my timings off of a simple ballistic trajectory which puts the robot in the air

for long enough to perform a flip at the desired rotation rate. I determined that a reasonable rotation rate would be 600 degrees per second from videos of small dogs doing flips found on the internet.

The dynamics are enforced with the manipulator equation and the constraint that feet on the ground should have no acceleration:

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{b}(\mathbf{q}, \dot{\mathbf{q}}) = \sum_k \mathbf{J}_k^\top(\mathbf{q})\mathbf{f}_k + \mathbf{S}\mathbf{u} \tag{5.1}$$

$$\mathbf{J}_k\ddot{\mathbf{q}} + \dot{\mathbf{J}}_k\dot{\mathbf{q}} = \mathbf{0} \tag{5.2}$$

I did not set a cost function for the optimization, but instead enforced everything through constraints:

- Initial position and velocity of the robot crouched on the ground
- Landing position of the body and legs, with the robot level and legs extended to absorb the impact of landing
- Knees should not go under the ground
- Front foot should not get too close to the rear foot, or rear hip
- Forces on a foot should be zero when the foot is not on the ground'
- Normal force on a foot on the ground should be at least $f_{\min}$ to prevent the foot from slipping if there is a small error
- Ground reaction forces must be in a friction cone $|f_x| \leq \mu f_z$
- Maximum torque limit
- Velocity dependent torque limit
- Euler integration of manipulator equation
- Feet on the ground has zero acceleration

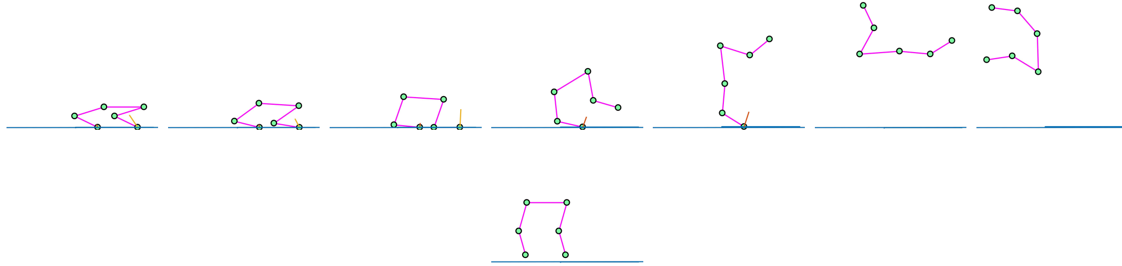


Figure 5-1: Frames from optimized trajectory

5.4 Optimization

The optimization took roughly 100 seconds to run, with most of the time spent computing the dynamics and setting up the problem with CasADi, and only 10 seconds spent in the nonlinear solver. Typically the problem had around 80 time-steps, which resulted in roughly 2000 variables and 4000 constraints. The results from an optimization are shown in Figure 5-1.

5.5 Results

To follow the trajectories, I combined feed-forward torque commands with joint position and velocity feedback. At the time of the first implementation, there was no IMU on the robot, so I did not implement any orientation feedback. The robot was able to track the joint positions quite well, which is shown in Figure 5-2. I believe some of the tracking error is caused because some joints reach their maximum torque, which is shown in Figure 5-2. There are likely small inaccuracies in the mass properties of rigid-body model, and unmodeled friction and other losses.

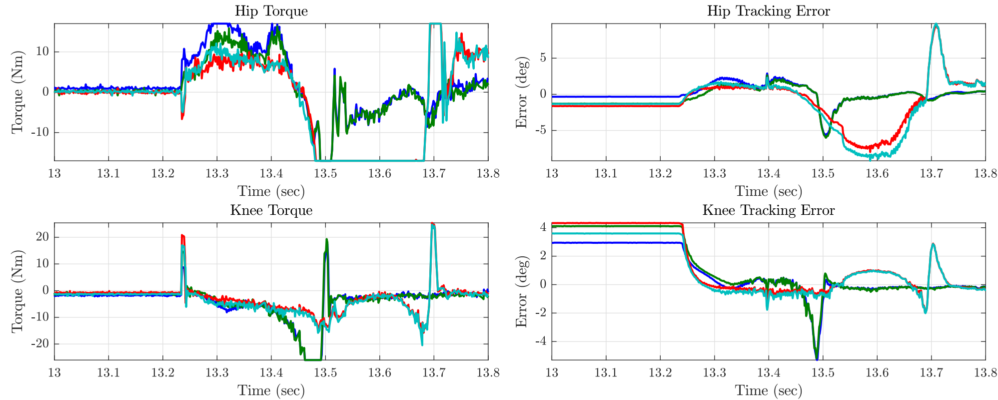
With the original implementation, Mini Cheetah was able to land the backflip 2 out of 3 times. Most falls were because the robot landed with a small amount of roll, and the robot bounced and tipped over. After decreasing the stiffness of the landing and tuning the position of the legs during landing, the reliability of the backflip increased dramatically.

In testing, a single robot was able to perform 23 consecutive backflips without falling. The fall was caused because the robot landed with too much roll. When

reviewing video footage of testing Mini Cheetahs outside, there were 35 consecutive flips without a fall, not counting a flip where one robot flipped, landed on top of a second robot, and was emergency-stopped.

A video of the flip can be seen in [14].

Figure 5-2: Joint torque and position tracking error during a backflip



5.6 Extensions

In simulation, I have repeated this process for 3D behaviors, like jumping while yawing, but they have not been tested in hardware yet.

A similar flip was developed for Cheetah 3 and was verified in the full simulator (Figure 5-4), but was not tested on the robot.

Another member of the lab modified the Cheetah 3 flip optimization program to design trajectories for jumping onto a box [13]. He simply changed the initial and final state constraints and all of the dynamics could be reused. This was used to jump onto a 76cm tall table, as shown in Figure 5-5.

Figure 5-3: Mini Cheetah doing a backflip outside

Figure 5-4: Cheetah 3 Simulated Backflip

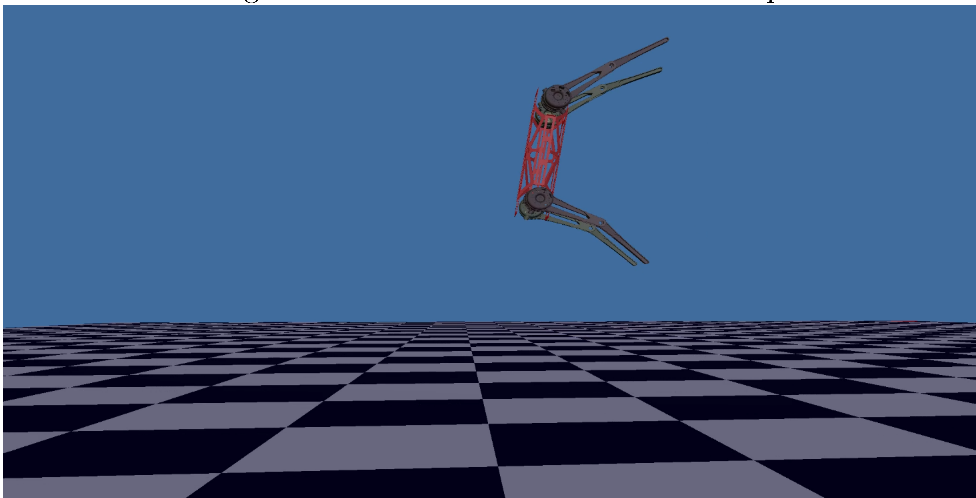
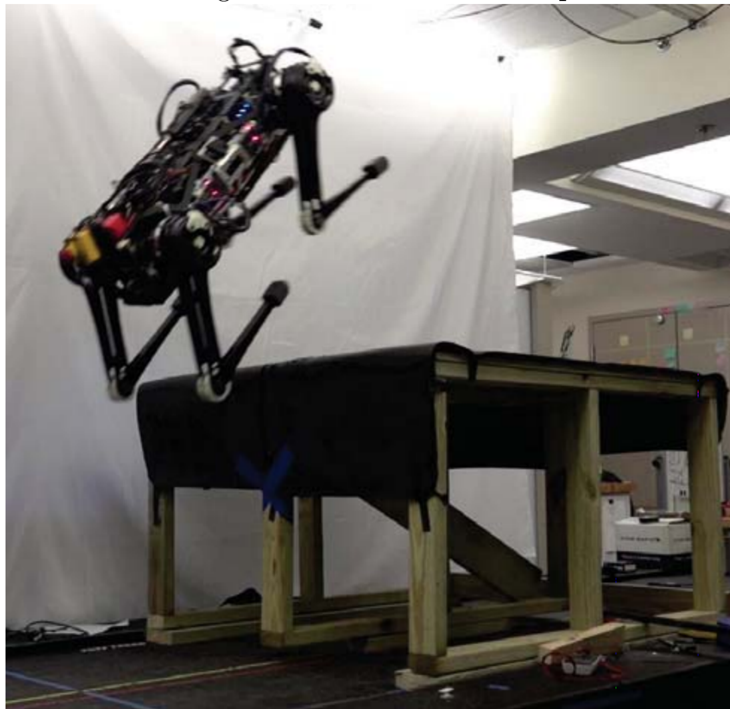


Figure 5-5: Cheetah 3 Jump



Appendix A

Dynamics with Rotors

This algorithm computes the mass matrix $\mathbf{H}$, the bias force $(\mathbf{C}\dot{\mathbf{q}} + \mathbf{g})$, contact point Jacobians $\mathbf{J}_k$, and contact point bias acceleration $\dot{\mathbf{J}}_k\dot{\mathbf{q}}$. The dynamics of the system can be written as

$$\mathbf{H} + \mathbf{C}\dot{\mathbf{q}} + \mathbf{g} = \mathbf{S}\mathbf{u} \sum_k \mathbf{J}_k^\top \mathbf{f}_k \tag{A.1}$$

where $\mathbf{S}$ is a matrix which selects actuated degrees of freedom, $\mathbf{u}$ is the torque produced by the motors, and $\mathbf{f}_k$ is the k-th contact force. Each contact point which is not slipping has zero acceleration. The contact point bias acceleration $\dot{\mathbf{J}}_k\dot{\mathbf{q}}$ is used to express this constraint as

$$\mathbf{J}_k\ddot{\mathbf{q}} + \dot{\mathbf{J}}_k\dot{\mathbf{q}} = 0 \tag{A.2}$$

A.1 System Description

The system has N_b bodies, not including rotors and N_r rotors. The bodies (not including rotors) are arranged into a tree, where each body i's parent is $\lambda(i)$. The tree of bodies has a maximum depth of d. Each rotor k is geared to $\gamma(k)$ and fixed to $\mu(k)$. For the Mini Cheetah robot, $\mu(k) = \lambda(\gamma(k))$, but this is not a requirement

for this algorithm. Each contact point j is attached to body $\sigma(j)$.

Each body's mass properties is described with a 6×6 spatial inertia matrix $\mathbf{I}$. This can be determined from a rigid body's center of mass, mass, and rotational inertia around its center of mass with the following

$$\mathbf{I} = \begin{bmatrix} \mathbf{I}_c + m(\mathbf{c}\times)(\mathbf{c}\times)^\top & m(\mathbf{c}\times) \\ m(\mathbf{c}\times)^\top & m\mathbf{1} \end{bmatrix} \tag{A.3}$$

where $\mathbf{I}_c$ is the rotational inertia around the center of mass, m is the mass, and $(\mathbf{c}\times)$ is the skew symmetric matrix based on the center of mass defined by

$$\mathbf{c}\times = \begin{bmatrix} 0 & -c_z & c_y \\ c_z & 0 & -c_x \\ -c_y & c_x & 0 \end{bmatrix} \tag{A.4}$$

The attachment of joints is described with the spatial motion transformation $\mathbf{X}_{T,i}$, which describes the coordinate transformation from body $\lambda(i)$ to i when $q_i = 0$. The spatial motion transformation from A to B coordinates is a 6×6 matrix which can be written as

$${}^B\mathbf{X}_A = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ -\mathbf{R}(\mathbf{r}\times) & \mathbf{R} \end{bmatrix} \tag{A.5}$$

where $\mathbf{R}$ is the rotational coordinate transformation from A to B and $\mathbf{r}$ is the translation from the origin of A to B, *in A coordinates.*

The movement of joints is described by the function `jcalc`(q_i) and ϕ_i. The function `jcalc` computes the spatial motion transformation $\mathbf{X}_{J,i}$ based on the position of the joint. For rotary joints, this is simply a rotation around the appropriate axis by q. Additionally, the projection ϕ_i from generalized coordinates (q_i) to spatial velocity in the link's frame caused by the joint velocity $(\mathbf{v}_J)$ is defined by $\mathbf{v}_J = \phi_i \dot{q}_i$. Note that I assume $\dot{\phi}_i = \mathbf{0}$ (in the link's frame), which is valid for rotary, linear, or screw

joints.

The location of ground contact points is specified by the spatial motion transformations $\mathbf{X}_{C,k}$ which describes the transformation from the body containing contact point k to the contact point.

A.2 Algorithm Derivation

The effect of rotors can be derived by first computing the dynamics of the system without rotors, then considering all effects of the rotors. This is a useful approach, as it can determine bounds on the effects of rotors, which can be used to omit negligible rotor effects to simplify dynamics.

A.2.1 Kinematics and Contact Points

Adding rotors will not change kinematics or $\dot{\mathbf{q}}$ so the contact point Jacobians $\mathbf{J}_k$ and contact point bias acceleration $\dot{\mathbf{J}}_k\dot{\mathbf{q}}$ will be unchanged. The algorithm for forward kinematics or bodies and contact points and rotors runs in $O(N_b + N_c)$ time. Computing body and contact Jacobians is done in $O(N_c + N_b d)$ time, where d is the maximum depth of any body on the kinematics tree.

A.2.2 Mass Matrix

The mass matrix $\mathbf{H}$ can be interpreted as the apparent inertia of the system. The rotors affect the apparent inertia in three main ways.

Making Links Heavier

If the rotors did not spin, they would still affect inertia by making certain links heavier. First, we compute the change in each body's inertia $\Delta\mathbf{I}_i = \sum_{\mu(k)=i} \mathbf{I}_\text{k}$ which is the net effect from all rotors attached to body i. Then, we can simply run the Composite-Rigid-Body Algorithm to compute the change in each body's composite inertia, and finally the change in mass matrix caused by links being heavier. It is

easy to determine how important this effect is by comparing the mass of the rotor to the mass of the link. Note that this effect is also independent of the gear ratio and this does not include the effect of gravity.

Reaction Torques

As the rotors accelerate, they produce a reaction torque on the link they are fixed on. If you hold a motor, you will feel a reaction torque as it accelerates due to the rotor accelerating. If the motor output acceleration is constant, the rotor acceleration and its reaction torque will scale linearly with the gear ratio. In the case of a multi-body system, this reaction torque also can affect all parents bodies. This torque does not effect the link they are geared to, so this will only affect the off-diagonal terms of the mass matrix. Intuitively, this effect should scale linearly with the gear ratio. If the link accelerates at $\ddot{q}$, the geared rotor k must experience spatial force

$$\begin{aligned} \mathbf{f} &= \mathbf{I}^{\text{rot}}\mathbf{a}^{\text{rot}} && \text{(A.6)} \\ &= \mathbf{I}^{\text{rot}}\phi^{\text{rot}}\ddot{q}^{\text{rot}} && \text{(A.7)} \\ &= \mathbf{I}^{\text{rot}}\phi^{\text{rot}}N_k\ddot{q}_i && \text{(A.8)} \end{aligned}$$

where N_k is the gear ratio. We define $\mathbf{b} = \mathbf{I}^{\text{rot}}\phi^{\text{rot}}N_k$ so that $\mathbf{b}\ddot{q} = \mathbf{f}$. The quantity $\mathbf{b}$ can be seen as an inertia term.

This force must be reacted by the other bodies in the tree. We use i as the index of the body that the rotor is geared to, and j as the body experiencing the reaction torque. The algorithm for projecting this force onto all joints is

1. Transform $\mathbf{b}$ from j to $\lambda(j)$ coordinates: $\mathbf{b} = {}^{j}\mathbf{X}_{\lambda(j)}^{\top}\mathbf{b}$
2. Update $j = \lambda(j)$
3. Update Mass Matrix $\Delta\mathbf{H}_{i,j} = \Delta\mathbf{H}_{i,j} + \phi_j^{\top}\mathbf{b}$
4. Update Mass Matrix $\Delta\mathbf{H}_{i,j} = \Delta\mathbf{H}_{j,i}$

5. Repeat until root of tree is reached

The first step uses the spatial force transform to transform $\mathbf{b}$ from the first link to parent link. The force transform is in the inverse transpose of the motion transformation which we have likely already calculated as part of forward kinematics. It is appropriate to use a force transformation as $\mathbf{b}\ddot{q} = \mathbf{f}$. The transformation can be found with

$$^{i}\mathbf{X}_{\lambda(i)} = \texttt{jcalc}(q_i)\mathbf{X}_{T,i} \tag{A.9}$$

The second step updates us to be in the parent frame. On the first iteration, this will set j to the link which rotor k is attached to (in the case where $\mu(k) = \lambda(\gamma(k))$).

The final step converts projects $\mathbf{b}$ to be a the generalized torque in coordinate j. The transpose of the motion subspace projection matrix is will project spatial forces to generalized torques. The $\Delta\mathbf{H}_{i,j}$ quantity will then satisfy $\tau = \phi^{\top}\mathbf{b}\ddot{q}$, which is familiar from the CRBA and from $\tau = \mathbf{H}\ddot{\mathbf{q}}$.

This algorithm can be seen as computing the effective inertia of a joint ($\mathbf{b}$), then determining how it "feels" from the perspective of all other joints.

Reflected Inertia

The previous algorithm does not account for all the spatial force the other bodies must react - the link geared to the rotor also feels the additional inertia of the rotor. We expect this term to be proportional to gear ratio squared. We can use the original $\mathbf{b}$ from the previous algorithm, which satisfies $\mathbf{f} = \mathbf{b}\ddot{q}$. This force creates a torque $\tau = N_k\phi_{\text{rot}}^{\top}\mathbf{f}$ on the geared link. Therefore, $\Delta\mathbf{H}_{i,i} = N_k\phi_{\text{rot}}^{\top}\mathbf{I}^{\text{rot}}\phi_{\text{rot}}N_k$, as expected. If N_k is large, this effect will likely dominate. This is a nice result, as this is by far the easiest effect to calculate.

Bias Forces (Gravity and Coriolis)

Using a very similar approach as in section A.2.2, we can compute the change in bias force. The bias force is independent of joint accelerations ($\ddot{\mathbf{q}}$), so we can set $\ddot{\mathbf{q}} = 0$ to simplify the derivation.

First, we determine the velocity of all bodies. This can be done recursively with

$$\mathbf{v}_i = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{v}_{\lambda}(i) + \phi_i \dot{q}_i \tag{A.10}$$

$$= {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{v}_{\lambda}(i) + \mathbf{v}_J \tag{A.11}$$

The velocity of the origin should be set to 0. Note that $\mathbf{v}_J = N_k \phi^{\mathrm{rot}} \dot{q}$ for rotors.

Next, we determine the acceleration of all bodies. This can be done recursively with

$$\mathbf{a}_i = {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{a}_{\lambda}(i) + \phi_i \ddot{q}_i + \dot{\phi}_i \dot{q}_i \tag{A.12}$$

$$= {}^{i}\mathbf{X}_{\lambda(i)}\mathbf{a}_{\lambda}(i) + \mathbf{v}_i \times \mathbf{v}_J \tag{A.13}$$

Note that in the second step, we eliminate the $\phi_i \ddot{q}_i$ term because we assume $\ddot{q}_i = 0$, and use the spatial motion cross product defined in [5]. Also, the acceleration of the origin should be set to $-g$ to add the effect of gravity.

With spatial vector algebra, a body's equation of motion can be written as

$$\mathbf{f} = \mathbf{I}\mathbf{a} + \mathbf{v} \times^{*} \mathbf{I}\mathbf{v} \tag{A.14}$$

which we can use to determine the spatial bias force acting on the rotor.

Like in section A.2.2, we propagate this to parent bodies, starting with $j = i$:

1. Transform $\mathbf{f}$ from j to $\lambda(j)$ coordinates: $\mathbf{f} = {}^{j}\mathbf{X}_{\lambda(j)}^{\top}\mathbf{f}$
2. Update $j = \lambda(j)$
3. Update bias $\Delta\mathbf{B}_j = \Delta\mathbf{B}_j + \phi_j^{\top}\mathbf{b}$
4. Repeat until root of tree is reached

Reflected Bias Forces

Similar to section A.2.2, the previous algorithm does not account for all the bias force - the link geared to the rotor can also feel bias effects: $\Delta\mathbf{B} = N_k \phi_{\text{rot}}^\top \mathbf{f}$ through the gearbox.

Algorithm 1 Forward Kinematics

Inputs: $\mathbf{q}, \dot{\mathbf{q}}, \text{model}$

Outputs: Body Jacobian ($\mathbf{J}_i$), Motion Transformation from origin (${}^i\mathbf{X}_0$), Body Velocity ($\mathbf{v}_i$), Body Bias Acceleration ($\mathbf{a}_i$), Body Velocity-Product Acceleration ($\mathbf{a}_{\text{vp},i}$), Rotor Velocity ($\mathbf{v}_k$), Rotor Bias Acceleration ($\mathbf{a}_k$), Contact point orientation (${}^0\mathbf{X}_{c,\text{rot}}$), Contact point position ($\mathbf{p}$), Contact Jacobian ($\mathbf{J}_c$), Contact point velocity ($\mathbf{v}_c$), Contact point bias acceleration ($\dot{\mathbf{J}}_{c,i}\dot{\mathbf{q}}$)

1: **for** $i = 1$ **to** N_b **do**
2: $\quad \mathbf{J}_i = \mathbf{0}$
3: $\quad \mathbf{X}_J = \text{jcalc}_i(q_i)$
4: $\quad \mathbf{v}_J = \boldsymbol{\phi}_i \dot{\mathbf{q}}_i$
5: $\quad {}^i\mathbf{X}_{\lambda(i)} = \mathbf{X}_J \mathbf{X}_{T,i}$
6: $\quad$ **if** $\lambda(i) = 0$ **then**
7: $\quad\quad {}^i\mathbf{X}_0 = {}^i\mathbf{X}_{\lambda(i)}$ ▷ transform from origin
8: $\quad\quad \mathbf{v}_i = \mathbf{v}_J$
9: $\quad\quad \mathbf{a}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{\text{grav}}$ ▷ bias acceleration
10: $\quad\quad \mathbf{a}_{vp,i} = \mathbf{0}$ ▷ velocity product acceleration
11: $\quad$ **else**
12: $\quad\quad {}^i\mathbf{X}_0 = {}^i\mathbf{X}_{\lambda(i)} {}^{\lambda(i)}\mathbf{X}_0$
13: $\quad\quad \mathbf{v}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{v}_{\lambda(i)} + \mathbf{v}_J$
14: $\quad\quad \mathbf{a}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{\lambda(i)} + \mathbf{v}_i \times \mathbf{v}_J$
15: $\quad\quad \mathbf{a}_{vp,i} = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{vp,\lambda(i)} + \mathbf{v}_i \times \mathbf{v}_J$
16: **for** $k = 1$ **to** N_r **do**
17: $\quad i = \gamma(k)$
18: $\quad \mathbf{v}_J = N_k \boldsymbol{\phi}_k q_{\gamma(k)}$
19: $\quad \mathbf{v}_k =^k \mathbf{X}_{\mu(k)} \mathbf{v}_{\mu(k)} + \mathbf{v}_J$
20: $\quad \mathbf{a}_k =^k \mathbf{X}_{\mu(k)} \mathbf{a}_{\mu(k)} + \mathbf{v}_k \times \mathbf{v}_J$
21: **for** $i = 1$ **to** N_b **do**
22: $\quad j = i$
23: $\quad {}^i\mathbf{X}_j = \mathbf{1}$ ▷ $i = j$
24: $\quad \mathbf{J}_{i,i} = \boldsymbol{\phi}_i$ ▷ in body coordinates
25: $\quad$ **while** $\lambda(j) > 0$ **do** ▷ Loop through i's ancestors
26: $\quad\quad {}^i\mathbf{X}_j = {}^i\mathbf{X}_j {}^j\mathbf{X}_{\lambda(j)}$
27: $\quad\quad j = \lambda(j)$
28: $\quad\quad \mathbf{J}_{i,j} = {}^i\mathbf{X}_j \boldsymbol{\phi}_j$ ▷ in i's body coordinates
29: **for** $i = 1$ **to** N_c **do** ▷ Loop through contacts
30: $\quad j = \sigma(i)$
31: $\quad \mathbf{R} = \text{MotionXform_rotation}({}^j\mathbf{X}_0)$ ▷ Coordinate Rotation
32: $\quad {}^0\mathbf{X}_{\text{rot}} = \text{blkdiag}(\mathbf{R}^\top)$ ▷ Rotation from contact to origin coordinate system
33: $\quad \mathbf{p}_i = \text{MotionXform_translation}({}^{\text{c}}\mathbf{X}_k {}^j\mathbf{X}_0)$ ▷ Contact Forward Kinematics
34: $\quad \mathbf{J}_{\text{c,i}} = {}^0\mathbf{X}_{\text{rot}} {}^{\text{c}}\mathbf{X}_k \mathbf{J}_j$ ▷ Contact Jacobian (rotated)
35: $\quad \mathbf{v}_{\text{c},i} = {}^0\mathbf{X}_{\text{rot}} \mathbf{v}_j$ ▷ Contact Velocity (rotated)
36: $\quad \dot{\mathbf{J}}_{\text{c},i}\dot{\mathbf{q}} = {}^0\mathbf{X}_{\text{rot}} {}^{\text{c}}\mathbf{X}_k \mathbf{a}_{vp,j} + \mathbf{v}^\omega_{\text{c},i} \times \mathbf{v}^v_{\text{c},i}$ ▷ Contact Bias Acceleration (rotated)

Algorithm 2 Add Rotors (Two-Step)

Inputs: $\mathbf{q}, \dot{\mathbf{q}}, \text{model}$

Outputs: $\Delta\mathbf{H}, \Delta\mathbf{C}$

1: **for** $k = 1$ **to** N_r **do**
2: $\quad i = \gamma(k)$
3: $\quad \Delta\mathbf{H}_{i,i} = \Delta\mathbf{H}_{i,i} + N_k^2 \mathbf{\Phi}_k^\top \mathbf{I}_k^{\text{rot}} \mathbf{\Phi}_k$ ▷ Gearing
4: $\quad \mathbf{b} = N_k \mathbf{I}_k^{\text{rot}} \mathbf{\Phi}_k$ ▷ Gearing
5: $\quad \mathbf{v}_J = N_k \mathbf{\Phi}_k q_{\gamma(k)}$
6: $\quad \mathbf{v}_k =^k \mathbf{X}_{\mu(k)} \mathbf{v}_{\mu(k)} + \mathbf{v}_J$
7: $\quad \mathbf{a}_k =^k \mathbf{X}_{\mu(k)} \mathbf{a}_{\mu(k)} + \mathbf{v}_k \times \mathbf{v}_J$
8: $\quad \mathbf{f}_k^r = \mathbf{I}_k \mathbf{a}_k + \mathbf{v}_k \times^* \mathbf{I}_k \mathbf{v}_k$
9: $\quad \Delta\mathbf{C}_{\gamma(k)} = N_k \mathbf{\Phi}_k^\top \mathbf{f}_k^r$ ▷ Gearing constraint, bias
10: $\quad$ **if** $\mu(k) \neq 0$ **then**
11: $\quad\quad \mathbf{f}_{\mu(k)} = \mathbf{f}_{\mu(k)} + {}^k\mathbf{X}_{\mu(k)}^\top \mathbf{f}_k^r$ ▷ non-geared $\mathbf{f}$
12: $\quad\quad \mathbf{I}_{\mu(k)}^c = \mathbf{I}_{\mu(k)}^c + {}^k\mathbf{X}_{\mu(k)}^\top \mathbf{I}_k^{\text{rot}} {}^k\mathbf{X}_{\mu(k)}$ ▷ Initialize $\mathbf{I}^c$
13: $\quad j = i$
14: $\quad$ **while** $\lambda(j) > 0$ **do** ▷ off-diagonal, gearing
15: $\quad\quad \mathbf{b} = {}^j\mathbf{X}_{\lambda(j)} \mathbf{b}$
16: $\quad\quad j = \lambda(j)$
17: $\quad\quad \Delta\mathbf{H}_{i,j} = \Delta\mathbf{H}_{i,j} + \mathbf{\Phi}_j^\top \mathbf{b}$
18: $\quad\quad \Delta\mathbf{H}_{j,i} = \Delta\mathbf{H}_{i,j}$
19: **for** $i = N_b$ **to** 1 **do**
20: $\quad \mathbf{I}_{\lambda(i)}^c = \mathbf{I}_{\lambda(i)}^c + {}^i\mathbf{X}_{\lambda(i)}^\top \mathbf{I}_i^c {}^i\mathbf{X}_{\lambda(i)}$
21: $\quad \Delta\mathbf{C}_i = \Delta\mathbf{C}_i + \mathbf{\Phi}_i \mathbf{f}_i$ ▷ RNEA for $\mathbf{f}$ change
22: $\quad$ **if** $\lambda(i) \neq 0$ **then**
23: $\quad\quad \mathbf{f}_{\lambda(i)} = \mathbf{f}_{\lambda(i)} + {}^i\mathbf{X}_{\lambda(i)} \mathbf{f}_i$
24: **for** $i = 1$ **to** N_b **do** ▷ CRBA change due to $\Delta\mathbf{I}^c$
25: $\quad \Delta\mathbf{H}_{i,i} = \mathbf{\Phi}_i^\top \mathbf{I}_i^c \mathbf{\Phi}_i$
26: $\quad \mathbf{b} = \mathbf{I}_i^c \mathbf{\Phi}_i$
27: $\quad j = i$
28: $\quad$ **while** $\lambda(j) > 0$ **do**
29: $\quad\quad \mathbf{b} = {}^j\mathbf{X}_{\lambda(j)}^\top \mathbf{b}$
30: $\quad\quad j = \lambda(j)$
31: $\quad\quad \Delta\mathbf{H}_{i,j} = \mathbf{\Phi}_j^\top \mathbf{b}$
32: $\quad\quad \Delta\mathbf{H}_{j,i} = \Delta\mathbf{H}_{i,j}$

Appendix B

Mini Cheetah Software Package

The Mini Cheetah software package can be downloaded from `https://github.com/mit-biomimetics/Cheetah-Software`.

Bibliography

[1] Introducing WildCat. https://www.youtube.com/watch?v=wE3fmFTtP9g, Oct 2013.

[2] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, In Press, 2018.

[3] Morteza Azad, Roy Featherstone, et al. Modeling the contact between a rolling sphere and a compliant ground plane. 2010.

[4] Gerardo Bledt, Matthew Powell, Benjamin Katz, Jared Carlo, Patrick Wensing, and Sangbae Kim. Mit cheetah 3: Design and control of a robust, dynamic quadruped robot. 10 2018.

[5] Roy Featherstone. *Rigid body dynamics algorithms*. Springer, 2014.

[6] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. qpoases: a parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, Dec 2014.

[7] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[8] Albert S. Huang, Edwin Olson, and David Moore. LCM: Lightweight communications and marshalling. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Taipei*, pages 4057–4062, Oct 2010.

[9] A. Jain. *Robot and Multibody Dynamics: Analysis and Algorithms*. Springer US, 2011.

[10] B. Katz, J. D. Carlo, and S. Kim. Mini cheetah: A platform for pushing the limits of dynamic quadruped control. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6295–6301, May 2019.

[11] Benjamin Katz. A low cost modular actuator for dynamic robots. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2018.

[12] Darren P Krasny and David E Orin. Evolution of a 3d gallop in a quadrupedal model with biological characteristics. *Journal of Intelligent & Robotic Systems*, 60(1):59–82, 2010.

[13] Q. Nguyen, M. J. Powell, B. Katz, J. D. Carlo, and S. Kim. Optimized jumping on the mit cheetah 3 robot. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7448–7454, May 2019.

[14] Massachusetts Institute of Technologyy. Backflipping mit mini cheetah. `https://www.youtube.com/watch?v=xNeZWP5Mx9s`.

[15] Hae-Won Park and Sangbae Kim. Quadrupedal galloping control for a wide range of speed via vertical impulse scaling. *Bioinspiration and Biomimetics*, 10(2):025003, 2015.

[16] Hae-Won Park, Patrick M Wensing, and Sangbae Kim. High-speed bounding with the mit cheetah 2: Control design and experiments. *The International Journal of Robotics Research*, 36(2):167–192, 2017.

[17] Bui Tuong Phong. *Illumination for Computer-Generated Images.* PhD thesis, 1973. AAI7402100.

[18] R. R. Playter and M. H. Raibert. Control of a biped somersault in 3d. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 582–589, Jul 1992.

[19] J. Pratt, J. Carff, S. Drakunov, and A. Goswami. Capture point: A step toward humanoid push recovery. In *2006 6th IEEE-RAS International Conference on Humanoid Robots*, pages 200–207, Dec 2006.

[20] Marc Raibert, Michael Chepponis, and H.Benjamin Jr. Running on four legs as though they were one. *Robotics and Automation, IEEE Journal of*, 2:70 – 82, 07 1986.

[21] Marc H. Raibert. *Legged Robots That Balance.* Massachusetts Institute of Technology, Cambridge, MA, USA, 1986.

[22] Unitree Robotics. Aliengo robot new progress: backflip etc. `https://www.youtube.com/watch?v=Ayxoji2xg3k`.

[23] D. Stewart and J. C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with coulomb friction. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 162–169 vol.1, April 2000.

[24] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, Mar 2006.

[25] P. M. Wensing, A. Wang, S. Seok, D. Otten, J. Lang, and S. Kim. Proprioceptive actuator design in the mit cheetah: Impact mitigation and high-bandwidth physical interaction for dynamic legged robots. *IEEE Transactions on Robotics*, 33(3):509–522, June 2017.

[26] Hongtao Zhang. Kick the dog when it walks. `https://www.youtube.com/watch?v=0P4E0bzkbXM`.

[27] Hongtao Zhang. Quadruped robot new video. `https://www.youtube.com/watch?v=gQV08cuaPYQ`.