University of California
Santa Barbara

Leveraging Trajectory Optimization to Improve Deep Reinforcement Learning, with Application to Agile Wheeled Robot Locomotion

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Guillaume D. Bellegarda

Committee in charge:

Professor Katie Byl, Chair
Professor João Hespanha
Professor Andrew Teel
Professor William Wang

December 2019

The Dissertation of Guillaume D. Bellegarda is approved.

Professor João Hespanha

Professor Andrew Teel

Professor William Wang

Professor Katie Byl, Committee Chair

December 2019

Leveraging Trajectory Optimization to Improve Deep Reinforcement Learning, with Application to Agile Wheeled Robot Locomotion

Copyright © 2019

by

Guillaume D. Bellegarda

Dedicated to my family

Acknowledgements

First and foremost I would like to thank my advisor, Professor Katie Byl, for supporting me and mentoring me over the last few years. She taught me many valuable academic and professional lessons, while giving me the freedom to explore different research topics with great independence. I would also like to thank my committee, Professor João Hespanha, Professor Andy Teel, and Professor William Wang, for their insightful comments and questions surrounding this dissertation.

The past and present members of the UCSB Robotics Lab have fostered good camaraderie in the lab: Nihar Talele, Sean Gillen, Roman Aguilera, Thomas Ibbetson, and Kees van Teeffelen. Nihar especially was always helpful in answering my many dynamics and optimization questions. I also acknowledge the two students I had the opportunity to mentor over the last year, Matthias Kargl and Danny Hernandez-Vitela, for their motivation and for making me a better leader.

Most of all, thank you to my family for their love and support. I am extremely lucky to have such amazing people in my life, and none of this would have been possible without them. And last but not least, thank you to Kaitlin for her love, support, and encouragement over the past few years.

Curriculum Vitæ

Guillaume D. Bellegarda

Education

2019	**University of California, Santa Barbara** **Ph.D.** in Electrical and Computer Engineering
2016	**University of California, Santa Barbara** **M.S.** in Electrical and Computer Engineering
2014	**University of California, Berkeley** **B.S.** in Electrical Engineering and Computer Science

Publications

G. Bellegarda and K. Byl. *Combining Benefits from Trajectory Optimization and Deep Reinforcement Learning.* submitted to IEEE International Conference on Robotics and Automation (ICRA), 2020, under review.

G. Bellegarda and K. Byl. *Versatile Trajectory Optimization for Dynamic Vehicle Maneuvers.* submitted to IEEE International Conference on Robotics and Automation (ICRA), 2020, under review.

G. Bellegarda and K. Byl. *Trajectory Optimization for a Wheel-Legged System for Dynamic Maneuvers that Allow for Wheel Slip.* in Proc. IEEE Conference on Decision and Control (CDC), 2019.

G. Bellegarda and K. Byl. *Training in Task Space to Speed Up and Guide Reinforcement Learning.* in Proc. IEEE International Conference on Intelligent Robots and Systems (IROS), 2019.

G. Bellegarda, N. Talele, K. Byl. *Nonintuitive Optima for Dynamic Locomotion: The Acrollbot.* in Proc. IEEE International Conference on Robotics and Automation (ICRA), 2018.

G. Bellegarda, K. van Teeffelen, K. Byl. *Design and Evaluation of Skating Motions for a Dexterous Quadruped.* in Proc. IEEE International Conference on Robotics and Automation (ICRA), 2018.

K. Driggs-Campbell, **G. Bellegarda**, V. Shia, S. Sastry, R. Bajcsy. *Experimental Design for Human-in-the-Loop Driving Simulations.* in EECS Department Technical Report, UC Berkeley, 2014.

Abstract

Leveraging Trajectory Optimization to Improve Deep Reinforcement Learning, with Application to Agile Wheeled Robot Locomotion

by

Guillaume D. Bellegarda

There exist several approaches to robot locomotion, ranging from more traditional hand-designed trajectories and optimization-based methods, to more recent successes with deep reinforcement learning (DRL). The advantage of traditional model-based approaches lies in the ability to perform stability and robustness analyses on the generated motions and trajectories, but such methods can suffer from modeling mismatches or expensive computation times for real-time control. DRL on the other hand requires very little information to be known a priori to (eventually) find a solution, at the cost of high sample complexity (long training times) and no stability or robustness guarantees on the learned policies. This thesis seeks to bridge the gap between these two areas by injecting model-based ideas and methods into deep reinforcement learning towards efficient, robust, and explainable learned control policies.

The application of this strategy is in agile wheeled robot locomotion, for example to JPL's RoboSimian and vehicles performing drift parking. A novel wheel model is presented, which when applied in an optimization framework, naturally results in trajectories that avoid or exploit wheel slipping depending on the task. These methods demonstrate the first planned and controlled drifting maneuvers for a wheel-legged system, with additional hardware results on a model vehicle.

Contents

Curriculum Vitae **vi**

Abstract **vii**

1 Introduction **1**

2 Wheeled Systems **6**
2.1 RoboSimian 6
2.2 Passive Wheel Systems 10
2.3 Active Wheel Systems 11
2.4 Motion Planning for Vehicles with Wheel Slip 12

3 Preliminaries **16**
3.1 General Dynamics 16
3.2 Nonholonomic Constraints 17
3.3 Existing Wheel Models 22
3.4 Friction as a Linear Complementarity Problem 24
3.5 Trajectory Optimization 28
3.6 Robot Learning 32

4 Design and Evaluation of Skating Motions for a Dexterous Quadruped **43**
4.1 Introduction and Motivation 44
4.2 Modeling 45
4.3 Design of Simple Skating Motions 49
4.4 Design for Variable Terrain Challenges 55
4.5 Conclusions and Discussion 61

5 Nonintuitive Optima for Dynamic Locomotion: The Acrollbot **63**
5.1 Introduction and Motivation 64
5.2 Acrollbot System 66
5.3 Optimization Framework 71
5.4 Results 73

5.5 Discussion and Conclusions 83

6 Versatile Trajectory Optimization for Dynamic Maneuvers that Allow for Wheel Slip 85
6.1 Modeling 86
6.2 Trajectory Optimization 92
6.3 RoboSimian Results 99
6.4 Car Results 105
6.5 Discussion and Conclusions 109

7 Training in Task Space to Speed Up and Guide Deep Reinforcement Learning 111
7.1 Introduction 112
7.2 Modeling 114
7.3 Training Environment 116
7.4 Results 120
7.5 Conclusion 129

8 Combining Benefits from Trajectory Optimization and Deep Reinforcement Learning 130
8.1 Introduction 131
8.2 Preliminaries 134
8.3 Trajectory Optimization 138
8.4 Cooperative Trajectory Optimization and Deep Reinforcement Learning . 140
8.5 Results 140
8.6 Conclusion 148

9 Conclusion 150

Bibliography 152

Chapter 1

Introduction

Research over the past decade has shown increasingly agile and dynamic robotic systems. Perhaps most famously, Boston Dynamics' biped Atlas has recently demonstrated abilities that are difficult even for humans, such as performing backflips, agile 'parkour' stunts, and running and jumping in nature [1–3]. Another Boston Dynamics robot, Handle, is a wheel-legged biped system that can perform a variety of dynamic maneuvers such as jumping over obstacles, descending stairs, and carrying payloads of over 100 pounds [4]. By leveraging a suction-cup end effector, it can also be used in a factory setting to move and arrange boxes onto pallets [5].

Within academia, the MIT Cheetah 2 has shown the ability to jump up to 80% of its nominal leg length while running at a speed of 2.4 (m/s) [6]. Its successor, the MIT Cheetah 3, demonstrates additional results in fast control of various gaits and jumps [7–9]. The smaller and more recent Mini Cheetah is the first four-legged robot to do a backflip [10]. Miniature robot jumpers such as Salto [11] or the EPFL jumper [12] can achieve impressive high jumping due to their small weight and design for storing energy using springs.

The above dynamic motions have all been executed from model-based trajectory

optimization methods, such as [13, 14], which involve the generation of a trajectory for a system by optimizing a certain cost function subject to various constraints. This optimal solution is then used as a reference trajectory for the actual system through the use of a low level controller, which enforces contraction of the system onto a low dimensional manifold. For example, for underactuated dynamics, some subset of directly-controlled degrees of freedom (DOFs) converge rapidly to desired trajectories, compared to other (passive, slower, but still stable) DOFs. Also within these methods is Model Predictive Control (MPC), which instead of first generating an entire reference trajectory and then tracking it, solves an online optimization at each time step for some finite horizon, executing only the first action and then re-solving the optimization from its new state [15–17].

Such methods have also shown impressive results in more general robotics applications, such as within the DARPA Robotics Challenge (DRC) [18, 19]. However, this same challenge produced a now famous robot 'blooper' reel of many different robots falling over, often times for what appears to be no reason [20]. A number of factors make robotics a challenging discipline, including perception, state estimation, modeling mismatches, and uncertainty in the environment. For example, if a biped 'thinks' it has contact with a handle and begins to turn it, when in reality it is not grasping anything, the forces it has planned for in the optimization are not actually present and a fall can be likely. Additionally, due to computational constraints, the models used in the optimizations are usually simplified from the true dynamics, and mismatches occur, especially if the true state is in poorly modeled areas of the workspace. In unmodeled regions, even the most well-designed controllers cannot provide guarantees or convergence. There is also no attempt to adapt or increase the fidelity of the model in these methods even as experience is gained from real-life executions.

In contrast, deep reinforcement learning (DRL) methods have shown recent success

on continuous control tasks in complicated robotic systems by making iterative improvements on the model and/or policy [21–24]. However such methods are applied with no prior knowledge of the systems, leading to problematic sample complexity and thus long training times. Unfortunately, little can be said about the stability or robustness of these resulting control policies, even if more traditional model-based optimal control solutions exist for these same systems. Additionally, DRL has been almost exclusively applied in simulation, where a failed trial has no repercussions. In the real world a failure can have catastrophic consequences, including damaging the robot or causing injury to humans in the area. Some recent works have successfully learned a policy for a real robot [25] [26], or transferred policies learned in simulation to the real system [27] [28]. Of particular note in [28] is that the learned policies outperform the authors' previous model-based methods with respect to both energy-efficiency and speed, as policies are not constrained with an incorrect or overly-simplified model.

A primary goal of this thesis is to bridge the gap between both areas of model-based trajectory optimization and deep reinforcement learning. In particular, DRL can be applied to many problems without needing any modeling or intuition about the system, and has empirically shown to outperform model-based methods both with respect to energy-efficiency and speed of resulting control policies. However, the high sample complexity and inability to prove any metrics about these learned policies makes real world training and deployment difficult. Model-based trajectory optimization methods, on the other hand, allow for stability and robustness analyses on generated motions and trajectories, but are only as good as the often over-simplified derived model, and may have prohibitively expensive computation times for real-time control. This work presents several methods of injecting model-based ideas and methods into deep reinforcement learning, towards efficient, robust, and explainable learned control policies. We also investigate both the emergence and use of non-intuitive optima in both of these areas.

In this thesis, the application of such methods is in agile wheeled robot locomotion. Wheel-legged systems in particular have the potential to combine benefits of two heavily studied research areas: the speed and efficiency of locomotion on wheels with the versatility and agility to surmount various obstacles of legged systems. However unlike in purely legged robots where a considerable challenge is avoiding falling down, to date most wheeled systems have been designed and studied only in the static stability sense, circumventing this issue. A notable exception is the aforementioned Boston Dynamics system Handle [4], and a comprehensive overview of existing wheeled robotic systems is given in Chapter 2. Most of these control systems either do not model or actively try to avoid wheel skidding/slipping (drifting), which can however easily occur in real world applications. We are particularly interested in such dynamic maneuvers here. Thanks to a novel wheel model formulation, we present a versatile trajectory optimization framework that can either avoid or exploit wheel skidding, depending on the desired trajectory length and goal, which also allows for gait scheduling that makes and breaks contact with the ground as in human skating.

The contributions and outline of this thesis are as follows:

- Chapter 2 introduces JPL's RoboSimian in the context of wheel-legged systems, for which a literature review is performed on the current state-of-the-art. Noting a research gap in dynamic maneuvers involving wheel slipping, we draw inspiration from the vehicle drifting community.

- Chapter 3 serves as a survey of the necessary background for the rest of this thesis, including discussion on robot dynamics and existing constraints in wheeled systems, and presents our novel wheel model to be used in our trajectory optimization framework. We also give an overview of trajectory optimization, and review concepts from applying machine learning in robotics, as used in the later chapters of

this work.

- Chapter 4 describes skating locomotion for RoboSimian for omnidirectional motion primitives with both three- and four-wheeled skating. These primitives are evaluated over mildly rough terrain with varying coefficients of friction.

- Chapter 5 explores the emergence of nonintuitive optima by optimizing both design parameters and motion trajectories for a novel wheeled system. By carefully analyzing and validating such solutions, we discover novel motion planning strategies that we, as human designers, would truly never have anticipated.

- Chapter 6 presents our versatile trajectory optimization framework for wheeled systems using our novel wheel model, that can either avoid or exploit wheel slipping for dynamic maneuvers.

- Chapter 7 proposes integrating forward and inverse kinematics into deep reinforcement learning for faster training and more robust control policies. We also show that we can model a high degree-of-freedom system such as RoboSimian as a representative simple one, and transfer learning of policies between these two systems.

- Chapter 8 proposes a method to combine the benefits from both areas of trajectory optimization (TO) and deep reinforcement learning while mitigating their drawbacks by (1) decreasing RL sample complexity by using existing knowledge of the problem with optimal control, and (2) providing an upper bound estimate on the time-to-arrival of the combined learned-optimized policy, allowing online policy deployment at any point in the training process by using the TO as a worst-case scenario action.

- Chapter 9 discusses conclusions and future work.

Chapter 2

Wheeled Systems

This chapter introduces the main robotic system studied throughout this thesis, Jet Propulsion Laboratory's (JPL's) RoboSimian. We also review the wide variety of wheel-legged systems that have been studied in the robotics community, where we differentiate between passive wheeled (no motor at the wheel) and actuated wheeled (motors directly providing forces to the wheel) systems. Lastly, noting that to date wheel-legged robotic systems have not planned for motions that exploit wheel slipping (drifting), we draw literature review from methods applied to vehicles, ranging from model RC cars to every day road vehicles.

2.1 RoboSimian

RoboSimian is a quadruped robot developed by NASA's Jet Propulsion Laboratory (JPL) in cooperation with the University of California, Santa Barbara (UCSB) [29–33]. Its purpose was primarily to compete in the DARPA Robotics Challenge (DRC), including trials in 2013 and finals in 2015, where Team RoboSimian achieved a fifth place finish. The robots competing in the DRC were challenged to perform various disaster

relief tasks in environments that would be unsafe for humans. These tasks included examples such as driving a vehicle, opening and entering doors, turning valves, using tools, and walking on uneven terrain [32, 33], with two snapshots shown in Figure 2.1.

Figure 2.1: Versatile locomotion with RoboSimian. At top: Walking on rough terrain at the DARPA Robotics Challenge in 2015. Bottom Left: RoboSimian bipedal and grasping a handhold during "egress of car" task. Bottom Right: RoboSimian with fully extended (and compact folded) limbs.

RoboSimian has four identical limbs, each with seven degrees of freedom. Each limb consists of three rigidly connected L-shaped elbow structures that contain two orthogonal actuators. All actuators in the limbs are identical, each consisting of a DC brushless motor that drives a (160:1) gearbox. These actuators produce peak torques of 580 [N·m] and a peak speed of 3.4 [rad/s] [33]. The distal end of each limb contains a seventh actuator allowing this end to rotate relative to the limb, without requiring any limb reconfiguration. In summary, RoboSimian has seven degrees of freedom (DOF) per limb, which results in a robot with high flexibility and many redundant solutions for end effector configurations.

The redundant solutions that come with such high flexibility offer both possibilities and challenges. For a desired end effector position, each 'elbow' of the limb has two configuration possibilities. With three such structures, an end effector configuration can be reached with up to $2^3 = 8$ different feasible limb configurations, as shown in Figure 2.2. For a given desired end effector configuration and terrain mapping, a subset of the 8 'IK families' may be better suited than others when considering collision avoidance, stability, and overall locomotion speed. Thus the high flexibility offers possibilities for avoiding collision by choosing a feasible limb configuration out of the redundant solutions, at the cost of increased complexity [31].

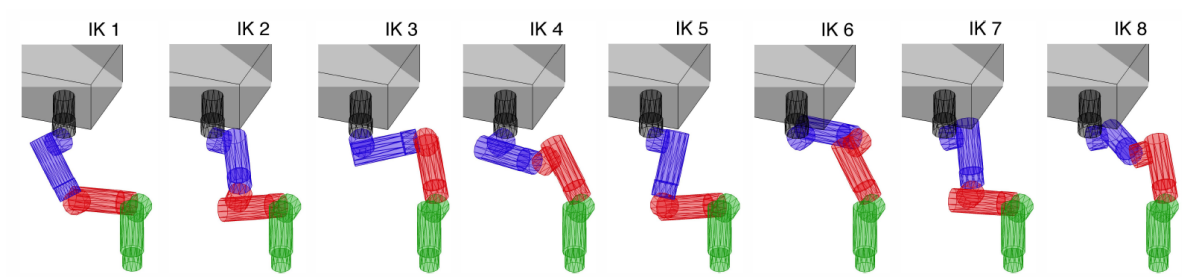

Figure 2.2: Eight different possible inverse kinematic (IK) solutions for reaching a particular end effector position.

As RoboSimian was designed with the ability to apply high torques at low speeds, it is very suitable for tasks involving heavy manipulation and climbing. However, such low actuator speeds result in slow locomotion, even over flat ground. Work in [34,35] explored walking with two limbs in contact with the ground at a time, commonly referred to as a static trotting gait, comparing with a static walking gait where at least three limbs were always in contact with the ground. Results showed that for the current actuator velocity limits, the overall body speeds were surprisingly similar for both methods.

During the DRC, RoboSimian was equipped with two omnidirectional driven wheels and two passive wheels to traverse relatively flat ground, as can be seen at the bottom right of Figure 2.3. Unfortunately, locomotion speed gains were limited by the actuators powering the wheels, with more powerful actuators increasing the weight of the robot. Additionally, due to the locations of the active/passive wheels, elaborate configuration changes were required to transition between walking and driving, further reducing the overall speed increase of using wheels.

It would be very beneficial if the high torques at low speeds of RoboSimian's limbs could be used to create high accelerations for the overall body. Attaching passive wheels at each forearm is one such idea, while minimizing weight from no added actuators, and leaving free six actuated degrees of freedom (DOFs) to set the 6-DOF pose for each skate, as shown in Fig. 2.3. Additionally, this allows for greater flexibility and quicker transitioning between walking and skating configurations than with the previously used driven wheels during the DRC. A caveat of mounting the skates at the forearms however is that there is currently no sensing available to sense contact of the skate with the ground. This eliminates the practical use of force feedback during any designed skating maneuvers.

Figure 2.3: RoboSimian on wheels.

2.2 Passive Wheel Systems

A wide variety of skating robots have already been studied. We first perform a literature review of passive-wheeled systems, which means those for which the wheels are

unactuated (no motor driving each wheel).

Among quadrupeds, perhaps the most famous skating robot is the roller-walker, which was developed and studied by Hirose and his research group [36–40]. The roller-walker has a reconfigurable end effector, to be used as either a foot or skate, quite similar conceptually to the added wheel we use with RoboSimian, as depicted in the lower left of Figure 2.3. A body of work exists on system design and motion planning for the roller-walker, for four-wheeled skating locomotion in a mostly-forward direction on flat ground. The leg-wheel concept has been extended to develop robot designs to minimize the number of required actuators, for example a unique design using a hexagonal frame [41].

Biped skating on passive wheels has also been explored by many researchers [42–45]. For such humanoid systems, balance becomes a greater concern, although few works deal directly with dynamic motions and generally employ static stability, using both skates always in contact. An exception includes [43], which includes two-wheel balance on a single skate, essentially becoming a special type of planar inverted pendulum system.

The focus for skating with RoboSimian is higher-dimensional limbs, in which all six degrees of freedom of each wheel can be controlled, with the motivation of enabling a wide range of motion primitives, ideally including agile and dynamic maneuvers.

2.3 Active Wheel Systems

In wheel-legged systems with actuated wheels, there are few examples of hybrid walking-driving motions. The focus has been predominantly on statically-stable driving motions, where the legs are used as an active suspension or to step over obstacles [46–54]. These applications either completely avoid wheel lift-off, or consider maintaining only static stability to avoid falling when planning motions that involve steps.

Within quadrupeds, ETH Zurich's ANYmal [55], equipped with non-steerable, torque-

controlled wheels, has demonstrated agile motions combining walking and driving, including full flight modes in simulation [56]. More recently in hardware, for example at the DARPA Subterranean Challenge, locomotion planning for ANYmal on wheels resulted in dynamic hybrid walking-driving motions on various terrains [57]. However, this planning did not explicitly account for wheel slipping, as modeling enforced no-slipping conditions, and due to the low coefficient of friction of the muddy environment, the robot had issues with balancing and falling.

A family of wheel-legged systems are the Skaterbots [58], in which the authors present a general optimization framework for motion planning for a wide variety of wheel-legged systems, including those with either passive or actuated wheels. Examples include passive wheel 'swizzlebots' and quadruped 'skatebots', to varying legged systems such as a 'dinobot'. Motions are generated by formulating a general trajectory optimization problem that explicitly models each wheel's full state and enforces no-slip constraints when a wheel is in contact with the ground. Thus this framework does not account for skidding or slipping of the wheel.

Perhaps the currently most agile and dynamic wheeled system is the previously discussed Boston Dynamics robot Handle [4]. Handle has actuated wheels and can perform a variety of impressive motions quickly and agilely, for sufficiently smooth terrain. However, since there is no associated publication, little is known about the design of its modeling and control framework.

2.4 Motion Planning for Vehicles with Wheel Slip

As accounting and planning for wheel slipping/skidding in wheel-legged systems has not been studied, we draw inspiration from work in dynamic vehicle maneuvers involving sustained and transient drift.

Vehicle dynamics have been well studied at varying levels of abstraction. These range from simple models such as kinematic and dynamic bicycle models with few state variables, to a popular high fidelity vehicle dynamics simulator, CarSim, which uses over 250 state variables to estimate full vehicle dynamics. Ideally, the simplest model that still accurately captures the true dynamics is desired for planning and control purposes.

We are specifically interested in dynamic maneuvers for both wheel-legged systems and vehicles involving skidding/drifting. There have been a number of recent works in this area, and of note are the different models used as well as the (mostly) decoupling of planning and control with a variety of techniques.

Borrelli et al. [59] note that a bicycle model with constant normal tire loads capture most of the relevant nonlinearities associated with lateral stabilization of the vehicle, and present an MPC based approach to active steering, such as used during lane changes, showing results in simulation. Kong et al. [60] studied using kinematic and dynamic vehicle models for autonomous driving with MPC, noting the kinematic model is both computationally less expensive as well as avoids the singularity of popular tire models, and so can be used at low speeds.

Velenis et al. introduced a bicycle model with suspension dynamics and applied numerical optimization to analyze drft cornering behavior [61] [62]. They proposed a framework for achieving maximum corner exit velocity or minimal time cornering, and validated their method in CarSim.

Kolter et al. [63] remarked that over short periods of time, even complex dynamics such as a car while skidding are often remarkably deterministic. With this intuition they developed a mixed open-loop and closed-loop control framework using a probabilistic method called multi-model LQR, which they applied on a real vehicle for a sliding parking maneuver. The vehicle model for normal driving was learned from experimental data from real driving, and the system was provided a demonstration of the desired maneuver.

Other works adopting mixed open-loop closed-loop control include [64], [65]. These works study drift parking and cornering, and use a rule-based algorithm to plan a reference trajectory. In [66] the authors divide the cornering problem into free, transit, and drifting regions; where path planning is done by an RRT and rule-based method, with a PI controller for the transit segment. A mixed open-loop closed-loop controller is used to track the trajectory, and the authors validate their methods in CarSim and/or on the BARC car model.

Williams et al. [67] present a MPC algorithm based on a stochastic optimal control framework, where the optimal controls take the form of a path integral, approximated with an importance sampling scheme. The algorithm is validated on a 1/5 scale Auto-Rally vehicle, at speeds beyond the friction limits of the vehicle.

Voser et al. [68] present an analytical framework to understand the dynamics of drifting with a simple bicycle model with nonlinear tires, modeling different coefficients of friction between the front and rear tires with the ground. Experimental evidence was collected on a physical all-electric vehicle, P1. Goh et al. [69] present a controller to drift while tracking a reference path, using a four-wheel model with steady-state weight transfer, with experimental validation on the physical vehicle MARTY.

Most of the above works decouple planning from control, that is to say, use different models for, as well as separate the processes of, planning and control of a desired motion. Ideally we would want to plan (and then execute) motions through the same well understood real dynamic model. Additionally, the tire approximations in the above works all rely on some variant of the linear, Fiala, or Pacejka models [70] [71], which all depend on the tire slip angle. This angle becomes singular at low vehicle speeds due to the vehicle velocity term in the denominator, and is thus difficult to use in a general trajectory optimization framework, additionally so because of the discontinuities in the latter two of these models. These tire models and the slip angle will be further discussed

in Section 3.3, and we will introduce our novel tire model to be used in our optimization framework for dynamic maneuvers for general wheeled systems in Section 3.4.

Chapter 3

Preliminaries

This chapter serves as a survey of the necessary background for the rest of this thesis. Sections 3.1 and 3.2 discuss robot dynamics and nonholonomic constraint considerations (usually) implemented for wheels, as well as methods to solve for the associated λ multipliers. Section 3.3 reviews existing tire models and the challenges involved with their integration into general trajectory optimization frameworks. Section 3.4 reviews friction as a linear complementarity problem and presents our novel wheel model. Section 3.5 presents an introduction to trajectory optimization methods and provides the formulation used in the rest of this thesis. Section 3.6 reviews background in deep reinforcement learning and imitation learning, to be used in Chapters 7 and 8.

3.1 General Dynamics

The equations of motion for a general robotic system can be written as:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + A(q)^T\lambda = B(q)u + F \tag{3.1}$$

where q are the generalized coordinates, $M(q)$ is the inertial matrix, $C(q, \dot{q})\dot{q}$ denotes centrifugal and Coriolis forces, $G(q)$ captures potentials (gravity), $A(q)^T\lambda$ are constraint forces (where λ are unknown multipliers a priori), $B(q)$ maps control inputs u into generalized forces, and F contains non-conservative forces such as friction.

3.2 Nonholonomic Constraints

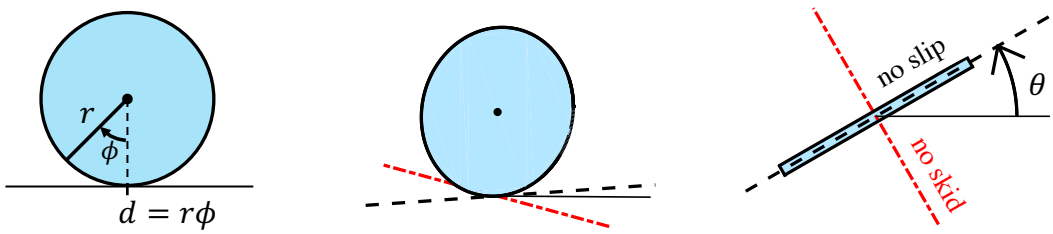

Figure 3.1: Simple wheel with no-slip and no-skid constraint axes: planar, angled, and overhead views.

For general wheeled mobile robots, $A(q)^T\lambda$ in Equation 3.1 typically contains constraints ensuring no slip (free rolling in the direction the wheel is pointing, x_w) and no skid (no velocity along the wheel's rotation axis y_w perpendicular to the free rolling direction), which come from writing these constraints in Pfaffian form $A(q)\dot{q} = 0$. These constraints are visually depicted in Figure 3.1, and derived below.

If we take x_w and y_w to be unit vectors in the wheel frame, then they are defined as:

$$x_w = R(\theta) \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3.2}$$

$$y_w = R(\theta) \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.3}$$

where $R(\theta)$ represents the counterclockwise rotation matrix for angle θ:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{3.4}$$

The displacement of a non-slipping wheel with radius r and angle ϕ is $d = r\phi$, and its velocity is $v = r\dot{\phi}$. Due to the non-holonomic nature of the system, the constraints appear in global (x, y) coordinates as follows:

$$x_w^T \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = r\dot{\phi} \tag{3.5}$$

$$y_w^T \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = 0 \tag{3.6}$$

These constraints can be added to the equations of motion through the term $A(q)^T\lambda$. In fact, the constraints can be written in Pfaffian form, where if the generalized coordinates for the 3D wheel model (assuming it cannot tip over) are

$$q = [x, y, \theta, \phi]^T \tag{3.7}$$

can be written in the form

$$A(q)\dot{q} = 0 \tag{3.8}$$

with

$$A(q) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & -r \\ -\sin\theta & \cos\theta & 0 & 0 \end{bmatrix} \tag{3.9}$$

The coordinates and matrix $A(q)$ have been derived here for a single wheel, but adding the same constraint for every wheel in the system is straightforward.

3.2.1 Solving for Lambda Multipliers

To determine the λ multipliers in Equation (3.1), we consider three different methods, which are more detailed in [72], for example.

Method 1

As $q \in \mathbb{R}^{n\times 1}$ and $A(q) \in \mathbb{R}^{m\times n}$, where m is the number of constraint equations (in general for a system with W wheels, $m = 2W$), we can define a matrix $S(q) \in \mathbb{R}^{n\times(n-m)}$ to be a full rank matrix formed by a set of smooth and linearly independent vector fields, spanning the null space of $A(q)$ such that:

$$S(q)^T A(q)^T = 0 \tag{3.10}$$

Then we can find a set of vectors of time functions $v(t) \in \mathbb{R}^{(n-m)\times 1}$ so that

$$\dot{q}(t) = S(q)v(t) \tag{3.11}$$

and use $S(q)$ to eliminate the Lagrange multipliers in (3.1). Differentiating (3.11) to get the state acceleration gives:

$$\ddot{q}(t) = \dot{S}(q)v(t) + S(q)\dot{v}(t) \tag{3.12}$$

Plugging (3.11) and (3.12) into (3.1), and pre-multiplying by $S(q)^T$:

$$S(q)^T\Big(M(q)\big(\dot{S}(q)v(t) + S(q)\dot{v}(t)\big) + C(q,\dot{q})S(q)v(t) + A(q)^T\lambda = B(q)u\Big) \tag{3.13}$$

$$S(q)^T M(q)S(q)\dot{v}(t) + \Big(S(q)^T M(q)\dot{S}(q) + S(q)^T C(q,\dot{q})S(q)\Big)\, v(t) = S(q)^T B(q)u \tag{3.14}$$

and dropping $(q), (t)$ for clarity gives:

$$S^T MS\dot{v} + \Big(S^T M\dot{S} + S^T CS\Big)\, v = S^T Bu \tag{3.15}$$

$$\tilde{D}\dot{v} + \tilde{C}v = \tilde{B}u \tag{3.16}$$

with $\tilde{D} = S^T MS$, $\tilde{C} = (S^T M\dot{S} + S^T CS)$, $\tilde{B} = S^T B$.

To map back from this transformed and reduced set of coordinates to our original generalized coordinates from the full system, Equations (3.16) and (3.12) can be solved together.

Method 2

It is also possible to solve the constraint equation (3.8) directly in conjunction with the general dynamics equation (3.1), by first differentiating (3.8):

$$A(q)\ddot{q} + \dot{A}(q)\dot{q} = 0 \tag{3.17}$$

$$A(q)\ddot{q} = -\dot{A}(q)\dot{q} \tag{3.18}$$

and then solving for the λ's as well in (3.1), in matrix form:

$$\begin{bmatrix} M & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} = - \begin{bmatrix} C \\ \dot{A} \end{bmatrix} \dot{q} + \begin{bmatrix} B \\ 0 \end{bmatrix} u \tag{3.19}$$

Method 3

We can also explicitly solve for the λ's by considering equations (3.17) and (3.1), reproduced below:

$$A(q)\ddot{q} + \dot{A}(q)\dot{q} = 0 \tag{3.20}$$

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + A(q)^T\lambda = B(q)u \tag{3.21}$$

Dropping $(q), (q,\dot{q})$ for clarity, we can rewrite (3.21) as:

$$\ddot{q} = M^{-1}\left(Bu - C\dot{q} - G - A^T\lambda\right) \tag{3.22}$$

and plug this result into (3.20):

$$A\Big(M^{-1}\left(Bu - C\dot{q} - G - A^T\lambda\right)\Big) + \dot{A}\dot{q} = 0 \tag{3.23}$$

$$-AM^{-1}A^T\lambda = -\Big(\left(AM^{-1}\right)\left(Bu - C\dot{q} - G\right) + \dot{A}\dot{q}\Big) \tag{3.24}$$

$$\lambda = \left(AM^{-1}A^T\right)^{-1}\Big(\left(AM^{-1}\right)\left(Bu - C\dot{q} - G\right) + \dot{A}\dot{q}\Big) \tag{3.25}$$

λ can then be plugged directly into (3.1).

3.2.2 Solving for λ conclusion

There are tradeoffs in the above methods. Method 1 requires finding a matrix $S(q)$ in the nullspace of $A(q)$, which can be non-obvious. Method 2 allows the λ's to be free variables in the optimization framework, which can however lead to unnecessary computation. Method 3 avoids the number of open variables in the optimization, but can lead to greater computation time due to the explicit expression for λ, which can be an expensive evaluation. Both methods 2 and 3 are used in this thesis, switching based on experimental computation time evaluations in the optimization framework. The interested reader is directed to [72] for more details.

3.3 Existing Wheel Models

In the previous section we considered adding nonholonomic constraints to the dynamics equation to represent non-slipping and non-skidding wheels. Such constraints can represent a good approximation at low speeds on terrains with high enough coefficients of friction. However, in this thesis, we are interested in dynamic and agile motions such as skidding/drifting into a parking spot, for which the above constraints do not capture the true dynamics. This section provides details on existing tire models in the context of vehicles.

Characterizing tire forces is a major subject of vehicle dynamics [70, 71, 73, 74], but developing accurate tire models remains a challenge due to the myriad of factors that affect the forces generated from them. Many mathematical models have been developed for tire forces, ranging from physics-based models that treat the tire as a spring, to empirical-based models that simply fit a function with parameters to data.

The forces generated at each tire depend on several variables, including the slip angle α, slip ratio ω, normal force F_n, coefficient of friction μ, and temperature of the tire. For

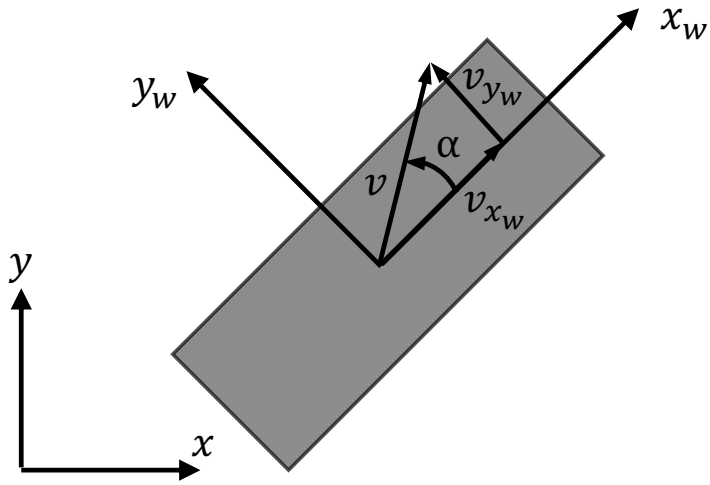

Figure 3.2: Overhead view of a tire. The tire side slip angle α measures the angle between the lateral and longitudinal component of the overall tire velocity vector.

dynamic skidding maneuvers, of particular interest is the slip angle α, which is defined as the angle between the lateral component and longitudinal component of the velocity of a tire, in the local tire frame. As shown in Figure 3.2, x_w and y_w are the local axes of the tire, with corresponding velocities v_{x_w} and v_{y_w}, respectively. The overall velocity of the tire is v, and thus the slip angle α captures the angle between the wheel's rolling direction and the actual direction of motion. A non-zero slip angle α means that slip is occurring, and there is a lateral force at the tire perpendicular to the direction of travel. More formally, the tire side slip angle is defined as:

$$\alpha = \tan^{-1}\left(\frac{v_{y_w}}{v_{x_w}}\right) \tag{3.26}$$

The tire approximations used in existing works investigating drifting [59–69], previously discussed in Section 2.4, all rely on some variant of the linear, Fiala, or Pacejka models [70, 71], which all depend on this tire slip angle. This angle becomes singular at low vehicle speeds due to the velocity term in the denominator, and is thus difficult to use to generate versatile motions through a general trajectory optimization framework,

additionally so because of the discontinuities in the latter two of these models.

For this reason, most of the above works decouple planning from control, that is to say, use different models for, as well as separate the processes of, planning and control of a desired motion. Examples have included rule-based methods to switch between models, or planning with a kinematic model only, or planning under very specific conditions avoiding the discontinuous ranges of the existing tire models. Ideally we would want to plan (and then execute) motions through the same well understood real dynamic model.

3.4 Friction as a Linear Complementarity Problem

Toward deriving a general wheel model not dependent on the aforementioned slip angle, that can be used in an optimization framework, we first consider the classical Coulomb friction model, which has been widely used in robotics applications [13, 19, 75, 76]. Determining the friction forces for a contact point can be posed as a set of Linear Complementarity Problems (LCPs), as detailed by Stewart and Trinkle [77, 78]. The circular Coulomb friction cone, assuming a single contact, can be approximated by a polyhedral cone $\mathcal{F}$ (see Figure 3.3):

$$\mathcal{F}(q) = \{\, F_n n \,+\, D\beta \,|\, F_n \geq 0,\ \beta \geq 0,\ e^T\beta \leq \mu F_n \} \tag{3.27}$$

where F_n is the magnitude of the normal contact force, n is the local unit z-axis representing the six dimensional unit wrench of the normal component of the contact force, the columns of D are direction vectors that positively span the space of possible generalized friction forces, $e = [1, 1, ..., 1]^T \in \mathbb{R}^p$ where p is the number of edges of the polyhedral approximation, and $\beta \in \mathbb{R}^p$ is a vector of weights.

The polyhedral cone in the Figure is made up of 8 direction vectors d_j, as an example,

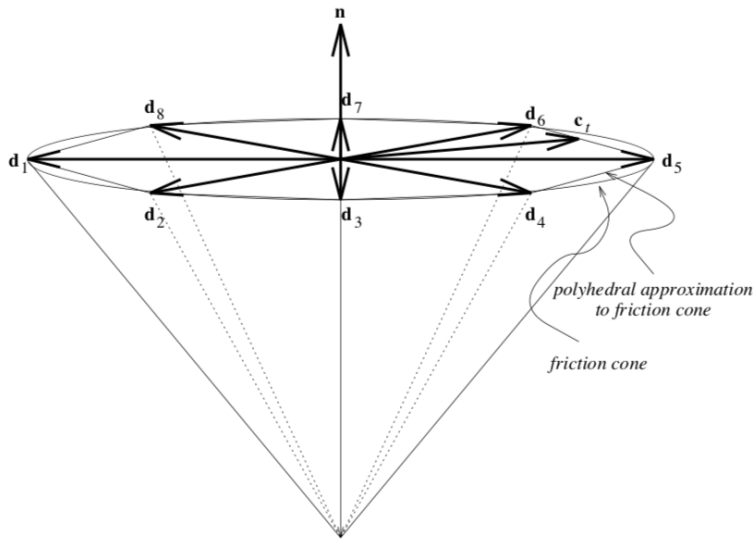

Figure 3.3: Circular friction cone and polyhedral approximation [77].

where increasing the number of vectors corresponds to a more accurate representation of the friction cone. Note that if s is a vector in the subspace of D, then $d_j^T s \geq 0$ for all j implies that $s = 0$. Here n and d_j are unit vectors for isotropic Coulomb friction modeled in the space of the contact force, however, the lengths of d_j would vary for anisotropic Coulomb friction.

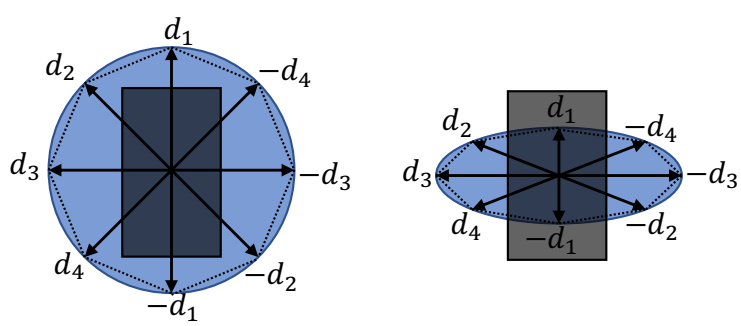

Figure 3.4: Isotropic and anisotropic polyhedral (shown here as octagonal) friction cone approximations of a contact point at the center of the wheel, when viewed from above. The normal force is coming out of the page, and each arrow represents a direction vector.

Under Coulomb friction, the frictional force opposes the direction of motion. In the wheels of a typical 2D planar car model, this would largely be opposite to the direction

of wheel roll, for most behavior, not lending to be super useful for an optimization framework to produce through-contact motions.

If we consider the wheels of a physical car model, we can directly input forces along the rolling direction x_w, and thus set the net force in these directions to overcome any static friction trying to oppose motion along this axis, if we neglect wheel slip. In Figure 3.4, this means that d_1 should be of minimal importance, and in fact we will set $|d_1| \approx 0$, and consider an anisotropic friction model. For RoboSimian with passive (unactuated) wheels, we cannot directly control forces along the direction of roll, which will depend on terrain properties and the body to which the wheels are attached. Some small damping should be expected to slow down a rolling wheel, motivating keeping $|d_1| \ll 1$.

Intuitively, when driving a car, there is a force perpendicular to a wheel's rolling direction preventing skidding from happening under normal circumstances. During aggressive maneuvers however, the body/tire may overcome the static frictional force in this direction, causing a loss of traction and thus skidding. This implies that the magnitude of vector d_3 from Fig. 3.4 will be most important to plan realistic skidding (or lack thereof) as in the dynamic parking tasks in Chapter 6.

In this thesis, we will consider two different anisotropic friction models, that essentially amount to considering each wheel as more of a (possibly powered) 'ice' skate than a 'roller' skate, as we do not model wheel rotation angles:

(1) $|d_1| = |d_2| = |d_4| = 0$, $|d_3| = 1$

(2) $|d_1| = 0.01, |d_2| = |d_4| = 0.3$, $|d_3| = 1$.

LCP Formulation

For a given wheel contact at each time step, using the above anisotropic friction cones, the following inequality constraints are enforced to produce the correct friction forces:

$$\gamma e + D^T v^{k+1} \geq 0, \quad \beta \geq 0 \tag{3.28}$$

$$\mu F_n - e^T \beta \geq 0, \quad \gamma \geq 0 \tag{3.29}$$

with the complementarity conditions:

$$(\gamma e + D^T v^{k+1})^T \beta = 0 \tag{3.30}$$

$$(\mu F_n - e^T \beta)\gamma = 0 \tag{3.31}$$

where γ can be interpreted as a scalar roughly equal to the magnitude of the relative tangential velocity at a contact, and v^{k+1} is the global 2D planar velocity vector of the contact point at the end of the next time step. In this thesis, the contact schedule is known, so $F_n \geq 0$ when a wheel is in contact, otherwise there are no friction forces. These conditions ensure that the friction force lies within the friction cone approximation, and that the friction forces push against the tangential acceleration. From the above constraints the friction F_{fric} at a particular contact can then be recovered with:

$$F_{fric} = \begin{bmatrix} F_{fric_x} \\ F_{fric_y} \end{bmatrix} = D\beta \tag{3.32}$$

F_{fric} for each contact can then be mapped from global coordinates to generalized coordinates with matrix $J(q)^T$ and input into Equation 3.1 as part of the non-conservative forces F.

3.5 Trajectory Optimization

In robotic locomotion applications, the term trajectory is often used to describe the full path from a starting position to an end goal location. This path usually includes both states (i.e. positions and velocities) and controls (i.e. motor torques) as functions of time. Trajectory optimization thus refers to a set of methods used to find the 'best' choice of trajectory, typically by selecting the inputs to the system (control inputs) as functions of time. An objective function is used to mathematically describe what is meant by 'best' trajectory, which may for example penalize control inputs (i.e. an efficient trajectory is desired) or maximize speed (i.e. we want the trajectory getting to the goal location as fast as possible, ignoring efficiency).

3.5.1 The Trajectory Optimization Problem

There are many ways to formulate trajectory optimization problems [79–83]. We will first focus on single-phase continuous-time trajectory optimization problems, where the system dynamics are continuous throughout the entire trajectory. In general, the objective function can include any number of terms depending on the desired task, but usually we consider two terms: a boundary objective $J(\cdot)$ and a path integral along the entire trajectory, with the integrand $w(\cdot)$, as follows:

$$\min_{t_0, t_F, x(t), u(t)} J\big(t_0, t_F, x(t_0), x(t_F)\big) + \int_{t_0}^{t_F} w\big(\tau, x(\tau), u(\tau)\big) d\tau \tag{3.33}$$

For the rest of this thesis, we will often refer to the entire cost function as $J(\cdot)$, to include both terms above. In the context of optimization, the term *decision variable* is used to describe the variables that the optimization solver is adjusting to minimize the objective function. In general these decision variables include the initial and final time

(t_0, t_F), the state $x(t)$ and control $u(t)$ trajectories, and possibly additional variables left free in the optimization (i.e. normal forces or constraint forces).

The optimization is subject to a variety of limits and constraints, shown in the following equations 3.34–3.41. For feasible and realistic trajectories, perhaps the most important of these constraints is the system dynamics, which are typically nonlinear and describe how the system changes over time:

$$\dot{x}(t) = f\big(t, x(t), u(t)\big) \qquad \text{System Dynamics} \tag{3.34}$$

Another important type of constraint enforces restrictions along the trajectory. These path constraints may arise, for example, to avoid collisions with obstacles in the environment or between robot links, or to maintain an end effector at a particular location during a time step.

$$h\big(t, x(t), u(t)\big) \leq 0 \qquad \text{Path Constraint} \tag{3.35}$$

In addition to path constraints, there may be nonlinear boundary constraints, which place restrictions on the initial and final states of the system. Such a constraint could be used to find a trajectory to a particular goal location for the robot, or to find periodic walking (or rolling) gaits.

$$g\big(t_0, t_F, x(t_0), x(t_F)\big) \leq 0 \qquad \text{Boundary Constraint} \tag{3.36}$$

There are also often limits on states and control, such as physical limitations from the motors which can only produce a finite and maximum torque. A robot's joints would then also have a maximum angular rate, and may have additional physical limits on

possible position angles.

$$x_{low} \leq x(t) \leq x_{upp} \qquad \text{Path Bound on State} \tag{3.37}$$

$$u_{low} \leq u(t) \leq u_{upp} \qquad \text{Path Bound on Control} \tag{3.38}$$

It might also be necessary to include specific limits on the initial and final time and state, to ensure that the solution to the path planning problem reaches the goal within a desired time window, or that it reaches a particular goal region in state space:

$$t_{low} \leq t_0 < t_F \leq t_{upp} \qquad \text{Bounds on Initial and Final Times} \tag{3.39}$$

$$x_{0,low} \leq x(t_0) \leq x_{0,upp} \qquad \text{Bound on Initial State} \tag{3.40}$$

$$x_{F,low} \leq x(t_F) \leq x_{F,upp} \qquad \text{Bound on Final State} \tag{3.41}$$

3.5.2 Transcription Methods

A popular class of techniques for solving trajectory optimization problems are transcription methods. Such methods form a finite dimensional optimization problem by discretizing the trajectories in time at N knot points as $x_1, ..., x_N$, $u_1, ..., u_N$, and between these knot points, enforcing the integral of the dynamics as a constraint. Within this class are multiple-shooting methods, which typically use variable step numerical integrators between each pair of knot points, and have been successfully applied to robotic locomotion tasks as in [84, 85]. Other common methods avoid the costly variable step integration by using implicit, single step schemes between knot points, such as via Euler integration schemes or by approximating trajectories as Hermitian splines [80, 82, 83, 86]. Direct collocation [86] and variants have been widely used to optimize walking and running gaits as in [13, 14, 19, 75, 76, 87, 88].

In this thesis, we take direct inspiration from the work done in [13, 19, 75, 76], where full body control for underactuated systems is achieved via trajectory optimization and stabilization under constraints. [13] in particular introduces an algorithm (DIRCON) that extends the direct collocation method, incorporating manifold constraints to produce nominal trajectories with third-order integration accuracy. In the rest of this thesis we will make use of such methods in the application to agile wheeled systems, using the novel wheel model explained in Section 3.4. This discrete time optimization has the following format:

$$\underset{x_k, u_k;\ k=1...N}{\text{minimize}} \quad J\big(x_N\big) + h\sum_{k=1}^{N} w\big(x_k, u_k\big) \tag{3.42}$$

$$\text{subject to} \quad d\big(x_k, u_k, x_{k+1}, u_{k+1}\big) = 0,\ \ k = 1, ..., N-1 \tag{3.43}$$

$$\phi\big(x_k, u_k\big) = 0,\ \ k = 1, ..., N \tag{3.44}$$

$$\psi\big(x_k, u_k\big) \geq 0,\ \ k = 1, ..., N \tag{3.45}$$

where x_k is the full state of the system at sample k along the trajectory, u_k is the corresponding control input, J and w are final and additive costs we are interested in minimizing, and N is the total number of samples along the trajectory. The function d captures the dynamic constraints between two consecutive knot points, which depends on the integration scheme. In this thesis we use the backward Euler method as follows:

$$d\big(x_k, u_k, x_{k+1}, u_{k+1}\big) := x_{k+1} - \big(x_k + hf(x_{k+1}, u_{k+1})\big) \tag{3.46}$$

where h is the time between sample points k and $k+1$. The functions $\phi(\cdot)$ and $\psi(\cdot)$ capture arbitrary constraints on the path, boundary, state, and control inputs.

3.6 Robot Learning

There is rich literature in applying machine learning to robotics, ranging from deep reinforcement learning to learning from demonstration. This section reviews the necessary background in reinforcement learning and imitation learning for Chapters 7 and 8. For a more comprehensive overview, the reinforcement learning (RL) framework is described thoroughly by Sutton and Barto [89], and more recent survey papers review the current state of learning in robotics, from deep reinforcement learning to learning from demonstration [90–93].

3.6.1 Reinforcement Learning

A robotics task such as navigation, manipulation, locomotion, etc. can be formalized as a Markov Decision Process (MDP), in which the agent interacts with the environment through a sequence of observations, actions, and reward signals. At each time step t, the agent receives a state s_t in state space $\mathcal{S}$, and selects an action a_t from an action space $\mathcal{A}$, following a policy $\pi(a_t|s_t)$, which maps from states s_t to actions a_t. The agent receives a scalar reward r_t, and transitions to the next state s_{t+1} according to the environment dynamics, or model, for reward function $\mathcal{R}(s, a, s')$ and transition function $\mathcal{P}(s_{t+1}|s_t, a_t)$ respectively. An MDP is thus given as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\gamma \in (0, 1)$ is a discount factor placing more emphasis on immediate rewards.

If an MDP is *episodic*, i.e. the state is reset after each episode of length T, then the sequence of states, actions, and rewards in an episode constitutes a *trajectory* (or *rollout*) of the policy. We will formally define a trajectory τ as:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{3.47}$$

The first state of the world, s_0, is randomly sampled from the start-state distribution ρ_0, by $s_0 \sim \rho_0(\cdot)$. The goal of the agent is to maximize some notion of cumulative reward over a trajectory, which will be notated as the return $R(\tau)$. One kind of return is the finite-horizon undiscounted return, which is just the sum of rewards obtained over the episode of length T:

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{3.48}$$

Another kind of return is the infinite-horizon discounted return, which is the sum of all rewards ever obtained by the agent, but discounted by how far off in the future they are obtained:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{3.49}$$

which intuitively corresponds to a reward now is better than a reward later. Mathematically we note that an infinite-horizon sum of rewards may not converge to a finite value, making such a scenario hard to deal with in the following discussion.

The Reinforcement Learning Problem

The goal of a reinforcement learning agent is to find a policy which maximizes the above expected return (whether infinite or finite horizon, discounted or undiscounted). If we suppose that both the environment transitions and the policy are stochastic, then the probability of a particular trajectory τ is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{3.50}$$

The expected return $J(\pi)$ is then:

$$J(\pi) = \int P(\tau|\pi)R(\tau)d\tau = \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{3.51}$$

The central optimization problem in RL can then be expressed as finding the policy that maximizes the expected return as

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{3.52}$$

where π^* is the optimal policy.

Value Functions

It will often be useful to know the value of a state, or state-action pair. The value of a state (state-action pair) means the expected return for starting in that state (state-action pair), and then acting according to a particular policy henceforth. Value functions are used in almost every RL algorithm.

The on-policy value function $V^\pi(s)$ gives the expected return for starting in state s and always acting according to policy π:

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s\right] \tag{3.53}$$

The optimal policy, π^*, has a corresponding state-value function, $V^*(s)$, and vice-versa, the optimal state-value function can be defined as:

$$V^*(s) = \max_{\pi} V^\pi(s) \quad \forall s \in \mathcal{S} \tag{3.54}$$

Similarly, the on-policy action-value function $Q^\pi(s)$ gives the expected return for

starting in state s, taking an arbitrary action a (which may not come from the policy), and then always acting according to policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s, a_0 = a\right] \tag{3.55}$$

The optimal policy, π^*, has a corresponding action-value function, $Q^*(s,a)$, and vice-versa, the optimal action-value function can be defined as:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \forall s \in \mathcal{S},\ \forall a \in \mathcal{A} \tag{3.56}$$

From the above definitions, there are straightforward but key connections between the value function and action-value functions:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a)\right] \tag{3.57}$$

$$V^*(s) = \max_{a} Q^*(s,a) \tag{3.58}$$

Another observation is the relation between the optimal policy when in state s with the optimal action-value function Q^*. If we know Q^*, the we can directly obtain the optimal action, $a^*(s)$, via

$$a^*(s) = \arg\max_{a} Q^*(s,a) \tag{3.59}$$

Bellman Equations

All of the above defined value functions obey self-consistency equations called Bellman equations, for which the basic idea is that the value of a particular starting point is the reward you expect to get from being there, plus the value of wherever you land next. For

the on-policy value functions, this is written as:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim\mathcal{P}}} [r(s,a) + \gamma V^{\pi}(s')] \tag{3.60}$$

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{s'\sim\mathcal{P}} \Big[r(s,a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi} \big[Q^{\pi}(s',a') \big] \Big] \tag{3.61}$$

where $s' \sim \mathcal{P}$ is shorthand for $s' \sim \mathcal{P}(\cdot|s,a)$, indicating that the next state s' is sampled from the environment transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are the maximum over actions, indicating that whenever the agent gets to choose its action, it has to pick whichever action leads to the highest value:

$$V^{*}(s) = \max_{a} \mathop{\mathbb{E}}_{s'\sim\mathcal{P}} [r(s,a) + \gamma V^{*}(s')] \tag{3.62}$$

$$Q^{*}(s,a) = \mathop{\mathbb{E}}_{s'\sim\mathcal{P}} \Big[r(s,a) + \gamma \max_{a'} Q^{*}(s',a') \Big] \tag{3.63}$$

Advantage Functions

The advantage function $A^{\pi}(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming the agent acts according to π forever after. Using the above definitions of the value and action-value functions, this can be formally defined as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{3.64}$$

The advantage function is crucial to policy gradient methods to solve RL problems, discussed in the next section.

3.6.2 Solving the RL Problem

Now that we have introduced the basic terminology and functions of a reinforcement learning problem, the question becomes: what do we need to learn? Generally reinforcement learning is divided into two sub-categories: model-based RL and model-free RL. Model-based algorithms allow the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. However, a ground-truth model of the environment is usually not available to the agent, which it has to learn purely from experience. A big challenge in this aspect is that bias in the learned model can be exploited by the agent, resulting in an agent that performs well with respect to the learned model, but sub-optimally in the real environment.

Model-free RL in contrast does not make any effort to learn the underlying dynamics that govern how the agent interacts with the environment. Model-free RL algorithms forego the potential gains in sample efficiency from using a model, but are typically easier to implement and tune, and are thus more popular. Such algorithms directly estimate the optimal policy or value functions.

We will specifically consider policy optimization algorithms for the rest of this section, which represent a policy explicitly as $\pi_\theta(a|s)$. The parameters θ are optimized either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is typically performed on-policy, which means that each update uses only the data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in determining the policy update. Towards discussing the current state-of-the-art, Proximal Policy Optimization (PPO) [22], we first review policy gradients, and its predecessor, Trust Region Policy Optimization (TRPO) [94].

Policy Gradient Methods

Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. A commonly used gradient estimator has the form

$$\hat{g} = \hat{\mathbb{E}}_t \Big[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \Big] \tag{3.65}$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t as in [95]. The expectation indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization. Implementations that use automatic differentiation software, such as OpenAI Baselines [96] which use TensorFlow, work by constructing an objective function whose gradient is the policy gradient estimator. The estimator $\hat{g}$ is thus obtained by differentiating the objective:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \Big[\log \pi_\theta(a_t|s_t) \hat{A}_t \Big] \tag{3.66}$$

However, while it may seem appealing to perform multiple steps of optimization on this loss function using the same trajectory, empirically doing so often leads to destructively large policy updates.

Trust Region Methods

To mitigate these possibly destructively large policy updates, the authors of TRPO [94] propose maximizing a "surrogate" objective function subject to a constraint on the size

of the policy update:

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \tag{3.67}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t \Big[\mathrm{KL}\big[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\big] \Big] \leq \delta \tag{3.68}$$

where δ is a hyperparameter for the upper bound of the KL divergence between the old and the updated policy, and θ_{old} is the vector of policy parameters before the update. This constrained optimization problem is solved using a conjugate gradient followed by a line search [94].

Proximal Policy Optimization

PPO [22] is motivated by the same question as TRPO: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse? Where TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to the old. PPO methods are significantly simpler to implement, have fewer hyperparameters to tune, and empirically seem to perform at least as well as TRPO. Open AI states that PPO has become their default reinforcement learning algorithm because of its ease of use and good performance.

To recap from TRPO, if we let $r_t(\theta)$ denote the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.69}$$

so that $r_t(\theta_{old}) = 1$, then TRPO maximizes the "surrogate" objective:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{3.70}$$

The superscript CPI refers to the conservative policy iteration [97] where this objective was proposed. Without a constraint, maximization of L^{CPI} would lead to an excessively large policy update; hence, PPO proposes a modification to the objective to penalize deviations of $r_t(\theta)$ from 1 as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \right) \right] \tag{3.71}$$

where ϵ is a hyperparameter ($\epsilon = 0.2$ in the implementation in this thesis). The first term is simply L^{CPI}, and the second term modifies the surrogate objective by clipping the probability ratio, removing the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Taking the minimum of the clipped and unclipped objective results in a final objective with a lower bound on the unclipped objective.

PPO is an actor-critic algorithm and thus uses an explicit representation of both the policy (actor) and value (critic) functions, implemented as Neural Networks. To compute the variance-reduced advantage function estimator using a learned state-value function $V(s)$, we must create a loss function that combines the policy surrogate and a value function error term. This objective is additionally augmented by adding an entropy bonus to ensure sufficient exploration [22, 98]. These three terms are combined into the following objective, to be (approximately) maximized at each iteration (after each policy rollout):

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{3.72}$$

where c_1, c_2 are hyperparameters, S denotes an entropy bonus, and $L_t^{VF}(\theta)$ is a squared-error loss:

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{targ}\right)^2 \tag{3.73}$$

3.6.3 Imitation Learning

Reinforcement learning offers a formulation for control skills acquisition, relying on learning from trial and error. Such methods typically require a significant amount of system interaction time (high sample complexity), and carefully designed well-shaped reward structure is necessary to guide the search of optimal policies. This can be especially non-trivial in complex scenarios.

The goal of imitation learning, on the other hand, is to learn a policy that reproduces the behavior of experts who demonstrate how to perform a desired task. Suppose that the behavior of the expert demonstrator (or the learner itself) can be observed as a trajectory τ as we defined for reinforcement learning:

$$\tau = (s_0, a_0, ..., s_T, a_T) \tag{3.74}$$

where a_i is the action taken by the agent in state s_i. One way to obtain a policy that reproduces the demonstrated behavior is to learn a policy that directly maps from the state to the action/trajectory. If we have a dataset of demonstrated trajectories of state-action pairs $\mathcal{D} = \{\tau_0, ..., \tau_N\}$, we can directly compute a mapping from states to control inputs as

$$a_t = \pi(s_t) \tag{3.75}$$

This kind of policy can be obtained through a standard supervised learning method. Learning a policy that directly maps from the state to the control input is often referred to as Behavioral Cloning (BC) [99].

In Chapter 8 we will use this classical behavioral cloning approach to imitation learning, and define the following objective function that seeks to minimize the error between an expert action and the maximum likelihood estimate action from the current policy π_θ:

$$L^{BC}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\Big(a_i^* - \arg\max_{a_i} \pi_\theta(a_i|s_i)\Big)^2 \tag{3.76}$$

for expert demonstration state-action pairs $\{s_i, a_i^*\}_{i=1}^N$, where $\arg\max_{a_i} \pi_\theta(a_i|s_i)$ is the maximum likelihood estimate action a_i for state s_i using policy π_θ.

Chapter 4

Design and Evaluation of Skating Motions for a Dexterous Quadruped

This chapter describes skating locomotion for RoboSimian, a quadruped robot with dexterous limbs. The emphasis is on design of omnidirectional motion primitives and quantification of resulting speed and accuracy when traversing different types of smooth but potentially non-flat terrain. In particular, we study trade-offs between using four-wheeled versus three-wheeled skating maneuvers. In four-wheeled skating, motions have the benefit of symmetry, so that errors due to wheel slip should theoretically cancel out on average. Three-wheeled skating, by contrast, introduces significantly more asymmetry in configuration and contact force distribution over time; however, it has the advantage of guaranteeing continuous ground contact for all skates when terrain has bumps or other curvature. We present simulation results quantifying errors for each approach, for various terrains. These results allow the possibility to tune motions to reduce biases and variability in motion, which are primarily due to accelerations as locomotion begins.

4.1 Introduction and Motivation

While previous work has been done on skating robots, it has largely focused on robot designs and/or relatively simple, symmetric motions, as discussed in Section 2.2. Here, more diverse, omnidirectional motions for skating are explored, to enable dual-locomotion strategies: rolling rapidly on relatively smooth terrain and switching to quadruped walking for rough or intermittent terrain. The emphases are on design parameterization and data-based evaluation, with the purposes of examining uncertainty propagation and enabling later algorithms (i.e., learning) for parameter tuning and optimization.

As discussed in Section 2.1, while force and torque sensing are available at the most distal tip of each of RoboSimian's limbs, there is currently no sensing available to sense contact of the skate with the ground. This eliminates the practical use of force feedback during skating, and as such, the focus is on execution of "open-loop" motion primitives, in which encoder feedback is available and used to control desired joint angle trajectories, while the joint trajectories themselves are not updated or modified during execution based on feedback. This opens the question of what planned trajectory length is suitable, to balance the reduced cost of replanning against the increasing variability in end state as such motion sequences become longer.

Another objective we have is to explore design and resulting performance of motion primitives for mildly stochastic terrain. Speed and accuracy of locomotion are our primary concerns. Friction variations, non-zero slopes, and bumps along terrain all clearly affect skating locomotion, and we will quantify some of these effects. We hypothesize that using three wheels instead of four may increase accuracy, since wheels will stay in contact across mild terrain curvatures. However, such motions may also introduce additional asymmetries and/or uncertainties in contact forces that may not produce a net benefit in performance, motivating this study.

The rest of this chapter is organized as follows. Section 4.2 describes our modeling framework for planning skate motions and subsequent joint trajectories of the robot. Sec. 4.3 outlines the design of simple, forward-facing motion primitives to skate in a straight line, for both pairwise skating with all four limbs and three-wheeled skating for more consistent ground contact at all skates. Section 4.4 presents design and results for speed and accuracy of more complex motions primitives and stochastic terrain situations, based on Monte Carlo simulations on stochastically generated terrain using Klamp't (Kris' Locomotion and Manipulation Planning Toolbox) [100]. Finally, conclusions and discussion of applications toward both high-level motion planning and low-level learning for our work are presented in Section 4.5.

4.2 Modeling

Planning effective, feasible skating motions for RoboSimian involves two complementary problems. The motions of the skates must enable generation of required ground reaction forces without excessive slipping, to move the robot as desired, and solutions for inverse kinematics must be tractable, smooth, and within the dynamic velocity and acceleration limits of the joint actuators of the robot.

4.2.1 Skating Kinematics and Dynamics

Each "skate" we use for RoboSimian is a simple, ideal wheel under standard constraint assumptions. For motion planning, we assume that no slip occurs along the axis of the wheel, and that the wheel can roll freely perpendicular to the no-slip axis. We additionally assume that the wheel can rotate freely about a point contact with the ground. In the no-slip direction, however, achievable lateral forces are limited by friction, so that in reality, $|F_x| \leq \mu F_z$, where μ is the coefficient of friction between the ground and the

wheel, and F_z is the normal force at the ground contact. Figure 4.1 illustrates the no-slip and free-wheeling directions; free rotation about the contact point corresponds to angle ϕ. We also assume no force or torque can occur along the freely-moving DOFs.

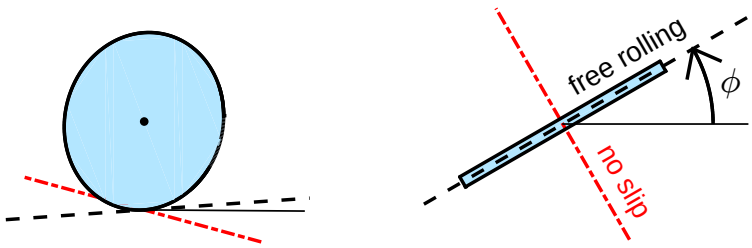

Figure 4.1: Simple wheel, with constraint axes: angled and overhead views.

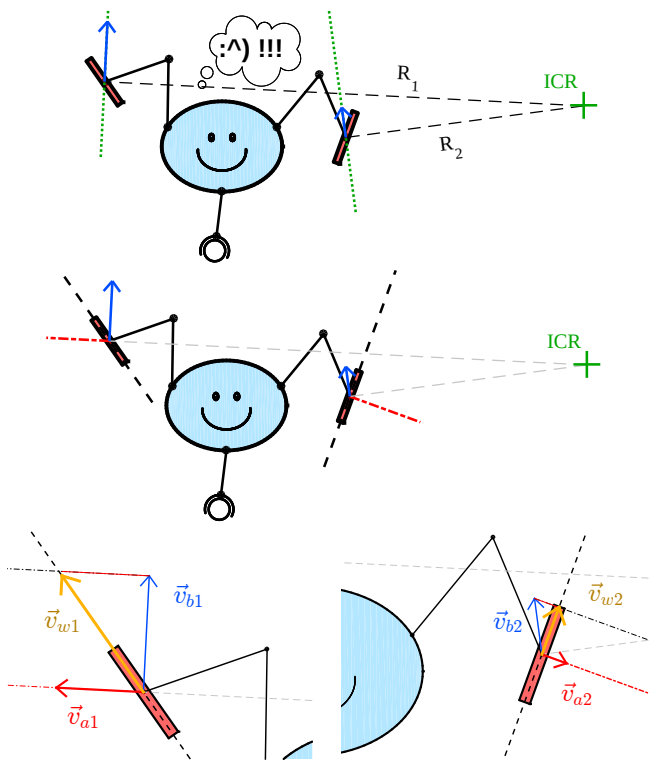

Figure 4.2: Overview of planning for skating kinematics.

Given these constraints, one can plan co-compatible motions for the body (globally) and for the skates (with respect to the local body coordinate frame). As with many locomotion problems, planning can be achieved by first reasoning about allowable com-

binations of body and end effector motions that satisfy the required constraints. Then, enforcing those particular motions must result in the desired forces, at least to the extent that modeling assumptions for the dynamics are correct. Zero-moment point (ZMP) planning is a classic example of this basic approach, in which desired forces are achieved not directly through force feedback, but indirectly through enforcing joint-level trajectories compatible with desired ground reaction forces [101, 102].

In Figure 4.2, the top image shows an overhead view of a particular pose and robot configuration. This robot example has two skates and a third spherical wheel (with no constraints on direction of free rolling), for simplicity. A desired instantaneous center of rotation (ICR) is defined for the robot body path. The ICR in turn defines the relative direction and magnitude of any point in the local body reference frame: direction will be perpendicular to the radial line, R_n, from the ICR to that point, and magnitude must be proportional to this distance. At the midpoint of each skate, the total velocity will be the sum of this local "body velocity", shown as blue vectors in this figure, plus the relative velocity of the skate with respect to the local body reference frame.

Let us also assume that we pick a particular direction of motion for each skate, depicted by the red dashed lines in the middle image. Referring to the lower left image of Fig. 4.2, in order for both the simple kinematic constraints shown in Fig. 4.1 and desired absolute body trajectory motions to be achieved, the sum of the velocity due to body motion (blue vector) plus the relative velocity of skate with respect to body (red vector) must align along the "free rolling" direction of the skate (yellow vector), since no motion (i.e., slip) is assumed perpendicular to this direction.

During execution of the resulting motion plan, forces are generated only along the no slip axis, with zero force along the free rolling direction. So long as friction limits along the no slip axis are not exceeded, the planned constraints should theoretically be obeyed, resulting in desired body motion.

Of course, real-world skates do not have point contacts, friction on terrain is not always well-characterized or consistent, and even guarantees of expected z-directed contact forces, F_z, required for corresponding lateral friction forces, cannot necessarily be made for quadruped skating, once terrain becomes uneven. Coping with such non-ideal situations is a large focus of this chapter.

4.2.2 Constraints Particular to RoboSimian

As mentioned in Section 2.1, we mount a single wheel on the most distal link, or "forearm", of the robot. Along each limb, there are six actuated joints, more proximal to the body, which set the six degree-of-freedom (DOF) pose, i.e., location and orientation (x, y, z, roll, pitch, and yaw), of each skate. A seventh motor is located at the very end of each leg, as depicted in Fig. 2.1.

Two issues make motion planning for RoboSimian particularly challenging. First, the inverse kinematics (IK) to set the 6-DOF pose of the skate require choosing from among one of eight IK families, each analogous to a choice of "elbow bending direction" for each of three elbows on a limb [31]. Providing guarantees of smoothness requires precalculation of IK solutions across a region as well as compromises between ideal theoretical contact locations and achievable solutions for our particular robot. In particular, achieving exact symmetry in end effector locations is either non-trivial or not achievable, for many desired skate configurations.

A second constraint in planning is a lack of force sensing, to detect contact and loading of the skate on the ground, as discussed in Section 4.1. More specifically, six-DOF force and torque sensing is available at the most distal "foot" contact of each limb, but not for the attached skate for the real robot. Our simulations in Klamp't obviously provide such contact force data, but we use this only in post-processing, to identify and quantify

slip.

4.3 Design of Simple Skating Motions

This section describes three-wheeled and four-wheeled skating along a straight, forward-facing path. Note throughout that skates are numbered 1 through 4, with limb/skate 1 at the front right of the robot, and numbering continuing sequentially going clockwise when viewing the robot from above. For example, Figure 4.3 shows RoboSimian facing to the left, within Klamp't. At right, the skate in midair is on limb 4. The position of each skate (x_{ee}, y_{ee}) is measured with respect to the body frame rather than the world frame.

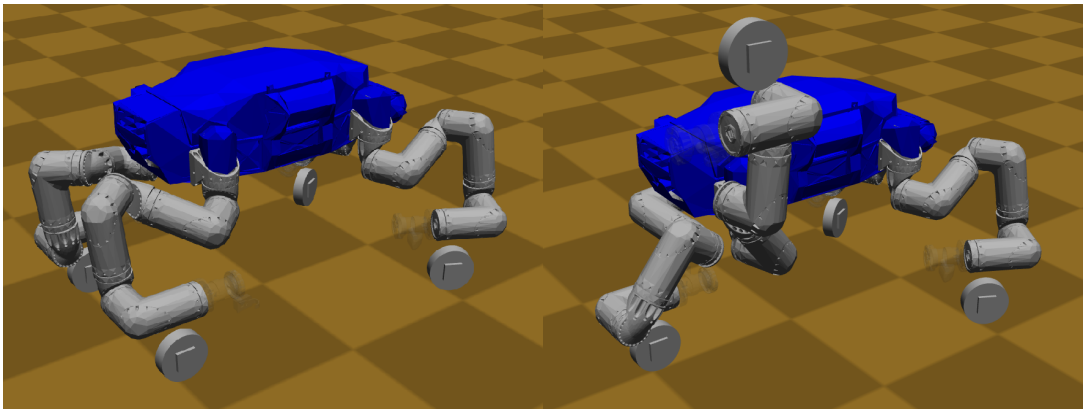

Figure 4.3: RoboSimian on 4 (left) and 3 skates (right), simulated in Klamp't.

For all trajectories presented in this chapter, acceleration and deceleration of planned body motions follow triangular waveforms, as illustrated in Figure 4.4.

4.3.1 Skating with Four Wheels

Skating with four limbs/wheels exploits symmetry, canceling out some effects of slip. For straight-line body trajectories, the left versus right limbs move symmetrically with

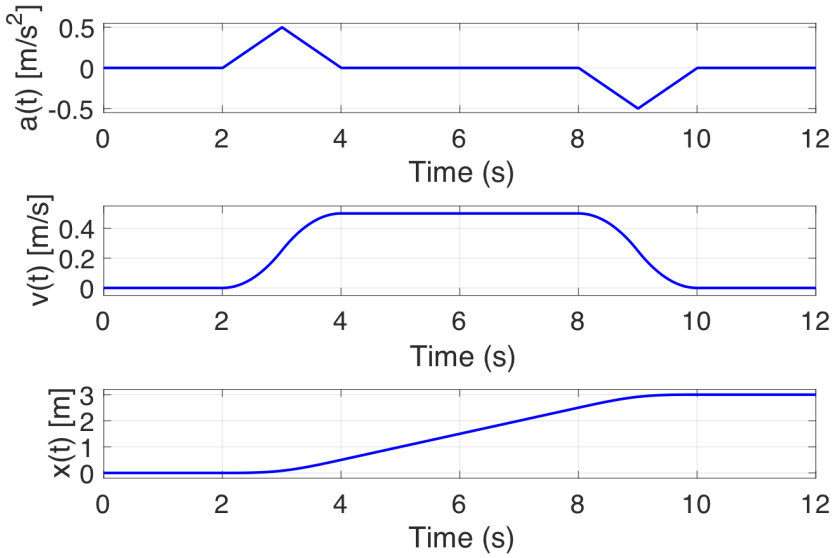

Figure 4.4: Acceleration ($a(t)$), velocity ($v(t)$), and position ($x(t)$) profiles for planned robot body motions.

respect to one another, so that limbs 1 and 4 (front two limbs) have a phase difference of π with respect to one another (as do limbs 2 and 3). There is a phase difference of $\pi/2$ between the front and rear limbs, so that one set of skates will have maximum angle of attack, ϕ, and (correspondingly) maximum force generation potential, exactly when the other pair is aligned with the free-rolling axis pointing forward, with no ability to generate a net ground reaction force in this forward direction. This effect is especially important during acceleration and deceleration of the robot.

Figure 4.5 shows $y_{ee}(t)$ and $\phi(t)$ trajectories for a desired robot body motion going straight, in the (forward-facing) local $+x$ direction. Skates do not move in the x direction, with respect to the robot body frame, as the relative skate motions are perpendicular to the robot's direction of motion.

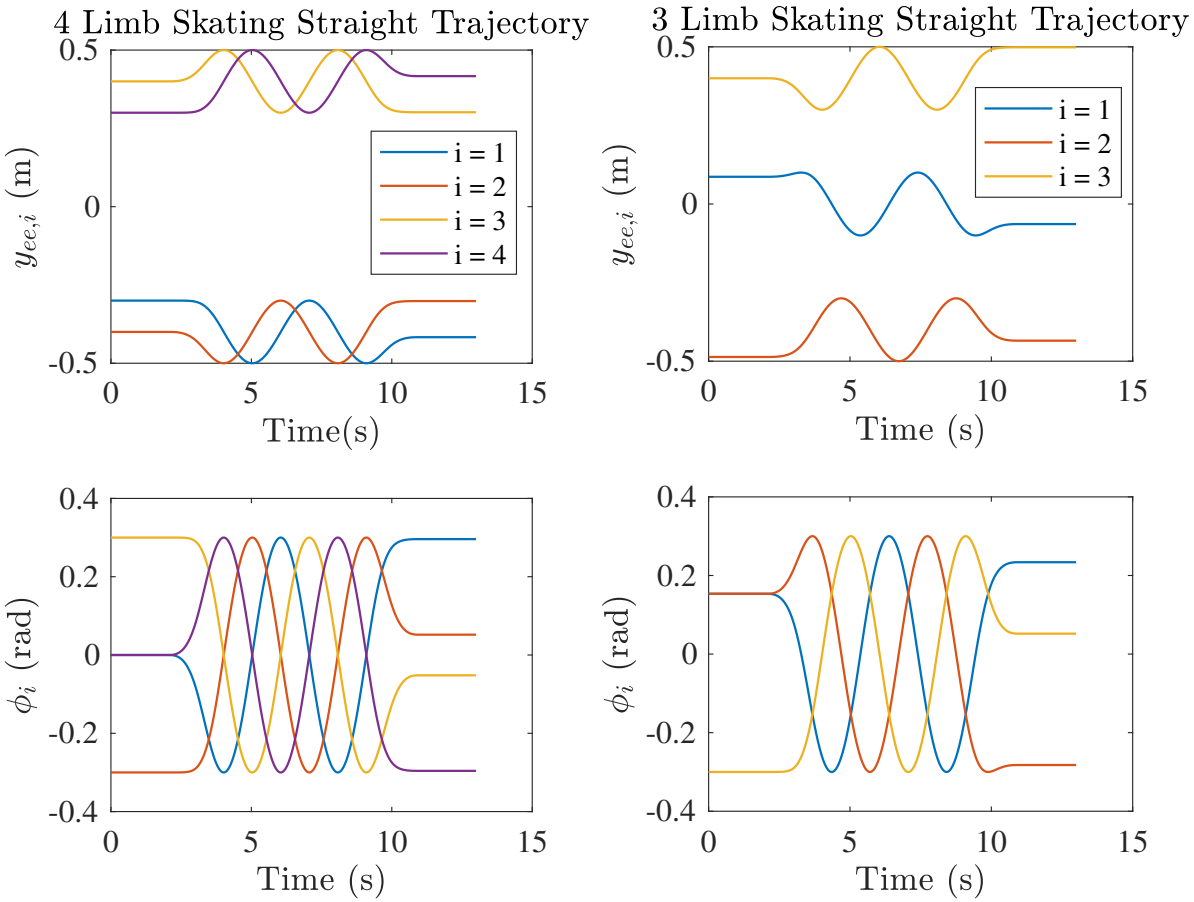

Figure 4.5: At left, local skate end effector positions and angles ϕ for a simple, straight body trajectory with 4 skates. The front two skates are out of phase by $\pi/2$ with the back skates, and due to symmetry, are out of phase by π with each other to ensure minimal slip and maximum propulsion forces. At right, analogous trajectories with 3 skates. Each skate is out of phase by $2\pi/3$, with this configuration being the optimal for straight ahead acceleration from stand still, as determined from simulations associated with Figures 4.6 and 4.7.

4.3.2 Skating with Three Wheels

Using only three wheels to skate has two motivations. It leaves the fourth limb free for manipulation tasks, and – more importantly – three skates provide more reliable ground contact on curved terrain profiles than four.

In planning, we assume that limb 4 (front left limb) is up and "out of the way" of any skating motions of the remaining three limbs, and primarily plan for trajectories

with limb 1 placed nearly front and center of RoboSimian, as shown in Fig. 4.3. We also designed and tested other three-skate configurations, which require additional planning and offsets, but focus on this quasi-symmetric case, which improves performance.

The basic idea in three-skate planning is to use equal phase increments between skates (i.e., "three-phase" waveforms). The asymmetry of using three skates has a significant drawback, however: it increases vulnerability to yaw due to slip during accelerations of the body. Toward mitigating this problem, we study three approaches, described below.

Careful selection of the phase offset of each skate

We start with initial phase offsets of 0 for skate 1, $2\pi/3$ for skate 2, and $4\pi/3$ for skate 3. Corresponding simulations in Klamp't show significant slip during the initial acceleration phase of the skating motion, causing unwanted yaw (to the left) in the actual body trajectory. Most of this yaw is due to wheel slip at skate 2 (i.e., right rear wheel).

We study the effects of changing the phase offset at start-up by performing simulations of a 10-meter straight trajectory on a flat terrain with friction coeffcicient $\mu = 0.3$. Phase offsets are parameterized as $(0+\gamma,\ 2\pi/3+\gamma,\ 4\pi/3+\gamma)$ for skates 1, 2, 3, respectively, with γ set to one particular value on each trial. Resulting body coordinate trajectories are shown in Figure 4.6. The top plot shows the (x, y) body trajectory, while the lower plot shows yaw angle over time. Figure 4.7 shows results for a similar case, except reversing the sign on the 120° phase offsets among skates, to use offsets of $(0+\gamma,\ 4\pi/3+\gamma,\ 2\pi/3+\gamma)$.

The two choices of symmetric phase offset for three-wheel skating should produce identical steady-state motions, in theory. In simulation, however, each has different (non-ideal) "problems", due to asymmetries now. For Fig. 4.6, using $(\gamma,\ \gamma+120°,\ \gamma+240°)$, the straightest trajectory is for $\gamma = 4\pi/3$; however, total distance traveled (upper subplot) has a 10% error, going only about 9 meters, instead of 10. For a reversed skate phasing

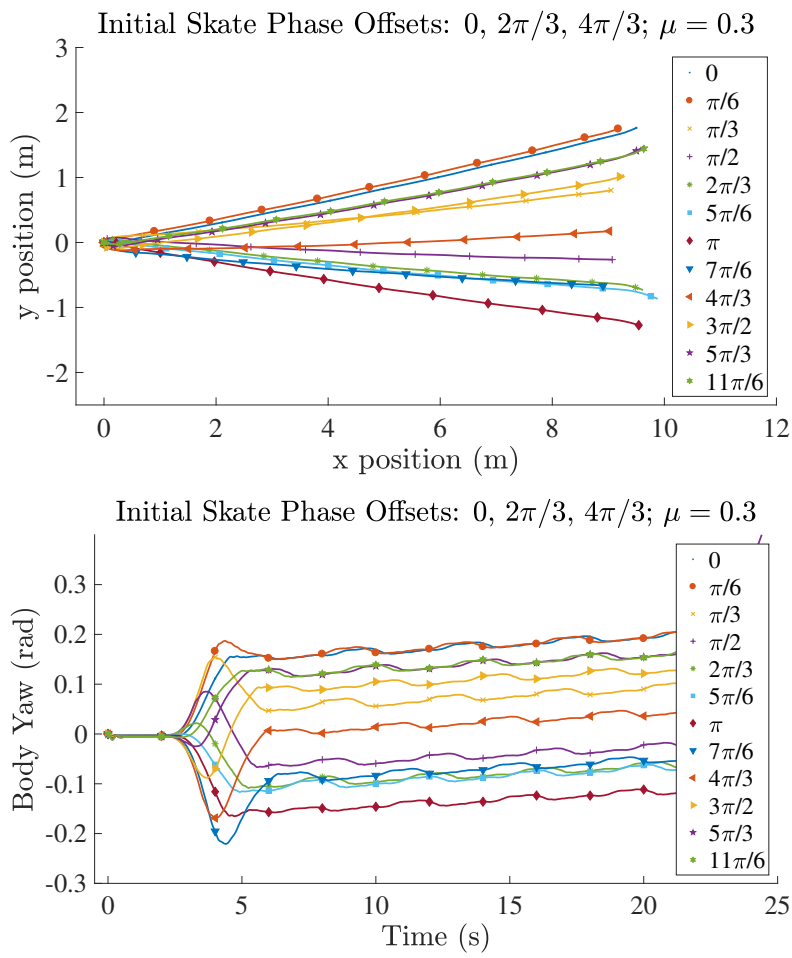

Figure 4.6: Results for a 10 (m) straight trajectory, designed for 3-wheel skating, with peak of $a(t) = 0.5$ (m/s^2) at the midpoint of a 2-second triangular acceleration profile, reaching a steady-state velocity of 0.5 (m/s), with a deceleration over the last 2 seconds of the trajectory. The phase offset, γ, is changed incrementally by $\pi/6$.

of $(\gamma, \gamma - 120°, \gamma - 240°)$, Fig. 4.7 shows a somewhat more significant steady-state rate of undesired body yaw (turning to the left, for each case), but this rate is quite small in both cases, i.e., 0.17 $(°/s)$ vs 0.30 $(°/s)$, which map to circular paths with radii of 170 vs. 95 meters (i.e., very slight turning, indeed).

More importantly, the steady-state velocity for the latter case is now much closer to its planned value. With $\gamma = 11\pi/6$, total distance traveled is almost exactly the desired distance of 10 meters, as shown on the top subplot of Fig. 4.7, versus the 10% error

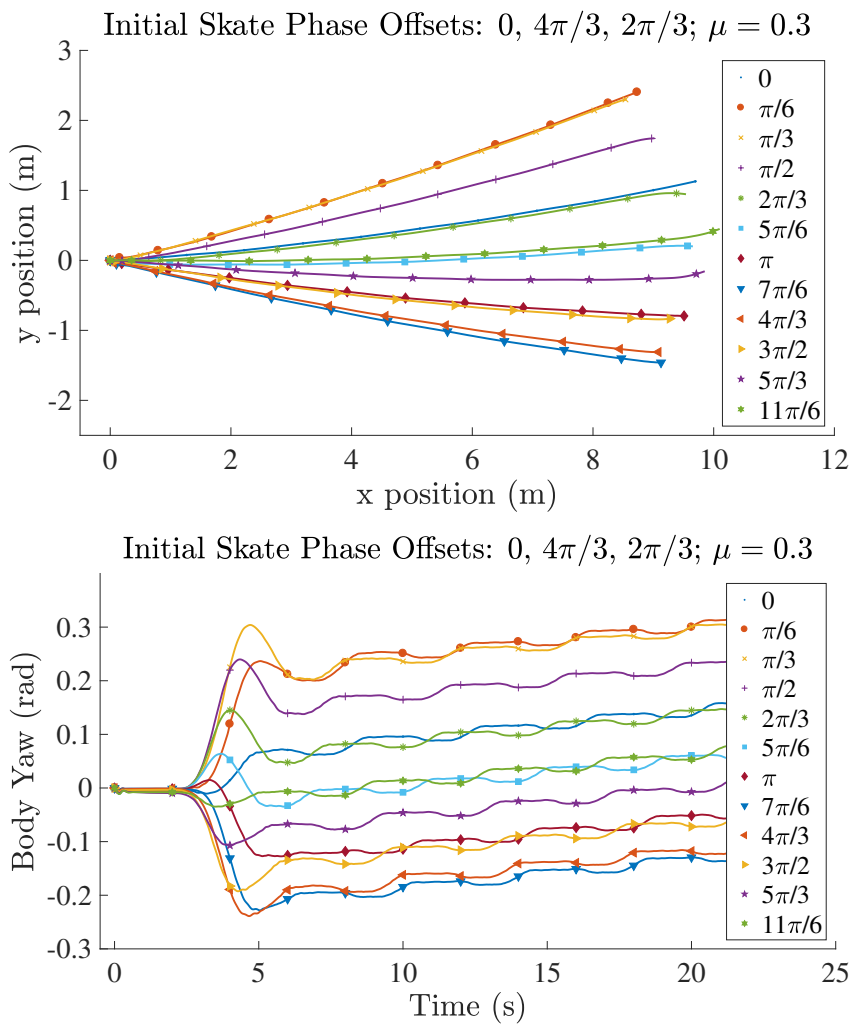

Figure 4.7: Results using the same body trajectory as in Fig. 4.6 and again changing the phase offset for each skate incrementally by $\pi/6$, but with the reversed order of phase increment among the three skates.

mentioned above for Fig. 4.6.

For the remaining three-skate trajectories presented in this paper, an initial configuration of $(0, 4\pi/3, 2\pi/3) + 11\pi/6$ is used as an "optimal offset", since it minimizes error in Fig. 4.7. The righthand trajectories in Figure 4.5 in fact present ϕ and y for this particular parameterization. Note that during acceleration (near $t = 4$ s), the front skate ($i = 1$) is the one pointing most nearly "straight forward", while the two back wheels, though still asymmetric with respect to one another, push in opposite directions, for an

effect that somewhat mimics the symmetry of four limb skating.

Slowing down the initial acceleration rate from a stand still

We compared simulation results for both the initial acceleration profile shown in Fig. 4.4 and for a similar profile that instead uses half the peak acceleration, spread over twice the time (four seconds instead of two), to reach the same steady-state velocity of 0.5 (m/s), to compare errors in body trajectory during acceleration. The slower acceleration reduces required lateral forces (reducing chance of slip) but also clearly increases required time to go a particular distance, so that we anticipate a classic trade-off between speed and accuracy, in tracking planned body motions.

Pushing off with the free limb

While push-off, analogous to skate boarding, was successful, it demonstrated no performance benefits (in speed or accuracy). See the video supplement for an example of this motion.

4.4 Design for Variable Terrain Challenges

Toward more generalized motion planning, the forward-rolling motion primitives discussed in the previous section are now augmented by including diagonal trajectories, curved paths, variable friction, and mild bumps on terrain. In this section we design (and simulate with Klamp't) such skating motions for both three- and four-limb skating. Videos[1][2] illustrating the motions described below, among others, can be found associated with the publication on which this chapter is based on [103].

[1] https://youtu.be/kLEUTthCQt4
[2] https://youtu.be/bD5w1cxFFqE

4.4.1 20 (m) Straight Trajectory

Straight trajectories were already discussed in Sections 4.3.1 and 4.3.2, where results were presented assuming a friction coefficient of $\mu = 0.3$ between each skate and the ground. In Figure 4.8, we present additional results for a range of values of μ, with a steady-state velocity of 0.5 (m/s). Errors are quite small for symmetric, 4-wheel skating, in the leftmost subplots. Even for the lowest friction tested ($\mu = 0.1$), error in end state is only about 0.5 meters, or 2.5% of the total 20-meter desired distance.

For 3-limb skating, results are similar for all $\mu \geq 0.3$, with desired forward velocity and a net yaw rate of about 0.26 $^\circ/s$ (i.e., turning radius of about 110 meters). Slip is quite noticeable for $\mu = 0.1$, however, with speed reduced to about 80% of the desired rate and a tighter (unplanned) turning radius (about 40 meters) due to slip.

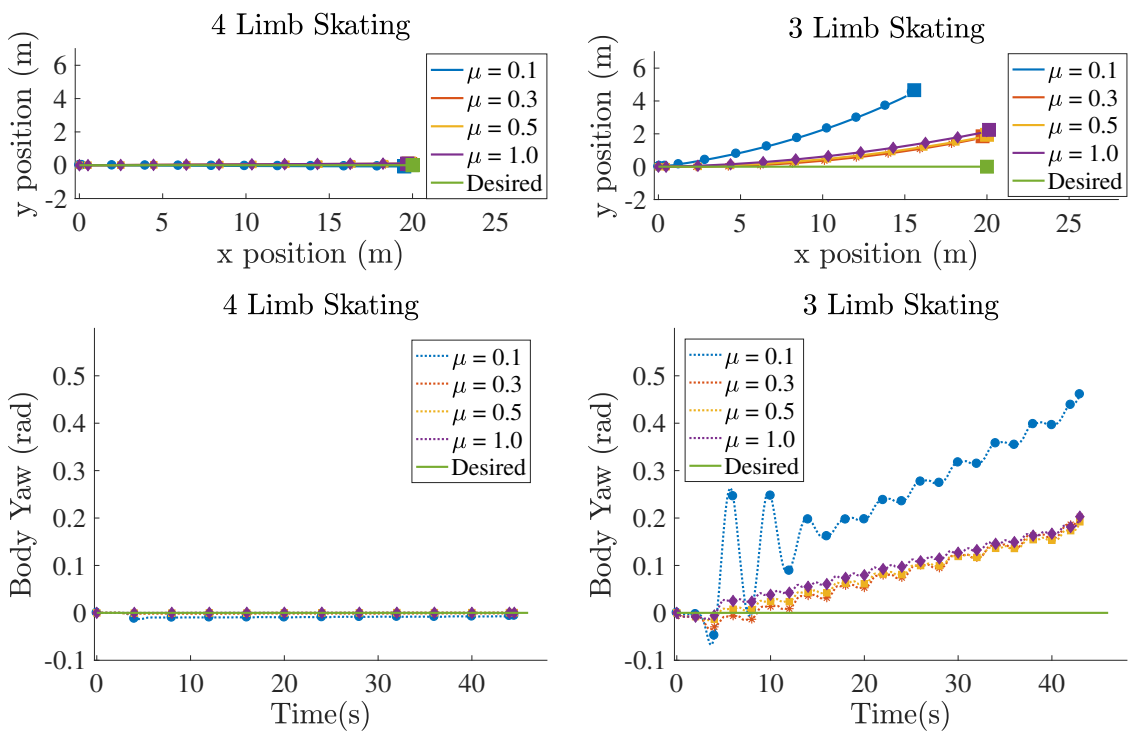

Figure 4.8: Tracking a 20 (m) straight ahead trajectory with four (left) or three (right) skates over varying μ.

4.4.2 10 (m) Diagonal Trajectory

Here, motions are planned to move the body along a straight line, oriented 30° to the left (nautical "port" side) with respect to the previous, forward-facing trajectories, to test omnidirectional locomotion. Relative motions of the skates (with respect to the body) are now generalized to travel perpendicular to the planned body motion, which is an effective parameterization across all tested motions. To better visualize this, refer to the lower left subplot of Fig. 4.2: the relative motion of the skate in this example is also planned to be exactly perpendicular to the absolute body velocity at the ground contact for this skate. This rule for planning relative skate directions is used across all examples within this chapter. Rules for planning relative phase between skates and for setting skate angles remain unchanged from Sec. 4.3.1.

Figure 4.9 shows results. As there is inherently less symmetry between limbs when the robot travels diagonally, there are greater tracking errors, as expected. Compared with the forward-facing trajectories (at left, in Fig. 4.8), four-limb skating does show more sensitivity to μ, but it is still a mild effect. Three-limb skating is significantly more sensitive to changes in μ; however, there is a relatively smooth mapping between μ and the resulting offset in yaw angle, which occurs primarily during acceleration at the start. A bifurcation in behavior occurs between friction values of 0.1 and 0.3, i.e., slip is dramatic for $\mu = 0.1$.

4.4.3 2 (m) Radius Circle Trajectory

For circular body trajectories, we again choose relative skate motions perpendicular to absolute velocity of the body-fixed coordinate frame at the ground contact of each skate. For curved trajectories, this means each skate oscillates back and forth on the line extending radially from the desired ICR to the ground contact (see bottom left of

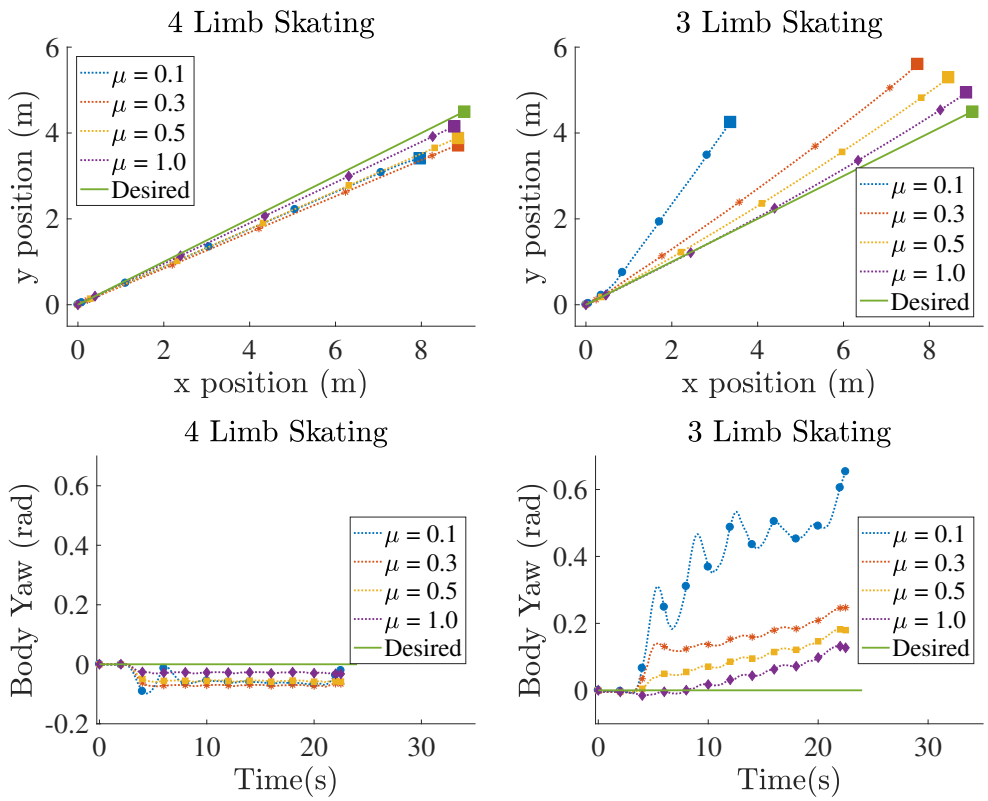

Figure 4.9: Tracking a $\pi/6$ diagonal trajectory from straight ahead with four (left) or three (right) skates over varying μ.

Fig. 4.2).

Simulations in Klamp't track a circle trajectory of radius 2 (m). For $\mu = 0.1$, three-limb skating cannot complete the trajectory, which is shown in the accompanying video. For the other coefficients of friction, three-limb skating performs quite well, and actually close to as well as four-limb skating, unlike for the previous motions.

4.4.4 Monte Carlo Simulations over Rough Terrain

Over flat terrain, regardless of the coefficient of friction, four-wheel skating outperforms the accuracy of three-wheel skating, albeit sometimes only by a small margin. This is no longer the case on bumpy terrain.

We created "smooth" wavy terrains via spline fits of randomly generated heights

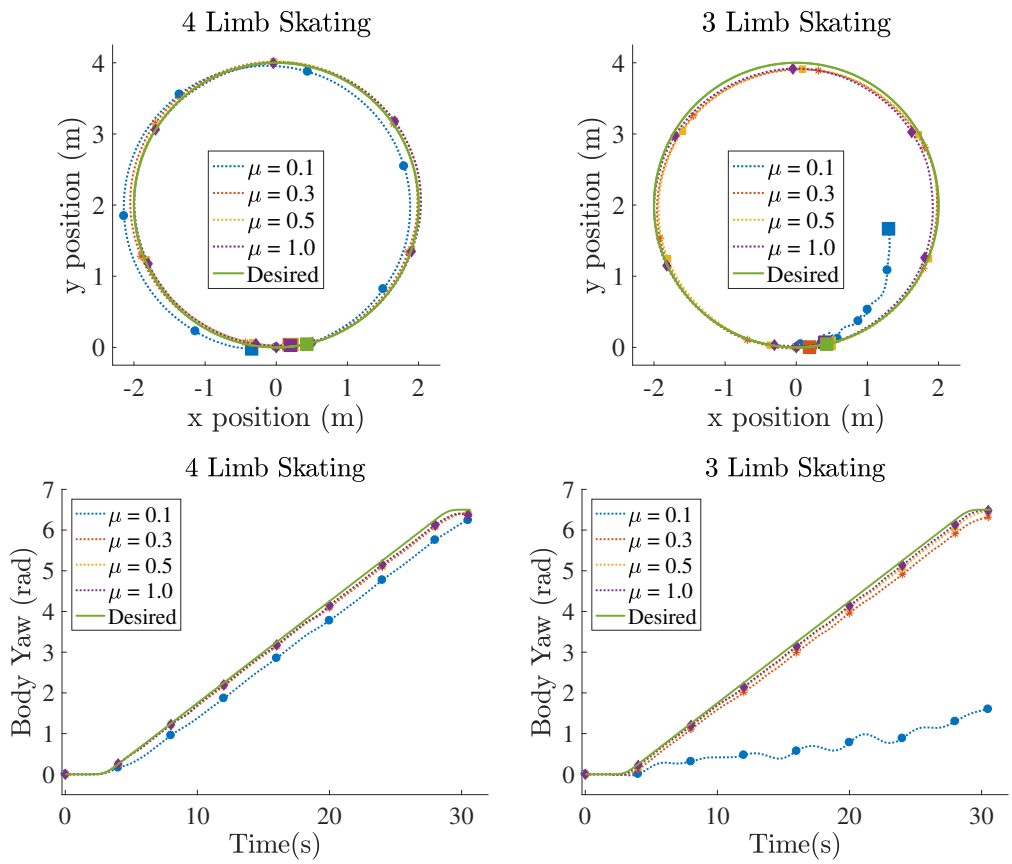

Figure 4.10: Tracking a circular trajectory with radius 2 (m) with four (left) or three (right) skates over varying μ.

across a 2D grid. We present results here for terrains with peak bump heights of 0.1 (m) and 0.2 (m), for the 20 (m) straight trajectory previously studied on flat terrain (in Sec. 4.4.1) for 20 open-loop trials with each of four values of μ, randomizing initial position and orientation on terrain for each trial.

Figures 4.11 and 4.12 show the final (x, y) coordinates of the robot for each of the trials. Over the 0.1 (m) maximum height rough terrain, save for the $\mu = 0.1$ case, three limb skating always travels close to the desired 20-m distance, though the body trajectory may be offset by up to 10°, due to initial yaw perturbations. This significantly outperforms four limb skating, which has a much broader distribution of end states, corresponding to intermittent slip throughout motion and not just during initial acceleration (as for

three-skate motions).

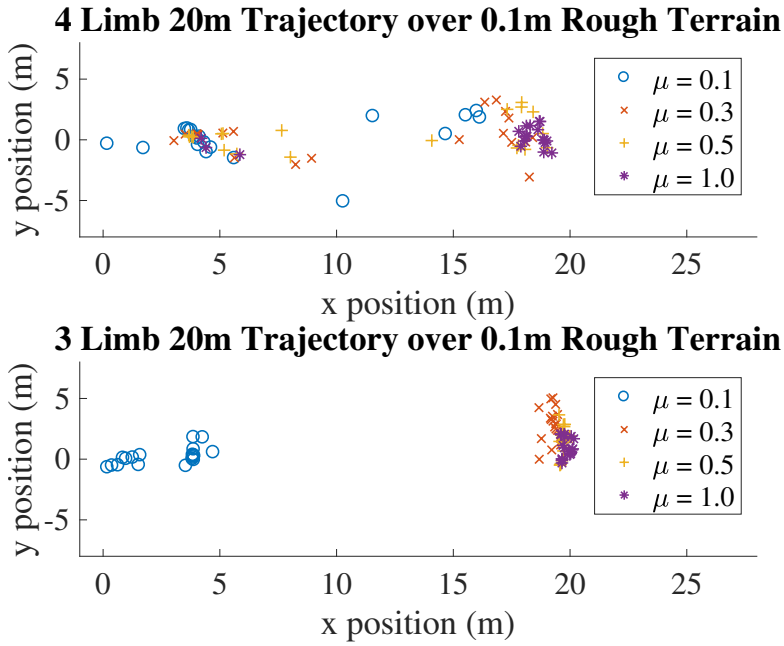

Figure 4.11: Monte Carlo simulations of a 20 (m) straight trajectory over "smooth" rough terrain that varies randomly uniformly over 0.1 (m).

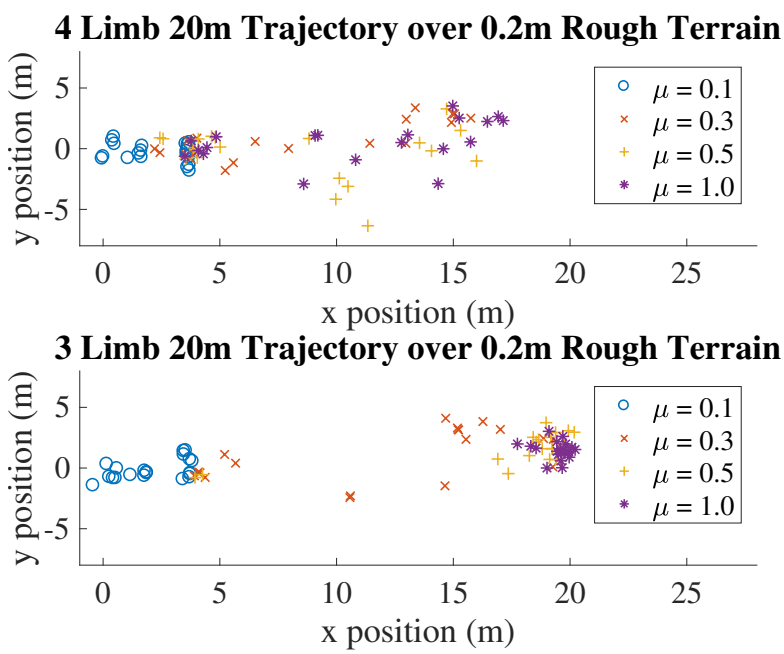

Figure 4.12: Monte Carlo simulations of a 20 (m) straight trajectory over "smooth" rough terrain that varies randomly uniformly over 0.2 (m).

The discrepancy between the two primitives becomes even more significant over the rougher terrain with 0.2 (m) bumps. Three-limb skating is still reliable for $\mu = 1$, with some bifuractions in behavior appearing at lower frictions. With four skates, however, behavior is unreliable regardless of the value of μ. RoboSimian consistently has trouble going up the small wavy hills and sometimes even rolls backwards, getting stuck in a trough where it often repeatedly slips and rolls/slides back down. The supplementary video shows an example of this for four limb skating.

4.5 Conclusions and Discussion

Agile wheeled locomotion is one route toward increasing energy efficiency, stability, and speed simultaneously on mild terrain for limbed robots also capable of rough terrain walking and/or dexterous manipulation tasks. In this chapter, we outline design of and present performance analyses for a small family of motion primitives, exploring trade-offs between speed and accuracy as a function of (stochastic) terrain properties, toward facilitating both high-level planning structures and low-level learning algorithms.

For high-level planning, we envision use of skating primitives within a larger framework, to avoid obstacles, meet desired waypoints, and reach particular end destinations on mild terrain. Real-world locomotion will involve terrain with variable contact friction and unevenness, along with inherent time delays for sensor estimates of 6-DOF body pose based on processed vision and lidar information. One practical planning strategy would therefore be to employ a sequence of well-characterized but stochastic open-loop dynamic maneuvers [102], using piece-wise composition with a receding horizon approach for replanning to cope with uncertainty [104]. Such a strategy is in the spirit of sequential composition via a funneling approach [105]; i.e., with a set of individual controllers that can be nested, with the variability in end state of one sequence easily within the basin

of attraction of the next sequential controlled action.

A second purpose of this work is to explore uncertainty propagation of various skating primitives, toward selecting and parameterizing low-level motion primitives for optimization through learning algorithms. In particular, we highlight that skating with three wheels (1) produces greater yaw disturbances and (2) results in more continuity and repeatability as parameterizations and/or noise vary. For example, Figures 4.6, 4.7 and 4.9 show yaw increasing smoothly as μ goes down, for $\mu > 0.1$, and Figures 4.11 and 4.12 show more repeatable traction (i.e., clustering of end state) for three-wheel skating, while results using four wheels yield more bifurcations and fewer examples of "smooth variations" in behavior. Correspondingly, we anticipate that three-wheeled skating can be used more effectively within gradient-based learning algorithms, to deliberately exploit slip for more dynamic, unactuated and agile maneuvers.

Finally, we note that the fidelity of dynamic simulations is an open question and ongoing research challenge in itself; real robot locomotion will vary somewhat from simulation predictions. In our experience, simulations provide an important step in efficiently identifying and pre-tuning behaviors that are later verified and/or tweaked on real hardware, also enabling safe and systematic development of dynamic motions. We highlight that ensuring robustness to parameter variability within simulation, which is a central goal in this work, typically improves real-world robustness, as well.

Chapter 5

Nonintuitive Optima for Dynamic Locomotion: The Acrollbot

This chapter explores locally-optimal, efficient locomotion of a two-link planar robot balancing on a single, unactuated wheel. Because this model is essentially an acrobot mounted on a passive wheel, we name this model the acrollbot. By actuating an internal degree of freedom, the model can indirectly produce ground reaction forces yielding net accelerations and decelerations, to achieve locomotion. As with bipedal robot locomotion, this toy system is particularly challenging to control due to the need to balance continuously while controlling forward locomotion speed. However, unlike typical legged or rolling locomotion solutions, it is not immediately obvious how best to exploit actuation, internal reconfigurations, and motions to produce and control forward velocity along the ground, providing a useful benchmarking system for exploring optimization techniques. We use a direct collocation optimization framework to study this toy system, both to achieve a range of feasible locomotion solutions for nonintuitive dynamic robot models, and to investigate optimization of physical robot parameterizations, in the sense of improving locomotion efficiency. The framework and example presented

throughout are designed with an aim toward bridging the gap between non-intuitive, data-driven optimization and model-based methods for design and control of underactuated and dynamically-stable locomotion.

5.1 Introduction and Motivation

Agile legged and/or wheel-legged robots can potentially exploit inverted pendulum dynamics for balance and locomotion. In this chapter, we study locally optimal solutions for an underactuated multi-link rolling inverted pendulum, toward finding non-intuitive strategies for balance and forward propulsion.

Our goal is to develop tools that can optimize both the mechanical design of a highly dynamic robot and optimal motion trajectories, simultaneously. We hypothesize that simple models, when chosen with care, are extremely useful for identifying fundamental principles, and also that many interesting phenomena for producing balance and locomotion (in particular, through creative use of internal degrees of freedom) may yet remain undiscovered. We focus on a simple yet unusual model, which includes the challenge of underactuated balance and potential use of internal degrees of freedom without the additional complication of (dissipative/stabilizing) collision events, toward investigating novel dynamic phenomena for locomotion.

While work has been done in recent years to optimize locomotion control for underactuated systems, we are particularly interested in methods with potential to explore and reveal non-intuitive solutions, to use computing power to improve performance. Many such examples were discussed in Chapter 1, ranging from switching between low level controllers to produce robust, agile bounding gaits [106–108], to automated gait generation through optimization frameworks [13, 19, 76, 109, 110]. Deep learning methods [23, 111–113] have also been widely used for robot motion planning. A particularly

popular Google DeepMind AI locomotion video associated with [21] highlights the emergence of a range of locomotion behaviors for various systems through deep reinforcement learning. These animations include examples on seemingly non-optimal – or at least non-intuitive – motions. For example, humanoid biped upper-body limbs sometimes flail dramatically, although they never contact the environment. While these results are definitely intriguing, there is arguably a practical desire to know whether such non-intuitive motions can be verified to have a specific (and significant) beneficial physical function or not. As a preview of our results, our work produces similarly "suspicious-looking" motions, which can be verified to assist in balance during efficient locomotion for our simple, planar model.

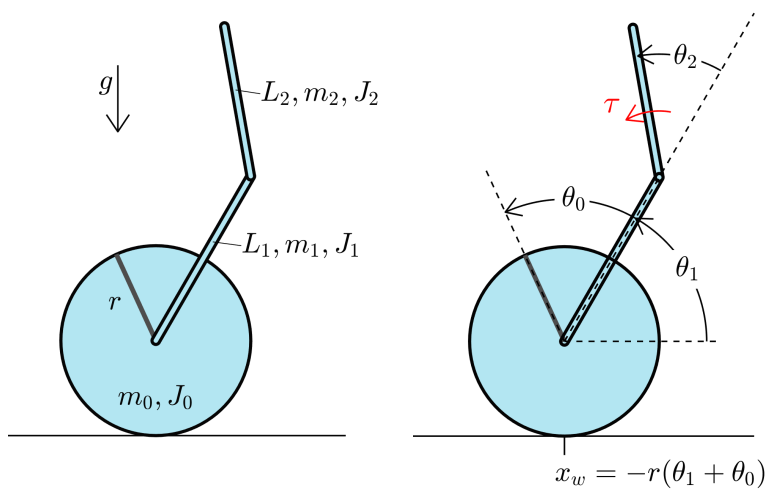

Figure 5.1: The Acrollbot: a two-link acrobot on a passive wheel.

The system we study here, the acrollbot, shown in Figure 5.1, resembles the classic two-link acrobot model [114] with a (passive) fixed pivot, except that it is mounted on a passive wheel that can roll on the ground. Conceptually, the model is similar to a person on a unicycle, except that the unicycle includes no pedals or other direct actuation at its axis, and locomotion control must be achieved indirectly, by moving various joints to exploit passive damping at the axle and/or forces generated indirectly at the ground

contact due to the no-slip assumption for the wheel. The original inspiration for the acrollbot was to create the simplest model to represent RoboSimian balancing on its rear passive wheels in the sagittal plane, with only internal degrees of freedom available for locomotion.

The rest of this chapter is organized as follows. Section 5.2 presents the Acrollbot system and its equations of motion. Section 5.3 presents the optimization framework used to develop motions for the system, as well as to find optimal parameters. Resulting motions, system parameters, comparisons to "normal" systems, and parameter sensitivity for various "optimal" systems are discussed in Section 5.4, and Section 5.5 concludes this chapter with comments on the accompanying video, and a discussion of ongoing and future work.

5.2 Acrollbot System

5.2.1 Parameters and Coordinates

The base of an acrobot is attached to a wheel at a "free axle", meaning there is no direct actuation between the wheel and the L_1 link. The only torque input, τ, is applied directly between the L_1 and L_2 links, as shown at right in Fig. 5.1. The wheel is modeled as a solid disk with mass m_0, radius r, and inertia $J_0 = \frac{1}{2}m_0 r^2$. Link 1 is modeled as a slender rod with mass m_1, length L_1 (center of mass [COM] at $L_{c1} = \frac{1}{2}L_1$), and inertia about COM of $J_1 = \frac{1}{12}m_1 L_1^2$. Link 2 is modeled as a slender rod with mass m_2, length L_2 (COM at $L_{c2} = \frac{1}{2}L_2$), and inertia $J_2 = \frac{1}{12}m_2 L_2^2$. θ_1 is the counter-clockwise (CCW) angle measured from the positive x-axis to link 1, analogous to link 1 of an acrobot. θ_0 is the wheel angle, measured CCW relative to θ_1, and θ_2 is the angle of the upper (second/last) link, similar to link 2 of an acrobot, measured CCW relative to θ_1.

5.2.2 Equations of Motion

The equations of motion for the system can be written in matrix form as:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \Xi \tag{5.1}$$

where $q = [\theta_0, \theta_1, \theta_2]^T$. Nonconservative torques on the right-hand side include viscous (linear) damping at the wheel axle, proportional to relative wheel velocity ($\dot{\theta}_0$), and a torque applied between links 1 and 2, u; i.e., Ξ is (3x1) with:

$$\Xi_1 = -f_0\dot{\theta}_0, \qquad \Xi_2 = 0, \qquad \Xi_3 = u \tag{5.2}$$

The torques due to centrifugal and Coriolis effects are:

$$C(q,\dot{q})\dot{q} = [C_1, C_2, C_3]^{\mathsf{T}} \tag{5.3}$$

where:

$$\begin{aligned}
C_1 &= rm_2L_{c2}\cos(\theta_1+\theta_2)\left(\dot{\theta}_1+\dot{\theta}_2\right)^2 + r\left(m_1L_{c1}+m_2L_1\right)\cos(\theta_1)\,\dot{\theta}_1^2 \\
C_2 &= C_1 - m_2L_1L_{c2}\sin(\theta_2)\left(2\dot{\theta}_1\dot{\theta}_2+\dot{\theta}_2^2\right) \\
C_3 &= m_2L_1L_{c2}\sin(\theta_2)\,\dot{\theta}_1^2
\end{aligned}$$

Similarly, the gravity-dependent terms are $G(q) = [G_1, G_2, G_3]^{\mathsf{T}}$, where

$$\begin{aligned}
G_1 &= 0 \\
G_2 &= G_3 + \left(m_1L_{c1}+m_2L_1\right)g\cos(\theta_1) \\
G_3 &= m_2L_{c2}g\cos(\theta_1+\theta_2)
\end{aligned}$$

Finally, the configuration-dependent terms in the inertia matrix $D(q)$ are given below, where D_{mn} corresponds to element $D(m,n)$ of the matrix:

$$
\begin{aligned}
D_{11} &= J_0 + (m_0 + m_1 + m_2)\, r^2 \\
D_{12} &= D_{11} + D_{13} + L_1 m_2 r \sin(\theta_1) + L_{c1} m_1 r \sin(\theta_1) \\
D_{13} &= L_{c2} m_2 r \sin(\theta_1 + \theta_2) \\
D_{21} &= D_{12} \\
D_{22} &= D_{11} + 2D_{13} + J_1 + J_2 + L_1^2 m_2 + L_{c1}^2 m_1 + L_{c2}^2 m_2 + ... \\
&\quad 2L_1 L_{c2} m_2 \cos(\theta_2) + 2\,(L_1 m_2 + L_{c1} m_1)\, r \sin(\theta_1) \\
D_{23} &= J_2 + L_{c2}^2 m_2 + L_1 L_{c2} m_2 \cos(\theta_2) + L_{c2} m_2 r \sin(\theta_1 + \theta_2) \\
D_{31} &= D_{13} \\
D_{32} &= D_{23} \\
D_{33} &= m_2 L_{c2}^2 + J_2.
\end{aligned}
$$

5.2.3 Note on the Linearized System

We can linearize this system about some equilibrium configuration, in which all angular velocities are zero and the center of mass of the system is aligned at the same x coordinate as the center of the wheel. We define the states of the system as:

$$X = \left[\theta_0, \theta_1, \theta_2, \dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2\right]^{\mathsf{T}} \tag{5.4}$$

and X_{eq} is some valid equilibrium configuration in which the system can balance (with zero velocity for each DOF). One example is a fully upright configuration:

$$X_{eq} = \left[q_{eq}^{\mathsf{T}}, \dot{q}_{eq}^{\mathsf{T}}\right]^{\mathsf{T}} = \left[0, \frac{\pi}{2}, 0, 0, 0, 0\right]^{\mathsf{T}} \tag{5.5}$$

Note that any choice is valid for the relative wheel angle, θ_0. Toward deriving state space equations, we can first write the linearized equations of motion in the following form:

$$M\ddot{q} + \beta\dot{q} + Kq = T = [0, 0, u]^{\mathsf{T}} \tag{5.6}$$

where $M = D(q_{eq})$, i.e., the inertia matrix $D(q)$ evaluated at the equilibrium configuration of interest. Terms due to $C(q, \dot{q})\dot{q}$ drop out when linearizing about zero velocity, so that the matrix β contains only the viscous damping term at the wheel, while K is the Jacobian of $G(q)$ evaluated at $q = q_{eq}$. For X_{eq} as defined in Equation 5.5:

$$\beta = \begin{bmatrix} f_0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.7}$$

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & (-m_2 g L_1 - m_1 L_{c1} g - m_2 g L_{c2}) & (-m_2 g L_{c2}) \\ 0 & (-m_2 g L_{c2}) & (-m_2 g L_{c2}) \end{bmatrix} \tag{5.8}$$

For example, linearizing about the upright position and using $g = 9.8$ (m/s^2), $m_0 = m_1 = m_2 = 1$ (kg), $L_1 = L_2 = 1$ (m), $r = 0.2$ (m), $J_0 = \frac{1}{2}m_0 r^2$, $J_1 = J_2 = \frac{1}{12}m_1 L_1^2$ and $f_0 = 0.3$ (Nm/(rad/s)), the linearized state space equation matrices become (i.e., for

$\dot{X} = AX + Bu$):

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -71.736 & 19.992 & -7.152 & 0 & 0 \\ 0 & 24.696 & -14.112 & 1.872 & 0 & 0 \\ 0 & -32.928 & 48.216 & -3.096 & 0 & 0 \end{bmatrix} \tag{5.9}$$

$$B = \begin{bmatrix} 0, 0, 0, 10.32, -5.52, 15.36 \end{bmatrix}^{\mathsf{T}} \tag{5.10}$$

The closed-loop poles of the linearized system can be set arbitrarily to regulate the system about some desired equilibrium point (e.g., via pole placement, or LQR control) if and only if the controllability matrix has full rank, i.e., rank 6 for our system (with 3 angles and 3 angular velocities as states). With non-zero wheel damping (f_0) in β, the controllability matrix for A and B has rank 5. When $f_0 = 0$, the fourth column of A becomes $[1, 0, 0, 0, 0, 0]^{\mathsf{T}}$ (with B and the remainder of A remaining unchanged), and the controllability matrix for the linearized system has rank 4.

5.2.4 Controllability

As summarized in Section 5.2.3 above, the full linearized system without friction is not controllable. However, with some damping friction at the passive wheel, the controllability matrix of the linearized system becomes rank 5. While this does not allow the system to be stabilized about some arbitrary x_w position for the wheel, an LQR controller using the 5 other states of the system allows it to stabilize about the vertical equilibrium position.

Additionally, using partial feedback linearization (PFL), it is also possible to control

the system using only the acrobot-specific states, namely $[\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]$. These states can be used to stabilize the system about the vertical equilibrium point, when damping is present at the wheel. Specifically, rank 5 of the controllability matrix here corresponds to a lack of ability to control the overall x_0 wheel location of the system, although velocity of the wheel can be driven to zero. With no damping or friction at the wheel, the linearized system can still be stabilized in an upright configuration, although now neither x_0 nor dx_0/dt can be regulated at (i.e., driven to) zero.

5.3 Optimization Framework

5.3.1 Problem Setup

As discussed in Section 3.5, we discretize the full nonlinear system and use direct collocation along with backward Euler integration as suggested in [75] to generate trajectories for the rolling motions. The problem is formulated as

$$\text{find} \quad q, \dot{q}, u \quad \text{at discrete timesteps } k = 1...N \tag{5.11}$$

$$\text{subject to} \quad \text{minimize cost } J \tag{5.12}$$

$$D(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = Hu \tag{5.13}$$

$$\phi(q, \dot{q}, u) = 0, \;\; \psi(q, \dot{q}, u) \geq 0 \tag{5.14}$$

where $q \in \mathbb{R}^n$ is the vector of generalized coordinates, $D(q) \in \mathbb{R}^{n \times n}$ is the mass inertia matrix, $C(q, \dot{q}) \in \mathbb{R}^{n \times n}$ represents the centrifugal and Coriolis forces, $G \in \mathbb{R}^n$ contains gravitational forces, and $H \in \mathbb{R}^n$ is the input mapping. The cost function J is discussed in the next subsection. $\phi(q)$ and $\psi(q)$ are vectors of constraints: we assume no slipping of the wheel, and constraints are imposed to ensure that the normal reaction force at the

point of contact is always positive.

The initial and end states ($[\theta_1, \theta_2, \dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2]^\mathsf{T}$) of any motion are also constrained to both be some (identical) vertical equilibrium point, to enable either a single motion or a repeated limit-cycle behavior, and we solve for the base of the wheel to move a certain distance d_{stride} in a fixed time interval through our framework. However, we note that the same framework can also find trajectories for arbitrary starting and ending system positions and velocities, if other limit cycles or continuing motions (when the system is not starting or ending at rest) are desired.

5.3.2 Cost Function

We set up our cost function, J, as the Cost of Transport (COT) using the integral of the absolute power:

$$J = \frac{\int_0^t \sum_{i=1}^{N} \sqrt{L + \epsilon}\, dt}{(m_0 + m_1 + m_2) g d_{stride}} \tag{5.15}$$

where

$$L = \left([\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2] U\right)^2 = \left(\dot{\theta}_2 u\right)^2 \tag{5.16}$$

since there is a single input, u; i.e., $U = [0, 0, u]^\mathsf{T}$. We use $\epsilon = 10^{-6}$ as a regularization term to help smooth the cost function as suggested in [115], where N is the number of sample points for the trajectory, and d_{stride} is the horizontal distance traversed by the center of the wheel. The motivation behind creating such a unitless COT is to allow meaningful comparison between optimization results from using different parameters for distance, time, system masses, and temporal discretizations. Without such a COT as we have defined, it would not be fair to compare costs if one system has more mass or has

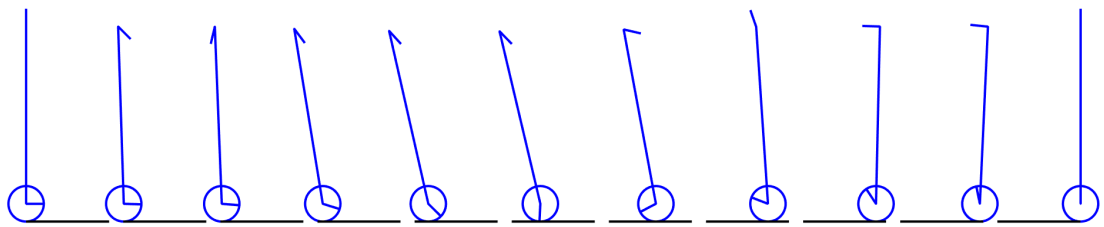

Figure 5.2: Snapshots of the two-link acrollbot during locomotion.

more sample points, for example.

5.3.3 Open System Parameters

An open question driving this current framework is: what effects and consequences does changing system parameters have on metrics such as efficiency, robustness, and agility? Motivated by this general question, with respect to the acrollbot system, we leave not only the states/trajectories of each joint as open parameters in the system, but also the system parameters themselves: $r, L_1, L_2, m_0, m_1, m_2$, with $r, L_1, L_2 \in [0.01, 1]$ meters, and $m_0, m_1, m_2 \in [0.01, 1]$ kilograms.

Leaving all of these parameters open and finding an optimal solution with respect to energy, for example, returns not only an optimal trajectory, but also the optimal robot design to use for that motion.

5.4 Results

Figure 5.2 shows snapshots of a typical solution found for optimal robot design and corresponding locomotion. Here, nearly all mass is concentrated as a small L_2 link. Unlike most inverted pendulum systems, such as a Segway or cart-pendulum system, the center of mass (COM) does not lean forward as the system accelerates to the right. Instead,

the mass lags behind the wheel contact, essentially driving the system forward much like an off-balance ladder would slide on the ground. Stability is maintained through an unanticipated vertical oscillation of the top-most mass of link 2. We have simulated various solutions using MATLAB's ode45, and the oscillation stabilizes the system using a similar phenomenon to that used in Kapitza's pendulum [116]; i.e., vertical oscillation of a mass stabilizes an unstable pendulum. In our case, this oscillation can also be exploited to control acceleration or deceleration, depending on the offset angle of oscillation, and we used our optimal solutions to develop generalized control algorithms for locomotion, which are illustrated briefly within videos[1][2] associated with the publication on which this chapter is based on [117].

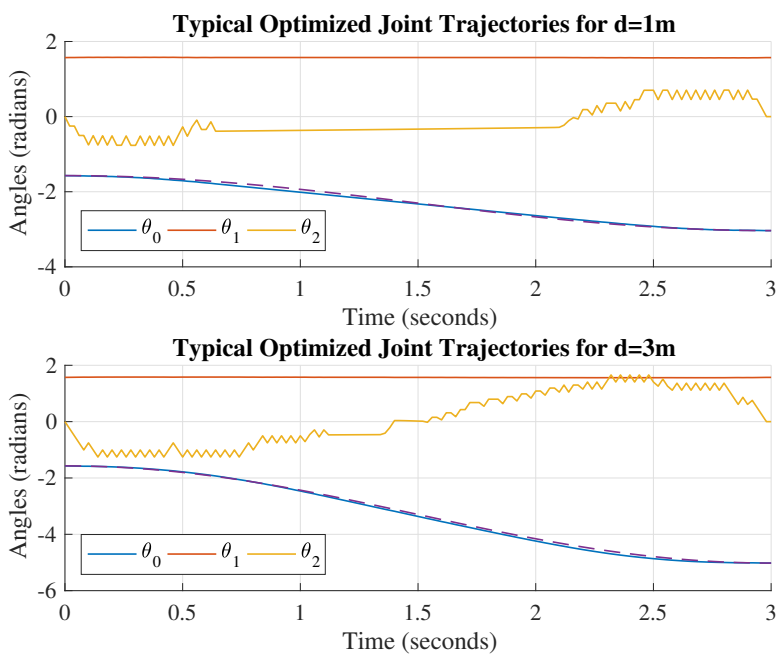

Figure 5.3: Optimal joint trajectories to move the acrollbot system 1.0 meters (at top, COT=0.0284) vs 3.0 meters (bottom, COT=0.0529) in 3 seconds. θ_1 remains nearly constant (about a vertical angle of $\frac{\pi}{2}$). A dashed, purple line, which represents a half-cosine waveform, is overlaid on the solid trajectory for θ_0, to show the smoothness of overall wheel motion along the ground.

[1] `https://youtu.be/C9-P72kI8xI`
[2] `https://youtu.be/PnD7-FHSfZ4`

For most locally-optimal solutions, L_2 is set near its minimal allowed length, with nearly all mass concentrated at this link. Figure 5.3 shows typical behaviors seen in these optimal trajectories. Of note are oscillations, which suspiciously seem to correspond to the knot points of our direct collocation solution. L_2 oscillates during both the initial acceleration period and final deceleration, and the overall motion of the wheel is rather smooth. (We overlay cosine waves on θ_0, the wheel angle, in this figure, for comparison.) After much investigation, oscillations are a repeatable phenomenon and can take on a range of frequencies. We have tested high-resolution simulations (ode45 set with 1E-8 tolerances) of feedback laws based on these motions, which stabilize and locomote the system, as expected.

5.4.1 Optimization Parameter "Typical" Results

Shown in Figure 5.4 is a sample "optimal" trajectory for each of the joints of the system, along with the input torques at link 2 that result in the system's motion. With masses and lengths constrained as described in Sec. 5.3.3 [$r, L_1, L_2, m_0, m_1, m_2 \in [0.01, 1]$, SI units], the framework found the lowest COT for the following "optimal" parameters: $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 0.5393$ (m), $L_1 = 0.8099$ (m), $L_2 = 0.0137$ (m). The framework typically generates trajectories for parameters in approximately these ranges, with minimal mass on the wheel and link 1, maximal mass on link 2, an intermediate value in the range for r, large L_1, and minimal L_2. We observe results in these ranges for any values of distance or time, so the parameters are not tied to specific motion requirements. The trajectory input torques and resulting system reactions always follow the same oscillation patterns at the beginning and end of the trajectory, in the optimal cases. To start the motion, rapid oscillations in torque put energy into the system and provide enough momentum to get the system to roll in

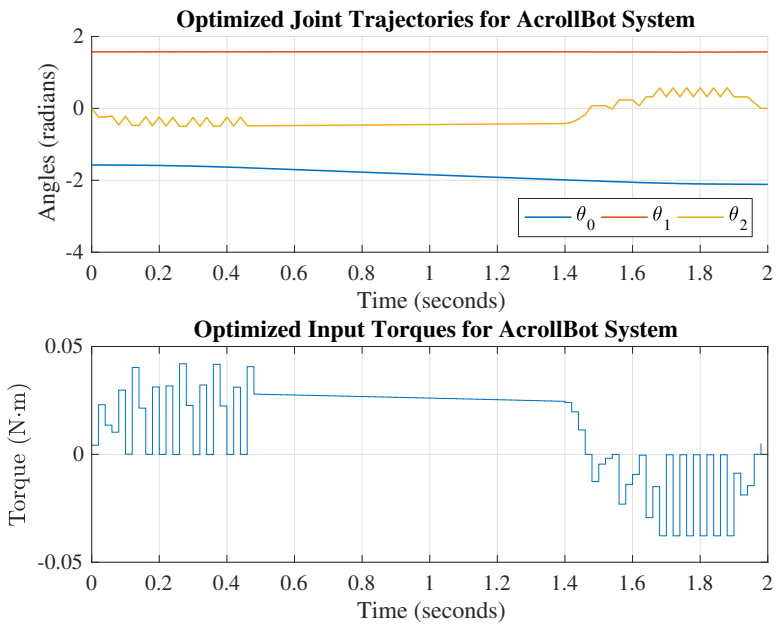

Figure 5.4: Optimal joint trajectories to move the acrollbot system 0.3 meters in 2 seconds, where the corresponding optimal parameters are $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 0.5537$(m), $L_1 = 0.8522$ (m), $L_2 = 0.0123$ (m), as well as the corresponding input torques at link 2 that results in the trajectories. Cost of Transport is 0.0432.

one direction, and the "opposite" reactions cause the system to slow down and stop at the desired distance. Between these beginning and ending oscillations, the torque stays approximately constant, simply transitioning and positioning link 2 from the midpoint of one set of oscillations to the other. The length of this transition period decreases as the desired lateral distance to move increases, placing an upper bound on the furthest distance the system can travel for a given total time interval.

5.4.2 Rare Optima: "Large Wheel, Short Links"

Although the local optima that the framework finds in repeated (and randomized) searches typically results in the ranges as mentioned in the preceding section, very occasionally returned is a lower COT solution consisting of a very large radius wheel $r = 1.0$ (m), with a minimal length L_1 (i.e. 0.0180 (m)) and a lower value L_2 (i.e. 0.1192 (m)),

($L_2 \gg L_1$). The optimal COT of this large-wheel system is typically less than half that of the best, perhaps "more realistic" system previously discussed (0.0197 vs. 0.0432). Mass distribution stays the same for both of these systems. Physically, with such a large wheel and small internal links, there is no longer really a stabilization issue, but the constraints are still there to keep the system balanced. Comparing Figs. 5.4 and 5.5, the large wheel system requires fewer oscillations to start/stop the system, with one main peak torque at the beginning and end of the motion. Magnitudes have also switched signs, though both move the system in the same direction.

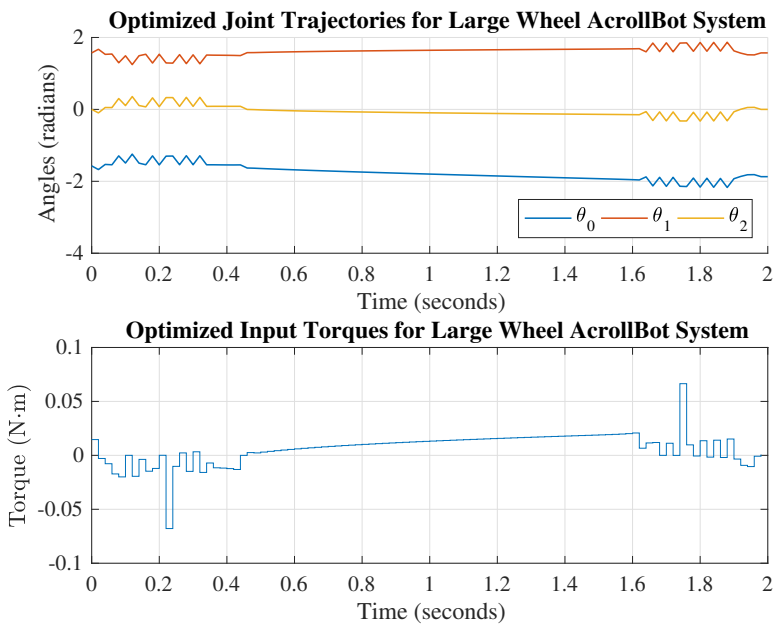

Figure 5.5: Optimal joint trajectories to move the large wheel acrollbot system 0.3 meters in 2 seconds, where the corresponding optimal parameters are $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 1.0$(m), $L_1 = 0.0180$ (m), $L_2 = 0.1192$ (m), as well as the corresponding input torques at link 2 that results in the trajectories. Cost of Transport is 0.0197.

5.4.3 Flywheel

Since our optimizations typically result in system parameters with all of the mass located at m_2 and a minimal length L_2, we consider that perhaps a flywheel at the end

of link 1 might be more optimal than a second link. From a controllability perspective, this would also help with system stabilization, since the flywheel is not constrained to a certain angle in order for the system to balance about an equilibrium point, as in the acrollbot system.

At first, we take the optimal parameters from the acrollbot as found by the optimization framework for m_0, m_1, m_2, r, L_1 and leave the inertia of the flywheel, J_2, as an open parameter. However, the resulting trajectories have a much larger COT, no matter what value J_2 takes. Compared to minimum COT value of 0.0432 for the acrollbot system we found for $T = 2$ (s), $d_{stride} = 0.3$ (m), the flywheel system returns trajectories costing 1.2750, which is very significantly worse.

Leaving all parameters open, however, the framework finds trajectories of lower cost than those of the optimal two-link acrollbot, as shown in Figure 5.6. The same constraints are set as before on the ranges for each parameter, except now we define a characteristic length $\widetilde{L}_2 \in [0.01, 1]$, to describe a flywheel centered at the end of link 1, with $J_2 = \frac{1}{2} m_2 \widetilde{L}_2^2$. The optimal flywheel system parameters are $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 1.0$ (m), $L_1 = 0.0189$ (m), $\widetilde{L}_2 = 0.9999$ (m), with a resulting COT of 0.0132. From a physical standpoint, the optimal system consists of a large-radius (but nearly massless) wheel, with a slightly offset from center, high-inertia flywheel. Leaving the range of $\widetilde{L}_2$ extremely large (on the order of 1E3 or more) does not seem to have a significant effect on the COT, though J_2 does tend toward a large value, given this range. This serves as inspiration for the next subsection of this chapter, where we check the sensitivity of the Cost of Transport to the system's optimal parameters found by the optimization framework. Specifically, how does the COT vary as each of the available parameters is de-tuned from its optimal solution?

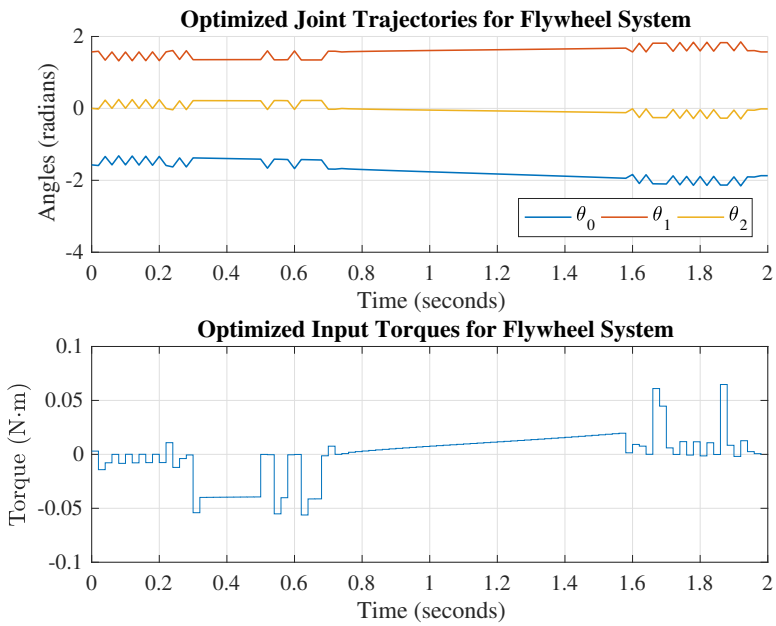

Figure 5.6: Optimal joint trajectories to move flywheel system 0.3 meters in 2 seconds, where the corresponding optimal parameters are $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 1.0$ (m), $L_1 = 0.0189$ (m), $\widetilde{L}_2 = 0.9999$ (m), as well as the corresponding input torques for the flywheel that results in the trajectories. Associated Cost of Transport is 0.0132. Here, L_1 oscillates while holding the flywheel out to the side, to offset the system COM. The directions of rotation for the wheel and flywheel are in the same direction, which defies physical intuition! Additional simulations in MATLAB confirm this phenomenon, however. (See also supplemental video.)

5.4.4 Parameter Sensitivity

Keeping 5 out of the 6 optimal parameters fixed and testing across a range of values in a general neighborhood of the remaining optimal one, we record its effect on the Cost of Transport. Figure 5.7, at top, shows such a (log) plot for the acrollbot model. The log of the COT, rather than just the COT, is plotted for formatting and clarity. The parameter that has the greatest effect on the COT as it varies from its optimal value is L_2, whereas m_2, r, L_1 can vary without having any apparent effect on the COT. A longer L_2 means that the center of mass of the link is farther from the joint, so it is reasonable that it takes larger torques to control its movement. m_0 and m_1, which are always optimized to

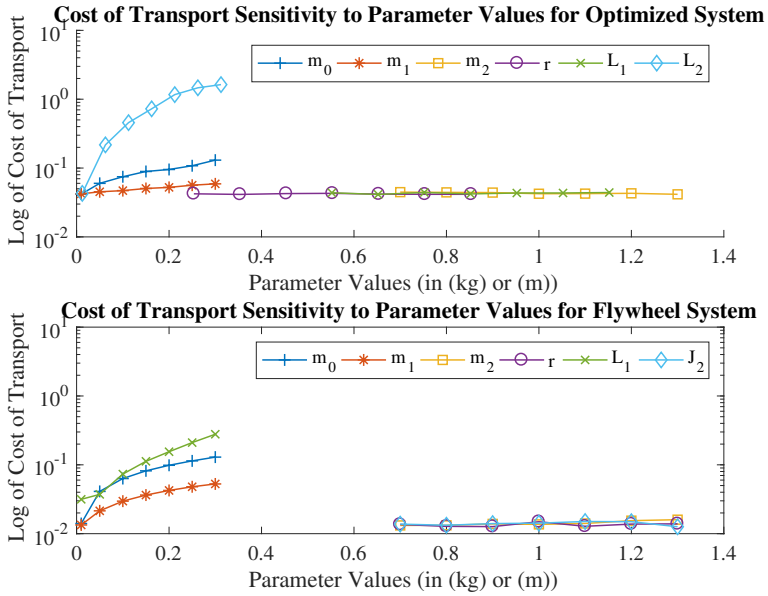

Figure 5.7: Top: Log Cost of Transport sensitivity to system parameters for the optimized acrollbot system with original parameters $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 0.5537$ (m), $L_1 = 0.8522$ (m), $L_2 = 0.0123$ (m) for $T = 2$ (s), $d_{stride} = 0.3$ (m). Increasing the length of L_2 even minimally has a significant effect on the COT. Increasing m_0 or m_1 also has some effect on the COT, though we see that the specific values of m_2, r, L_1 have minimal effect on the COT. Bottom: Log of Cost of Transport sensitivity to system parameters for the optimized flywheel-on-a-wheel system with original parameters $m_0 = 0.01$ (kg), $m_1 = 0.01$ (kg), $m_2 = 1.0$ (kg), $r = 1.0$ (m), $L_1 = 0.0189$ (m), $\widetilde{L}_2 = 0.9999$ (m) for $T = 2$ (s), $d_{stride} = 0.3$ (m). For this system, length L_1 is the most sensitive parameter, with respect to COT. Increasing m_0 or m_1 also has some effect on the COT, though we see that the specific values of m_2, r, J_2 have minimal effect on the COT.

be the minimum value possible, also affect the COT as they increase in magnitude.

Fig. 5.7, at bottom, shows the log of the Cost of Transport sensitivity for the flywheel model with a large radius wheel and a small flywheel. m_2, r, J_2 have almost no effect on the COT when offset from their optimal values, as found by the framework. Increasing L_1 from its minimal value, which puts the flywheel further from the center of the large wheel, increases the COT most. m_0 and m_1, which were again optimized to their minimums possible, also increase the COT as they increase, but at a slower rate than for L_1.

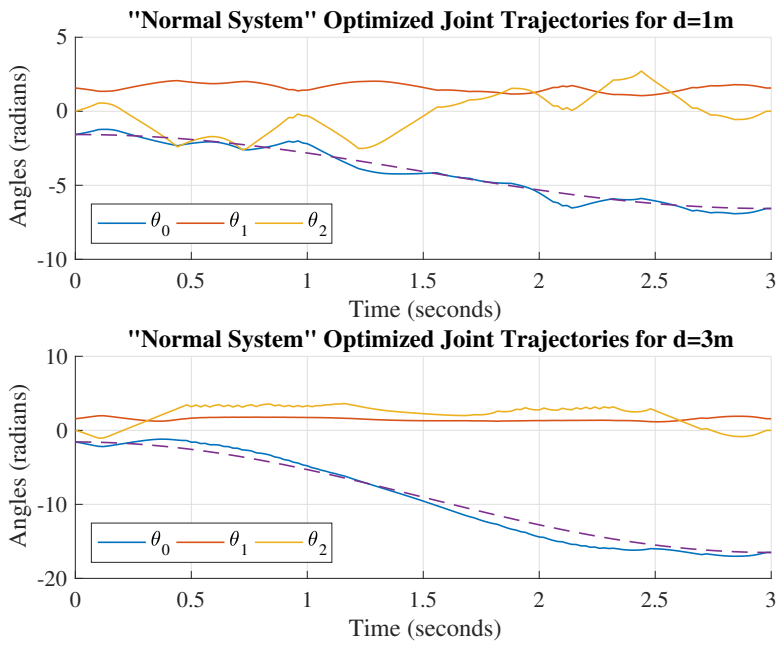

Figure 5.8: Trajectories for hand-picked design parameters. Optimal motions are more extreme and exaggerated, and high-frequency oscillations in the last link are no longer observed. At top, a slower motion ($d = 1$ (m) in $T = 3$ (s)) with a COT of 2.3739; at bottom, faster locomotion ($d = 3.0$ (m) in $T = 3$ (s)) with a COT of 5.4221.

5.4.5 Comparison to a "Normal" (Hand-Designed) System

To put the trajectories and system parameters discovered via the optimization framework in perspective, we compare our optimal system with one that might be built in the real world as an initial design. Intuitively, one might pick L_1 and L_2 to be roughly similar, as in a typical acrobot design. Specifically, we choose the following system parameters as a "typical" design choice: $m_0 = 0.2$ (kg), $m_1 = 0.4$ (kg), $m_2 = 0.4$ (kg), $r = 0.2$ (m), $L_1 = 0.4$ (m), $L_2 = 0.4$ (m). The total mass and height of this system is approximately the same as that of our optimal system. For these parameter choices, our optimization framework discovers trajectories with an associated Cost of Transport on the order of 1E2 times worse than any trajectories returned with optimal parameters, emphasizing the importance of optimizing design parameters in tandem with motion trajectories.

Figure 5.8 shows example trajectories for such a hand-picked system. Of note is the

lack of high-frequency oscillations here. Instead, more exaggerated (i.e., higher amplitude), slower-frequency motions of the last link are used for simultaneous balance and locomotion.

5.4.6 Effects of Passive Damping on System

Clearly, a passive mechanical spring could help generate the required oscillations for the system more efficiently. However, it is less obvious how damping might help or hurt the system. In general, damping seems likely to reduce efficiency, since energy is dissipated. And yet, damping at the passive wheel (f_0) increases the controllability of the linearized system, as discussed in Section 5.2.4. Sweeping independently across values of f_0 and f_2 (damping in parallel with the actuator) for travel of $d = 0.3$ (m) in $T = 2$ (s), using a fixed set of parameters (r,L_1,L_2,m_0,m_1,m_2) that are optimal in the zero-damping case, we found a linear relationship between f_2 and the COT, while f_0 did not affect the COT very much across the range tested, as shown in Figure 5.9.

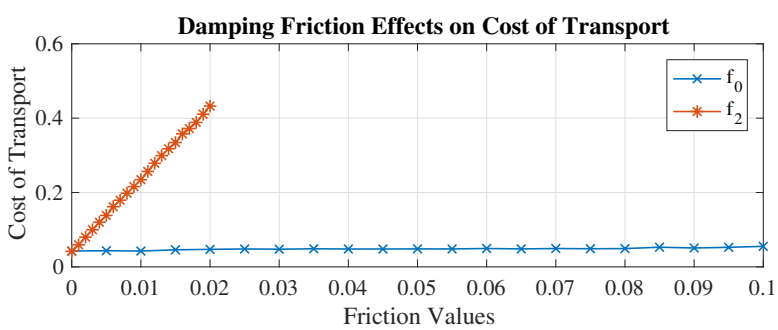

Figure 5.9: COT vs. friction for an optimized acrollbot system. Damping at the wheel has minimal effect on the COT, but increasing damping at link 2, where the input torques are, significantly increases required energy.

5.5 Discussion and Conclusions

As mentioned in the previous section, the various motions described in this chapter are illustrated in the supplementary videos[3][4] associated with [117]. Below, we highlight some details.

First, note basic elements of locomotion, which include two key aspects. (1) COM is offset in a direction opposite the direction of COM acceleration. To remain balanced, most inverted pendulums instead bias COM toward the direction of acceleration ("leaning forward"), so the net ground reaction force (GRF) acts along the line of the pendulum. (2) Vertical oscillations stabilize the system, as discussed earlier [116].

Next, when non-optimal design parameters are used, more exaggerated, bobbing motions are required (see Fig. 5.8). Also, since most of the optimal design solutions for the two-link acrollbot include a tiny L_2 value, close to 0.01(m), which is barely visible in video, we highlight the angle of this link in such cases by including a dashed, red line. For the flywheel acrollbot, note that L_1 is quite short (see Sec. 5.4.3).

Finally, note that the simulations provided at the end[4] (from 1:17 onward) all involve control laws and dynamics simulations implemented using ode45 in MATLAB, as an independent generalization (and confirmation) of the surprising dynamic locomotion techniques we observe. Specifically, we can control forward and backward locomotion for both the "Typical" (Sec. 5.4.1) and "Flywheel" (Sec. 5.4.3) systems.

Several goals motivate this work. In particular, while parameterizations (e.g., lengths and mechanical impedances) of a robot clearly affect performance, better tools for optimizing such design parameters are needed. Also, optimization tools frequently suffer from local minima, in turn requiring appropriate initialization for useful results. By contrast, our framework allows us to optimize both design parameters (e.g., lengths and

[3] `https://youtu.be/C9-P72kI8xI`
[4] `https://youtu.be/PnD7-FHSfZ4`

masses) and motion trajectories simultaneously, and it works with completely randomized initial conditions (e.g., trajectories which initially violated dynamic constraints), which simplifies our search and in turn has allowed us to discover multiple local optima. Finally, methods based on deep learning or other data-driven approaches typically do not elucidate basic physics-based phenomena. In response, we have studied an intentionally non-intuitive yet simple model, which captures key aspects of balance and underactuation. By studying this simplified toy system, we have been successful in carefully analyzing and validating non-intuitive solutions, and we have discovered novel motion planning strategies that we, as human designers, would truly never have anticipated. For example, the unanticipated cyclical "arm-pumping" style motions in our optimal solutions, which can be explained as a Kapitza's pendulum [116] effect, seem quite similar to us to those seen in the Google Deepmind video mentioned in the introduction [21].

Recent work within the UCSB Robotics Lab has applied this optimization framework to a 5-link biped model [118]. Future goals include improved optimization metrics to quantify robustness and agility, in addition to the more easily-calculable energetic costs typically used, and better understanding of the trade-offs and principles at play in optimal humanoid locomotion, particularly as terrain properties, sensor noise, and external disturbances (e.g., process noise) each vary.

Chapter 6

Versatile Trajectory Optimization for Dynamic Maneuvers that Allow for Wheel Slip

Wheel-legged systems have the potential to combine the benefits of two heavily studied research areas: the speed and efficiency of locomotion on wheels with the versatility and agility to surmount various obstacles of legged systems. However, there have been limited results in dynamic locomotion taking advantage of both of these areas. The previously discussed exceptions [4, 56, 58] enforce traditional wheeled-system conditions of non-slipping (the velocity in the rolling direction is exactly equal to the wheel angular velocity multiplied by the wheel radius, without loss) and non-skidding (no velocity perpendicular to the wheel roll direction), which may be violated in practice, especially over terrains with low coefficients of friction such as icy or muddy roads as in [57]. Noting the research gap in wheel-legged systems for dynamic maneuvers that include and account for wheel slip, we reviewed existing work in dynamic vehicle maneuvers involving drifting [59–69] in Section 2.4. The tire models [70, 71] used in these works

however do not lend themselves well to a trajectory optimization framework for general motions, as discussed in Sections 3.3 and 3.4.

In this chapter, using our novel wheel model proposed in Section 3.4, we present a versatile trajectory optimization framework for general (passive or active) wheeled systems that naturally discovers dynamic maneuvers to avoid and/or exploit wheel slipping and skidding depending on the cost function. In simulation with RoboSimian, we show the first results of deliberate and controlled skidding to a halt for a wheel-legged system, and illustrate several other examples such as energy-efficient forward locomotion and hybrid wheel-legged skating. We additionally show experimental evidence of planning and executing dynamic drift parking on a 1/16 scale model car.

The rest of this chapter is organized as follows. Section 6.1 discusses modeling details for RoboSimian and the car model, respectively, to be used in our optimization framework, which is discussed in detail in Section 6.2. Results for various tasks for both systems are presented in Sections 6.3 and 6.4, and discussions and conclusions are given in Section 6.5.

6.1 Modeling

This section discusses modeling details for RoboSimian and the car model, respectively, as used in our optimization framework.

6.1.1 RoboSimian

Kinematics

We first ignore all internal degrees of freedom for the full RoboSimian model and consider a free floating body with floating skate end effectors, whose coordinate frames

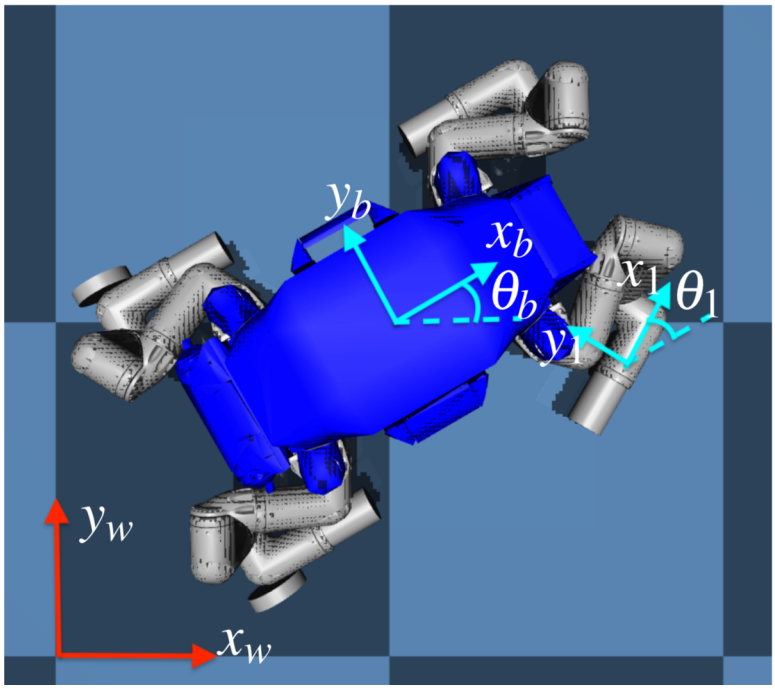

Figure 6.1: RoboSimian coordinates for optimization in MuJoCo [119], shown here for skate 1, where skate numbering is clockwise. θ is the rotation about the z axis, which points out of the page for the world, body, and skate frame coordinates.

are as shown in Figure 6.1. Throughout this work we will assume no pitch or roll from any body.

In the figure, (x_w, y_w, z_w) are the fixed world frame coordinates for the global origin, and thus (x_b, y_b, z_b) are the robot's body center coordinates relative to the fixed world frame. θ_b represents the counterclockwise (CCW) rotation of the body x-axis, relative to the fixed world frame x-axis x_w.

Each skate's coordinates (x_i, y_i, z_i) are relative to the body frame, where $i \in \{1, 2, 3, 4\}$, starting with the front right skate (shown), and proceeding clockwise (so skate 2 is the rear right, skate 3 is the rear left, skate 4 is the front left, when viewed from above). Each skate is also assumed to have an additional degree of freedom to rotate about its z-axis (yaw), θ_i, which is relative to the body angle θ_b.

By not modeling each wheel's rotational angle, we are treating each skate more as

an "ice"-skate than a "roller"-skate. This decision comes from the fact that no sensing is available at the wheel, as they are strapped onto the forearms, and since we are interested in dynamic maneuvers, including those involving slipping and skidding, we hypothesize that the wheel's true position angle at a given time step may not be essential to locomotion.

Dynamics

The dynamics of this simplified system can be derived via a Lagrangian approach. The robot body has mass m_b and inertia J_b, and each skate has mass m_i and inertia J_i. The equations of motion for the system can be written as:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + A(q)^T\lambda = B(q)u + F \tag{6.1}$$

where q are the generalized coordinates, $M(q)$ is the inertial matrix, $C(q,\dot{q})\dot{q}$ denotes centrifugal and Coriolis forces, $G(q)$ captures potentials (gravity), $A(q)^T\lambda$ are constraint forces (where λ are unknown multipliers a priori), $B(q)$ maps control inputs u into generalized forces, and F contains non-conservative forces such as friction. In our model for RoboSimian, $q = [x_b, y_b, z_b, \theta_b, x_i, y_i, z_i, \theta_i]^T \in \mathbb{R}^{20}$ where $i \in \{1,2,3,4\}$. Since each skate's $(x_i, y_i, z_i, \theta_i)$ can be individually set with inverse kinematics in the real system, each skate's coordinates are actuated, for 16 total actuators, i.e. $u = [u_{x_i}, u_{y_i}, u_{z_i}, u_{\theta_i}]^T \in \mathbb{R}^{16}$ where $i \in \{1,2,3,4\}$.

6.1.2 Car Model

The dynamics for the car model can also be derived via a Lagrangian approach, and we use the dynamic bicycle model approximation as our basis.

For the car model, following from Figure 6.2, $q = [x_b, y_b, \theta_b, \theta_f]^T \in \mathbb{R}^4$. (x_b, y_b) are

the global coordinates for the center of mass (COM) of the body, and θ_b represents the counterclockwise (CCW) rotation of the body x-axis, relative to the fixed world frame x-axis x. The steering angle θ_f, relative to body angle θ_b, is not often included as a state in the literature, but instead only as a control input. However, constraints on the angular velocity $\dot{\theta}_f$ and limited steering torque input to the state must be included for realistic motions, motivating our choice of including θ_f.

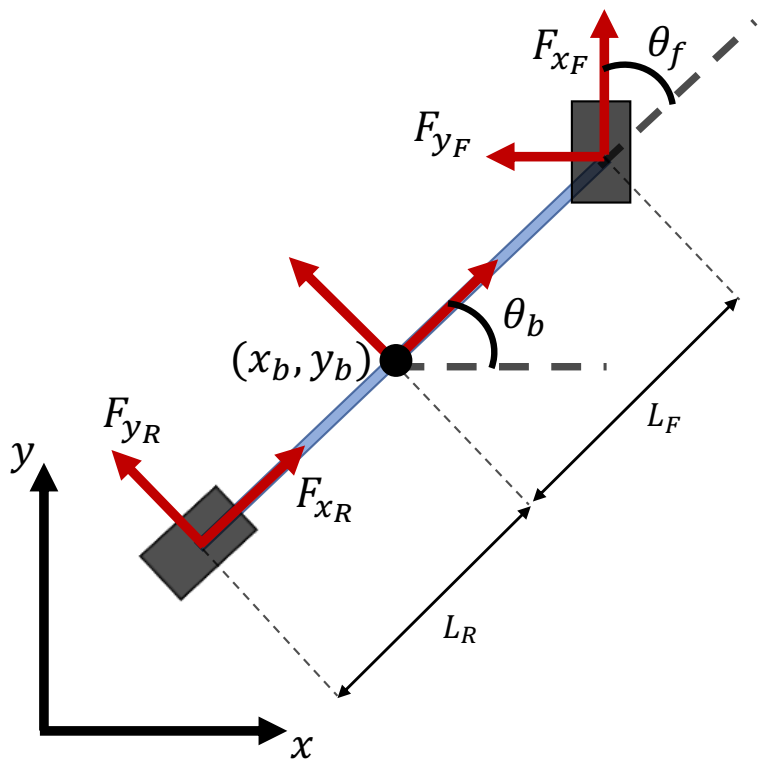

Figure 6.2: Dynamic bicycle model.

The body has a point mass m_b and inertia J_b, and each wheel has a point mass, m_1 and m_2, with the front wheel having inertia J_1 about the z-axis (out of the page). Since the COM may not be at the geometric center of the body, we consider the distance from the COM to the center of the front wheel L_F and the distance from the COM to the center of the rear wheel L_R. Friction forces are considered both in the direction of, and perpendicular to, the direction of wheel roll.

The inertial matrix $M(q)$ can be written as:

$$M(q) = \begin{bmatrix} M_{11} & 0 & M_{13} & 0 \\ 0 & M_{22} & M_{23} & 0 \\ M_{31} & M_{32} & M_{33} & M_{34} \\ 0 & 0 & M_{43} & M_{44} \end{bmatrix} \tag{6.2}$$

where M_{mn} corresponds to element $M(m,n)$ of the matrix:

$$\begin{aligned}
M_{11} &= (m_b + m_1 + m_2) \\
M_{13} &= \sin(\theta_b)(m_2 L_R - m_1 L_F) \\
M_{22} &= M_{11} \\
M_{23} &= -\cos(\theta_b)(m_2 L_R - m_1 L_F) \\
M_{31} &= M_{13} \\
M_{32} &= M_{23} \\
M_{33} &= m_2 L_R^2 + m_1 L_F^2 + J_1 + J_b \\
M_{34} &= M_{44} \\
M_{43} &= M_{44} \\
M_{44} &= J_1
\end{aligned}$$

The torques due to centrifugal and Coriolis effects are:

$$C(q,\dot{q})\dot{q} = [C_1, C_2, 0, 0]^{\mathsf{T}} \tag{6.3}$$

where:

$$C_1 = \dot{\theta}_b^2 \cos\left(m_2 L_R - m_1 L_F\right)$$
$$C_2 = \dot{\theta}_b^2 \sin\left(m_2 L_R - m_1 L_F\right)$$

The two inputs to the system are a force to be set at all wheels (All Wheel Drive) in the wheel-rolling directions, and a torque to control the steering angle of the front wheels, or $u = [u_{wheels}, u_{steering}]^T$. These are mapped into generalized coordinates with matrix $B(q)$, defined as:

$$B(q) = \begin{bmatrix} \cos\left(\theta_b + \theta_f\right) + \cos(\theta_b) & 0 \\ \sin\left(\theta_b + \theta_f\right) + \sin(\theta_b) & 0 \\ L_f \sin(\theta_f) & 0 \\ 0 & 1 \end{bmatrix} \tag{6.4}$$

If we first define the friction forces in global coordinates as:

$$\begin{bmatrix} F_{x_{Fg}} \\ F_{y_{Fg}} \end{bmatrix} = \begin{bmatrix} \cos\left(\theta_b + \theta_f\right) & \sin\left(\theta_b + \theta_f\right) \\ -\sin\left(\theta_b + \theta_f\right) & \cos\left(\theta_b + \theta_f\right) \end{bmatrix} \begin{bmatrix} F_{x_F} \\ F_{y_F} \end{bmatrix}$$

$$\begin{bmatrix} F_{x_{Rg}} \\ F_{y_{Rg}} \end{bmatrix} = \begin{bmatrix} \cos\left(\theta_b\right) & \sin\left(\theta_b\right) \\ -\sin\left(\theta_b\right) & \cos\left(\theta_b\right) \end{bmatrix} \begin{bmatrix} F_{x_R} \\ F_{y_R} \end{bmatrix}$$

then F can be written as:

$$F = [F_1, F_2, F_3, 0]^\mathsf{T} \tag{6.5}$$

where

$$
\begin{aligned}
F_1 = &\ F_{x_{Fg}} + F_{x_{Rg}} \\
F_2 = &\ F_{y_{Fg}} + F_{y_{Rg}} \\
F_3 = &- F_{x_{Fg}} L_F \sin(\theta_b) + F_{x_{Rg}} L_R \sin(\theta_b) \\
&+ F_{y_{Fg}} L_F \cos(\theta_b) - F_{y_{Rg}} L_R \cos(\theta_b)
\end{aligned}
$$

6.1.3 Note on No-Skid Constraints

For general wheeled mobile robots, $A(q)^T\lambda$ typically contains constraints ensuring no slip (free rolling in the direction the wheel is pointing) and no skid (no velocity along the wheel's rotation axis perpendicular to the free rolling direction), which come from writing these constraints in Pfaffian form $A(q)\dot{q} = 0$. λ can be explicitly solved for by differentiating $A(q)\dot{q} = 0$ and substituting in $\ddot{q}$ from Equation 6.1. Details were discussed in Section 3.2. We note that our framework can also find trajectories enforcing these constraints when the assumptions of no slip and no skid are valid, but for the rest of this chapter we set $A(q)^T\lambda = 0$.

6.2 Trajectory Optimization

This section provides details for formulating the locomotion problem for a wheeled system as a trajectory optimization. The full nonlinear system is discretized, and direct collocation along with backward Euler integration is used to generate motion as in [75] [120] and Chapters 3.5 and 5. More precisely, with slight abuse of notation, the

problem is formulated as:

$$\text{find} \quad q, \dot{q}, u, F_n, F_{fric} \text{ at discrete timesteps } k = 1...N \tag{6.6}$$

$$\text{subject to} \quad \text{minimize cost } J$$

- State Constraints:

$$\phi(q, \dot{q}, u, F_n) = 0 \tag{6.7}$$

$$\psi(q, \dot{q}, u, F_n) \geq 0 \tag{6.8}$$

- Dynamics Constraints:

$$\begin{aligned} &M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \\ &\qquad + A(q)^T\lambda = B(q)u + F_n + J(q)^T F_{fric} \end{aligned} \tag{6.9}$$

- Friction Constraints (for each contact i):

$$\gamma e + D^T v^{k+1} \geq 0, \quad \beta \geq 0 \tag{6.10}$$

$$\mu F_{n_i} - e^T\beta \geq 0, \quad \gamma \geq 0 \tag{6.11}$$

$$(\gamma e + D^T v^{k+1})^T \beta = 0 \tag{6.12}$$

$$(\mu F_{n_i} - e^T\beta)\gamma = 0 \tag{6.13}$$

$$F_{fric_i} = D\beta \tag{6.14}$$

- ZMP Constraints (RoboSimian Only)

- Gait Constraints (RoboSimian Only)

where each of the above constraints is detailed below, along with cost function considerations.

6.2.1 Objectives

The cost function J is made up of terms penalizing both energy use over the entire time interval, J_E, as well as the offset from a set of final desired goal states, J_F, in the format discussed in Section 3.5. It is straightforward to add additional cost function terms, depending on the desired task, for example measures between desired final and intermediate body positions and/or orientations. To penalize energy use, for RoboSimian, J_E approximates the integral of the absolute power, as in Section 5.3:

$$J_E = \int_0^t \sum_{i=1}^{N} \sqrt{L + \epsilon}\, dt \tag{6.15}$$

where $L = (\dot{q}^T U)^2$, $\epsilon = 10^{-6}$ is a regularization term to help smooth the cost function as suggested in [115], and N is the number of sample points for the trajectory.

For RoboSimian, $U = [0, 0, 0, 0, u_{x_i}, u_{y_i}, u_{z_i}, u_{\theta_i}]^T \in \mathbb{R}^{20}$ for $i \in \{1, 2, 3, 4\}$. Note that the above cost function aims to minimize power consumption due to moving the end effector in the simplified skating model, while the actual power cost for the full system results from setting the joint positions via inverse kinematics, which will include some additional energy, in moving internal joints appropriately. We are assuming here that such considerations are on average highly correlated to motion of end effector, so that this is still a good measure to account for energy-efficient motion. For the car model, energy is more simply penalized by adding $u^T u$ to the cost function at each time step.

The cost term J_F is defined as the weighted squared error between a set of goal coordinates (x_g, y_g, θ_g) and the final body coordinates (x_N, y_N, θ_N) of the trajectory:

$$J_F = \alpha_x (x_g - x_N)^2 + \alpha_y (y_g - y_N)^2 + \alpha_\theta (\theta_g - \theta_N)^2 \tag{6.16}$$

where weights $\alpha_x, \alpha_y, \alpha_\theta$ can vary based on the desired task, i.e. if final body orientation

is important. Additional terms can be added to minimize final velocities or intermediate states.

6.2.2 State Constraints

The initial states q_0 and $\dot{q}_0$ are constrained exactly based on the robot's current state. For the rest of the N time points, these are bounded by the feasible work space we are considering (for example for placing each skate with inverse kinematics) and by the physical limits of the robot. The input u is also bounded explicitly, as well as implicitly by $\dot{q}$ ranges. For RoboSimian, the normal force on each skate F_{n_i} is constrained based on the contact schedule as explained in Sec. 6.2.6, and forced to be approximately evenly distributed between all skates currently in contact with the ground.

For the car model, we are also not explicitly modeling load transfer, but the normal force F_{n_i} at each contact (where $i \in \{1, 2\}$) is of large importance as its magnitude directly sets a limit on the frictional force available at each contact point. We investigate (1) leaving F_{n_i} as a non-negative open variable, (2) constraining F_{n_i} within "reasonable bounds" (i.e. range in $[1/4(m_b + m_1 + m_2)g,\ (m_b + m_1 + m_2)g]$ at each contact), and (3) setting F_{n_i} as $1/2(m_b + m_1 + m_2)g$. Clearly we can expect different results in each of these scenarios, and future work will involve incorporating an accurate load transfer model.

6.2.3 Dynamics Constraints

At each time step k, with $h = \Delta t$ the time step interval, the dynamics are constrained:

$$q_{k+1} = q_k + h\dot{q}_{k+1} \tag{6.17}$$

$$\dot{q}_{k+1} = \dot{q}_k + h\ddot{q}_{k+1} \tag{6.18}$$

with

$$\ddot{q}_{k+1} = M_{k+1}^{-1}(B_{k+1}u_{k+1} + F_{n_{k+1}} + J_{k+1}^T F_{fric_{k+1}} \\ - C_{k+1}\dot{q}_{k+1} - G_{k+1}) \tag{6.19}$$

where we write $M(q_{k+1})$ as M_{k+1}, and use similar notation for other terms.

6.2.4 Friction as a Linear Complementarity Problem

We have already discussed posing the friction for each wheel in the optimization as a set of Linear Complementarity Problems (LCPs) at length in Section 3.4. As a reminder, the circular friction cone is approximated by a polyhedral cone $\mathcal{F}$:

$$\mathcal{F}(q) = \{\, F_n n \,+\, D\beta \,|\, F_n \geq 0,\ \beta \geq 0,\ e^T\beta \leq \mu F_n \} \tag{6.20}$$

where F_n is the magnitude of the normal contact force, n is the local unit z-axis representing the six dimensional unit wrench of the normal component of the contact force, the columns of D are direction vectors that positively span the space of possible generalized friction forces, $e = [1, 1, ..., 1]^T \in \mathbb{R}^p$ where p is the number of edges of the polyhedral approximation, and $\beta \in \mathbb{R}^p$ is a vector of weights. Due to the nature of the wheel, we consider anisotropic friction models, as shown in Figure 6.3. For RoboSimian, we take this one step further and consider a friction triangle, where friction occurs only perpendicular to the skate's rolling direction (i.e. like an 'ice-skate'). For the car we will investigate two different anisotropic friction models:

(1) $|d_1| = |d_2| = |d_4| = 0,\ |d_3| = 1$

(2) $|d_1| = 0.01, |d_2| = |d_4| = 0.3,\ |d_3| = 1.$

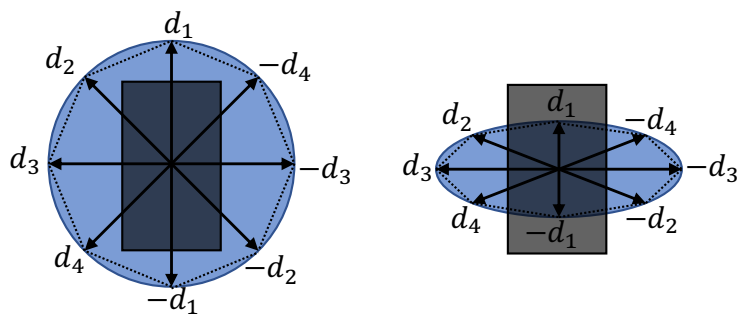

Figure 6.3: Isotropic and anisotropic polyhedral (shown here as octagonal) friction cone approximations of a contact point at the center of the wheel, when viewed from above. The normal force is coming out of the page, and each arrow represents a direction vector.

LCP Formulation

In the optimization, for each contact at each time step k, the following inequality constraints are enforced to produce the correct friction forces:

$$\gamma e + D^T v^{k+1} \geq 0, \quad \beta \geq 0 \tag{6.21}$$

$$\mu F_n - e^T \beta \geq 0, \quad \gamma \geq 0 \tag{6.22}$$

with the complementarity conditions:

$$(\gamma e + D^T v^{k+1})^T \beta = 0 \tag{6.23}$$

$$(\mu F_n - e^T \beta)\gamma = 0 \tag{6.24}$$

where γ can be interpreted as a scalar roughly equal to the magnitude of the relative tangential velocity at a contact, and v^{k+1} is the global 2D planar velocity vector of the contact point at the end of the next time step. From the above constraints the friction

F_{fric} at a particular contact can then be recovered with:

$$F_{fric} = \begin{bmatrix} F_{fric_x} \\ F_{fric_y} \end{bmatrix} = D\beta \tag{6.25}$$

F_{fric} for each contact is then mapped from global coordinates to generalized coordinates with $J(q)^T$ and input into Equation 6.9.

6.2.5 Zero Moment Point (RoboSimian Only)

The ZMP is defined as the point on the ground where the sum of all the moments of the active forces is equal to zero. Planning to ensure the ZMP remains in the support polygon created by the ground contacts of the skates is crucial for stability so that the robot will not fall over. For the simplified skating model with 5 rigid-body links, the ZMP can be calculated as in [121]. The x and y components of the ZMP located at point $p_{ZMP} = [p_x, p_y, p_z]^T$ are then:

$$p_x = \frac{Mgx + p_z\dot{\mathcal{P}}_x - \dot{\mathcal{L}}_y}{Mg + \dot{\mathcal{P}}_z} \tag{6.26}$$

$$p_y = \frac{Mgy + p_z\dot{\mathcal{P}}_y + \dot{\mathcal{L}}_x}{Mg + \dot{\mathcal{P}}_z} \tag{6.27}$$

where M is the total mass of the system, g is gravity, (x, y) are the coordinates of the Center of Mass (COM), p_z is the height of the floor, $\mathcal{P}$ is the total linear momentum, and $\mathcal{L}$ is the total angular momentum.

At each time step, depending on the gait and configuration of the robot's skates in contact with the ground, a ZMP constraint is added to the optimization to ensure stability during that time interval. When four (or three) skates are in contact with the ground, the ZMP is constrained to lie within the quadrilateral (or triangle) created from

connecting the four (three) skates. If only two skates are in contact with the ground, which in this work we assume would occur only for diagonal skates, the ZMP must lie on the line segment connecting these two skates.

6.2.6 Gait (RoboSimian Only)

A gait can be provided as input to the optimization in the form of a 4 by N schedule matrix of boolean values indicating which skates are in contact with the ground. This pattern directly determines the desired height z_i of each skate in the form of a spline that depends on the duration(s) a skate should not be in contact with the ground. In turn this pattern also directly sets the desired normal forces evenly for skates remaining on the ground.

6.3 RoboSimian Results

6.3.1 Implementation Details

The trajectory optimization is implemented in both MATLAB and Python with CasADi [122], using IPOPT [123] to solve the NLP. Depending on complexity and planning horizon, the computation time of the unoptimized MATLAB code can vary from a few seconds to 1 hour, solved on a laptop. Warm-starting the optimization initial conditions with a suitable guess, or with the solution of a previous run, has a significant impact on increasing convergence speed.

In particular, solving the NLP can be computationally intensive due to the equality constraints of the LCP, which does not allow us to formulate the problem as a single QP even if the ZMP constraints were linearized such as in [56]. One idea to mitigate this convergence issue is to place the equalities into the cost function as in [58]. However,

then we no longer have the guarantee they will be exactly 0, and care must be taken to properly weight the cost function terms to heavily penalize deviations from 0.

Since the trajectories found with the framework are for a simplified skating model ignoring internal degrees of freedom, the solutions must be mapped back to the full system with inverse kinematics. We verify the trajectories found on the full system using PD control on the joint positions with inverse kinematics to track the skate $(x_i, y_i, z_i, \theta_i)$ desired positions in MuJoCo [119]. Due to the mismatching dynamics between the simplified system and full system, a whole-body controller is needed to more accurately track the full trajectory, to be implemented for future work.

6.3.2 Experiments

To show the versatility of our framework and the sorts of dynamic trajectories it can generate, we consider the results from 3 desired tasks:

1. Skating forward from rest optimally with respect to speed and efficiency
2. A dynamic skidding parking maneuver, showing the main strengths of our approach
3. Hybrid gaits for walk-skating with 2-3 wheels remaining in contact with the ground

The reader is encouraged to watch the accompanying video[1] for more clear visualizations and simulations of the trajectories discussed [120].

Efficient Skating Forwards

For the first task we consider energy optimal locomotion with all four wheels in contact with the ground for the duration of the trajectory. The cost function is a weighted sum of the modified COT J_E from Eq. 6.15 as well as a term maximizing the final x_b position

[1] `https://youtu.be/SV7Kw3vlRQI`

of the body. We examine a horizon of 2 seconds, starting RoboSimian from rest. A locally optimal trajectory for this scenario is shown in Figure 6.4. We note that for this trajectory, no slipping or skidding is occurring, as the optimization finds the friction forces and torques necessary to propel the RoboSimian model forward.

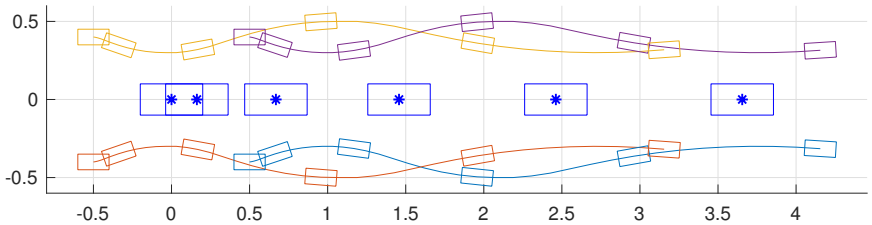

Figure 6.4: Trajectory found for skating in a 2 second horizon from at rest, with snapshots every 0.4 (s). The cost function includes a term minimizing energy as well as maximizing the final body x position.

Dynamic Skidding Parking Maneuver

Next we consider a task of parallel parking in a limited time horizon of 2 seconds. Specifically, we assume RoboSimian is moving at 4 (m/s) in the x direction, with otherwise the same initial configuration at the origin. The "parking spot" is located at (4,-1), and we desire RoboSimian to be parked there with body angle $\theta_b = -\pi$ [rad]. Thus the cost function consists of weighted terms penalizing deviations from these final states for x_b, y_b, and θ_b. Intuitively, we note that with such a small horizon to complete the task, and with no means of braking due to the passive nature of the wheels, we should expect skidding on purpose to slow the system down. As can be seen in Figure 6.5, the trajectory optimization framework finds precisely such a solution exploiting the frictional forces acting perpendicular to the skate roll directions, and is able to effectively minimize the cost function, arriving at the desired end position. Note in particular the angle of skate 1 (the light blue skate) in the third snapshot from the left, which is approximately perpendicular to the direction of motion, and thus being used to skid to slow down.

Screenshots from MuJoCo are shown in Figure 6.6, and additional viewing angles can be seen in the aforementioned video.

As a slight variation, by changing the cost function to reward only the final body position $|\theta_b|$ angle, where we now initialize $\dot{x}_b$ to 8 (m/s), we would anticipate some sort of spinning behavior, or at least the robot to skate around in the narrowest circle it can. Since we do not enforce the non-slip and non-skid conditions, which would limit the subspace of true available motions, the framework finds a trajectory that slips and skids

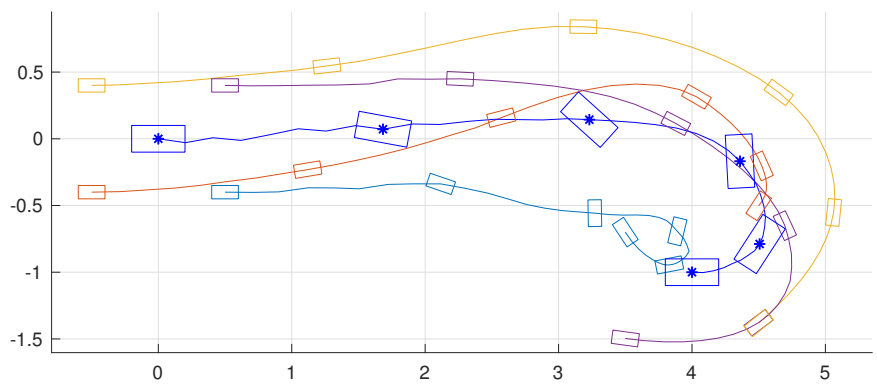

Figure 6.5: Trajectory found for a 2 second horizon initialized with 4 (m/s) body velocity in the x direction, with snapshots every 0.4 (s). The cost function is minimizing squared final body orientation θ_b from $-\pi$, and final body x and y offsets from (4,-1).

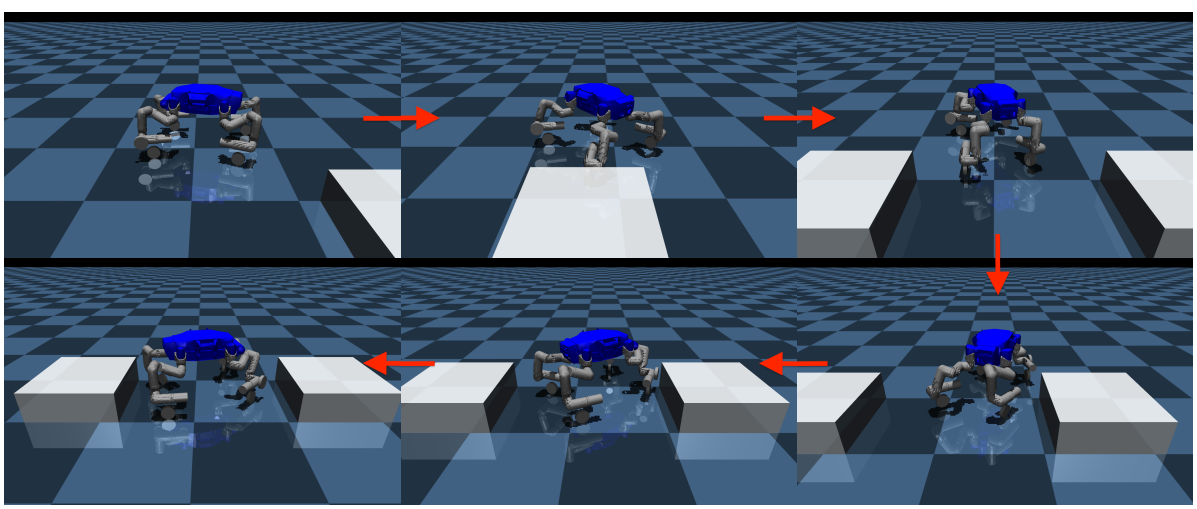

Figure 6.6: Clockwise from top left, a dynamic parking skidding trajectory, in MuJoCo [119].

throughout the 2 (s) interval, similar to how a stunt driver might perform a "donut" in a car, shown in Figure 6.7.

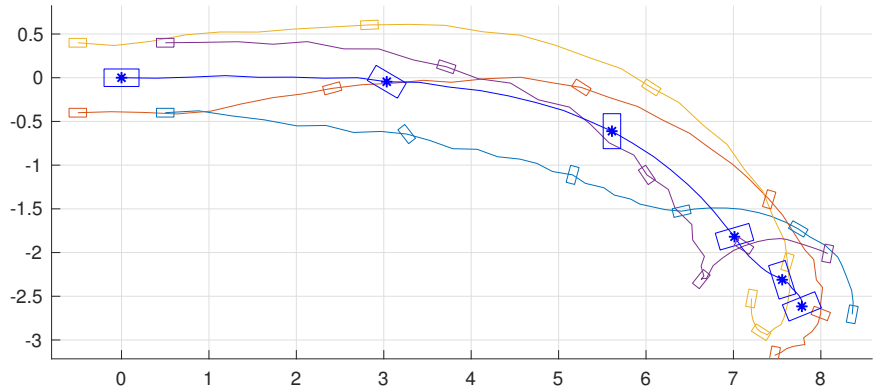

Figure 6.7: Trajectory found for a 2 second horizon initialized with 8 (m/s) body velocity in the x direction, with snapshots every 0.4 (s). The cost function is minimizing negative squared final body orientation θ_b (so attempts to find large magnitudes for θ_b).

6.3.3 Walking-Rolling Gaits

Our framework is able to generate trajectories for gaits with any pattern of three wheels in contact with the ground, as well as patterns of any two diagonal wheels in contact with the ground, and alternating combinations of these two types. Ensuring the ZMP is within the support polygon of the remaining wheels in contact is crucial for stability of the system during the trajectory. As an example, we show one trajectory of static wheel-walking with skates 2, 1, 3, 4 lifted off the ground, in that order, as in Figure 6.8, as well as a static trotting gait, with diagonal wheels alternating in the air, as in Figure 6.9.

The trajectory in Fig. 6.8 is for a 1.5 second horizon and gait for which each skate is lifted for 0.25 (s), in order for skates 2, 1, 3, 4; shown with '+' symbols in the figure. The ZMP is overlaid with the red dash-dot line, which is constrained to be within the support polygon of skates in contact with the ground. The cost function minimizes energy use

and maximizes x_b final state, while attempting to keep final y_b and θ_b at 0.

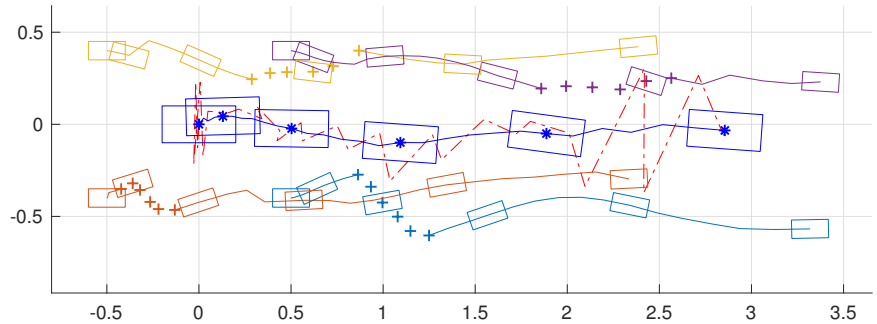

Figure 6.8: Trajectory found for a 1.5 second horizon and predetermined rolling-walking gait where skates 2,1,3,4 are in order lifted off the ground each for 0.25 (s). Snapshots are taken every 0.3 (s). The '+' symbols show the sections of the trajectory where that skate is off the ground, and the red dash-dotted line is the location of the ZMP throughout the trajectory. The cost function is minimizing energy and maximizing final x offset from the origin, and minimizing final y_b and θ_b offsets.

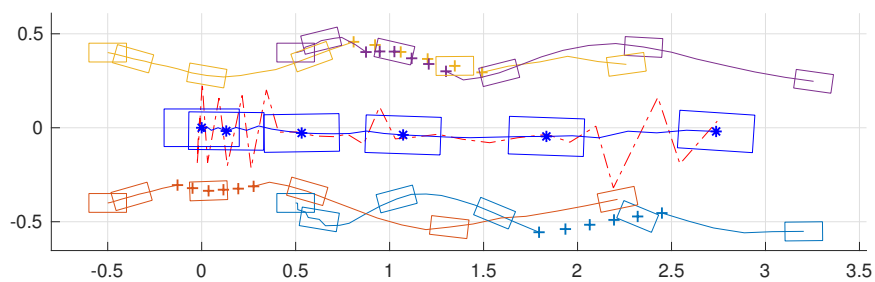

Figure 6.9: Trajectory found for a 1.5 second horizon and predetermined static trotting gait where diagonal skates are always in contact with the ground, with snapshots every 0.3 (s). The '+' symbols show the sections of the trajectory where a skate is off the ground, and the red dash-dotted line is the location of the ZMP throughout the trajectory. The cost function is minimizing energy and maximizing final x offset from the origin, and minimizing final y_b and θ_b offsets.

The trajectory in Fig. 6.9 is for a 1.5 second horizon and gait where diagonal skates 2 and 4 are in the air from 0.5-0.75 (s) and diagonal skates 1 and 3 are in the air from 1.0-1.25 (s), which is shown by '+' symbols. The cost function minimizes energy while maximizing final x_b position (also minimizing y_b and θ_b offsets from 0). While a skate pair is in the air, the ZMP must lie on the line between the other two skates still on the ground. We note that when a skate is not in contact with the ground, it cannot produce

any direct forces to locomote the system, and thus the final x_b position is not as large as when all four skates are on the ground throughout the trajectory.

6.4 Car Results

6.4.1 Implementation Details

The trajectory optimization is again implemented in MATLAB with CasADi [122], using IPOPT [123] to solve the NLP. The car hardware is an Exceed RC Blaze model, which is an All Wheel Drive (AWD) vehicle. A Raspberry Pi with a Navio2 board is used to interface with the motor and steering servo. The forces found with the optimization are converted to PWM signals to send to the motor and servo.

The car parameters are shown in Table 6.1. The masses and distances are measured, and inertias are estimated from the masses and physical dimensions of the relevant components. The COM is estimated to be at the geometric center of the vehicle, and the coefficient of friction is estimated from the material properties.

Parameter	Value
m_b [kg]	1.26
m_1 [kg]	0.01
m_2 [kg]	0.01
L_F [m]	0.09
L_R [m]	0.09
J_b [kg $\cdot$ m^2]	0.0064
J_1 [kg $\cdot$ m^2]	3.5E-6
μ	0.7

Table 6.1: Car and environment physical parameters.

6.4.2 Dynamic Skidding Parking Maneuver

We illustrate the effectiveness of our method by considering two different dynamic drift parking scenarios. The reader is encouraged to watch the accompanying video[2] for better visualizations.

In the first scenario, shown in Figure 6.10, the car starts from rest at $(x_0, y_0) = (0, 0)$ with body angle $\theta_0 = 0$. The cost function penalizes deviations from the goal location of $(x_N, y_N) = (2.5, 0)$ with body angle $\theta_N = \pi/2$, under a short planning horizon of $T = 0.75$ (s), where $N = 15$. The anisotropic friction model used is (1) from Sec. 6.2.4, and the normal forces are as (2) in Sec. 6.2.2 (though we note (3) produces reasonable results as well). The result is a forward sliding parking maneuver with a 90-degree turn.

In the second scenario, shown in Figure 6.11, the car starts from rest at $(x_0, y_0) = (0, 0)$ but with body angle $\theta_0 = -\pi$. The cost function penalizes deviations from the goal location of $(x_N, y_N) = (2, 0.8)$ with body angle $\theta_N = 0$, under a short planning horizon of $T = 1$ (s), where $N = 20$. The anisotropic friction model used is (2) from Sec. 6.2.4, and the normal forces are as (2) in Sec. 6.2.2, (though we note (3) produces reasonable results as well). The result is a 180-degree backward sliding parking maneuver.

For both of these scenarios under such short planning horizons, it is not physically possible to achieve either goal without skidding. The snapshots in the figures are the result of open-loop execution (in red) of the desired trajectory (in blue) found with the optimization framework, and we could expect better tracking performance if using a mixed open-loop closed-loop controller such as in [63] or [64].

[2] `https://youtu.be/SA5w_x0fv9Y`

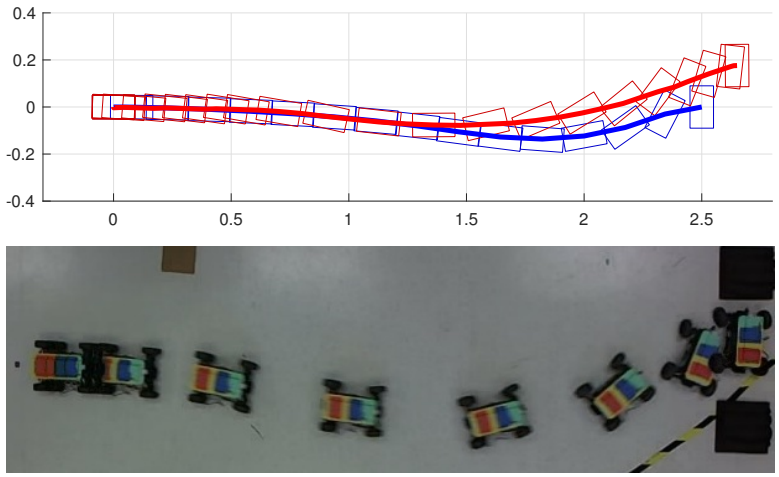

Figure 6.10: Forward park skidding.

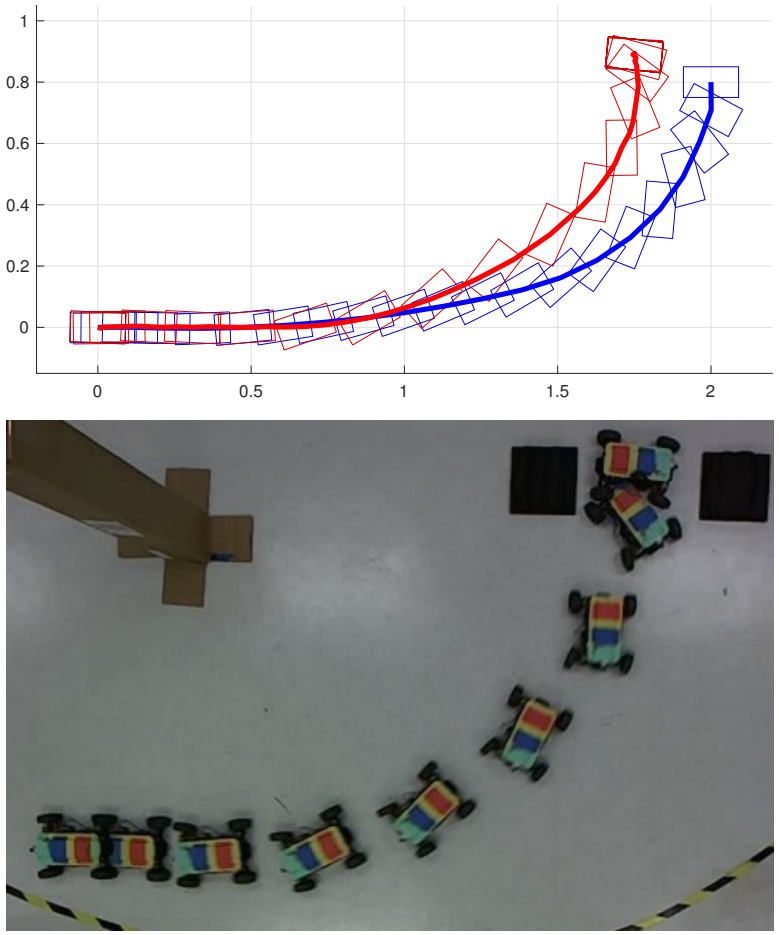

Figure 6.11: Backward park skidding.

6.4.3 Versatility of Framework

In section 6.4.2 we showed two examples of dynamic trajectories found with our framework under short planning horizons, where we noted that skidding was required in order to minimize the cost function. However we would also like to note the versatility of our framework as we consider longer planning horizons for the same tasks. Figure 6.12 shows three locally optimal trajectories found by our framework for the forward skidding parking task over planning horizons of 0.75, 1.0, and 1.5 seconds. The cost function is still minimizing the difference between the Nth (last) knot point and the goal in all three trajectories. As the planning horizon lengthens, skidding is no longer required to achieve the goal, and our framework finds a longer, slower path that minimizes the cost function. This shows that the anisotropic friction model in conjunction with our framework can produce intuitive and realistic trajectories regardless of vehicle velocity, and can be used as a general tool including non-drifting scenarios.

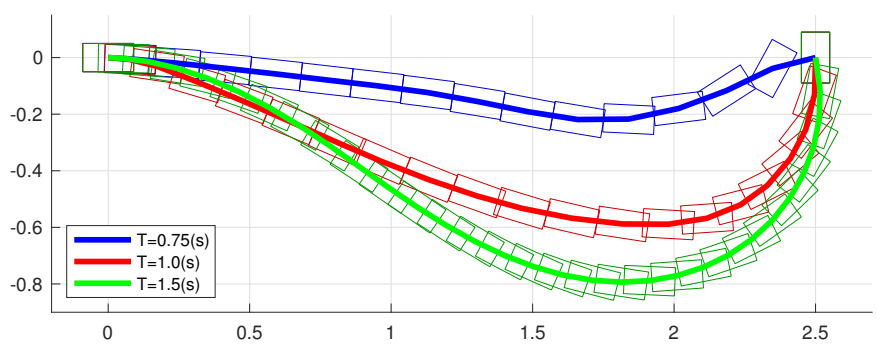

Figure 6.12: Trajectories found for horizons of 0.75 (s), 1.0 (s), and 1.5 (s) (with $N = 15, 20, 30$, respectively) where the cost function penalizes distance from $(x_{b_N}, y_{b_N}, \theta_{b_N}) = (2.5, 0, \pi/2)$. For the shortest planning horizon, skidding is required in order to minimize the cost function and reach the goal.

6.5 Discussion and Conclusions

In this work we have shown the benefit of not only accounting for, but also optimizing over, frictional forces in wheeled systems to produce intuitive and dynamic motion primitives. By modeling wheel dynamics as an anisotropic Coulomb friction cone, we formulated the wheel/tire dynamics as a linear complementarity problem and optimized over all states and forces, producing realistic motions that may or may not include drifting behavior, depending on the planning horizon and cost function.

One point warranting further discussion and future work is the direction vectors that make up the matrix D for the anisotropic Coulomb friction cone approximation. Experimentally we found that the vectors perpendicular to the roll direction are most important to capture the true dynamics and produce feasible trajectories, both for the optimization and evaluation on the real systems. Future work will investigate if we can empirically fit the magnitudes of the direction vectors D, similar in spirit to the system identification approach often used with the Pacejka tire model [70, 71].

Another point of discussion is the magnitude of the normal force. In this chapter we did not model load transfer, but considered evenly distributing the normal force between all wheels in contact with the ground, or leaving it as a lightly/positively-constrained variable in the optimization. The problem with the former is that it does not model load transfer, which may result in unnecessarily conservative or inaccurate trajectories. The problem with the latter is that this lets the optimization choose a friction force with essentially any magnitude, or in unrealistic ranges, which can produce unrealistic trajectories. Future work will explore accurate normal force modeling, perhaps involving modeling additional states.

Ongoing work includes porting the framework to C++ for computational efficiency and speed, as well as developing a better whole-body controller for the full system to

accurately track the desired trajectory that is not just a PD controller over the trajectory states. In particular, due to RoboSimian's heavy limbs, the trajectories requiring lifting a skate off the ground inadequately simplifies the true dynamics. This mismatch in dynamics also causes drift between the desired trajectory and actual robot states in MuJoCo. Additionally, we would like to increase the modeled dimensions to include pitch and roll for each body, to take advantage of RoboSimian's overactuated limbs, which may result in even more dynamic motions. We hypothesize these could include motions such as banking on turns to avoid slip and maintain greater stability at high speeds.

Chapter 7

Training in Task Space to Speed Up and Guide Deep Reinforcement Learning

As discussed in Chapter 1, recent breakthroughs in the reinforcement learning (RL) community have made significant advances towards learning and deploying policies on real-world robotic systems. However, even with current state-of-the-art algorithms and computational resources, these methods are still plagued with high sample complexity, and thus long training times, especially for high degree of freedom (DOF) systems. There are also concerns arising from lack of perceived stability or robustness from emerging policies. This chapter aims at mitigating these drawbacks by: (1) modeling a complex, high DOF system with a representative simple one, (2) making explicit use of forward and inverse kinematics without forcing the RL algorithm to "learn" them on its own, and (3) learning locomotion policies in Cartesian space instead of joint space. In this chapter, these methods are applied to JPL's RoboSimian, but they can be readily used on any system with a base and end effector(s). These locomotion policies can be produced in just

a few minutes, trained on a single laptop. We compare the robustness of the resulting learned policies to those of other control methods.

7.1 Introduction

Impressive recent results applying reinforcement learning algorithms such as Proximal Policy Optimization (PPO) [21] [22], Trust Region Policy Optimization (TRPO) [94], Actor Critic using Kronecker-Factored Trust Region (ACKTR) [124], Deep Deterministic Policy Gradients (DDPG) [23], and Asynchronous Advantage Actor-Critic (A3C) [98] to continuous control tasks in robotics suggests the possibility of using (deep) reinforcement learning as a way to increase skating stability and robustness, to improve on Chapters 4 and 6 . However, as powerful and promising as these recent results have been, the sample complexity and training time of these methods remains a major issue when seeking to deploy solutions in real time in the real world. Even for state-of-the-art algorithms and implementations, and especially for high degree of freedom (DOF) complex systems such as RoboSimian, a policy can take millions of iterations to train for a solution that may or may not be stable. It must also be noted that there are no robustness, stability, or performance guarantees on the policies learned, or at least no way to readily quantify these metrics.

One prominent example is shown in the video associated with Heess et al.'s *Emergence of Locomotion Behaviours in Rich Environments* [21], where for the higher DOF system humanoid, we see the emergence of (probably) non-optimal and non-intuitive arm-flailing to "help" locomote the system, as a probable local optimum. The video[1] accompanying this chapter shows an example of the emergence of similar non-intuitive behavior when learning a locomotion policy with PPO [22] for RoboSimian in joint space. We seek to

[1] `https://youtu.be/xDxxSw5ahnc`

avoid such local optima for locomotion policies in our system, and propose intuitively limiting the action space for reinforcement learning algorithms towards quickly generating robust and stable motions.

Most of this recent work in applying reinforcement learning to robotic systems seeks to learn a policy that, given an observation of the current state, outputs raw motor torques to the available actuators in joint space to maximize rewards for the task at hand. However, for an overactuated system such as RoboSimian, which has 28 actuators with high (160:1) gear ratios as well as velocity limits of 1 rad/sec at each joint [32], applying a torque from a learned distribution at each time step is not an intuitively practical approach. For our work, we instead model each motor as a position actuator, to be supplied with a reference position at each time step. This naturally extends to, instead of selecting a torque at each time step, incrementing each motor's current desired position by $\Delta \in \{ -\varepsilon, 0, +\varepsilon \}$.

We also note that these recent learning algorithms (proudly) incorporate no prior knowledge of the system during training, and thus the agent must essentially "learn" forward and inverse kinematics through interacting with its environment, early termination conditions specified by a human, and hand-crafted reward shaping functions. Work in imitation learning, transfer learning, and warm-starting the policy network, either with existing trajectories or using more traditional controllers, has been done to try to reduce the high sample complexity of the vanilla reinforcement learning methods. In this chapter we propose a much simpler idea of incorporating control techniques readily available for most systems such as forward and inverse kinematics, in the spirit that we should use the domain knowledge of the problem that we possess, rather than requiring the system to learn it on its own.

The reinforcement learning framework has already been discussed in Section 3.6, where we consider an environment modeled as an MDP 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ of states,

actions, transition function, and reward function. In this chapter, the states consist of a subset of the robot's (RoboSimian's) positions and velocities, the actions are motor positions or Cartesian coordinate end effector offsets, the transition function is modeled by a physics engine (MuJoCo [119]), and the reward changes based on the task (for example forward velocity or distance to a goal). Although we expect any of the aforementioned reinforcement learning algorithms to learn effective skating maneuvers for locomotion (especially when we change the action space from continuous to discrete), we will use the current state-of-the-art, Proximal Policy Optimization (PPO) [22], discussed in Section 3.6.

The rest of this chapter is organized as follows. Sections 7.2 and 7.3 describe modeling and training environment details, respectively. Section 7.4 presents results for tasks such as skating with maximum velocity or to a goal location over noisy terrain, and a brief conclusion is given in Section 7.5.

7.2 Modeling

As outlined in Chapter 4, planning effective, feasible skating motions for RoboSimian involves two complementary problems. The motions of the skates must enable generation of required ground reaction forces without excessive slipping, to move the robot as desired, and solutions for inverse kinematics must be tractable, smooth, and within the dynamic velocity and acceleration limits of the joint actuators of the robot.

7.2.1 Simple "Representative" System

The first of the complementary problems is primarily focused on the skate locations, contact forces, and directions of motion. In fact, when designing the skating motions by hand, reasoning about the skate (x, y, z) and yaw (ϕ) positions over time are the first

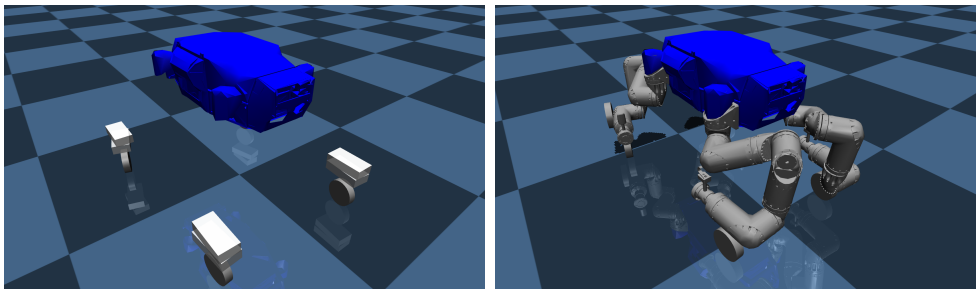

Figure 7.1: Simple Cartesian space model and full RoboSimian model, in MuJoCo [119].

things accounted for, and inverse kinematics are computed along this desired trajectory afterwards to ensure smoothness.

This gives rise to developing a more simple model to "represent" the full RoboSimian model. As shown in Figures 7.1 and 7.2, the model consists of the same torso/body, but without legs, which are replaced by floating bases above the skates. The top rectangular link of the floating bases is moved with slide (prismatic) joints in MuJoCo [119], meaning motion is allowed along a single axis. Depending on the desired task, these can be actuated in the x, y, and/or z directions in the floating base local frame. The bottom rectangular link is actuated by a hinge joint about the z axis, allowing yaw rotation only, which sets (ϕ) for the skate. The total mass of one of RoboSimian's limbs (excluding the skate) is distributed evenly between these two links. Although this simple system does not exactly model the true dynamics of the full RoboSimian system, we hypothesize that it will be "close enough", making it faster to train a locomotion policy with deep reinforcement learning algorithms in Cartesian space. We hope to transfer this learned policy on the simple system onto the full system.

7.2.2 Inverse Kinematics

To use learned policies from the simple model, at run time (or for transfer learning and additional training on the full model), inverse kinematics (IK) must be used to map

the desired skate motion back to the full model.

The IK to set the 6-DOF pose of the skate require choosing from among one of eight IK families, each analogous to a choice of "elbow bending direction" for each of three elbows on a limb [31]. Providing guarantees of smoothness requires precalculation of IK solutions across a region as well as compromises between ideal theoretical contact locations and achievable solutions for our particular robot. In particular, for many desired skate configurations, achieving exact symmetry in end effector locations is either non-trivial or not achievable.

Algorithmic solutions for computing IK tables that satisfy the above conditions are detailed in [31]. However, as calculating such an IK table can be computationally expensive, there is a trade-off for training on the full model in joint space (without IK) vs. training in Cartesian space (making use of an IK table, or computing IK at each time step).

Depending on the implementation, a function calculating IK can add significant overhead to training time in Cartesian space, as each time step requires 4 calls to this function (once for each limb). However, if the range of (x,y,z,ϕ) of each skate in the simple model is limited to a subspace for which IK solutions of the full model both exist and are smooth, either through intuition or some pre-calculation, we can learn a policy on the simple model in Cartesian space, and then map the solution back at test time to the full system, computing IK at each time step for each limb. This eliminates the need to compute IK during training altogether.

7.3 Training Environment

This section describes the environment set up and MDP details of our implementations to learn a locomotion policy for either the simple system or full RoboSimian.

7.3.1 Observation Space

In order to use the policy trained on the simple model for the full model, the observation space (input to the network) must be the same, or similar. As there is no sensing available at the passive wheel in the real model, it is not fair to include any related observations to learn a policy in simulation. So neither the rotational position nor velocity of each skate wheel about its axis are included in the observation space.

At minimum, the observation space for the simple model consists of the following:

- (x_b, y_b, z_b), body global coordinates
- (w, x_w, y_w, z_w), body orientation (from origin x-axis) in the form of a quaternion
- $(x_{s,i}, y_{s,i}, z_{s,i}, \phi_{s,i})$, skate local Cartesian positions and yaws, with respect to the body
- $(dx_b/dt, dy_b/dt, dz_b/dt)$, body translational velocities
- $(d\theta_{x_b}/dt, d\theta_{y_b}/dt, d\theta_{z_b}/dt)$, body rotational velocities
- $(dx_{s,i}/dt, dy_{s,i}/dt, dz_{s,i}/dt, d\phi_{s,i}/dt)$, skate local translational and rotational velocities with respect to the body

The above observation space is (more than) enough to locomote the system in any given direction; for example to train for x-directed locomotion, the reward function can be a difference in potential between the current and previous body positions in the x direction. For tasks that involve moving to a specific (x_g, y_g) goal coordinate in the environment, the above observation space is augmented with the following:

- (x_g, y_g, z_g), global goal coordinates
- $-d$, negative of the absolute Euclidean distance between (x_b, y_b, z_b) and (x_g, y_g, z_g)

- θ_{goal}, signed local angle between current body heading (w, x_w, y_w, z_w) and (x_g, y_g, z_g)

The simple representative model and the full model using IK in Cartesian space thus have the same observation space in their respective simulations. The observations for which the real robot does not have direct sensing can readily be estimated with forward kinematics or Jacobians in real time on the real system.

When training in joint space for the full system, in addition to the body positions, orientations, and velocities, each joint's position and velocity (28 actuated motors) is now part of the observation space. Again, the rotational positions and velocities of the skate wheels are not included.

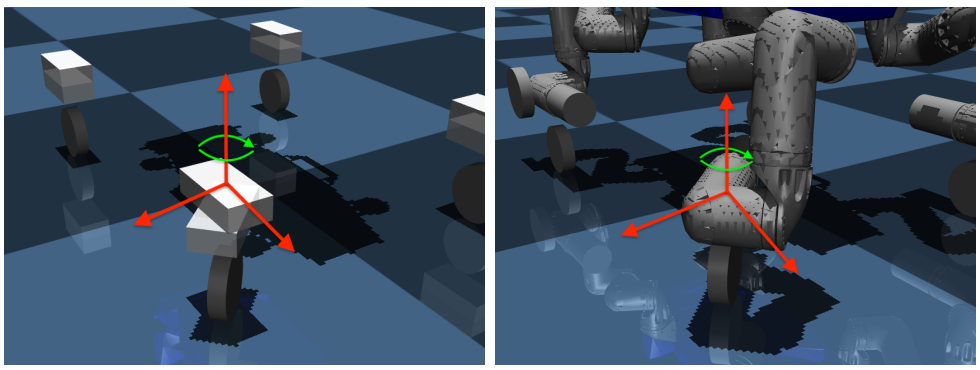

Figure 7.2: Equivalent states and action spaces for the simple Cartesian space model and full RoboSimian model using IK. Actions are translation offsets along each skate's local x,y,z axes, and rotational about the z axis, shown here for $\phi = \pi/4$[rad].

7.3.2 Action Space

As discussed in Sections 7.1 and 7.2, it is more intuitive to model the actuators for RoboSimian as position servos rather than torque motors. For the full system trained in joint space, the action space is chosen to be discrete, and the policy chooses an offset from $\{-\epsilon, 0, +\epsilon\}$ from each motor's current desired position as its new reference.

For the simple model and full model using IK, the action space is also discrete, but in Cartesian coordinates. This is shown in Figure 7.2. At each time step, the policy chooses

an offset from $\{-\epsilon, 0, +\epsilon\}$ from each skate's ($x_{s,i}$, $y_{s,i}$, $z_{s,i}$, $\phi_{s,i}$) current desired positions. This ϵ is a design parameter, and can change based on the task and state. It is logical for $y_{s,i}$ and $\phi_{s,i}$ to change by different offsets ϵ_y and ϵ_ϕ, for example, due to the difference in units (meters vs. radians). A caveat is that the IK joint position differences of the full model between time steps must be bounded. We note that a small difference in the end effector location can have a large difference in the IK solution, even when using the same IK family, if near a singularity. We seek to minimize these events by keeping the simple model's workspace within smooth IK solution spaces for the full model.

7.3.3 Reward Functions

We consider potential-based shaping functions of the form:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{7.1}$$

to guarantee consistency with the optimal policy, as proved by Ng et al. in [125]. The real-valued function $\Phi : S \rightarrow \mathbb{R}$ varies between tasks, with two such example tasks consisting of:

1. maximizing forward velocity in the x direction:

$$\Phi(s) = \frac{x_b}{\Delta t} \tag{7.2}$$

2. minimizing the distance to a target goal (x_g, y_g, z_g):

$$\Phi(s) = -\sqrt{(x_b - x_g)^2 + (y_b - y_g)^2 + (z_b - z_g)^2} \tag{7.3}$$

This reward scheme gives dense rewards at each time step, towards ensuring the opti-

mal policy is learned, and allows us to avoid complicated hand-crafted reward functions with many variables that ultimately output a single number anyway.

7.3.4 Implementation Details

We use a combination of OpenAI Gym [126] to represent the MDP and MuJoCo [119] as the physics engine for training and simulation purposes. We additionally use the OpenAI Baselines [96] implementation of PPO (PPO2) as a basis, making some key modifications for our system. Our neural network architecture is the default Multi-Layer Perceptron (MLP), which consists of 2 fully connected hidden layers of 64 neurons each, with tanh activation. The policy and value networks both have this same network structure.

The default design parameters of these implementations are kept the same, while perhaps not producing the most optimal or time-efficient policies for our system, to show that intuitively reducing the action space has a large effect on training time and policy robustness.

7.4 Results

We seek to compare the training times and robustness of learned policies for the following systems:

- *SS*: Simple "representative" system
- *FS in JS*: Full system trained in joint space
- *FS in CS*: Full system trained in Cartesian space with IK (to set joints)

We also seek to evaluate how well the learned policy of the simple system transfers to the full system with inverse kinematics. *SS* and *FS in CS* always have the same

observation and action spaces, for fair comparisons. We consider tasks of skating at maximum velocity in the $+x$ direction, and of locomoting to a particular goal location (x_g, y_g). All policies along with additional comparisons to other control methods are shown in the supplementary video[2] accompanying the publication for which this chapter is based on [127].

7.4.1 Skate Straight

First we consider the task of maximizing forward velocity in the $+x$ direction. The observation and action spaces are as detailed in Sections 7.3.1 and 7.3.2, with $\epsilon = 0.01$. The action space for *SS* and *FS in CS* is limited to a ± 0.1[m] offset of $y_{s,i}$ and ± 0.3[rad] offset of $\phi_{s,i}$ for each skate from its given starting position, with $x_{s,i}$ and $z_{s,i}$ fixed, to match the constraints used in designing skating motions for forward locomotion in Chapter 4. The action space for *FS in JS* is bounded only by the joint limits. Training on *FS in JS* also enforces early termination of an episode if any self-collisions are detected, or if contact occurs with the ground from any part of the robot other than the skates. The reward at each time step is a potential-based shaping function as in Equation 7.1, with Φ as in Equation 7.2, rewarding body velocity in the $+x$ direction.

Training Sample Complexity Comparison

Figure 7.3 shows the episode reward mean vs. number of training time steps, with mean and standard deviation of three training runs for each system. The episode reward is the sum of the individual rewards at every time step, so it is the sum of all instantaneous velocities, a maximum of 1000 values (without early termination). We see that training in Cartesian space gives much higher total returns, with far fewer time steps. This is likely due to both the smaller observation and action spaces for *SS* and *FS in CS*, as

[2] `https://youtu.be/xDxxSw5ahnc`

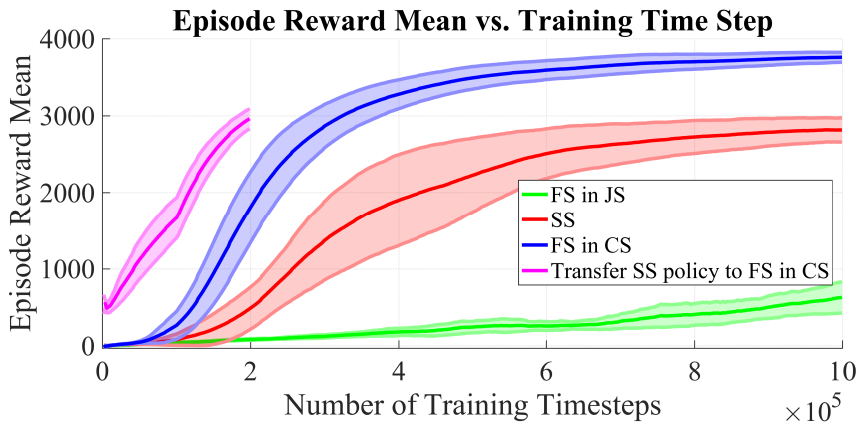

Figure 7.3: Average episode reward over training 1 million time steps for the full system in joint space (*FS in JS*), simple system (*SS*), and full system in limited Cartesian space (*FS in CS*) for a task rewarding forward velocity in the $+x$ direction. The pink line shows the results of transferring the learned policy on the simple system (*SS*) to the full system in Cartesian space (*FS in CS*) and further training for 2e5 timesteps. As the dynamics do not match perfectly, using the *SS* can be a means to accelerate training, especially if fast IK computation during training is not available.

well as to the fact that the agent for the full system is being forced to learn forward and inverse kinematics in conjunction with trying to maximize returns. A training episode terminates early if there are any self-collisions in the current robot configuration, or if non-skate limbs come in contact with the ground, where the average duration of a training episode before one of these occurrences is shown in Figure 7.4.

In addition to learning a policy that produces returns that are not nearly as high as those for *SS* and *FS in CS*, the resulting motions for training on *FS in JS* also do not *look* optimal, where typical behavior appears to be sinking quite low to the ground (always on the brink of a collision) and waving one of its rear limbs around in the air while only the other three limbs actually skate (see video). Even after training for 10 million time steps, which takes about 12 hours on a single laptop, the policy has still not completely learned kinematics, frequently terminating episodes early, and it is still not achieving the same returns as the policy trained in Cartesian space.

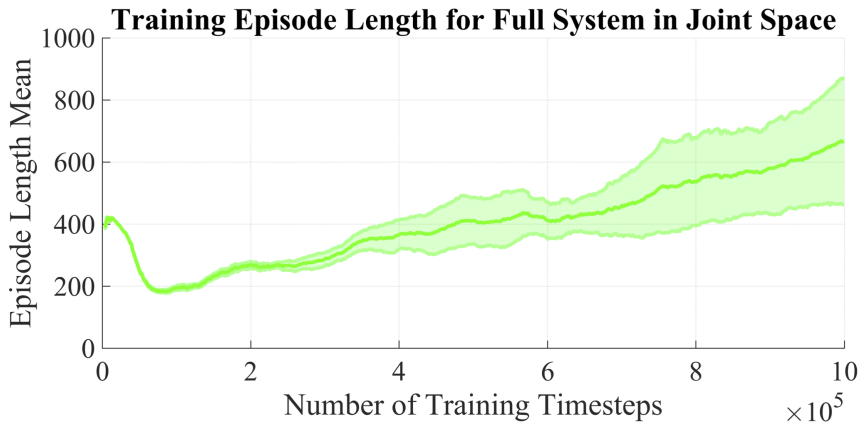

Figure 7.4: Episode length mean vs. number of training time steps, where each episode normally lasts 1000 training time steps unless ended prematurely. Early termination is the result of any internal self-collisions, or any part of the robot body touching the ground other than the skates, which is still occurring even after training for over 1e6 timesteps.

Training Wall Clock Time Comparison

Figure 7.5 shows the episode reward mean vs. wall clock time. The overhead in training time for the FS in JS compared to the SS can be attributed to more weights in the neural networks having to be learned due to the larger observation and action spaces, more frequent environment resets from early terminations, and possibly also from simulating more complex dynamics. If an IK table (implemented as a hash table, where the keys are (x, y, z, ϕ) position tuples and values are the corresponding joint positions) is available during training time, this adds a small overhead from 4 constant time look ups per training time step. Without such a table, calculating IK 4 times at each time step has the potential to introduce a more significant overhead, depending on the implementation. This may suggest transferring a policy learned on the SS onto the *FS in CS*, calculating IK only at run time.

An IK table can also be dynamically generated as training progresses, to avoid frequently re-calculating the same position tuples as the policy learns subspaces of the workspace that maximize rewards. We also note that the training time from slow IK

calculations is still worth it, as the alternative of training in joint space produces less robust policies that attain far fewer rewards.

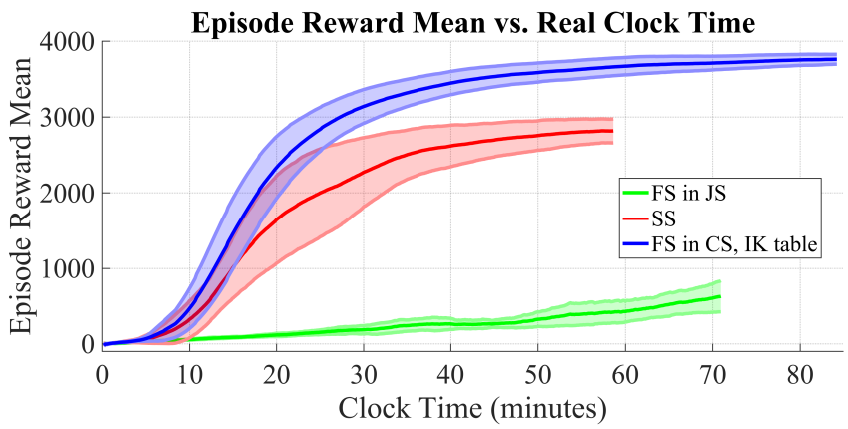

Figure 7.5: Episode reward mean vs. real clock time for training over 1 million time steps. The *SS* is fastest to train due to the fewest number of weights to learn in the network, limited search space (along with *FS in CS*), and the simplest dynamics to simulate in MuJoCo. Our IK table implementation incurs a small overhead from repeated constant time lookups while training.

Policy Transfer from *SS* to *FS in CS*

To test how well the policy learned on the simple model transfers to the full system, we initialize the weights of the policy and value networks for the *FS in CS* with those learned from the *SS*. Since the observation and action spaces are the same for both models, we do not run into any implementation complications. The pink line in Fig. 7.3 shows the episode reward mean for training 2e5 time steps after initializing the *FS in CS* network weights with those learned with the *SS*.

The initial returns start much higher than 0, but not as high as the returns from running the *SS* policy on its own system. This is expected since the dynamics do not match perfectly, but it is encouraging that locomotion can be transferred in this manner. This suggests the possibility of training on the simple system for other tasks, and then executing the learned policy on the full system, to accelerate training.

A noteworthy observation from our experiments is that the learned policy for the *SS* could not traverse randomly varying, sinusoidally-smooth terrain, as the amplitude increased, as defined in Section 7.4.2. This could be due to the dynamics of the system being incapable of generating the forces to traverse this new terrain, or that this new observation space had not been explored during training, and thus the policy had not learned which actions to take in these slightly varying states. However, the full robot *was* able to make forward progress after transferring that same *SS* policy onto the *FS in CS*, without further training of that policy. This implies that the policy was robust enough, but the *SS* version of the system itself may not have the means to produce the forces necessary to get over the small hills. This further encourages the use of training a policy on the *SS* and then transferring it to the full system for other tasks.

7.4.2 Skate To Goal Under Uncertainty

The next task we consider is locomoting the system from the origin (0,0) to a goal location (x_g, y_g), in particular (5,0)[m], over randomly varying, sinusoidally-smooth terrain with varying friction coefficients. For this task, the observation spaces from the maximum velocity skating task are augmented with the global goal coordinates, negative distance to the goal from the robot's current location, and the angle between the robot's heading and the goal, as discussed in the latter part of Section 7.3.1. The action spaces are left unchanged, as the limited spaces for *SS* and *FS in CS* should be enough to locomote the system to the goal (it is always possible to increase ranges, or allow x and z skate position changes if the terrain is *too* rough). The reward at each time step is again a potential-based function as in Equation 7.1, with Φ as in Equation 7.3 where we assume $z_g = z_b$, rewarding a decrease in the distance to the goal, and penalizing moving away from the goal.

We train the policy to reach the goal (5,0) for 2 million time steps over smooth, sinusoidal terrain with amplitude $A = 0.1$ [m], period 2π [m] in both x and y directions, and coefficient of friction $\mu \in [0.5, 1]$. The terrain is randomly generated and moved in the xy plane by $(\delta x, \delta y) \in [-1, 1]$ [m] at the start of each training episode (environment reset), on top of random perturbations of the joints, positions, and velocities of the initial states. An episode is considered completed when (x_b, y_b) is within 0.2 meters of (x_g, y_g), or when the episode times out after 1000 time steps.

We then test the policy over the same family of smooth sinusoidally varying terrain now also varying amplitude $A \in [0, 0.2]$ [m] (the period 2π [m] and coefficient of friction range $\mu \in [0.5, 1]$ are unchanged), and compare results with runs of our previously designed open-loop trajectory to locomote the system 5 meters forwards from Chapter 4.

Figure 7.6 shows the end (x_b, y_b) positions and distributions for 100 runs each for the trained stochastic policy on the *FS in CS* as well as for the hand-designed trajectory. Reaching the goal is defined as successful if (x_b, y_b) ends within 0.2 meters of (x_g, y_g), for both methods. The large cluster of policy points in a radius of 0.2 meters around (5,0) is due to the episode completion condition. Of the 100 trials, there are 57 successes using the learned policy, vs. only 23 successes for the hand-designed open loop trajectory. From the resulting end positions and observing the policy in action, we see that if the robot has too much lateral error, due to the small action space ranges, it cannot move laterally towards the goal and gets "stuck". An even more robust policy might thus be learned by increasing the action space ranges, and incorporating more knowledge into the observation space such as sensor readings of terrain height, for future work.

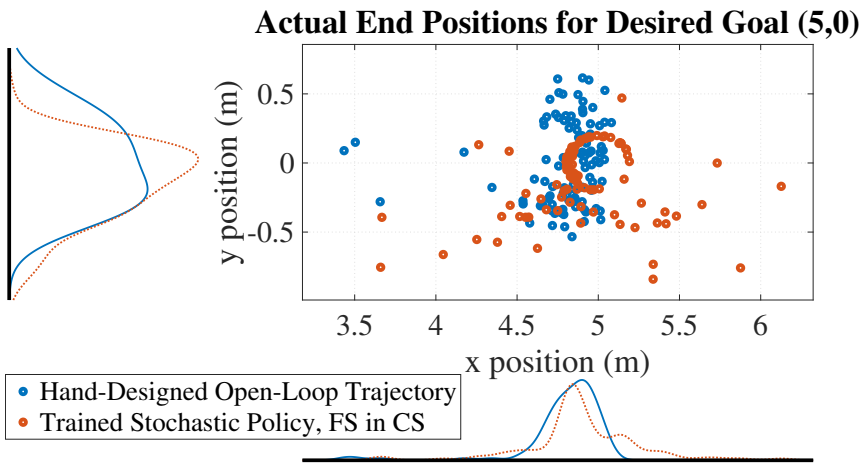

Figure 7.6: Starting from the origin and with a goal of reaching (5,0) in the environment, resulting end positions for the robot body center of mass from 100 runs each of the learned policy on the *FS in CS* as well as of a hand-designed trajectory. The terrain consists of randomly varying smooth, sinusoidal curves with amplitude $A \in [0, 0.2]$ [m], period 2π [m], and coefficient of friction $\mu \in [0.5, 1]$.

7.4.3 Policy and Hand-Designed Trajectory Comparison

The first 15 (s) of a sample skating trajectory from a policy trained to achieve maximum velocity in the x direction for the *FS in CS* are shown in Figure 7.7. The resulting motions are quite similar to our hand-designed trajectories from Chapter 4, an example of which is shown in Figure 7.8, with symmetric motions in both ϕ and y position between the left and right limbs, as well as an approximate phase difference of $\pi/2$ between the front and rear limbs. There are two key differences: (1) although ϕ is allowed to vary in $[-0.3, 0.3]$ radians, the policy learns trajectories that never reach those limits, and (2) as the robot builds speed, the ϕ offsets chosen decay to a smaller range and oscillate more quickly, along with y, once slip is avoided at the start of the motion. The non-smooth ϕ motions could be due to the nature of executing a stochastic policy, or that the policy has learned to make small corrections in its heading with ϕ, rather than with y, to maximize forward velocity reward.

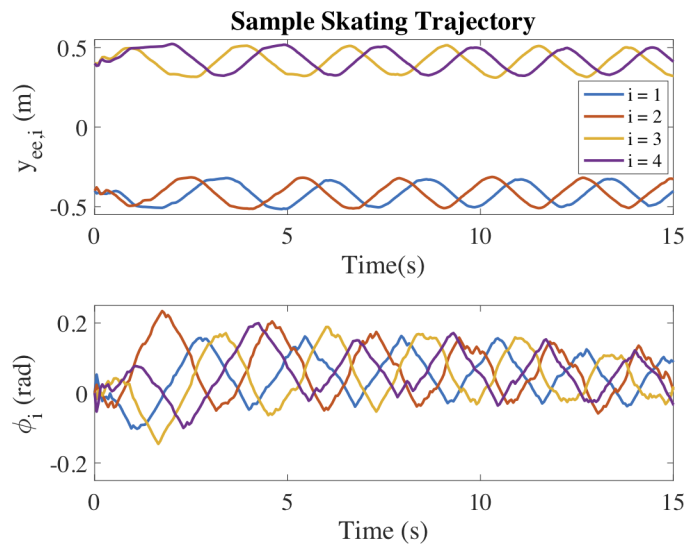

Figure 7.7: The first 15 (s) of a sample skating trajectory from a policy learned for the full system in Cartesian space, after training for 1 million time steps while rewarding forward velocity in the $+x$ direction. When viewing the robot from above, skate $i = 1$ is on the front right limb, $i = 2$ is the rear right, $i = 3$ is on rear left, and $i = 4$ is the front left.

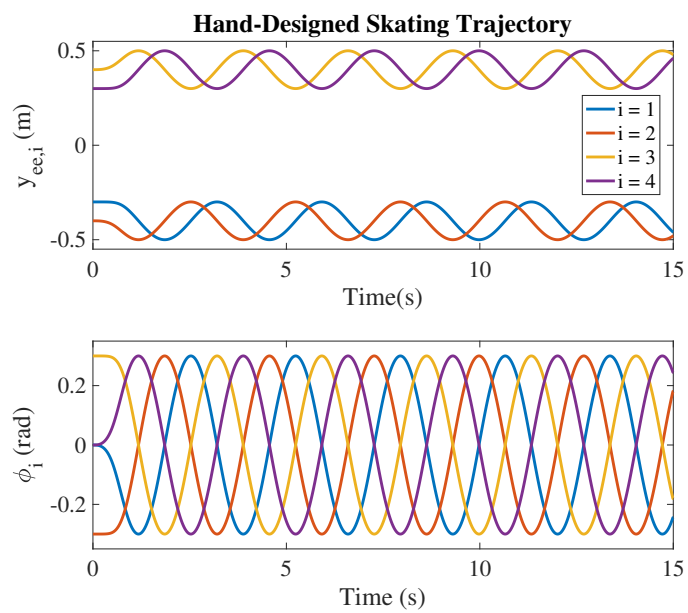

Figure 7.8: Local skate end effector positions and angles ϕ for a simple hand-designed, straight body trajectory with 4 skates. The front two skates are out of phase by $\pi/2$ with the back skates, and due to symmetry, are out of phase by π with each other to ensure minimal slip and maximum propulsion forces.

7.5 Conclusion

This chapter detailed the use of targeting state-of-the-art reinforcement learning algorithms on simple, representative systems with standard control techniques, and intuitively limiting the action space to reduce sample complexity, increase robustness, and accelerate training time.

We showed that using inverse kinematics and training in Cartesian space significantly speeds up training time. Even without a fast way to compute IK, the resulting policies are more stable and robust than training in joint space, where even after millions of time steps, the agent is still learning kinematics, producing non-intuitive motions, and terminating training episodes early. Such considerations are especially important for a high DOF system such as RoboSimian.

We also provide a workaround in the case of no fast IK being available, by training a policy on a simple representative system, and transfer learning onto the full system, calculating IK only at run time, or for fewer time steps if further training of the policy is needed due to mismatching dynamics.

Although it may seem intuitive to use inverse kinematics to learn on a lower dimensional system, to the best of our knowledge, this is the first such work in this area. Here this is applied to a skating system, but the ideas can be readily implemented on any system with a base and end effector(s). A more recent work showed the benefit of training in task space with impedance control for manipulators in contact-rich environments, such as when wiping a board [128]. Another example application could be motion planning for a quadruped when placing down a foot during a particular gait.

Chapter 8

Combining Benefits from Trajectory Optimization and Deep Reinforcement Learning

Summarizing from Chapter 1, recent breakthroughs both in trajectory optimization and reinforcement learning have made significant advances towards real world robotic system deployment. Reinforcement learning (RL) can be applied to many problems without needing any modeling or intuition about the system, at the cost of high sample complexity and the inability to prove any metrics about the learned policies. Trajectory optimization (TO) on the other hand allows for stability and robustness analyses on generated motions and trajectories, but is only as good as the often over-simplified derived model, and may have prohibitively expensive computation times for real-time control. This chapter proposes a method to combine the benefits from these two areas while mitigating their drawbacks by (1) decreasing RL sample complexity by using existing knowledge of the problem with optimal control, and (2) providing an upper bound estimate on the time-to-arrival of the combined learned-optimized policy, allowing online policy deployment at

any point in the training process by using the TO as a worst-case scenario action. The method is evaluated for a car model, with applicability to any mobile robotic system.

8.1 Introduction

Deep reinforcement learning (DRL) methods have shown recent success on continuous control tasks in robotics systems in simulation [21–23]. Such methods are applied using no prior knowledge of the systems, leading to problematic sample complexity and thus long training times. Unfortunately, little can be said about the stability or robustness of these resulting control policies, even if more traditional model-based optimal control solutions exist for these same systems.

Additionally, DRL has been almost exclusively applied in simulation, where a failed trial has no repercussions. In the real world a failure can have catastrophic consequences, including damaging the robot or causing injury to humans in the area. Some recent works have successfully learned a policy for a real robot [25] [26], or transferred policies learned in simulation to the real system [27] [28]. Of particular note in [28] is that the learned policies outperform the authors' previous model-based methods with respect to both energy-efficiency and speed. However, instead of training from scratch, it would seem intuitive to use the model-based methods as an initial starting point for DRL, with the reasoning that at any given moment, our learned policy should do *at worst as well as* existing control solutions.

One might be tempted to perform imitation learning updates on trajectories taken from running a model-based optimal control policy, for example using DAgger [129]. However, due to often mismatching dynamics between the simplified system on which this model-based control policy is based on, and the real physical system, this may lead to overfitting a suboptimal policy. There is also the additional concern of deviating too

much from the expert trajectories into regions of the state space not previously visited, in which the policy learned only from expert data may perform poorly.

Instead, we propose interweaving optimal control samples during the policy rollouts of model-free DRL methods in the following manner: at each timestep we can evaluate our policy network to get action a_{RL}, as well as query our trajectory optimization to get action a_{TO}. We then simulate the execution of each of these actions individually, and select the one which gave the larger reward as our true action to use in the real world. Such a scheme, shown in Figure 8.1, should ensure that at worst the agent will always do as well as the model-based optimal control policy, and can only do better. At the beginning of training, we expect this approach to almost exclusively pick a_{TO}; due to the network weights being randomly initialized, it is very unlikely to consistently outperform a model-based method. However as training progresses, and from added policy exploration/exploitation, the number of selected on-policy samples will increase.

Related work on using trajectory optimization to help learn or guide a control policy include [130], [131], and [132]. These works differ from the proposed method in this chapter as they focus more on incorporating offline demonstrations or trajectories into training to guide the policy search, whereas in this work the trajectory optimization is run online at each timestep and compared with the current policy action, ensuring a worst-case scenario.

A related work combining prior knowledge of the system with learning is [133], where the policy chooses between actions computed with a simple PID controller and from evaluating the current actor network. Although the authors observe this controller helps achieve faster and more stable learning performance, it is not optimal and much can still be improved in terms of sample complexity. Additionally, there are no guarantees on the policy at any given time, and no minimum time to goal estimates or worst-case scenarios.

Another related work that combines learning with MPC is POLO [134]. POLO

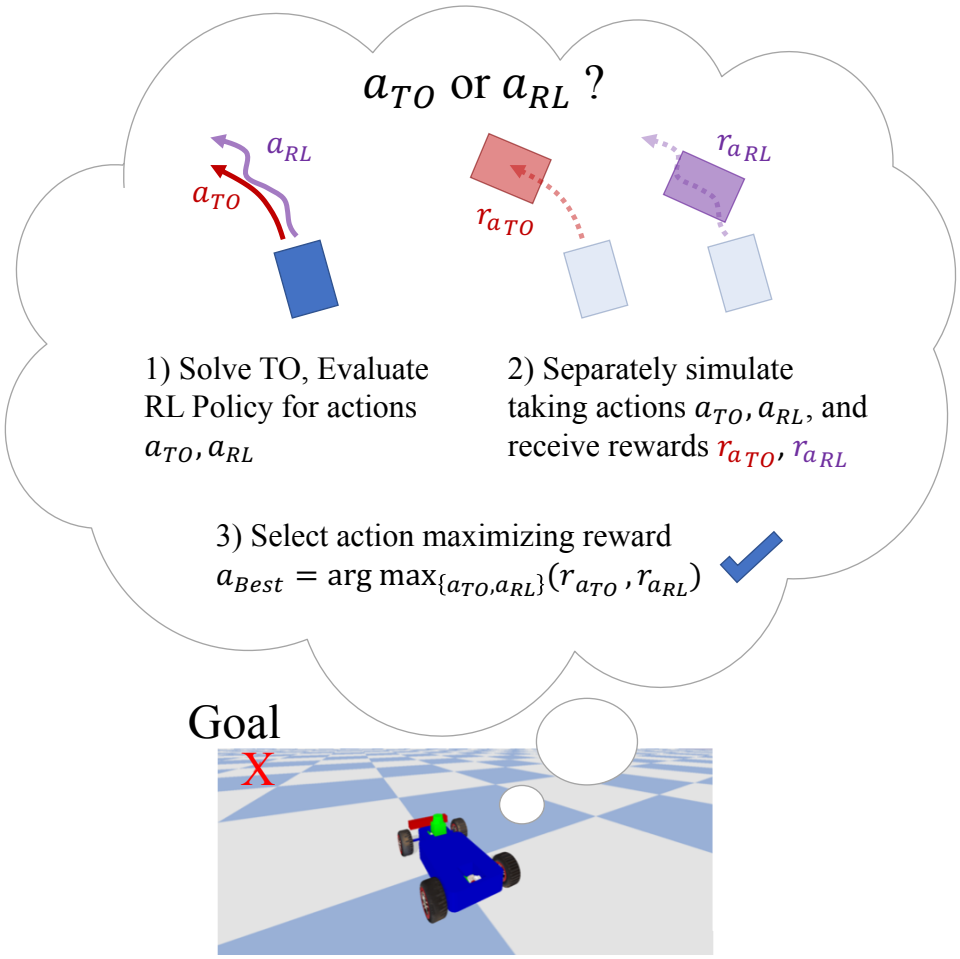

Figure 8.1: Agent picking between actions from trajectory optimization or from reinforcement learning, simulating taking each respectively, and then selecting the one leading to greater reward to execute in the real world.

seeks to improve MPC by learning a global value function, of which it has only a local estimate when initialized. As a result, it cannot be run online and provides no policy guarantees, as it cannot achieve a desired result without first learning an estimate of the global value function. This chapter in contrast seeks to improve on policy learning by using a trajectory optimization framework to guide the learning process, and provides a worst-case scenario action that can be run online.

The rest of this chapter is organized as follows: Section 8.2 will review details on reinforcement learning, imitation learning as behavioral cloning, and robot dynamics in the context of a car model. Section 8.3 describes the trajectory optimization framework used to calculate optimal trajectories for the car, and our algorithm combining this trajectory optimization with deep reinforcement learning (in this case PPO) is presented in Section 8.4. Section 8.5 shows results on the benefits of using our algorithm, and a brief conclusion is given in Section 8.6.

8.2 Preliminaries

8.2.1 Reinforcement Learning and Proximal Policy Optimization

The reinforcement learning framework has already been discussed in Section 3.6, where we consider an environment modeled as an MDP 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ of states, actions, transition function, and reward function. In this chapter, the states consist of a subset of the robot's positions and velocities, the actions are motor torques or positions, the transition function is modeled by a physics engine [135], and the reward is a potential-based function to minimize distance to a target goal.

Although we expect to see benefits from combining trajectory optimization with any

deep reinforcement learning algorithm, in this chapter we will also use the previously discussed Proximal Policy Optimization (PPO). As the objective functions are relevant to our algorithm, we remind the reader of the surrogate objective with clipped probability ratio:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{8.1}$$

where $\hat{A}_t$ is an estimator of the advantage function at time step t as in [95], and $r_t(\theta)$ denotes the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{8.2}$$

where π_θ is a stochastic policy, and θ_{old} is the vector of policy parameters before the update. This objective seeks to penalize too large of a policy update, which means penalizing deviations of $r_t(\theta)$ from 1.

As PPO is an actor-critic algorithm, the full objective function combines terms for this policy surrogate with a value function error term, as well as an entropy bonus to ensure sufficient exploration [22, 98], which we more succinctly define as $L^{PPO}(\theta)$ as

$$L^{PPO}(\theta) := L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \Big[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \Big] \tag{8.3}$$

where c_1, c_2 are hyperparameters, S denotes an entropy bonus, and $L_t^{VF}(\theta)$ is a squared-error loss:

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{targ} \right)^2 \tag{8.4}$$

8.2.2 Learning from Demonstration

In this work we use the classical behavioral cloning (BC) approach to imitation learning where we seek to minimize the error between an expert action and the maximum likelihood estimate action from the current policy:

$$L^{BC}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\Big(a_i^* - \arg\max_{a_i} \pi_\theta(a_i|s_i)\Big)^2 \tag{8.5}$$

for expert demonstration state-action pairs $\{s_i, a_i^*\}_{i=1}^N$, where $\arg\max_{a_i} \pi_\theta(a_i|s_i)$ is the maximum likelihood estimate action a_i for state s_i using policy π_θ.

8.2.3 Robot Dynamics

The equations of motion for a robotic system can be written as:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + A(q)^T\lambda = B(q)u + F \tag{8.6}$$

where q are the generalized coordinates, $D(q)$ is the inertial matrix, $C(q,\dot{q})\dot{q}$ denotes centrifugal and Coriolis forces, $G(q)$ captures potentials (gravity), $A(q)^T\lambda$ are constraint forces (where λ are unknown multipliers a priori), $B(q)$ maps control inputs u into generalized forces, and F contains non-conservative forces such as friction.

In this work, we specifically consider a simple car model, shown in Figure 8.2, with $q = [x_b, y_b, \theta_b, \theta_f]^T$, where (x_b, y_b) are the center of mass coordinates in the world frame, θ_b is the yaw of the body with respect to the global x-axis, and θ_f is the steering angle of the front wheels.

As discussed in Section 3.2, for general wheeled mobile robots, $A(q)^T\lambda$ typically contains constraints ensuring no slip (free rolling in the direction the wheel is pointing) and no skid (no velocity along the wheel's rotation axis perpendicular to the free rolling di-

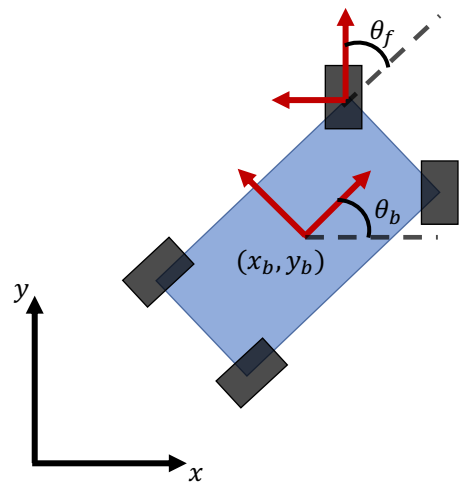

Figure 8.2: Car model used for trajectory optimization.

rection), which come from writing these constraints in Pfaffian form $A(q)\dot{q} = 0$. λ can be explicitly solved for by differentiating $A(q)\dot{q} = 0$ and substituting in $\ddot{q}$ from Equation 8.6.

Remark

Because this constraint is in place, the trajectory optimization will not find solutions where skidding is a viable option allowing for greater reward (such as skidding into a parking space instead of parallel parking, or an aggressive turning maneuver to more quickly change directions). The alternative to this constraint would be to add friction approximations such as in [77], for which the optimization must then solve a Linear Complementarity Problem at each contact point, as in Chapter 6. As this can be a very expensive computation, we avoid this consideration here and instead entrust the DRL algorithm to use the conservative real-time trajectory optimization as a guide towards learning a better policy, in which slip may or may not be optimal.

8.3 Trajectory Optimization

This section provides details for formulating the locomotion problem for the car model as a trajectory optimization. This optimization can be solved in real-time due to the simplified dynamics model with nonholonomic constraints. As before, the full nonlinear system is discretized, and direct collocation along with backward Euler integration is used to generate motion as in [75] [120] and Chapters 3.4, 5, 6. More precisely, the problem is formulated as

$$\begin{aligned} \text{find} \quad & q, \dot{q}, u \quad \text{at discrete timesteps } k = 1...N && (8.7) \\ \text{subject to} \quad & \text{minimize cost } J \\ & \text{State Constraints} \\ & \phi(q, \dot{q}, u) = 0 && (8.8) \\ & \psi(q, \dot{q}, u) \geq 0 && (8.9) \\ & \text{Dynamics Constraints} \\ & D(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + A(q)^T\lambda = B(q)u + F && (8.10) \end{aligned}$$

where each of the above constraints are detailed below, along with cost function considerations.

8.3.1 Objectives

The cost function J is defined as the weighted squared error between a set of goal coordinates (x_g, y_g, θ_g) and the body coordinates (x_N, y_N, θ_N), where N is the number of

sample points for the trajectory:

$$J = \alpha_x(x_g - x_N)^2 + \alpha_y(y_g - y_N)^2 + \alpha_\theta(\theta_g - \theta_N)^2 \tag{8.11}$$

where weights $\alpha_x, \alpha_y, \alpha_\theta$ can vary based on the desired task, i.e. if final body orientation is important.

8.3.2 State constraints

The initial states q_0 and $\dot{q}_0$ are constrained exactly based on the robot's current state. For the rest of the N time points, q and $\dot{q}$ are bounded by joint position and velocity limits. The input torques u are also bounded explicitly by the physical constraints of the robot, as well as implicitly by $\dot{q}$ ranges.

8.3.3 Dynamics constraints

At each time step k, with $h = \Delta t$ the time step interval, the dynamics are constrained:

$$q_{k+1} = q_k + h\dot{q}_{k+1} \tag{8.12}$$

$$\dot{q}_{k+1} = \dot{q}_k + h\ddot{q}_{k+1} \tag{8.13}$$

with

$$\begin{aligned}\ddot{q}_{k+1} = D_{k+1}^{-1}(&B_{k+1}u_{k+1} + F_{k+1} - C_{k+1}\dot{q}_{k+1} \\ &- G_{k+1} - A_{k+1}^T\lambda_{k+1})\end{aligned} \tag{8.14}$$

where we write $D(q_{k+1})$ as D_{k+1}, and similar for other terms.

8.4 Cooperative Trajectory Optimization and Deep Reinforcement Learning

In this section we detail our algorithm, Cooperative Trajectory Optimization and PPO (CoTO-PPO), shown in Algorithm 1. The main idea is that for each new observation s_t at time step t, the current PPO actor network is queried for action a_{RL}, and a trajectory optimization is solved for action a_{TO}. Each of these actions is simulated individually to get rewards $r_{a_{RL}k+1}$ and $r_{a_{TO}k+1}$. The action that produces the larger simulated reward is the one that is selected as the true best action and used in the real world (or to step the actual simulation). Necessary transition information is then appended to either the PPO dataset D_{PPO} or Supervised Learning (SL) dataset D_{SL}, depending on which action was selected. After T time steps corresponding to the current policy/trajectory optimization roll out, the actor-critic PPO networks are updated by optimizing $L^{PPO}(\theta)$ on dataset D_{PPO}, and the actor network is additionally updated with supervised learning by optimizing $L^{BC}(\theta)$ on dataset D_{SL}.

8.5 Results

8.5.1 Implementation Details

We use a combination of OpenAi Gym [126] to represent the MDP and PyBullet [135] as the physics engine for training and simulation purposes.

We additionally use the OpenAI Baselines [96] implementation of PPO (which optimizes $L^{PPO}(\theta)$ as discussed in Sec. 8.2) with the default hyperparameters, but with the Beta distribution to select continuous actions as suggested in [136] to avoid the bias introduced with limited control ranges when using the standard Gaussian distribution.

Algorithm 1: Cooperative Trajectory Optimization and PPO (CoTO-PPO)

```
Initialize function approximation parameters θ
Initialize PPO and Supervised Learning datasets D_PPO, D_SL
for training epoch=1,2,... do
    for timestep=1,2,...T do
        Solve trajectory optimization for action a_TO
        Evaluate PPO policy network for a_RL ~ π_θ(a_t|s_t)
        Simulate taking each action separately and select action maximizing next
         step reward:
                a_Best = arg max_{(a_TO, a_RL)} (r_{a_RL k+1}, r_{a_TO k+1})
        Step environment with a_Best: s_{t+1} ~ p(s_{t+1}|s_t, a_Best)
        if a_Best == a_RL then
            D_k ~ partial trajectory, transition information
            D_PPO = D_PPO ∪ D_k
        else if a_Best == a_TO then
            D_SL = D_SL ∪ {(s_t, a_TO)}
        end
    end
    for K epochs on D_PPO do
        normal PPO updates by optimizing L^PPO(θ)
    end
    for K epochs on D_SL do
        supervised learning LfD updates by optimizing L^BC(θ)
    end
end
```

The Beta distribution parameters α and β are TensorFlow variables and are therefore updated during each SGD minibatch, so the action variance will decrease as the policy converges. The Gym environment we use is similar to the standard HumanoidFlagRun environments, but the humanoid is replaced with the Bullet MIT racecar. Example snapshots from the environment are shown in Figure 8.3. The goal destination/flag is updated only when the car center of mass lies within 0.2 [m] of the current goal location, and then placed randomly in a 10 by 10 [m] grid. The agent has 10 seconds to maximize rewards each trial, which will typically consist of reaching several goal locations consecutively.

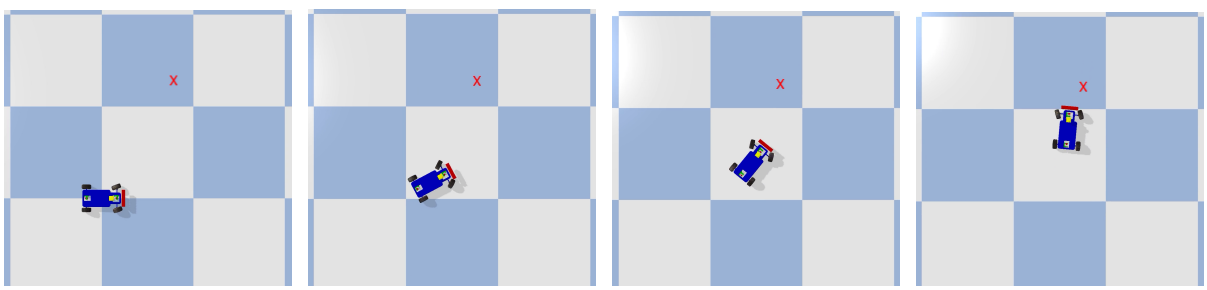

Figure 8.3: Environment snapshots of Bullet racecar with desired goal denoted by the red 'X'.

Our neural network architecture is the default Multi-Layer Perceptron, consisting of 2 fully connected hidden layers of 64 neurons each, with tanh activation. The policy and value networks each have this same structure.

The observation space is:

$$[\ ||(x_g, y_g) - (x_b, y_b)||,\ \theta_g - \theta_b,\ \theta_f,\ \dot{x}_b,\ \dot{y}_b,\ \dot{\theta}_b,\ \dot{\theta}_f\ ] \tag{8.15}$$

which is the distance from the center of mass to the goal location, the angle from the current body heading to the goal, the steering angle of the front wheels, the body velocity in the global x and y directions, the body yaw rate, and the yaw rate of the steering wheels.

The action space is [v, θ_f], which is a desired body velocity to be set with velocity control mapped to a differential drive, and desired steering angle to be set with position control.

We consider potential-based shaping reward functions of the form:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{8.16}$$

to guarantee consistency with the optimal policy, as proved by Ng et al. in [125]. The real valued function $\Phi : S \to \mathbb{R}$ seeks to minimize the distance to a target goal (x_g, y_g):

$$\Phi(s) = -\sqrt{(x_b - x_g)^2 + (y_b - y_g)^2} \tag{8.17}$$

This reward scheme gives dense rewards at each time step, towards ensuring the optimal policy is learned, rewarding incremental motion in the direction of the current goal. Having dense rewards is important in this framework as we are choosing between actions based on the simulated instantaneous reward, which would likely be 0 at most time steps under sparse reward scenarios.

The trajectory optimization is implemented in Python with CasADi [122], using IPOPT [123] to solve the NLP. Due to the imposed torque and velocity limits as well as the nonholonomic constraints added to the dynamics, the trajectory optimization will find solutions that are suboptimal to the true best policy. We will see in the following subsection that even this suboptimal trajectory optimization has a large benefit when combined with policies either learned from scratch or with our method, both during training and testing.

8.5.2 Experiments

We seek to compare and evaluate the following methods:

1. *pure PPO*
2. *pure trajectory optimization*
3. *CoTO-PPO* - (our method)
4. *CoTO-PPO, policy only* - how well does the policy learned from CoTO-PPO perform on its own, without the fail-safe action of the trajectory optimization?
5. *CoTO-(pure PPO)* - how well does combining trajectory optimization with an entirely separately trained agent with PPO perform?

The reader is encouraged to watch the accompanying video[1] for simulations of the discussed policies.

Figure 8.4 shows the episode reward mean vs. number of training time steps for running pure PPO as well as CoTO-PPO on the CarFlagRun environment. The episode reward mean indicates how well the agent was able to continue to progress in the direction of the goal location(s) during each trial. Due to doing at worst as well as the trajectory optimization, CoTO-PPO begins with very high reward mean, and only improves from there as the networks are updated with both DRL and supervised learning updates. Pure PPO on the other hand is forced to learn from scratch, and even after training for 1 million time steps, is only able to do as well as the trajectory optimization combined with essentially uniformly distributed random noise of the uninitialized policy from CoTO-PPO.

Figure 8.5 shows the percentage of samples picked by the policy network of PPO in CoTO-PPO that outperform the actions from the trajectory optimization over training.

[1] `https://youtu.be/mv2xw83NyWU`

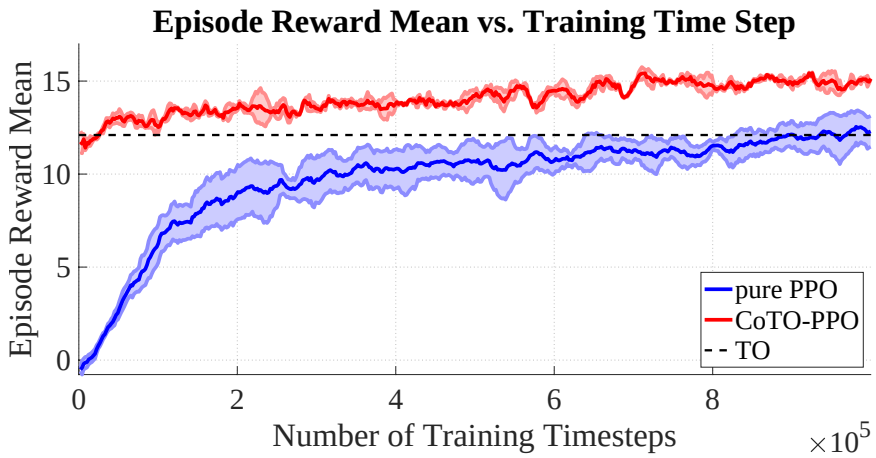

Figure 8.4: Episode reward mean for pure PPO, and cooperative trajectory optimization and PPO (CoTO-PPO). The episode reward mean from using only the trajectory optimization is plotted as a dashed line.

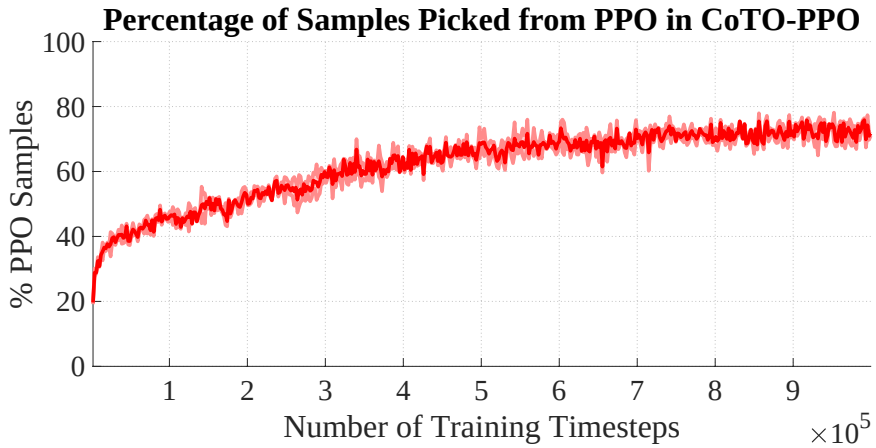

Figure 8.5: At the beginning of training, about 20% of the random actions selected by PPO produce larger rewards than the optimal actions found with the trajectory optimization. Over time the policy is guided towards better regions and roughly 75% of the samples selected from PPO result in maximal reward. As a fail safe, the TO action is taken to keep our bounds on time to goal.

As expected, when the policies are randomly initialized, few samples from PPO will outperform even a suboptimal trajectory optimization. Eventually as training progresses, the percentage of maximal reward samples picked with PPO converges to around 75% of the time. This shows there is still a benefit to using the trajectory optimization as a worst-case scenario, as it is still being picked 25% of the time after 1 million training time steps.

Table 8.1 details the reward mean and percentage of PPO actions picked (if relevant) with various algorithms and scenarios across 100 trials, after training for 1 million time steps. We do trials of evaluating each policy by sampling from the output Beta distributions stochastically, as well as deterministically evaluating the distributions with the maximum likelihood estimate. We see that combining the trajectory optimization with PPO significantly increases the mean reward, with our method CoTO-PPO having the best performance. If we evaluate only the policy trained from CoTO-PPO, it is still a significant improvement over the policy trained from pure PPO alone. We also evaluate the effect of combining the policy trained with pure PPO with the trajectory optimization, labeled CoTO-(pure PPO), which makes it clear that pure PPO has learned a suboptimal policy, as the combination with the TO leads to a larger reward mean.

In this latter case, we also track what percent of the time CoTO-(pure PPO) picks the TO action vs. the action selected from the policy network of pure PPO. Despite training for 1 million time steps, our algorithm finds that the trajectory optimization performs better than the policy trained from scratch roughly half of the time. In comparison, the policy trained from our method CoTO-PPO is picked over 75% of the time, despite far fewer on-policy samples (due to using the TO samples for supervised updates).

	Reward Mean		Percent PPO Actions	
Algorithm	Stochastic	Deterministic	Stochastic	Deterministic
pure PPO	12.9	13.8	-	-
Trajectory Optimization	-	12.1	-	-
CoTO-PPO	15.1	**15.7**	75	**81**
CoTO-PPO, policy only	14.1	14.7	-	-
CoTO-(pure PPO)	14.5	14.5	44	57

Table 8.1: Episode reward mean and percent of samples chosen with PPO for different algorithms across 100 trials, using either stochastic or deterministic (maximum likelihood) actions from the output distributions of the policy network. The percent of PPO actions are only listed for algorithms which choose between both DRL and TO actions, and the trajectory optimization action is always evaluated deterministically.

8.5.3 Maximum Instantaneous Reward Discussion

A first look at the algorithm may seem to imply that it is greedy, rather than optimal, as the agent selects the action leading to the maximum instantaneous reward, rather than a function of expected returns. We experimented with simulating taking multiple actions from the TO and from RL over varying horizons, but found significantly worse performance with this method. One plausible explanation for this result is that, when first initialized, the PPO actor network is essentially taking random actions. As the horizon h increases on which we simulate taking h actions from TO and RL separately, the expected returns of taking a series of random actions regresses toward 0 under our reward scheme, and thus the probability that RL will outperform the TO tends to 0. Said another way, it becomes increasingly unlikely to have multiple "lucky actions" from RL in a row during exploration as the horizon h increases. Since the agent will correspondingly almost always choose the actions from the suboptimal TO, the policy network will almost always be updated with supervised learning updates and will correspondingly converge, approximately, to this same suboptimal policy.

On the other hand, by using the maximum instantaneous transition reward from a horizon of 1, there is a much higher probability of sampling a "good" action from an unini-

tialized random policy. This allows for more efficient exploration of the environment than by overwhelmingly following the (suboptimal) trajectory optimization actions, leading to a better overall policy (such as more aggressive throttle values, steering angles during turns, etc.), while still ensuring a reasonable worst-case scenario action from the more dynamically conservative TO solution, for cases in which we sample a worse-performing action with RL. The short horizon also avoids overfitting to the suboptimal trajectory optimization expert.

8.6 Conclusion

In this work we have shown the benefits of combining trajectory optimization and deep reinforcement learning methods into one training process. Using these two methods cooperatively allows for online use of our algorithm at any point in the training process, knowing that the worst-case scenario will be as good as a model-based trajectory optimization. This additionally leads to much greater sample efficiency, and avoids unnecessary exploration of randomly initialized policies, towards avoiding local optima.

Even if the trajectory optimization is suboptimal due to mismatching dynamics or overly conservative constraints, there is a clear advantage to incorporating prior knowledge of the system to speed up and guide learning. We also observe that trained policies, whether exclusively learned with deep reinforcement learning or from our combined method, are likely to converge to local optima and cannot exhaustively span all observation states, showing the benefit of model-based methods as a proven fail-safe option. The need to be able to put bounds on learned policies and guarantee some sort of behavior is clear, and this work presents preliminary steps in this direction.

The method detailed in this chapter can be readily applied to any robotic system, and should be an effective way to reduce sampling complexity, accelerate training, guide

the policy search, deploy policies online at any point in the training process, and give an upper bound estimate on time-to-goal through the trajectory optimization.

Chapter 9

Conclusion

In this thesis, we have presented several methods for robot locomotion, ranging from designing trajectories by hand and model-based trajectory optimization, to deep reinforcement learning and the intersection of each of these areas. We proposed two different methods for combining model-based methods with deep reinforcement learning that resulted in better sample efficiency and thus faster training, and more robust control policies. Our algorithm CoTO-PPO, in particular, can be run online with any motion planner potentially replacing the trajectory optimization (TO), to steadily improve the learned control policy with more experience. We also observed that choosing the right action space in DRL, and including simple prior knowledge such as forward and inverse kinematics, can have a huge impact on training and performance, especially for high degree-of-freedom systems such as RoboSimian. These ideas are a first step in incorporating well-studied model-based methods into learning, towards fast, efficient, and more robust control policies.

In general, it is not obvious what the best method may be to fuse control theory with learning. Deep learning methods are still brittle, and heavily dependent on data and hyperparameters, while providing no guarantees on performance or robustness. And yet,

there is experimental evidence that such methods can outperform human designers and model-based methods. However, it is possible that existing or developed models may help put deep learning 'on the right track' so to speak, towards guiding learning towards the optimal policy by incorporating prior knowledge about a given problem.

Agile robots that account for their dynamic environments have the potential to better react in the face of uncertainty. Wheel-legged systems in particular can exploit benefits from two heavily studied research areas in this respect, and also enjoy the speed and efficiency of locomotion on wheels with the versatility and agility to surmount various obstacles of legged systems. For RoboSimian, we analyzed the robustness of our designed trajectories on varying terrain, and formulated the locomotion problem for general wheeled systems in a versatile optimization framework for motions possibly involving wheel slip. We showed that our proposed wheel model can be used to plan both slipping and non-slipping motions, and a future direction to investigate is how to best fit empirical data to this wheel model. We also noticed the emergence of non-intuitive optima, both in our optimization framework with the Acrollbot and in DRL when training in joint space with RoboSimian. We showed that such principles can be used to stabilize and locomote a system in ways that human designers would not have attempted or anticipated.

Bibliography

[1] Boston Dynamics, "What's New, Atlas?." https://youtu.be/fRj34o4hN4I, November, 2017. Accessed 2019-11-14.

[2] Boston Dynamics, "Parkour Atlas." https://youtu.be/LikxFZZO2sk, October, 2018. Accessed 2019-11-14.

[3] Boston Dynamics, "More Parkour Atlas." https://youtu.be/_sBBaNYex3E, September, 2019. Accessed 2019-11-14.

[4] Boston Dynamics, "Introducing Handle." https://youtu.be/-7xvqQeoA8c, February, 2017. Accessed 2019-11-14.

[5] Boston Dynamics, "Handle Robot Reimagined for Logistics." https://youtu.be/5iV_hB08Uns, March, 2019. Accessed 2019-11-14.

[6] H.-W. Park, P. Wensing, and S. Kim, *Online planning for autonomous running jumps over obstacles in high-speed quadrupeds*, in *Proceedings of Robotics: Science and Systems*, (Rome, Italy), July, 2015.

[7] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, *Dynamic locomotion in the mit cheetah 3 through convex model-predictive control*, in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1–9, Oct, 2018.

[8] G. Bledt, M. J. Powell, B. Katz, J. Di Carlo, P. M. Wensing, and S. Kim, *Mit cheetah 3: Design and control of a robust, dynamic quadruped robot*, in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2245–2252, Oct, 2018.

[9] 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)*, pp. 7448–7454, May, 2019.

[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)*, pp. 6295–6301, May, 2019.

[11] D. W. Haldane, M. M. Plecnik, J. K. Yim, and R. S. Fearing, *Robotic vertical jumping agility via series-elastic power modulation*, *Science Robotics* **1** (2016), no. 1.

[12] M. Kovac, M. Fuchs, A. Guignard, J.-C. Zufferey, and D. Floreano, *A miniature 7g jumping robot*, in *2008 IEEE International Conference on Robotics and Automation*, pp. 373–378, IEEE, 2008.

[13] M. Posa, S. Kuindersma, and R. Tedrake, *Optimization and stabilization of trajectories for constrained dynamical systems*, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1366–1373, IEEE, 2016.

[14] A. Hereid, E. A. Cousineau, C. M. Hubicki, and A. D. Ames, *3d dynamic walking with underactuated humanoid robots: A direct collocation framework for optimizing hybrid zero dynamics*, in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1447–1454, May, 2016.

[15] M. Morari and J. H. Lee, *Model predictive control: past, present and future*, *Computers & Chemical Engineering* **23** (1999), no. 4-5 667–682.

[16] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. Scokaert, *Constrained model predictive control: Stability and optimality*, *Automatica* **36** (2000), no. 6 789–814.

[17] L. Grüne and J. Pannek, *Nonlinear model predictive control*, in *Nonlinear Model Predictive Control*, pp. 45–69. Springer, 2017.

[18] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, *Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot*, *Autonomous Robots* **40** (2016), no. 3 429–455.

[19] S. Feng, E. Whitman, X. Xinjilefu, and C. G. Atkeson, *Optimization-based full body control for the DARPA Robotics Challenge*, *Journal of Field Robotics* **32** (2015), no. 2 293–312.

[20] E. Guizzo and E. Ackerman, *DARPA Robotics Challenge: A Compilation of Robots Falling Down*, *IEEE Spectrum: Technology, Engineering, and Science News* (June, 2015). Accessed 2019-11-14.

[21] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver, *Emergence of locomotion behaviours in rich environments*, *CoRR* **abs/1707.02286** (2017) [arXiv:1707.0228].

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, *CoRR* **abs/1707.06347** (2017) [arXiv:1707.0634].

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, *CoRR* **abs/1509.02971** (2015).

[24] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, *Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning*, in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7559–7566, IEEE, 2018.

[25] S. Gu, E. Holly, T. Lillicrap, and S. Levine, *Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates*, in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3389–3396, May, 2017.

[26] T. Haarnoja, A. Zhou, S. Ha, J. Tan, G. Tucker, and S. Levine, *Learning to walk via deep reinforcement learning*, *CoRR* **abs/1812.11103** (2018) [arXiv:1812.1110].

[27] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, *Sim-to-real: Learning agile locomotion for quadruped robots*, *CoRR* **abs/1804.10332** (2018) [arXiv:1804.1033].

[28] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, *Learning agile and dynamic motor skills for legged robots*, *Science Robotics* **4** (2019), no. 26 [https://robotics.sciencemag.org/content/4/26/eaau5872.full.pdf].

[29] B. W. Satzinger, C. Lau, M. Byl, and K. Byl, *Experimental results for dexterous quadruped locomotion planning with Robosimian*, in *Proc. 2014 Int. Symp. on Exp. Robotics (ISER)*, pp. 33–46, 2016.

[30] B. W. Satzinger, C. Lau, M. Byl, and K. Byl, *Tractable locomotion planning for Robosimian*, *The International J. of Robotics Research* **34** (2015), no. 13 1541–1558.

[31] K. Byl, M. Byl, and B. Satzinger, *Algorithmic optimization of inverse kinematics tables for high degree-of-freedom limbs*, in *Proc. ASME Dynamic Systems and Control Conference (DSCC)*, 2014.

[32] P. Hebert, M. Bajracharya, J. Ma, N. Hudson, A. Aydemir, J. Reid, C. Bergh, J. Borders, M. Frost, M. Hagman, J. Leichty, P. Backes, B. Kennedy, K. Karplus, K. Byl, B. Satzinger, K. Shankar, and J. Burdick, *Mobile manipulation and mobility as manipulation–design and algorithms of Robosimian*, *Journal of Field Robotics (Special Issue on the DRC)* **32** (2015), no. 2 255–274.

[33] S. Karumanchi, K. Edelberg, I. Baldwin, J. Nash, J. Reid, C. Bergh, J. Leichty, K. Carpenter, M. Shekels, M. Gildner, D. Newill-Smith, J. Carlton, J. Koehler, T. Dobreva, M. Frost, P. Hebert, J. Borders, J. Ma, B. Douillard, P. Backes, B. Kennedy, B. Satzinger, C. Lau, K. Byl, K. Shankar, and J. Burdick, *Team Robosimian: Semi-autonomous mobile manipulation at the 2015 DARPA Robotics Challenge finals*, *J. of Field Rob.* **34** (2017), no. 2 305–332.

[34] M. Byl and K. Byl, *Design of fast walking with one- versus two-at-a-time swing leg motions for Robosimian*, in *Proc. IEEE Int. Conf. on Tech. for Practical Robot Apps. (TePRA)*, 2015.

[35] K. Byl and F. Tobler, *Bang-bang trajectory plans with dynamic balance constraints: Fast rotational reconfigurations for Robosimian*, in *Proc. ASME Dynamic Systems and Control Conference (DSCC)*, 2014.

[36] S. Hirose and H. Takeuchi, *Study on roller-walk (basic characteristics and its control)*, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, pp. 3265–3270, 1996.

[37] G. Endo and S. Hirose, *Study on roller-walker (system integration and basic experiments)*, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, pp. 2032–2037, 1999.

[38] G. Endo and S. Hirose, *Study on roller-walker (multi-mode steering control and self-contained locomotion)*, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, pp. 2808–2814, 2000.

[39] G. Endo and S. Hirose, *Study on roller-walker - adaptation of characteristics of the propulsion by a leg trajectory -*, in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1532–1537, 2008.

[40] G. Endo and S. Hirose, *Study on roller-walker - energy efficiency of roller-walk -*, in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5050–5055, 2011.

[41] K. Inagaki and N. I. b. Azlizan, *Skating motion by a leg-wheeled robot with passive wheels*, in *Proc. IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pp. 524–529, 2013.

[42] N. Ziv, Y. Lee, and G. Ciaravella, *Inline skating motion generator with passive wheels for small size humanoid robots*, in *Proc. IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pp. 1391–1395, 2010.

[43] O. Matsumoto, S. Kajita, and K. Komoriya, *Flexible locomotion control of a self-contained biped leg-wheeled system*, in *Proc. IEEE/RSJ International*

Conference on Intelligent Robots and Systems (IROS), vol. 3, pp. 2599–2604, IEEE, 2002.

[44] N. Takasugi, K. Kojima, S. Nozawa, Y. Kakiuchi, K. Okada, and M. Inaba, *Real-time skating motion control of humanoid robots for acceleration and balancing*, in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1356–1363, IEEE, 2016.

[45] C. Iverach-Brereton, J. Baltes, J. Anderson, A. Winton, and D. Carrier, *Gait design for an ice skating humanoid robot*, *Robotics and Autonomous Systems* **62** (2014), no. 3 306–318.

[46] F. Cordes, C. Oekermann, A. Babu, D. Kuehn, T. Stark, F. Kirchner, and D. Bremen, *An active suspension system for a planetary rover*, in *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, pp. 17–19, 2014.

[47] W. Reid, F. J. Pérez-Grau, A. H. Göktoğan, and S. Sukkarieh, *Actively articulated suspension for a wheel-on-leg rover operating on a martian analog surface*, in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5596–5602, May, 2016.

[48] P. R. Giordano, M. Fuchs, A. Albu-Schaffer, and G. Hirzinger, *On the kinematic modeling and control of a mobile platform equipped with steering wheels and movable legs*, in *2009 IEEE International Conference on Robotics and Automation*, pp. 4080–4087, May, 2009.

[49] A. Suzumura and Y. Fujimoto, *Real-time motion generation and control systems for high wheel-legged robot mobility*, *IEEE Transactions on Industrial Electronics* **61** (July, 2014) 3648–3659.

[50] M. Giftthaler, F. Farshidian, T. Sandy, L. Stadelmann, and J. Buchli, *Efficient kinematic planning for mobile manipulators with non-holonomic constraints using optimal control*, in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3411–3417, May, 2017.

[51] C. Grand, F. Benamar, and F. Plumet, *Motion kinematics analysis of wheeled–legged rover over 3d surface with posture adaptation*, *Mechanism and Machine Theory* **45** (2010), no. 3 477 – 495.

[52] B. H. Wilcox, *Athlete: A limbed vehicle for solar system exploration*, in *2012 IEEE Aerospace Conference*, pp. 1–9, IEEE, 2012.

[53] T. Klamt and S. Behnke, *Anytime hybrid driving-stepping locomotion planning*, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4444–4451, IEEE, 2017.

[54] M. Kamedula, N. Kashiri, and N. G. Tsagarakis, *On the kinematics of wheeled motion control of a hybrid wheeled-legged centauro robot*, in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2426–2433, Oct, 2018.

[55] M. Hutter, C. Gehring, A. Lauber, F. Gunther, C. D. Bellicoso, V. Tsounis, P. Fankhauser, R. Diethelm, S. Bachmann, M. Blösch, *et. al.*, *Anymal-toward legged robots for harsh environments*, *Advanced Robotics* **31** (2017), no. 17 918–931.

[56] Y. de Viragh, M. Bjelonic, C. D. Bellicoso, F. Jenelten, and M. Hutter, *Trajectory optimization for wheeled-legged quadrupedal robots using linearized zmp constraints*, *IEEE Robotics and Automation Letters* **4** (April, 2019) 1633–1640.

[57] M. Bjelonic, P. K. Sankar, C. D. Bellicoso, H. Vallery, and M. Hutter, *Rolling in the deep – hybrid locomotion for wheeled-legged robots using online trajectory optimization*, 2019.

[58] M. Geilinger, R. Poranne, R. Desai, B. Thomaszewski, and S. Coros, *Skaterbots: Optimization-based design and motion synthesis for robotic creatures with legs and wheels*, *ACM Trans. Graph.* **37** (July, 2018) 160:1–160:12.

[59] F. Borrelli, P. Falcone, T. Keviczky, J. Asgari, and D. Hrovat, *Mpc-based approach to active steering for autonomous vehicle systems*, 2005.

[60] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, *Kinematic and dynamic vehicle models for autonomous driving control design*, in *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1094–1099, June, 2015.

[61] E. Velenis and P. Tsiotras, *Minimum time vs maximum exit velocity path optimization during cornering*, in *Proceedings of the IEEE International Symposium on Industrial Electronics, 2005. ISIE 2005.*, vol. 1, pp. 355–360, June, 2005.

[62] E. Velenis, D. Katzourakis, E. Frazzoli, P. Tsiotras, and R. Happee, *Steady-state drifting stabilization of rwd vehicles*, *Control Engineering Practice* **19** (2011), no. 11 1363 – 1376.

[63] J. Z. Kolter, C. Plagemann, D. T. Jackson, A. Y. Ng, and S. Thrun, *A probabilistic approach to mixed open-loop and closed-loop control, with application to extreme autonomous driving*, in *2010 IEEE International Conference on Robotics and Automation*, pp. 839–845, May, 2010.

[64] E. Jelavic, J. Gonzales, and F. Borrelli, *Autonomous drift parking using a switched control strategy with onboard sensors*, *IFAC-PapersOnLine* **50** (2017), no. 1 3714 – 3719. 20th IFAC World Congress.

[65] F. Zhang, J. Gonzales, K. Li, and F. Borrelli, *Autonomous drift cornering with mixed open-loop and closed-loop control*, *IFAC-PapersOnLine* **50** (2017), no. 1 1916 – 1922. 20th IFAC World Congress.

[66] F. Zhang, J. Gonzales, S. E. Li, F. Borrelli, and K. Li, *Drift control for cornering maneuver of autonomous vehicles*, *Mechatronics* **54** (2018) 167 – 174.

[67] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, *Aggressive driving with model predictive path integral control*, in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1433–1440, May, 2016.

[68] C. Voser, R. Y. Hindiyeh, and J. C. Gerdes, *Analysis and control of high sideslip manoeuvres*, *Vehicle System Dynamics* **48** (2010), no. sup1 317–336, [https://doi.org/10.1080/00423111003746140].

[69] J. Y. Goh and J. C. Gerdes, *Simultaneous stabilization and tracking of basic automobile drifting trajectories*, in *2016 IEEE Intelligent Vehicles Symposium (IV)*, pp. 597–602, June, 2016.

[70] H. Pacejka, *Tire and vehicle dynamics*. Elsevier, 2005.

[71] H. B. Pacejka and E. Bakker, *The magic formula tyre model*, *Vehicle System Dynamics* **21** (1992), no. sup001 1–18, [https://doi.org/10.1080/00423119208969994].

[72] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2007.

[73] K. Popp and W. Schiehlen, *Ground vehicle dynamics*. Springer Science & Business Media, 2010.

[74] M. Guiggiani, *The science of vehicle dynamics*, *Pisa, Italy: Springer Netherlands* (2014).

[75] M. Posa and R. Tedrake, *Direct trajectory optimization of rigid body dynamical systems through contact*, in *Algorithmic foundations of robotics X*, pp. 527–542. Springer Tracts in Adv. Robotics, 2013.

[76] M. Posa, C. Cantu, and R. Tedrake, *A direct method for trajectory optimization of rigid bodies through contact*, *International Journal of Robotics Research* **33** (2014), no. 1 69–81.

[77] D. E. Stewart and J. C. Trinkle, *An implicit time-stepping scheme for rigid body dynamics with Coulomb friction*, in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pp. 162–169, 2000.

[78] D. E. Stewart and J. C. Trinkle, *An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction*, *International Journal for Numerical Methods in Engineering* **39** (1996), no. 15 2673–2691.

[79] A. V. Rao, *A survey of numerical methods for optimal control*, *Advances in the Astronautical Sciences* **135** (2009), no. 1 497–528.

[80] J. T. Betts, *Survey of numerical methods for trajectory optimization*, *Journal of Guidance, Control, and Dynamics* **21** (1998), no. 2 193–207, [https://doi.org/10.2514/2.4231].

[81] M. A. Patterson and A. V. Rao, *Gpops-ii: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming*, *ACM Trans. Math. Softw.* **41** (Oct., 2014) 1:1–1:37.

[82] M. P. Kelly, *Transcription methods for trajectory optimization: a beginners tutorial*, *arXiv preprint arXiv:1707.00284* (2017).

[83] M. Kelly, *An introduction to trajectory optimization: How to do your own direct collocation*, *SIAM Review* **59** (2017), no. 4 849–904.

[84] K. Mombaur, *Using optimization to create self-stable human-like running*, *Robotica* **27** (2009), no. 3 321–330.

[85] G. Schultz and K. Mombaur, *Modeling and optimal control of human-like running*, *IEEE/ASME Transactions on Mechatronics* **15** (Oct, 2010) 783–792.

[86] C. R. HARGRAVES and S. W. PARIS, *Direct trajectory optimization using nonlinear programming and collocation*, *Journal of Guidance, Control, and Dynamics* **10** (1987), no. 4 338–342, [https://doi.org/10.2514/3.20223].

[87] M. Buss, M. Hardt, J. Kiener, M. Sobotka, M. Stelzer, O. von Stryk, and D. Wollherr, *Towards an autonomous, humanoid, and dynamically walking robot: Modeling, optimal trajectory planning, hardware architecture, and experiments*, in *Proceedings of the 3rd International Conference on Humanoid Robotics*, pp. 2491–2496, 2003.

[88] C. D. Remy, *Optimal exploitation of natural dynamics in legged locomotion.* PhD thesis, ETH Zurich, 2011.

[89] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[90] J. Kober, J. A. Bagnell, and J. Peters, *Reinforcement learning in robotics: A survey*, *The International Journal of Robotics Research* **32** (2013), no. 11 1238–1274.

[91] Y. Li, *Deep reinforcement learning: An overview*, *arXiv preprint arXiv:1701.07274* (2017).

[92] L. Tai, J. Zhang, M. Liu, J. Boedecker, and W. Burgard, *A survey of deep network solutions for learning control in robotics: From reinforcement to imitation*, *arXiv preprint arXiv:1612.07139* (2016).

[93] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters, *et. al.*, *An algorithmic perspective on imitation learning*, *Foundations and Trends® in Robotics* **7** (2018), no. 1-2 1–179.

[94] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, *CoRR* **abs/1502.05477** (2015) [arXiv:1502.0547].

[95] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, *CoRR* **abs/1506.02438** (2015).

[96] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[97] S. Kakade and J. Langford, *Approximately optimal approximate reinforcement learning*, in *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, (San Francisco, CA, USA), pp. 267–274, Morgan Kaufmann Publishers Inc., 2002.

[98] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous methods for deep reinforcement learning*, *CoRR* **abs/1602.01783** (2016).

[99] M. Bain and C. Sommut, *A framework for behavioural claning*, *Machine intelligence* **15** (1999), no. 15 103.

[100] K. Hauser, *Robust contact generation for robot simulation with unstructured meshes*, in *Proc. International Symposium on Robotics Research (ISRR)*, pp. 357–373. Springer, 2016.

[101] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa, *Biped walking pattern generation by using preview control of zero-moment point*, in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2, pp. 1620–1626, IEEE, 2003.

[102] K. Byl, A. Shkolnik, S. Prentice, N. Roy, and R. Tedrake, *Reliable dynamic motions for a stiff quadruped*, in *Proc. International Symposium on Experimental Robotics (ISER 2008)*, pp. 319–328, Springer, 2009.

[103] G. Bellegarda, K. van Teeffelen, and K. Byl, *Design and evaluation of skating motions for a dexterous quadruped*, in *2018 IEEE Int. Conf. on Robotics and Automation (ICRA)*, pp. 1703–1709, May, 2018.

[104] N. E. Du Toit and J. W. Burdick, *Robot motion planning in dynamic, uncertain environments*, *IEEE Transactions on Robotics* **28** (2012), no. 1 101–115.

[105] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, *Sequential composition of dynamically dexterous robot behaviors*, *International Journal of Robotics Research* **18** (June, 1999) 534–555.

[106] K. Byl, T. Strizic, and J. Pusey, *Mesh-based switching control for robust and agile gaits*, in *Proc. American Control Conference (ACC) [in press]*, 2017.

[107] C. O. Saglam and K. Byl, *Quantifying and optimizing robustness of bipedal walking gaits on rough terrain*, in *Proc. Internationa Symposium Robotics Research (ISRR)*, 2015.

[108] C. O. Saglam and K. Byl, *Robust policies via meshing for metastable rough terrain walking*, in *Proc. Robotics: Science and Systems X (RSS X)*, p. Paper 49, RSS, 2014.

[109] W. Xi, Y. Yesilevskiy, and C. D. Remy, *Selecting gaits for economical locomotion of legged robots*, *Int Journal of Robotics Research [online first]* (Nov, 2015). doi 10.1177/0278364915612572.

[110] I. Mordatch, J. M. Wang, E. Todorov, and V. Koltun, *Animating human lower limbs using contact-invariant optimization*, *ACM Transactions on Graphics (TOG)* **32** (2013), no. 6 203.

[111] R. Tedrake, T. Zhang, and H. Seung, *Stochastic policy gradient reinforcement learning on a simple 3d biped*, in *Proc. of IEEE Int. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3, pp. 2849–2854 vol.3, 2004.

[112] N. Kohl and P. Stone, *Policy gradient reinforcement learning for fast quadrupedal locomotion*, in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 3, pp. 2619–2624 Vol.3, April, 2004.

[113] R. Calandra, A. Seyfarth, J. Peters, and M. P. Deisenroth, *Bayesian optimization for learning gaits under uncertainty*, *Annals of Mathematics and Artificial Intelligence* **76** (Feb., 2016) 5–23.

[114] M. W. Spong, *Swing up control of the acrobot using partial feedback linearization*, in *Proc. American Control Conference (ACC)*, pp. 2158–2162, 1994.

[115] M. Srinivasan, *Why walk and run: energetic costs and energetic optimality in simple mechanics-based models of a bipedal animal*. Cornell University Ithaca, NY, 2006.

[116] P. L. Kapitza, *A pendulum with oscillating suspension*, *Uspekhi Fizicheskikh Nauk* **44** (1951) 7–20.

[117] G. Bellegarda, N. Talele, and K. Byl, *Nonintuitive optima for dynamic locomotion: The Acrollbot*, in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, pp. 3130–3136, May, 2018.

[118] N. Talele and K. Byl, *Methods and performance analyses for design and feedback control of efficient and robust planar biped walking*, in *2019 American Control Conference (ACC)*, pp. 4567–4572, July, 2019.

[119] E. Todorov, T. Erez, and Y. Tassa, *Mujoco: A physics engine for model-based control*, in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, Oct, 2012.

[120] G. Bellegarda and K. Byl, *Trajectory optimization for a wheel-legged system for dynamic maneuvers that allow for wheel slip*, in *2019 IEEE International Conference on Decision and Control (CDC), in Press.*

[121] S. Kajita and B. Espiau, *Legged robots*, in *Springer Handbook of Robotics*, pp. 361–389. Springer, 2008.

[122] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, *CasADi – A software framework for nonlinear optimization and optimal control*, *Mathematical Programming Computation* **11** (2019), no. 1 1–36.

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

[124] Y. Wu, E. Mansimov, S. Liao, R. B. Grosse, and J. Ba, *Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation*, *CoRR* **abs/1708.05144** (2017).

[125] A. Y. Ng, D. Harada, and S. J. Russell, *Policy invariance under reward transformations: Theory and application to reward shaping*, in *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, (San Francisco, CA, USA), pp. 278–287, Morgan Kaufmann Publishers Inc., 1999.

[126] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016.

[127] G. Bellegarda and K. Byl, *Training in task space to speed up and guide reinforcement learning*, in *2019 IEEE International Conference on Intelligent Robots and Systems (IROS), in Press.*

[128] R. Martín-Martín, M. Lee, R. Gardner, S. Savarese, J. Bohg, and A. Garg, *Variable impedance control in end-effector space. an action space for reinforcement learning in contact rich tasks*, in *Proceedings of the International Conference of Intelligent Robots and Systems (IROS)*, 2019.

[129] S. Ross, G. J. Gordon, and J. A. Bagnell, *No-regret reductions for imitation learning and structured prediction*, *CoRR* **abs/1011.0686** (2010) [arXiv:1011.0686].

[130] I. Mordatch and E. Todorov, *Combining the benefits of function approximation and trajectory optimization*, in *Robotics: Science and Systems (RSS)*, 2014.

[131] S. Levine and V. Koltun, *Variational policy search via trajectory optimization*, in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, (USA), pp. 207–215, Curran Associates Inc., 2013.

[132] S. Levine and V. Koltun, *Guided policy search*, in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1–9, PMLR, 17–19 Jun, 2013.

[133] L. Xie, S. Wang, S. Rosa, A. Markham, and N. Trigoni, *Learning with training wheels: Speeding up training with a simple controller for deep reinforcement learning*, *CoRR* **abs/1812.05027** (2018) [arXiv:1812.0502].

[134] K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch, *Plan online, learn offline: Efficient learning and exploration via model-based control*, in *International Conference on Learning Representations*, 2019.

[135] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2019.

[136] P.-W. Chou, D. Maturana, and S. Scherer, *Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution*, in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, (International Convention Centre, Sydney, Australia), pp. 834–843, PMLR, 06–11 Aug, 2017.