LEARNING AND CONTROL WITH INACCURATE MODELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

J. Zico Kolter
August 2010

Abstract

A key challenge in applying model-based Reinforcement Learning and optimal control methods to complex dynamical systems, such as those arising in many robotics tasks, is the difficulty of obtaining an accurate model of the system. These algorithms perform very well when they are given or can learn an accurate dynamics model, but often times it is very challenging to build an accurate model by any means: effects such as hidden or incomplete state, dynamic or unknown system elements, and other effects, can render the modeling task very difficult.

This work presents methods for dealing with such situations, by proposing algorithms that can achieve good performance on control tasks even using only *inaccurate* models of the system. In particular, we present three algorithmic contributions in this work that exploit inaccurate system models in different ways: we present an approximate policy gradient method, based on an approximation we call the Signed Derivative, that can perform well provided only that the sign of certain model derivative terms are known; we present a method for using a distribution over possible inaccurate models to identify a linear subspace of control policies that perform well in all models, then learn a member of this subspace on the real system; finally, we propose an algorithm for integrating previously observed trajectories with inaccurate models in a probabilistic manner, achieving better performance than is possible with either element alone.

In addition to these algorithmic contributions, a central focus of this thesis is the application of these methods to challenging robotic domains, extending the state of the art. The methods have enabled a quadruped robot to cross a wide variety of challenging terrain, using a combination of slow static walking, dynamic trotting gaits, and dynamic jumping maneuvers. We also apply these methods to a full-sized autonomous car, where they enable it to execute a "powerslide" into a narrow parking spot, one of the most challenging maneuvers demonstrated on an autonomous car. Both these domains represent highly challenging robotics tasks where the dynamical system is difficult to model, and our methods demonstrate that we can attain excellent performance on these tasks even without an accurate model of the system.

To my mom, Janine, my dad, Roberto, and my wife, KD.

Acknowledgments

This thesis has been the product of a long and close collaboration with my advisor, Andrew Ng. I am hugely indebted to Andrew for all the help, guidance, and insight he has provided to me over the years. While I couldn't succinctly describe all the ways in which he has helped shaped my academic mindset, the way he approaches new research problems, how he analyzes theoretical questions, and his logical mindset on "diagnostics" for machine learning methods have all become an integral part of how I look at any new problems. In addition, his enthusiasm and happiness to talk about any new ideas have constantly made it a joy to work with him.

I want to thank the other members of my reading committee for their help with this work. Daphne Koller, in addition to providing a great many detailed comments on this work, has for the past year been an amazing source of knowledge and insight. Often times upon hearing just briefly about an algorithm or idea I had been working on, she would immediately have an idea for building upon it in a new and interesting way. Sebastian Thrun has been a constant source of inspiration, and in all my talks with him always exudes a genuine excitement about how the work we do in AI can have a real effect on the world.

I also want to thank the other members of my orals committee: Stephen Boyd and Ilan Kroo. Stephen's classes have profoundly shaped the way I approach almost every new problem I look at, and his enthusiasm for talking about any new application or algorithmic idea is truly inspirational. I only met Ilan later in my PhD program, and his knowledge and intuition about aerodynamics, as well as his enthusiam, has been wonderful to experience.

Recently I have begun several projects relating to energy issues, and Carrie Armel has been extremely helpful in getting me involved in this research. Her passion and excitement about getting new people involved in this work has made me feel welcome and even more excited about this new area of research.

Looking back, I owe a large debt of gratitude to Mark Maloof, my undergraduate advisor who first got me interested in machine learning research. Mark's enthusiasm and commitment to clear and engaging presentation of machine learning work is one of the main reasons why I got so excited about this field in the first place.

I want to thank my officemates, Morgan Quigley and Honglak Lee, as well as the members of our frequent lunch groups, Adam Coates and Pieter Abbeel. Numerous times I've been grateful to have Honglak around for commiseration during late night paper writing sessions. Talking with Morgan and Adam about various potential robot platforms and algorithms, both plausible and wildly implausible versions, has been one of the most fun parts of the past five years; it will also be a huge loss not to be able to constantly bug Morgan with questions about any and every mechanical or engineering issue I'd run into. Working with Pieter was another highlight of my earlier PhD experience; his excitement in talking about new algorithms and ideas has always been an inspiration.

I'm very grateful for having had the chance to work with and talk to many of the other members of Andrew's group, both past and present. I want to thank Ashutosh Saxena, Tom Do, Rion Snow, Rajat Raina, Olga Russakovsky, Quoc Le, Andrew Maas, Andrew Saxe, and Jiquan Ngiam.

Thanks to all those who I have collaborated with on various research and papers: Sam Schreiber, Mike Rodgers, Danny Jachowski, Yi Gu, Charles DuHadway, Youngjun Kim, Christian Plagemann, and David Jackson. In the last year, I have also begun working with Zouhair Mahboubi, Geoff Bower, and Tao Wang. Their constant excitement in our projects, even when it means getting to Gates at 6:30 AM to go fly RC airplanes, has made this work a true joy, even when we were all sleep deprived.

During the two quarters that I TA'd Andrew's machine learning class I had the pleasure of working with a number of other students: Sam Ieong, Erick Delage, Haidong Wang, Catie Chang (both times!), Tom Do, Dan Ramage, Joseph Koo and

Paul Baumstarck. Sam and Tom in particular were invaluable in taking a huge part of the workload, and made even the experience of grading problem sets until two in the morning seem less of a burden.

Finally, moving away from the academic sphere, I can't imagine getting through the past several years without the support of family and friends. To my mom, Janine, my dad, Roberto, and my sister, Amanda, I want to thank you for all your support and encouragement.

Lastly, and most importantly, I want to thank my companion throughout my time at Stanford and beyond, my wife KD. Your unconditional love and support has sustained me through this all, your strength has inspired me, and I will be forever grateful for all that you do. Thank you from the bottom on my heart.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work focuses on controlling complex dynamical systems, such as a legged robot climbing over large obstacles and a full-sized autonomous car sliding sideways over the ground. In these and many other control tasks a similar theme emerges: the challenge of creating an accurate model of the system. Many control methodologies rely on the presence of an accurate model in order to obtain good performance, and there exists a variety of methods for creating such models; the models can be built from physical principles or learned from data collected on the real system. But often times it is very challenging to build an accurate model even using all these methods: there can be hidden state in the system that is difficult to sense or model, leading to non-Markovian dynamical effects; there are often properties of the system that are constantly changing, with no way to know these properties a priori; or the dynamics of the system may simply be sufficiently complex such that efficient simulation of the system is not possible. In all these settings it is useful, then, to consider how we may develop learning and control methods that rely only on *inaccurate* models of the system in order to achieve good performance.

1.1 Model-based Reinforcement Learning, Optimal Control, Robust and Adaptive Control

Broadly speaking, the fields of model-based Reinforcement Learning (RL) (Sutton and Barto, 1998) and optimal control (Bertsekas, 2005a,b; Stengel, 1994) provide a canonical set of methods for controlling dynamical systems.[1] While there are numerous different approaches to model-based RL and optimal control (we will describe the framework as well as some of the more relevant algorithms in much greater detail in Chapter 2) such methods typically work in the following manner. As input, these algorithms take a dynamical model of the system of interest and a cost function that specifies what constitutes good or bad behavior, and they then output a control strategy that attempts to minimize this cost function in the dynamical system. While the algorithms can work in many different ways, intuitively, we can think of such approaches as simulating multiple possible experiences in the dynamics model, then outputting the control strategy that performs best (i.e., minimizes the cost function) in the model.

There are many things that could potentially go wrong with this approach. For instance, the cost function may incorrectly capture the desired notion of good behavior in the system. Alternatively, the reinforcement learning or optimal control algorithm may fail to minimize the desired criterion. However, in practice we find that for nearly all cases, for the types of control problems we consider here, the fault lies in the dynamics model: the resulting controller output by the algorithm typically performs very well in the dynamical model we provide to the algorithm, but does not perform well in the real world. Thus, the fundamental issue often is one of model accuracy: we simply do not have a sufficiently accurate model of the real system to perform well on the task of interest. Indeed, one of the fundamental challenges in model-based RL and optimal control is the task of either coming up with suitably accurate dynamics model, or avoiding the need for an accurate model entirely.

A number of classical control methodologies exist that attempt to remedy the

[1]Model-based RL and optimal control generally refer to a very similar body of algorithms and methodologies, so we make no distinction between the two for the purposes of this work.

problem described above. Perhaps the simplest method to address the issue of an inaccurate dynamical system is to attempt to use data from the real system to *learn* a better dynamics model; this is the strategy employed by methods such as adaptive control (Sastry and Bodson, 1994; Astrom and Wittenmark, 1994) or many methods of system identification (Ljung, 1999). In a similar vein, a great deal of work in model-based RL focuses on methods for exploring the state space to extent that guarantee a sufficiently accurate learned dynamics model (Strehl et al., 2009; Kearns and Singh, 2002); recent work also focuses on the ability to use expert demonstrations when learning such dynamics models (Abbeel and Ng, 2005). However, the problem here, which we alluded to previously, is that often times it is still very challenging to develop an accurate model of the system, even *given* a large amount of data from the real system: effects like hidden state/non-Markovian dynamics, or unknown and constantly changing parameters render certain problems unsuitable to such approaches.

This situation, then, motivates the use of *inaccurate* models in control. Of course, if we concede the ability to develop an accurate dynamics model of the system of interest, the question naturally arises as to what, and how, we can learn from such a model. The classical control methodology that deals most directly with inaccurate models is the area of robust control (Zhou et al., 1996). Broadly speaking, robust control deals with scenarios where we do not know the true model of the system, but where we have some distribution or uncertainty set over possible models, and the goal is to find a controller that performs well in *all* these models (or with high probability given a distribution over possible models). One difficulty of this method is that, in order to ensure that the controller does indeed perform well in all models, the result is often a very conservative control law, that in exchange for robustness does not perform optimally in *any* of the models in question. There have also been approaches that combine elements of both adaptive and robust control (Ioannou and Sun, 1995), where one identifies a system along with uncertainty bounds suitable for robust control methods, but these methods still inherit the general limitations of both these approaches.

1.2 Overview of Contributions

The work we present in this thesis also deals with the question of how to learn good control strategies given an inaccurate model of the system, but it does so in a different manner from the techniques described above. Unlike adaptive control or system identification, the methods we present here will typically make no efforts to learn an improved model of the dynamical system. Indeed, we will often use a learned model as our *inaccurate* model, so the assumption is that we have exhausted all our options in terms of building a good model, and now wish to use the model we have to the best of our abilities. Likewise, our methods differ from robust control in that we do not seek a control system that performs well in a wide variety of models; rather, we want a control policy that performs well in just one system: the real system in the world. Having said this, there are certainly manners in which our work overlaps with adaptive and robust control techniques, and we will highlight these connections in the individual chapters. But as a general theme, the task we focus on is as follows: we want to obtain optimal (or as close to optimal as possible) performance on the real system, through techniques that exploit an inaccurate dynamics model of the system.

While most of the chapters in this thesis present algorithmic or theoretical contributions, a key contribution of this work is the application of these algorithms to real-world problems. Indeed, the examples of a quadruped robot climbing over large obstacles and a robotic car sliding over gravel, described previously, are real-world tasks that have motivated many of the algorithms we present here. In particular, much of the work in this thesis centers around applications to the LittleDog robot (Murphy et al., 2010), a small quadruped robot intended for negotiating rough terrain. Although we have spent a great deal of time attempting to develop an accurate simulation model of the robot, we have repeatedly found that it is very difficult to come up with an accurate simulator of the real robotic system, due to hard-to-model effects like backlash in the gears and the friction of the robot's feet and body sliding over the ground. Thus, many of the techniques we present in this paper, designed to operate using inaccurate models, were developed precisely for this real-world task of learning and control with this quadruped robot. Using our methods, we achieve state-of-the-art results in terms of both navigating over large obstacles, and executing dynamic maneuvers/gaits in the presence of large obstacles.[2]

Likewise, the previously mentioned task of sliding an autonomous car sideways over gravel is another real-world task to which we apply these methods. In particular, Chapter 5 presents a method for using inaccurate models that we apply to the task of "powersliding" a full-sized autonomous car ("Junior," Stanford's entry into the DARPA Urban Challenge (Montemerlo et al., 2008)) into a narrow parking spot. Again the challenge that arises in this task is one of model accuracy: effects like friction of the car's tires sliding over the gravel are very difficult to model, especially when we do not sense all the relevant quantities, such as the speed of the individual tires of the car. Despite this, our methods are able to accurately and repeatedly slide the car into a narrow target area, representing the state of the art in terms of accurately controlling a full-sized car in such a maneuver.

1.3 Outline of Thesis

The chapters in this work are organized as follows:

- **Chapter 2: Background.** Here we present background on reinforcement learning and optimal control that forms the basis for the algorithms in future sections. Since RL and optimal control are obviously very broad fields, we focus here particularly on those elements that we later build upon in our own algorithms. In particular, after presenting the formal framework of optimal control, we focus in particular on Policy Gradient and Linear Quadratic Regulator (LQR) methods as two techniques for solving optimal control tasks.

[2]Several different institutions have been simultaneously working with the LittleDog platform, through the Learning Locomotion program, a competitive effort to build the software that enables this robot to cross challenging terrain. The robot and program will be described much more in Chapter 6. Here we just claim that the sum total of all this work, including our own, represents the state of the art in quadruped locomotion, and leave a more detailed detailed description of the individual strengths and weaknesses of our approach in this later chapter.

- **Chapter 3: Approximate Policy Gradient via the Signed Derivative.** This chapter looks at one of the fundamental questions that arises when dealing with learning control policies from inaccurate models: how can an inaccurate model, unsuitable for directly learning a control policy, still help us obtain good performance on the system of interest? The criterion we look at in this chapter states that an inaccurate model can still be used to improve performance when the *signs* of certain dominant model derivative terms are correct; this motivates the development of a highly simplified form of model, which we call the signed derivative, that only specifies the signs of these dominant derivative terms. We show, both theoretically and empirically, that policy gradient methods can exploit such a model to obtain good performance on a system without ever using an accurate model. We evaluate this approach on a number of tasks, including a simulated two-link arm, an autonomous RC car, and a quadruped robot learning to jump up steps.

- **Chapter 4: Dimensionality Reduction in Policy Search.** This chapter also deals with policy search and policy gradient methods, but exploiting a different type of inaccuracy. In particular, we assume here that we have a parametrized dynamics model that is described by some number of free parameters, and we have a distribution over these parameters encoding our uncertainty in the model. However, unlike robust control procedures, which try to find a single policy that performs well over this entire distribution, we use dimensionality reduction techniques to identify a *linear subspace* that contains (near) optimal control laws for models drawn from that distribution. This enables us to greatly reduce the number of parameters we need to learn for control policies, and enables us to efficiently learn a policy on the real system, using either the techniques described in Chapter 3 or via model-free techniques. We demonstrate the approach on a task of learning fast omni-directional locomotion on a quadruped.

- **Chapter 5: Multi-model Control for Mixed Closed-loop/Open-loop Behavior.** This chapter deals with the question of what happens when there is a dynamical system that we simply cannot model accurately, even in terms of its dominant derivative signs or a distribution over parameters. In such a setting it may appear as though there is little chance that we will be able to control the system as desired; however, a common feature of many control tasks, even in highly challenging domains, is that they are often remarkably deterministic over short periods of time. This motivates a control strategy where we use models to control the system in regions where the model is accurate, and execute previously observed trajectories “open-loop” in regions where the model is inaccurate, but where we have observed a previous demonstration that accomplishes the desired behavior. In this chapter we develop a probabilistic approach for such a strategy, which smoothly trades off between LQR-based control and open loop based upon a measure of model variance. We first evaluate the method on a standard cart-pole benchmark, and then show that it is able to achieve very good performance on the challenging task of sliding a full-sized autonomous car into a narrow parking spot.

- **Chapter 6: Application to the LittleDog Robot** In this chapter we focus on developing a control system that allows a quadruped robot to quickly and robustly cross a wide variety of challenging terrain. As mentioned one of the key challenges that drives our approaches to this problem is that accurately modeling the LittleDog, in particular how its feet will react with different portions of the terrain, is a highly challenging task. This chapter differs from previous chapters in that the focuses is largely on the application itself, and we present a variety of different algorithms that we have developed for this task. We show that the final system is able to repeatedly cross a wide variety of challenging terrain, with obstacles as large as the robot’s legs.

1.4 First Published Appearances of Contributions

Much of the material here has appeared in previous publications, though in some chapters substantial portions have also been expanded. The signed derivative algorithm in Chapter 3 first appeared in (Kolter and Ng, 2009a), though the algorithm description has been greatly expanded upon. Similarly, the dimensionality reduction for policy gradient method was first published in (Kolter and Ng, 2007), though again the algorithm has been expanded upon here. Our work on multiple model control for mixed open-loop and closed-loop behavior, and its application to the an autonomous sliding parking maneuver in Chapter 5 appears in (Kolter et al., 2010). Finally, the LittleDog work we present in Chapter 6 encompasses a number of papers: (Kolter et al., 2008a,b; Kolter and Ng, 2009b).

Chapter 2

Background

In this chapter, we present a basic background on learning and control techniques that provide the foundation for the later algorithmic chapters. Machine learning and control are clearly both large fields of research, so we focus here particularly on those algorithms that we will later build upon in subsequent chapters. In particular, this chapter will mainly introduce the notation for dynamical systems and the optimal control framework, and focus on two methods for optimizing controllers based upon a model: policy-gradient methods and Linear Quadratic Regulator (LQR) based algorithms. For broader introductions to the topics, a number of references are available (e.g., Bertsekas, 2005a; Sutton and Barto, 1998; Stengel, 1994).

2.1 Dynamical Systems and Optimal Control

2.1.1 States and Controls

Foremost in the technical description of dynamical systems is the notion of a system *state*. Although "state" can potentially refer to a number of things, informally, the state captures those elements of the system that are relevant to its evolution and control, but which typically are *not* directly specified by the agent. For instance, for the quadruped robot, the state of the robot itself may include the 3D position and orientation of the robot's center in space, as well as the the joint angles for each of its legs plus velocities for all these terms; for a car the system state may consist of the robot's position, orientation, and steering wheel angle, again plus velocities for these terms. Throughout this work we will use the notation

$$s \in \mathbb{R}^n \tag{2.1}$$

to denote the state of the system, where n denotes the dimension of the state space. Often times there is much greater structure than this definition implies (for instance, if the state space is discrete), but because we want to work with a general representation for states, we will adopt this notation.

Controls, in contrast, specify those elements, relevant to the evolution of the system, that the agent can specify directly. For instance, in the quadruped robot, the controls may correspond to torques on the robot joints; the distinction here is that while it is not possible for the agent to directly command the state of the system (we could not, for instance, direct the robot to immediately assume some position or orientation), it *is* possible to directly command control torques to the robot's joints (for our sake ignoring the complexity of the joint motors themselves, so we assume the ability to instantaneously apply a torque to one of the motors). Similarly, for the car, the controls could correspond to torque applied to the steering wheel, or throttle/brake commands. We will use the notation

$$u \in \mathbb{R}^m \tag{2.2}$$

to denote controls, where m in the dimension of the control input.

2.1.2 First-order Markov Models

Given these notions of states and controls, we must then determine how the state evolves as a function of past states and control inputs. Typically this is done either in a *continuous-time* manner, by specifying the time derivative $\dot{s}$ as a function of past states and controls (thus specifying the evolution of the state as an ordinary differential equation), or in a *discrete-time* manner, specifying the next state, denoted

s_{t+1}, as a function of previous states and controls. We will focus on the discrete-time case in this work, both for intuitive simplicity of computing state updates (computing the next state from past events will involve just evaluation a function, rather than numerically integrating a differential equation), and because in practice, we typically control any actual system at a discrete set of points. However, we note that all the results presented here can be carried over to the continuous time case with little difficulty.

In this discrete-time setting, we can formalize the notion of a model as some function that predicts the next state given past states and control

$$s_{t+1} = f(s_t, u_t, s_{t-1}, u_{t-1}, \ldots). \tag{2.3}$$

However, in practice such general models can be quite cumbersome, as the next state could depend on a long history of controls. Thus, in this work, as is common in the control literature, we will restrict our attention to *first-order Markov models*, a subset of dynamical systems where the next state depends only on the current state and control

$$s_{t+1} = f(s_t, u_t). \tag{2.4}$$

Despite the simplification, we are actually not loosing a great deal of expressive power with this limitation; since the state can be augmented to include a history of past states, any model that depends on a finite number of previous states can controls can be represented as a first-order Markov model. This framework also helps to clarify the definition of states and controls; in a first-order Markov model, the state and control input contain everything needed to determine the next state of the system, where the controls and states respectively represent those quantities that the agent can and cannot directly affect.

2.1.3 Stochastic models

There are many times that we want to capture some element of randomness or stochasticity in the model. Indeed, when represented using typical state and control spaces, complex systems like quadruped robots and cars exhibit some amount of stochasticity: even given perfect knowledge of the current state and control, no model could predict the next state with perfect accuracy, because the next states can actually differ slightly even for identical initial states and controls. Note that this use of stochasticity is independent of whether there are "real" stochastic effects in the world. What's occurring here is that our parametrization of the state is typically incomplete: we choose some representation of the system's state so that "most" of the system's evolution can be described using a first-order Markov model, but there are almost always certain elements that affect the system evolution to some degree, but that we either cannot observe or simply don't want to include in the model. In a car, for instance, it may be the case that the next state could be affected by small differences in how much the tire in worn on the different wheels; but we may have no desire to actual include tire wear on each wheel as a state in our model, because it is both hard to measure and has a relatively negligible effect the system. Thus, we can instead add some amount of stochasticity to the system, both to capture any "real" stochasticity that may exist in the world, but also to account for unmodeled effects due to our choice of state representation. Indeed, this notion of "stochasticity as a proxy for model inaccuracy" will be further discussed and exploited in later chapters.

For the present time, however, we simply note that stochasticity is often a desirable element in models, and we can represent such stochasticity by a model of the form

$$s_{t+1} = f(s_t, u_t) + \epsilon_t \tag{2.5}$$

where ϵ_t denotes some zero-mean noise term, such as a Gaussian random variable with covariance Σ:

$$\epsilon_t \sim \mathcal{N}(0, \Sigma). \tag{2.6}$$

Indeed, we will often use noise terms of this form, but it is important to note that (2.5) is a fully general means for representing any stochastic model (assuming we allow ϵ_t to depend arbitrarily on the state, control and time), and that (2.6) is a very specific special case, that is no longer fully general.

2.1.4 Control policies

A control *policy* (also referred to simply as a *controller*) provides a means for choosing control actions based upon the current state. Formally, we define a policy

$$\pi : \mathbb{R}^n \to \mathbb{R}^m \tag{2.7}$$

simply as a mapping from states to actions. In practice, however, because the state space can be large and continuous, we typically want to work with some restricted class of policies.

In this thesis, we will typically work with *parametrized* policies, where the policy function is specified by some small set of parameters $\theta \in \mathbb{R}^k$. We will use the notation

$$u_t = \pi(s_t; \theta) \tag{2.8}$$

to denote that, for instance, u_t is given by the policy π, evaluated for state s_t, and parametrized by θ. One common example of a parametrized policy is a *linear* policy where the controls are prescribed as a linear function of the state

$$\pi(s_t; \theta) = K s_t \tag{2.9}$$

where $K \in \mathbb{R}^{m \times n}$ is a matrix of the policy parameters, with $\theta = \{K\}$. Notice a subtle distinction here: for notational simplicity in many cases, we will typically use θ to denote a *vector* of the policy parameters. However, in the case of linear policies, we need a *matrix* to output m controls for n state variables; thus, we use a different symbol (typically K) to denote this matrix, and use θ to represent a vector containing all the elements of K. In control terminology, the matrix K is often called the *gain matrix* in this setting.

Another common form of policies are those linear in *state features.* Formally,

$$\pi(s_t; \theta) = K\phi(s_t) \tag{2.10}$$

where $\phi : \mathbb{R}^n \to \mathbb{R}^p$ is a function mapping from states to some p-dimensional feature vector, $K \in \mathbb{R}^{m\times p}$ is again a matrix of parameters, and again $\theta = \{K\}$. Notice that this is a completely general policy: because ϕ can be an arbitrary feature mapping, we can actually represent any (time independent) parametrized policy in this form.

Finally, as we will see below, there are times where it is actually desirable to use *stochastic* control policies. As with the dynamics model, we can for example define a stochastic policy by adding a zero-mean noise term to the output of a deterministic policy

$$\pi(s_t;\theta) = K\phi(s_t) + \nu_t \tag{2.11}$$

where $\nu_t \sim \mathcal{N}(0,\Lambda)$. Unlike the dynamics model, stochastic policies are often viewed as an undesirable element, introducing artificial noise into the system where none existed before. Thus, as we will show shortly, while stochastic policies can be useful for learning methods, one often removes this stochasticity when applying the final policy on the real system.

2.1.5 Optimal Control

Finally, in order to determine how to choose a policy that can control some dynamical system, it is necessary to specify what constitutes "good" behavior in this system. The typical method for doing this is through a *cost function*, that specifies the "badness" of a given state (and possibly control). We formally define cost functions as mappings from states and controls to a real number:

$$C : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}. \tag{2.12}$$

The goal of a control policy in the optimal control framework is then to minimize the *expected sum of costs* over some *time horizon* H. Formally, we define the *value function* for a policy π and state s starting at time t, denoted $J_t^\pi(s)$ (also called the cost-to-go function) as the sum of expected costs, starting at time t, in state s, and acting according to policy π

$$J_t^\pi(s) = \mathbf{E}\left[\sum_{t'=t}^{H} C(s_{t'},\pi(s_{t'})) \,\middle|\, s_{t'+1} = f(s_{t'},\pi(s_{t'})) + \epsilon_{t'}, s_t = s\right]. \tag{2.13}$$

When we use the notation J^π, the subscript $t = 0$ is implied, and we will typically only use a subscript when defining a value function for some other time horizon. The value function J^π_t also satisfies a recurrence relation know as *Bellman's equation*

$$J^\pi_t(s) = C(s, \pi(s)) + \mathbf{E}\left[J^\pi_{t+1}(f(s, \pi(s)) + \epsilon)\right] \tag{2.14}$$

and with $J^\pi_H(s) = C(s, \pi(s))$. Intuitively, this relation says that the value of a state is equal to the cost of that state, plus the expected value of the next state, following policy π.

The *optimal policy*, denoted $\pi^\star$, is the policy that minimizes the value function (2.13) over all possible policies. The value of function of this optimal policy, denoted $J^\star$, obeys its own version of Bellman's equation, sometimes referred to as Bellman's optimality equation to distinguish it from the case above:

$$J^\star_t(s) = \min_u \left\{C(s, u) + \mathbf{E}\left[J^\star_{t+1}(f(s, u) + \epsilon)\right]\right\}. \tag{2.15}$$

In certain situations, such as discrete state spaces, or linear systems with quadratic costs (a case that we will discuss at length below), it is possible to use the Bellman optimality equation to analytically solve for the optimal value function, which in turn gives us the optimal policy at time t

$$\pi^\star_t(s) = \arg\min_u \left\{C(s, u) + \mathbf{E}\left[J^\star_{t+1}(f(s, u) + \epsilon)\right]\right\}. \tag{2.16}$$

In most cases, though, computing an exact solution to Bellman's optimality equation is intractable, and so we must resort to approximate methods or methods that only find locally optimal policies. Indeed, much of the remainder of this chapter will focus on two methods for finding approximately optimal control laws: policy gradient approaches, and Linear Quadratic Regulator (LQR) techniques.

2.1.6 Markov Decision Processes

The learning and control community, and in particular the Reinforcement Learning community, often uses the machinery of Markov Decision Processes (MDPs) to formalize the optimal control framework. Intuitively, an MDP is simply a structure that contains all the elements described previously in this section: state and control spaces, a dynamics model (also called transition probabilities), a cost function (or reward function, which just corresponds to negative cost), a time horizon, and a distribution over initial states, and a cost function. Formally, using the notation from this chapter, an MDP is a tuple $M = (S, U, P, D, H, C)$ where S and U are the state and control spaces; P is a set of transition probabilities given by the dynamics model, D is a distribution over initial states, H is the time horizon, and C is the cost function. There are many slightly different formulations for MDPs, such as the those with a reward function instead of a cost function, or those with an infinite (possibly discounted) value function instead of the finite time horizon we use above. However, these are minor differences, and the basic machinery and algorithms for MDPs are very similar across all the different formulations. Thus, we include this section just to note the connection, as much other work and many later chapters will use the MDP formulation specifically. For a more complete introduction to MDPs, see e.g., Putterman (2005).

2.2 Multi-variate Calculus

This thesis makes extensive use of calculus in multiple variables. While the actual calculus we use is actually very simple (almost entirely differential calculus), the notation may be initially confusing, as there are actually a variety of different notations in use for these functions. Thus here we briefly review some of the notation that we will use in the remainder of the thesis.

2.2.1 The Gradient

The *gradient* of a function is defined for *real-valued* functions with *vector or matrix* inputs $f : \mathbb{R}^{m\times n} \to \mathbb{R}$. The gradient is a vector (or matrix) of equal size as the function input, and is defined as

$$\nabla_A f(A) \in \mathbb{R}^{m\times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}, \tag{2.17}$$

i.e., the ijth entry of $\nabla_A f(A)$ is equal to the derivative of $f(A)$ with respect to the ijth entry of A.

2.2.2 The Jacobian

The *Jacobian* of a function is defined for *vector-valued* functions with *vector* inputs $f : \mathbb{R}^n \to \mathbb{R}^m$. The Jacobian is an $m \times n$ matrix, defined as

$$\frac{\partial f(x)}{\partial x} \in \mathbb{R}^{m\times n} = \begin{bmatrix} \frac{\partial f(x)_1}{\partial x_1} & \cdots & \frac{\partial f(x)_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x)_m}{\partial x_1} & \cdots & \frac{\partial f(x)_m}{\partial x_n} \end{bmatrix}, \tag{2.18}$$

i.e., the ijth entry of $\frac{\partial f(x)}{\partial x}$ is equal to the the derivative of $f(x)_i$ with respect to x_j. Because Jacobians behave much like scalar derivatives, we use the same notation, but is is important to remember that $\frac{\partial f(x)}{\partial x}$ is a *matrix* when used for multi-variate functions and inputs. It is also important to note the subtle difference between the Jacobian and the gradient: for a real-valued vector function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is the *transpose* of the Jacobian

$$\nabla_x f(x) = \left(\frac{\partial f(x)}{\partial x}\right)^T. \tag{2.19}$$

A useful property of the Jacobian, completely analogous to the scalar case, is the chain rule. For functions $f : \mathbb{R}^k \to \mathbb{R}^m$ and $g : \mathbb{R}^n \to \mathbb{R}^k$,

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(y)}{\partial y}\frac{\partial g(x)}{\partial x}, \quad y \equiv g(x) \tag{2.20}$$

A useful special case of this rule applies when we have f that is explicitly a function of two vector arguments, $f : \mathbb{R}^k \times \mathbb{R}^\ell \to \mathbb{R}^m$, $g : \mathbb{R}^n \to \mathbb{R}^k$, $h : \mathbb{R}^n \to \mathbb{R}^\ell$,

$$\frac{\partial f(g(x), h(x))}{\partial x} = \frac{\partial f(y,z)}{\partial y}\frac{\partial g(x)}{\partial x} + \frac{\partial f(y,z)}{\partial z}\frac{\partial h(x)}{\partial x}, \quad y \equiv g(x), z \equiv h(x). \tag{2.21}$$

2.3 Policy Gradient Methods

When global optimization of the control policy is intractable, one of the simplest and most common methods for finding a good policy is via policy gradient techniques. In this section we use the terminology of parametrized policies as defined above, and we use $J(\cdot;\theta)$ as shorthand for $J^{\pi(\cdot;\theta)}$, the value function of the policy parametrized by $\theta \in \mathbb{R}^k$. Policy gradient methods use a simple gradient descent rule to find the parameters that minimize the value function. In particular, if we let

$$\nabla_\theta J(s;\theta) \in \mathbb{R}^k \tag{2.22}$$

be the gradient of the value function, evaluated as state s, with respect to the parameters (this quantity itself is known as the policy gradient), then by performing the simple update

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(s;\theta) \tag{2.23}$$

for a small step-size $\alpha \in \mathbb{R}_+$, we will adjust θ so as to incrementally improve $J(s;\theta)$. Under certain smoothness conditions, this process will find a locally optimal set of control parameters. Also note that if we desire a policy that performs well over some distribution D over states, we can just as easily compute gradients with respect to

$$J(D;\theta) = \mathbf{E}_{s\sim D}\left[J(s;\theta)\right] \tag{2.24}$$

by using sampling methods, for instance.

The key to policy gradient approaches, of course, is how we compute these gradient terms $\nabla_\theta J(s;\theta)$. This section discusses several methods for doing so in different scenarios.

2.3.1 Known deterministic model

The most straightforward scenario for computing policy gradients is the case of a known deterministic model, (2.4). First, note that we can easily compute $J(s;\theta)$ by the procedure

1. Run the policy for H steps: $s_{t+1} = f(s_t, \pi(s_t))$ for $t = 0, \ldots, H-1$, with $s_0 = s$.
2. Sum the costs: $J(s;\theta) = \sum_{t=0}^{H} C(s_t, \pi(s_t))$.

Given this (deterministic) method for computing $J(s,\theta)$ we can easily compute its gradient with respect to θ by finite differencing, for example: we simply make minor adjustments by adding some some small δ to one of the elements of θ, then recompute the value function and divide by δ:

$$(\nabla_\theta J(s,\theta))_i \approx \frac{J(s,\theta+\delta e_i) - J(s,\theta)}{\delta} \tag{2.25}$$

where $e_i \in \mathbb{R}^k$ denotes the ith unit basis (a vector whose elements are all zero except for the ith element, which is one).

While finite differencing is useful in its simplicity, if we are able to compute derivatives of the model analytically, then we can also compute the policy gradient analytically by applying the chain rule (a strategy known as real-time recurrent learning in the Neural Network community (Williams and Zisper, 1989)). The material here is more advanced than that in the remainder of this chapter, and is not crucial for its understanding, but the way in which we derive the equations here will be used in later chapters as well. To determine an analytical form of the policy gradient, we can apply the chain rule to see that

$$\frac{\partial C(s_t,u_t)}{\partial s_t} = \frac{\partial C(s_t,u_t)}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial u_t}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial u_t}{\partial \theta}, \tag{2.26}$$

and

$$\frac{\partial s_{t+1}}{\partial \theta} = \frac{\partial f(s_t,u_t)}{\partial \theta} = \frac{\partial f(s_t,u_t)}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial f(s_t,u_t)}{\partial u_t}\frac{\partial u_t}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial f(s_t,u_t)}{\partial u_t}\frac{\partial u_t}{\partial \theta}. \tag{2.27}$$

By also noting that

$$\nabla_\theta J(s;\theta) = \sum_{t=0}^{H} \nabla_\theta C(s_t,u_t) = \sum_{t=0}^{H} \left(\frac{\partial C(s_t,u_t)}{\partial \theta}\right)^T \tag{2.28}$$

we can derive a simple algorithm for computing the gradient that maintains the matrix $G = \frac{\partial s_t}{\partial \theta}$, uses this term to compute each element of the policy gradient, and updates this term according to (2.27). Thus, the procedure is as follows:

- Initialize $s_0 \leftarrow s$, $\nabla_\theta J(s;\theta) \leftarrow 0$, $G \leftarrow 0$.
- For $t = 0, \ldots, H-1$,
  1. $u_t \leftarrow \pi(s_t;\theta)$, $s_{t+1} \leftarrow f(s_t,u_t)$
  2. $\nabla_\theta J(s;\theta) \leftarrow \nabla_\theta J(s;\theta) + G^T \left(\frac{\partial C(s_t,u_t)}{\partial s_t} + \frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial s_t}\right)^T$
     $+ \left(\frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial \theta}\right)^T$
  3. $G \leftarrow \left(\frac{\partial f(s_t,u_t)}{\partial s_t} + \frac{\partial f(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial s_t}\right) G + \frac{\partial f(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial \theta}$

2.3.2 Known stochastic model

When the dynamics model of the system is known, but includes a stochastic noise term as in (2.5), then additional techniques are needed: simply computing the policy gradient ignoring the noise term will not give an unbiased estimate of the true gradient. The most common means for computing policy gradients in this domain is via sampling: we compute the policy gradient (by any of the methods above) using some *fixed* $\epsilon_1, \ldots, \epsilon_H$ drawn by random sampling. We repeat this process a number of times and average all the resulting policy gradients to get our final estimate of the gradient. This is known as the PEGASUS algorithm (Ng and Jordan, 2000), and it can be shown that this procedure produces an unbiased estimate of the gradient, and furthermore that given a suitable number of samples it will be close to the true gradient with high probability.

2.3.3 Unknown deterministic or stochastic model

Finally, we note that there are algorithms which can compute an estimate of the policy gradient even when an explicit model is *unknown* (but, of course, assuming we can execute control policies in the system). Because these methods are not the focus of this thesis, we just introduce the basic ideas here, and leave the numerous extensions of these methods to the references. However, since this thesis is about *inaccurate* models (which are hopefully easy to obtain), the methods presented here are often the best comparisons to our approaches, as the relevant question is: how much can it help to have an inaccurate model versus no model at all?

The method presented here is known in the context of Reinforcement Learning as the REINFORCE, or Episodic REINFORCE algorithm (Williams, 1992), but it is based on a method known as likelihood ratio gradient estimation (Glynn, 1987). Unlike the previous approaches, this method and extensions require *stochastic* policies, so we will use the notation

$$p(u|s;\theta) \tag{2.29}$$

to denote the probability density of action u given state s, when executing the policy parametrized by θ. Typically this takes the form of a deterministic policy plus noise, as in (2.11).

To simplify the presentation, we will also introduce here the notion of a *trajectory*, denoted τ, which is simply a sequence of states and actions

$$\tau = (s_0, u_0, \ldots, s_H, u_H). \tag{2.30}$$

We also overload the cost notation, so that

$$C(\tau) \equiv \sum_{t=0}^{H} C(s_t, u_t). \tag{2.31}$$

Using this notation, notice that the value function can also be written as

$$J(s;\theta) = \mathbf{E}_\tau[C(\tau)] = \int p(\tau|s_0 = s;\theta)C(\tau)d\tau \tag{2.32}$$

(for the remainder of this section, we will omit the $s_0 = s$ qualification, as this will always be implied).

Given this form of the value function, the policy gradient can be written as

$$\begin{aligned} \nabla_\theta J(s;\theta) &= \nabla_\theta \int p(\tau;\theta)C(\tau)d\tau \\ &= \int \frac{p(\tau;\theta)}{p(\tau;\theta)} \nabla_\theta p(\tau;\theta)C(\tau)d\tau \\ &= \int p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)C(\tau)d\tau \\ &= \mathbf{E}_\tau\left[\nabla_\theta \log p(\tau;\theta)C(\tau)\right]. \end{aligned} \tag{2.33}$$

By this "trick" of introducing the gradient of a log term, we can transform the gradient into an expectation, which we will then approximate using sampling. However, we must first show how to compute the derivative term $\nabla_\theta \log p(\tau;\theta)$. Fortunately, this can be done *without* the need for an explicit dynamics model, since

$$\begin{aligned} \nabla_\theta \log p(\tau;\theta) &= \nabla_\theta \log\left(\prod_{t=0}^{H} p(s_{t+1}|s_t,u_t)p(u_t|s_t;\theta)\right) \\ &= \sum_{t=0}^{H} \left(\nabla_\theta \log p(s_{t+1}|s_t,u_t) + \nabla_\theta \log p(u_t|s_t;\theta)\right) \\ &= \sum_{t=0}^{H} \nabla_\theta \log p(u_t|s_t;\theta) \end{aligned} \tag{2.34}$$

where the $\nabla_\theta \log p(s_{t+1}|s_t,u_t)$ term drops out because it does not depend on θ. Thus, as claimed above, we see that this term can be computed without an analytical model of the system. The complete procedure for computing the gradient is as follows, where we approximate the expectation in (2.33) using M sample trajectories:

- For $i = 1,\ldots,M$, sample a trajectory $\tau^{(i)}$ by starting in state s and executing the policy (with parameters θ) for H steps.

- $\nabla_\theta J(s;\theta) \approx \frac{1}{M}\sum_{i=1}^{M}\left(\sum_{t=0}^{H}\nabla_\theta \log p(u_t^{(i)}|s_t^{(i)};\theta)\right)C(\tau^{(i)})$.

One requirement of the algorithm, as written, is that we need the ability to "reset" the system to the same initial state s and perform multiple simulations of the policy. While this could be avoided by using $M = 1$, this typically produces a very noisy estimate of the gradient, which is of limited use for improving the policy. Thus, these model-free policy gradient methods are typically applied to scenarios where we *can* reset the state to the same or very similar initial state. Furthermore, as mentioned above, the algorithm above is the simplest version of a likelihood ratio policy gradient method; obtaining good performance with such algorithms typically requires more advanced techniques, such as the selection of an optimal baseline (Greensmith et al., 2004) (a bias on the cost, which doesn't change the gradient, but which reduces the variance of the estimate), and natural gradient methods (Kakade, 2001; Bagnell and Schneider, 2003; Peters et al., 2005) (a technique that performs steepest descent updates according to a metric on the trajectory probabilities, rather than the parameter space, and which typically results in much faster convergence). However, these extensions are beyond the scope of this background introduction.

2.4 Linear Quadratic Regulator Methods

Methods based upon linear quadratic regulator (LQR) control form another common family of algorithms for find locally optimal control laws. In contrast to policy gradient methods, which can be applied to any arbitrary form of parametrized policy, LQR methods typically output a very specific form of parametrized policy: (possibly time-varying) open-loop controls plus (also possibly time-varying) linear feedback controllers. LQR control is based upon a special case of continuous state and action dynamics and cost, where the solution to Bellman's optimality equation can be solved analytically. Thus, we begin our presentation with a discussion of this special case. For more detailed discussion of Linear Quadratic methods, there are a number of available references (e.g. Bertsekas, 2005a; Anderson and Moore, 1989; Stengel, 1994)

2.4.1 Linear Dynamics and Quadratic Cost

We begin by considering the special case of a *linear* dynamics model

$$s_{t+1} = As_t + Bu_t \tag{2.35}$$

and a (positive definite) *quadratic* cost function[1]

$$C(s,u) = s^T Qs + u^T Ru, \quad Q \in \mathbb{R}^{n\times n} \succeq 0,\ R \in \mathbb{R}^{m\times m} \succ 0 \tag{2.36}$$

i.e., the goal is to maintain the system at the zero state, $s = 0$, while applying zero control, $u = 0$. In this case, we will show that Bellman's optimality equation admits a closed form solution, and that the optimal value function is quadratic in the state

$$J^\star(s) = s^T Ps \tag{2.37}$$

for some $P \in \mathbb{R}^{n\times n} \succeq 0$. The derivation here is relatively brief: a more detailed derivation is given in the references above. First note that

$$J_H^\star(s) = \min_u (s^T Qs + u^T Ru) = s^T P_H s \quad (\text{for } P_H = Q). \tag{2.38}$$

Now suppose $J_t^\star = s^T P_t s$ for some $P_t \succeq 0$. Then by Bellman's optimality equation we have

$$\begin{aligned} J_{t-1}^\star(s) &= \min_u \left(s^T Qs + u^T Ru + J_t^\star(As + Bu)\right) \\ &= s^T Qs + \min_u \left(u^T Ru + (As + Bu)^T P_t (As + Bu)\right) \\ &= s^T Qs + s^T A^T P_t As + \min_u \left(u^T Ru + u^T B^T P_t Bu + 2s^T A^T P_t Bu\right) \end{aligned} \tag{2.39}$$

[1]Here $A \succeq 0$ and $A \succ 0$ denote positive semidefinite and positive definite matrices respectively.

The u that minimizes the right hand size is given by

$$u_t^\star = (R + B^T P_t B)^{-1} B^T P_t A s \tag{2.40}$$

and substituting this expression back into (2.39), the value function takes the form (after some simplification)

$$J_{t-1}^\star = s^T (Q + A^T P_t A - A^T P_t B (R + B^T P_t B)^{-1} B^T P_t A) s = s^T P_{t-1} s, \tag{2.41}$$

which again is a quadratic form (and, though we don't prove it here, is also positive semidefinite). This implies that the optimal policy in this domain is a (time-varying) *linear feedback policy*

$$\pi_t^\star(s_t) = K_t s_t, \quad K_t \equiv (R + B^T P_t B)^{-1} B^T P_t A. \tag{2.42}$$

Using the notation of parametrized policies from the preceding section, we would say that the policy parameters here are a separate gain matrix for each time step $\theta = \{K_1, K_2, \ldots, K_H\}$ with $K_i \in \mathbb{R}^{m \times n}$, and the LQR algorithm allows us to find the *globally optimal* set of parameters for the case of linear dynamics and quadratic cost.

While the requirement of linear dynamics and quadratic cost may seem overly restrictive, the algorithm can be extended to certain more general settings while maintaining global optimality:

- **Time-varying dynamics and costs**. The algorithm can be easily generalized to *time-varying* linear dynamics models and costs:

  $$s_{t+1} = A_t s_t + B_t u_t, \quad C_t(s, u) = s^T Q_t s + u^T R_t u. \tag{2.43}$$

  We omit the derivation because it is identical to that above, but the resulting optimal value function and optimal policy have the form

  $$\begin{aligned} &J_t^\star(s) = s^T P_t s, \quad \pi_t^\star(s) = (R_t + B_t^T P_t B_t)^{-1} B_t^T P_t A_t s \equiv K_t s \\ &P_{t-1} = Q_t + A_t^T P_t A_t - A_t^T P_t B_t (R_t + B_t^T P_t B_t)^{-1} B_t^T P_t A_t), \quad P_H = Q_H. \end{aligned} \tag{2.44}$$

- **Affine systems**. We can apply LQR to affine systems

$$s_{t+1} = As_t + Bu_t + a \tag{2.45}$$

by adding a constant term to the and applying LQR to the augmented system

$$\bar{s} = \begin{bmatrix} s \\ 1 \end{bmatrix}, \ \bar{A} = \begin{bmatrix} A & a \\ 0 & 1 \end{bmatrix}, \ \bar{B} = \begin{bmatrix} B \\ 0 \end{bmatrix}, \ \bar{Q} = \begin{bmatrix} Q & 0 \\ 0 & 0 \end{bmatrix}, \ \bar{R} = R. \tag{2.46}$$

- **Trajectory tracking**. We can also express costs that penalize deviation from a time-varying *desired state and control*

$$C_t(s,u) = (s - s_t^\star)^T Q (s - s_t^\star) + (u - u_t^\star)^T R (u - u_t^\star). \tag{2.47}$$

While it is possible to solve this equation by augmenting the state as in the previous example, this also requires a cost function that has cross-terms that depend on both the state and control (to account for a term that is linear in the control input). Instead, using the same method as above it can be shown that the optimal value function and policy are of the form

$$\begin{aligned}
&J_t^\star(s) = s^T P_t s + 2q_t^T s + r_t, \quad \pi_t^\star(s) = K_t s_t + g_t \\
&K_t = -(R + B^T P_t B)^{-1} B^T P A, \quad g_t = -(R + B^T P_t B)^{-1}(Bq_t - Ru_t^\star) \\
&P_{t-1} = Q + A^T P_t A - A^T P_t B (R + B^T P_t B)^{-1} B^T P_t A, \quad P_H = Q_H \\
&q_{t-1} = (A + BK_t) q_t - K_t^T R u_t^\star - Q s_t^\star, \quad q_T = -Q s_T^\star \\
&r_{t-1} = r_t + u_t^{\star T} R u_t^\star + s_t^{\star T} Q s_t^\star - (B^T q_t - R u_t^\star)^T g_t, \quad r_T = s_t^{\star T} Q s_t^\star
\end{aligned} \tag{2.48}$$

- **Stochastic models**. LQR can also be easily applied to stochastic linear models of the form

$$s_{t+1} = As_t + Bu_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \Sigma). \tag{2.49}$$

In fact, it turns out that the optimal control law for this system with Gaussian noise is identical to the optimal control law for the system without noise, and thus we can solve this setting by the same method as above. Although we do not prove it formally here, the basic intuition is that, because the noise is independent of the state and control, it has no impact on the optimal policy; the only difference in the stochastic setting is that there is an additional term in the value function, accounting for the additional expected cost due to noise. Thus, in general we won't distinguish between the stochastic and deterministic case for LQR models.

Many other simple extensions to the setting can be solved exactly, and of course we can combine any of these extensions. However, the constraint of linear (or affine) models is still a large restriction, and thus in the next section we discuss how these techniques can be extended (approximately) to non-linear systems.

2.4.2 Non-linear models

Consider a general nonlinear model of the form previously considered

$$s_{t+1} = f(s_t, u_t) \tag{2.50}$$

and initially suppose further that $s = 0, u = 0$, is an *equilibrium point* of the system

$$0 = f(0, 0) \tag{2.51}$$

i.e., starting in state $s = 0$ and applying control $u = 0$ keeps the system in state $s = 0$ (notice that this holds for the linear dynamical systems described above, so we are merely making this requirement to keep the parallels to the linear case as close as possible at first). Again we assume a positive definite quadratic cost function $C(s, u) = s^T Q s + u^T R u$.

To apply LQR to this setting, we *linearize* the dynamics around this equilibrium point, equivalent to taking a first-order Taylor expansion of the non-linear system, to obtain the approximate system matrices

$$A \equiv \frac{\partial f(0,0)}{\partial s}, \quad B \equiv \frac{\partial f(0,0)}{\partial u}. \tag{2.52}$$

We then solve the resulting LQR problem to obtain a controller $\pi_t(s) = K_t s$ exactly in the manner described above. While this controller will no longer be optimal for the non-linear system, in practice these control laws typically perform very well, especially when the system begins in a state "close" to the equilibrium point, so that the linearization does not introduce very much error. Indeed, although the mathematical analysis is beyond the scope of this material, under suitable conditions on the dynamics model the resulting controller is guaranteed to *stabilize* the non-linear system (bring the system to the desired $s = 0$, $u = 0$ state) for some suitable ellipse around the equilibrium point. Indeed, the quadratic cost function of LQR methods can generally be thought of as producing policies that stabilizes the system around some the zero point. For a more detailed explanation of this linearization procedure and the application of LQR to non-linear systems, see e.g., (Anderson and Moore, 1989, Ch. 3) and (Stengel, 1994, Ch. 5).

2.4.3 Non-linear trajectory stabilization

The requirement in the preceding section, that we stabilize the system around the zero state and control, may seem overly restrictive, and indeed the same technique can be applied to stabilize the system around a trajectory. Given a trajectory of the form described earlier in this section

$$\bar{\tau} = (\bar{s}_0, \bar{u}_0, \bar{s}_1, \bar{u}_1, \ldots, \bar{s}_H, \bar{u}_H) \tag{2.53}$$

we consider a quadratic cost function that penalizes deviations from this trajectory

$$C_t(s,u) = (s - \bar{s}_t)^T Q (s - \bar{s}_t) + (u - \bar{u}_t)^T R (u - \bar{u}_t). \tag{2.54}$$

We construct a time-varying LQR task by linearizing the dynamics along the trajectory:

$$A_t = \frac{\partial f(\bar{s}_t, \bar{u}_t)}{\partial s_t}, \quad B_t = \frac{\partial f(\bar{s}_t, \bar{u}_t)}{\partial u_t}, \tag{2.55}$$

and we solve the time-varying LQR task (but with time-invariant Q and R), according to equation (2.44). The resulting policy will be to apply control

$$\pi_t(s) = \bar{u}_t + K_t(s - \bar{s}_t). \tag{2.56}$$

There is a subtle distinction here, but the LQR algorithm we apply here is *not* the "trajectory tracking" LQR cost function described above in (2.47); in particular, we can solve this LQR task using just the normal quadratic equations, without the need for a linear term in the value function. This is because we are linearizing the non-linear system *around* the trajectory $\bar{\tau}$; in effect the linearization implies a (time-varying) linear system in the *trajectory error*

$$\delta s_{t+1} = A_t \delta s_t + B_t \delta u_t, \quad C_t(\delta s_t, \delta u_t) = \delta s_t^T Q \delta s_t + \delta u_t^T R \delta u_t \tag{2.57}$$

where $\delta s_t \equiv s_t - \bar{s}_t$ and $\delta u_t \equiv u_t - \bar{u}_t$ denote the deviation from the desired trajectory. Thus, the final policy specifies the control deviation will be a linear function of the state deviation, $\delta u_t = K_t \delta s_t$, which is equivalent to (2.56) above.

2.4.4 Iterative LQR

Finally, the trajectory stabilization methods described above is still limited in that it requires a priori knowledge of a realizable trajectory (i.e., both states and the controls that result in these states in the model). In this last section, we briefly illustrate how we can iterate LQR techniques to generate a full sequence of open-loop controls, as well as feedback controllers, that minimize some cost function without an a priori trajectory. This general strategy goes by many different names, such as iterative LQR (Li and Todorov, 2005), Gauss-Newton LQR (Boyd, 2003), or sequential linear quadratic methods (Sideris and Bobrow, 2005). The method is highly related to the Differential Dynamic Programming (Jacobson and Mayne, 1970) and the successive sweep method (Dyer and McReynolds, 1970), and largely involves a very minor simplification of these classical control strategies.

Suppose we wish to minimize some trajectory tracking cost function

$$C_t(s,u) = (s - s_t^\star)^T Q (s - s_t^\star) + u^T R u \tag{2.58}$$

in the nonlinear model $s_{t+1} = f(s_t, u_t)$. To solve this problem, we begin with some arbitrary sequence of controls $\bar{u}_0, \ldots, \bar{u}_H$ and execute these controls in the model to obtain the resulting states $\bar{s}_0, \ldots, \bar{s}_H$. We then linearize around this trajectory, as in the previous section but instead solve a time-varying tracking LQR problem with cost

$$C_t(\delta s_t, \delta u_t) = (\delta s_t + \bar{s}_t - s_t^\star)^T Q (\delta s_t + \bar{s}_t - s_t^\star) + (\delta u_t + \bar{u}_t)^T R (\delta u_t + \bar{u}_t) \tag{2.59}$$

under the linear approximation of the error dynamics, $\delta s_{t+1} = A_t \delta s_t + B_t \delta u_t$, described in the previous section. Intuitively, since $u_t \approx \bar{u}_t + \delta u_t$ and $s_t \approx \bar{s}_t + \delta s_t$ for small enough δu_t and δs_t (i.e., in a region close to the linearization point of the system), (2.59) closely approximates the cost function of interest (2.58). Thus, adjusting the controls according to

$$\bar{u}_t \leftarrow \bar{u}_t + \delta u_t = \bar{u}_t + K_t \delta s_t \tag{2.60}$$

will result in a new sequence of states and controls $\bar{s}_0, \bar{u}_0, \ldots, \bar{s}_H, \bar{u}_H$ that typically has lower cost than the previous set of controls and states; we then iterate this process until convergence.

There is, however, the possibility that the algorithm as describe above will not converge, due to the fact that for large δs_t or δu_t, the linearized model may be a poor approximation of the non-linear model. To overcome this difficulty, we can add an additional penalty term on δs_t and δu_t, to ensure that the resulting controller does not deviate too far from the linearization — i.e., we add an additional penalty to the cost function of the form

$$\sum_{t=0}^{H} \left(\delta s_t^T Q_0 \delta s_t + \delta u_t^T R_0 \delta u_t \right). \tag{2.61}$$

By including such a term, and using large enough Q_0 and R_0 matrices such that the resulting δs_t and δu_t do not deviate too much from the linearization point, we can guarantee that the algorithm convergences to a locally optimal set of open loop controls $\bar{u}_t$ and feedback matrices K_t.

2.5 Summary

This chapter presented a briefly background on reinforcement learning and optimal control algorithms. We focused specifically on two categories of algorithms that we will build upon in the subsequent work: policy gradient approaches and linear quadratic regulator methods.

Chapter 3

Approximate Policy Gradient via the Signed Derivative

The previous chapter discussed a number of standard methods for optimizing a policy given a (presumably accurate) model of the system. However, as discussed in the introduction, the theme of this work is the ability to use *inaccurate* models to find good policies. The natural question, then, is how an inaccurate model of the system, which may be unable to control the system using the methods from the previous chapter alone, could still allow us to find policies that perform well.

The algorithm we present in this chapter is based upon the fact that, as we will show shortly, the policy gradient term discussed in the previous chapter can be written such that it depends only on certain model *derivative* terms. The intuition of our algorithm is that while such derivative terms may be hard to determine precisely (since we assume that the system is difficult to model), it is often very easy to estimate the *sign* of these derivative terms. Furthermore, if we compute an approximate policy gradient by simply substituting these signs for the derivatives themselves, the resulting algorithm works well (both theoretically and empirically) in many situations, and does not require an accurate model of the system.

Our method requires an additional constraint on the dynamics as well; for technical reasons that we will discuss in detail shortly, both our theoretical and empirical analysis focuses on dynamical systems where each state variable is primarily controlled by only *one* control variable (although a single control can affect multiple state variables). While most state variables will naturally be affected by all control inputs to some degree, we consider the scenario where each state variable is merely *primarily* affected by one control (i.e., affects of other controls are treated as noise terms), where we will formalize this notion shortly. The rationale behind this focus will be discussed later in this chapter, but for the time being, we merely mention that the algorithm we present here does focus on this restricted class of dynamics models.

To highlight the critical intuition that certain model derivative signs are “easy” to determine, and to illustrate why settings with the “orthogonal” control inputs mentioned above still capture interesting problems, we consider the simple example task of driving a car along a trajectory. In this setting, the state could be the car’s lateral deviation from the trajectory, its orientation relative to the trajectory, and its velocity, the cost could penalize lateral deviation from the trajectory and deviation from a desired velocity, and the policy could determine steering and throttle and a simple (e.g., linear) function of the current state or state features. Consider now the relationship between the car’s lateral deviation from the trajectory and the commanded steering angle: it may be very difficult to determine the precise relationship between how a change in steering angle results in a change in lateral deviation (this would correspond to the *derivative* of lateral deviation states with respect to the steering control input). However, the *sign* of the derivative term in this case is very obvious: turning more to the left typically results in a lateral deviation that is also more to the left. Furthermore, it should be apparent that the car driving domain is one where the above constraint on control inputs and states holds: while the lateral deviation from the trajectory is affected by both the throttle and the steering angle, it is primarily affected by the steering angle. While such “obvious” derivative signs, and constraints on the control inputs, clearly don’t apply to all control tasks, we demonstrate in this paper that they do apply in many interesting domains, and that when they do apply, the approximate policy gradient techniques based on these signed derivatives can perform very well.

3.1 Related Work

The algorithm we present in this chapter relates most to two threads of research in the control and optimization communities. The first of these is the focus on the accuracy of model derivative terms, rather than the predictive accuracy of the model itself. Recalling the policy gradient and LQR approaches described in the previous chapter, note that nowhere in the equations for the policy gradient (2.26) – (2.28), or in the linearization of the LQR model (2.53), does the actual equation of the model itself, $f(s_t, u_t)$, come into play; rather, these terms depend on the model only through its *derivative* terms $\frac{\partial f(s,u)}{\partial s}$ and $\frac{\partial f(s,u)}{\partial u}$. The only way in which these methods actually use the model is for simulating the sequence of states $s_0, \ldots, s_H$ given the control inputs $u_0, \ldots, u_H$. But if we have the ability to run trials on the real system, then we can perform this step on the actual system itself, and if only the model derivative terms are accurate, then we can use the equations above to effectively compute the policy gradient or run LQR on the *real system*. This insight has been observed in a number of different works, such as Jordan and Rumelhart (1992), and recently Abbeel et al. (2006) provide a theoretical analysis and experimental results on such algorithms.

However, although this insight is quite useful, from a practical standpoint the requirement of “only” needing accurate derivatives is not a huge gain. In particular, for the standard model $s_{t+1} = f(s_t, u_t)$, $\frac{\partial f(s,u)}{\partial s}$ contains n^2 terms and $\frac{\partial f(s,u)}{\partial u}$ contains mn terms. The derivative terms will be identical for the model with any added bias

$$s_{t+1} = f(s_t, u_t) + b_t \tag{3.1}$$

where $b_t \in \mathbb{R}^n$ is a (state and control independent) bias term. Requiring only accurate derivative, instead of accurate derivative and prediction, means we don't need to learn this bias term, but in the general case this only reduces the number of model parameters needed by n, and it is unclear if there is any better way to learn the model derivatives than to simply minimize prediction error. Instead, the algorithm we present here requires that we only know the *sign* of the dominant derivative terms (and, in fact, as we will show shortly only the signs of $\frac{\partial f(s,u)}{\partial u}$ and similar terms), a requirement that can be much easier to satisfy, as argued in the car driving example above. Of course, as we will also discuss, the algorithm we present is also more limited in the situations we can apply it to, so there is certainly a trade-off between the two approaches.

The second main area of work which we build upon is work within the optimization community on using sign terms in optimization. The idea here is simple: if we want to minimize some function $f(x)$, we can take small descent direction steps in the direction of the gradient's sign,

$$x \leftarrow x - \alpha \operatorname{sign}(\nabla_x f(x)) \tag{3.2}$$

where the sign function defined in the standard way

$$\operatorname{sign}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases} \tag{3.3}$$

and is applied component-wise to vectors, and where α is some small step size. For small enough step sizes (and assuming certain continuity requirements), this step is guaranteed to decrease the objective; this fact can be shown by simply noting that the sign of the gradient makes a positive inner product with the gradient, $(\nabla_x f(x))^T \operatorname{sign}(\nabla_x f(x)) \geq 0$, which is the condition for a valid descent direction (Boyd and Vandenberg, 2004, pg. 463).

This general technique of using the sign of the gradient term as the update direction has a long history in optimization, machine learning, and control. One of the first algorithms to use this method is the so-called sign-sign LMS algorithm (Dasgupta and Johnson, 1986), a modification of the classical LMS algorithm (Widrow and Hoff, 1960) (a stochastic algorithm for least-squares minimization), where the normal gradient update is replaced by the sign of the gradient; an early application of this technique was used for channel equalization (Lucky, 1966). In this machine learning community, such sign-update methods have been used extensively within neural networks (Anderson, 1986), especially in the context of the RPROP algorithm (Riedmiller and Braun, 1992), which uses the same idea of gradient sign updates, but also adds additional machinery to automatically select step size parameters based on whether the gradient changes sign in successive updates. Indeed, the notion of gradient sign updates was also discussed briefly in (Jordan and Rumelhart, 1992) specifically in the context of optimizing control performance.

The way in which our proposed method differs from these past approaches is that, as we will show shortly, our algorithm does not propose to update the policy parameters via the actual sign of the policy gradient; rather, our method replaces only certain *model derivative* terms in the policy gradient with their signs, but includes many other elements of the gradient which are not signed. Indeed, as we will show, the final approximate policy gradient produced by our approach does *not* necessarily share the same signs as the true policy gradient (although it often does in practice). Thus, we cannot guarantee that the updates produced by our algorithm are descent directions of the value function. However, as we will show below, the updates produced by our approximation *are* descent directions for a modified version of the value function, and one which often minimizes the true value function as well. We will of course discuss these points in greater detail below, but we merely mention this now to highlight the difference between out approach and standard signed optimization methods. Finally, we note that while a gradient descent approach that *did* always have the same signs as the true gradient term may ultimately have better convergence properties than our algorithm, computing the signs of the true gradient seems no simpler than computing the actual gradient (i.e., it requires a model of the system).

Finally, we want to note the connection between the algorithm we propose here and the field of adaptive control (Sastry and Bodson, 1994; Astrom and Wittenmark, 1994) — in particular the subtopics of Model Reference Adaptive Control (MRAC) and Self-Tuning Regulators — and Iterative Learning Control (ILC) (Moore, 1999). The general philosophy of these approaches is similar to our own: they use an error signal (i.e., between the actual and desired state) to directly adapt the parameters. However, typical formulations of MRAC or ILC use hand-crafted update rules to modify the controller, with update gains that are typically chosen by a system designer. From a high level, though, the signed derivative policy gradient approximation could certainly be viewed as an instance of MRAC or ILC, with a very particular form for the update rule.

3.2 The Signed Derivative Policy Gradient Approximation

In this section we derive a simple approximate policy gradient method, using the approximation we call the *signed derivative.* We want to emphasize that the final form of the algorithm, shown in Algorithm 1, is quite simple, even though the derivation is somewhat involved.

To reiterate our domain settings, we assume the (true) system evolves according to some non-linear, and unknown, model

$$s_{t+1} = f(s_t, u_t). \tag{3.4}$$

For simplicity of the presentation we will assume the model is deterministic, but the methods easily extend to the setting of generating a single sample of the (approximate) gradient in the stochastic setting. We suppose a parametrized policy $\pi(s;\theta)$ for some parameters $\theta \in \mathbb{R}^k$, a cost function $C(s,u)$, and again our goal is to (approximately) compute the policy gradient $\nabla_\theta J(s;\theta)$, which we will use to iteratively improve the policy parameters.

The signed derivative approximation is based on the following intuition. As we will show below, is it possible to explicitly write the analytical form of the policy gradient in a different manner as that presented in the previous chapter, such that the only terms which depend on the dynamics model are terms of the form

$$\frac{\partial s_t}{\partial u_{t'}} \tag{3.5}$$

for $t > t'$. Writing the gradient in this manner is not particularly useful for analytical computation, since the exact analytical form of these derivative terms for $t \gg t'$ it itself quite complex. However, these terms provide the critical motivation for the signed derivative approximation, so it is worth looking at them more closely. These

Jacobian matrices are $n \times m$ matrices, where the ijth element of $\frac{\partial s_t}{\partial u_{t'}}$ indicates how the ith element of s_t changes if we made a small adjustment to $u_{t'}$, but otherwise continued to follow the policy parametrized by θ. For instance, when $t = t' + 1$, this term is simply the partial derivative of the model with respect to the controls

$$\frac{\partial s_{t'+1}}{\partial u_{t'}} = \frac{\partial f(s_{t'}, u_{t'})}{\partial u_{t'}}, \tag{3.6}$$

which corresponds for example to the "B" matrix in the LQR linearization of the dynamics model. The terms are more complex for $t \gg t'$, but the overall intuition of the terms of the same even for longer time horizons: these terms represent how past control inputs affect future states. Thus, we can reasonably assume that over a relatively short time horizon, these terms will be similar to the single-step term, $\frac{\partial f(s_{t'}, u_{t'})}{\partial u_{t'}}$.

The signed derivative approximation is based on the intuition that while it may be difficult to know these derivative terms exactly, it is often fairly easy to estimate their sign. Returning to the above example of driving a car, while it is difficult to know exactly how turning the wheel will affect future states, we can easily intuit that turning the steering wheel more to the left results in future states where the lateral deviation from the trajectory is also more to the left; further, this holds not just for the immediate next time step as implied by $\frac{\partial f(s_{t'}, u_{t'})}{\partial u_{t'}}$, but for some sequence of future time steps as well. The signed derivative approximation, then, simply computes an approximate policy gradient, where we replace *all* the Jacobian terms $\frac{\partial s_t}{\partial u_{t'}}$ for $t > t'$ with a single signed matrix $S \in \mathbb{R}^{m \times n}$ (i.e., a matrix consisting of -1, 0, and 1 entries, which we will also refer to as the signed derivative), that captures the signs of the dominant entries of these derivative terms. We will shortly present several examples of such matrices for a variety of different systems, but we first want to present the formal derivation and algorithm, as well as highlight the restrictions on these signed derivative matrices that are needed for the theory.

3.2.1 The Signed Derivative Approximation and Algorithm

As mentioned in the previous section, we begin the derivation of our algorithm by expanding the policy gradient term in a manner that removes any dependence on the model except via terms of the form (3.5). To do this, first note that for a sequence of states and actions executed according to a policy $(s_0, u_0 = \pi(s_0;\theta), \ldots, s_H, u_H = \pi(s_H;\theta))$ s_t depends on the parameters θ only through the inputs $u_0, \ldots, u_H$. Thus, we can apply the chain rule to obtain

$$\frac{\partial s_t}{\partial \theta} = \sum_{t'=0}^{t-1} \frac{\partial s_t}{\partial u_{t'}} \frac{\partial \pi(s_{t'};\theta)}{\partial \theta}. \tag{3.7}$$

Repeating the derivation from the previous chapter for clarity, we can apply the chain rule to write the policy gradient as

$$\begin{aligned}\frac{\partial J(s;\theta)}{\partial \theta} &= \sum_{t=0}^{H} \frac{\partial C(s_t,u_t)}{\partial \theta} \\ &= \sum_{t=0}^{H} \left(\frac{\partial C(s_t,u_t)}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial s_t}\frac{\partial s_t}{\partial \theta} + \frac{\partial C(s_t,u_t)}{\partial u_t}\frac{\partial \pi(s_t;\theta)}{\partial \theta}\right) \\ &\equiv \sum_{t=0}^{H} \left((q_t + r_t K_t)\frac{\partial s_t}{\partial \theta} + r_t \Phi_t \right)\end{aligned} \tag{3.8}$$

where for ease of notation we use the definitions

$$q_t \equiv \frac{\partial C(s_t,u_t)}{\partial s_t},\ r_t \equiv \frac{\partial C(s_t,u_t)}{\partial u_t},\ K_t \equiv \frac{\partial \pi(s_t;\theta)}{\partial s_t},\ \Phi_t \equiv \frac{\partial \pi(s_t;\theta)}{\partial \theta} \tag{3.9}$$

(note that because the cost function and policy are known analytically, we can compute these terms analytically as well). Substitution our expansion of $\frac{\partial s_t}{\partial \theta}$, (3.7), into (3.8), we obtain

$$\frac{\partial J(s;\theta)}{\partial \theta} = \sum_{t=0}^{H} \left((q_t + r_t K_t) \left(\sum_{t'=0}^{t-1} \frac{\partial s_t}{\partial u_{t'}} \Phi_t \right) + r_t \Phi_t \right). \tag{3.10}$$

```
**Algorithm 1** Policy Gradient with Signed Derivative (PGSD)
Input:
  $S \in \mathbb{R}^{n \times m}$: signed derivative matrix
  $H \in \mathbb{Z}_+$: horizon
  $C : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$: cost function
  $\pi : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^m$: parametrized policy
  $\theta_0 \in \mathbb{R}^k$: initial policy parameters
  $\alpha \in \mathbb{R}_+$: learning rate
Repeat until convergence:
  1. Initialize policy gradient and feature sum: $G \leftarrow 0$, $\Psi_t \leftarrow 0$.
  2. For $t = 0, \ldots H$,
     • Observe state $s_t$, take action $u_t = \pi(s_t; \theta)$.
     • Update policy gradient $G \leftarrow G + (q_t + r_t K_t) S \Psi_t + r_t \Phi_t$.
     • Update feature sum $\Psi_{t+1} \leftarrow \Psi_t + \Phi_t$.
  3. Update policy parameters: $\theta \leftarrow \theta - \alpha G^T$.
```

We now introduce the signed derivative approximation into this expression. As mentioned previously, the signed derivative approximation replaces all derivatives of the form $\frac{\partial s_t}{\partial u_{t'}}$ with a single matrix $S \in \mathbb{R}^{n \times m}$, with 1, 0, and -1 entries, that captures the dominant signs of these derivative terms. Thus, our approximation of the entire policy gradient term becomes

$$\frac{\widetilde{\partial J(s;\theta)}}{\partial \theta} = \sum_{t=0}^{H} \left((q_t + r_t K_t) S \Psi_t + r_t \Phi_t \right) \tag{3.11}$$

where we define

$$\Psi_t \equiv \sum_{t'=0}^{t-1} \Phi_t. \tag{3.12}$$

The final form of the policy gradient with signed derivative (PGSD) algorithm is simply to repeatedly compute approximations to the policy gradient using (3.11), and use these to update the parameters θ. The full method is shown in Algorithm 1.

3.2.2 The Signed Derivative Term

Before presenting theoretical analysis of the policy gradient with signed derivative algorithm, we want to focus in greater detail on the signed derivative term itself. In particular, we will discuss the validity of the assumption that we can reasonably replace *all* the derivative terms with a single matrix representing their dominant signs, and we discuss restrictions on this signed matrix term that manifest themselves in the later analysis.

We begin by looking more thoroughly at the actual form of the model derivative terms $\frac{\partial s_t}{\partial u_{t'}}$. For simplicity, we will use the notation

$$B_{t,t'} \equiv \frac{\partial s_t}{\partial u_{t'}} \tag{3.13}$$

as well as the definition of K_t above and the standard definitions from LQR-based linearization as described in the previous section

$$A_t \equiv \frac{\partial s_{t+1}}{\partial s_t} = \frac{\partial f(s_t, u_t)}{\partial s_t},\ B_t \equiv \frac{\partial s_{t+1}}{\partial u_t} = \frac{\partial f(s_t, u_t)}{\partial u_t}. \tag{3.14}$$

Once again applying the chain rule, we can derive the following recurrence relation for the $B_{t,t'}$ matrices

$$\begin{aligned} B_{t,t'} &= \frac{\partial s_t}{\partial u_{t'}} = \frac{\partial f(s_{t-1}, u_{t-1})}{\partial u_{t'}} \\ &= \frac{\partial f(s_{t-1}, u_{t-1})}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial u_{t'}} + \frac{\partial f(s_{t-1}, u_{t-1})}{\partial u_{t-1}} \frac{\partial u_{t-1}}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial u_{t'}} \\ &= (A_{t-1} + B_{t-1}K_{t-1})B_{t-1,t'} \end{aligned} \tag{3.15}$$

with $B_{t+1,t} = B_t = \frac{\partial f(s_t,u_t)}{\partial u_t}$. The matrix $A_{t-1} + B_{t-1}K_{t-1}$ corresponds to the linearization of the closed loop system at time $t-1$, and a common property of most practical dynamical systems is that this term is close to the identity matrix.[1] Of course, the

[1]To see this more formally , note that if the dynamics are based for example on a discretization of the differential equation $\dot{s} = g(s, u)$, then $s_{t+1} \approx s_t + \Delta t g(s_t, u_t)$, so $A_t \approx I + \Delta t \frac{\partial g(s_t,u_t)}{\partial s_t}$, $B_t \approx \Delta t \frac{\partial g(s_t,u_t)}{\partial u_t}$, and thus if K_t is bounded the $A_t + B_t K_t$ term will only differ from the identity by $O(\Delta t)$.

"interesting" elements of the dynamics are precisely how this term *differs* from the identity, but the fact that this matrix is close to the identity just corresponds to the intuition that the state isn't likely to change widely from one time step to the next, assuming a relatively small time step. The above recurrence relation simply establishes that this also holds for the $B_{t,t'}$ matrices, and so these terms are likely to change relatively little between time steps.

Of course, just because the $B_{t,t'}$ matrices change little over relatively short time periods, this does not imply that their *signs* do not change. In particular, it is very possible that an element of $B_{t,'t}$ with low magnitude (relative to the other entries) may change sign, as even a small deviation from the identity in $A_t + B_t K_t$ could multiply by one of the larger-magnitude entries of $B_{t,t'}$ and change the sign of a smaller-magnitude entry. Thus, as alluded to previously, we actually stipulate that the S matrix only capture the signs of the *dominant* entries of the $B_{t,t'}$ terms, while lower-magnitude entries are simply ignored by setting $S_{ij} = 0$ for such variables. While we are intentionally imprecise in terms of what we mean by the "dominant" entries, for many domains it is apparent that some controls exert significantly less effect on some states than on others. For car driving, for example, we expect the steering wheel angle to have a large effect on the car's orientation and lateral deviation from the trajectory, but less effect on the car's velocity; of course, in practice the steering wheel *does* affect velocity, but the magnitude of this dependence is much less, and so we don't represent it in the signed derivative matrix.

Finally, to highlight an additional point about the signed derivative matrices, we note that in the formal analysis to follow, we will assume that the true derivative terms $B_{t,t'}$ can be expressed in terms of the signed derivative as

$$B_{t,t'} = D_t(S + E_{t,t'}) \tag{3.16}$$

where S is the signed derivative, D_t is a positive diagonal matrix, and $E_{t,t'}$ is an additional error term. Note how this captures the notion of sign-correctness: the entries of S do not need to have the correct magnitude, because they can be re-scaled by the D_t matrix; but they do need to have the correct sign, because the diagonal matrix D_t has only positive entries, and thus the signs of $B_{t,t'}$ (ignoring the error term) will be the same as the signs of S. Finally, this error term $E_{t,'t}$ accounts for the fact, mentioned above, that the smaller-magnitude entries of the derivative terms are ignored in the signed derivative, and so this term accounts for the fact that $B_{t,t'}$ will contain additional, smaller entries that are not captured by the D_tS term; however, we will assume that this matrix has relatively small magnitude.

The above expression also highlights an important limitation to the signed derivative, at least as it applies to the theoretical guarantees that we will make about the method. Because we are *pre-multiplying* S by a diagonal matrix, we can re-scale the rows of S arbitrarily, but we cannot re-scale the columns of S. Thus, if we have more than one non-zero entry per row of S (i.e., two controls that affect the same future state variable), these entries need to have the correct relative magnitudes, and we would thus need to include fractional entries in the signed derivative unless the two controls had the same effect on the future state. Fortunately, a common property of many control domains is that each state is primarily affected by only *one* of the control inputs, and thus we do not need to worry about this issue. Returning one last time to the car driving example, we note that lateral deviation and car orientation are primarily affected by the steering wheel angle, while velocity is primarily controlled by the throttle. In contrast, imagine trying to drive a car where both the steering wheel and throttle controlled some different combinations of the car's orientation and velocity; while such a control system is technically "equivalent" to a standard car, it would take much more work to learn. This suggests, at least anecdotally, that humans also exploit these orthogonal control effects, and so we can expect many control tasks to be designed in this way. We also note that while the theory below does require the assumption (3.16), the algorithm could still be applied to situations with more than one non-zero entry per row in the signed derivative, though the performance or analysis in this case remains an open question.

3.3 Theoretical Analysis

Before presenting a theoretical analysis of situations where the PGSD algorithm is guaranteed to perform well, we begin by demonstrating why the algorithm may *not* perform like typical sign-based gradient optimization techniques. In particular, we consider the exact form of the policy gradient (3.10), and consider what would happen if we replaced the $\frac{\partial s_t}{\partial u_{t'}}$ terms with their *true* sign

$$\widetilde{\frac{\partial J(s;\theta)}{\partial \theta}} = \sum_{t=0}^{H} \left((q_t + r_t K_t) \left(\sum_{t'=0}^{t-1} \operatorname{sign}\left(\frac{\partial s_t}{\partial u_{t'}} \right) \Phi_{t'} \right) + r_t \Phi_t \right). \tag{3.17}$$

It should be apparent that the signs of this term are not guaranteed to be the same as the signs of the true gradient $\frac{\partial J(s;\theta)}{\partial \theta}$, since it is certainly possible that

$$\operatorname{sign}\left(\sum_i x_i \right) \neq \operatorname{sign}\left(\sum_i \operatorname{sign}(x_i) \right). \tag{3.18}$$

Thus, there are two potential sources of error for the signed derivative policy gradient approximation: the error in approximating all the derivative sign terms with a *single* matrix S, and the error introduced by putting the sign terms within a summation in the gradient.

Despite these potential sources of error, there is a informal sense in which we might still expect the method to perform well: when the dominant derivative terms do keep consistent signs over a trajectory (as we argued that they would above), replacing the inner terms with their sign still can capture the general descent directions for the function. Indeed, while the standard practice in sign-based optimization is to take the sign of the final gradient terms, some algorithms do take signs of intermediate terms, and these can perform well in practice even if they lack the guarantees of the standard approaches (Anderson, 1986).

Of course, this informal justification alone is not particularly satisfying, and so in the remainder of this section we develop guarantees showing that under certain assumptions the PGSD algorithm will perform nearly as well as updating the parameters with the true policy gradient. However, the assumptions needed to make such guarantees are indeed rather strong, so we do want to allude to the preceding informal argument as an additional intuitive justification for the approach.

3.3.1 Overview of the Theoretical Results

In this section we present the basic intuition of our theoretical results. Because the results deal with the convergence properties of the algorithm, and since convergence properties of any policy gradient method depend on the structure (e.g. convexity properties) of the cost function itself (a non-convex cost function could imply an unbounded value function, for instance), we assume for the sake of these results that the cost function is quadratic, i.e.,

$$C(s,u) = s^T Q s + u^T R u. \tag{3.19}$$

The theory extends without modification to trajectory-dependent cost functions

$$C_t(s,u) = (s - s_t^\star)^T Q (s - s_t^\star) + (u - u_t^\star)^T R (u - u_t^\star), \tag{3.20}$$

and for general convex cost functions the analysis below extends to the second-order quadratic expansion of these functions. Note that given these definitions, the q_t and r_t terms take the form

$$q_t = s_t^T Q,\ r_t = u_t^T R. \tag{3.21}$$

The basic intuition of the theoretical proof is as follows. Suppose that the true matrix derivative terms $B_{t,t'}$ obey the condition described previously, that $B_{t,t'} = D_t(S + E_{t,t'})$ for all t', with $\|E_{t,t'}\| \leq \epsilon$. Then the gradient given by the signed derivative approximation is

$$\begin{aligned}\widetilde{\frac{\partial J(s;\theta)}{\partial\theta}} &= \sum_{t=0}^{H}((q_t + r_t K_t)S\Psi_t + r_t\Phi_t) \\ &= \sum_{t=0}^{H}\left((s_t^T Q + u_t^T R K_t)\left(\sum_{t'=0}^{t-1}(D_t^{-1}B_{t,t'} - E_{t,t'})\Phi_t\right) + u_t^T R\Phi_t\right) \\ &= \sum_{t=0}^{H}\left((s_t^T Q D_t^{-1} + u_t^T R K_t D_t^{-1})\left(\sum_{t'=0}^{t-1} B_{t,t'}\Psi_t\right) + u_t^T R\Phi_t\right) + O(\epsilon).\end{aligned} \tag{3.22}$$

This term looks almost like true policy gradient (with an added $O(\epsilon)$ term), except for the presence of the D_t^{-1} terms, which can render the approximation not even a descent direction of the true value function. However, if for all t we can find a $\tilde{Q}_t \succeq 0$ and $\tilde{R}_t \succeq 0$ such that

$$s_t^T \tilde{Q}_t = s_t^T Q D_t^{-1}, \quad \text{and} \quad u_t^T \tilde{R}_t K_t = u_t^T R K_t D_t^{-1} \tag{3.23}$$

then the approximate gradient given by the signed derivative at this point is equal (up to the $O(\epsilon)$ term) to the *true* policy gradient for a *different* cost function, given by the $\tilde{Q}_t$ and $\tilde{R}_t$ matrices. Classical results (Khatri and Mitra, 1976) show that positive semidefinite $\tilde{Q}_t$ and $\tilde{R}_t$ matrices exist that satisfy these equalities if and only if[2]

$$s_t^T Q D_t^{-1} s_t \geq 0, \quad \text{and} \quad u_t^T R K_t D_t^{-1} K^{\dagger} u \geq 0. \tag{3.24}$$

While these conditions often hold in practice, it is difficult to guarantee them a priori, and it is also difficult to bound the differences between the solutions $\tilde{Q}_t$ and $\tilde{R}_t$ and Q and R. Thus, while this overall procedure provides additional insight into why the algorithm may perform well, for the precise results below we will make substantially more restrictive assumptions, and assume that Q is diagonal and that R is zero (i.e., the cost function depends only on the state, though we can introduce constraints on the controls by restricting the allowable controls or regularizing the policy parameters θ directly). While these may seem to be large restrictions, in practice we almost

[2]Here the † symbol denotes the Moore-Penrose pseudoinverse.

always choose Q to be diagonal anyway (populating a full cost matrix Q can be quite unintuitive), and restrictions on controls by directly penalizing parameters or control magnitudes are often actually more intuitive than quadratic cost penalties. When Q is diagonal and $R = 0$, the above approximate policy gradient takes the much simpler form

$$\widetilde{\frac{\partial J(s;\theta)}{\partial\theta}} = \sum_{t=0}^{H} s_t^T \tilde{Q}_t \left(\sum_{t'=0}^{t-1} B_{t,t'} \Psi_t \right) + O(\epsilon) \tag{3.25}$$

where $\tilde{Q}_t \equiv QD_t^{-1}$ is naturally also diagonal and positive definite. Thus, the approximate gradient given by the signed derivative is indeed approximating the gradient of a *different* cost function. Therefore, we expect the procedure to converge to a near local optimum of the value function with this modified cost function, and the only remaining issue to discuss is how this relates to the policy's performance on the original (true) cost function.

To understand why optimizing a modified cost function with cost matrix $\tilde{Q}_t = QD_t^{-1}$ can still lead to a policy that performs well on the original cost function, we can consider the extreme case where the policy class is chosen such that we can actually find a policy that achieves zero total cost. In this case, it doesn't matter how we scale the cost function, since minimizing any quadratic function (globally) will also achieve zero cost. The reason why the different cost matrices come into play is that often times we *cannot* achieve zero cost (i.e., we are restricted by virtue of the system dynamics to a certain subset of allowable state action pairs, and the global optimum is not within this set) and so the contours of the cost function determine which suboptimal points look best. This same intuition holds when the policy class can obtain a controller with merely near-zero cost: in this case, optimizing the modified cost function also results in a policy with near-zero cost, with an additional scaling factor that may be as large as $\kappa(\mathrm{diag}(D_1, \ldots, D_H))$, the condition number of a diagonal matrix formed from all the diagonal scaling matrix[3]. In other words, (given all the assumptions above) if we choose a policy class such that true policy gradient could achieve low cost, and if the control inputs are reasonably well scaled, then we expect the PGSD

[3]The conditional number of matrix A is defined as the ratio between the smallest and lowest singular value$\sigma_{\max}(A)/\sigma_{\min}(A)$.

algorithm to perform well.

3.3.2 Formal Results

The following theorem formalizes the intuition we described in the previous section. For the purposes of the proof, we assume that the dynamics are deterministic and that the initial state s_0 is fixed, and does not depend on the policy (this would be the case, for instance, if the control task was episodic, and the system was reset to an initial state before each trial). Although the assumption of deterministic dynamics may seem quite restrictive, this is mainly done so that we can work with the simpler notation of deterministic gradient descent methods, rather than stochastic gradient descent methods, which would be required for stochastic settings. The results can be extended to the stochastic setting, as originally published in Kolter and Ng (2009a), but the extension requires a great deal of additional machinery, and adds little to the actual intuition of the approach.

The theorem statement to follow requires a number of technical assumptions, which we list here.

Assumption 1. *As described above, the true gradients of the system dynamics are related to the signed derivative by*

$$B_{t,t'} = D_t(S + E_{t,t'}), \ \|E_{t,t'}\| \leq \epsilon. \tag{3.26}$$

Assumption 2. *The dynamics function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is deterministic, and the initial state s_0 is fixed and independent of the policy π.*

Assumption 3. *The modified value function*

$$\tilde{J}(s;\theta) = \mathbf{E}\left[\sum_{t=0}^{H} s_t^T Q_t D_t^{-1} s_t \big| s_0 = s, \pi(\cdot;\theta)\right] \tag{3.27}$$

has a Lipschitz continuous gradient[4] *with Lipschitz constant K (this property will*

[4]A function f has Lipschitz continuous gradient, with Lipschitz constant K, if $\|\nabla f(x_1) - \nabla f(x_2)\| \leq K\|x_1 - x_2\|$ for all x in the domain of f.

hold, for example, if the dynamics and policy also have Lipschitz continuous gradients by the fact that compositions of Lipschitz continuous functions are also Lipschitz continuous). Furthermore, the gradient step size α satisfies

$$0 < \alpha < \frac{2}{9K}. \tag{3.28}$$

Given these assumptions, we now present the main theorem.

Theorem 1. *Under the assumptions above, the PGSD algorithm will converge to some region that is close to a local optimum of the modified value function* (3.27)*, i.e.,*

$$\|\nabla_\theta \tilde{J}(s,\theta)\| \leq O(\epsilon). \tag{3.29}$$

Furthermore, if (global) optimization of the true value function would result in η-optimal policy parameters — i.e., $J(\theta^\star) \leq \eta$ — then (global) optimization of the PGSD objective will result in a solution $\tilde{\theta}$ that is an order η-optimal solution

$$J(\tilde{\theta}) \leq \kappa(D)\eta. \tag{3.30}$$

where $\kappa(D)$ denotes the condition number of the diagonal matrix

$$D = \mathrm{diag}(D_1, D_2, \ldots, D_T). \tag{3.31}$$

Before turning to the proof of the theorem, we want to address one element in its statement that may seem odd. In particular, the theorem first states that the PGSD algorithm converges to a near *local* optimum of a modified cost function, then claims that under suitable conditions a *global* optimum of this modified cost function will be close to the global optimum of the true value function. Since both the modified and true value functions are non-convex, we don't expect to be able to find global optima of either of these functions in general. However, the theorem merely shows the expected behavior that optimizing one function also tends to minimize the other: the same fact would hold for a local optimum of the value functions if we additionally assume that both gradient procedures (gradient descent on the true value function versus the modified value function) were restricted to the same convex region, but since non-convexity could cause the two gradient descent procedure to converge to different convex regions the above is the strongest statement we can make without additional assumptions.

Our proof uses two lemmas (proved in the next section) regarding the convergence and solutions of optimization procedures. Recall from the discussion above that under suitable assumptions, the signed derivative approximate gradient is equal to a gradient of a different function, plus a bounded error term. The following lemma shows that optimizing a function using gradient descent plus an error term converges to an approximate (local) optimum. This is a fairly intuitive result, and for simplicity we focus here on minimizing a deterministic function with Lipschitz continuous gradient. The result is a straightforward extension of well-known results for the minimization of function with Lipschitz continuous derivatives (Armijo, 1966). The result generalizes to more complex situations, such as stochastic gradient descent methods or different rules for stepsize selection, but these convergence results require much more machinery and assumptions, and don't add significantly to the actual intuition of the PGSD algorithm.

Lemma 2. *Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is bounded below and has Lipschitz continuous derivative with Lipschitz constant K, and we employ the approximate gradient descent procedure*

$$x_{t+1} \leftarrow x_t - \alpha g_t, \quad \|g_t - \nabla f(x_t)\| \leq \epsilon. \tag{3.32}$$

Then for constant step size $0 < \alpha < \frac{2}{9L}$, as $t \to \infty$, x_t converges to a region where

$$\|\nabla f(x_t)\| \leq 2\epsilon. \tag{3.33}$$

The second lemma is a general statement about the optimization of quadratic functions. Namely, it states that if we globally optimize some positive semidefinite quadratic function $f_1(x) = x^T Q_1 x$ over an arbitrary (possibly non-convex) set $\mathcal{C}$, then the solution will also be close to the globally optimal solution of another positive semidefinite quadratic function $f_2 = x^T Q_2 x$ over the same set. The difference in the solutions depends on how close to the solution is to the zero point, and a ratio of certain eigenvalue terms. This captures the following intuition. Since the zero point $x = 0$ is the globally optimal unconstrained minimum for both functions, if $0 \in \mathcal{C}$, then the solutions will coincide. If $0 \notin \mathcal{C}$, however, then the global optimum of f_1 over $\mathcal{C}$ can differ from the globally optimal solution to f_2, but this difference is bounded by 1) how close the optimal point is to the zero point, and 2) a particular ratio of the eigenvalues of the two quadratic forms.

Lemma 3. *Consider two positive definite quadratic functions $f_1, f_2 : \mathbb{R}^n \to \mathbb{R}$, defined by*

$$f_1(x) = x^T P_1 x,\ f_2(x) = x^T P_2 x,\ P_1, P_2 \succeq 0. \tag{3.34}$$

Let $\mathcal{C} \subseteq \mathbb{R}^n$ be some arbitrary (and possibly non-convex) subset of $\mathbb{R}^n$, and consider the solutions of the global optimization problems

$$x_1^\star = \arg\min_{x\in\mathcal{C}} f_1(x),\ x_2^\star = \arg\min_{x\in\mathcal{C}} f_2(x). \tag{3.35}$$

Then

$$f_1(x_2^\star) \le \frac{\lambda_{\max}(P_1^{-1}P_2)}{\lambda_{\min}(P_1^{-1}P_2)} f_1(x_1^\star). \tag{3.36}$$

Given these two lemmas, the proof of Theorem 1 is straightforward.

Proof. (of Theorem 1). First note that from (3.22),

$$\begin{aligned}
\frac{\widetilde{\partial J(s;\theta)}}{\partial\theta} &= \sum_{t=0}^{H} s_t^T Q \left(\sum_{t'=0}^{t-1}(D_t^{-1}B_{t,t'} - E_{t,t'})\right)\Phi_t \\
&= \sum_{t=0}^{H} s_t^T Q D_t^{-1}\left(\sum_{t'=0}^{t-1} B_{t,t'}\right)\Phi_t - \sum_{t=0}^{H} s_t^T Q\left(\sum_{t'=0}^{t-1} E_{t,t'}\right)\Phi_t \\
&\le \frac{\partial\tilde{J}(s,\theta)}{\partial\theta} + \sum_{t=0}^{H}\|s_t\|\|Q\|\left(\sum_{t'=0}^{t-1}\|E_{t,t'}\|\right)\|\Phi_t\| \\
&\le \frac{\partial\tilde{J}(s,\theta)}{\partial\theta} + K_1\epsilon
\end{aligned} \tag{3.37}$$

where we bound the second term using the sub-multiplicative property of matrix norms and by repeated application of the triangle inequality. Thus, given the assumptions above, PGSD satisfies the conditions of Lemma 2, and we can guarantee that it converges near to a local optimum, as desired.

The second claim of the theorem follows directly from Lemma 3, by noting that the true and modified cost function are quadratic in the states with respective cost matrices

$$Q = \mathrm{diag}(Q, \ldots, Q) \; (T \text{ times}) \;, \quad \tilde{Q} = \mathrm{diag}(\tilde{Q}_1, \ldots \tilde{Q}_T). \tag{3.38}$$

Thus, applying Lemma 3 we have that optimizing the modified cost function leads to some factor times the optimal solution for the true cost function, where the factor is given by

$$\frac{\lambda_{\max}(Q^{-1}\tilde{Q})}{\lambda_{\min}(Q^{-1}\tilde{Q})} = \frac{\lambda_{\max}(\tilde{Q}Q^{-1})}{\lambda_{\min}(\tilde{Q}Q^{-1})} = \frac{\lambda_{\max}(D^{-1})}{\lambda_{\min}(D^{-1})} = \kappa(D) \tag{3.39}$$

as desired, where we can exchange the order of multiplication because both Q and $\tilde{Q}$ are diagonal. □

3.3.3 Proofs of Technical Lemmas

Proof. (of Lemma 2) By the mean value theorem,

$$f(x_{t+1}) - f(x_t) = (x_{t+1} - x_t)^T \nabla f(\tilde{x}_t) \tag{3.40}$$

for some $\tilde{x}_t$ on the line segment connecting x_t and x_{t+1}. Since under our assumptions, $x_{t+1} - x_t = -\alpha(\nabla f(x_t) + e)$ with $\|e\| \leq \epsilon$,

$$\begin{aligned}
f(x_{t+1}) - f(x_t) &= -\alpha(\nabla f(x_t) + e)^T \nabla f(\tilde{x}_t) \\
&= -\alpha(\nabla f(x_t) + e)^T (\nabla f(x_t) - \nabla f(x_t) + \nabla f(\tilde{x}_t)) \\
&= -\alpha(\nabla f(x_t) + e)^T \nabla f(x_t) + -\alpha(\nabla f(x_t) + e)^T (\nabla f(\tilde{x}_t) - \nabla f(x_t)) \\
&\leq -\alpha\|\nabla f(x_t)\|^2 + \alpha\|e\|\|\nabla f(x_t)\| + \alpha\|\nabla f(x_t) + e\|\|\nabla f(\tilde{x}_t) - \nabla f(x_t)\| \\
&\leq -\alpha\|\nabla f(x_t)\|^2 + \alpha\|e\|\|\nabla f(x_t)\| + \alpha\|\nabla f(x_t) + e\|K\|\tilde{x}_t - x_t\| \\
&\leq -\alpha\|\nabla f(x_t)\|^2 + \alpha\|e\|\|\nabla f(x_t)\| + \alpha\|\nabla f(x_t) + e\|K\|x_{t+1} - x_t\| \\
&\leq -\alpha\|\nabla f(x_t)\|^2 + \alpha\|e\|\|\nabla f(x_t)\| + \alpha\|\nabla f(x_t) + e\|\alpha K\|\nabla f(x_t) + e\| \\
&= -\alpha\|\nabla f(x_t)\|^2 + \alpha\|e\|\|\nabla f(x_t)\| + \alpha^2 K\|\nabla f(x_t) + e\|^2 \\
&= -(\alpha - \alpha^2 K)\|\nabla f(x_t)\|^2 + (\alpha + 2\alpha^2 K)\|\nabla f(x_t)\|\|e\| + \alpha^2 K\|e\|^2 \\
&\leq -(\alpha - \alpha^2 K)\|\nabla f(x_t)\|^2 + (\alpha + 2\alpha^2 K)\|\nabla f(x_t)\|\epsilon + \alpha^2 K\epsilon^2.
\end{aligned} \tag{3.41}$$

Now suppose $\|\nabla f(x_t)\| \geq 2\epsilon$. Then

$$\begin{aligned}
f(x_{t+1}) - f(x_t) &\leq -(\alpha - \alpha^2 K)\|\nabla f(x_t)\|^2 + \frac{\alpha + 2\alpha^2 K}{2}\|\nabla f(x_t)\|^2 + \frac{\alpha^2 K}{4}\|\nabla f(x_t)\|^2 \\
&= \left(\frac{-1}{2}\alpha + \frac{9K}{4}\alpha^2\right)\|\nabla f(x_t)\|^2.
\end{aligned} \tag{3.42}$$

Since for $\alpha > 0$,

$$\frac{-1}{2}\alpha + \frac{9K}{4}\alpha^2 < 0 \iff \alpha < \frac{2}{9K} \tag{3.43}$$

then by our assumption that $0 < \alpha < \frac{2}{9K}$, at each iteration of gradient descent we have

$$f(x_{t+1}) \leq f(x_t) - \delta\|\nabla f(x_t)\|^2 \tag{3.44}$$

for some $\delta > 0$ independent of x_t. Thus, for $\|\nabla f(x_t)\| \geq 2\epsilon$, the objective function must decrease by at least $4\delta\epsilon^2$ at each iteration. Since the function f is bounded below, this can only happen a finite number of times, implying that as $t \to \infty$, x_t

must converge to a region where $\|\nabla f(x_t)\| \leq 2\epsilon$. □

Proof. (of Lemma 3) Consider the set $\mathcal{C}_0 = \{x : x \in \mathbb{C}, f_2(x) \leq f_2(x_1^\star)\}$. Clearly $x_2^\star$ is a member of this set, so we therefore have

$$f_1(x_2^\star) \leq \max_{x \in \mathcal{C}_0} f_1(x) \leq \max_{f_2(x) \leq f_2(x_1^\star)} f_1(x) \equiv \max_{x^T P_2 x \leq f_2(x_1^\star)} x^T P_1 x. \tag{3.45}$$

By a standard variational formulation of the generalized eigenvalue problem **?**, this term can be bounded as

$$\max_{x^T P_2 x \leq f_2(x_1^\star)} x^T P_1 x \leq \lambda_{\max}(P_2^{-1} P_1) f_2(x_1^\star). \tag{3.46}$$

Dividing the previous equations by $f_1(x_1^\star)$ and again applying the variational formulation of the generalized eigenvalue problem

$$\begin{aligned} \frac{f_1(x_2^\star)}{f_1(x_1^\star)} &\leq \lambda_{\max}(P_2^{-1} P_1) \frac{f_2(x_1^\star)}{f_1(x_1^\star)} \\ &\leq \lambda_{\max}(P_2^{-1} P_1) \max_x \frac{x^T P_2 x}{x^T P_1 x} \\ &\leq \lambda_{\max}(P_2^{-1} P_1) \lambda_{\max}(P_1^{-1} P_2). \end{aligned} \tag{3.47}$$

Since $\lambda_{\max}(A^{-1}) = 1/\lambda_{\min}(A)$, the lemma follows.

□

3.4 Experimental Results

3.4.1 Simulated Two-Link Arm

While we will present experiments on real systems shortly, we begin our experimental analysis by presenting an evaluation of our proposed method on a simulated two-link arm, in order to rigorously compare to previous policy gradient approaches, and to provide a readily available implementation of our approach. Code for the all the results in this section is available at `http://cs.stanford.edu/~kolter/rss09sd`.

We emphasize that the purpose of this section is to specifically compare PGSD with other policy gradient approaches: the control task itself is fairly straightforward, and many other approaches such as adaptive control or iterative learning control could also be applied, though this is beyond the scope of this work.

The two-link pendulum is a well-known control task in robotics and control. The system, shown in Figure 3.1 consists of two planar links; the state consists of the joint angles and velocities of both joints and the control specifies a torque at each of the joints

$$s_t = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}, \quad u_t = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} \tag{3.48}$$

The equations of motion can be easily derived from Lagrangian dynamics, and we introduce stochasticity to the system by adding Gaussian noise to the torques before integrating the equations of motion. The task we consider here, also shown in the figure, is to move the end effector along some desired trajectory. When the model of the system is known, it is fairly easy to apply classical control methodologies such as inverse dynamics or LQR to find an optimal controller, but of course we don't provide this model to PGSD or other comparable algorithms. We feel that this is a particularly demonstrative example for the Signed Derivative algorithm, since it is well-known that there *are* cross terms that cause all joints to be affected by all the control inputs — for instance, a common (more challenging) task is to swing the pendulum upright and balance by applying torques only to the elbow — yet we claim that the Signed Derivative approximation is still reasonable, since joints are *primarily* affected by their own control. In particular, since we reason that each torque primarily affects both the joint angle and joint velocity of that state, the signed

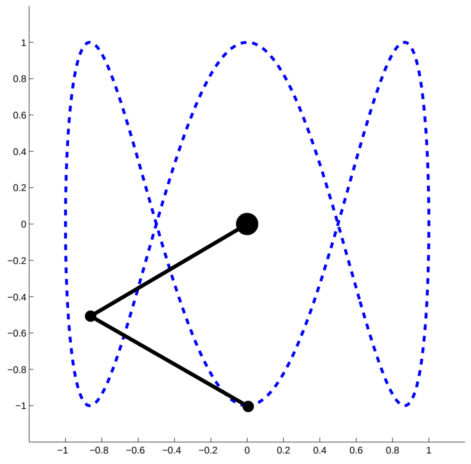


Figure 3.1: Two-link pendulum trajectory following task.

derivative matrix for this task is given by

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.49}$$

The cost function for this domain penalizes deviations from the desired joint angles

$$C_t(s,u) = (q_1 - q^\star_{1,t})^2 + (q_2 - q^\star_{2,t})^2 \tag{3.50}$$

(we transform the trajectory to joint space via inverse kinematics), and we use a time horizon of $H = 5$. Note that this doesn't mean that the controller only needs to follow the trajectory for 5 steps, but rather that at each time the controller should ideally act optimally with respect to a receding horizon of $H = 5$; since the cost function itself "guides" the arm along the trajectory, such a horizon is suitable. We use a linear control policy $u_t = \theta^T \phi(s_t, t)$ where $\phi(s_t, t) \in \mathbb{R}^7$ contains:

1. Deviations from desired joint angles, $q_{1,2} - q^\star_{1,2}$

2. Deviations from desired joint velocities, $\dot{q}_{1,2} - \dot{q}^\star_{1,2}$

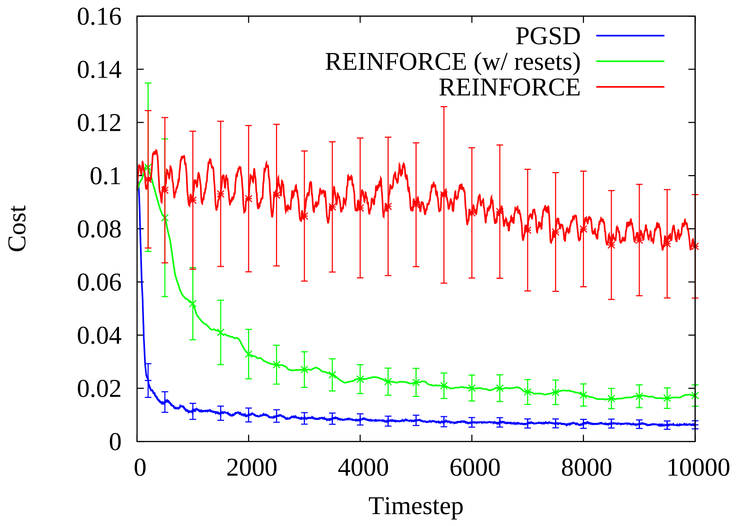


Figure 3.2: Average cost versus time for different policy gradient methods. Costs are averaged over 20 runs, and shown with 95% confidence intervals.

3. Desired joint accelerations $\ddot{q}^{\star}_{1,2}$
4. A term equal to $\sin(2\pi t/t_{\text{total}})$ where t_{total} is total time for the complete trajectory (this was added to account for a visible periodic pattern in the controls).

This leads to a total of 14 parameters for the policy (7 for each different control input). For algorithms that require a stochastic policy, we added Gaussian noise to the parameters: $u_t = (\theta + \epsilon_t)^T \phi(s_t, t)$, $(\epsilon_t)_{ij} \sim \mathcal{N}(0, \sigma)$.

Figure 3.2 compares the performance versus time of PGSD, and a well-known policy gradient RL algorithm, the REINFORCE algorithm.[5] All free parameters of the learning algorithms (gradient step sizes, policy noise, number of episodes) were hand-optimized to give that fastest convergence that didn't cause any divergence issues. As the figure shows, PGSD drastically outperform the other methods, converging much faster to a low-cost policy. This improvement is especially notable given that the REINFORCE algorithm is actually given an advantage: since the task we're considering

[5]We intentionally scaled the parameters of this control task to be the same order of magnitude, so more advanced techniques such as natural gradients(Kakade, 2001; Peters and Schaal, 2006) didn't improve performance significantly. In preliminary experiments we also evaluated a variety of finite difference and weight perturbation methods, but didn't notice a substantial improvement over REINFORCE for this task.

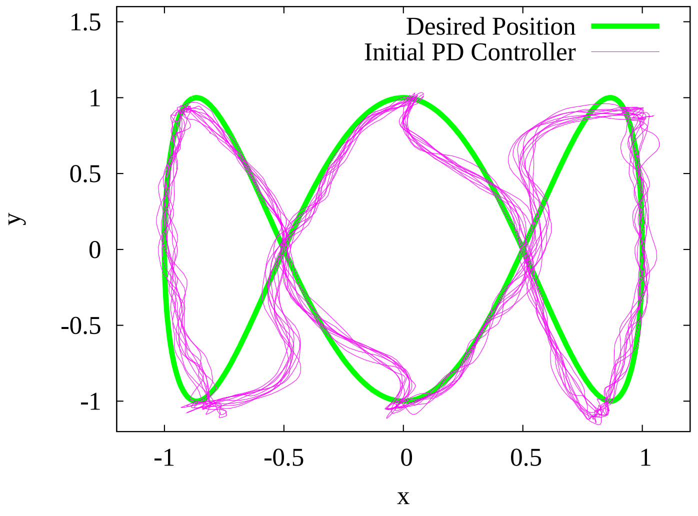


Figure 3.3: Trajectories from initial controller.

is not episodic (at least not at the time-scale of the horizon), episodic algorithms don't immediately apply, and so we instead allow the algorithm the ability to reset to previous states observed along the trajectory. The REINFORCE without resets in the figure does not have such an advantage, but also performs much worse. Figures 3.3 and 3.4 show trajectory achieved by the initial controller (used to initialize all the learning algorithms), and the controller learned by the PGSD algorithm after 2000 time steps (4 times through the trajectory).

We also compare, in Figure 3.5, the performance of the PGSD algorithm, policy gradient using the true gradient from the model, and an optimal LQR controller. Not surprisingly, the LQR controller performs best: this controller is built by linearizing around the (known) dynamics at each operating point, then computing a series of non-stationary policies for each point (in total, the LQR controller has 9000 parameters). However, using only 14 parameters, the true policy gradient and PGSD algorithm are able to obtain a controller that performs relatively close to this full LQR controller. Furthermore, the most important result is that the learning curve for PGSD is virtually *indistinguishable* from the true policy gradient learning curve; despite the rather crude approximation made by the signed derivative, this resulting algorithm performs *just as well* on this task, and requires no model of the system

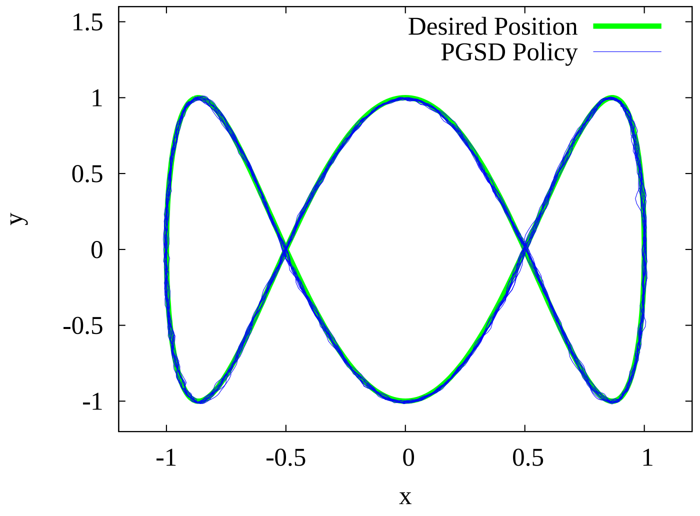


Figure 3.4: Trajectories from controller learned using PGSD.

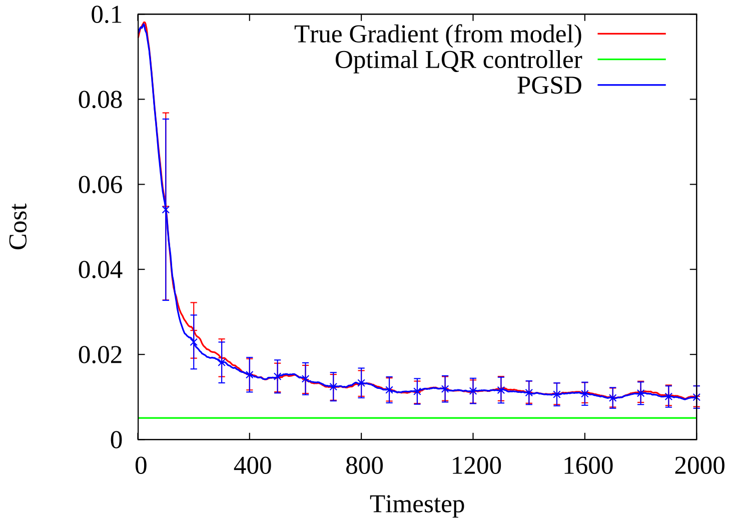


Figure 3.5: Average cost versus time for PGSD versus model-based methods. Costs are averaged over 20 runs, and shown with 95% confidence intervals.

(and therefore also less computation time, since there is no need for time-consuming finite difference computations).

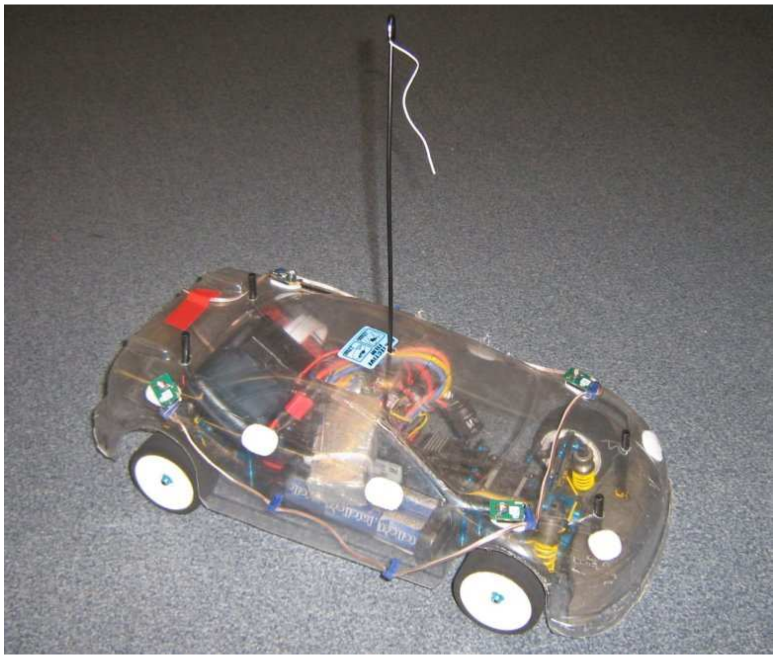


Figure 3.6: RC car used for the driving experiments.

3.4.2 Autonomous RC Driving

In this section we apply the PGSD algorithm to the task of learning to drive an autonomous RC car along a desired trajectory. Figure 3.6 shows the car, a Tamiya TRF415, which is about 40cm long and 20cm wide. A pattern of LED lights is attached to the car, and tracked by an external PhaseSpace motion capture system for pose estimation. All processing is done on a workstation PC, with controls transmitted to the car at 50hz.

The simplest representation of the car's state is as six dimensional vector representing the 2D position x, y, the orientation θ, and the time derivatives $\dot{x}, \dot{y}, \dot{\theta}$. However, a more natural representation for the signed derivative approach is to represent the car's state relative to some desired trajectory — here the trajectory is specified as a continuous spline that gives the desired state as a function of time. In this alternate representation, the state consists of the longitudinal, lateral, and angular deviation (and their derivatives) from the desired trajectory. The control is two dimensional, consisting of a commanded throttle and steering angle. Thus, the signed derivative

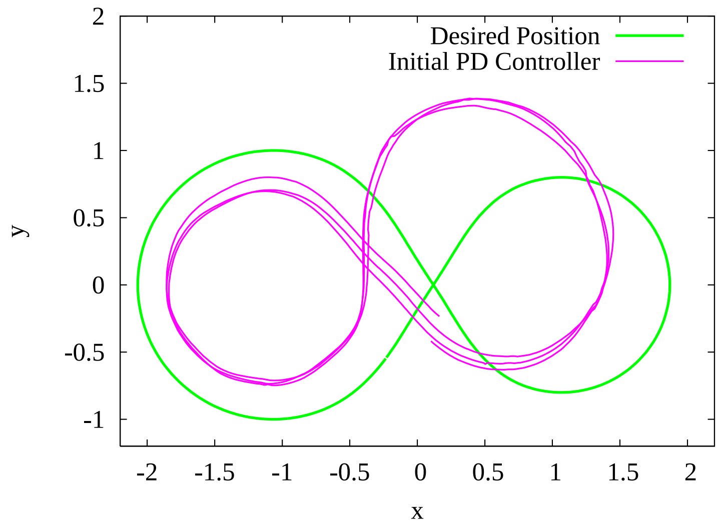


Figure 3.7: Desired trajectory for the autonomous RC driving experiments, with trajectory for initial controller.

is a 6×2 matrix, given by

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \tag{3.51}$$

We use the same form of linear controller as in the previous sections, but where $\phi(s,t)$ now contains 1) the full state (represented as the deviation terms), 3) the desired velocities, relative to the car frame, 3) the deviations for a target state 0.5 seconds in the future and 4) a constant term. Some of the θ parameters are forced to be zero (so that, for instance, the throttle doesn't depend on the lateral deviation), for a total of 16 parameters in the policy. The cost function penalizes the longitudinal, lateral, and angular deviation, any control outside a specified valid range, and control that changes more that some amount between two time steps (to minimize oscillations). We used a time horizon of $H = 25$.

Figures 3.7 and 3.8 show the control task we consider: driving the car in an

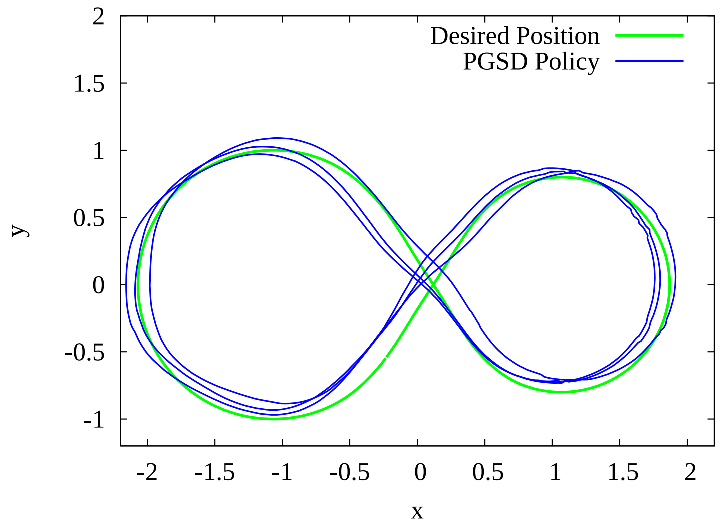


Figure 3.8: Desired trajectory for the autonomous RC driving experiments, with typical trajectory learned using PGSD after approximately 20 seconds of learning.

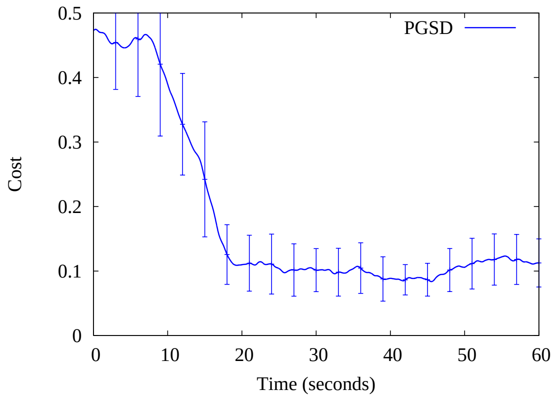


Figure 3.9: Average cost versus time for the PGSD algorithm on the RC car task. Costs are averaged over 10 runs, and shown with 95% confidence intervals.

irregular figure-eight pattern at varying speeds (2.0 m/s along the larger loop, 1.5 m/s along the smaller loop). The figure also shows the trajectory followed by an initial controller: while the initial controller follows the overall pattern of the trajectory, it clearly does not perform very well. Figure 3.9 shows the learning curve of the PGSD algorithm. As the figure shows, PGSD is able to very quickly — within an average of 20 seconds, about 3 times around the trajectory — obtain a policy that performs far better than the initial controller. We show a typical trajectory from one of these

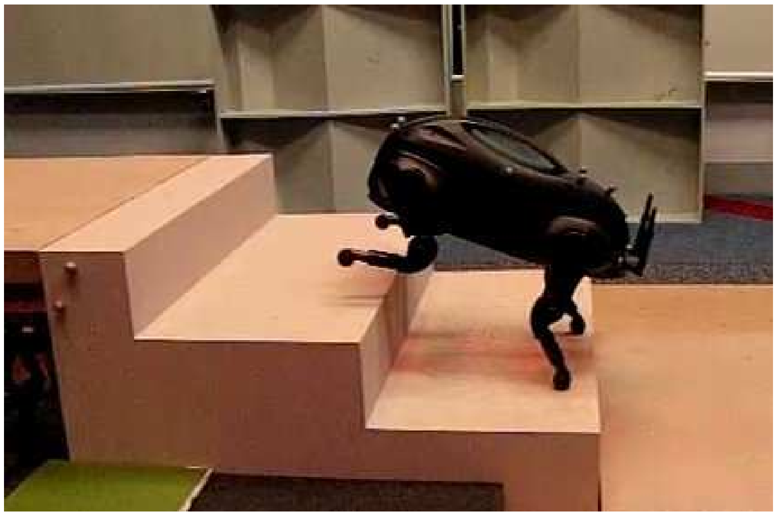


Figure 3.10: The desired task for the LittleDog: climb over three large steps.

learned controllers in Figure 3.8. The learned policies do not perform flawlessly — the car still sometimes veers off the desired path — but we feel this is largely due to the limited policy class itself; to perform better, one might need more complex, time-varying policies, to capture the fact that the car needs to behave differently at different points along the path. Nonetheless, PGSD converges to a very reasonable policy — in fact, better than any we were able to hand tune in the same policy class — in just 20 seconds of learning.

3.4.3 LittleDog Jumping

In this section we present results on applying PGSD to the task of "jumping" the front legs of a quadruped robot up a large step, as shown in Figure 3.10 The LittleDog robot, which we use for this task, will be described in much more detail in Chapter 6, but here we briefly describe the jumping task and the challenges involved.

The goal of the front leg jump maneuver is to quickly and simultaneously lift both front legs onto a step or over a gap. "Jumping" is perhaps a misnomer for the action, since the robot does not actually have the power to force its front legs off the ground from a typical standing position. Rather, the strategy we employ is to shift the robot's center of gravity (COG) backwards until the front legs become unloaded

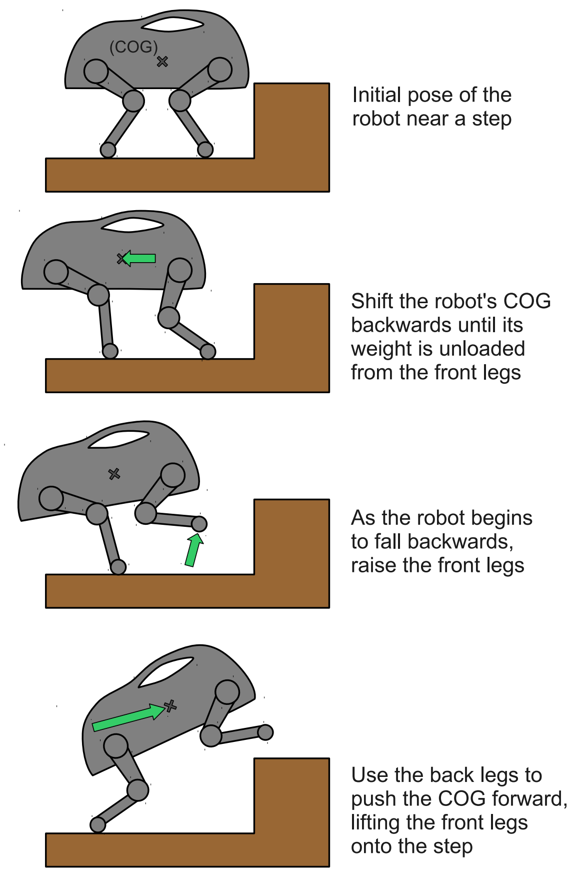

(COG)
Initial pose of the robot near a step
Shift the robot's COG backwards until its weight is unloaded from the front legs
As the robot begins to fall backwards, raise the front legs
Use the back legs to push the COG forward, lifting the front legs onto the step

Figure 3.11: Overview of the front jump maneuver.

and the robot is falling backwards. At this point, we quickly shift the COG forward again, and raise the front legs. If the maneuver executes as planned, then this will lift the front legs off the ground, and move them simultaneously to a new location by the time the robot falls forward, as shown in Figure 3.11. A mosaic of the real robot performing a jump is shown in Figure 3.12.

Due to the nature of the LittleDog, this is a delicate maneuver: shifting the COG not far enough or too far can either fail to unload the robot's front legs or cause the robot to flip backward, respectively. Because this is an extremely fast transition, we have been unable to develop a typical stabilizing controller: by the time we receive the necessary sensory data from the robot, it would have already entered into one of these failure modes, and lacks the control authority to recover. Furthermore, the

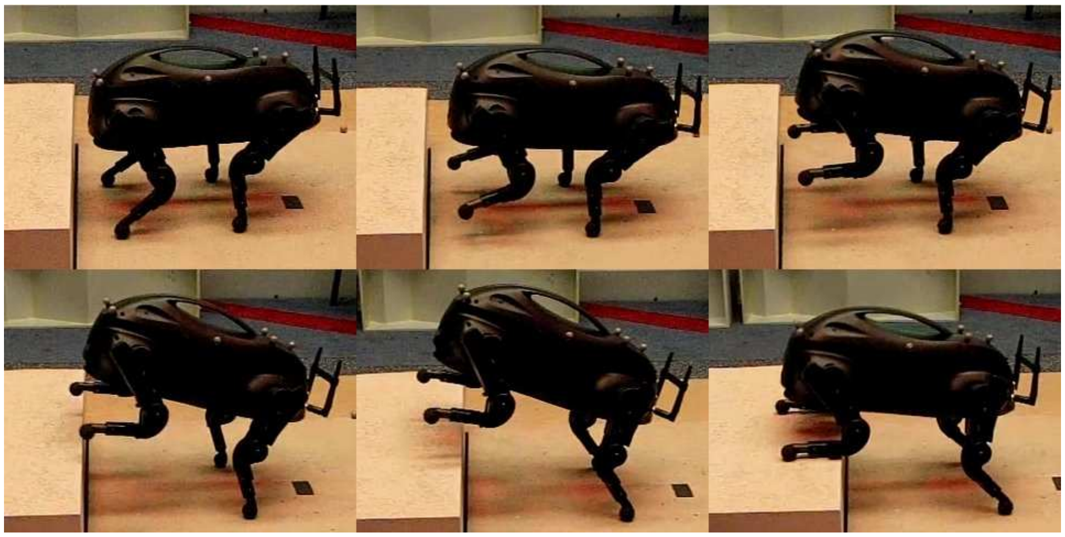


Figure 3.12: A properly executed jump.

correct amount to shift the COG depends on very small changes to the initial state of the robot. While ensuring a successful jump could be done via precise modeling of the robot, or a "calibration" procedure that would put the robot into a known state (such strategies were employed in (Byl et al., 2008) to achieve a similar jump), our goal was to develop a single policy that could execute such a jump immediately from a variety of different initial states. To accomplish this task, we parametrize the jumping maneuver as a function of features of the robot's initial state, and apply the PGSD algorithm to learn a policy for shifting the robot's COG based upon these features.

Although the full state space for the LittleDog is 36 dimensional, we don't need to take into account the complete state. Rather, we have found through experimentation that the correct amount to shift the COG backwards depends mainly on five features, so the policy we employ determines the amount to shift back as linear function of: 1) the current shift of the COG, 2) the forward velocity of the dog and 3) the initial pitch of the dog, 4) the rotational velocity around the pitch axis, and 5) a constant term; intuitively, these feature provide a natural description of the robot's longitudinal state, and thus are the primary elements relevant for jumping forward. Since there is only one control input (how far back to shift) and one state that is relevant to the success of the jump (the resulting pitch of the robot), the signed derivative in this case is just the singleton matrix $S = 1$, indicating that shifting the robot further back makes it pitch more.

Recall that the success or failure of the jump can be quantified by how much the robot pitches. Letting $\beta^\star$ denote the "optimal" pitch angle of the jump letting β denote the amount that the robot did pitch. We use the cost function that depends on the absolute error between the desired and actual pitch,

$$C(s) = |\beta - \beta^\star|, \tag{3.52}$$

so that $\frac{\partial C(s)}{\partial s} = 1$ if $\beta > \beta^\star$ and $\frac{\partial C(s)}{\partial s} = -1$ if $\beta < \beta^\star$ — i.e., the gradient is just the direction in which we should adjust our control. When $H = 1$, the PGSD update then takes on the very simple form:

$$\theta \leftarrow \begin{cases} \theta - \alpha\phi(s) & \text{robot didn't clear step} \\ \theta & \text{jump succeeded} \\ \theta + \alpha\phi(s) & \text{robot flipped backwards} \end{cases} \tag{3.53}$$

Despite the simplicity of this update rule, it works remarkably well in practice. We evaluated this PGSD variant on the LittleDog robot, attempting to climb the three steps as shown in Figure 3.10. After 28 failures (either flipping backwards or failing to clear the step), the robot successfully jumped all three steps for the first time. After 59 failures, the learning process had converged on a sufficiently accurate maneuver: the robot succeeded in immediately climbing all three steps for 13 out of the next 20 trials (and failures mostly involved failing to clear the step, where the robot would then retry the jump and typically succeed). This is far better than any policy we had been able to code by hand. A video of the learning process on the dog is available at `http://ai.stanford.edu/~kolter/ijrr09ld/`.

3.5 Summary

In this chapter, we have presented the policy gradient with signed derivative (PGSD) method, an approximate policy gradient algorithm that can be used to optimize a policy's performance *without* an accurate model of the system. The algorithm exploits the fact that in many control tasks, the sign of certain model derivative terms (relating how control inputs affect future states) are obvious, and many control tasks further have the structure that each state is primarily affected by only one control. In such settings, we have shown, both theoretically and empirically, that the PGSD algorithm can achieve very good performance. In particular, we evaluated the algorithm and a number of other policy gradient approaches (including those which have knowledge of the true model) on a two-link arm benchmark task, and demonstrated that on this task the PGSD algorithm greatly improves upon model free approaches and indeed performs as well as a policy gradient algorithm that is given the true model of the system. We additionally demonstrated the approach on two real-world domains: driving an RC car along a fixed route, and the challenging task of learning to jump up stairs with a quadruped robot.

While the PGSD algorithm can perform well on many domains, we emphasize that the algorithm is not applicable to all tasks. In particular, the algorithm exploits the fact that certain derivative signs are obvious (and fixed over time). While this is the case for many control tasks, there are also many domains where this is *not* the case. For instance, consider the two-link arm example considered in the chapter, but where only the second joint is actuated; this is the well-known "acrobot" domain, a task studied a great deal in Reinforcement Learning (Sutton and Barto, 1998). While this domain can still be controlled, the control input (torque applied to the second joint), no longer affects all the states in an "obvious" manner; in particular, the control input affects the first joint in a manner that depends on the other states of the system, and the sign of this relationship actually changes in different parts of the state space. Thus, the PGSD algorithm would not be applicable to such a domain. Furthermore, there are many domains where state variables are largely affected by more than one control. While we mentioned briefly that the PGSD algorithm could potentially be applied to such domains, the, both our theoretical and empirical analysis in this chapter does focus on the situation where this constraint holds. Thus, further analysis would be needed before we could make any performance claims about the PGSD algorithm in domains where this "orthogonality" of the control inputs does not hold.

Chapter 4

Dimensionality Reduction in Policy Search

In the previous chapter we explored how we can use inaccurate models within a policy search setting. However, we did not previously consider the actual size of the parameter vector θ, and in general a policy with many parameters is significantly harder to learn on real systems, regardless of the method we use, due to noise and the amount of data needed to accurately learn a large number of parameters; just *running* enough trajectories to gather sufficient data for this task is often impractical on the real system. In contrast, learning such high-dimensional policies *is* often possible in a model of the system (where there is potentially less noise, and we can simulate as much data as we would like), but here the inaccuracy of model is again a problem: because we would now use the model to actually simulate trajectories in the system, we need the model to be accurate in a predictive sense.

The main idea that we present in this chapter is to use an inaccurate model to identify a *low-dimensional subspace* of policy parameters, which can effectively decrease the number of parameters we need to learn on the real system. The intuition is that by considering a large set of *different* inaccurate models, we can identify a subspace of policy parameters that perform well in all these models, and then search only within this smaller subspace on the real system (which then typically requires less data).

The question remains as to how we generate these several different inaccurate models for use in the dimensionality reduction task. In practice, dynamics models are typically themselves parameterized by many different free parameters, which we will denote ψ to distinguish from the policy parameters θ; these could be, for example, mass parameters of the physical bodies, friction parameters, torque magnitudes, etc. Thus, a natural way to generate a large number of inaccurate models is to consider a distribution over these parameters and sample from the distribution to generate some number of different inaccurate models. Furthermore, if the model of the true system *is* known to a certain degree, but has additional unknown elements (such as a robot with known dynamics carrying an object with unknown weight), then we can of course only consider the distribution over these unknown parameters, and fix the known model parameters.

In this chapter we present the basic algorithm for learning a reduced subspace of controllers from a distribution over inaccurate models. We first present the algorithm in its generic form as a policy gradient approach, but we will also specifically consider the special case where the policy search procedure can be solved analytically via a least-squares problem. In this case we show that we can find the globally optimal reduced subspace using a dimensionality reduction algorithm known as Reduced Rank Regression (RRR) (Reinsel and Velu, 1998). Finally, we apply the method to the task of learning an omnidirectional trotting policy for a quadruped robot: a control law that is able to move the robot in any direction while turning in any manner. The full policy in this domain has almost 100 parameters, but by applying our method we can exploit an inaccurate model to learn a reduced policy class with only four parameters, which still works well on the real system.

4.1 Related Work

The work in this chapter relates to a number of areas from both machine learning and reinforcement learning, most notably the areas of transfer learning, specifically applied to reinforcement learning domains, and dimensionality reduction applied to control.

Transfer learning, broadly speaking, is a subfield of machine learning that focuses on the question of how learning in one domain can improve learning in another. Transfer learning methods have been applied to a number of reinforcement learning tasks: see e.g., (Taylor and Stone, 2009) for a survey of several approaches. The particular sub-area of transfer learning that relates most to the approach we present here is multi-task learning (Thrun, 1996; Caruana, 1997), a setting where a learning algorithm is presented with several different learning problems, and where each task has a different optimal solution but similar structure; in this setting, the goal is typically to use the previous tasks to learn some meta-structure of the tasks in general that can be used to speed up learning on a new task. The method we propose can certainly be viewed as an instance of multi-task learning (we learn policy classes in the different sampled models to improve our performance on the new task: learning a policy on the real system), though it does differ from most standard formulations of multi-task learning slightly the multiple tasks are constructed automatically and we do not ultimately care about performance on most of the tasks except insofar as it helps us learn the one task we care about — i.e., we don't care how well the policies perform in the inaccurate models, just how well the final controller performs on the real system. This idea of creating artificial "auxiliary problems" in multi-task learning has also been explored in (Ando and Zhang, 2005), but this was applied specifically to the task of semi-supervised learning, rather than the control domains we consider. Similarly, the work of Argyriou et al. (2006, 2007) bears a great deal of resemblance to the strategies we present here: like our algorithm, their work looks at extracting a feature subspace in the multi-task learning setting. However, the authors focus specifically on sparse feature constructions that can be expressed as convex optimization problems, which differs from our focus on rank constraints. More broadly, however, again their work focuses on a supervised learning setting rather than the control setting we consider here.

In addition to the survey described above, multi-task learning for reinforcement learning and control tasks has received a great deal of interest in recent years. These algorithms typically consider a setting where the agent interacts with a number of different related MDPs, $M_1, \ldots M_N$, and the goal of the algorithm is to use experience in past MDPs to improve performance when interacting with a new MDP; this is very similar to our setup, with the caveat again being that we do not care about performance of the method on any domain other than the real system. Methods for these settings differ in terms of what information they store to facilitate faster learning: (Tanaka and Yamamura, 2003; Konidaris and Barto, 2006) store value functions from past domains, and use these in different ways to speed up learning on a new task; (Mehta et al., 2008) assume identical transition distributions but different rewards across tasks, which enables them to transfer the reward-independent value function parameters; (Wilson et al., 2007) use a hierarchical Bayesian approach to learn an informative prior over MDP parameters themselves; (Taylor et al., 2007) use hand-coded functions that map state-action value functions from one task to state-action value function on another task. Our algorithm differs from these approaches in that it transfers *policy class* information between the different tasks, finding a subspace of control parameters that span good policies across all tasks. Perhaps most similar to our method is the method of (Li et al., 2009), who also transfer a form of policy parameters between tasks; however, their work explicitly considers a particular form of policy for partially observable (POMDP) domains, and the details of the approach are quite different from those we present here.

Our work also relates broadly to approaches for dimensionality reduction, specifically in control settings. Dimensionality reduction as a whole is a very broad topic, see e.g. (Fodor, 2002) for a survey, but in recent years there have been a number of applications of these methods to control tasks. In particular, there has been a great deal of work in recent years on dimensionality reduction for learning suitable bases for value function approximation, using Graph Laplacian methods in particular (Mahadevan and Maggioni, 2007). Also related in work on finding low-dimensional representation of belief state in POMDPs (Roy et al., 2005). Our work is similar in spirit to these approaches, but notably differs in that we are specifically looking for a low dimensional representation of the policy class.

Finally, as mentioned in the introduction, the notion of considering a distribution over possible models bears a great deal of similarity to robust control techniques (Zhou et al., 1996). The difference in our approach, though, is that we are not seeking a single control strategy that performs well in all domains. Rather, we are seeking a small *subspace* of control strategies, such that, in expectation, when we sample a dynamics model from our distribution, we can find a near-optimal controller for that model within the subspace.

4.2 Dimensionality Reduction Policy Search

Here we present our general method for (linear) dimensionality reduction in policy search algorithms. We first present the generic algorithm for a dimensionality reduction procedure in policy search; the presentation of the algorithm here is quite simple, but it also involves a non-trivial optimization problem. We then present progressively more specialized versions of the algorithm: a general policy gradient approach, and a globally optimal version that can be applied when the policy search task can be framed as a least squares problem.

We begin by formalizing the notion of parameterized dynamics models that we briefly described above. Let $\psi \in \mathbb{R}^\ell$ be a set of real-valued parameters that describe the dynamics model $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^n$. We use the notation

$$s_{t+1} = f(s_t, u_t; \psi) \tag{4.1}$$

to denote the dynamics model, evaluated at state s_t and control u_t, and parameterized by ψ. In reality most simulation models we will ever encounter will be of this form, so it is not a significant additional requirement that we assume a model of this type. Of course, the real system itself cannot be described in this manner, as there is no obviously notion of "model parameters" in the real world, but *simulated* models are typically of this form.

We also assume some distribution D over these model parameters. Naturally a large question to answer for our method is how we may come up with this distribution, because for a suitably broad distribution over models, there may be no coherent structure within the resulting policies — consider, for instance, a linear model $s_{t+1} = As_t + Bu_t$ where we have high uncertainty over all entries of A and B; in such a setting there would be no discernible structure in the optimal LQR gains K. However, we have found in practice that there is often a relatively small subset of model parameters over which we have a large amount of uncertainty. In physics simulators, for instance, it can often be fairly easy to obtain an accurate model of robot masses, link lengths, etc, but properties like friction and mass distribution within the rigid bodies can be much harder to model; thus, we would expect to put large uncertainty over these difficult-to-model parameters, and little to no uncertainty over the easy-to-model parameters.

Given this distribution, we sample N different sets of parameters from the distribution D,

$$\psi^{(1)}, \ldots, \psi^{(N)} \sim D. \tag{4.2}$$

We denote the value function for the the ith model as $J^{(i)}(s;\theta)$,

$$J^{(i)}(s;\theta) = \mathbf{E}\left[\sum_{t=0}^{H} C(s_t, u_t) \big| s_0 = s, s_{t+1} = f(s_t, \pi(s_t;\theta); \psi^{(i)})\right]. \tag{4.3}$$

Note that while we assume different dynamics models for each sample, we suppose here that the cost function is the same for all models; if there is some reason to prefer a distribution over cost functions, the extension is straightforward.

Our goal now is to find $\theta^{(1)}, \ldots, \theta^{(N)}$ that minimize these N different value functions. Optimizing these directly would simply be equivalent to solving N different policy search tasks, but we want to add the additional constraint that all the policy parameters lie is some low dimensional subspace of the entire set of policy parameters. In order to enforce this constraint, we form the design matrix

$$\Theta \in \mathbb{R}^{k\times N} \equiv \left[\theta^{(1)} \cdots \theta^{(N)}\right]. \tag{4.4}$$

The constraint that the θ's all lie in a low dimensional linear subspace of dimension p, is then equivalent to the constraint that $\operatorname{rank}(\Theta) = p$. Thus our dimensionality reduction policy search task can be stated generally as the optimization problem

$$\begin{aligned} \min_{\Theta \equiv \left[\theta^{(1)} \cdots \theta^{(N)}\right]} \quad & \sum_{i=1}^{N} J^{(i)}(s; \theta^{(i)}) \\ \text{s.t.} \quad & \operatorname{rank}(\Theta) = p. \end{aligned} \tag{4.5}$$

Note that the constraint that $\operatorname{rank}(\Theta) = p$ is equivalent to the constraint that $\Theta = UV$ for some $U \in \mathbb{R}^{k \times p}$ and $V \in \mathbb{R}^{p \times N}$ with the assumption being that $p \ll k$, so that we are reducing the dimension of the policy class significantly. Using this factorization, and the notation $V = [v^{(1)} \cdots v^{(N)}]$, we see that

$$\theta^{(i)} = Uv^{(i)} \tag{4.6}$$

so we can interpret the factorization in the following manner: the U matrix corresponds to a set of p "basis functions" (linear combinations of different policy parameters) that are shared across *all* the tasks, while the $v^{(i)}$ vectors correspond to the linear coefficients of the ith policy. Using this terminology, we can phrase our optimization problem in a slightly more general way, in terms of the actual distribution D rather than a set of samples drawn from the distribution

$$\min_{U} \mathbf{E}_{\psi \sim D} \left[\min_{v} J(s; Uv, \psi) \right] \tag{4.7}$$

where $J(s; \theta, \psi)$ denotes the value function for policy parameters θ and model parameters ψ. This is a very intuitive formulation of the goal of our dimensionality reduction procedure: we want to minimize over all possible linear subspaces, such that when we perform policy search within that linear subspace on a randomly sampled model, we get good performance.

Before describing a specialization of this algorithm, we make a few observations. First, the rank constraint in the optimization problem (4.5) is non-convex and so even if the original policy search procedure *was* globally solvable, this would evidently raise the possibility of new local optima in this dimensionality reduction policy search task. Equivalently, the factorization is (4.7) would be non-convex jointly in U and v.

However, given that policy search procedures are usually non-convex to begin with, this is typically not a large concern. And as we shall show shortly, for a certain case where the policy search procedure *is* solvable in a globally optimal way, it turns out that we are able to solve (4.5) optimally as well.

Second, to further solidify the intuition of our approach, we consider an alternative procedure, where we first perform policy search to find optimal parameters for each sample $\theta^{(1)\star}, \ldots, \theta^{(N)\star}$, then run PCA to find a linear subspace of the matrix $\Theta^\star = \left[\theta^{(1)\star} \cdots \theta^{(N)\star}\right]$. This will also procedure a factorization $\Theta^\star = \tilde{U}\tilde{V}$, but it should be apparent that

$$\mathbf{E}_{\psi \sim D}\left[\min_v J(s; \tilde{U}v, \psi)\right] \geq \mathbf{E}_{\psi \sim D}\left[\min_v J(s; U^\star v, \psi)\right] \tag{4.8}$$

where $U^\star$ is the solution to (4.5), i.e., the PCA solution does not give the best linear subspace in terms of minimizing the cost function (this follows trivially since (4.5) minimizes this criterion exactly). Intuitively, this is due to the fact that PCA produces the optimal reconstruction of the policy *coefficients*, but the coefficients themselves are somewhat irrelevant: what we really care about is the *performance* of the policy induced by these coefficients. Thus, PCA is really focusing on the wrong criterion here, and although it has its benefits (a globally optimal solution that is easily computed), these mainly indicate that we could potentially use the PCA solution as an initial point, and then optimize the criterion (4.5).

4.2.1 Dimensionality Reduction Policy Gradient

Of course, the objective function (4.5) is just an optimization problem, and so we need an actual algorithm for optimizing this objective. The simplest approach here is just to consider the factorized form $\Theta = UV$ and perform joint minimization over U and V. This could be accomplished either by alternatively optimizing over one of the matrices until converging to a local optima, or just by taking a gradient step with respect to both parameters. Since the latter is the conceptually simpler of the two approaches, and since the potential advantages of alternating minimization techniques (particularly pronounced when the optimization problem can be easily solved in one

```
Algorithm 2 Dimensionality Reduction Policy Gradient (DRPG)
Input:
  $D$: distribution over model parameters
  $N$: number of model samples
  $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^n$: parameterized dynamics model
  $C : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$: cost function
  $H \in \mathbb{Z}_+$: horizon
  $\pi : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^m$: parameterized policy
  $\alpha \in \mathbb{R}_+$: learning rate
  $U \in \mathbb{R}^{k \times p}, v^{(1)}, \ldots, v^{(N)} \in \mathbb{R}^p$: initial policy parameters
Initialization:
  • Draw $N$ samples of model parameters $\psi^{(1)}, \ldots \psi^{(N)} \sim D$.
Repeat until convergence:
  1. For $i = 1, \ldots, N$, compute gradient: $g^{(i)} \leftarrow \nabla_{\theta^{(i)}} J(s; \theta^{(i)}, \psi^{(i)}) \quad (\theta^{(i)} \equiv Uv^{(i)})$.
  2. Take a gradient steps in $U$ and $v^{(i)}$ parameters:
    • Update $U$: $U \leftarrow U - \alpha \left( \sum_{i=1}^{N} g^{(i)} (v^{(i)})^T \right)$.
    • For $i = 1, \ldots, N$, update $v^{(i)}$: $v^{(i)} \leftarrow v^{(i)} - \alpha U^T g^{(i)}$.
For new dynamics $s_{t+1} = f'(s_t, u_t)$:
  • Minimize value function over parameters in the span of $U$: $\min_{v'} J(s; Uv', f')$.
```

variable while keeping the other fixed) are unclear when the objective is convex in neither variable alone, we adopt this method as our basic algorithm for optimizing the objective (4.5). We describe the full method in Algorithm 2. We present the algorithm assuming a deterministic dynamics model, but it can be easily extended to the stochastic setting via the methods described in Chapter 2.

To see how the algorithm is taking gradient steps in U and $v^{(i)}$, we discuss here how to compute the gradient terms $\nabla_U J(s; Uv^{(i)}, \psi^{(i)})$ and $\nabla_{v^{(i)}} J(s; Uv^{(i)}, \psi^{(i)})$. While not identical to previously described terms, these can easily be related to the standard policy gradient. Using U_j to denote the jth column of U, and $\theta = Uv^{(i)}$ we have

$$\frac{\partial J(s; Uv^{(i)})}{\partial U_j} = \frac{\partial J(s; \theta)}{\partial \theta} \frac{\partial Uv^{(i)}}{\partial U_j} = \frac{\partial J(s; \theta)}{\partial \theta} (v_j^{(i)} I) \tag{4.9}$$

and

$$\frac{\partial J(s; Uv^{(i)})}{\partial v^{(i)}} = \frac{\partial J(s;\theta)}{\partial \theta}\frac{\partial Uv^{(i)}}{\partial v^{(i)}} = \frac{\partial J(s;\theta)}{\partial \theta}U \tag{4.10}$$

so

$$\nabla_U J(s; Uv^{(i)}) = (\nabla_\theta J(s;\theta))(v^{(i)})^T, \quad \text{and} \quad \nabla_{v^{(i)}} J(s; Uv^{(i)}) = U^T(\nabla_\theta J(s;\theta)). \tag{4.11}$$

Thus, we can compute $\nabla_\theta J(s;\theta,\psi^{(i)})$ for $\theta = Uv^{(i)}$ using the standard (model-based) methods described in Chapter 2, and then convert these to the factorized parameter gradients using (4.11). It should be apparent that the algorithm is updating the U and $v^{(i)}$ terms according to these gradients.

4.2.2 Least Squares Policy Search

In some situations the task of policy search can in fact be expressed as a least squares problem. While this is obviously a very restricted special case of policy search, given the preceding discussion in this work, we indeed will soon focus on a control task (learning an omnidirectional trot gait on a quadruped robot) that can be framed in this manner. In such a setting, it turns out that we are able to solve the optimization problem (4.5) for dimensionality reduction policy search *optimally*, despite the non-convexity of the problem; we do this using an algorithm known as Reduced Rank Regression (Reinsel and Velu, 1998).

More formally, we suppose that the optimal parameters for the ith model can be solved (or at least, approximated) via the least-squares problem

$$\min_\theta J(\theta) \equiv \|X\theta - y^{(i)}\|_2 \tag{4.12}$$

for some $X \in \mathbb{R}^{q\times k}$, and $y^{(i)} \in \mathbb{R}^q$. Setting up a policy search task in such a manner is not an obvious transformation, so we will shortly describe in some detail how one policy search task for the quadruped robot can be framed in this manner. For the time being, though, we will take (4.12) as an assumption, and demonstrate how to solve the optimization problem (4.5) optimally.

Defining the policy matrix Θ as in (4.4) and defining the design matrix $Y \in \mathbb{R}^{q\times N}$

similarly as

$$Y = \left[y^{(1)} \cdots y^{(N)}\right] \tag{4.13}$$

then the optimization problem (4.5) becomes equivalent to

$$\begin{aligned} \min_{\Theta} \quad & \|X\Theta - Y\|_F^2 \\ \text{s.t.} \quad & \operatorname{rank}(\Theta) = p. \end{aligned} \tag{4.14}$$

The optimal solution to this task, given by the Reduced Rank Regression algorithm is,

$$\Theta^\star = (X^T X)^{-1} X^T Y W W^T \tag{4.15}$$

where the columns of $W \in \mathbb{R}^{N \times p}$ correspond to the p principal eigenvectors of the matrix

$$Y^T X (X^T X)^{-1} X^T Y. \tag{4.16}$$

This result is proved in (Reinsel and Velu, 1998, Theorem 2.2). The optimal values for U and V can be read directly from this solution as

$$U = (X^T X)^{-1} X^T Y W, \quad V = W^T. \tag{4.17}$$

Notice that the Reduced Rank Regression solution can be interpreted as the least squares solution $(X^T X)^{-1} X^T Y$ projected into the subspace spanned by W. When W is full rank (i.e., there is no rank constraint), then $WW^T = I$, and the solution coincides with the least squares solution, as we would expect.

4.3 Application to Omnidirectional Path Following

The task we are interested in here is one of omnidirectional path following using a “trot” gait on a quadruped robot. Again, we will be using the LittleDog robot, described in greater detail in Chapter 6, but here we briefly describe the particular task we focus on. Omnidirectional path following refers to the task of following an arbitrary spline with the robot's body, with turning in an arbitrary manner; in other words, the robot acts as a holonomic system: able to move in any direction while orienting itself in any manner. Trot gaits refer to gaits that move two feet of the robot at once (trots move diagonally opposing feet at the same time), as opposed to slower "walking" gaits that move only one foot at a time.

The primary challenging in developing a controller capable of omnidirection trotting is one of *balance*. Because only two of the robot's feet are on the ground at one time, the robot will not be statically stable; furthermore, the LittleDog robot does not have the requisite power in its legs to maintain any kind of dynamic stability during the trot. Rather, our approach to a trot gait merely attempts to balance "on average"; as soon as the robot raises its legs it will start to fall, but by properly placing the center of gravity (COG) before taking the step, our goal is to balance the robot as well as possible, such that we can quickly move both moving feet before the robot has tipped to either side.

4.3.1 A Balancing Policy for Omnidirectional Trotting

When the robot is trotting in a fixed direction, then learning a good balancing policy is fairly straightforward. In particular, we can achieve good balance by simply offsetting the robot's COG some amount with respect to the average "center" position of the moving feet. In other words, finding a policy for balancing when traveling in a fixed direction can be framed as a policy search task with two parameters $\theta = (x_{\text{off}}, y_{\text{off}})$. We can thus use the signed derivative algorithm from the previous chapter to easily estimate these parameters. Skipping the precise details of the formulation, the final form of the signed derivative update for this task is to repeat the update

$$\begin{aligned} x_{\text{off}} &\leftarrow x_{\text{off}} + \alpha\left((t_{\text{FL}} - t_{\text{BR}}) + (t_{\text{FR}} - t_{\text{BL}})\right) \\ y_{\text{off}} &\leftarrow y_{\text{off}} + \alpha\left((t_{\text{FL}} - t_{\text{BR}}) - (t_{\text{FR}} - t_{\text{BL}})\right) \end{aligned} \tag{4.18}$$

where α is the learning rate, and $t_{\text{FL}}, t_{\text{FR}}, t_{\text{BL}}, , t_{\text{BR}}$ are the touchdown times for the four different feet (front left, front right, back left, back right), as measured by foot force sensors on the robot. Intuitively, this update is adjusting the COG location in response to which of the robot's feet touches down first: for example, if the back left leg hits before the front right, $t_{\mathrm{FR}} - t_{\mathrm{BL}} > 0$, so x_{off} is increased, shifting the COG more forward. A video of the robot learning how to balance using this method for forward walking is available at `http://ai.stanford.edu/~kolter/ijrr09ld/`.

The challenge comes, however, when the robot is not walking in a fixed manner, but is constantly changing its direction and rate of turn, as is the case for omnidirectional path following. In particular, each step of a trot gait in an omnidirectional motion is described by two parameters: a *direction* $\chi \in [-\pi, \pi]$, which denotes the direction of travel relative to the robot's orientation (so that $\chi = 0$ implies we are walking forward), and a *turn angle* $\omega \in [-\pi, \pi]$, which denotes the amount to turn per step (so that $\omega = 0$ implies we are not turning, and $\omega = \pi/2$ implies we turn 90 degrees to the left in one step). Of course, different directions and turn angles necessitate different COG offsets, denoted

$$x_{\mathrm{off}}(\chi, \omega),\ y_{\mathrm{off}}(\chi, \omega) \tag{4.19}$$

and through experimentation we have found that it is non-trivial to devise a simple strategy to choose these COG offsets as a function of the direction and turn angle. Thus, we parametrize a function to choose these COG offsets, and let the parameters of this function be our policy parameters that we want to find.

Since the direction angle and turning angle are inherently periodic — i.e., a direction of 2π is identical to a direction angle of 0 — the Fourier bases are a natural means of representing these functions. We therefore represent $x_{\mathrm{off}}(\chi, \omega)$ and $y_{\mathrm{off}}(\chi, \omega)$ as

$$x_{\mathrm{off}}(\chi, \omega) = \theta_x^T \phi(\chi, \omega),\ y_{\mathrm{off}}(\chi, \omega) = \theta_y^T \phi(\chi, \omega) \tag{4.20}$$

where

$$\theta = \begin{bmatrix} \theta_x \in \mathbb{R}^{k/2} \\ \theta_y \in \mathbb{R}^{k/2} \end{bmatrix} \tag{4.21}$$

are the parameters of our policy, and

$$\phi(\chi, \omega) = \begin{bmatrix} \cos(i\chi)\cos(j\omega) \\ \sin(i\chi)\cos(j\omega) \\ \cos(i\chi)\sin(j\omega) \\ \sin(i\chi)\sin(j\omega) \\ \vdots \end{bmatrix}, \quad i, j = 0, 1, \ldots \tag{4.22}$$

denote the first $k/2$ principal Fourier basis functions of χ and ω — here the range of i and j are chosen so that the dimension of ϕ is also $k/2$. In particular, for this domain we found that in order to express a sufficiently general policy to balance properly, we needed to choose the range of i and j above such that there are $k = 98$ parameters.

4.3.2 Policy Search as Least Squares

Given this class of policies, we can solve for the optimal policy class θ in the following manner. For any *fixed* χ and ω, we can run the signed derivative approach to find the x_{off} and y_{off} that are optimal for this direction and turn angle. Then, given a collection of direction angles, turning rates, and their corresponding center offsets, we can learn the coefficients θ_x and θ_y by least squares regression. Specifically, if we are given a set of n direction angles, turning rates, and resulting x center offsets, $\{\chi_i, \omega_i, x_{\text{off},i}\}$, $i = 1 \ldots n$, then we can learn the parameters $\theta_x \in \mathbb{R}^k$ by solving the optimization problem

$$\min_{\theta_x} \|y - X\theta_x\|_2^2 \tag{4.23}$$

where $X \in \mathbb{R}^{n \times k}$ and $y \in \mathbb{R}^n$ are design matrices defined as

$$X = [\phi(\chi_1, \omega_1) \ldots \phi(\chi_n, \omega_n)]^T \quad y = [x_{\text{off},1} \ldots x_{\text{off},n}]^T. \tag{4.24}$$

The solution to this problem is given by $\theta_x = (X^T X)^{-1} X^T y$, and using a well known sample complexity result (Anthony and Bartlett, 1999), we need $\Omega(k)$ data points to find such a solution.

Unfortunately, computing a sufficient number of center offsets on the real robot is a time-consuming task. Although the signed derivative can find an optimal COG offset in about a minute under ideal circumstances, several difficulties arise if we try to apply this algorithm more than 100 times to find these different parameters: most notably, running the robot this long, especially at the early stages of learning where the balancing policy is quite poor, causes a great deal of strain on the robot's gears, often degrading joint angle calibration by bashing its feet into the ground. And although it is significantly easier to find proper joint offsets in a simulation model (we can run the same algorithm without any fear of damaging the hardware), it is difficult to create a simulator that accurately reflects the center offset positions in the real robot. Indeed, we invested a great deal of time trying to learn parameters for the simulator that reflected the real system as accurately as possible, but still could not build a simulator that behaved sufficiently similarly to the real robot. Thus, in order to learn a policy on the real system we applied our dimensionality reduction algorithm in the following manner.

4.3.3 Experimental Setup and Results

Here we present experimental results on applying our method to learn a controller for omnidirectional path following. The simulator we built is based on the physical specifications of the robot and uses the Open Dynamics Engine (ODE) physics environment.[1]

Our experimental design was as follows: We first sampled 100 simulation models from a distribution over the simulator parameters. In particular, we varied the simulators primarily by adding a constant bias to each of the joint angles, where these bias terms were sampled from a Gaussian distribution. We also experimented with varying several other parameters, such as the centers of mass, weights, torques, and friction coefficients, but found that none of these had as great an effect on the resulting policies as the joint biases. This is somewhat unsurprising, since the real robot has constant joint biases. However, we reiterate the caveat mentioned in the previous section: it is not simply that we need to learn the correct joint biases in

[1]ODE is available at http://www.ode.org.

order to achieve a perfect simulator; rather, the results suggest that perturbing the joint biases results in a class of policies that is robust to the typical variations in model dynamics.

In each of these simulation models, we used the online balancing algorithm described in Section 4.3.1 to find center offsets for a variety of fixed directions and turning rates. For each model, we generated 100 data points, with turning angles spaced evenly between -1.0 and 1.0, and direction angles from 0 to 2π. We constrained the centering function (in both the x and y directions), to be a linear combination of the first $k = 49$ Fourier bases of the direction angle and turning rate. We then applied the Reduced Rank Regression algorithm to learn a low-dimensional representation of this function with only $\ell = 2$ parameters, effectively reducing the number of parameters by more than 95%. Two bases was the smallest number that achieved a good controller with the data we collected: one basis vector was not enough, three basis vectors performed comparably to two, but had p no visible advantage, and four basis vectors began to over-fit to the data we collected from the real robot, and started to perform worse.

Finally, to learn a policy on the actual robot, we used the online centering algorithm to compute proper center locations for 12 fixed maneuvers on the robot and used these data points to estimate the parameters of the low-dimensional policy. In greater detail, for 12 different turning angles and direction angles, we ran the online learning algorithm presented in Section 4.3.1, to find the correct x and y center offsets in the real system. Given these 12 data points (far too few to learn the full dimensional policy, we searched within the linear subspace found by our algorithm to find a controller that captured these optimal center locations as closely as possible, and used this controller for balancing the robot.

To evaluate the performance of the omnidirectional gait and the learned centering function, we used three benchmark path splines: 1) moving in a circle while spinning in a direction opposite to the circle's curvature; 2) moving in a circle, aligned with the circle's tangent curve; and 3) moving in a circle keeping a fixed heading. To quantify performance of the robot on these different tasks, we used four metrics: 1) the amount of time it took for the robot to complete an entire loop around the circle; 2) the root

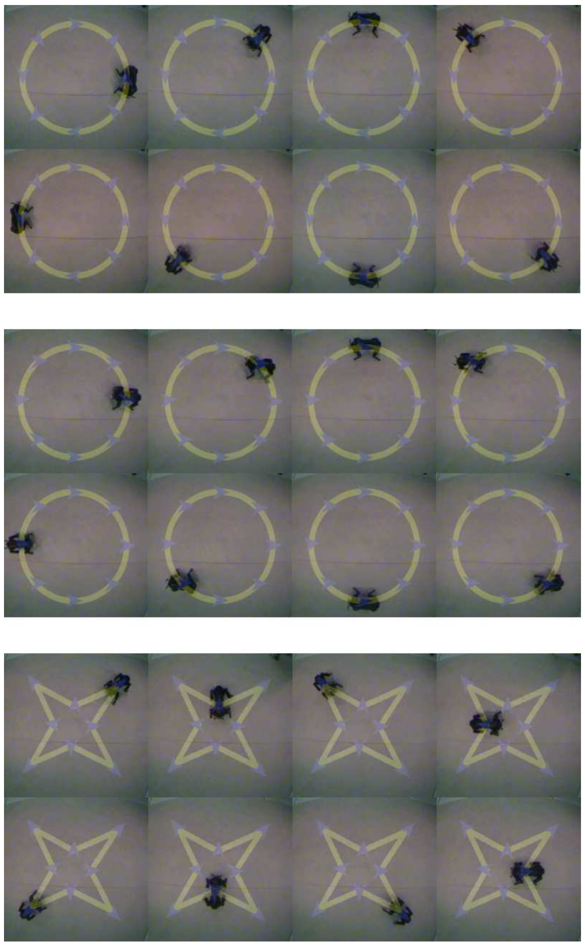
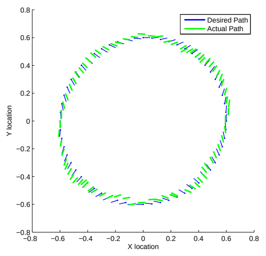
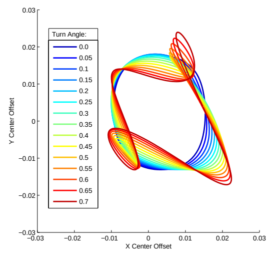


Figure 4.1: Pictures of the quadruped robot following several paths.



(a) Desired and actual trajectories for the learned controller on path 1.



(b) Learned center offset curves for several different turning angles.

Figure 4.2: Trajectory and center offsets for the learned controller.

Metric	Path	Learned Centering	No Centering	Hand-tuned Centering
Loop Time (sec)	1	**31.65 ± 2.43**	46.70 ± 5.94	34.33 ± 1.19
	2	**20.50 ± 0.18**	32.10 ± 1.79	31.69 ± 0.45
	3	**25.58 ± 1.46**	40.07 ± 0.62	28.57 ± 2.21
Foot Hit RMSE (sec)	1	**0.092 ± 0.009**	0.120 ± 0.013	0.098 ± 0.009
	2	**0.063 ± 0.007**	0.151 ± 0.016	0.106 ± 0.010
	3	**0.084 ± 0.006**	0.129 ± 0.007	0.097 ± 0.006
Distance RMSE (cm)	1	**1.79 ± 0.09**	2.42 ± 0.10	1.84 ± 0.07
	2	**1.03 ± 0.36**	2.80 ± 0.41	1.98 ± 0.21
	3	**1.58 ± 0.11**	2.03 ± 0.07	1.85 ± 0.16
Angle RMSE (rad)	1	0.079 ± 0.006	0.075 ± 0.009	**0.067 ± 0.013**
	2	**0.070 ± 0.011**	**0.070 ± 0.002**	0.077 ± 0.006
	3	**0.046 ± 0.007**	0.058 ± 0.012	0.071 ± 0.009

Table 4.1: Performance of the different centering methods on each of the three benchmark paths, averaged over 5 runs, with 95% confidence intervals.

mean squared difference of of the foot hits (i.e., the time difference between when the two moving feet hit the ground); 3) the root mean squared error of the robot's Euclidean distance from the desired path; and 4) the root mean squared difference between the robot's desired angle and its actual angle.

Note that these metrics obviously depend on more than just the balancing controller — speed, for example, will of course depend on the actual speed parameters of the trot gait. However, we found that good parameters for everything but the balancing controller were fairly easy to choose, and the same values were optimal, regardless of the balancing policy used. Therefore, the differences in speed/accuracy between the different controllers we present is entirely a function of how well the controller is capable of balancing — for example, if the robot is unable to balance it will slip frequently and its speed will be much slower than if it can balance well.

We note that prior to beginning our work on learning basis functions, we spent a significant amount of time attempting to hand-code a centering controller for the robot. We present results for this hand-tuned controller, since we feel it represents an accurate estimate of the performance attainable by hand tuning parameters. We also evaluated the performance of the omnidirectional gait with no centering. The controller using a policy based entirely from the "mean" simulator model performed comparably to the policy with no balancing at all; this suggests that in this case, our simulation model was quite different from the real system, but it still enabled us to learn a suitable subspace.

Figure 4.1 shows pictures of the robots following some of the benchmark paths, as well as an additional star-shaped path. Videos of these experiments are available at: `http://ai.stanford.edu/~kolter/omnivideos`.

Table 4.1 shows the performance of each centering method, for each of the four metrics, on all three benchmark paths.[2] As can be seen, the learned controller outperforms the other methods in nearly all cases. As the distance and angle errors indicate, the learned controller was able to track the desired trajectory fairly accurately. Figure 4.2(a) shows the actual and desired position and orientation for the learned centering controller on the first path. Figure 4.2(b) shows the learned center offset predictor trained on data from the real robot. This figure partially explains why hand-tuning a controller can be so difficult: at higher turning angles the proper centers form unintuitive looping patterns.

4.4 Summary

In this chapter we presented a method for dimensionality reduction in policy search tasks. The algorithm finds a low dimensional subspace of control parameters that contains good controllers over a variety of different models: thus, by considering a distribution over possible models we can find a reduced subspace of possible control policies. This reduces the dimension of the policy search task on the real system, and can make such methods much more practical in real domains. The method is particularly applicable to situations where we have some uncertainty over certain model parameters (either due to poor modeling or the fact that the system can change over time), and we want to find a small subspace of controllers that perform well over this entire distribution. We began the chapter with a general form of the algorithm applicable to policy gradient methods, then specialized the algorithm to the special case where the policy search task can be framed as a least squares problem. Using the latter formulation, we apply our method to the task of learning a low-dimensional

[2]The foot hit errors should not be interpreted too literally. Although they give a sense of the difference between the three controllers, the foot sensors are rather imprecise, and a few bad falls greatly affect the average. They therefore should not be viewed as an accurate reflection of the foot timings in a typical motion.

control parametrization for omnidirectional path following with the LittleDog robot, and demonstrate the method is able to learn a highly reduced subset of control laws that achieve very good performance on the task.

Of course, we again note that the method we present here is not applicable to all control tasks. In particular, although we are using an inaccurate model to learn the low-dimensional control parametrization, we still require that the model capture some degree of accuracy, or else we may be unable to achieve good performance on the real system using *any* policy from this subspace. Thus, for domains where the dynamics are difficult to model, even in the sense of determining some distribution over possible models, our method likely will not find a suitable controller subspace. Nonetheless, as we demonstrate in this chapter, there are certainly cases where we *can* find a good subspace of controllers, even when the exact dynamics of the true system are very difficult to model, and we have demonstrated the potential of the algorithm in such a setting.

Chapter 5

Multi-model Control for Mixed Closed-loop/Open-loop Behavior

The previous two chapters considered methods for exploiting inaccurate models, though in both cases the model still captured *some* element of the true model correctly: either through the dominant signs of the model, or by capturing some distribution over unknown parameters. In contrast, this chapter deals with the scenario where a model truly *cannot* capture the relevant portions of a system, either because the true system is too complex or because of non-Markovian and/or hidden state variables. In such settings it may indeed seem as though we would be unable to use a model in order to obtain good performance on the system. And to some extent this is true: if the model is unable to capture the relevant portions of the system dynamics, then it seems unlikely (almost by definition), that we could use the model alone to create a good control policy.

Fortunately, there is an aspect of many control tasks that mitigates this problem. In particular, we repeatedly observe in practice that even if the model is difficult to learn, real dynamical systems are often remarkably deterministic over short periods of time: if we execute the same sequence of control inputs from an identical or nearly identical initial state, the system often behaves in a very similar manner, even if it is very hard to predict how the system would respond to different inputs. We have repeatedly observed such behavior in the LittleDog robot (many static climbing maneuvers can be executed in an entirely open-loop manner as long as the robot carefully positions itself first), in helicopter control (aerobatic maneuvers are executed virtually open-loop by skilled pilots, again once the system has been put into a proper initial state), in the aerodynamic task of landing a hang-glider (approach to the runway involves careful control, but the actual landing is a largely open loop procedure that stops the glider quickly by inducing an uncontrolled stall), and of course in the car-driving task that we will describe in great detail. Intuitively, these observations motivate an alternating control strategy, where we would want to actively control the system using classical methods in well-modeled regimes, but execute open-loop trajectories in poorly modeled regions, provided we have seen a previous demonstration of the desired behavior. Indeed, as we will discuss shortly, such strategies have previously been used to control a variety of systems, though these have mostly relied on hand-tuned switching rules to determine when to execute the different types of control.

In this chapter, we make two contributions. First, we develop a probabilistic approach to mixed open-loop and closed-loop control that achieves performance far superior to each of these elements in isolation. In particular, we propose a method for applying optimal control algorithms to a probabilistic combination of 1) a "simple" model of the system that can be highly inaccurate in some regions and 2) a dynamics model that describes the expected system evolution when executing a previously-observed trajectory. By using variance estimates of these different models, our method probabilistically interpolates between open-loop and closed-loop control in an optimal manner, thus producing mixed controllers without the typical hand-tuning. Second, we apply this algorithm to the challenging task of a "sliding parallel park" on a full-sized autonomous car — that is, accelerating a car to 25 mph, then slamming on the brakes while turning the steering wheel, skidding the car sideways into a narrow desired location. To the best of our knowledge, this is the first autonomous demonstration of such behavior on a car, representing the state of the art in accurate control for such types of maneuvers.

5.1 Related Work

There is a great deal of work within both the machine learning and control communities that is relevant to the work we present here. As mentioned previously, the mixed use of open-loop and closed-loop control is a common strategy in robotics and control (e.g., Atkeson and Schaal, 1997; Hodgins and Raibert, 1990; Gillula et al., 2010), though typically these involve hand-tuning the regions of closed-loop/open-loop control. One exception to this is the work of Gillula et al. (2010), developed concurrently to the techniques we present here, which use robust control methods to determine execution regions for the different controllers; however, the algorithms and focus of this method and our own are quite different. The work of Hansen et al., (Hansen et al., 1996) also considers a mixture of open-loop and closed-loop control, though here the impetus for such control comes from a cost to sensing actions, not from difficulty of modeling the system, and thus the algorithms are very different.

Our work also relates to work on multiple models for control tasks (Murray-Smith and Johansen, 1997). However, with few exceptions the term "multiple models" is typically synonymous with "local models," building or learning a collection of models that are each accurate in a different portion of the state space — for example, *locally linear models*, (Schaal, 1994; Christopher G. Atkeson, 1997; Vijayakumar et al., 2005; Doya et al., 2002) or non-parametric models such as Gaussian Processes (Ko et al., 2007). Standard local models are typically defined such that the model designer must manually specify the regions of validity for each of the model, and only *one* model is used at each point in the state space; this contrasts with our approach, where we use a Gaussian observation models to combine predictions from all models jointly, using estimates of their variance. More broadly, though, fundamentally our method certainly could be considered a local model method, but novelty of our approach comes largely from the specific nature of the different models we use, and how this naturally leads to the desired behavior of mixed open-loop and closed-loop control.

From the control literature, our work perhaps most closely resembles the work on multiple model adaptive control (Narendra and Balakrishnan, 1997; Schott and Bequette, 1997). Like our work, this line of research takes motivation from the fact that different models can capture the system dynamics with different degrees of uncertainty, but again like the previous work in machine learning, this research has typically focused on learning or adapting multiple local models of the system to achieve better performance.

Our work also relates somewhat more loosely to a variety of other research in reinforcement learning. For instance, the notion of "trajectory libraries" has been explored for reinforcement learning and robotics (Stolle and Atkeson, 2006; Stolle et al., 2007), but this work has primarily focused on the trajectories as a means for speeding up *planning* in a difficult task. The notion of learning *motor primitives* has received a great deal of attention recently (e.g. (Peters and Schall, 2004)), though such research is generally orthogonal to what we present here. These motor primitives typically involve feedback policies that are not model-based, and we imagine that similar techniques to what we propose here could also be applied to learn how to switch between model-based and motor-primitive-based control. Indeed to some extent our work touches on the issue of model-based versus model-free reinforcement learning (Atkeson and Santamaria, 1997) where our open-loop controllers are simply a very extreme example of model-free control, but we could imagine applying similar algorithms to more general model-free policies.

Finally, there has been a great deal of past work on autonomous car control, for example (Hoffmann et al., 2007). Some of this work has even dealt with slight forays into control at the limits of handling (Hsu and Gerdes, 2005), but all such published work that we are aware of demonstrates significantly less extreme situations than what we describe here, or includes only simulation results. We are also aware of another group at Stanford working independently on control at the limits of handling (Gerdes, 2009), but this work revolves more around physics-based techniques for stabilizing the system in an unstable "sliding" equilibrium, rather than executing maneuvers with high accuracy.

5.2 A Probabilistic Framework for Multiple Model Control

Here we present a general method for combining multiple control models, and demonstrate that when we apply this algorithm to two particular types of models, we naturally achieve behavior that trades off between executing a closed-loop maneuver when we have a good model of the dynamics, and an open-loop maneuver when our model is inaccurate. In particular, we will first present an extension of LQR, which we call Multi-model LQR, that can deal with predictions from multiple probabilistic dynamics models. We then develop a simple model for capturing the dynamics of previously observed trajectories, and we show that when we provide both this model and an inaccurate dynamics model to our Multi-model LQR algorithm, the algorithm will smoothly interpolate between closed-loop and open-loop control in the proper manner.

To formalize our framework, we are operating in the trajectory-following setting from Chapter 2, where we want to follow some desired trajectory $\tau^\star = (s_0^\star, u_0^\star, \ldots, s_H^\star, u_H^\star)$ (which we assume is realizable on the real system), and the cost function is given by

$$C_t(s,u) = (s - s_t^\star)^T Q (s - s_t^\star) + (u - u_t^\star)^T R (u - u_t^\star). \tag{5.1}$$

The ability to execute the trajectory on the real system does not imply that we will necessarily be able to execute the entire sequence repeatedly in an open-loop manner. In particular, although we motivated our overall approach by appealing to the fact that real system were often remarkably deterministic, we need to add the caveat that this does of course only apply over relatively short timescales (on the order of a few seconds), and only when initial states are *very* similar. In the setting of general trajectory following, where the trajectory may span many seconds and the initial state of the system may differ from the initial state in trajectory, we need additional methods (such as LQR) to control the system.

We also suppose that we have access to an (inaccurate) stochastic model of the form

$$s_{t+1} = f(s_t, u_t) + \epsilon_t \tag{5.2}$$

and the assumption is that this model may be accurate in certain portions of the state space, but inaccurate in others. To capture this inaccuracy, we extend our previous definitions and assume that the noise term for this model is a state and control dependent Gaussian, i.e.,

$$\epsilon_t \sim \mathcal{N}(0, \Sigma(s_t, u_t)). \tag{5.3}$$

The covariance terms $\Sigma_i(s_t, u_t)$ are interpreted as maximum likelihood covariance estimates, i.e., for *true* next state s_{t+1},

$$\Sigma(s_t, u_t) = E\left[(s_{t+1} - f(s_t, u_t))(s_{t+1} - f(s_t, u_t))^T\right]. \tag{5.4}$$

This notion of the variance as a maximum likelihood estimate is very important, because it implies that the variance term for our model actually captures two sources of error: 1) the true stochasticity of the world and 2), the inaccuracy of the model. Because we assume that the stochasticity of real system is small over short time periods, this implies that the variance term will typically be dominated by the second element; thus, the model variance in a sense acts as a proxy for the inaccuracy of the model, one that algorithms can exploit to determine when to trust the model and when to execute open loop controls.

5.2.1 LQR with multiple probabilistic models

Here we present our general extension of LQR, called Multi-model LQR, for combining multiple probabilistic models with the LQR algorithm. We suppose that we are given two approximate dynamical models of the form above, M_1 and M_2 — the generalization to more models is trivial and we consider two just for simplicity of presentation. To combine multiple probabilistic models of this form, we interpret each prediction as an independent observation of the true next state, and can then compute the posterior distribution over the next state given both models by a standard manipulation of Gaussian probabilities,

$$p(s_{t+1}|M_1, M_2) = \mathcal{N}(\bar{f}(s_t, u_t), \bar{\Sigma}(s_t, u_t)) \tag{5.5}$$

where

$$\bar{\Sigma} = \left(\Sigma_1^{-1} + \Sigma_2^{-1}\right)^{-1}, \text{ and } \bar{f} = \bar{\Sigma}\left(\Sigma_1^{-1} f_1 + \Sigma_2^{-1} f_2\right) \tag{5.6}$$

and where the dependence on s_t and u_t is omitted to simplify the notation in this last line. In other words, the posterior distribution over next states is a weighted average of the two predictions, where each model is weighted by its inverse covariance. This combination is analogous to the Kalman filter measurement update.

We could now directly apply LQR to this joint model by simply computing $\bar{f}(s_t^\star, u_t^\star)$ and its derivatives at each point along the trajectory. Specifically, we could run LQR exactly as described in Chapter 2, using the linear approximation of the error dynamics

$$\delta s_{t+1} = A_t \delta s_t + B_t \delta u_t \tag{5.7}$$

where $\delta s_t \equiv s_t - s_t^\star$ and $\delta u_t \equiv u_t - u_t^\star$, where A_t and B_t are the Jacobians of $\bar{f}$ evaluated at $s_t^\star$ and $u_t^\star$,

$$A_t = \frac{\partial \bar{f}(s_t^\star, u_t^\star)}{\partial s_t^\star}, \quad B_t = \frac{\partial \bar{f}(s_t^\star, u_t^\star)}{\partial u_t^\star}, \tag{5.8}$$

and the noise term is given by $w_t \sim \mathcal{N}(0, \bar{\Sigma}(s_t^\star, u_t^\star))$. However, combining these models in this manner will lead to poor performance. This is due to the fact that we would be computing $\Sigma_1(s_t^\star, u_t^\star)$ and $\Sigma_2(s_t^\star, u_t^\star)$ only at the desired location, which can give a very inaccurate estimate of each model's true covariance. For example, consider a learned model that is trained only on the desired trajectory; such a model would have very low variance at the actual desired point on the trajectory, but much higher variance when predicting nearby points. Thus, what we really want to compute is the *expected* covariance of each model, in the region where we expect the system to be. We achieve this effect by maintaining a *distribution* D_t over the expected state (errors) at time t (i.e., a distribution over δs_t). Given such a distribution, we can approximate the average covariance

$$\Sigma_i(D_t) \equiv \mathbf{E}_{\delta s_t, \delta u_t \sim D_t} \left[\Sigma_i(s_t^\star + \delta s_t, u_t^\star + \delta u_t) \right] \tag{5.9}$$

via sampling or other methods. This notion of maintaining a distribution over expected states is well-known in reinforcement learning (Bagnell et al., 2004), but we have not seen a specialization of such ideas to this form of LQR algorithm.

The last remaining element we need is a method for computing the state (error) distributions D_t. Fortunately, given the linear model assumptions that we already employ for LQR, computing this distribution analytically is straightforward. In particular, we first assume that our initial state distribution D_0 is a zero-mean Gaussian with mean zero and covariance Γ_0. Since LQR outputs a closed-loop controller of the form $\delta u_t = K_t \delta s_t$, our closed-loop state error dynamics evolve according to the linear model $\delta s_{t+1} = (A_t + B_t K_t)\delta s_t$, the covariance of D_t, Γ_t, is updated by

$$\Gamma_{t+1} = (A_t + B_t K_t)\Gamma_t (A_t + B_t K_t)^T + \bar{\Sigma}(D_t). \tag{5.10}$$

Of course, since we average the different models according to the $\Sigma_i(D_t)$, changing D_t will therefore also change A_t and B_t. Thus, we iteratively compute all these quantities until convergence. A formal description of the algorithm is given in Algorithm 3.

5.2.2 A dynamics model for open-loop trajectories

The method above is a general algorithm for combining probabilistic models with an LQR-based approach, but recall that our overall goal is to combine closed-loop control in well-modeled regions with open-loop control in poorly modeled regions. Thus, in this section we develop a probabilistic model that describes how the state will evolve when executing a sequence of previously observed controls; combining this with a typical inaccurate dynamics model using the algorithm above will then naturally lead to the desired behavior. While generally one would want to select from any number of possible trajectories to execute, for the simplified algorithm in this section we consider only the question of executing the control actions of the ideal trajectory $u_{0:H}^\star$.

Algorithm 3 Multi-model LQR (MM-LQR)

```
Input:
  $\Gamma_{0:H}$: initial state error covariance
  $M_1, M_2$: probabilistic dynamics models
  $s^\star_{0:H}, u^\star_{0:H}$: desired trajectory
  $Q, R$: LQR cost matrices
Repeat until convergence:
  1. For $t = 0, \ldots, H$, compute dynamics models
    • For $i = 1, 2$, compute expected covariances:
        $\Sigma_i(D_t) \leftarrow \mathbf{E}_{\delta s_t \sim \mathcal{N}(0,\Gamma_t)}\left[\Sigma_i(s^\star_t + \delta s_t, u^\star_t + K_t \delta s_t)\right].$
    • Compute averaged model:
        $\bar{\Sigma}_t \leftarrow \left(\Sigma_1^{-1}(D_t) + \Sigma_2^{-1}(D_t)\right)^{-1},\ \bar{f} \leftarrow \bar{\Sigma}_t\left(\Sigma_1^{-1}(D_t) f_1 + \Sigma_2^{-1}(D_t) f_2\right).$
    • Linearize dynamics of averaged model:
        $A_t \leftarrow \frac{\partial \bar{f}(s^\star_t, u^\star_t)}{\partial s^\star_t},\ B_t \leftarrow \frac{\partial \bar{f}(s^\star_t, u^\star_t)}{\partial u^\star_t}.$
  2. Run LQR to find optimal controllers $K_t$:
        $K_{0:H-1} \leftarrow \mathrm{LQR}(A_{0:H-1}, B_{0:H-1}, Q, R).$
  3. For $t = 0, \ldots, H-1$, update state distributions:
        $\Gamma_{t+1} \leftarrow (A_t + B_t K_t)\Gamma_t (A_t + B_t K_t)^T + \bar{\Sigma}_t$
```

To allow our model to be as general as possible, we assume a very simple probabilistic description of how the states evolve when executing these fixed sequences of actions. In particular, we assume that the error dynamics evolve according to

$$\delta s_{t+1} = \rho \delta s_t + w_t$$

where $\rho \in \mathbb{R}$ (typically ≈ 1) indicates the stability of taking such trajectory actions, and where w_t is a zero-mean Gaussian noise term with covariance $\Sigma(\delta s_t, \delta u_t)$, which depends only on the state and control error and which captures the covariance of the model as a function of how far away we are from the desired trajectory. This model captures the following intuition: if we are close to a previously observed trajectory, then we expect the next error state to be similar, with a variance that increases the more we deviate from the previously observed states and actions. While it may seem odd to suppose that the next predicted mean state is independent of the control, a single trajectory couldn't convey any additional information: we only know how the system responded to a particular control input $u_t^\star$ for a particular state, $s_t^\star$, and we have no information about how the system would have responded had we acted otherwise.

To see how this model naturally leads to trading off between actively controlling the system and largely following known trajectory controls, we consider a situation where we run the Multi-model LQR algorithm with an inaccurate model[1] M_1 and this described trajectory model, M_2. Consider a situation where our state distribution D_t is tightly peaked around the desired state, but where M_1 predicts the next state very poorly. In this case, the maximum likelihood covariance $\Sigma_1(D_t)$ will be large, because M_1 predicts poorly, but $\Sigma_2(D_t)$ will be much smaller, since we are still close to the desired trajectory. Therefore $\bar{f}$ will be extremely close to the trajectory model M_2, and so will lead to system matrices $A_t \approx \rho I$ and $B_t \approx 0$ (since these are the derivatives of the trajectory model). Therefore, since no delta can affect the state very much, LQR will choose controls $\delta u_t \approx 0$ — i.e., execute controls very close to $u_t^\star$, as this will minimize the expected cost function. In contrast, if we are far away from the desired trajectory, or if the model M_1 is very accurate, then we would expect to largely follow this model, as it would have lower variance than the naive trajectory model. In practice, the system will smoothly interpolate between these two extremes based on the variances of each model.

5.2.3 Estimating Variances

Until now, we have assumed that the covariance terms $\Sigma_i(s_t, u_t)$ have been given, though in practice these will need to be estimated from data. Recall that the variances we want to learn are ML estimates of the form

$$\Sigma_i(s_t, u_t) = E\left[(s_{t+1} - f_i(s_t, u_t))(s_{t+1} - f_i(s_t, u_t))^T\right]$$

[1]For the purposes of this discussion it does not matter how M_1 is obtained, and could be built, for example, from first principles, or learned from data, as we do the car in Section 5.3.

so we could use any number of estimation methods to learn these covariances, and our algorithm is intentionally agnostic about how these covariances are obtained. Some modeling algorithms, such as Gaussian Processes, naturally provide such estimates, but for other approaches we need to use additional methods to estimate the covariances. Since we employ different methods for estimating the variance in the two sets of experiments below, we defer a description of the methods to these sections below.

5.3 Experiments

In this section we present experimental results, first on a simple simulated cart-pole domain, and then on our main application task for the algorithm, the task of autonomous sideways sliding into a parking spot.

5.3.1 Cart-pole Task

To easily convey the intuition of our algorithm, and also to provide a publicly available implementation, we evaluated our algorithm on a simple cart-pole task (where the pole is allowed to freely swing entirely around the cart). The cart-pole is a well-studied dynamical system, and in particular we consider the control task of balancing an upright cart-pole system, swinging the pole around a complete revolution while moving the cart, and then balancing upright once more. Although the intuition behind this example is simple, there are indeed quite a few implementation details, and the source code for this example is available at: `http://ai.stanford.edu/~kolter/icra10`.

The basic idea of the cart-pole task for our setting is as follows. First, we provide our algorithm with an inaccurate model of the dynamical system; the model uses a linear function of state features and control to predict the next state, but the features are insufficient to fully predict the cart-pole dynamics. As a result the model is fairly accurate in the upright cart-pole regions, but much less accurate during the swing phase. In addition, we provide our algorithm with a single desired trajectory (the target states and a sequence of controls that will realize this trajectory under a zero-noise system). Because we are here focused on evaluating the algorithm without

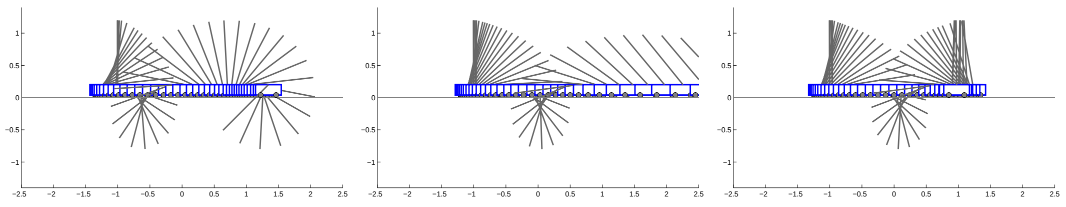


Figure 5.1: (left) Trajectory followed by the cart-pole under open-loop control (middle) LQR control with an inaccurate model and (right) Multi-model LQR with both the open loop trajectory and inaccurate model.

Method	**Avg. Total Cost**
LQR (true model)	18.03 ± 1.61
LQR (inaccurate model)	96,191 ± 21,327
LQR (GP model)	96,712 ± 21,315
Open Loop	67,664 ± 13,585
Multi-model LQR	33.51 ± 4.38
Hand-tuned Switching Control	73.91 ± 13.74

Table 5.1: Total average cost at the last time step for different algorithms on the cart-pole task.

consideration for the method used to estimate the variances, for this domain we estimate the variances exactly using the true simulation model and sampling.

Figure 5.1 shows the system performance using three different methods: 1) fully "open loop" control, which just replays the entire sequence of controls in the desired trajectory, 2) running LQR using only the inaccurate model, and 3) using the Multi-model LQR algorithm with both the inaccurate model and the trajectory. As can be seen, both pure open-loop and LQR with the inaccurate model fail to control the system, but Multi-model LQR is able to reliably accomplish the control task. Figure 5.2 and Table 5.1 similarly show the total cost achieved by each of the methods, averaged over 100 runs. Although open loop and inaccurate LQR fail at different points, they are never able to successfully swing and balance the pole, while Multi-model LQR performs comparably to LQR using the actual model of the system dynamics by naturally interpolating between the two methods in order to best control the system. In the figure we also show the error of a hand-tuned switching policy that switches between purely model-based and open-loop control; despite exhaustive search to find

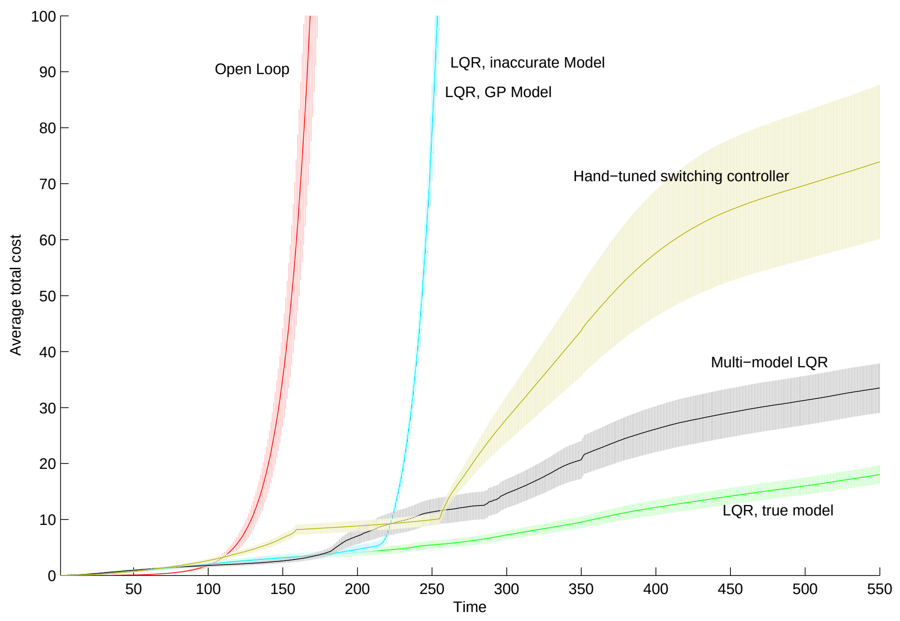


Figure 5.2: Average total cost on cart-pole task, versus time. Shaded regions indicate 95% confidence intervals.

the optimal switch points, the method still performs worse here than Multi-model LQR.

In Figure 5.2, we also compare to a Gaussian process (GP) model (see, e.g. (Rasmussen and Williams, 2006) for detailed information about Gaussian processes) that attempted to learn a better dynamics model by treating the inaccurate model as the "prior" and updating this model using state transitions from the desired trajectory. However, the resulting model performs no better than LQR with the inaccurate model. We emphasize that we are *not* suggesting that Gaussian processes cannot model this dynamical system — they can indeed do so quite easily given the proper training data. Rather, this shows that the desired trajectory alone is not sufficient to improve a GP model, whereas the Multi-model LQR algorithm can perform well using only these two inputs; this is an intuitive result, since observing only a single trajectory says very little about the state and control *derivatives* along the trajectory, which are ultimately what is needed for good fully LQR-based control. Indeed, we have been unable to develop any other controller based only on the inaccurate model and the desired trajectory that performs as well as Multi-model LQR.

5.3.2 Extreme Autonomous Driving

Finally, in this section we present our chief applied result of the chapter, an application of the algorithm to the task of extreme autonomous driving: accurately sliding a car sideways into a narrow parking space. Concurrent to this work, we have spent a great deal of time developing an LQR-based controller for "normal" driving on the car, based on a non-linear model of the car learned entirely from data; this controller is capable of robust, accurate forward driving at speeds up to 70 mph, and in reverse at speeds up to 30 mph. However, despite significant effort we were unable to successfully apply the fully LQR-based approach to the task of autonomous sliding, which was one of the main motivations for this current work.

Briefly, our experimental process for the car sliding task was as follows. We provided to the system two elements; first we learned a driving model for "normal" driving, learned with linear regression and feature selection, built using about 2 minutes of driving data. In addition, we provided the algorithm an example of a human driver executing a sideways sliding maneuver; the human driver was making no attempt to place the car accurately, but rather simply putting the car into an extreme slide and seeing where it wound up. This demonstration was then treated as the target trajectory, and the goal of the various algorithms was to accurately follow the same trajectory, with cones placed on the ground to mark the supposed location of nearby cars.

For this domain, we learned domain parameters needed by the Multi-model LQR algorithm (in particular, the covariance terms for the inaccurate and open-loop models, plus the ρ term) from data of a few executions of the sliding maneuver on the real system. To reduce the complexity of learning the model variances, we estimated the covariance terms as follows: for the inaccurate model we estimated a time-varying (but state and control independent) estimate of the variance by computing the error of the model's predictions for each point along the trajectory, i.e.,

$$e_t = (s_{t+1} - f_1(s_t, u_t))(s_{t+1} - f_1(s_t, u_t))^T$$

then averaged these (rank-one) matrices over a small time window to compute the covariance for time t. It is possible to let this term depend on the time index alone because, since we are executed a time-indexed trajectory, the time index itself also implies the state and control that the system should be in; thus, by looking at how well the inaccurate model predicts the trajectory, we can develop a good estimate of the model's variance *along this trajectory* without attempting to estimate its variance over the entire space (a challenging task, as much of the space is never experienced by the system, and the model's variance would therefore be difficult to model). For the open-loop trajectory model, we employ the opposite approach, and learn a state and control dependent (but time independent) estimate of the covariance of the form $\Sigma_2(s_t, u_t) = (w_1\|\delta u_t\|^2 + w_2\|\delta s_t\|^2 + w_3)\, I$, where we learned the parameters $w_1, w_2, w_3 > 0$ via least-squares by executing the open-loop maneuver one again from a slightly different starting point, and observing how the two trajectories diverged; this model captures the intuition that the variance of the open-loop model increases for points that are farther from the desired trajectory, but that this divergence does not depend on where we are along the trajectory (i.e., we assume that the true stochasticity of the world is constant along the trajectory). Finally we selected $\rho = 1$ due to the fact that the system rarely demonstrates extremely unstable behavior, even during the slide.

Figure 5.3 shows snapshots of the car attempting to execute the maneuver under the three methods of control: open-loop, pure LQR, and our Multi-model LQR approach integrating both the inaccurate model and the trajectory. Videos of the different slides are available at: `http://ai.stanford.edu/~kolter/icra10car`. It is easy to understand why each of the methods perform as they did. Purely open-loop control actually does perform a reasonable slide, but since it takes some distance for the car to build up enough speed to slide, the trajectory diverges significantly from the desired trajectory during this time, and the slide slams into the cones. Pure LQR

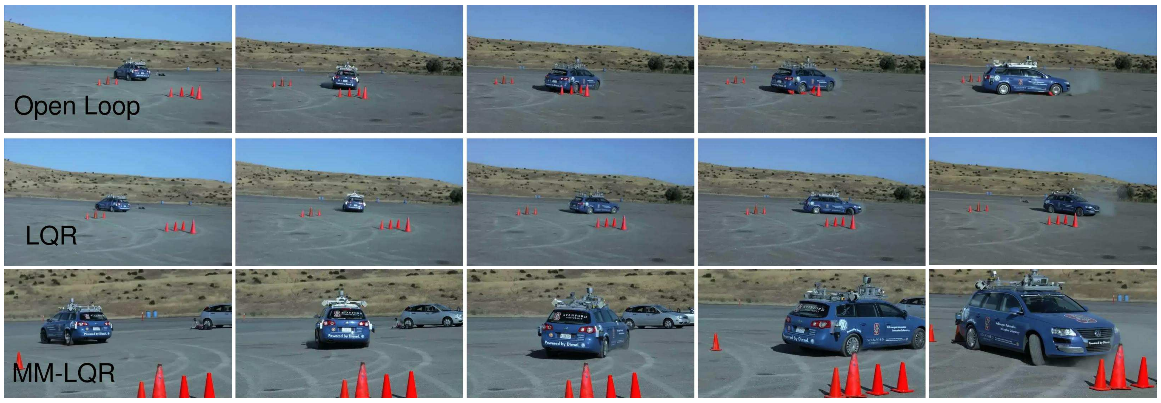


Figure 5.3: Snapshots of the car attempting to slide into the parking spot using (top) open-loop control, (middle) pure LQR control, and (bottom) Multi-model LQR control. The desired trajectory in all cases is to slide in between the cones.

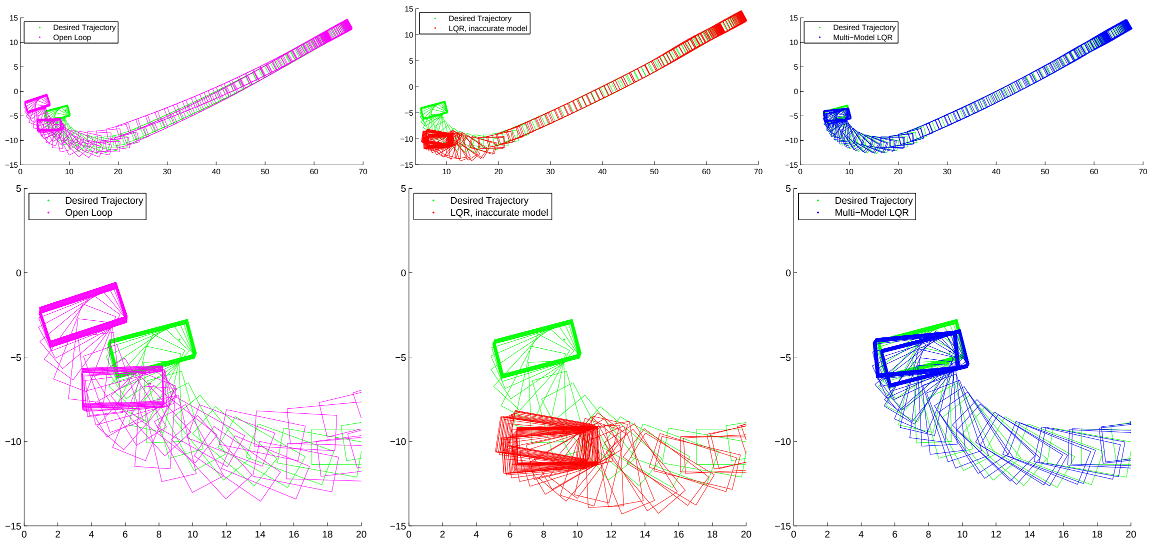


Figure 5.4: Plots of the desired and actual trajectories followed by the car under (left) open-loop control, (middle) pure LQR control, and (bottom) Multi-model LQR control. Bottom plots show a zoomed-in view of the final location.

control, on the other hand, is able to accurately track the trajectory during the backward driving phase, but is hopeless when the car begins sliding: in this regime, the LQR model is extremely poor, to the point that the car executes completely different behavior while trying to "correct" small errors during the slide. In contrast, the Multi-model LQR algorithm is able to capture the best features of both approaches, resulting in an algorithm that can accurately slide the car into the desired location. In particular, while the car operates in the "normal" driving regime, Multi-model LQR is able to use its simple dynamics model to accurately control the car along the trajectory, even in the presence of slight stochasticity or a poor initial state. However, when the car transitions to the sliding regime, the algorithm realizes that the simple dynamics model is no longer accurate, and since it is still very close to the target trajectory, it largely executes the open-loop slides controls, thereby accurately following the desired slide. Figure 5.4 shows another visualization of the car for the different methods. As this figure emphasizes, the Multi-model LQR algorithm is both accurate and repeatable on this task: in the trajectories shown in the figure, the final car location is about two feet from its desired location.

5.4 Summary

This chapter presented a method for combining inaccurate models of the system with trajectories, previously observed demonstrations of certain behaviors, in order to obtain better performance than what is possible by using either element in isolation. In particular, we present an algorithm that uses a variance-based criterion to determine how to properly weight the different models, which has the effect of executing largely closed-loop controls when the system is in a well-modeled states, but open-loop controls when the system is in a state that is poorly modeled. We evaluate the approach on a cart-pole swinging task, and on the challenging task of powersliding a full-sized autonomous car into a parking spot. Our performance on this latter task represents the state of the art in terms of accurately placement of an autonomous full-sized car in this type of maneuver.

The major drawback of our proposed approach is that in order to achieve such performance without an accurate model of the system, we do require that we have previously observed behavior that accomplishes the desired goal for the control task. For the car, for instance, we require that we have previously seen a sliding maneuver (though not necessarily one executed in a precise manner). In the event that the system enters a state where there is no accurate model, and if the algorithm does *not* have a demonstration of any behavior that would perform well when executed open-loop, then the variance both for any models *and* for all trajectories would be very large, resulting in behavior that would almost certainly not accomplish the desired task. More broadly speaking, our algorithm will not plan entirely new trajectories in difficult-to-model regions, it will simply use trajectories is has already seen in conjunction with the models that are accurate only in certain parts of the state space. Despite this restriction, the algorithm can achieve impressive performance when it is given such trajectories at its disposal.

Chapter 6

Application to the LittleDog Robot

One of the primary application domains in the previous chapters has been the LittleDog robot, a quadruped robot built to traverse challenging terrain. While the algorithms we have discussed were crucial to their respective tasks on the LittleDog (learning dynamic maneuvers and trotting), there are naturally many other components to the system that were crucial to the overall performance of the robot. This chapter provides a general overview of our work on the LittleDog robot, highlighting several additional elements that were needed to achieve good performance, and demonstrating the complete system crossing a wide variety of terrains. As such, the presentation in this chapter differs from the previous three chapters. While the overall notion of inaccurate modeling is certainly a prevailing theme in the LittleDog robot (indeed, the difficulty of creating an accurate physical simulator of the robot has affected virtually all our design choices on this platform), in this chapter we are not focusing on a particular algorithm based upon the use of inaccurate modeling. Rather, the material in this chapter can be viewed as a case study in developing a full control architecture for a system where fully accurate modeling is not feasible. We will particularly highlight areas where the inaccuracy of our models has influenced our design decisions.

6.1 Introduction to Legged Locomotion

Legged robots offer a simple promise: the potential to navigate areas inaccessible to their wheeled counterparts. While wheeled vehicles may excel in regards to speed and fuel efficiency, they are only able to access about one half of the earth's land mass (Raibert, 1986). Contrast this with legged animals, which are able to access virtually 100% of the earth's land surface. Thus, to enable robotic applications in a variety of rugged terrains, such as search and rescue operations in hard-to-access locations, legged locomotion is a promising approach. However, while there has been a great deal of work on legged locomotion, current legged robots still lag far behind their biological cousins in terms of the ability to navigate challenging, previously unseen terrain.

In this chapter we specifically consider the task of navigating a quadruped robot over a variety of challenging terrain, including terrain that the robot has not previously seen until execution time. At run-time, we assume that the robot obtains a model of the terrain to cross, and that we have very accurate localization during execution; thus, the problem we are focusing on is the *planning* and *control* tasks of quadruped locomotion in highly irregular terrain.

The central methodological theme in this chapter is the ubiquitous use of *rapid recovery and replanning methods*. Often times robotic control tasks are framed as lengthy search problems, where we use a model of the system to explore possible control sequences to move the system from its initial state to the goal; probabilistic road maps (Kavraki et al., 1996) and rapidly-exploring random trees (LaValle and Kuffner, 1999) are classic examples of such methods. However, much like the control methods we have discussed previously, these algorithms are best applied when we have an accurate model of the system; if the model is not accurate, then the robot will likely deviate from the plan as it executes its motion, and due to the computationally intensive planning process, the system cannot quickly replan a new trajectory. Instead, we designed our system to allow for some degree of failure in the robot's execution, and focus on extremely fast replanning methods (within one or two control cycles) that would allow the robot to continue even after most failures. This strategy mitigates the harm that inaccurate models may cause in the planning process, and as we will show, we are able to use very inaccurate models to successfully and reliably navigate over challenging terrain.

6.2 Related Work

6.2.1 Background on legged locomotion

Research in legged robotics has a long history, dating back to the beginning of modern robotics. One of the first major achievements in legged robots was a human-operated robot developed by GE in the 60s (Mosher, 1968). Despite the fact that the robot relied on a human driver, it was capable of walking, climbing over small obstacles, and pushing large obstacles such as trucks.

The mathematical study of autonomous legged robot gaits began short thereafter with the work of McGhee and others (McGhee, 1967; McGhee and Frank, 1968; McGhee, 1968). This work focused mainly on what is known as *static* gaits, gaits where the robot maintains *static stability*, which involves keeping its center of gravity (COG) within the support triangle formed by the non-moving feet; thus, for a quadruped, a static gait implies that only one leg can be moved at a time, with the remaining three resting on the ground. McGhee implemented many of these principles in a six-legged walking machine that was able to navigate over a nominal amount of roughness in the terrain (McGhee, 1985). Another milestone in the development of legged robots came with a series of robots (known as the TITAN robots), developed by Hirose et al. (1984), that could walk up stairs, again using a static gait.

Another vein of research that began at this time was the work by Raibert on balancing robots (Raibert, 1986). Unlike the static gaits described previously, these robots were capable of *dynamic* gaits, where fewer than three legs were on the ground at a time, including the extreme case of a single hopping leg, though at the time these robots only operated on flat ground. Recent work in this vein includes the KOLT robot (Nichol et al., 2004), the Scout II robot (Poulakakis et al., 2005), and the BigDog robot (Raibert et al., 2008); some of these robots are now able to navigate over ground with significant irregularities. However, the obstacles that these robots are able to navigate over are still small (relative to the size of the robot) compared to what we consider in this work.

Returning to static gait legged robots, a number of system have been built in recent years. The Ambler (Krotkov et al., 1995) and Dante II (Bares and Wettergreen, 1999) robots were large-scale legged robots built for testing planetary exploration techniques and for volcanic exploration respectively. Although sometimes partially operated by humans, these robots had modes of fully autonomous behavior in rugged environments, and represented another significant improvement in the state of the art for the time.

The Sony Aibo robot, and in particular the RoboCup competitions (which involve playing soccer games with teams of small quadruped robots), have spawned a great deal of work on learning fast and efficient gaits (Hengst et al., 2002; Hornby et al., 2005; Kohl and Stone, 2004), as well as more high-level work on robot cooperation and strategy. Many of these systems explicitly focused on learning techniques to improve the performance of the robot. However, these works have focused mainly on locomotion over flat ground, due to both the physical capabilities of the Aibo and to the fact that the RoboCup involved moving only over flat ground.

Another branch of research in quadruped locomotion has focused on so-called "Central Pattern Generators" or CPGs. This work is inspired by biological evidence that animals regulate locomotion by relatively simple, reflexive systems located in the spinal chord rather than the brain (Grillner, 1985). CPGs are neural-like control laws that produce period behavior to drive leg motion, a technique that can be particularly well suited to robots with built-in compliant mechanisms. A number of robots have been built using such behavior, including some which can adapt to some degree of roughness and irregularity in the terrain (Fukuoka et al., 2003; Kimura et al., 2007). However, again these robots typically can only handle a relatively small amount of irregularity in the terrain, not the large obstacles that we consider.

Finally, as we mentioned briefly in the introduction, a separate thread of research in legged locomotion has been to develop mechanical systems that are naturally much more robust to irregular terrain, or specifically suited to a certain type of challenging

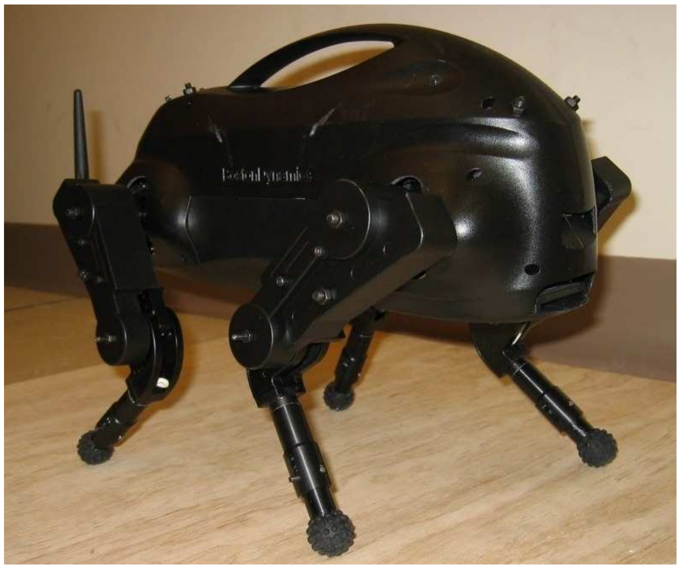


Figure 6.1: The LittleDog robot, designed and built by Boston Dynamics, Inc.

locomotion. Such work includes the Rhex hexapedal robot (Saranli et al., 2001), which uses six flexible legs to rapidly move over relatively large obstacles, and the RiSE climbing robot (Saunders et al., 2006), which uses special leg and foot design to scale vertical surfaces. These robots can achieve very impressive locomotion, but their design strategy is roughly orthogonal to our work: these robots demonstrate that clever mechanical design, even with largely open-loop behavior, can achieve good performance in a variety of scenarios. However, these robots can fail in scenarios where careful sensing and foot placement are a necessity, which is the focus of our work.

6.2.2 The Learning Locomotion program and LittleDog robot

The robotic platform for the DARPA Learning Locomotion program was the LittleDog robot, shown in Figure 6.1, a small quadruped designed and built by Boston Dynamics, Inc. While the robot underwent a sequence of small upgrades during the program, at all points each team working in the program had access to the same hardware system, so that the capabilities of the different systems resulted entirely from the different software developed by the different teams.

The LittleDog robot is a small (roughly 3kg) robot, with a 40cm long body and 15 cm legs. Each leg has three degrees of freedom, two hip joints and a knee joint, that together allow for free placement in 3D space, subject to the kinematic limits of the joints. The hip motors provide 1.47 Nm of torque, while the knee provides 1.02 Nm, enough power to move the robot relatively quickly in static gaits, but not enough power to truly jump off the ground; due to this constraint, the majority of the program focused either on static gaits (keeping three feet on the ground at one time), or limited dynamic gaits, such as a trot and two-legged jumps, rather than the more dynamic "jumping" gaits exhibited, for example, by the BigDog robot (Raibert et al., 2008).

Control for the robot consisted of two control cycles. An internal processor on the robot runs a simple PD controller at 500 hz, applying torques to achieve specified joint angles. Meanwhile, a separate workstation computer runs a control loop at 100 hz, wirelessly sending commands to the robot. While the workstation may send either PD angle setpoints (which are then executed by the aforementioned on-board PD controller) or actual motor torques, all teams that we are aware of primarily used the PD setpoint interface, as the higher bandwidth of the on-board control loop then allowed for quicker feedback control of the robot.

Although the robot has some degree of onboard sensing (an internal IMU, foot force sensors, and an IR proximity sensor) most of the perception for the Learning Locomotion project was performed offboard. The robot operates in a Vicon motion capture system, with retro-reflective markers on the dog's body and terrain. This gives the system real-time estimates of the robot's pose in relation to the terrain. Together with scanned 3D models of the terrain and joint encoder readings, this system provides a very accurate estimate of the robot's state, even without any advanced filtering techniques. Such "ideal" perception is an admitted benefit for these robots, which would not be present to such a high degree in a real quadruped walking outdoors, but the goal of the Learning Locomotion program was to focus on the planning and control elements of locomotion, leaving the challenging perception task to other work. However, in addition and separate from the official government tests of the program, we have conducted extensive experiments using only an onboard stereo camera

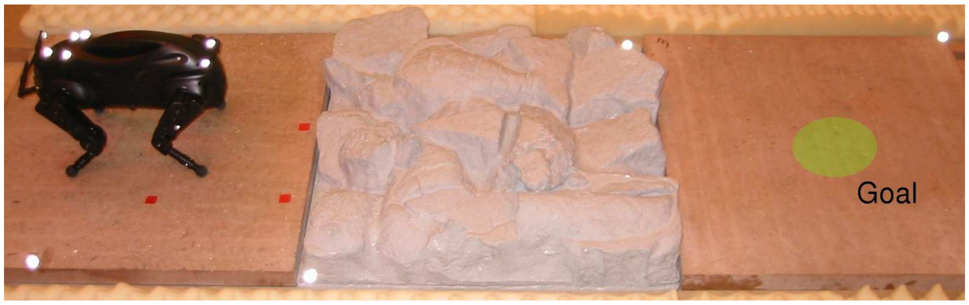


Figure 6.2: Typical quadruped locomotion task: navigate robot over terrain to goal.

for vision. We will present these results briefly in a later section, but they suggest that many of the techniques developed for this work by both ourselves and other teams are indeed applicable to realistic situations where the robot has only onboard sensing.

6.3 A software architecture for quadruped locomotion

Our goal for quadruped locomotion is to execute a sequence of joint torques (or, if using the PD controller, desired joint angles) that move the robot from its initial location to the goal. A typical setup of the robot and terrain is pictured in Figure 6.2. Given the complex obstacle shapes and high dimensionality of the state space (36-D, including position and velocity variables), naively discretizing the state space would not scale. Instead, we develop a hierarchical control approach, which decomposes the problem into several layers.

Figure 6.3 shown the overall architecture of our system. We will spend the rest of this section describing the different components of this system in detail, but from a high level the process is fairly straightforward. Before execution, the system first plans an approximate route over terrain. Next, in an online manner, the system selects which type of gait to use: a static walk, a trot, or a specialized maneuver policy. For the static walk and trot gaits, we then plan footsteps for the robot, plan motion paths for the body and feet, and execute these plans using closed-loop control.

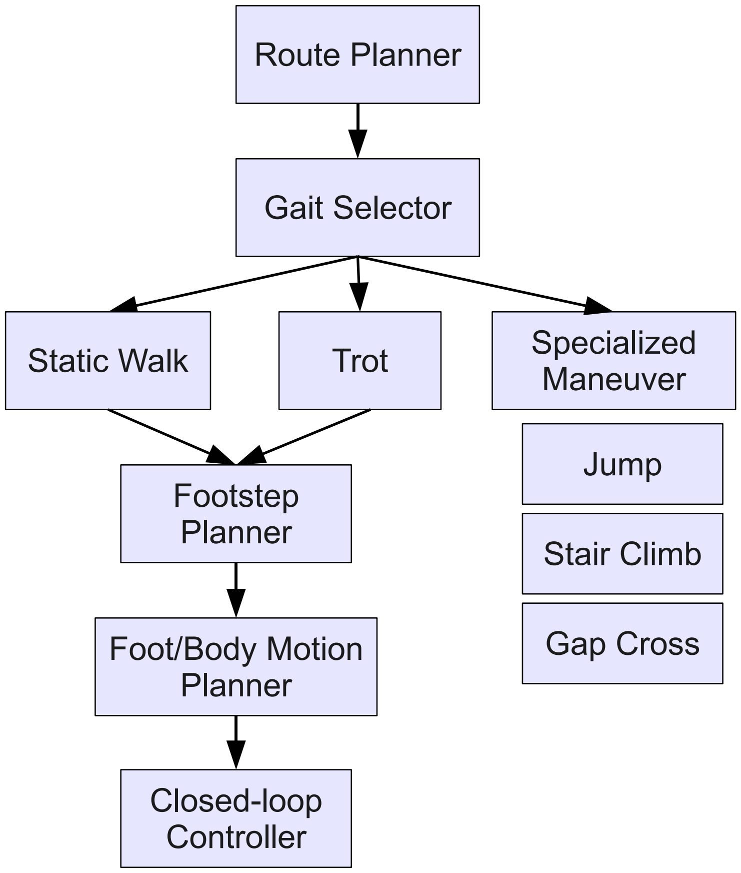


Figure 6.3: Hierarchical software architecture for quadruped planner and controller.

6.3.1 Route Planner

The goal of the route planner is to plan an approximate path for the robot's body across the terrain. The actual path followed by the robot will of course deviate from this planned path (for example, when using a static walking gait, the robot's actual body path will be adjusted to maintain stability of the robot), but this initial route allows the robot to better focus its search for footsteps or maneuvers.

A overview of the route planning process is shown in Figure 6.4. The method uses a *route cost map* of the terrain that quantifies the goodness or badness of the robot's body center being at a specific location on the terrain. Given this body cost map, the route planner uses value iteration (using a finely discretized state, with ≈1

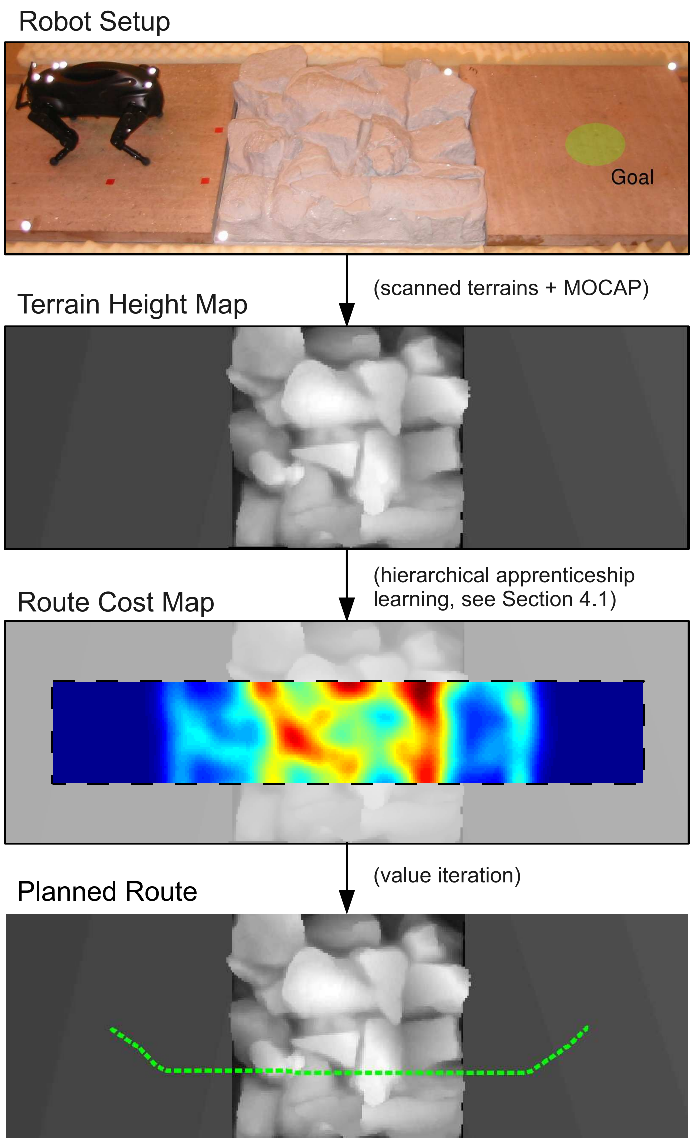

Robot Setup
Goal
(scanned terrains + MOCAP)
Terrain Height Map
(hierarchical apprenticeship learning, see Section 4.1)
Route Cost Map
(value iteration)
Planned Route

Figure 6.4: Overview of route planning element.

million states) to find a 2D path across the terrain. Since the state transitions are very sparse, we can solve this planning task using value iteration in an average of five seconds.[1] An advantage of using value iteration for this task, which relates to

[1]Here and in the remainder of the chapter, all run times correspond to run times on the government-provided host machine, running an Intel Xeon processor at 3.0 Ghz

the aforementioned “rapid replanning” philosophy, is that we can compute the value function once, prior to execution, and subsequent replanning then is extremely fast; if, during execution, the robot falls significantly off its desired path, we simply replan a new route using this value function, and continue execution. We also note that the value iteration is the only part of the planning process that is run offline: all subsequent elements are run as the robot is crossing the terrain, and thus execution time is of paramount importance.

The efficacy of this approach rests chiefly on the quality of the cost map: since we are not checking kinematic feasibility along the path, the process has the potential to lead to a “dead end” — in contrast, global footstep planning techniques (e.g., (Chestnutt et al., 2003)) can guarantee a full solution at the footstep level, but are correspondingly more computationally expensive. However, we have found that for reasonable cost maps, the resulting routes work very well in practice, guiding the robot over the most reasonable portions of the terrain, thus speeding up planning without significant negative effects. Of course, manually specifying such a cost function is very difficult in practice, so instead we rely on a learning approach to learn an approximate cost map as a linear function of certain *features* that describe the center location; which we will discuss at greater length in Section 6.6.

6.3.2 Gait Selector

The next step in the execution process is to select, before each step, a gait to use for the current situation, either a slow static walk (for highly irregular terrain), a faster dynamic trot (for relatively flatter terrain), or a maneuver specialized to a specific class of obstacles. Although preliminary experiments indicate that the best gait type can be accurately predicted using a learning algorithm (where the input is a local height map of terrain near the robot, and the output is the type of gait or maneuver to execute), we did not use such an approach for the official Learning Locomotion submission. Because the project stipulated that our software was provided, a priori, with terrain IDs that indicated the terrain type, we were able to devise a fairly small set of rules based upon the terrain ID and height differential of the robot’s legs that were very accurate in predicting the best gait type.

6.3.3 Static Walking Gait

For highly irregular terrain, the robot typically uses a static walking gait, a relatively slow mode of locomotion that moves one leg at a time and maintains static stability by keeping the robot's center of gravity (COG) within the supporting triangle formed by the remaining feet. However, even given the desired route from the higher levels of the system, robustly executing a static walking gait remains a challenging task, and so we once again decompose the problem into multiple levels: first we plan an upcoming footstep for the robot; we then plan trajectories for the moving foot and body that move the desired foot while maintaining static stability of the robot; finally, we execute this plan on the robot using a controller that constantly checks for loss of stability, and recovers/replans when necessary.

Walking Gait Footstep Planner

To plan footsteps for the static walk, we use a common fixed repeating sequence of leg movements: back-right, front-right, back-left, front-left. In addition to being the gait typically by biological animals during static walking, it has the benefit of maximizing the kinematically feasible distance that each foot can move — for more detailed discussion of the benefits of this pattern, see (McGhee and Frank, 1968).

The planning process is as follows: to take a step, the system first averages the current positions of the four feet to determine the "center" of the robot — in quotations because of course the true COG need not be located at this point; this center is merely used for planning purposes. Then, we find the point some distance d_{body} ahead of this center in the route line, and find the "home" position of the moving foot relative to this new point (the "home" position here just denotes a fixed position for each foot relative to the robot's center, in a box pattern). Finally, because this precise location may coincide with a poor location on the terrain, we search in a box around this location and find the footstep with the lowest cost. Again, the cost here quantifies the goodness of the terrain, this time at the individual footstep level, and

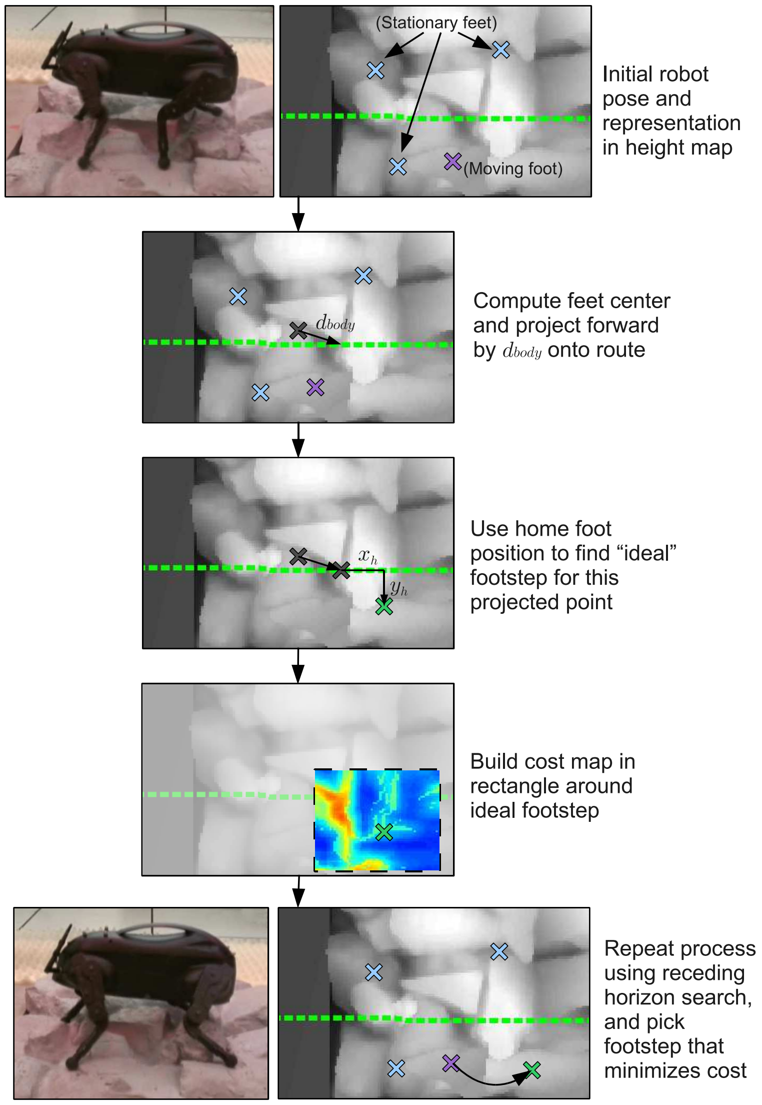

(Stationary feet)
(Moving foot)
Initial robot pose and representation in height map
d_{body}
Compute feet center and project forward by d_{body} onto route
x_h
y_h
Use home foot position to find “ideal” footstep for this projected point
Build cost map in rectangle around ideal footstep
Repeat process using receding horizon search, and pick footstep that minimizes cost

Figure 6.5: Overview of the footstep planning process.

we will describe in Section 6.6 how we learn this footstep cost simultaneously with the body cost mentioned above. The complete planning system is illustrated in Figure 6.5.

Finally, because selecting only one footstep at a time can lead to overly myopic behavior, we use a branching search of some fixed horizon, then choose the footstep that leads to the least total cost To ensure that the search is tractable, we use a relatively small branching factor (typically 3 and 4), and perform a greedy search at each step to find the 3 or 4 best points in the search box that are not within 3 cm of each other. Because the process can still be somewhat computationally intensive, we plan the robot's next step in a separate thread while it executes its current step (assuming that the current step will be achieved). In cases where the step is not achieved and we need to quickly replan anew, then we use just a single step horizon in the search.

The advantage of this approach is that each foot movement can be planned independent of any previous footsteps: all that determines the next footstep is the moving foot and the mean of the four current footsteps. Despite it's simplicity, it is possible to show that this simple method will quickly converge to a symmetric pattern that staggers in the footsteps in an optimal manner. Because we check kinematic feasibility and collisions only at the beginning and end of each proposed footstep, there is some chance that no kinematically feasible, non-colliding path exists between these positions, though as before we find that this is rarely the case in practice.

Walking Gait Foot/Body Motion Planner

The foot/body motion planner is tasked with determining a sequence of joint angles that achieve the planned footstep while maintaining static stability of the robot. Motion planning in such high dimensional state spaces is typically a challenging and computationally intensive problem in robotics. However, following our theme of fast replanning necessitates a very quick method for planning such motions; if the robot deviates from its desired trajectory and we need to replan its motions, then we do not want to wait for a slow planner to finish executing before we begin moving again. Thus, as part of our system we developed a novel motion planning approach, based on convex optimization and cubic splines, that enables us to quickly (in a matter of milliseconds, on the same time scale as the control loop) plan motions that satisfy the above constraints. However, we defer the discussion of this method until Section 6.5.

Walking Gait Controller

Finally, after planning the precise joint motions that will move the foot to its desired location while maintaining static stability, the final task of the walking gait is to execute these motions on the robot. Due to the challenging nature of the terrains we consider, using PD control alone is highly unreliable: regardless of how well we plan, and regardless of how well the individual joints track their desired trajectories, it is almost inevitable that at some point the robot will slip slightly and deviate from its desired trajectory. Therefore, a critical element of our system is a set of closed-loop control mechanisms that detect failures and either stabilize the robot along its desired trajectory or re-plan entirely. In particular, we found three elements to be especially crucial for the walking gait: 1) stability detection and recovery, 2) body stabilization, and 3) closed-loop foot placement. Again, to simplify the discussion here we defer discussion of these elements until Section 6.4.

6.3.4 Trot Gait

When the terrain is flatter, a static walking gait is unnecessary, and the robot can move much faster using a dynamic trotting gait, which moves two feet at a time. Our approach to a trot gait differs from many other approaches in that we do not explicitly try to maintain even some measure of dynamic stability, but merely learn a gait that tends to balance well "on average." As with the walking gait, the trot gait is also subdivided hierarchically into a footstep planner, a joint planner, and a controller, but due to the nature of the trot (specifically the fact that we will only execute the gait on relatively flat ground), these elements are considerably simpler than for the walking gait, so we describe them only briefly.

The trot footstep planner operates in the same manner as the walking footstep planner, with the exception that we are now planning two moving feet at a time rather than one. We again use a fixed alternating foot pattern, this time alternating moving back-right/front-left and back-left/front-right. To plan footsteps we again use the current foot locations to determine the effective center of the robot, then find the point some distance d_{body} ahead on the route, find the "home" position of the moving feet relative to this new point, and search in a box around these desired positions to find the foot location with lowest cost. Further differences from the walking footstep planner is that 1) we use a smaller d_{body}, to take smaller steps, 2) we don't search in as large a box as for the walking gait and 3) we don't use any receding horizon search, but simply use the minimal greedy one-step costs. As with the walking gait, this process will quickly converge to a symmetric trot pattern, regardless of the initial configuration of the feet.

The joint planner and controller for the trot is similarly simplified from the static walk. We move the moving feet in ellipses over the ground, and use inverse kinematics to determine the joint angles that move the feet along their desired location, while linearly interpolating the body position between the center locations for the different footsteps. The controller in this case uses PD control alone; indeed, we have found that closed-loop mechanisms actually degrade performance here, since the trot is quick enough such that the natural periodic motion will tend to stabilize the robot and interrupting this periodic motion with additional closed-loop mechanisms typically causes more harm than good. The one challenging aspect to designing the trot controller is to position the COG in a manner that maintains balance as much as possible; this is the task we discussed at length in Chapter 4, so we will not discuss it further here.

6.3.5 Specialized Maneuvers

Finally, in addition to the static walk and dynamic trot, we have developed a number of specialized maneuvers, each intended to cross some specific class of obstacles. While these were admittedly specialized to the types of obstacles prescribed by the Learning Locomotion project (for instance, steps or gaps), they are general enough to cross a wide variety of obstacles within their intended class. We use a total of four specialized maneuvers for the Learning Locomotion terrains: a front leg jump (for quickly jumping both front legs onto a step or over a gap), a stair climb (for climbing the back legs onto a stair), a gap cross (for sliding the back legs over a gap), and a barrier cross (for bringing the back legs over a barrier).

The challenge for the dynamic maneuvers, such as the front jump, is devising a policy that properly executes the maneuvers without falling forward or backward; we learned the dynamic maneuvers using the Policy Search with Signed Derivative algorithm, as described in Chapter 3, so again we don't discuss these further here. The non-dynamic specialized maneuvers, such as the stair climb or gap cross, were manually designed to cross their respective obstacles while maintaining static stability of the robot, usually by using the robot's body itself to balance. While an important part of our overall system, these maneuvers were fairly easy to hand-tune, as the static motion made them less sensitive to parameter choice, and thus we also don't discuss them further.

6.3.6 Introduction to Rapid Replanning Methods

This section has described our software system for the quadruped robot, but we deferred discussion of several novel methods, related to our central theme of rapid replanning, that we have developed over the course of this program. In the remainder of this chapter we will present each of these methodologies. In particular, we present and analyze three approaches that we have developed over the course of the program: 1) a method for recovery and stabilization at the control level, 2) a cubic spline optimization approach to fast foot and body motion planning, and 3) a method for Hierarchical Apprenticeship Learning, used to learn the cost functions for route and footstep planning.

6.4 Recovery and Stabilization Control

As discussed in the previous section, even after planning a full trajectory for the foot or leg, it is undesirable to simply execute these motions open-loop. During the natural course of execution, the robot's feet may slip and possibly loose stability. Thus, we have implemented a number of low-level recovery and stabilization methods that continuously monitor the state of the robot and try to either maintain the current plan, or notify the system when it must replan entirely. We discuss three elements:

1) stability detection and recovery, 2) body stabilization, and 3) closed-loop foot placement.

6.4.1 Control Elements

Stability Detection and Recovery. Recall that (ignoring friction effects, which do not play a major role in stability for the LittleDog) the robot is statically stable only if the projection of the COG onto the ground plane lies within the triangle formed by the supporting feet. If the robot slips while following its trajectory, the COG can move outside the supporting triangle, causing the robot to tip over. To counteract this effect, we compute the current support triangle at each time step, based on the current locations of the feet as determined by state estimation. If the COG lies outside this triangle, then we re-run the walking gait foot and body motion planner. This has the effect of lowering all the robot's feet to the ground, then re-shifting the COG back into the inset support triangle.

Body Stabilization. While sometimes the recovery procedure is unavoidable, as much as possible we would like to ensure that the COG does *not* move outside the supporting triangle, even in light of minor slips. To accomplish this, we adjust the commanded positions of the supporting feet so as to direct the COG toward its desired trajectory. In particular, we multiply the commanded positions of the supporting feet by a transformation that will move the robot's COG from its current position and orientation to its desired position and orientation (assuming the supporting feet are fixed to the ground).

More formally, let T_{des} be the 4×4 homogeneous transformation matrix specifying the *desired* position and orientation of the robot relative to the world frame, and similarly let T_{cur} be the homogeneous transformation specifying the *current* position and orientation of the robot relative to the world frame. In addition, let $feet$ denote the default commanded positions of the supporting feet expressed in the robot's frame of reference, based on the desired trajectory for the COG. If we transform the commanded positions for the feet by

$$T_{des}^{-1} T_{cur} feet \tag{6.1}$$

then (assuming the supporting feet remain fixed) this would move the COG to its desired position and orientation. In practice, to avoid oscillations we apply the smoothed command

$$(1-\alpha)feet + \alpha T_{des}^{-1} T_{cur} feet \tag{6.2}$$

for some $0 < \alpha < 1$ (in our experiments we found $\alpha = 0.1$ to be a good value). This causes the robot's COG to move gradually to track the desired trajectory, even if the robot slips slightly. In addition, we project the desired position T_{des} into the current supporting triangle. During our development we found this approach to be more robust than attempting to move the supporting feet individually to stabilize the body, as our method keeps intact the relative positions of the supporting feet.

Closed-loop Foot Placement. Finally, we want to ensure that the moving foot tracks its desired trajectory as closely as possible, even if the body deviates from its desired path. To accomplish this, at each time step we compute the desired location of the foot along its (global) trajectory, and use inverse kinematics based on the *current* pose of the robot's body to find a set of joint angles that achieves the desired foot location. This is particularly important in cases where the robot slips downward. If the robot's body is below its desired position and we merely execute an open loop trajectory for the moving foot, then the foot can punch into the ground, knocking the robot over faster than we can stabilize it. Closed-loop foot tracking avoids this problem.

It may seem as if there are also cases where closed-loop foot placement could actually hinder the robot rather than help. For example, if the robot is falling, then it may be best to simply put its foot down, rather than attempt to keep its foot along the proper (global) trajectory. However, in our experience this nearly always occurs in situations where the recovery procedure mentioned previously will catch the robot anyway, so the closed-loop mechanism rarely affects the system negatively in practice.

6.4.2 Experimental Evaluation

To evaluate the three methods proposed above, we conducted experiments on different terrains of varying difficulty. Pictures of the terrains and their corresponding height

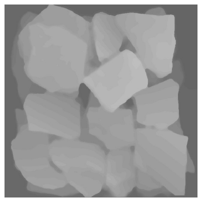
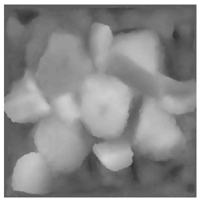
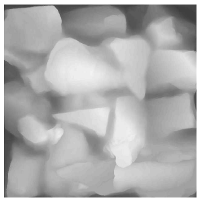
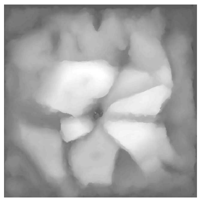
Terrain #	1	2	3	4
Max Height	6.4 cm	8.0 cm	10.5 cm	11.7 cm
Picture				
Height map				

Table 6.1: The four terrains used for evaluation of recovery and stabilization procedures.

maps are shown in Table 6.1. We evaluated the performance of our system with and without each element described above. In addition, we evaluated the performance of the system with none of these elements enabled. As shown in Table 6.2, the controller with all elements enabled substantially outperforms the controller when disabling any of these three elements. This effect becomes more pronounced as the terrains become more difficult: Terrain #1 is easy enough that all the controllers achieve 100% success rates, but for Terrains #3 and #4, the advantage of using all the control elements is clear. Statistically, over all four terrains the full controller outperforms the controller with no recovery, with no stabilization, with no closed-loop foot placement, and with none of these elements in terms of success probability with p-values of $p = 2.2 \times 10^{-13}$, $p = 0.0078$, $p = 0.0012$, and $p = 5.8 \times 10^{-11}$ respectively, via a pairwise Bernoulli test.

Subjectively, the failure modes of the different controllers are as expected. Without the stability detection and recovery, the robot frequently falls over entirely after slipping a small amount. Without body stabilization, the robot becomes noticeably less stable during small slips, which sometimes leads to falls that even the recovery routine cannot salvage. Without closed-loop foot placement, the feet can punch into the ground during slips, occasionally flipping the robot. One interesting effect is that without recovery, the controller actually performs *worse* with body stabilization and

Terrain	**All**	**w/o Rec.**	**w/o Stab.**	**w/o CLF**	**None**
1	100%	100%	100%	100%	100%
2	**100%**	60%	95%	95%	55%
3	**95%**	25%	55%	75%	35%
4	**95%**	0%	75%	85%	35%
Total	**97.5%**	46.25%	81.25%	88.75%	56.25%

Table 6.2: Success probabilities out of 20 runs across different terrains for the controller with and without recovery, body stabilization, and closed-loop foot placement.

closed loop foot movement enabled, especially on the more challenging terrains. This appears to be due to the fact that when the robot falls significantly (and makes no attempt to recover) both the body stabilization and closed-loop foot placement attempt to make large changes to the joint angles, causing the robot to become less stable. However, with recovery enabled the robot never strays too far from its desired trajectory without attempting to re-plan; in this case the advantage of using the body stabilization and closed-loop foot placement is clear from the experiments above.

6.5 Motion Planning via Cubic Spline Optimization

We now return to the problem of the static walk foot/body motion planner, planning full foot and body trajectories that move the moving foot from its initial to desired location while maintaining static stability of the robot. In order to plan smooth motions, we use cubic splines to parametrize these trajectories, a common approach in robotic applications (Lin et al., 1983). However, the typical usage of cubic splines within motion planning algorithms suffers from a number of drawbacks. Typically, one uses a standard motion planning algorithm, such as a randomized planner, to generate a sequence of feasible *waypoints*, then fits a cubic spline to these waypoints. However, due to the stochastic nature of the planner, these waypoints often do not lead to a particularly nice final trajectory. Trajectory optimization techniques (Betts, 1998) can help mitigate this problem to some degree, but they usually involve a slow search process, and still typically do not take into account the final cubic spline form of the trajectory.

The basic insight of the method we develop here is that if we initially parametrize the trajectory as a cubic spline, then in many cases we can accomplish both the planning and trajectory fitting simultaneously. That is, we can directly optimize the location of the cubic spline waypoints while obeying many of the same constraints (or approximations thereof) required by a typical planning algorithm. Specifically, we show how to plan smooth task-space trajectories — that is, trajectories where we care primarily about the position of the robot's end effector — while maintaining kinematic feasibility, avoiding collision, and limiting velocities or accelerations, all via a convex optimization problem. Convex optimization problems are beneficial in that they allow for efficiently finding global optimums (Boyd and Vandenberg, 2004) — this allows us to solve the planning tasks in a few milliseconds using off-the-shelf software, suitable for real-time re-planning and control.

6.5.1 The Cubic Spline Optimization Algorithm

Cubic Splines Preliminaries

Here we review the standard methods for fitting cubic splines to a series of waypoints output by a planner. Suppose that a planner outputs some path specified by $T + 1$ desired time-location pairs

$$(t_0, x_0^\star), (t_1, x_1^\star), \ldots, (t_T, x_T^\star) \tag{6.3}$$

where $x_i^\star \in \mathbb{R}^n$ denotes the desired location of the robot at time $t_i \in \mathbb{R}$, specified in task space.

Given these waypoints, there is a unique piecewise-cubic trajectory that passes through the points and satisfies certain smoothness criteria. Specifically, we model the trajectory between times t_i and t_{i+1}, denoted $x_i(t) : \mathbb{R} \to \mathbb{R}^n$, as a cubic function

$$x_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3 \tag{6.4}$$

where $a_i, b_i, c_i, d_i \in \mathbb{R}^n$ are parameters of the cubic spline. The final trajectory $x(t) : \mathbb{R} \to \mathbb{R}^n$ is a piecewise-cubic function that is simply the concatenation of these different cubic trajectories

$$x(t) = \begin{cases} x_0(t) & \text{if } t_0 \le t < t_1 \\ \vdots & \\ x_{T-1}(t) & \text{if } t_{T-1} \le t \le t_T \end{cases} \tag{6.5}$$

where we can assume that the trajectory is undefined for $t < t_0$ and $t > t_T$.

Given the desired waypoints, there exists a unique set of coefficients $\{a_i, b_i, c_i, d_i\}_{i=0,\ldots,T-1}$ such that the resulting trajectory passes through the waypoints and has continuous velocity and acceleration profiles at each waypoint.[2] To compute these coefficients, we first define the matrices $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^{T+1 \times n}$

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} x_0^\star & x_1^\star & \cdots & x_T^\star \end{bmatrix}^T && (6.6) \\ \mathbf{a} &= \begin{bmatrix} a_0 & a_1 & \cdots & a_T \end{bmatrix}^T && (6.7) \end{aligned}$$

with $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$ defined similarly (we define $T+1$ sets of parameters in order to simplify the equations, though we will ultimately only use the $0, \ldots, T-1$ parameters, as described above). Given the $\mathbf{x}$ matrix, we can find the parameters of the cubic splines using the following set of linear equations

$$\begin{aligned} \mathbf{a} &= \mathbf{x} && (6.8) \\ \mathbf{H}_1 \mathbf{b} &= \mathbf{H}_2 \mathbf{x} && (6.9) \\ \mathbf{c} &= \mathbf{H}_3 \mathbf{x} + \mathbf{H}_4 \mathbf{b} && (6.10) \\ \mathbf{d} &= \mathbf{H}_5 \mathbf{x} + \mathbf{H}_6 \mathbf{b} && (6.11) \end{aligned}$$

where the $\mathbf{H}_i \in \mathbb{R}^{(T+1)\times(T+1)}$ matrices depend only (non-linearly) on the times $t_0, \ldots, t_T$. The $\mathbf{H}_i$ matrices are somewhat complex, but are also well-known, so for brevity we

[2]Technically, in order to ensure uniqueness of the spline we also need to impose a constraint on the velocity or acceleration of the endpoints, but we ignore this for the time being.

omit the full definitions here; explicit formulas for the matrices as used here are available in the appendix of (Kolter and Ng, 2009b). However, the important point to glean from this presentation is that the parameters of the cubic splines are *linear* in the desired locations $\mathbf{x}$. Furthermore, the $\mathbf{H}_i$ matrices are all either tridiagonal or bidiagonal, meaning that we can solve the above equations to find the parameters in time $O(T)$.

Cubic Spline Optimization

In this section we present our algorithm: a method for optimizing task-space cubic spline trajectories using convex programming. As before, we assume that we are given an initial plan, now denoted

$$(t_0, \hat{x}_0), (t_1, \hat{x}_1), \ldots, (t_T, \hat{x}_T). \tag{6.12}$$

However, unlike the previous section, we will not require that our final cubic trajectory pass through these points. Indeed, most of the real planning is performed by the optimization problem itself, and the initial plan is required only for some of the approximate constraints that we will discuss shortly; an initial "plan" could simply be a straight line from the start location to the goal location.

The task of optimizing the location of the waypoints while obeying certain constraints can be written formally as

$$\begin{array}{ll} \min\limits_{\mathbf{x}} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{x} \in \mathcal{C} \end{array}$$

where $\mathbf{x}$ is the optimization variable, representing the location of the waypoints, $f : \mathbb{R}^{(T+1)\times n} \to \mathbb{R}$ is the optimization objective, and $\mathcal{C}$ represent the set of constraints on the waypoints. In the subsequent sections, we discuss several possible constraints and objectives that we use in order to ensure that the resulting trajectories are both feasible and smooth. The following is not an exhaustive list, but conveys a general idea of what can be accomplished in this framework.

Additional Variables and Constraints

Spline derivatives at the waypoints. Often we want objective and constraint terms that contain not only the position of the waypoints, but also the velocity, acceleration, and/or jerk (derivative of acceleration) of the resulting cubic spline. Using (6.8) – (6.11), these terms are *linear* functions of the desired positions, and can therefore be included in the optimization problem while maintaining convexity. For instance, since $\dot{x}_i(t_i) = b_i$, we can add the constraint

$$\mathbf{H_1\dot{x}} = \mathbf{H_2x} \tag{6.13}$$

and constrain the these $\dot{\mathbf{x}}$ variables. The same procedure can be used to create variables representing the acceleration or jerk at each waypoint.

Spline position and derivatives at arbitrary times. Oftentimes we may also want to constrain the position, velocity, etc, of the splines not only at the waypoints, but also at the intermediate times. Using the cubic spline formulation, such variables are also a linear function of the waypoint locations. For example, suppose we wanted to add a variable $x(t')$ representing the position of the trajectory at time $t_i < t' < t_{i+1}$. Using equations (6.4) and (6.5),

$$x(t') = a_i + b_i(t' - t_i) + c_i(t' - t_i)^2 + d_i(t' - t_i)^3. \tag{6.14}$$

But from (6.8)–(6.11), a_i, b_i, c_i and d_i are all linear in the desired positions $\mathbf{x}$, so the variable $x(t')$ is also linear in these variables. The same argument applies to adding additional variables that represent the velocity, acceleration, or jerk at any time.[3] Thus, it should be clear from the discussion above that if we use $\mathbf{x}' \in \mathbb{R}^{N\times n}$ to denote the spline positions at a variety of intermediate times, where N denotes the number of additional times that we are constraining, we can solve for these positions via a

[3]In theory, if we wanted to ensure that the entire spline obeys a position or derivative constraint, we would have to add an infinite number of such variables. However, as we will show, in practice we can obtain good results by introducing a very small number of additional variables, greatly increasing the practicality of the approach. This same consideration applies to the kinematic feasibility and collision constraints.

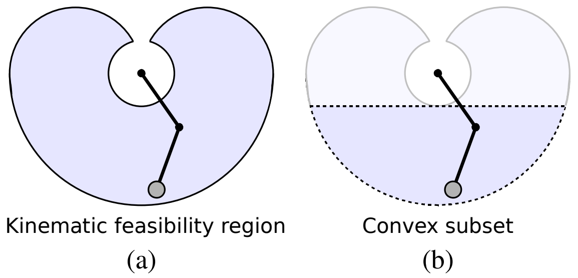


Figure 6.6: Illustration of kinematic feasibility constraints. (a) Kinematically feasible region for $q_1 \in [-\pi/2, \pi/2]$, $q_2 \in [-2.6, 2.6]$. (b) Convex subset of the feasible region.

linear system

$$\mathbf{x}' = \mathbf{G}_1\mathbf{x} + \mathbf{G}_2\dot{\mathbf{x}} \tag{6.15}$$

and similarly for other derivatives.

Kinematic feasibility constraints. Since we are specifically focused on planning trajectories in task-space, a key requirement is that points on the spline must be kinematically feasible for the robot. While the kinematic feasibility region of an articulated body with joint stops is not typically a convex set, we can usually find a suitable *subset* of this kinematic region that *is* convex.

For example, consider the double pendulum shown in Figure 6.6 (which has very similar kinematics to a 2D view of the LittleDog's leg). The kinematically feasible region, when joint one is restricted to the range $[-\pi/2, \pi/2]$ and joint two is restricted to the range $[-2.6, 2.6]$, is shown in Figure 6.6 (a). Although this region is not convex, we can easily find a convex subset, such as the region shown in Figure 6.6 (b). Thus, if we constrain motion to occur in some convex subset of the kinematically feasible region, we can ensure kinematic feasibility while retaining benefits of the convex optimization procedure.

Collision constraints. Although general collision constraints can be quite difficult to handle in our framework, in many simple cases we can approximate such constraints using a simple method shown in Figure 6.7. Here Figure 6.7 (a) shows the initial plan used by the cubic spline optimizer (recall from above such a plan need to be feasible). Two waypoints on this initial trajectory violate the collision constraint, so we simply add the constraint, at each of these times, that the resulting

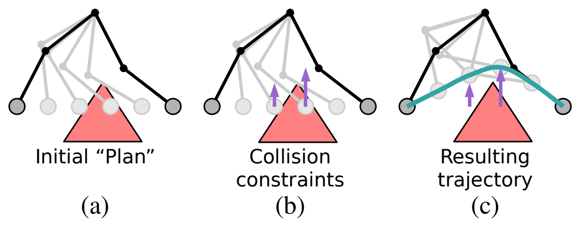


Figure 6.7: Illustration of collision constraints. (a) Initial (infeasible) plan. (b) Height constraints imposed to avoid collision with obstacle. (c) Resulting optimized cubic spline trajectory.

waypoint must lie above the the obstacle by some margin; Figure 6.7 (b) shows this constraint, and Figure 6.7 (c) shows the resulting trajectory. This technique is an approximation, because 1) it only constrains the end-effector position and still could lead to a collision with the articulated body, 2) it assumes that the x-position of the waypoints after optimization doesn't change, when in fact it can and 3) as mentioned in Footnote 3, these collision constraints are only imposed at a finite number of points, so we have to insure that the "resolution" of these points is smaller than any thin obstacles. Nonetheless, as we show, this simple approximation works quite well in practice, and allows us to maintain convexity of the optimization problem.

For some planning problems, adding enough constraints of any of the preceding types can lead to an infeasible optimization problem. Thus, this approach is *not* suited to all planning situations; if plans must traverse through non-convex, narrow "corridors" in the robot's configuration space, then slower, traditional motion planning algorithms may be the only possible approach. Furthermore, the technique as described is somewhat specialized to domains, like the LittleDog, where the "up" direction tends to move the end effector away from obstacles; to handle more general collision constraints it would be necessary to constrain spline positions based on normals from arbitrary surfaces, but for these types of more complex situations it is likely that the non-convexity of the robot's free space would cause additional concerns. However, for situations where our method can be applied, such as the LittleDog planning tasks, our method can produce highly-optimized trajectories extremely quickly.

Optimization Objectives Given the variables and constraints described above, we lastly need to define our final optimization objective. While we have experimented with several different possible optimization objectives, one that appears to work quite well is to penalize the squared velocities at the waypoints and at a few intermediate points between each waypoint. More formally, we use the optimization objective

$$f(\dot{\mathbf{x}}, \dot{\mathbf{x}}') = \operatorname{tr} \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \operatorname{tr} \dot{\mathbf{x}}'^T \dot{\mathbf{x}}' \tag{6.16}$$

where $\dot{\mathbf{x}}$ represents the velocity at each waypoint and $\dot{\mathbf{x}}'$ represents the velocities at the midpoint between each waypoint. This objective has the effect of discouraging very large velocities at any of the spline points, which leads to trajectories that travel minimal distances while keeping fairly smooth. However, while this objective works well for our settings, there are many other possible objective functions that might work better in other cases such as minimizing the maximum velocity, average or maximum acceleration, average distance between spline points, or (using approximations based on the Jacobians along the initial trajectory) average or maximum joint velocities or torques.

One objective that cannot be easily minimized is the total time of the trajectory, because equations (6.8)–(6.11) involve non-linear, non-convex functions of the times. However, there has been previous work in approximately optimizing the times of cubic splines (Cao et al., 1997; Park et al., 1997; Vaz and Fernandes, 2006), and if the total time of the trajectory is ultimately the most important objective, these techniques can be applied.

Application to Fast Foot/Body Motion Planning

Here we describe how the cubic spline optimization technique can be applied to the task of planning foot and body trajectories for the static walking gait. Recall that the basic planning task that we are considering is as follows: given a current position of the robot and an upcoming footstep, plan trajectories for the feet and robot center of gravity (COG) that achieves this footstep, while requiring that the COG and foot locations are kinematically feasible, collision free, and maintain static stability. Although the footstep planner plans one footstep at a time, in practice we plan trajectories here for two steps at a time to ensure that the COG ends in a desirable

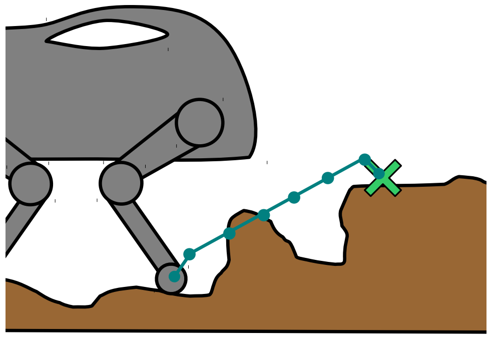


Figure 6.8: Foot planning task, and initial trajectory.

location for the next step.

6.5.2 Planning Foot Trajectories

An illustration of the foot trajectory planning task is shown in Figure 6.8, along with the initial plan we supply to the cubic spline optimization. The initial plan is a simple trapezoid, with three waypoints allocated for the upward and downward "ramps", and the remaining waypoints in a line, spaced in 2cm intervals. We use the following optimization problem to plan the foot trajectories:

$$\min_{\mathbf{x},\mathbf{x}',\dot{\mathbf{x}},\dot{\mathbf{x}}',\ddot{\mathbf{x}}} \quad \operatorname{tr} \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \operatorname{tr} \dot{\mathbf{x}}'^T \dot{\mathbf{x}}' \tag{6.17}$$

$$\text{subject to} \quad \mathbf{H}_1 \dot{\mathbf{x}} = \mathbf{H}_2 \mathbf{x} \tag{6.18}$$

$$\ddot{\mathbf{x}} = \frac{1}{2}(\mathbf{H}_3 \mathbf{x} + \mathbf{H}_4 \dot{\mathbf{x}}) \tag{6.19}$$

$$\mathbf{x}' = \mathbf{G}_1 \mathbf{x} + \mathbf{G}_2 \dot{\mathbf{x}} \tag{6.20}$$

$$\dot{\mathbf{x}}' = \mathbf{G}_3 \mathbf{x} + \mathbf{G}_4 \dot{\mathbf{x}} \tag{6.21}$$

$$\mathbf{x}_{0,:} = \hat{x}_0, \ \mathbf{x}_{T,:} = \hat{x}_T, \ \dot{\mathbf{x}}_{0,:} = 0, \ \dot{\mathbf{x}}_{T,:} = 0 \tag{6.22}$$

$$\ddot{\mathbf{x}}_{t,x} \geq 0, \ \ddot{\mathbf{x}}_{t,y} \geq 0, \quad t = 0, 1 \tag{6.23}$$

$$\ddot{\mathbf{x}}_{t,x} \leq 0, \ \ddot{\mathbf{x}}_{t,y} \leq 0, \quad t = T-1, T \tag{6.24}$$

$$\left.\begin{array}{l} \ddot{\mathbf{x}}_{t,x} \geq \ddot{\mathbf{x}}_{t+1,x} \\ \ddot{\mathbf{x}}_{t,y} \geq \ddot{\mathbf{x}}_{t+1,y} \end{array}\right\} \quad t = 2, \ldots, T-3 \tag{6.25}$$

$$\ddot{\mathbf{x}}_{t,z} < 0, \quad t = 2, \ldots, T-2 \tag{6.26}$$

$$\begin{aligned}(\mathbf{x}_{t,x} - \mathbf{x}_{t+2,x})^2 + (\mathbf{x}_{t,y} - \mathbf{x}_{t+2,y})^2 \\ \leq (3\text{cm})^2, \quad t = 0, T-2\end{aligned} \tag{6.27}$$

$$\mathbf{x}'_{i,z} \geq \hat{x}_z(t'_i) + 2\text{cm}, \quad i = 1, \ldots, N \tag{6.28}$$

$$\mathbf{x}_{t,z} \leq \max_{i=1,\ldots,N} \hat{x}_z(t'_i), \quad t = 1, \ldots T. \tag{6.29}$$

While there are many different terms in this optimization problem, the overall idea is straightforward. The optimization objective (6.17) is the squared velocity objective we discussed earlier; constraints (6.18)–(6.21) are the standard cubic spline equations for adding additional variables representing respectively the velocity at the waypoints, the acceleration at the waypoints, additional position terms, and additional velocity terms;[4] (6.22) insures that the spline begins and ends at the start and goal, with zero velocity; (6.23) and (6.24) ensure that the x, y accelerations are positive (negative) at the start (end) ramp, which in turn ensures that the trajectory will never overshoot

[4]In greater detail, we add four additional velocity terms, in the midpoints of the waypoints on the "ramp" portion of the initial trajectory. We add N additional position terms, one at each 1cm interval along the top portion of the initial trajectory.

the start and end locations;[5] (6.25) extends the previous constraint slightly to also ensure that the x, y accelerations during the main trajectory portion are monotonic; (6.26) forces the z accelerations during the main portion of the trajectory to be negative, which ensures that the spline moves over any obstacles in one single arch; (6.27) ensures that the last waypoints in the ramp don't deviate from the start and end positions by more than 3cm; finally, (6.28) and (6.29) ensure that the z position of the spline is 2cm above any obstacle, and that no waypoint is more than 2cm higher than the tallest obstacle (here $\hat{x}_z(t_i)$ denotes the height of the terrain at time t_i along the initial trajectory). This particular set of objectives and constraints were developed specifically for the foot trajectory planning task, and many of the constraints were developed over time in response to specific situations that caused simpler optimization problems to produce sub-optimal plans.

Planning COG Trajectories

The aim of planning a trajectory for the COG is twofold: maintaining stability of the robot while allowing the feet to reach their targets. Thus, we want to position the body so as to maintain kinematic feasibility for the moving feet, and keep the robot's COG in the support triangle; since we are planning for two steps, when planning the body movement for a back foot step we require the COG to be in a double support triangle, which is stable for both steps (see, e.g., (Kolter et al., 2008b)). We always use nine waypoints in the COG trajectory splines: three to move the COG into the supporting triangle, and three for each foot movement. Since planning the COG trajectory requires knowledge of the foot locations, we first use the method above to plan trajectories for the moving feet, which we denote $x_{f_1}(t)$ and $x_{f_2}(t)$. We then use

[5]This constraint and next assume that the $\hat{x}_{0,x} \leq \hat{x}_{T,x}$ and $\hat{x}_{0,y} \leq \hat{x}_{T,y}$. In the case that these inequalities are reversed, the corresponding inequalities in the constraints are also reversed.

the following optimization problem to plan the COG trajectory:

$$\begin{array}{rll} \displaystyle\min_{\mathbf{x},\dot{\mathbf{x}},\dot{\mathbf{x}}'} & \begin{array}{l} \operatorname{tr} \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \operatorname{tr} \dot{\mathbf{x}'}^T \dot{\mathbf{x}}' \\ +\lambda \sum_{i=3}^{5} \operatorname{feas}(\mathbf{x}_{i,:}, x_{f_1}(t_i), f_1) \\ +\lambda \sum_{i=5}^{7} \operatorname{feas}(\mathbf{x}_{i,:}, x_{f_2}(t_i), f_2) \end{array} & (6.30) \\ \text{subject to} & \mathbf{H}_1 \dot{\mathbf{x}} = \mathbf{H}_2 \mathbf{x} & (6.31) \\ & \dot{\mathbf{x}}' = \mathbf{G}_3 \mathbf{x} + \mathbf{G}_4 \dot{\mathbf{x}} & (6.32) \\ & \mathbf{x}_{0,:} = \hat{x}_0, \dot{x}_0,\ \mathbf{x}_{2,:} = \hat{x}_2,\ \mathbf{x}_{8,:} = \hat{x}_8 & (6.33) \\ & \dot{\mathbf{x}}_{0,:} = \dot{x}_{\text{init}} & (6.34) \\ & \mathbf{x}_{i,:} \in \mathcal{S},\ \ i = 3, \ldots, 7 & (6.35) \end{array}$$

where

$$\operatorname{feas}(x_{\text{body}}, x_{\text{foot}}, f) \qquad (6.36)$$

denotes the squared distance from x_{foot} to the kinematically feasible region of foot f, given that the COG is positioned at point x_{body}, and where $\mathcal{S}$ denotes the support triangle. Intuitively, the optimization objective (6.30) is a weighted combination of the kinematic infeasibility of the moving feet plus the velocity terms we discussed earlier — in practice, we choose $\lambda = 100$ to try to make the system as close to kinematically feasible as possible, and only later try to minimize the velocities; constraints (6.31) and (6.32) are again the standard cubic spline equations for velocity terms — here we add eight additional velocity terms, one at the midpoint of each two waypoints; (6.33) ensures that the trajectory begins, enters the supporting triangle, and ends at the initially specified waypoints, while (6.34) ensures that the initial velocity of the spline is equal to the COG's current velocity; finally, (6.35) ensures that the COG waypoints will be inside the supporting triangle during the times that the feet are moving. Since the optimization problem treats distance to the feasible set as an optimization objective rather than a constraint, there is no concern that the optimizer will fail to find a solution.

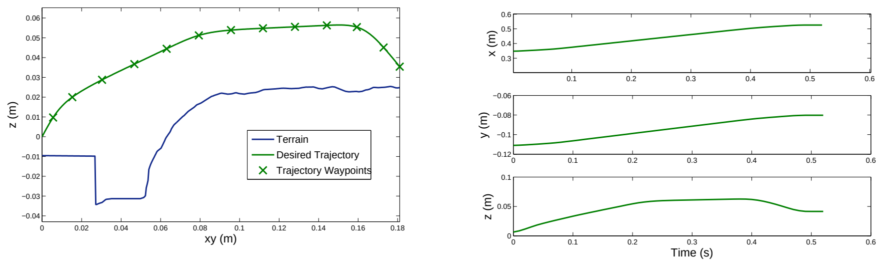


Figure 6.9: Typical example of a foot trajectory generated by the algorithm. The top figure shows the resulting trajectory in 3D space, while the bottom shows each component as a function of time.

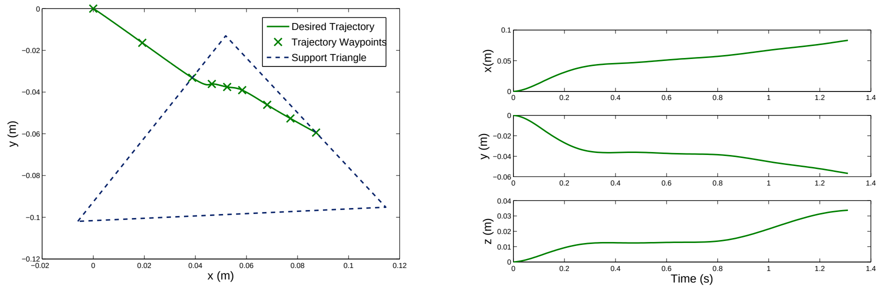


Figure 6.10: Typical example of a COG trajectory generated by the algorithm.

6.5.3 Experimental Evaluation

The cubic spline optimization procedure is used for all the experiments reported in Section 6.7, but here we evaluate this specific component of the system. We begin with a qualitative look at the trajectories generated by this method. Figure 6.9 shows a typical footstep trajectory generated by this method. As we can see, the trajectory moves the foot from its initial location to the desired location in one fluid motion, stepping high enough to avoid any obstacles. Likewise, Figure 6.10 shows a typically COG trajectory generated by the algorithm. Notice that the trajectory inside the supporting triangle is not just a straight line: the algorithm adjusts the trajectory in this manner to maximize the kinematic feasibility of the moving feet.

Of course, while examining the splines in this manner can help give an intuition about kind of trajectories generated by our algorithm, we are ultimately interested

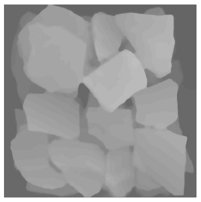
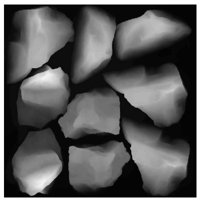
Terrain	1	2
Max Height	6.4 cm	10.4 cm
Picture		
Height map		

Table 6.3: The training and testing terrains used to evaluate the cubic spline trajectory planning on LittleDog..

	Easy Terrain(1)		Challenging Terrain(2)	
Metric	CSO	Previous	CSO	Previous
% Successful Trials	100%	100%	**100%**	70 %
Speed (cm/sec)	**7.02 ± 0.10**	5.99 ± 0.07	**6.30 ± 0.12**	5.59 ± 0.31
Avg. Tracking Error (cm)	**1.28 ± 0.03**	1.40 ± 0.06	**1.27 ± 0.05**	1.55 ± 0.09
Avg. # Recoveries	0.0	0.0	**0.5 ± 0.37**	2.22 ± 1.45

Table 6.4: Performance of cubic spline optimization on the two quadruped terrains. Terms include 95% confidence intervals where applicable.

in whether or not the method actually improves performance on the LittleDog. To evaluate this, we tested the system on two terrains of varying difficulty, shown in Table 6.3. We compare the cubic spline optimization approach to an older trajectory planning method described in (Kolter et al., 2008b), which uses simple box-shapes to step over terrain, and linear splines for moving the COG.

Table 6.4 shows the performance of the quadruped both with and without the cubic spline optimization. We ran 10 trials on each of the terrains, and evaluated the systems using 1) fraction of successful runs, 2) speed over terrain, 3) average number of "recoveries," as specified by the method of the previous section and 4) average tracking error (i.e., distance between the planned and actual location) for the moving foot. Perhaps the most obvious benefit of the cubic spline optimization method is that the resulting speeds are faster; this is not particularly surprising, since the splines output by our planner will clearly be more efficient than a simple box pattern over obstacles. However, equally important is that the cubic spline optimization also leads to more *robust* behavior, especially on the challenging terrain: the previous method only succeeds in crossing the terrain 70% of the time, and even when it does succeed it typically needs to execute several recoveries, whereas the cubic spline optimization method crosses the terrain in all cases, and executes many fewer recoveries. We believe that this is because the cubic spline method attempts to maintain kinematic feasibility of the moving foot at all times. This is seen in Table 6.4 from the fact that the cubic spline optimization approach has lower tracker error, implying more accurate placement of feet, and therefore greater robustness is challenging environments.

6.6 Cost Learning via Hierarchical Apprenticeship Learning

Recall that both the route planner and footstep planner make use of *cost maps*, functions which indicate, for every point on the terrain, the goodness of placing the robot's center or a foot at that location. However, it is very challenging to specify a proper cost function manually, as this requires quantifying the trade-off between many features, including progress toward a goal, the height differential between feet, the slope of the terrain underneath the feet, etc. Because of this, we adopt a method known as *apprenticeship learning*, based upon the insight that often it is easier for an "expert" to demonstrate the desired behavior than it is to specify a cost function that induces this behavior. In the footstep planning example, for instance, it may be challenging to manually specify the weights of a cost function that leads to good footsteps; however, it is much easier, for example after seeing the robot step in a poor location on the terrain, to indicate a better location for its footstep. Apprenticeship learning has been successfully applied to many other robotics domains, and is a natural fit for such problems (Abbeel and Ng, 2004; Ratliff et al., 2006; Neu and

Szepesvári, 2007).

However, a difficulty arises in applying existing apprenticeship learning methods to a task like the LittleDog. Typical apprenticeship learning algorithms require the expert to demonstrate a complete trajectory from the start to the goal, which in this case corresponds to a complete set of footsteps across the terrain; this itself is a highly non-trivial task, even for an expert. Motivated by these difficulties, we developed for this task an algorithm for *hierarchical apprenticeship learning*. Our approach is based on the insight that, while it may be difficult for an expert to specify entire optimal trajectories in a large domain, it is much easier to "teach hierarchically": that is, if we employ a hierarchical control scheme to solve our problem, it is much easier for the expert to give advice independently at each level of this hierarchy. At the lower levels of the control hierarchy, our method only requires that the expert be able to demonstrate good *local* behavior, rather than behavior that is optimal for the entire task. This type of advice is often feasible for the expert to give even when the expert is entirely unable to give full trajectory demonstrations. Thus the approach allows for apprenticeship learning in extremely complex, previously intractable domains. We first present the general hierarchical apprenticeship learning algorithm, then describe its application to route and footstep planning.

6.6.1 The Hierarchical Apprenticeship Learning Algorithm

Definitions

To describe the hierarchical apprenticeship learning algorithm (abbreviated HAL), we use the formalism of Markov Decision Processes (MDPs), as described briefly in Chapter 2. To review, an MDP is a tuple (S, A, P, D, H, C), where S is a set of states; A is a set of actions, $P : S \times A \to S$ is a set of state transition probabilities; D is a distribution over initial states; H is the horizon which corresponds to the number of time-steps considered; and $C : S \to \mathbb{R}$ is a cost function.[6] We use the notation MDP\C to denote an MDP minus the cost function. A policy π is a mapping from

[6]Here we assume the cost does not depend on the action, though the extension to state-action costs is straightforward.

states to a probability distribution over actions. The value or cost-to-go of a policy π is given by $J(s,\pi) = E\left[\sum_{t=0}^{H} C(s_t)|\pi\right]$, where the expectation is taken with respect to the random state sequence $s_0, s_1, \ldots, s_H$, with s_0 drawn from D, and picking actions according to π.

Often the cost function C can be represented more compactly as a function of the state. Let $\phi : S \rightarrow \mathbb{R}^n$ be a mapping from states to a set of features. We consider the case where the cost function C is a linear combination of the features

$$C(s) = w^T \phi(s) \tag{6.37}$$

for parameters $w \in \mathbb{R}^n$. Then we have that the value of a policy ϕ is linear in these cost function weights

$$J(\pi) = E\left[\sum_{t=0}^{H} C(s_t)\middle|\pi\right] = E\left[\sum_{t=0}^{H} w^T\phi(s_t)\middle|\pi\right] = w^T E\left[\sum_{t=0}^{H} \phi(s_t)\middle|\pi\right] \equiv w^T \mu_\phi(\pi)$$

where we used linearity of expectation to bring w outside of the expectation. The last quantity defines the vector of *feature expectations* $\mu_\phi(\pi) = E[\sum_{t=0}^{H} \phi(s_t)|\pi]$.

Cost Decomposition in HAL

At the heart of the HAL algorithm is a simple decomposition of the cost function that links the two levels of control. Suppose that we are given a hierarchical decomposition of a control task in the form of two MDP\Cs — a low-level and a high-level MDP\C, denoted $M_\ell = (S_\ell, A_\ell, T_\ell, H_\ell, D_\ell)$ and $M_h = (S_h, A_h, T_h, H_h, D_h)$ respectively — and a partitioning function $\psi : S_\ell \rightarrow S_h$ that maps low level states to high-level states (the assumption here is that $|S_h| \ll |S_\ell|$ so that this hierarchical decomposition actually provides a computational gain). For example, for the LittleDog planner task the low-level MDP\C is the footstep planning domain, where the state consists of all four foot locations, whereas the high-level MDP is the route planning domain, where the state consists of only the robot's center. As standard in apprenticeship learning, we suppose that the cost in the low-level MDP\C can be represented as a linear function of state features, $C(s_\ell) = w^T\phi(s_\ell)$. The HAL algorithm then assumes that the cost of a high-level state is equal to the average cost over all its corresponding low-level states. Formally

$$\begin{aligned} C(s_h) &= \frac{1}{N(s_h)} \sum_{s_\ell \in \psi^{-1}(s_h)} C(s_\ell) \\ &= \frac{1}{N(s_h)} \sum_{s_\ell \in \psi^{-1}(s_h)} w^T \phi(s_\ell) = \frac{1}{N(s_h)} w^T \sum_{s_\ell \in \psi^{-1}(s_h)} \phi(s_\ell) \end{aligned} \tag{6.38}$$

where $\psi^{-1}(s_h)$ denotes the inverse image of the partitioning function and $N(s_h) = |\psi^{-1}(s_h)|$. While this may not always be the most ideal decomposition of the cost function in many cases—for example, we may want to let the cost of a high-level state be the *minimum* of its low level state costs to capture the fact that an ideal agent would always seek to minimize cost at the lower level, or alternatively the *maximum* of its low level state costs to be robust to worst-case outcomes—it captures the idea that in the absence of other prior information, it seems reasonable to assume a uniform distribution over the low-level states corresponding to a high-level state. An important consequence of (6.38) is that the high level cost is now also linear in the low-level cost weights w. This will enable us in the subsequent sections to formulate a unified hierarchical apprenticeship learning algorithm that is able to incorporate expert advice at both the high level and the low level simultaneously.

Expert Advice and a Convex Formulation

As in standard apprenticeship learning, expert advice at the high level consists of full policies demonstrated by the expert. However, because the high-level MDP\C can be significantly simpler than the low-level MDP\C, this task can be substantially easier. If the expert suggests that $\pi_{h,E}^{(i)}$ is an optimal policy for some given MDP\C $M_h^{(i)}$, then this corresponds to the following constraint, which states that the expert's policy outperforms all other policies:

$$J^{(i)}(\pi_{h,E}^{(i)}) \leq J^{(i)}(\pi_h^{(i)}) \quad \forall \pi_h^{(i)}. \tag{6.39}$$

Equivalently, using (6.38), we can formulate this constraint as follows

$$w^T \mu_\phi^{(i)}(\pi_{h,E}^{(i)}) \leq w^T \mu_\phi(\pi_h^{(i)}) \quad \forall \pi_h^{(i)}. \tag{6.40}$$

and where in practice we will use observed feature counts $\hat{\mu}_\phi^{(i)}(\pi_{h,E}^{(i)})$ in lieu of the true expectations.

Our approach differs from standard apprenticeship learning when we consider advice at the low level. Unlike the apprenticeship learning paradigm where an expert specifies full trajectories in the target domain, we allow for an expert to specify single, greedy actions in the low-level domain. Specifically, if the agent is in state s_ℓ and the expert suggests that the best greedy action would move to state s'_ℓ, this corresponds directly to a constraint on the *cost* function, namely that

$$C(s'_\ell) \leq C(s''_\ell) \tag{6.41}$$

for all other states s''_ℓ that can be reached from the current state (we say that s''_ℓ is "reachable" from the current state s_ℓ if $\exists a$ s.t.$P_{s_\ell a}(s''_\ell) > \epsilon$ for some $0 < \epsilon \leq 1$). This is equivalent to the following constraint on the constraints on the cost function parameters w,

$$w^T \phi(s'_\ell) \leq w^T \phi(s''_\ell) \tag{6.42}$$

for all s''_ℓ reachable from s_ℓ.

Since the high level and low level expert advice are both given as linear constraints on the features w, we can combine both types of advice into a single convex optimization problem. To resolve the ambiguity in w, and to allow the expert to provide noisy advice, we use regularization and slack variables (similar to standard SVM formulations), which results in the optimization problem

$$\begin{aligned} \min_{w,\eta \geq 0, \xi \geq 0} & \frac{1}{2}\|w\|_2^2 + \lambda_\ell \sum_{j=1}^{m} \xi^{(j)} + \lambda_h \sum_{i=1}^{n} \eta^{(i)} \\ \text{s.t.} & w^T \phi(s_\ell'^{(j)}) \leq w^T \phi(s_\ell''^{(j)}) - 1 + \xi^{(j)} \quad \forall s_\ell''^{(j)}, j \\ & w^T \hat{\mu}_\phi^{(i)}(\pi_{h,E}^{(i)}) \leq w^T \mu_\phi(\pi_h^{(i)}) - 1 + \eta^{(i)} \quad \forall \pi_h^{(i)}, i. \end{aligned} \tag{6.43}$$

where $\pi_h^{(i)}$ indexes over all high-level policies, η_i and ξ_i are slack variables, i indexes over all MDPs, $s_\ell''^{(j)}$ indexes over all states reachable from $s_\ell'^{(j)}$ and j indexes over all low-level demonstrations provided by the expert, and λ_h and λ_ℓ are a regularization parameters. Despite the fact that there are an exponential number of possible policies we can solve this optimization problem using a number of techniques, including dual formulations (Taskar et al., 2005), subgradient algorithms (Ratliff et al., 2006), and constraint generation (Tsochantaridis et al., 2005). We use a method based on this last approach, and use the following constraint generation algorithm:

1. Begin with no expert path constraints.
2. Find the current cost weights by solving the current optimization problem.
3. Solve the reinforcement learning problem at the high level of the hierarchy to find the optimal (high-level) policies for the current cost for each MDP\C, i. If the optimal policy violates the current (high level) constraints, then add this constraint to the current optimization problem and go to Step (2). Otherwise, no constraints are violated and the current cost weights are the solution of the optimization problem.

Solving this optimization problem provides us with a single cost function that is consistent with the expert advice both for the low and high levels.

6.6.2 Application to Cost Map Learning

As mentioned briefly in the previous section, the application of the HAL algorithm to route and footstep planning is conceptually straightforward: the route planner takes the place of the high-level planner and the footstep planner takes the place of the low-level planner.

Of course, a crucial element of the actual implementation of this approach is determining what features to use to represent the cost function for the footstep and high-level planner. We use the following set of features (note that for a given point on the terrain we actually form four feature vectors, one corresponding to each foot, with local features properly reflected to account for symmetry of the robot):

- At each point in the height map (discretized at a resolution of 0.5 cm), we consider local heights maps of different sizes (squares of 5, 7, 11, and 21 grid cells around the current point), and generate five features for each of these maps: 1) standard deviation of the heights, 2) average slope in the x direction, 3) average slope in the y direction 4) maximum height relative to the center point and 5) minimum height relative to the center point, for a total of 20 features.
- A boolean feature indicating whether or not the foot location leads to a collision when the robot is placed in its default position.
- The distance of the point from the desired foot location (i.e., the location that the footstep planner would place its foot if all costs were equal).
- The area and in-radius of the support triangle formed by the stationary feet for the upcoming step.
- A constant value.

While the first two features above can be generated once before execution, the second two require the actual pose of the robot, and so are generated in real time by the footstep planning mechanism. This leads to a cost function that is a linear function of 25 state features.

To form the cost for the high-level route planner, we aggregate features from the footstep planner. In particular, for a given center location of the robot's body we consider all the footstep features within a 3 cm radius of the each foot's default position, and aggregate these features to form the features for the route planner. Note that the features above that can only be generated during execution (the distance from the desired footstep, the area and in-radius of the support triangle) will be the same for each high-level center location, and so can be ignored, allowing us to run the route planner prior to any execution. While this cost function is naturally an approximation, we found that it performed very well in practice, possibly due to its ability to account for stochasticity of the domain.

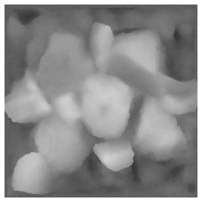
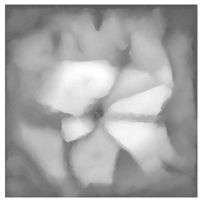
Terrain	Training	Testing
Max Height	8.0 cm	11.7 cm
Picture		
Height map		

Table 6.5: The training and testing terrains used to evaluate HAL.

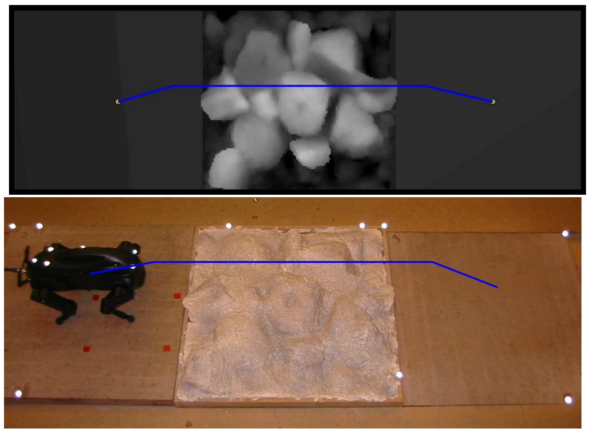


Figure 6.11: High-level (route) expert demonstration.

6.6.3 Experimental Evaluation

While the results that we present in Section 6.7 will all use the cost function learned using this algorithm, in this section we present results explicitly demonstrating the performance of the system with and without the HAL algorithm. All experiments were carried out on two terrains: a relatively easy terrain for training, and a significantly more challenging terrain for testing, shown in Table 6.5. To give advice at the high level, we specified complete body trajectories for the robot's center of mass, as shown in Figure 6.11. To give advice for the low level we looked for situations

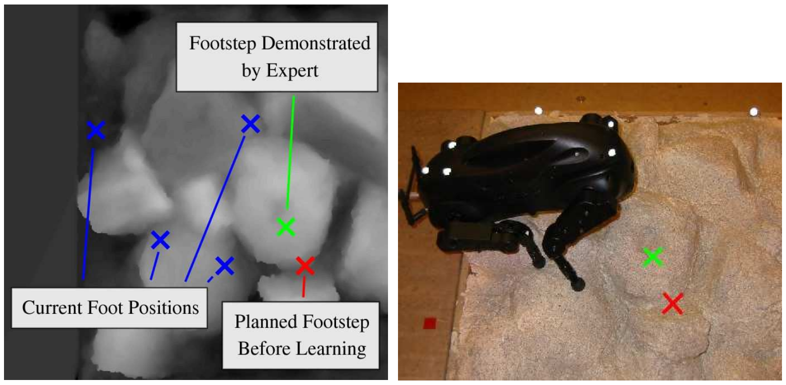

Footstep Demonstrated by Expert
Current Foot Positions
Planned Footstep Before Learning

Figure 6.12: Low-level (footstep) expert demonstration.

	Training	**Testing**
HAL	31.03	33.46
Feet Only	33.46	45.70
Path Only	—	—
No Learning	45.70	—

Table 6.6: Execution times for different constraints on training and testing terrains. Dashes indicate that the robot fell over and did not reach the goal.

in which the robot stepped in a suboptimal location, and then indicated the correct greedy foot placement (by clicking on a point in the terrain in a computer interface), as shown in Figure 6.12. The entire training set consisted of a single high-level path demonstration across the training terrain, and 20 low-level footstep demonstrations on this terrain; it took about 10 minutes to collect the data. The general cost map that was learned, as expected, typically preferred flat areas over areas close to a cliff, but the precise trade-offs implied by the learned cost function are difficult to evaluate except with respect to the resulting performance.

Even from this small amount of training data, the learned system achieved excellent performance, not only on the training board, but also on the much more difficult testing board. Figure 6.13 shows the route and footsteps taken for each of the different possible types of constraints, which shows a very large qualitative difference between the footsteps chosen before and after training. Table 6.6 shows the crossing times for each of the different types of constraints. As shown, the HAL algorithm outperforms

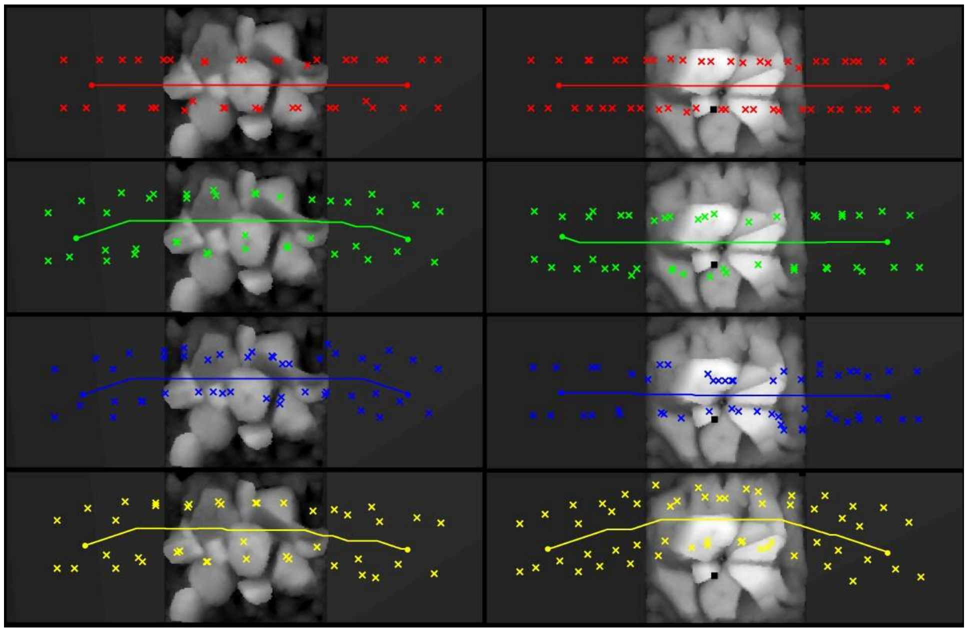


Figure 6.13: Body and footstep plans for different constraints on the training (left) and testing (right) terrains: (First/Red) No Learning, (Second/Green) HAL, (Third/Blue) Path Only, (Fourth/Yellow) Footstep Only.

all the intermediate methods. Using only footstep constraints does quite well on the training board, but on the testing board the lack of high-level training leads the robot to take a very roundabout route, and it performs much worse. The LittleDog fails at crossing the testing terrain when learning from the path-level demonstration only or when not learning at all.

6.7 Learning Locomotion Program Results

While the previous sections have included results as a means of evaluating the various approaches, here we present the results of our system in the official tests of the Learning Locomotion program. As discussed in Section 6.2, the Learning Locomotion program consisted of three phases, each with specific metrics in terms of the speed and terrain height: for Phase I, the metric was 1.2 cm/s over 4.8 cm obstacles; for Phase II, 4.2 cm/s over 7.8 cm obstacles; and for Phase III, 7.2 cm/s over 10.8 cm

Terrain Type	**Average Speed**
Rocks	5.6 cm/s
Slope	6.5 cm/s
Steps	5.4 cm/s
Modular Rocks	7.4 cm/s
Barrier	5.9 cm/s
Modular Logs	7.2 cm/s
Gap	5.9 cm/s
Average	6.3 cm/s

Table 6.7: Speeds for Phase II system and terrains. Average speed is based upon the best two out of three runs. We received no information as to whether the system based all 3/3 test or only 2/3, but all test passed at least 2/3 of the runs.

obstacles. As the metric evaluations were much more standardized in Phases II and III than in Phase I, we present here our results for these later phases of the project, along with analysis about how we obtained these results.

For Phase II and Phase III, the testing procedures operated as follows. Throughout each phase, all the teams had access to seven terrain boards of various types (the different types are listed in the tables below), known as the "A" terrains. For the final metric evaluations, all teams were tested on a different set of terrains (the "B" terrains) that were informally from the same "class" of obstacles as the A boards, but which were not available prior to the tests. Teams did have the opportunity test a very limited number of times on these boards prior to the final metric tests, but received no information other than the number of times the system successfully crossed and the speed of the gait. Thus, the tests below represent an evaluation on terrain that truly was novel to the robot, and required that the robot be able to adapt to situations that we could not simply hand-engineer.

Table 6.7 shows our performance on the Phase II metrics. For Phase II, our research was focused entirety on static gaits: the static gait described in Section 6.3 was built in its entirely during Phase II, with only small changes in Phase III to allow for the ability to switch to other gaits in real-time. We made use of no specialized maneuvers or trotting, so run times for all the different terrains were similar (the more challenging rocks and steps were slightly slower, but there was ultimately little

Terrain Type	**Average Speed**	**Success**
Gap	13.1 cm/s	3/3
Barrier	13.2 cm/s	3/3
Sloped Rocks	5.7 cm/s	3/3
Modular Rocks	11.3 cm/s	3/3
Logs	4.8 cm/s	3/3
Steps	6.2 cm/s	3/3
Average	9.7 cm/s	3/3
Side Slope	23.7 cm/s	2/3
V-ditch	16.8 cm/s	3/3
Scaled Steps	6.6 cm/s	2/3

Table 6.8: Speeds and success rates for Phase III system and terrains. Average speed is based upon best two runs. The first six terrains represent the "standard" Phase III tests, while the last three represent three additional optional terrains. The video references in the text shows all nine terrains.

variation in the run times).

Table 6.8 shows our performance on the Phase III metrics. Unlike Phase II, for Phase III our focus was entirely on dynamic gaits, in particular the trot and the specialized maneuvers. Ultimately, we were able to develop controllers that allowed us to use trots and specialized maneuvers for all but two of the terrains: the sloped rocks and the logs. Predictably, our running times on these terrains were similar to our Phase II running times (even a bit slower for the logs, owing to the added difficulty of the higher obstacles), but were much faster for terrains where we could exploit dynamic gaits. Nonetheless, the results from this phase also served to convince us that, while dynamic gaits are suitable to many scenarios, static walking still serves a genuine purpose for robots similar to the LittleDog: despite extensive development, we were unable to achieve reliable performance from dynamic behavior on either the sloped rocks or the logs terrains. Thus, we feel that robots which can exhibit both these modes of locomotion will be the most interesting research testbeds in the years to come.

Detailed performance results for all teams during the testing have not been officially released, so we compare only briefly to the other teams. Our average speed in Phase II was the highest of all six competing Learning Locomotion teams, and we crossed all terrains above metric speed; our average speed was the second highest during Phase III though other teams crossed more terrains above the metric speed. Videos of the our robot crossing all the Phase III terrains (only the "A" boards, but the performance on these boards was virtually identical to the performance on the metric "B" boards), are available at: `http://ai.stanford.edu/~kolter/ijrr09ld`.

6.8 Summary

In this chapter we have presented a software system for a quadruped robot that allows it to quickly and reliably negotiate a wide variety of challenging terrain, using both static and dynamic modes of locomotion. The techniques we apply to this task, in particular the constant use of rapid recovery and replanning methods, are motivated by the inability to develop an accurate model of the LittleDog system. Despite this difficulty in modeling, we show our system is able to quickly and robustly cross a wide variety of challenging terrains. While we have developed a substantial system for this work, capable of navigating terrains well beyond what quadrupeds robots could handle when this work began, the research has also lead to new topics and directions.

Advanced robot hardware and compliance. Although the LittleDog is a capable robotic platform, other legged robots have significantly greater physical capabilities. Compliant legs, in particular, able to store energy from impacts and providing a purely mechanical means of adjusting to some level of irregularity, offer a large potential advantage over the relatively stiff legs of LittleDog. An important topic for future work involves how to extend the careful planning methods developed on the LittleDog to these more compliant and mechanically robust robotic systems. And while these additions further the mechanical capability of the robot, they can be correspondingly harder still to model accurately, and thus methods that function with inaccurate models are a priority here.

Onboard vision. Related to the goal of bringing these robots into the real world, we do of course have to recognize that most of the work we present in this chapter has been conducted in a highly idealized setting, where we have full knowledge

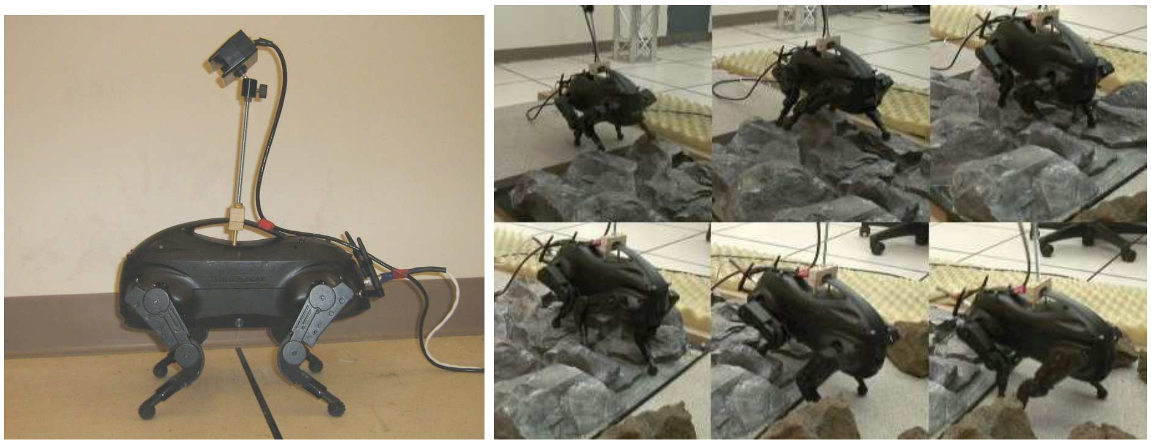


Figure 6.14: Prototype system with onboard stereo camera, and snapshots of the system crossing one of the more difficult terrains, using only onboard vision.

of the terrain at run-time, and nearly flawless sensing. If the robots are to move outside the lab, we need vision which is completely on-board. We have pursued this topic a fair amount in tandem with the research presented here, and in Kolter et al. (2009) we present a prototype of a stereo vision system for the LittleDog, which is able to cross one of the more challenging terrains from the Learning Locomotion project (the Phase II rocks terrain, used as the challenging terrain in Section 6.5) using only onboard vision for both localization and mapping. Figure 6.14 shows this prototype system, as well as snapshots of it crossing this terrain. However, the robot admittedly moves much slower and less robustly when it uses only onboard vision, and a major direction for future research involves increasing the capability of vision systems such that they can compete with the offboard vision systems we have used for the majority of this work. In addition, such systems could be integrated with the existing system to provide a smooth degradation in the quality of perception, making it possible to determine with greater precision how trade-offs in perception accuracy affect performance. Finally, this work again raises the issue of inaccurate models, as imperfect knowledge of the terrain near the robot corresponds exactly to an inaccurate or uncertain model of the robot's dynamics; thus, techniques that can handle such imperfect information are key to this work.

Chapter 7

Conclusion

A key challenge in applying model-based Reinforcement Learning and optimal control methods to complex dynamical systems, such as those arising in many robotics tasks, is the difficulty of obtaining an accurate model of the system. These algorithms perform very well when they are given or can learn an accurate dynamics model, but often times it is very challenging to build an accurate model by any means: effects such as hidden or incomplete state, dynamic or unknown system elements, and other effects, can render the modeling task very difficult.

This work has proposed a number of algorithms and empirical demonstrations for dealing with such situations. In particular, we have develop three algorithms for exploiting inaccurate models in different manners. We presented the Policy Gradient with the Signed Derivative algorithm, an approximate policy gradient method that enables us to learn policy parameters using a model that must only capture the correct signs of certain matrix derivative terms. We also presented a dimensionality reduction method for policy search, which uses a distribution over inaccurate models to identify a linear subspace of controllers, then learns an element from this subspace on the real system. Finally, we developed a probabilistic method for combining inaccurate models and observed trajectories, using a method we call Multi-model LQR, to achieve a mixture of open-loop and closed-loop behavior for control tasks where we cannot model the system accurately in certain regions.

In addition to these algorithms, a key contribution of this work has been the application of these methods to challenging tasks in robotic control. In particular, several of the algorithms we present here were applied to the LittleDog robot, a small quadruped robot, and have enabled it to cross a wide variety of complex terrain, including jumping over large obstacles, and quickly following complex motion paths. We also use the methods to enable a full-sized autonomous car to perform a "power-slide" maneuvers, accurately skidding the car sideways into a narrow parking spot. In both cases, the applications demonstrate state-of-the-art results on these challenging control tasks.

Looking forward, several key challenges emerge as directions for future work. Algorithmically, one of the key challenges for such methods is finding the proper integration with learning or adaptive control approaches. While the work in this thesis avoided the issue of directly learning a model, or tried to exhaust all options for model-building before applying the described algorithms, the fields of machine learning for control and adaptive control have also shown great abilities to control and model complex systems. While this work has specifically looked at the question of what can be done without learning any accurate model (and rather learning an control policy directly, in many cases), the most powerful techniques are likely to come from an iterative combination of both using these algorithms to address those model elements that truly cannot be learned, and adaptive control or machine learning techniques to address those elements that can be modeled.

From an application standpoint, there remains many challenging problems in both quadruped locomotion and "extreme" autonomous driving. One of the primary challenges with the LittleDog robot is that it was not intended for the high-speed dynamic maneuvers that we focus on for much of this work. Indeed, more capable robotic systems will both greatly enhance the ability of quadruped robots, and also will likely require yet more methods for dealing with inaccurate models, as the additional mechanical complexity only compounds the difficulties in modeling. Likewise, accurate control of autonomous vehicles in all modes at the limits of handling remains a challenging open problem, and one which will likely require a combination of advanced modeling techniques, along with algorithms that can deal with inaccurate models to overcome these difficulties.

Bibliography

Abbeel, P. and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Abbeel, P. and Ng, A. Y. (2005). Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Abbeel, P., Quigley, M., and Ng, A. Y. (2006). Using innaccurate models in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Anderson, B. D. O. and Moore, J. B. (1989). *Optimal Control: Linear Quadratic Methods*. Prentice-Hall.

Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts.

Ando, R. K. and Zhang, T. (2005). A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853.

Anthony, M. and Bartlett, P. L. (1999). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.

Argyriou, A., Evgeniou, T., and Pontil, M. (2006). Multi-task feature learning. In *Neural Information Processing Systems*.

Argyriou, A., Micchelli, C. A., Pontil, M., and Ying, Y. (2007). A spectral regularization framework for multi-task structure learning.

Armijo, L. (1966). Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*, 16(1):1–3.

Astrom, K. J. and Wittenmark, B. (1994). *Adaptive Control*. Prentice Hall.

Atkeson, C. G. and Santamaria, J. C. (1997). A comparison of direct and model-based reinforcement learning. In *International Conference on Robotics and Automation*.

Atkeson, C. G. and Schaal, S. (1997). Learning tasks from a single demonstration. In *Proceedings of the International Conference on Robotics and Automation*.

Bagnell, J. A., Kakade, S., Ng, A. Y., and Schneider, J. (2004). Policy search by dynamic programming. In *Neural Information Processing Systems 16*.

Bagnell, J. A. and Schneider, J. (2003). Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Bares, J. and Wettergreen, D. (1999). Dante II: Technical description, results and lessons learned. *International Journal of Robotics Research*, 18(7):621–649.

Bertsekas, D. P. (2005a). *Dynamic Programming and Optimal Control, Vol I*. Athena Scientific.

Bertsekas, D. P. (2005b). *Dynamic Programming and Optimal Control, Vol II*. Athena Scientific.

Betts, J. (1998). Survery of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207.

Boyd, S. and Vandenberg, L. (2004). *Convex Optimization*. Cambridge University Press.

Boyd, S. P. (2003). Linear quadratic regulator: Discrete-time finite horizon. Lecture Notes.

Byl, K., Shkolnik, A., Prentice, S., Roy, N., and Tedrake, R. (2008). Reliable dynamic motions for a stiff quadruped. In *Proceedings of the 11th International Symposium on Experimental Robotics*.

Cao, B., Dodds, G., and Irwin, G. (1997). Constrained time-efficient smooth cubic spline trajectory generation for industrial robots. In *Proceedings of the IEE Conference on Control Theory and Applications*.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Chestnutt, J., Kuffner, J., Nishiwaki, K., and Kagami, S. (2003). Planning biped navigation strategies in complex environments. In *Proceedings of the International Conference on Humanoid Robotics*.

Christopher G. Atkeson, Andrew W. Moore, S. S. (1997). locally weighted learning for control. *artificial intelligence review*, 11:75–113.

Dasgupta, S. and Johnson, C. R. (1986). Some comments on the behavior of sign-sign adaptive identifiers. *Systems and Control Letters*, 7:75–82.

Doya, K., Samejima, K., ichi Katagiri, K., and Kawato, M. (2002). Multiple model-based reinforcement learning. *Neural Computation*, pages 1347–1369.

Dyer, P. and McReynolds, S. R. (1970). *The Computation and Theory of Optimal Control*. Academic Press.

Fodor, I. (2002). A survey of dimension reduction techniques. Technical report, US DOE Office of Scientific and Technical Information.

Fukuoka, Y., Kimura, H., and Cohen, A. H. (2003). Adaptive dynamic walking of a quadruped robot on irregular terrain based on biological concepts. *The International Journal of Robotics Research*, 22:187–202.

Gerdes, C. (2009). Personal communication.

Gillula, J., Huang, H., Vitus, M. P., and Tomlin, C. J. (2010). Design of guaranteed safe maneuvers using reachable sets: Autonomous quadrotor aerobatics in theory and practice. In *Proceedings of the International Conference on Robotics and Automation*.

Glynn, P. (1987). Likelihood ratio gradient estimation: an overview. In *Proceedigns of the 1987 Winter Simulation Conference*.

Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5:1471–1530.

Grillner, S. (1985). Neurobiological bases of rhythmic motor acts in vertebrates. *Science*, 228(4696):143–149.

Hansen, E., Barto, A., and Zilberstein, S. (1996). Reinforcement learning for mixed open-loop and closed-loop control. In *Neural Information Processing Systems*.

Hengst, B., Ibbotson, D., Pham, S. B., and Sammut, C. (2002). Omnidirectional locomotion for quadruped robots. In *RoboCup 2001: Robot Soccer World Cup V*, pages 368–373.

Hirose, S., Nose, M., Kikuchi, H., and Umetani, Y. (1984). Adaptive gait control of a quadruped walking vehicle. *International Journal of Robotics Research*, 1:253–277.

Hodgins, J. K. and Raibert, M. H. (1990). Biped gymnastics. *International Journal of Robotics Research*, 9(2):115–128.

Hoffmann, G. M., Tomlin, C. J., Montemerlo, M., and Thrun, S. (2007). Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In *Proc. 26th American Control Conf.*

Hornby, G. S., Takamura, S., Yamamoto, T., and Fujita, M. (2005). Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410.

Hsu, Y.-H. J. and Gerdes, J. C. (2005). Stabilization of a steer-by-wire vehicle at the limits of handling using feedback linearization. In *Proceedings of the 2005 ASME International Mechanical Engineering Congress and Exposition*.

Ioannou, P. A. and Sun, J. (1995). *Robust Adaptive Control*. Prentice Hall.

Jacobson, D. H. and Mayne, D. Q. (1970). *Differential Dynamic Programming*. American Elsevier.

Jordan, M. I. and Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354.

Kakade, S. (2001). A natural policy gradient. In *Neural Information Processing Systems 14*.

Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilisitc roadmaps for path planning high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580.

Kearns, M. and Singh, S. (2002). Near-optimal reinforcment learning in polynomial time. *Machine Learning*, 49(2–3):209–232.

Khatri, C. G. and Mitra, S. K. (1976). Hermitian and nonnegative definite solutions of linear matrix equations. *SIAM Journal on Applied Mathematics*, 31(4):579–585.

Kimura, H., Fukuoka, Y., and Cohen, A. H. (2007). Adaptive dynamic walking of a quadruped robot on natural ground based on biological concepts. *The International Journal of Robotics Research*, 26(5):475–490.

Ko, J., Klein, D. J., Fox, D., and Hhnel, D. (2007). Gaussian processes and reinforcement learning for identification and control of an autonomous blimp. In *International Conference on Robotics and Automation*.

Kohl, N. and Stone, P. (2004). Machine learning for fast quadrupedal locomotion. In *The Nineteenth Conference on Artificial Intelligence*.

Kolter, J. Z., Abbeel, P., and Ng, A. Y. (2008a). Hierarchical apprenticeship learning, with application to quadruped locomotion. In *Neural Information Processing Systems 20*.

Kolter, J. Z., Kim, Y., and Ng, A. Y. (2009). Stereo vision and terrain modeling for quadruped robots. In *Proceedings of the International Conference on Robotics and Automation*.

Kolter, J. Z. and Ng, A. Y. (2007). Learning omnidirectional path following using dimensionality reduction. In *Robotics Science and Systems*.

Kolter, J. Z. and Ng, A. Y. (2009a). Policy search via the signed derivative. In *Proceedings of Robotics: Science and Systems*.

Kolter, J. Z. and Ng, A. Y. (2009b). Task-space trajectories via cubic spline optimization. In *Proceedings of the International Conference on Machine Learning*.

Kolter, J. Z., Plagemann, C., Jackson, D. T., Thrun, S., and Ng, A. Y. (2010). A probabilistic approach to mixed open-loop and closed-loop control, with application to extreme autonomous driving. In *Proceedings of the International Conference on Robotics and Automation*.

Kolter, J. Z., Rodgers, M. P., and Ng, A. Y. (2008b). A complete control architecture for quadruped locomotion over rough terrain. In *Proceedings of the International Conference on Robotics and Automation*.

Konidaris, G. and Barto, A. (2006). Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Krotkov, E., Simmons, R., and Whittaker, W. L. (1995). *Ambler: Performance of a Six-Legged Planetary Rover*, 35(1):75–81.

LaValle, S. M. and Kuffner, J. J. (1999). Randomized kinodynamic planning. In *Proceedings of the International Conference on Robotics and Automation*.

Li, H., Liao, X., and Carin, L. (2009). Multi-task reinforcement learning in partially observable stochastic environments. *Journal of Machine Learning Research*, 10:1131–1186.

Li, W. and Todorov, E. (2005). Iterative linear qaudratic regulator design for nonlinear biological movement systems. In *Proceedings of the International Conference on Informatics in Control*.

Lin, C.-S., Chang, P.-R., and Luh, J. (1983). Formulation and optimization of cubic polynomial joint trajectories for industrial robots. *IEEE Transactions on Automatic Control*, 28(12):1066–1074.

Ljung, L. (1999). *System Identification: Theory for the User*. Prentice Hall.

Lucky, R. W. (1966). Techniques for adaptive equalization of digital communication systems. *Bell Systems Technical Journal*, 45:255–286.

Mahadevan, S. and Maggioni, M. (2007). Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8:2169–2231.

McGhee, R. (1985). Vehicular legged locomotion. In *Advances in Automation and Robotics*, pages 259–284.

McGhee, R. B. (1967). Finite state control of quadruped locomotion. *Simulation*, 5:135–140.

McGhee, R. B. (1968). Some finite state aspects of legged locomotion. *Mathematical Biosciences*, 2:67–84.

McGhee, R. B. and Frank, A. A. (1968). On the stability properties of quadruped creeping gaits. *Mathematical Biosciences*, 3:331–351.

Mehta, N., Natarajan, S., Tadepalli, P., and Fern, A. (2008). Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, 73(3):289–312.

Montemerlo, M., Becker, J., Bhat, S., Dahlkamp, H., Dolgov, D., Ettinger, S., Haehnel, D., Hilden, T., Hoffman, G., Huhnke, B., Johnston, D., Klumpp, S., Langer, D., Levandowski, A., Levinson, J., Marcil, J., Orenstein, D., Paefgen, J., Penny, I., Petrovskaya, A., Pflueger, M., Stanek, G., Stavens, D., Vogt, A., and Thrun, S. (2008). Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597.

Moore, K. L. (1999). Iterative learning control: an expository overview. *Applied and Computational Controls, Signal Processing, and Circuits*, 1(1):151–214.

Mosher, R. S. (1968). Test and evaluation of a versatile walking truck. In *Proceedings of off-road mobili6ty research symposium*, pages 359–379.

Murphy, M. P., Saunders, A., Moreira, C., Rizzi, A. A., and Raibert, M. (2010). The littleDog robot. *To appear in The International Journal of Robotics Research*.

Murray-Smith, R. and Johansen, T. A. (1997). Taylor and Francis.

Narendra, K. S. and Balakrishnan, J. (1997). Adaptive control using multiple models. *IEEE Transactions on Automatic Control*, pages 171–187.

Neu, G. and Szepesvári, C. (2007). Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proceedings of Uncertainty in Artificial Intelligence*.

Ng, A. Y. and Jordan, M. (2000). Pegasus: A policy search method for large mdps and pomdps. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Nichol, J. G., Singh, S. P., Waldron, K. J., III, L. R. P., and Orin, D. E. (2004). System design of a quadrupedal galloping machine. *International Journal of Robotics Research*, 23(10–11):1013–1027.

Park, J.-H., Kim, H.-S., and Choi, Y.-K. (1997). Trajectory optimization and control for robot manipulator using evolution strategy and fuzzy logic. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*.

Peters, J. and Schaal, S. (2006). Policy gradient methods for robotics. In *Proceedings of the IEEE Conference on Intelligent Robotics Systems*.

Peters, J. and Schall, S. (2004). Learning motor primatives with reinforcement learning. In *Proceedings of the 11th Joint Symposium on Neural Computation.*

Peters, J., Vijayakumar, S., and Schaal, S. (2005). Natural actor-critic. In *Proceedings of the European Conference on Machine Learning*.

Poulakakis, I., Smith, J. A., and Buehler, M. (2005). Modeling and experiments of untethered quadrupedal running with a bounding gait: The scout II robot. *The International Journal of Robotics Research*, 24(4):239–256.

Putterman, M. L. (2005). *Markov Decision Processes: Discrete Stochastic Dynamic Progamming*. Wiley Interscience.

Raibert, M., Blankespoor, K., Nelson, G., and Playter, R. (2008). Bigdog, the rough-terrain quadruped robot. In *Proceedings of the International Federation of Autonomous Control*.

Raibert, M. H. (1986). *Legged Robots that Balance*. MIT Press.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. The MIT Press.

Ratliff, N., Bagnell, J. A., and Zinkevich, M. (2006). Maximum margin planning. In *Proceedings of the International Conference on Machine Learning*.

Reinsel, G. C. and Velu, R. P. (1998). *Multivariate Reduced-Rank Regression: Theory And Appplications*. Springer-Verlag.

Riedmiller, M. and Braun, H. (1992). RPROP – a fast adaptive learning algorithm. In *Proceedings of the International Symposium on Computer and Information Sciences*.

Roy, N., Gordon, G., and Thrun, S. (2005). Finding approximate pomdp solutions through belief compression. *Journal of Artificial Intelligence Research*, 23:1–40.

Saranli, U., Buehler, M., and Koditschek, D. (2001). Rhex: A simple and highly mobile hexapod robot. *Int. Journal of Robotics Research*, 20:616–631.

Sastry, S. and Bodson, M. (1994). *Adaptive Control: Stability, Convergence, and Robustness*. Prentice-Hall.

Saunders, A., Goldman, D., Full, R., and Buehler, M. (2006). The RiSE climbing robot: Body and leg design. *Proc. SPIE Int. Soc. Opt. Eng.*, 6230:623017.

Schaal, S. (1994). Nonparametric regression for learning. In *Proceedings of the Conference on Adaptive Behavior and :earning*.

Schott, K. D. and Bequette, B. W. (1997). Multiple model adaptive control. In *Multiple Model Approaches to Modelling and Control*.

Sideris, A. and Bobrow, J. E. (2005). An efficicent sequential linear quadratic algorithm for solving nonlinear optimal control problems.

Stengel, R. F. (1994). *Optimal Control and Estimation*. John Wiley and Sons.

Stolle, M. and Atkeson, C. G. (2006). Policies based on trajectory libraries. In *International Conference on Robotics and Automation*.

Stolle, M., Tappeiner, H., Chestnutt, J., and Atkeson, C. G. (2007). Transfer of policies based on trajectory libraries. In *International Conference on Intelligent Robots and Systems*.

Strehl, A. L., Li, L., and Littman, M. (2009). Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research*, 10:2413–2444.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Tanaka, F. and Yamamura, M. (2003). Multitask reinforcement learning on the distribution of mdps. In *Proceedings of the International Symposium on Computational Intelligence in Robotics and Automation*.

Taskar, B., Chatalbashev, V., Koller, D., and Guestrin, C. (2005). Learning structured prediction models: A large margin approach. In *Proceedings of the International Conference on Machine Learning*.

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685.

Taylor, M. E., Stone, P., and Liu, Y. (2007). Transfer learning via inter-task mappings for temporal different learning. *Journal of Machine Learning Research*, 8:2125–2167.

Thrun, S. (1996). Is learnging the n-th thing any easier than learning the first? In *Neural Information Processing Systems*.

Tsochantaridis, I., Joachims, T., Hofmann, T., and Altun, Y. (2005). Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 6:1453–1484.

Vaz, A. I. F. and Fernandes, E. M. (2006). Tools for robotic trajectory planning using cubic splines and semi-infinite programming. In Seeger, A., editor, *Recent Advances in Optimization*, pages 399–413. Springer.

Vijayakumar, S., D'Souza, A., and Schaal, S. (2005). Incremental online learning in high dimensions. *Neural Computation*, 17:2602–2634.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Williams, R. J. and Zisper, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280.

Wilson, A., Fern, A., Ray, S., and Tadepalli, P. (2007). Multi-task reinforcement learning: A hierarchical bayesian approach. In *Proceedings of the International Conference on Machine Learning*.

Zhou, K., Doyle, J., and Glover, K. (1996). *Robust and Optimal Controler*. Prentice Hall.