The Stanford LittleDog: A Learning and Rapid Replanning Approach to Quadruped Locomotion

J. Zico Kolter and Andrew Y. Ng

June 21, 2010

Abstract

Legged robots offer the potential to navigate a wide variety of terrains that are inaccessible to wheeled vehicles. In this paper we consider the planning and control tasks of navigating a quadruped robot over a wide variety of challenging terrain, including terrain which it has not seen until run-time. We present a software architecture that makes use of both static and dynamic gaits, as well as specialized dynamic maneuvers, to accomplish this task. Throughout the paper we highlight two themes that have been central to our approach: 1) the prevalent use of learning algorithms, and 2) a focus on rapid recovery and replanning techniques; we present several novel methods and algorithms that we developed for the quadruped and that illustrate these two themes. We evaluate the performance of these different methods, and also present and discuss the performance of our system on the official Learning Locomotion tests.

1 Introduction

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 [33]. Contrast this will 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 paper 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.

There is a long history of legged locomotion in robotics. Initial work in this field included a large human-operated quadruped [27], six-legged robots able to cross small amounts of rough terrain [26, 11], a quadruped that could climb stairs [13], and a variety of "hopping" robots that could quickly cross flat ground [33]. While these robots all had significant limitations, they showcased the potential of legged machines, and spawned a great deal of the future work in this field. In more recent years there has been a great deal of subsequent work on legged machines, including large-scale legged robots designed for planetary and volcanic exploration [21, 2], work with learning fast gaits for the Sony Aibo quadruped and the associated RoboCup competition (e.g., [12, 17, 14]), biologically inspired quadruped gaits based on periodic motion (e.g., [9, 16, 5]), and quadruped robots intended to quickly cross a variety of rough ground [32]. Although these works represent a significant advance in terms of speed and maneuverability compared to their predecessors, these robots still operate in situations where the terrain features are relatively small compared to the robot's legs.

In contrast, there has also in recent years been a number of biologically inspired robots that use clever mechanical design to cross highly irregular terrain even with minimal sensing. For example, the Rhex robot [35] uses six flexible spinning legs, and can cross rocky terrain with simple open-loop control. Similarly, the RiSE [36] and StickyBot robots [15] can scale vertical walls using feet with microspines and adhesive "Gecko-like" feet respectively. These robots can achieve extremely impressive results on the terrains for which they were intended, but can perform poorly in situations where careful footstep planning or significant terrain adaptation is needed.

In this paper we focus on the former line of research, but consider obstacles and terrain irregularities that are far larger (relative to the size of the robot's legs) than what has been dealt with in previous work, and situations where careful footstep planning is a necessity. We present a software system that is capable of navigating a small quadruped robot over a variety of challenging and previously unseen terrains. This work was conducted under the DARPA Learning Locomotion program, and we will present and discuss the official results of the program along with our own internal experiments on the software platform we developed.

While the complete system that we present naturally contains a large number of components (which we will discuss in detail), two general themes have guided our overall approach to this problem, and which have broad applicability well beyond quadruped locomotion alone. The first of these themes is the ubiquitous use of learning algorithms. We have found that learning algorithms are particularly important in generalizing from specialized solutions that would work in a particular setting to general behavior that are robust to changes in the environment. In many situations it was possible to hand engineer an approach that would successfully cross one particular obstacle given the ideal starting conditions, but to extend this to a system that could work on different obstacles, from different starting conditions, we employed a number of learning approaches that greatly improved the system.

The second theme in this work is the constant use of rapid recovery and replanning methods. Often motion planning tasks in robotics are framed as lengthy search problems, where an algorithm first determines a plan, then executes it on the robot. However, in a complex domain like quadruped locomotion, much can go wrong with this approach: the robots feet can slip, it may hit its knee unexpectedly, and the original plan may be rendered invalid. Thus, 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.

These two principles form the general basis for our approach to the Learning Locomotion project, and the goal of this paper is to describe that system, highlighting the novel learning and rapid replanning elements, and showing how they improve performance of the system. The remainder of this paper is organized as follows: In Section 2, we give a brief overview of work on legged locomotion in general, and the Learning Locomotion project and LittleDog robot in particular. In Section 3 we then present our software architecture for quadruped locomotion; this section is intended as a high-level overview of our entire software system. Then, in Sections 4–7 we look specifically at several novel algorithms and methods developed for this program, highlighting the above themes of learning and rapid replanning. Finally, in Section 8 we present and analyze the official results for the DARPA Learning Locomotion program, and finish in Section 9 with future directions and conclusions.

2 Background

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 [27]. 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 [23, 25, 24]. 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 [26]. Another milestone in the development of legged robots came with a series of robots (known as the TITAN robots), developed by Hirose et al. [13], 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 [33]. 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 [29], the Scout II robot [31], and the BigDog robot [32]; 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 [21] and Dante II [2] 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 [12, 14, 17], 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 [10]. 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 [9, 16]. However, again these robots typically can only handle a relatively small amount of irregularity in the terrain, not the large obstacles that we consider in this paper.

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 locomotion. Such work includes the Rhex hexapedal robot [35], which uses six flexible legs to rapidly move over relatively large obstacles, and the RiSE climbing robot [36], 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.

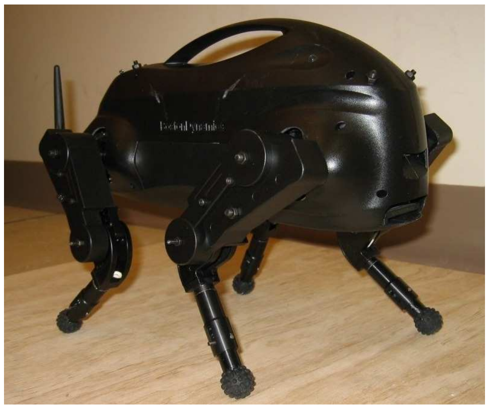


Figure 1: The LittleDog robot, designed and built by Boston Dynamics, Inc.

2.2 The Learning Locomotion program and LittleDog robot

The robotic platform for this the DARPA Learning Locomotion program was the LittleDog robot, shown in Figure 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 joints, 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 [32].

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

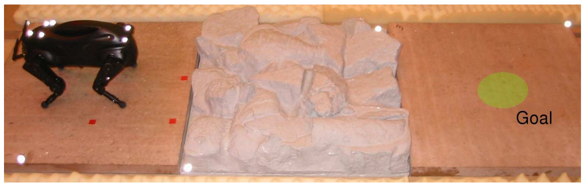


Figure 2: Typical quadruped locomotion task: navigate robot over terrain to goal.

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 for vision. We will present these results in later sections, 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.

The program was divided into three phases, each with go/no-go tests that involved crossing terrains of specified height, at a specified speed. Six teams participated in the first two phases of the program, and five in the last phase. The system we describe in this paper is the final system produced for the "Phase Three" testing, but several portions were developed in their entirely for previous phases, and we will note this in their descriptions.

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 2. Given the complex obstacle shapes and high dimensionality of the state space (36D, 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 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,

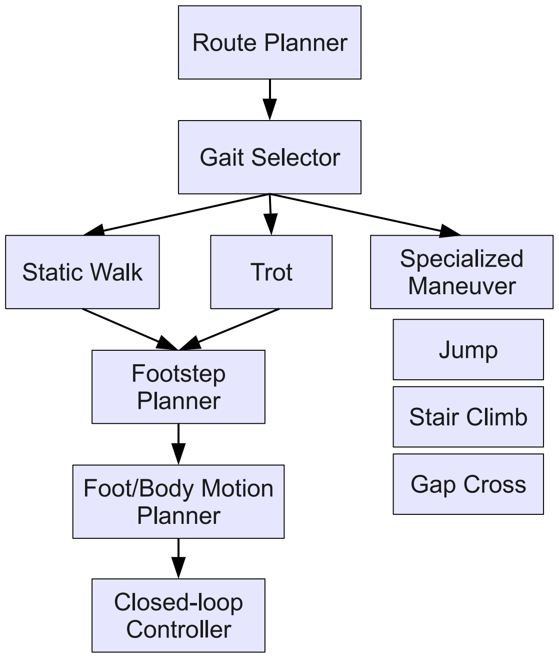


Figure 3: Hierarchical software architecture for quadruped planner and controller.

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.

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 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 million states) to find a 2D path across the terrain. Since the

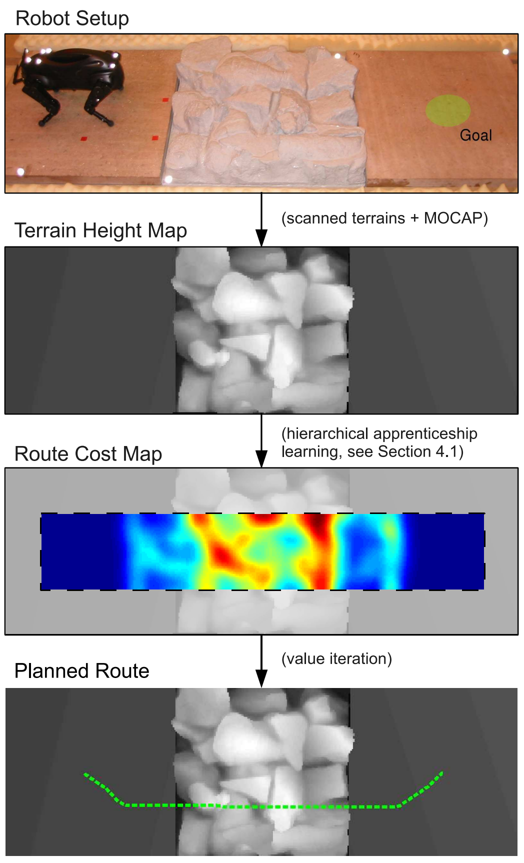

Robot Setup
Goal
(scanned terrains + MOCAP)
Terrain Height Map
(hierarchical apprenticeship learning, see Section 4.1)
Route Cost Map
(value iteration)
Planned Route

Figure 4: Overview of route planning element.

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 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:

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

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., [8]) 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.

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.

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) withing 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.

3.3.1 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

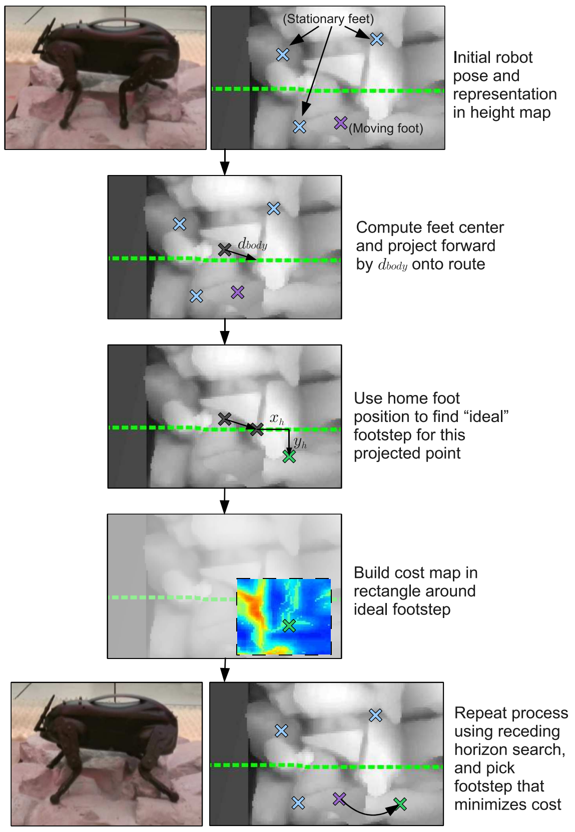

(Stationary feet)
(Moving foot)
Initial robot pose and representation in height map
d_{body}
Compute feet center and project forward by d_{body} onto route
x_h
y_h
Use home foot position to find "ideal" footstep for this projected point
Build cost map in rectangle around ideal footstep
Repeat process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

Figure 5: Overview of the footstep planning process.

benefit of maximizing the kinematically feasible distance that each foot can move — for more detailed discussion of the benefits of this pattern, see [25].

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 we will describe in Section 6 how we learn this footstep cost simultaneously with the body cost mentioned above. The complete planning system is illustrated in Figure 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. While this is a fairly straightforward principle, we know of no explicit mention of this fact in the quadruped locomotion literature, so we prove this convergence property in Appendix A. 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.

3.3.2 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 5.

3.3.3 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 4.

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 even maintain 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.

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).

3.5.1 Front Leg Jumping

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, which was first demonstrated by the MIT Learning Locomotion team [6], is to shift the robot's COG backwards until front legs become unloaded 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 show in Figure 6.

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 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 (a strategy employed by most of the Learning Locomotion teams), 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 employ a novel policy search algorithm, called the Signed Derivative, to learn the parameters of this policy from experience. This algorithm and application are described in Section 7.

3.5.2 Back Leg Climbing and Other Maneuvers

Another typical maneuver that we use is a "back leg climb," which raises both back legs simultaneously on to a step — a similar maneuver also crosses the back legs over a gap, or over a barrier. Unlike the front jump, these are static maneuvers, and for these tasks we typically balance for a small portion of the robot's "nose" or body. We conducted preliminary experiments with learning parameters for the back climb and other maneuvers using the same method as for

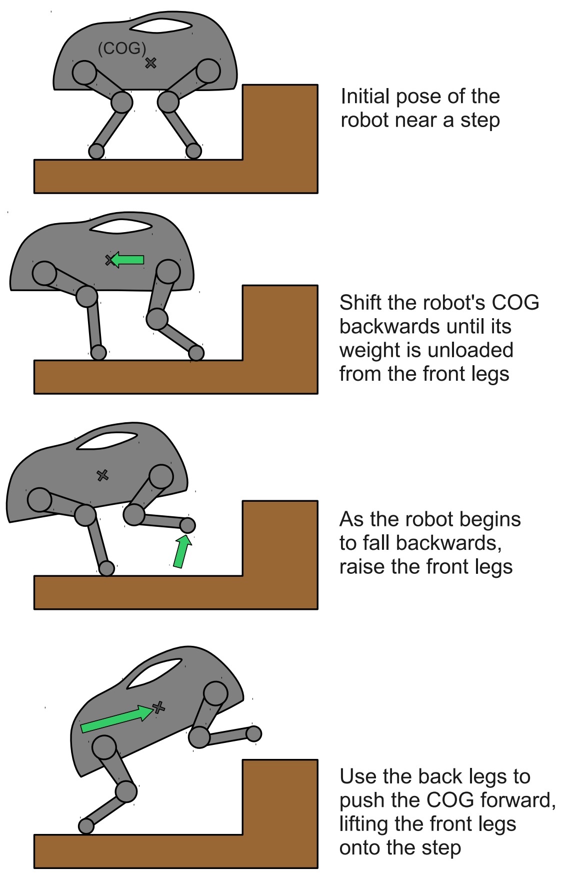

(COG)
Initial pose of the robot near a step
Shift the robot's COG backwards until its weight is unloaded from the front legs
As the robot begins to fall backwards, raise the front legs
Use the back legs to push the COG forward, lifting the front legs onto the step

Figure 6: Overview of the front jump maneuver.

the front jump, but this wasn't nearly as crucial for this maneuver, as they are purely kinematic maneuvers and less sensitive to parameter choice. Thus, while these specialized maneuvers play an important role in the overall performance of our system on the Learning Locomotion terrains, we don't discuss them further in this paper.

3.6 Introduction to Learning and Rapid Replanning Methods

This section has described our software system for the quadruped robot, but we deferred discussion of several novel methods, related both to learning and rapid replanning, that we have developed over the course of this program. In the remainder of this paper we will present each of these methodologies. In particular, we present and analyze four 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, 3) a method for Hierarchical Apprenticeship Learning, used to learn the cost functions for route and footstep planning and 4) a policy search method known as the Signed Derivative, applied to learning dynamic maneuvers.

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.

4.1 Control Elements

Stability Detection and Recovery. Recall that (ignoring friction effects, which do plan 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

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

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

$$(1 - \alpha) feet + \alpha T_{des}^{-1} T_{cur} feet \tag{2}$$

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 (double) 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 avoid 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.

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 maps are shown in Table 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 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

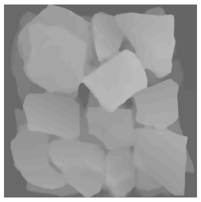
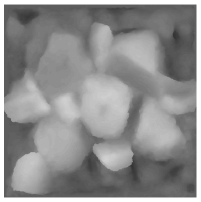
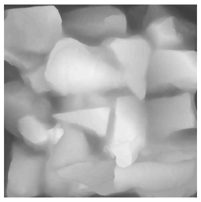
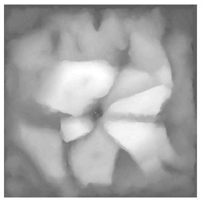
Terrain #	1	2	3	4
Max Height	6.4 cm	8.0 cm	10.5 cm	11.7 cm
Picture				
Height map				

Table 1: The four terrains used for evaluation.

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

#3 and #4, the advantage of using all the control elements is clear.[2]

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 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

[2]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.

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.

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 [22]. 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 [3] 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 [4] — 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.

5.1 The Cubic Spline Optimization Algorithm

5.1.1 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

$$(t_0, x_0^\star), (t_1, x_1^\star), \ldots, (t_T, x_T^\star) \tag{3}$$

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

$$x_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3 \tag{4}$$

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

$$x(t) = \begin{cases} x_0(t) & \text{if } t_0 \le t < t_1 \\ \vdots & \\ x_{T-1}(t) & \text{if } t_{T-1} \le t \le t_T \end{cases} \tag{5}$$

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.[3] To compute these coefficients, we first define the matrices $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^{T+1 \times n}$

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} x_0^\star & x_1^\star & \cdots & x_T^\star \end{bmatrix}^T & (6) \\ \mathbf{a} &= \begin{bmatrix} a_0 & a_1 & \cdots & a_T \end{bmatrix}^T & (7) \end{aligned}$$

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

$$\begin{aligned} \mathbf{a} &= \mathbf{x} & (8) \\ \mathbf{H}_1\mathbf{b} &= \mathbf{H}_2\mathbf{x} & (9) \\ \mathbf{c} &= \mathbf{H}_3\mathbf{x} + \mathbf{H}_4\mathbf{b} & (10) \\ \mathbf{d} &= \mathbf{H}_5\mathbf{x} + \mathbf{H}_6\mathbf{b} & (11) \end{aligned}$$

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 omit the full definitions here; explicit formulas for the matrices as used here are available in the appendix of [**?**]. However, the important point to gleam 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 $O(T)$.

[3] 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.

5.1.2 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

$$(t_0, \hat{x}_0), (t_1, \hat{x}_1), \ldots, (t_T, \hat{x}_T). \tag{12}$$

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, $f : \mathbb{R}^{(T+1)\times n} \rightarrow \mathbb{R}$ 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.

5.1.3 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 (8) – (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

$$\mathbf{H_1}\dot{\mathbf{x}} = \mathbf{H_2}\mathbf{x} \tag{13}$$

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 (4) and (5),

$$x(t') = a_i + b_i(t' - t_i) + c_i(t' - t_i)^2 + d_i(t' - t_i)^3. \tag{14}$$

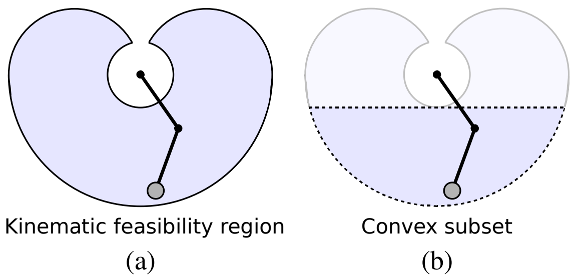


Figure 7: Illustration of kinematic feasibility constraints. (a) Kinematically feasible region for $q_1 \in [-\pi/2, \pi/2]$, $q_2 \in [-2.6, 2.6]$. (b) Convex subset of the feasible region.

But from (8)–(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.[4] Space constraints prohibit providing the complete definition of the following matrices, but 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, we can solve for these positions via a linear system

$$\mathbf{x}' = \mathbf{G}_1\mathbf{x} + \mathbf{G}_2\dot{\mathbf{x}} \tag{15}$$

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 7 (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 show in Figure 7 (a). Although this region is not convex, we can easily find a convex subset, such as the region shown in Figure 7 (b).

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 8. Here Figure 8 (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

[4]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.

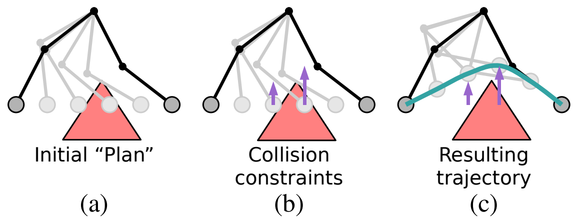


Figure 8: Illustration of collision constraints. (a) Initial (infeasible) plan. (b) Height constraints imposed to avoid collision with obstacle. (c) Resulting optimized cubic spline trajectory.

collision constraint, so we simply add the constraint, at each of these times, that the resulting waypoint must lie above the the obstacle by some margin; Figure 8 (b) shows this constraint, and Figure 8 (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 2, 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. 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

$$f(\dot{\mathbf{x}}, \dot{\mathbf{x}}') = \mathrm{tr}\, \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \mathrm{tr}\, \dot{\mathbf{x}}'^T \dot{\mathbf{x}}' \tag{16}$$

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 function better in other cases such as minimizing the maximum velocity, average or maximum acceleration, average distance

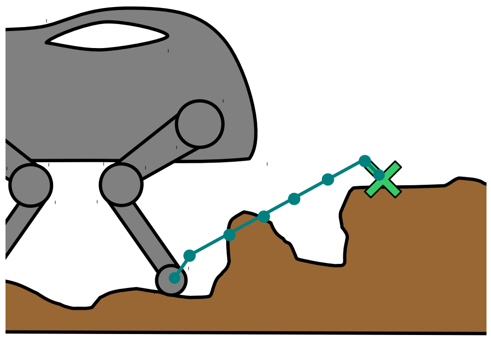


Figure 9: Foot planning task, and initial trajectory.

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 (8)–(11) involve non-linear, non-convex functions of the times. However, there has been previous work in approximately optimizing the times of cubic splines [7, 30, 38], and if the total time of the trajectory is ultimately the most important objective, these techniques can be applied.

5.2 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 we 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 location for the next step.

5.3 Planning Foot Trajectories

An illustration of the foot trajectory planning task is shown in Figure 9, 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:

$$
\begin{aligned}
\min_{\mathbf{x},\mathbf{x}',\dot{\mathbf{x}},\dot{\mathbf{x}}',\ddot{\mathbf{x}}} \quad & \operatorname{tr} \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \operatorname{tr} \dot{\mathbf{x}}'^T \dot{\mathbf{x}}' && (17) \\
\text{subject to} \quad & \mathbf{H}_1 \dot{\mathbf{x}} = \mathbf{H}_2 \mathbf{x} && (18) \\
& \ddot{\mathbf{x}} = \frac{1}{2}(\mathbf{H}_3 \mathbf{x} + \mathbf{H}_4 \dot{\mathbf{x}}) && (19) \\
& \mathbf{x}' = \mathbf{G}_1 \mathbf{x} + \mathbf{G}_2 \dot{\mathbf{x}} && (20) \\
& \dot{\mathbf{x}}' = \mathbf{G}_3 \mathbf{x} + \mathbf{G}_4 \dot{\mathbf{x}} && (21) \\
& \mathbf{x}_{0,:} = \hat{x}_0, \ \mathbf{x}_{T,:} = \hat{x}_T, \ \dot{\mathbf{x}}_{0,:} = 0, \ \dot{\mathbf{x}}_{T,:} = 0 && (22) \\
& \ddot{\mathbf{x}}_{t,x} \geq 0, \ \ddot{\mathbf{x}}_{t,y} \geq 0, \ \ t = 0, 1 && (23) \\
& \ddot{\mathbf{x}}_{t,x} \leq 0, \ \ddot{\mathbf{x}}_{t,y} \leq 0, \ \ t = T-1, T && (24) \\
& \left.\begin{array}{l} \ddot{\mathbf{x}}_{t,x} \geq \ddot{\mathbf{x}}_{t+1,x} \\ \ddot{\mathbf{x}}_{t,y} \geq \ddot{\mathbf{x}}_{t+1,y} \end{array}\right\} \ t = 2, \ldots, T-3 && (25) \\
& \ddot{\mathbf{x}}_{t,z} < 0, \ \ t = 2, \ldots, T-2 && (26) \\
& (\mathbf{x}_{t,x} - \mathbf{x}_{t+2,x})^2 + (\mathbf{x}_{t,y} - \mathbf{x}_{t+2,y})^2 \\
& \qquad \leq (3\text{cm})^2, \quad t = 0, T-2 && (27) \\
& \mathbf{x}'_{i,z} \geq \hat{x}_z(t'_i) + 2\text{cm}, \quad i = 1, \ldots, N && (28) \\
& \mathbf{x}_{t,z} \leq \max_{i=1,\ldots,N} \hat{x}_z(t'_i), \ \ t = 1, \ldots T. && (29)
\end{aligned}
$$

While there are many different terms in this optimization problem, the overall idea is straightforward. The optimization objective (17) is the squared velocity objective we discussed earlier; constraints (18)–(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;[5] (22) insures that the spline begins and ends at the start and goal, with zero velocity; (23) and (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 the start and end locations;[6] (25) extends the previous constraint slightly to also ensure that the x, y accelerations during the main trajectory portion are monotonic; (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; (27) ensures that the last waypoints in the ramp don't deviate from the start and end positions by more than 3cm; finally, (28) and (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. This particular set of objectives and constraints were developed specifically for the foot trajectory planning task, and many of

[5]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.

[6]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 constraints were developed over time in response to specific situations that caused simpler optimization problems to produced sub-optimal plans.

5.3.1 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 step (see, e.g., [20]. 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 the following optimization problem to plan the COG trajectory:

$$\begin{aligned}
\min_{\mathbf{x},\dot{\mathbf{x}},\dot{\mathbf{x}}'} \quad & \begin{array}{l} \operatorname{tr} \dot{\mathbf{x}}^T \dot{\mathbf{x}} + \operatorname{tr} \dot{\mathbf{x}'}^T \dot{\mathbf{x}}' \\ +\lambda \sum_{i=3}^{5} \operatorname{feas}(\mathbf{x}_{i,:}, x_{f_1}(t_i), f_1) \\ +\lambda \sum_{i=5}^{7} \operatorname{feas}(\mathbf{x}_{i,:}, x_{f_2}(t_i), f_2) \end{array} && (30) \\
\text{subject to} \quad & \mathbf{H}_1 \dot{\mathbf{x}} = \mathbf{H}_2 \mathbf{x} && (31) \\
& \dot{\mathbf{x}}' = \mathbf{G}_3 \mathbf{x} + \mathbf{G}_4 \dot{\mathbf{x}} && (32) \\
& \mathbf{x}_{0,:} = \hat{x}_0, \dot{x}_0,\ \mathbf{x}_{2,:} = \hat{x}_2,\ \mathbf{x}_{8,:} = \hat{x}_8 && (33) \\
& \dot{\mathbf{x}}_{0,:} = \dot{x}_{\text{init}} && (34) \\
& \mathbf{x}_{i,:} \in \mathcal{S},\quad i = 3, \ldots, 7 && (35)
\end{aligned}$$

where

$$\operatorname{feas}(x_{\text{body}}, x_{\text{foot}}, f) \qquad (36)$$

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 (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 (31) and (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; (33) ensures that the trajectory begins, enters the supporting triangle, and ends at the initially specified waypoints, while (34) ensures that the initial velocity of the spline is equal to the COG's current velocity; finally, (35) ensures that the COG waypoints will be inside the supporting triangle during the times that the feet are moving.

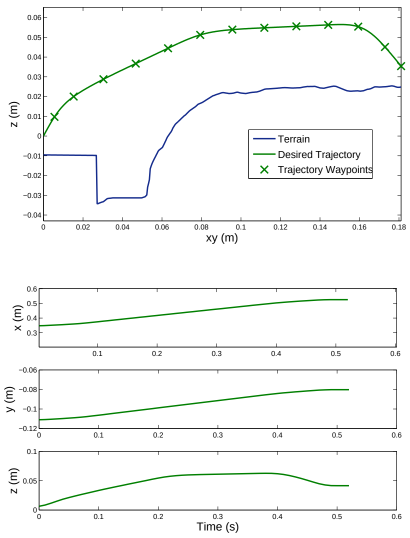


Figure 10: Typical example of a foot trajectory generated by the algorithm. The top figure shows the resulting trajectory in 3D space, while the bottom shows each component as a function of time.

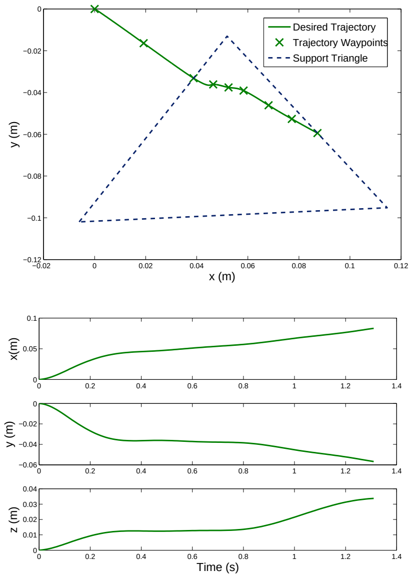


Figure 11: Typical example of a COG trajectory generated by the algorithm.

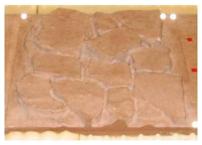
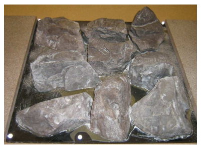
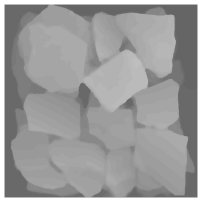
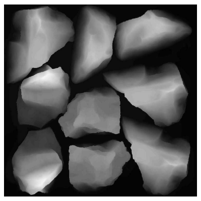
Terrain	1	2
Max Height	6.4 cm	10.4 cm
Picture		
Height map		

Table 3: The training and testing terrains used to evaluate HAL.

5.4 Experimental Evaluation

The cubic spline optimization procedure is used for all the experiments reported in Section 8, but here we evaluate this specific component of the system. We begin with a qualitative look at the trajectories generated by this method. Figure 10 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 11 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 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 3. We compare the cubic spline optimization approach to an older trajectory planning method described in [20], which uses simple box-shapes to step over terrain, and linear splines for moving the COG.

Table 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* be-

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

havior, 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 much 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 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 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 [1, 34, 28].

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.

6.1 The Hierarchical Apprenticeship Learning Algorithm

6.1.1 Definitions

To describe the hierarchical apprenticeship learning algorithm, we use the formalism of Markov Decision Processes (MDPs). 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. We use the notation MDP\C to denote an MDP minus the cost function. A policy π is a mapping from 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 $\phi : S \to \mathbb{R}^n$ 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

$$C(s) = w^T \phi(s) \tag{37}$$

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

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

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

6.1.2 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 $\psi : S_\ell \to S_h$ 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

$$\begin{aligned} C(s_h) &= \frac{1}{N(s_h)} \sum_{s_\ell \in \psi^{-1}(s_h)} C(s_\ell) \\ &= \frac{1}{N(s_h)} \sum_{s_\ell \in \psi^{-1}(s_h)} w^T \phi(s_\ell) = \frac{1}{N(s_h)} w^T \sum_{s_\ell \in \psi^{-1}(s_h)} \phi(s_\ell) \end{aligned} \tag{38}$$

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 (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.

6.1.3 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:

$$J^{(i)}(\pi_{h,E}^{(i)}) \leq J^{(i)}(\pi_h^{(i)}) \quad \forall \pi_h^{(i)}. \tag{39}$$

Equivalently, using (38), we can formulate this constraint as follows

$$w^T \mu_\phi^{(i)}(\pi_{h,E}^{(i)}) \leq w^T \mu_\phi(\pi_h^{(i)}) \quad \forall \pi_h^{(i)}. \tag{40}$$

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

$$C(s'_\ell) \leq C(s''_\ell) \tag{41}$$

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,

$$w^T \phi(s'_\ell) \leq w^T \phi(s''_\ell) \tag{42}$$

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

$$\begin{aligned} \min_{w,\eta\geq 0,\xi\geq 0} & \frac{1}{2}\|w\|_2^2 + \lambda_\ell \sum_{j=1}^{m} \xi^{(j)} + \lambda_h \sum_{i=1}^{n} \eta^{(i)} \\ \text{s.t.} & w^T \phi(s'^{(j)}_\ell) \leq w^T \phi(s''^{(j)}_\ell) - 1 + \xi^{(j)} \quad \forall s''^{(j)}_\ell, j \\ & w^T \hat{\mu}_\phi^{(i)}(\pi_{h,E}^{(i)}) \leq w^T \mu_\phi(\pi_h^{(i)}) - 1 + \eta^{(i)} \quad \forall \pi_h^{(i)}, i. \end{aligned} \tag{43}$$

where $\pi_h^{(i)}$ indexes over all high-level policies, η_i and ξ_i are slack variables, i indexes over all MDPs, $s''^{(j)}_\ell$ indexes over all states reachable from $s'^{(j)}_\ell$ 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 [37], subgradient algorithms [34], and constraint generation [?]. We use a method based on this last 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.

6.2 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

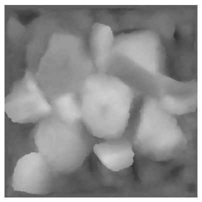
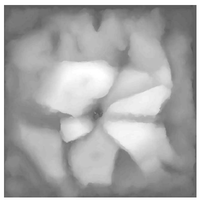
Terrain	Training	Testing
Max Height	8.0 cm	11.7 cm
Picture		
Height map		

Table 5: The training and testing terrains used to evaluate HAL.

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

Table 6: Execution times for different constraints on training and testing terrains. Dashes indicate that the robot fell over and did not reach the goal.

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 foot 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.

6.3 Experimental Evaluation

While the results that we present in Section 8 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 Ta-

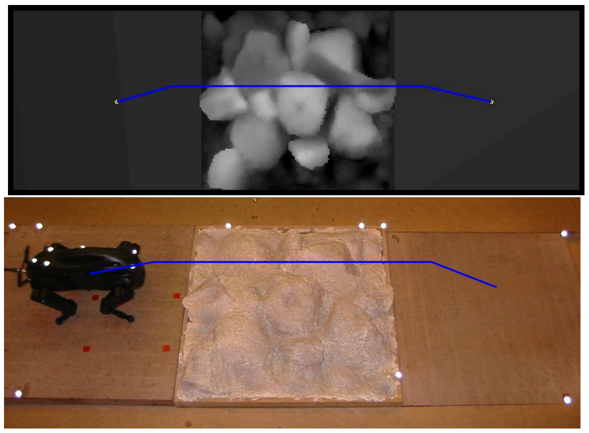


Figure 12: High-level (route) expert demonstration.

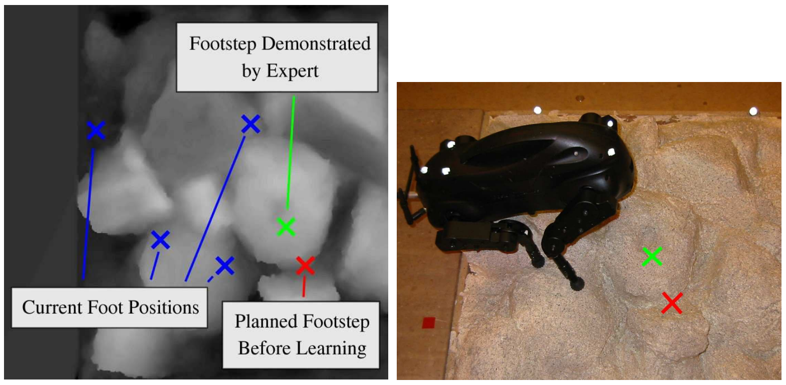

Footstep Demonstrated by Expert
Current Foot Positions
Planned Footstep Before Learning

Figure 13: Low-level (footstep) expert demonstration.

ble 5. To give advice at the high level, we specified complete body trajectories for the robot's center of mass, as shown in Figure 12. To give advice for the low level we looked for situations 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 13. 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 as 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 14 shows the route and footsteps taken

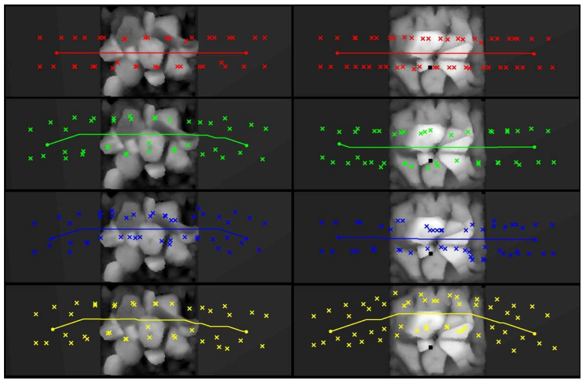


Figure 14: Body and footstep plans for different constraints on the training (left) and testing (right) terrains: (First/Red) No Learning, (Second/Green) HAL, (Third/Blue) Path Only, (Fourth/Yellow) Footstep Only.

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 shows the crossing times for each of the different types of constraints. As shown, the HAL algorithm outperforms 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.

7 Dynamic Maneuvers via the Signed Derivative

Finally, in this section we present a policy search method for learning dynamic maneuvers on the robot, in particular applied to learning parameters for the front leg jump, described in Section 3. Recall that in order to perform a front leg jump, we need to shift the robot's COG backwards to unload the front legs, then shift the COG forwards to catch the robot before it falls backwards. However, this is a delicate maneuver, and we need to know how far to shift the robot backwards given the initial conditions, or the jump will likely fail. Furthermore, building a dynamics model that is accurate enough to reliably predict such properties is a highly non-trivial task.

The intuition behind the signed derivative approach is that even if we do not have an accurate model of the system, in many cases the relationship between controls and future states is "obvious," and this can be exploited by a policy search algorithm. For example, in the case of the front jump maneuver, suppose we have some initial policy that predicts how much to shift the COG for a jump, based on the initial position of the robot; now suppose that we run this policy on the robot, and the robot flips backwards when trying to jump. Since we have no model of the system, we may not know the exact amount that we should have shifted for an ideal jump, but we do know that we likely shifted back too far, causing the robot to flip. These "obvious" control/state relationships formally correspond to knowing the *sign* of certain derivatives in the dynamics model, and we show how to exploit this knowledge to build a policy search algorithm.

7.1 The Signed Derivative Policy Search Algorithm

7.1.1 Definitions

Again we will use same formalism and notation for Markov Decision Processes, described in the previous section. In addition, as we are focused on the policy-search setting in this section, we will consider policies that are parametrized by some set of parameters θ — we use the notation $u = \pi_\theta(s)$ to denote the policy π, parametrized by θ, evaluated at state s. In particular, we will focus on policies that are linear in the state features

$$u = \pi(s;\theta) = \theta^T \phi(s) \tag{44}$$

where $\phi : S \to \mathbb{R}^k$ is a mapping from states to features and $\theta \in \mathbb{R}^{m \times k}$ is a set of parameters that linearly map these features into controls. As before, we will use $J(\pi_\theta)$ or simply $J(\theta))$ to denote the value of for the policy parametrized by θ.

One of the simplest policy search algorithms is just a gradient descent method that repeatedly updates the parameters according to

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta) \tag{45}$$

where α is a step size and $\nabla_\theta J(\theta)$ is the gradient of the cost function with respect to the policy parameters (known hereafter as the policy gradient). Although computing this gradient term can be quite complicated without a model of the system, in the next section we describe a simple approximation method.

7.1.2 The Signed Derivative Approximation

The signed derivative approximation allows us to quickly approximate the policy gradient of certain systems *without* a model. Although we omit the complete derivation for brevity, the intuition of the approach is as follows. It turns out that the policy gradient, written in an analytic form, depends on the dynamics model of the system only through terms of the form

$$\left(\frac{\partial s_t}{\partial u_{t'}}\right) \tag{46}$$

for $t > t'$. These terms are the *Jacobians* of future states with respect to previous inputs. They provide the critical motivation for the signed derivative

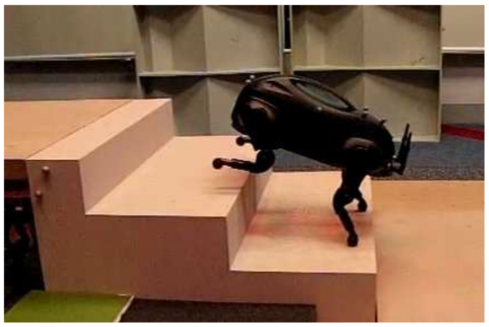


Figure 15: The desired task for the LittleDog: climb over three large steps.

approximation, so it is worth looking at them more closely. These Jacobians are matrices $\left(\frac{\partial s_t}{\partial u_{t'}}\right) \in \mathbb{R}^{n \times m}$ where the i, j element of the $\left(\frac{\partial s_t}{\partial u_{t'}}\right)$ denotes the derivative of the ith element of the state s_t with respect to the jth element of $u_{t'}$, i.e.,

$$\left(\frac{\partial s_t}{\partial u_{t'}}\right)_{ij} \equiv \frac{\partial (s_t)_i}{\partial (u_{t'})_j}. \tag{47}$$

In other words $(\frac{\partial s_t}{\partial u_{t'}})_{ij}$ indicates how the ith element of s_t would change if we made a small adjustment to the j element of the control at a previous time t' (and assuming we are following the policy θ).

In general, the elements of these Jacobians are quite difficult to compute, as they depend on the true dynamics model of the environment and the policy parameters θ. However, the signed derivative approximation is based on the insight that oftentimes it is fairly easy to guess the *signs* of the dominant entries of these matrices: this only requires knowing the general *direction* of how previous control inputs will affect future states. Returning to the front jump maneuver, it may be very difficult to determine the derivative of the future states (for instance the future orientation of the robot) with respect to the back shift amount, but the *direction* of the gradient seems fairly obvious: shifting the robot back further will increase the pitch, as it will tend to back the robot fall back more. In the signed derivative approach to policy gradient, we approximate *all* the Jacobian terms with a single matrix $S \in \mathbb{R}^{n \times m}$, called the signed derivative, where entries in S correspond to the signs of the dominant entries in the Jacobians. The resulting algorithm is shown in Algorithm 1. A complete derivation of the approach, as well as theoretic results on the method, are beyond the scope of this paper, but are available in [19]. We merely note that while the application of the signed derivative method is straightforward for the jumping task, the algorithm itself is quite general and can be applied to many different domains.

Algorithm 1 Policy Gradient w/ Signed Derivative (PGSD)

```
Input:
    $S \in \mathbb{R}^{m \times n}$: signed derivative matrix
    $H \in \mathbb{Z}_+$: horizon
    $C : \mathbb{R}^n \to \mathbb{R}$: cost function
    $\alpha \in \mathbb{R}_+$: learning rate
    $\phi : \mathbb{R}^n \to \mathbb{R}^k$: feature vector function
    $\theta_0 \in \mathbb{R}^{k \times m}$: initial policy parameters
Repeat:
    1. Execute policy for $H$ steps to obtain $u_0, s_1, \ldots, u_{H-1}, u_H$.
    2. Compute approximate gradients w.r.t. controls:
           $\widetilde{\nabla}_{u_t} J(s_0, \Theta) \leftarrow \sum_{t'=t+1}^{H} S^T \nabla_{s_{t'}} C(s_{t'})$
    3. Update parameters:
           $\theta \leftarrow \theta - \frac{\alpha}{H} \sum_{t=0}^{H-1} \phi(s_t, t)(\widetilde{\nabla}_{u_t} J(s_0, \Theta))^T$
```

7.2 Application to Learning Jumping Maneuver

Here we present results on applying PGSD to the previously mentioned task of jumping the front legs of the robot to quickly climb up large steps.

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, $C(s) = |\beta - \beta^\star|$, so that $\nabla_s C(s) = 1$ if $\beta > \beta^\star$ and $\nabla_s C(s) = -1$ if $\beta < \beta^\star$ — i.e., the gradient is just the direction in which we should adjust our

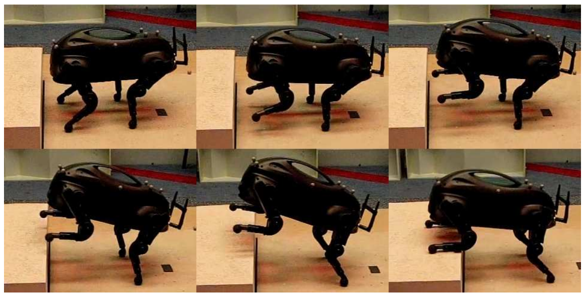


Figure 16: A properly executed jump.

control. When $H = 1$, the PGSD update then takes on the very simple form:

$$\theta \leftarrow \begin{cases} \theta - \alpha\phi(s) & \text{robot didn't clear step} \\ \theta & \text{jump succeeded} \\ \theta + \alpha\phi(s) & \text{robot flipped backwards} \end{cases} \tag{48}$$

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 15. 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/`. Although we do not discuss it further in the paper, we also applied the method to learning a balancing policy for the trot gait, and a video of this result is also available at the above page.

8 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 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 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

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 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.

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 7 shows our performance on the Phase II metrics. For Phase II, our research was focused entirely on static gaits: the static gait described in Section 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 variation in the run times).

Table 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 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

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 8: Speeds and success rates for Phase III system and terrains. Average speed is based upon best two runs. The first 9 terrain represent the "standard" Phase III tests, while the last three represent three additional optional terrains.

the 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`.

9 Conclusions and Future Directions

In this paper 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. Central to our approach was the use of learning algorithms to learn route planning, footstep planning, and dynamic maneuvers. We also highlight the use of rapid recovery and replanning techniques, which allow us to overcome situations where the robot deviates from its desired trajectory.

While we have developed a substantial system for this program, capable of navigating terrains beyond what quadrupeds could handle when this work began, the research has also lead to new topics and directions. Some of these directions, which we feel will further advance the state of the art in quadruped locomotion, are as follows.

Intelligent gait selection, higher level planning. As discussed in Section 3 the gait selection mechanism we use for our complete system is a rather simple set of rules, engineered for the terrains provided as part of the Learning Locomotion project, and thus we feel it is an element that could certainly be

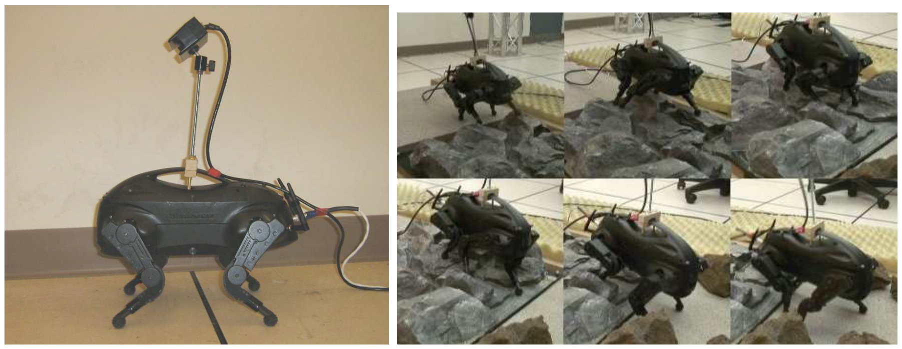


Figure 17: Prototype system with onboard stereo camera, and snapshots of the system crossing one of the more difficult terrains, using only onboard vision.

improved upon, using learning approach to automatically select the best gait given the robot's current configuration. Even more broadly, the highest-level planning throughout our work has been relatively simplistic, as the terrain setup is fairly straightforward: there is a terrain in front of the robot and a goal beyond that; while route planning in this local sense is still crucial, there is little truly high-level logic involved here, dealing with distant goals and multiple choices of terrain to reach them. As legged robots venture out of the lab and into the real world, this is a topic that will certainly arise.

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.

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 here has been conducted in a highly idealized setting, where we have full knowledge 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 actually pursued this topic a fair amount in tandem with the research presented here, and in [18] 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 5) using only onboard vision for both localization and mapping. Figure 17 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.

Fully autonomous, life-long learning. Finally, we note that many of the learning methods we employ in this work require some form of teacher or human supervisor, either an "expert" that can demonstrate good behavior to the robot, or a person with enough knowledge of the system dynamics to design the signed derivative matrices. These approaches are extremely useful given the current state of legged robots, as they can quickly build a system that goes from having little ability at all to one which can perform very robustly. But as the time scale over which we need these robots to function increases, so does the need for system that can learn entirely on their own, without any human intervention. We ultimately envision a setting where algorithms such as the ones we present here are used to initialize robot behavior, and life-long learning approaches operate slowly over time, gradually building a system that performs better and better.

Acknowledgments

This work was supported by the DARPA Learning Locomotion program under contract number FA8650-05-C-7261. J. Zico Kolter was partially supported by a National Science Foundation Graduate Research Fellowship.

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 2004.

[2] John Bares and David Wettergreen. Dante II: Technical description, results and lessons learned. *International Journal of Robotics Research*, 18(7):621–649, July 1999.

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

[4] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[5] Jonas Buchli and Auke Jan Ijspeert. Self-organizing adaptive legged locomotion in a compliant quadruped robot. *Autonomous Robots*, 25:331–345, 2008.

[6] Katie Byl, Alexander Shkolnik, Sam Prentice, Nicholas Roy, and Russ Tedrake. Reliable dynamic motions for a stiff quadruped. In *Proceedings of the 11th International Symposium on Experimental Robotics*, 2008.

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

[8] Joel Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning biped navigation strategies in complex environments. In *Proceedings of the International Conference on Humanoid Robotics*, 2003.

[9] Yasuhiro Fukuoka, Hiroshi Kimura, and Avis H. Cohen. Adaptive dynamic walking of a quadruped robot on irregular terrain based on biological concepts. *The International Journal of Robotics Research*, 22:187–202, 2003.

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

[11] V.S. Gurfinkel, E.V. Gurfinkel, A. Yu Shneider, E. A. Devjanin, A.V. Lensky, and L.G. Shitliman. Walking robot with supervisory control. *Mechanism and Machine Theory*, 16:31–36, 1981.

[12] Bernhard Hengst, Darren Ibbotson, Son Bao Pham, and Claude Sammut. Omnidirectional locomotion for quadruped robots. In *RoboCup 2001: Robot Soccer World Cup V*, pages 368–373, 2002.

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

[14] Gregory S. Hornby, Seichi Takamura, Takashi Yamamoto, and Masahiro Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.

[15] Sangbae Kim, Matthew Spenko, Salomon Trujillo, Barett Heyneman, Daniel Santos, and Mark R. Cutkosky. Smooth vertical surface climbing with directional adhesion. *IEEE Transactions on Robotics*, 24(1):65–74, 2008.

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

[17] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth Conference on Artificial Intelligence*, 2004.

[18] J. Zico Kolter, Youngjun Kim, and Andrew Y. Ng. Stereo vision and terrain modeling for quadruped robots. In *Proceedings of the International Conference on Robotics and Automation*, 2009.

[19] J. Zico Kolter and Andrew Y. Ng. Policy search via the signed derivative. In *Proceedings of Robotics: Science and Systems*, 2009.

[20] J. Zico Kolter, Mike P. Rodgers, and Andrew Y. Ng. A complete control architecture for quadruped locomotion over rough terrain. In *Proceedings of the International Conference on Robotics and Automation (to appear)*, 2008.

[21] Eric Krotkov, Reid Simmons, and William L. Whittaker. *Ambler: Performance of a Six-Legged Planetary Rover*, 35(1):75–81, 1995.

[22] Chun-Shin Lin, Po-Rong Chang, and J.Y.S. Luh. Formulation and optimization of cubic polynomial joint trajectories for industrial robots. *IEEE Transactions on Automatic Control*, 28(12):1066–1074, 1983.

[23] R. B. McGhee. Finite state control of quadruped locomotion. *Simulation*, 5:135–140, 1967.

[24] R. B. McGhee. Some finite state aspects of legged locomotion. *Mathematical Biosciences*, 2:67–84, 1968.

[25] R. B. McGhee and A. A. Frank. On the stability properties of quadruped creeping gaits. *Mathematical Biosciences*, 3:331–351, 1968.

[26] R.B. McGhee. Vehicular legged locomotion. In *Advances in Automation and Robotics*, pages 259–284. 1985.

[27] R. S. Mosher. Test and evaluation of a versatile walking truck. In *Proceedings of off-road mobili6ty research symposium*, pages 359–379, 1968.

[28] Gergeley Neu and Csaba Szepesvári. Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proceedings of Uncertainty in Artificial Intelligence*, 2007.

[29] J. Gordon Nichol, Surya P.N. Singh, Kennetch J. Waldron, Luther R. Palmer III, and David E. Orin. System design of a quadrupedal galloping machine. *International Journal of Robotics Research*, 23(10–11):1013–1027, 2004.

[30] Jin-Hyun Park, Hyun-Sik Kim, and Young-Kiu Choi. 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*, 1997.

[31] Ioannis Poulakakis, James Andrew Smith, and Martin Buehler. 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, 2005.

[32] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter. Bigdog, the rough-terrain quadruped robot. In *Proceedings of the International Federation of Autonomous Control*, 2008.

[33] Marc H. Raibert. *Legged Robots that Balance.* MIT Press, 1986.

[34] Nathan Ratliff, J. Andrew Bagnell, and Martin Zinkevich. Maximum margin planning. In *Proceedings of the International Conference on Machine Learning*, 2006.

[35] U. Saranli, M. Buehler, and D.E. Koditschek. Rhex: A simple and highly mobile hexapod robot. *Int. Journal of Robotics Research*, 20:616–631, 2001.

[36] A. Saunders, D.I. Goldman, R.J. Full, and M. Buehler. The RiSE climbing robot: Body and leg design. *Proc. SPIE Int. Soc. Opt. Eng.*, 6230:623017, 2006.

[37] Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. Learning structured prediction models: A large margin approach. In *Proceedings of the International Conference on Machine Learning*, 2005.

[38] A. Ismael F. Vaz and Edite M.G.P. Fernandes. Tools for robotic trajectory planning using cubic splines and semi-infinite programming. In Alberto Seeger, editor, *Recent Advances in Optimization*, pages 399–413. Springer, 2006.

A Convergence of myopic walking patterns

Here we briefly illustrate that the simple "myopic" gait pattern described in Section 3 converges to symmetric gait regardless of the initial configuration of the feet. While there are slightly more intuitive ways of deriving this same result, a simple method for proving this convergence is via analyzing an affine system that corresponds to the gait.

Recall that the process for generating footsteps is: average the foot locations to find their center, move ahead on the path by d_{body} (in this example, we assume the robot is moving in the x direction), and step to the home location of the moving foot relative to this new point. This can be mathematically specified in the following way. Let $x \in \mathbb{R}^4$ be a vector describing the *local* foot x positions of each of the four legs. We define the matrices

$$\begin{aligned}
\bar{A} &= \begin{bmatrix} 3/4 & -1/4 & -1/4 & -1/4 \\ -1/4 & 3/4 & -1/4 & -1/4 \\ -1/4 & -1/4 & 3/4 & -1/4 \\ -1/4 & -1/4 & -1/4 & 3/4 \end{bmatrix} \\
A_0 &= \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

(and similarly for A_1, A_2, A_3, each being the identity where the ith row has all elements equal to 1/4) so that so that $\bar{A}x$ corresponds to centering x (i.e., subtracting the mean from all its components), and A_i corresponds to setting foot i to be the average of the four feet. We also define vectors

$$b_0 = \begin{bmatrix} d_{body} + x_h \\ 0 \\ 0 \\ 0 \end{bmatrix}, \; b_1 = \begin{bmatrix} 0 \\ d_{body} + x_h \\ 0 \\ 0 \end{bmatrix}, \; b_2 = \begin{bmatrix} 0 \\ 0 \\ d_{body} - x_h \\ 0 \end{bmatrix}, \; b_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ d_{body} - x_h \end{bmatrix} \tag{49}$$

Then

$$\bar{A}(A_0\bar{A}x + b_0) \tag{50}$$

corresponds to taking a single footstep with the front-left leg according the myopic strategy: averaging the feet to find the center, leaving all feet but the front-left fixed, while letting that foot be equal to the average of all the feet plus d_{body} plus x_h, then again averaging the feet. Thus,

$$\bar{A}(A_0\bar{A}(A_2\bar{A}(A_1\bar{A}(A_3\bar{A}x + b_3) + b_1) + b_2) + b_0) \tag{51}$$

corresponds to taking four footsteps from an initial configuration x in the standard walking gait order: back-right, front-right, back-left, front-left. To find the steady-state pattern of the gait, we want to find a some $x^\star$ that is a fixed point of this operation, i.e. an $x^\star$ that satisfies

$$\bar{A}(A_0\bar{A}(A_2\bar{A}(A_1\bar{A}(A_3\bar{A}x^\star + b_3) + b_1) + b_2) + b_0) = x^\star \tag{52}$$

which is given by

$$x^\star = (I - A^\star)^{-1}b^\star \tag{53}$$

where we define

$$A^\star \equiv \bar{A}A_0\bar{A}A_2\bar{A}A_1\bar{A}A_3\bar{A} \tag{54}$$

and

$$b^\star \equiv \bar{A}(A_0\bar{A}(A_2\bar{A}(A_1\bar{A}b_3 + b_1) + b_2) + b_0). \tag{55}$$

While these are somewhat cumbersome quantities, we can easily find using symbolic algebra software that

$$x^\star = \begin{bmatrix} (3/5)d_{body} + x_h \\ (-1/5)d_{body} + x_h \\ (1/5)d_{body} - x_h \\ (-3/5)d_{body} - x_h \end{bmatrix} \tag{56}$$

which is a symmetric walking gait pattern. Since $A^\star$ is a stable matrix (with maximum singular value $\sigma_1(A) \approx 0.2265$), the myopic gait procedure will converge to with a fast exponential rate of convergence to $x^\star$ regardless of the initial foot positions.

The same procedure can show convergence for the trot gait pattern, similarly defining

$$\begin{aligned} A_0 &= \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix} \\ A_1 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

and

$$b_0 = \begin{bmatrix} d_{body} + x_h \\ 0 \\ 0 \\ d_{body} - x_h \end{bmatrix}, \; b_1 = \begin{bmatrix} 0 \\ d_{body} + x_h \\ d_{body} - x_h \\ 0 \end{bmatrix}. \tag{57}$$

A complete progression through the gait is now even simpler,

$$\bar{A}(A_1\bar{A}(A_0\bar{A}x + b_0) + b_1) \tag{58}$$

and performing using the same logic as above we now find that

$$x^\star = \begin{bmatrix} (-1/3)d_{body} + x_h \\ (1/3)d_{body} + x_h \\ (1/3)d_{body} - x_h \\ (-1/3)d_{body} - x_h \end{bmatrix}. \tag{59}$$

and $\sigma_1(A^\star) = 0.25$. Thus, in this case the myopic procedure also converges to a symmetric trot pattern, regardless of the initial foot configuration.