Learning Contact-Aware Robot Controllers from Mixed Integer Optimization

by

Robin L. H. Deits

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Februrary 2019

Signature of Author .....
Department of Electrical Engineering and Computer Science
December 7, 2018

Certified by .....
Russ Tedrake
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Theses

Learning Contact-Aware Robot Controllers from Mixed Integer Optimization

by

Robin L. H. Deits

Submitted to the Department of Electrical Engineering and Computer Science
on December 7, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The problem of handling contact is central to the task of controlling a walking robot. Robots can only move through the world by exerting forces on their environment, and choosing where, when, and how to touch the world is the fundamental challenge of locomotion. Because the space of possible contacts is a high-dimensional mix of discrete and continuous decisions, it has historically been difficult or impossible to make complex contact decisions online at control rates. This work first presents an approach to contact planning which is able to make some guarantees of global optimality through mixed-integer programming. That method is applied successfully to a humanoid robot in laboratory conditions, but proves difficult to rely on when the robot is experiences unmodeled disturbances. To overcome those limitations, this thesis also introduces LVIS (Learning from Value Interval Sampling) a new approach to the control of walking robots which allows complex contact decisions to be made online using a cost function trained from offline trajectory optimizations. The LVIS algorithm is demonstrated on a simple cart-pole system with walls as well as a simplified bipedal robot model, and its success at allowing both models to use contact decisions to recover from external disturbances is demonstrated in simulation.

Thesis Supervisor: Russ Tedrake
Title: Professor of Electrical Engineering and Computer Science

Contents

Acknowledgments

To the members of the Robot Locomotion Group: Thank you for providing the most exciting, intelligent, supportive, and rewarding environment in the world and for being the best people to learn from. I've wanted to work on walking robots at MIT ever since I was a kind playing with Legos, and you all made it better than I could have ever imagined.

To the Fanny and John Hertz Foundation: Thank you for your support, for connecting me with some of the smartest and most interesting people in the world, and for always reminding me to strive to achieve something even greater.

To my parents: Thank you for everything, for just the right balance of encouragement and freedom, for encouraging me every step of the way, for showing me how proud you were of what I'd done and making me feel like I could still do even more. I couldn't have done it without you.

To Michele: Thank you for being my biggest fan and my best friend, for appreciating (or at least tolerating) my scientific papers and dumb jokes alike, and for making me feel special and loved every single day <3

Preface

Several of the chapters in this thesis are adapted from previously published works. Chapter 2 is adapted from *Footstep Planning on Uneven Terrain with Mixed-Integer Convex Optimization* [1], Chapter 3 is adapted from *Efficient Mixed-Integer Planning for UAVs in Cluttered Environments* [2], and Chapter 5 is adapted from work currently under review, available in pre-print form as *LVIS: Learning from Value Function Intervals for Contact-Aware Robot Controllers* [3].

Chapter 1

Introduction

Walking robots have a unique promise for service to the world, allowing access to environments unsuitable for wheels, treads, or propellers. Bipedal robots, in particular, offer the ability to fit naturally into environments designed for humans, alongside them in a home or in place of them in a dangerous situation. Current walking robots, however, tend to simultaneously be over-cautious and, paradoxically, unreliable (at least as far as staying upright is concerned). One need only look as far as the finals of the DARPA Robotics Challenge [4] to see bipedal robots laboriously and carefully moving through the world, with a constant danger of falling. The key to bringing walking robots out of the lab will be giving them the ability to move safely and quickly through the world, using the environment to help maintain balance rather than trying to avoid all contact with the world.

Why, then, has this proven to be so difficult? Legged robots have a unique constraint which is not shared with flying robots, fixed-base robots like industrial arms, or wheeled robots: the only way for a legged robot to control its momentum is to make and break contact with the world. This restriction is fundamental: a robot which is not in contact with anything is, by definition, in free-fall and has no control over its momentum. In a walking robot, this contact most often occurs between the robot's feet and the floor, but more complex environments might require the robot to use its hands or body to make contact with walls, railings, ladders, chairs, vehicles, or other robots. Control of contact constitutes

the central challenge for the control of legged robots.

The most mature tools for the control of robots come from the study the control of linear systems. For a system with perfectly linear dynamics and no hard constraints on its state or actions, we can produce a stabilizing linear controller, optimal with respect to a chosen cost function, using the Linear Quadratic Regulator (LQR) technique [5]. Even for a robot with somewhat nonlinear dynamics, we can often make a reasonable linear approximation and apply LQR with some success. But the addition of contact between the robot and the world complicates the situation in a number of ways. First, the presence or absence of contact between some part of the robot and its environment can profoundly affect the robot's dynamics. The difference between just reaching a foothold and just missing one can be catastrophic, and that difference is not well expressed by a single linear approximation of the robot's dynamics as a function of its state. As a result, techniques like LQR tend to perform poorly when the robot is rapidly changing contact modes. Second, even if the presence of a contact is known and fixed, there are significant limitations in the way the robot can move to exploit that contact to control its momentum: Physics requires that the robot's body not penetrate whatever rigid object it might be touching and that the force between the robot and the world be consistent with friction. A robot's flat foot cannot pull on the flat ground, nor can it push on the ground from a meter away, nor can it apply a force parallel to the ground without some corresponding force downward. These physical constraints require a controller that can reason about constraints online and take only whatever action is optimal and consistent with physics.

Control of a legged robot can be thought of as an optimization problem: at each time step, the robot must choose an action which is optimal with respect to some cost, subject to constraints imposed by the robot's current task and by physics. For unconstrained linear systems, LQR provides an optimal solution without any need to solve the optimization online, but for more complex systems such complete solutions are generally not available. Instead, we typically represent the control problem not just *conceptually* as an optimization, but *literally* in the software as well; our control code consists of writing down the optimization

problem at each time step, solving it, and then applying the resulting solution to the robot. This approach is powerful: our cost function and constraints can be as complex as we want and can encode any imaginable combination of penalties and restrictions. But it is also limiting, as more complex optimization problems take longer to solve, which limits the rate at which we can control the robot. A controller which uses an intricate and detailed set of costs and constraints is of no use for balance if it can only produce a new command for the robot every few seconds, minutes, or hours.

The aim of this thesis is to explore ways to make the difficult task of controlling a legged tractable and efficient. In Part I, I will describe a decomposition of the problem into one of finding optimal contact locations and then applying existing smooth control techniques given those contacts. I will introduce a method of planning foot contacts on complicated terrain, and a segmentation algorithm that is useful in pre-processing the environment. I will discuss how this allows a robot to plan a sequence of footholds which are likely to be feasible, and I will show how that footstep planning was sufficient to allow a humanoid robot to walk, unsupervised, across an unmapped row of cinder blocks. As an aside, I will describe how the footstep planner led to a new technique for planning the flights of UAVs through cluttered environments using a similar optimization approach. Finally, despite the optimality promises of the footstep planner, I will discuss the ways in which it failed when the robot was asked to traverse more complex rough terrain, due to a lack of reasoning about the dynamics of the robot and the complexities of its kinematics.

In Part II, I will return to the original problem with a new question: What would it take to truly make contact decisions online instead of trying to stick to some pre-generated plan? I will write out the general control optimization that we might want to solve, then a linearized version we can actually solve to global optimality (but very slowly). I will introduce the notion of learning the solution from offline examples and discuss how even sampling from the optimal policy offline is difficult. Instead, I will introduce the idea of collecting value function intervals in order to train an approximation of the cost to go, and I will demonstrate that approach on the simplified humanoid model. Finally, I will discuss the challenges that

remain in scaling the approach up to the full humanoid and in applying this new technique on a real robot.

Part I

Making Contact Decisions Ahead-of-Time

Control of a system in contact with the world involves reasoning about a set of discrete choices (what objects to touch and with what parts of the robot to touch them) and a set of continuous choices (what precise efforts to apply at each joint and what contact forces those efforts will produce). Reasoning about all of those decisions simultaneously has historically resulted in optimization problems which are too difficult to solve at an acceptable control rate. It is tempting, then, to divide and conquer: first choose the contacts that the robot will make with the world, then choose the precise actions and contact forces subject to those contacts.

For a bipedal walking robot, contact primarily occurs between the robot's feet and the ground, so we can further reduce the problem of choosing contacts to one of simply choosing a set of footstep positions. Chapter 2 introduces a method of planning footsteps on rough terrain with some guarantees of global optimality. Given those footsteps, we can create a controller capable of following that footstep plan and choosing the robot's joint efforts online. The online control component is discussed in [6].

Chapter 3 takes a detour from the focus on contact planning for humanoids to discuss how the same tools from footstep planning can be applied to the motion planning of a flying robot. The optimization framework based on convex segmentation and mixed-integer optimization from Chapter 2 proved to be a useful approach to general obstacle-avoidance problems. By treating the problem of an aerial vehicle moving through space analogously to a walking robot navigating through terrain, we can construct a similar optimization problem with similar guarantees of global optimality.

Chapter 4 returns to the topic of walking robots to discuss the success and failures of the footstep planning approach in the lab and in field tests. It explores the way in which the footstep planning approach was used to successfully navigate a robot across unmapped rough terrain in a lab setting. It also discusses why it was difficult to successfully apply the footstep planner outside of the lab and why the footstep planner was not successfully used for the rough terrain traversal at the DARPA Robotics Challenge finals.

Chapter 2

Planning Footsteps with Mixed-Integer Convex Optimization

This chapter is adapted from work previously published in [1].

2.1 Introduction

The purpose of a footstep planner is to find a list of footstep locations that a walking robot can follow safely to reach some goal. Footstep planning is a significant simplification of motion planning through contact, one in which the whole-body kinematics and dynamics are typically coarsely approximated or ignored in order to produce a tractable problem. The challenge of footstep planning thus consists of finding a path through a constrained environment to a goal while respecting constraints on the locations of and displacements between footsteps. An example of such a plan is shown in Figure 2-1.

Broadly speaking, there exist two families of approaches to footstep planning: discrete searches and continuous optimizations. The discrete search approaches have typically involved a precomputed action set: either represented as a set of possible displacements from one footstep to the next or a set of possible footholds in the environment. Chaining actions together forms a tree of possible footstep plans, which can be explored using existing

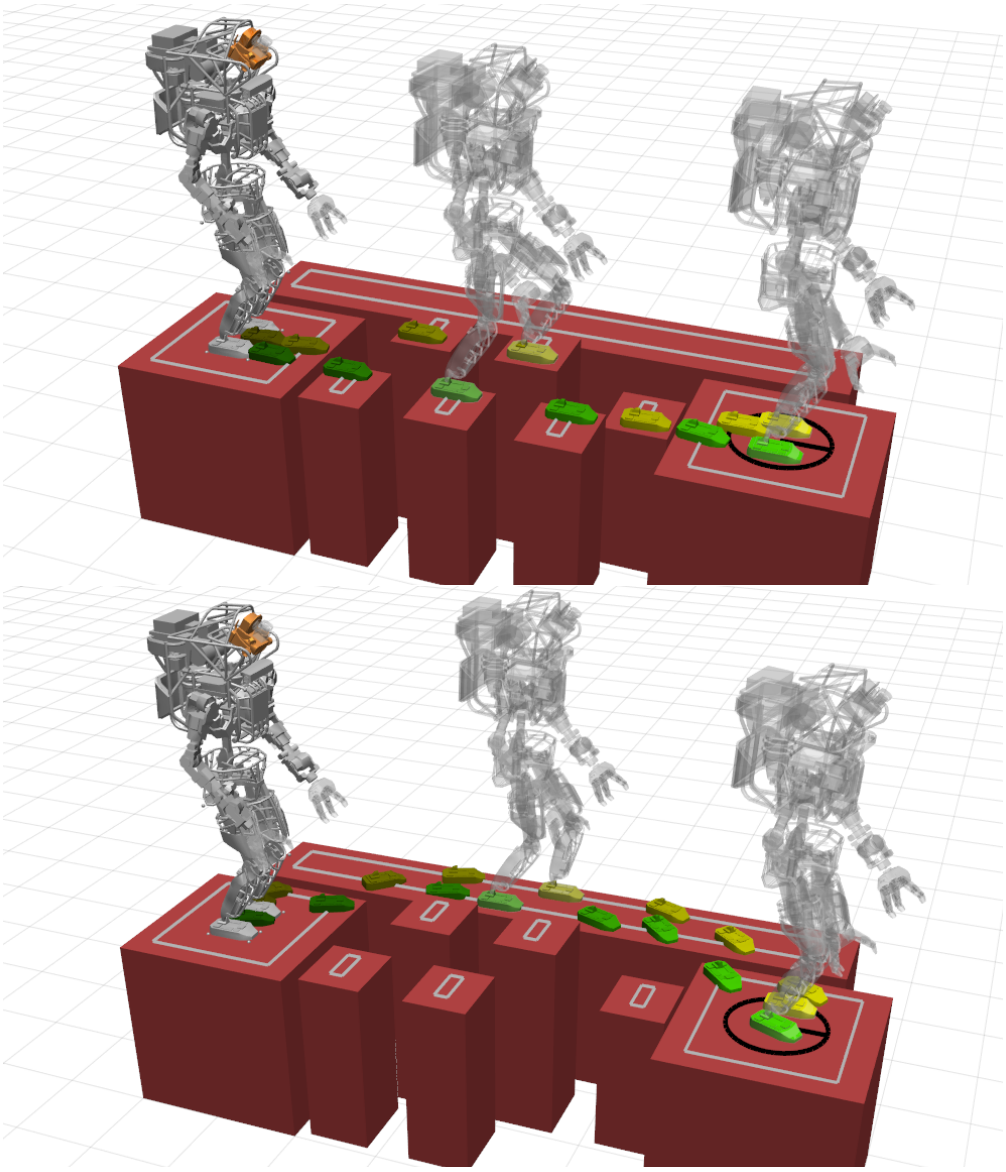


Figure 2-1: Two examples of the output of our MIQCQP footstep planner. Above: An Atlas biped planning footsteps across a set of stepping stones. Below: With one stepping stone removed, Atlas must take the longer detour around the stepping stones. The gray rectangles are the boundaries of convex regions of obstacle-free configuration space generated by IRIS [7] projected into the xy plane.

discrete search methods like A^* and RRT. Action set approaches using pre-computed step displacements have been used by Hornung [8], Michel [9], Baudouin [10], Chestnutt [11, 12], and Kuffner [13, 14]. Similarly, Shkolnik et al. used a precomputed set of dynamic motions, rather than foot displacements, and an RRT search to find motions for a quadruped [15]. The fixed foothold sets have been used by Bretl for climbing robots [16] and by Neuhaus for the LittleDog quadruped [17]. These approaches can easily handle obstacle avoidance by pruning the tree of actions when a particular action would put a foot in collision with an obstacle [13, 14, 10], including obstacle avoidance in the cost function evaluated at each leaf of the tree [11, 9], or adapting the set of actions when a collision is detected [12]. However, they have also tended to suffer from the tradeoff between a small action set, which reduces the branching factor of the search tree, and a large action set, which covers a larger set of the true space of foot displacements but is much harder to search [10]. In addition, applying A^* or other informed search methods to our problem is complicated by the difficulty in defining a good heuristic for partial footstep plans: we cannot generally know how many additional footsteps a partial plan will need in order to reach the goal without actually searching for those steps [11].

The continuous optimization approaches, on the other hand, operate directly on the poses of the footsteps as continuous decision variables. This avoids the restriction to a small set of fixed actions and thus allows more possible footstep plans to be explored, but correctly handling rotation and obstacle avoidance turns out to be difficult in a continuous optimization. Both footstep rotation and obstacle avoidance generally require non-convex constraints to enforce them, since the set of rotation matrices and the set of points outside a closed obstacle are non-convex. We typically cannot find guarantees of completeness or global optimality for such non-convex problems [18]. We have presented a non-convex continuous optimization for footstep planning, used by Team MIT during the DARPA Robotics Challenge 2013 Trials [19], but this optimization could not guarantee optimality of its solutions or find paths around obstacles. Alternatively, footstep rotation and obstacle avoidance can simply be ignored: Herdt et al. fix the footstep orientations and do not consider obstacle avoidance.

This allows them to form a single quadratic program (QP) which can choose optimal footstep placements and control actions for a walking robot model [20].

We choose to use a mixed-integer convex program (specifically, a mixed-integer quadratically constrained quadratic program) to provide a more capable continuous footstep planner. Such a program allows us to perform a continuous optimization of the footstep placements, while using integer variables to absorb any non-convex constraints. We handle orientation of the footstep placements by approximating the trigonometric sin and cos functions with piecewise linear functions, using a set of integer variables to choose the appropriate approximation. We also avoid the non-convex constraints inherent in avoiding obstacles by instead enumerating a set of convex obstacle-free configuration space regions and using additional integer variables to assign footsteps to those regions. The presence of integer constraints significantly complicates our formulation, but a wide variety of commercial and free tools for mixed-integer convex programming exist, all of which can provide globally optimal solutions or proofs of infeasibility, as appropriate [21, 22, 23]. Thus, we can solve an entire footstep planning problem to optimality while ensuring obstacle avoidance, a task that, to our knowledge, has not been accomplished before. Footstep plans produced by our algorithm are shown in Figures 2-1 and 2-2

This is not unlike the work of Richards et al., who constructed a mixed-integer linear program to plan UAV trajectories while avoiding obstacles [24]. They represented each (convex, 2D) obstacle as a set of linear constraints, each of which generates a pair of half-spaces, one containing the obstacle and one not. They assigned a binary integer variable to every pair of half-spaces and required that, for every obstacle, the vehicle's location must be in at least one of the non-obstacle half-spaces. Instead of adding binary variables for every single face of every obstacle, we precompute several convex obstacle-free regions, then assign a single binary variable to each region and require that every footstep is assigned to one such region, which dramatically reduces the number of binary variables required. The precomputed convex obstacle-free regions are produced by the IRIS algorithm, presented in [7].

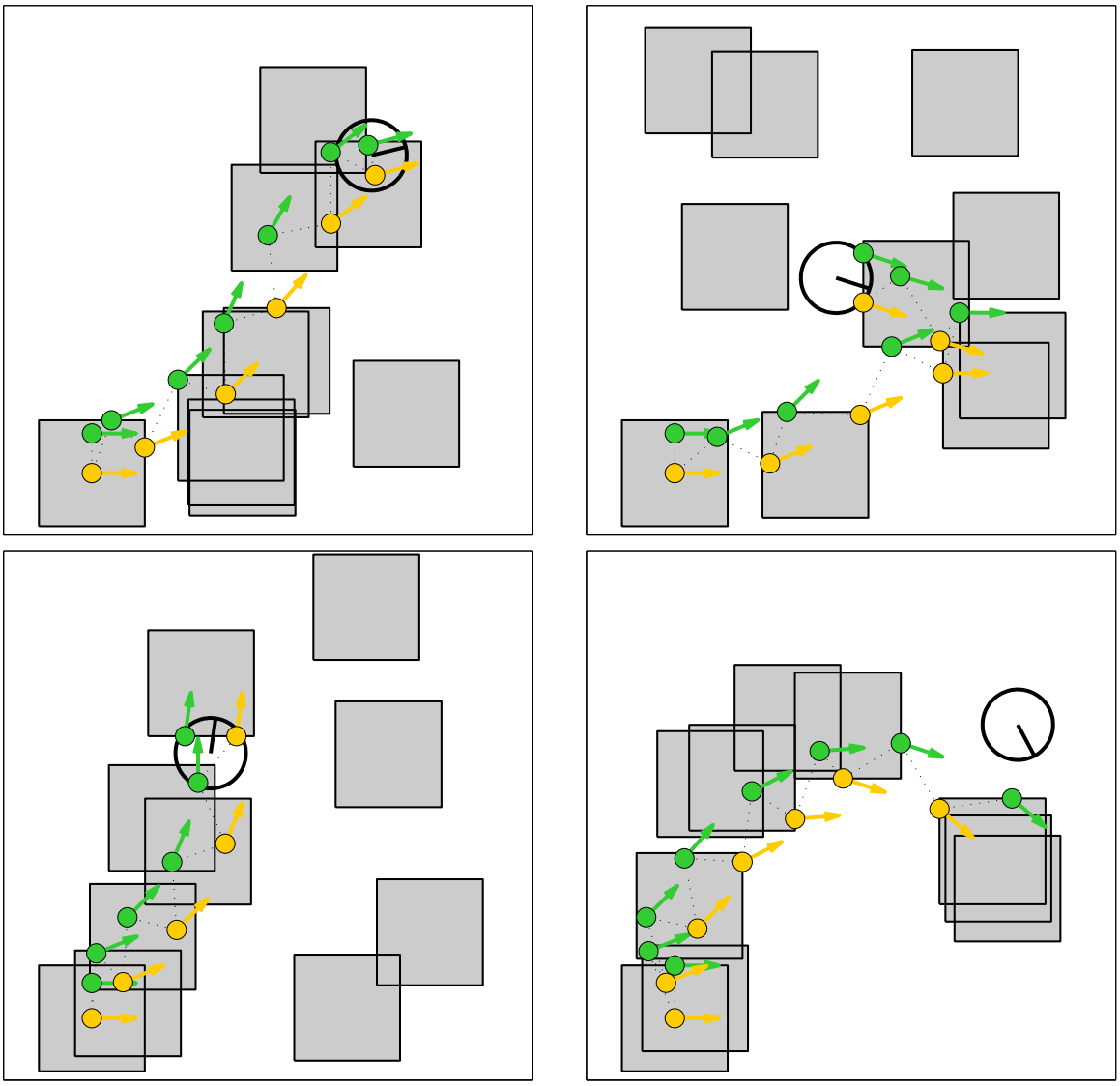


Figure 2-2: Four randomly generated 2D environments demonstrating the MIQCQP footstep planner. The gray squares are the randomly placed regions of safe terrain; the black circle is the goal location, with a line indicating the desired orientation of the robot; and the green and yellow markers are the locations of the right and left footsteps, respectively, with arrows showing the orientation of each step. The foot is assumed to be a point for these examples.

2.2 Technical Approach

Our task is to determine the precise x, y, z and θ (yaw) positions of N footsteps, subject to the constraints that

1. Each step does not intersect any obstacle
2. Each step is within some convex reachable region relative to the position of the prior step

To accomplish this, we invert the non-convex problem of *avoiding* every obstacle and instead reformulate the problem as one of assigning each footstep to some pre-computed convex region of obstacle-free terrain. We then add quadratic constraints to ensure that each footstep is reachable from the prior step, using a piecewise linear approximation of sin and cos to handle step rotation.

2.2.1 Assigning Steps to Obstacle-Free Regions

Our first task is to decompose the 3D environment into a set of convex regions in which the foot can safely be placed. The IRIS algorithm, first presented in [7] and designed for this particular task, quickly generates obstacle-free convex regions in the x, y, and θ (yaw) configuration space of the robot's feet. To do so, IRIS begins with a single obstacle-free seed point and constructs a small ellipsoidal ball around that point. The main IRIS algorithm is an alternation between two convex optimizations: First, a set of hyperplanes is generated to separate the ellipsoid from the set of configuration-space obstacles. The intersection of the obstacle-free half-spaces defined by those hyperplanes is a polytope. Second, a semidefinite program is solved to find the maximum volume ellipsoid inside that polytope. These two steps can be repeated until the ellipsoid ceases to grow, and the result is the final obstacle-free polytope [7].

For each obstacle-free terrain region, we fit a plane in x, y, and z to the local terrain and add additional linear constraints to force footsteps in that region to lie on the plane. We label the total number of convex regions as R and for each region r we identify the associated

polytope with the matrix A_r and vector b_r such that if footstep f_j is assigned to region r, then

$$A_r f_j \leq b_r$$

where

$$f_j \equiv \begin{bmatrix} x_j \\ y_j \\ z_j \\ \theta_j \end{bmatrix} \text{ and } r \in \{1, \ldots, R\}$$

To describe the assignment of footsteps to safe regions, we construct a matrix of binary variables $H \in \{0,1\}^{R \times N}$, such that if $H_{r,j} = 1$ then footstep j is assigned to region r:

$$H_{r,j} \implies A_r f_j \leq b_r \tag{2.1}$$

$$\sum_{r=1}^{R} H_{r,j} = 1 \quad \forall j = 1, \ldots, N \tag{2.2}$$

The *implies* operator in (2.1) can be converted to a linear constraint using a standard big-M formulation [25] or handled directly by mixed-integer programming solvers such as IBM ILOG CPLEX [23]. The constraint (2.2) requires that every footstep be assigned to exactly one safe terrain region.

2.2.2 Ensuring Reachability

We choose to approximate the reachable set of footstep positions as an intersection of circular regions in the xy plane, with additional linear constraints on footstep displacements in yaw and z. The reachable set defined by the intersection of two such circular regions is shown in

Figure 2-4. For each footstep j, we require that

$$\left\| \begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \left(\begin{bmatrix} x_j \\ y_j \end{bmatrix} + \begin{bmatrix} \cos\theta_j & -\sin\theta_j \\ \sin\theta_j & \cos\theta_j \end{bmatrix} p_1 \right) \right\| \leq d_1 \tag{2.3}$$

$$\left\| \begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \left(\begin{bmatrix} x_j \\ y_j \end{bmatrix} + \begin{bmatrix} \cos\theta_j & -\sin\theta_j \\ \sin\theta_j & \cos\theta_j \end{bmatrix} p_2 \right) \right\| \leq d_2 \tag{2.4}$$

where p_1, p_2 are the centers of the circles, expressed in the frame of footstep j and d_1, d_2 are their radii. When θ_j is fixed, constraints (2.3) and (2.4) are convex quadratic constraints, but including θ as a decision variable makes the constraint non-convex by introducing the trigonometric functions of θ. We will handle this problem by introducing two new variables for every footstep: s_j and c_j, which will approximate $\sin\theta_j$ and $\cos\theta_j$, respectively. Constraints (2.3,2.4) thus become:

$$\left\| \begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \left(\begin{bmatrix} x_j \\ y_j \end{bmatrix} + \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix} p_1 \right) \right\| \leq d_1 \tag{2.5}$$

$$\left\| \begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \left(\begin{bmatrix} x_j \\ y_j \end{bmatrix} + \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix} p_2 \right) \right\| \leq d_2. \tag{2.6}$$

Since p_1, p_2, d_1, d_2 are constants, this is a convex quadratic constraint.

Our work is not yet complete, however, since we now must enforce that s_j and c_j approximate sin and cos without introducing non-convex trigonometric constraints. We choose instead to create a simple piecewise linear approximation of sin and cos and a set of binary variables to determine which piece of the approximation to use. We construct a binary matrix $S \in \{0,1\}^{L \times N}$, where L is the number of piecewise linear segments and add constraints

of the form

$$S_{\ell,j} \implies \begin{cases} \phi_\ell \leq \theta_j \leq \phi_{\ell+1} \\ s_j = g_\ell \theta_j + h_\ell \end{cases} \tag{2.7}$$

$$\sum_{\ell=1}^{L} S_{\ell,j} = 1 \quad \forall j = 1, \ldots, N \tag{2.8}$$

where g_ℓ and h_ℓ are the slope and intercept of the linear approximation of $\sin\theta$ between ϕ_ℓ and $\phi_{\ell+1}$. We likewise add piecewise linear constraints of the same form for c_j.

The particular choice of g_ℓ and h_ℓ turns out to be quite important: we must ensure that our approximation never overestimates the reachable space of foot placements. We can verify this empirically for the approximation shown in Figure 2-3 by checking that the intersection of constraints (2.5,2.6) is contained within the intersection of constraints (2.3,2.4) for all values of θ. The reachable sets for footsteps at a variety of orientations are shown in Figure 2-5.

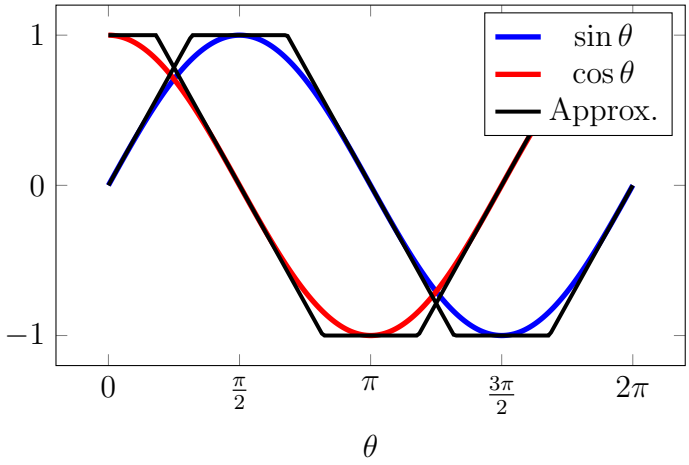


Figure 2-3: Piecewise linear approximation of sine and cosine

A more typical approach would be to define a convex polytope with m faces, fixed in the frame of one foot, into which the next foot must be placed. This approach has been successfully in convex optimization without rotations [20], but it produces non-convex constraints

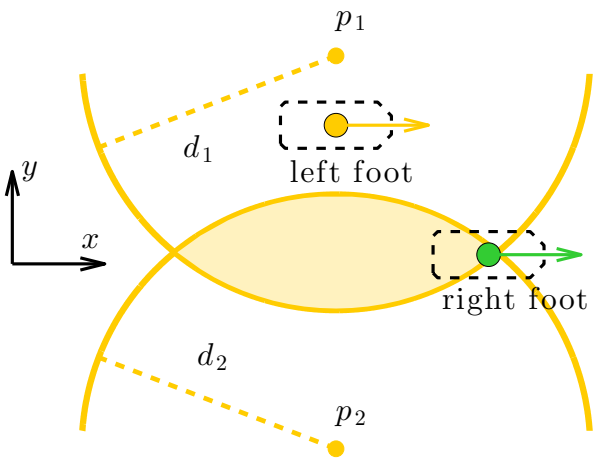


Figure 2-4: Approximation of the reachable set of locations for the right foot, given the position of the left foot. The gold arrow shows the position and orientation of the left foot, viewed from above. The shaded region shows the set of reachable poses for the right foot in the xy plane, defined as the intersection of constraints (2.5,2.6) at orientation of $\theta_j = 0$, for which our approximation of sin and cos is exact. One possible future pose of the right foot is shown for reference.

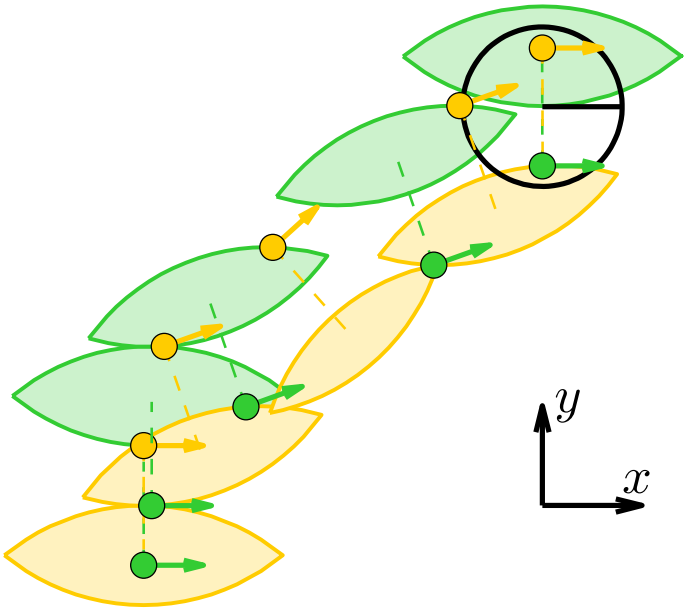


Figure 2-5: A simple footstep plan with 2D reachable regions shown. The goal is the black circle in the top right, and each arrow shows the position and orientation of one footstep. For each step, we draw the shaded region defined by constraints (2.5,2.6) into which the next step must be placed. Note that the feasible region shrinks when the step orientation is not a multiple of $\pi/2$, as our approximation of sin and cos becomes inexact.

when rotations are introduced. Such a polytope would involve constraints of the form

$$\Omega \begin{bmatrix} c_j & s_j \\ -s_j & c_j \end{bmatrix} \left(\begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \begin{bmatrix} x_j \\ y_j \end{bmatrix} \right) \leq \omega \tag{2.9}$$

for some constant matrix $\Omega \in \mathbb{R}^{m \times 2}$ and vector $\omega \in \mathbb{R}^m$. Even under our piecewise linear approximation of sin and cos, (2.9) is a non-convex quadratic constraint and thus not compatible with our mixed-integer convex formulation.

2.2.3 Determining the Total Number of Footsteps

In general we cannot expect to know *a priori* the total number of footsteps which must be taken to bring the robot to a goal pose, so we need some method for determining N efficiently. We could certainly just repeat the optimization for different values of N, performing a binary

search to find the minimum acceptable number of steps, but for efficiency's sake we would prefer to avoid the many runs of the optimizer that this would require.

We might attempt to determine the number of required steps by setting N sufficiently large and simply adding a cost on the squared distance from each footstep to the goal pose, which will stretch our footstep plan towards the goal. If the footsteps reach the goal before N steps have been taken, then we can trim off any additional steps at the end of the plan. However, this approach allows the footstep planner to produce strides of the maximum allowable length for every footstep, even on obstacle-free flat terrain. During experiments leading up to the DARPA Robotics Challenge trials, we determined that a forward stride of 40 cm was achievable on the Atlas biped, but that a nominal stride of approximately 20 cm was safer and more stable. We would thus like to express in our optimization a preference for a particular nominal stride length while still allowing occasional longer strides needed to cross gaps or clear obstacles. We can try to create this result by adding additional quadratic costs on the relative displacement between footsteps, but this requires very careful tuning of the weights of the distance-to-goal cost and the relative step cost for each individual step in order to ensure that the costs balance precisely at the nominal step length for each step.

Instead, we choose a much simpler cost function, with a quadratic cost on the distance from the last footstep to the goal and identical cost weights on the displacement from each footstep to the next. To control the number of footsteps used in the plan, and thus the length of each stride, we add a single binary variable to each footstep, which we will label as t_j (for 'trim'). If t_j is true, then we require that step j be fixed to the initial position of that same foot:

$$t_j \implies f_j = \begin{cases} f_1 & \text{if } j \text{ is odd} \\ f_2 & \text{if } j \text{ is even.} \end{cases} \tag{2.10}$$

Note that f_1 and f_2 are the *fixed* current positions of the robot's two feet.

Since each footstep for which $t_j = 1$ is fixed to the current position of the robot's feet, the number of footsteps which are actually used to move the robot to the goal is $N - \sum_{j=1}^{N} t_j$.

By assigning a negative cost value to each binary variable t_j, we can create an incentive to reduce the number of footsteps used in the plan. To tune the nominal stride length, we can simply adjust the cost assigned to the t_j. Increasing the magnitude of this cost will lengthen the nominal stride uniformly, and decreasing the magnitude will shorten the stride. Thus, we have a single value to tune in order to set the desired stride length, while still allowing strides which exceed this length. After the optimization is complete, we can remove any footsteps at the beginning of the plan for which t_j is true.

2.2.4 Complete Formulation

Putting all of the pieces together gives us the entire footstep planning problem:

$$\underset{f_1,\ldots,f_j,S,C,H,t_1,\ldots,t_j}{\text{minimize}} (f_N - g)^\top Q_g (f_N - g) + \sum_{j=1}^{N} q_t t_j + \sum_{j=1}^{N-1} (f_{j+1} - f_j)^\top Q_r (f_{j+1} - f_j))$$

subject to, for $j = 1, \ldots, N$

safe terrain regions:

$$H_{r,j} \implies A_r f_j \leq b_r \qquad r = 1, \ldots, R$$

piecewise linear $\sin\theta$:

$$S_{\ell,j} \implies \begin{cases} \phi_\ell \leq \theta_j \leq \phi_{\ell+1} \\ s_j = g_\ell \theta_j + h_\ell \end{cases} \qquad \ell = 1, \ldots, L$$

piecewise linear $\cos\theta$:

$$C_{\ell,j} \implies \begin{cases} \phi_\ell \leq \theta_j \leq \phi_{\ell+1} \\ c_j = g_\ell \theta_j + h_\ell \end{cases} \qquad \ell = 1, \ldots, L$$

approximate reachability:

$$\left\| \begin{bmatrix} x_{j+1} \\ y_{j+1} \end{bmatrix} - \left(\begin{bmatrix} x_j \\ y_j \end{bmatrix} + \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix} p_i \right) \right\| \leq d_i \quad i = 1, 2$$

fix extra steps to initial pose:

$$t_j \implies f_j = \begin{cases} f_1 & \text{if } j \text{ is odd} \\ f_2 & \text{if } j \text{ is even.} \end{cases}$$

$$\sum_{r=1}^{R} H_{r,j} = \sum_{\ell=1}^{L} S_{\ell,j} = \sum_{\ell=1}^{L} C_{\ell,j} = 1$$

$$H_{r,j}, S_{\ell,j}, C_{\ell,j}, t_j \in \{0, 1\}$$

bounds on step positions and differences:

$$f_{\min} \leq f_j \leq f_{\max}$$

$$\Delta f_{\min} \leq (f_j - f_{j-1}) \leq \Delta f_{\max}$$

where $g \in \mathbb{R}^4$ is the x, y, z, θ goal pose, $Q_g \in \mathbb{S}^4_+$ and $Q_r \in \mathbb{S}^4_+$ are objective weights on the distance to the goal and between steps, $q_t \in \mathbb{R}$ is an objective weight on trimming unused steps, and $f_{\min}, f_{\max}, \Delta f_{\min}, \Delta f_{\max} \in \mathbb{R}^4$ are bounds on the absolute footstep positions and their differences, respectively. We also fix f_1 and f_2 to the initial poses of the robot's feet.

2.2.5 Solving the Problem

We have implemented this approach in MATLAB [26], using the commercial solver Gurobi [21] to solve the MIQCQP itself. Typical problems involving 10 to 20 footsteps and 10 convex safe terrain regions can be solved in a few seconds to one minute on a Lenovo laptop with an Intel i7 clocked at 2.9 GHz. Smaller footstep plans involving just a few steps can be solved in well under one second on the same hardware, so this method is capable of providing short-horizon footstep plans at realtime rates while walking or longer footstep plans involving complex path planning while stationary. The Mosek optimizer [22] is also capable of solving the problem our formulation to optimality with comparable computational speed.

2.3 Results in Simulation

To demonstrate the MIQCQP footstep planning algorithm, we first generated a collection of random 2D environments. Each environment consisted of 10 square regions of safe terrain, 9 of which were uniformly randomly placed within the bounds of the environment, and 1 of which was placed directly under the starting location of the robot to represent the robot's currently occupied terrain. A goal pose was uniformly randomly placed within the xy bounds of the environment with a desired orientation between $\pm\frac{\pi}{2}$ relative to the robot's starting orientation. Several such environments and the resulting footstep plans are shown in Figure 2-2. Computation times for those environments were between 1 and 10 seconds on a 2.9GHz Intel i7. All the footstep plans shown in this chapter are the result of convergence to within 0.1% or less of the globally optimal cost value, as reported by Gurobi.

Next, we manually generated several 3D example environments using Drake, a software

toolbox for planning and control [27]. These environments and the resulting footstep plans are shown in Figures 2-1, 2-6, 2-8. The IRIS algorithm [7] was used to generate convex regions in the configuration space of a very simple box model of the entire robot, shown in Figure 2-7, in order to avoid collisions between the upper body and the environment. This approach is sufficient to generate rich behaviors such as turning sideways to move through a narrow gap, as in Figure 2-8. Currently, we only ensure that each footstep admits a collision-free posture of the robot, but we do not account for collisions during the transitions between those postures; we will discuss possible ways to address this in Section 2.4.

We have also demonstrated the MIQCQP planner on real terrain, using sensor data collected by Atlas. To generate this plan, we captured LIDAR scans of a stack of cinderblocks like those encountered in the DRC Trials and constructed a heightmap of the scene using the perception tools developed by Team MIT for the DRC [19]. A Sobel filter was used to classify areas of the terrain which were steeper than a predefined threshold [28], and these steep areas were represented as obstacles for the footstep planner. We used the IRIS algorithm [7] to generate seven convex safe terrain regions which covered the terrain of interest in front of the robot. Several footstep plans to different goal poses using these regions are shown in Figure 2-9. Planning times in this environment ranged from 2 seconds or less for plans of 5 footsteps up to 60 seconds for plans of more than 20 footsteps.

2.4 Conclusion

We have demonstrated a novel footstep planning approach, which replaces nonlinear, non-convex constraints with mixed-integer convex constraints. This allows us to solve footstep planning problems to their global optimum, within our linear approximation of rotation. We have inverted the problem of avoiding obstacles into a problem of assigning steps to known convex safe regions, which we can construct efficiently. The primary advantage of our MIQCQP footstep planner is its ability to generate rich footstep sequences in difficult terrain with guarantees of completeness and global optimality. We do not rely on sampling or fixed action sets, which may miss small regions of safe terrain entirely. Our approach also

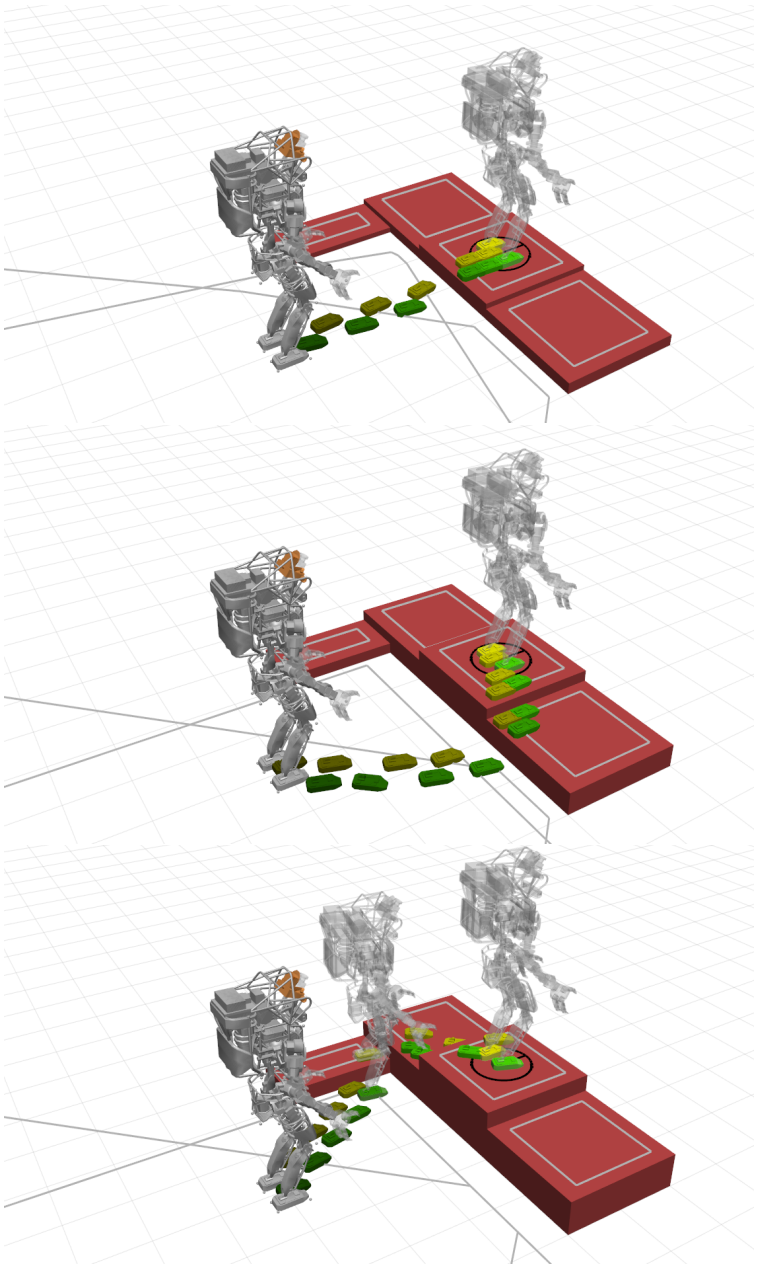


Figure 2-6: Three similar environments, with footstep plans for each. Top: Atlas can mount the center pedestal in one stride. Middle: Raising the center pedestal to twice Atlas' maximum vertical stride forces the robot to detour to the right. Bottom: Raising the pedestals even further requires Atlas to use three platforms to get to the required height. Gray lines are the boundaries of the convex regions of obstacle-free configuration space, generated by IRIS.

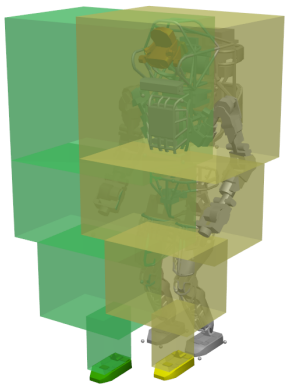


Figure 2-7: Simplified bounding box model of the robot used to plan footstep locations that will be collision-free for the entire robot.

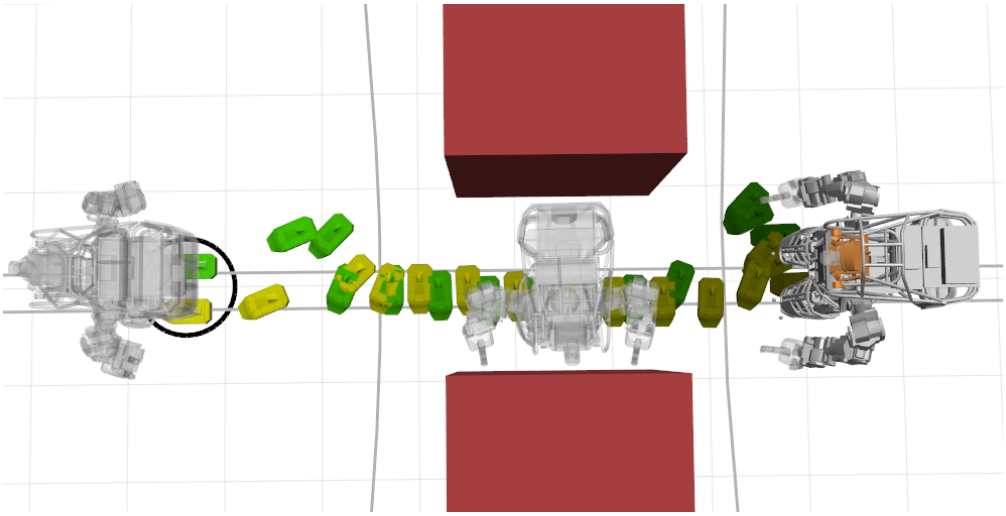


Figure 2-8: A footstep plan through a narrow gap, for which Atlas must turn sideways.

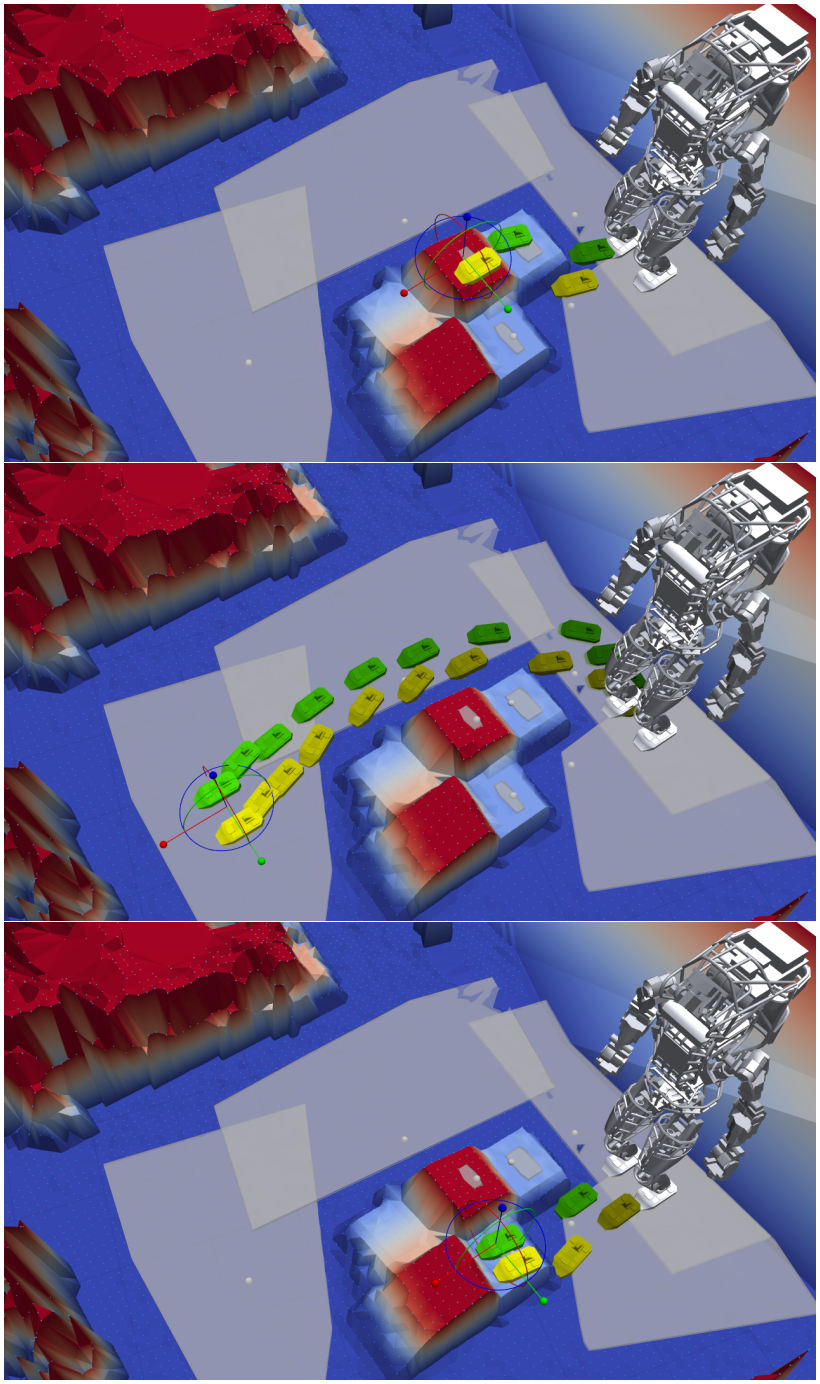


Figure 2-9: Footstep plans for to navigate over and around a set of cinderblocks, using data sensed by the Atlas humanoid. Gray shaded areas are the obstacle-free regions of configuration-space, into which the center of each footstep must be placed.

handles terrain of varying height gracefully, as the same object can be treated as both an obstacle and a walking surface when appropriate, as in Figure 2-6.

On the other hand, we are entirely reliant on the existence of sufficiently many convex obstacle-free regions to ensure that a path through the environment can be found. The IRIS algorithm generates these regions efficiently, but currently requires user input to seed the position of each region. This is a mixed blessing: by seeding such a region, the human operator provides valuable input about the possible walking surfaces for the robot, but this requirement clearly sacrifices autonomy of the robot. In Chapter 3 we will discuss a heuristic for automatically seeding IRIS regions without any human input using a coarse discretization of the space, and in Chapter 4 we will demonstrate its use on a simulated terrain map for footstep planning.

Chapter 3

From Footsteps to Flight Plans

This chapter is adapted from work previously published in [2].

3.1 Introduction

Chapter 2 introduced an approach to planning footsteps on rough terrain: by segmenting the world into convex regions, we could invert the problem of avoiding obstacles into one of assigning foot poses to known safe areas. The need to avoid obstacles, however, is hardly limited to just bipedal robots planning footsteps; it is, instead, a general issue for all mobile robots.

As an example, consider the problem of planning a feasible trajectory for a quadrotor UAV from an initial state to a goal state while avoiding obstacles (Figure 3-1). The problem of obstacle avoidance is particularly challenging because the set of points outside a closed, bounded obstacle is non-convex. As a result, we must generally add non-convex constraints to an optimization in order to ensure that our trajectory remains outside the set of obstacles. Such constraints generally prevent us from finding guarantees of completeness or global optimality in our program [18]. We can avoid some of the problems of non-convex constraints by adding a discrete component to our optimization. If we model the non-convex set of obstacle-free states as the union of finitely many convex regions (analogous to the convex

terrain regions of Chapter 2), then we can perform a mixed-integer convex optimization in which the integer variables correspond to the assignment of trajectory segments to convex regions. This is not without consequences, since even the addition of binary variables (that is, integer variables constrained to take values of 0 or 1) turns linear programming into mixed-$\{0, 1\}$ linear programming, which is known to be NP-complete [29]. However, excellent tools for solving a variety of mixed-integer convex problems have been developed in the past decade, and these tools can often find globally optimal solutions very efficiently for mixed-integer linear, quadratic, and conic problems [21, 22, 23]. These tools also offer some *anytime* capability, since they can be commanded to return a good feasible solution quickly or to spend more time searching for a provably global optimum.

Earlier implementations of mixed-integer UAV obstacle avoidance have typically used the faces of the obstacles themselves to define the convex safe regions. The requirement that a point be outside all obstacles is converted to the requirement that, for each obstacle, the point must be on the outside of at least one face of that obstacle. For convex obstacles these conditions are equivalent [30]. This approach has been successfully used to encode obstacle avoidance for UAVs as a mixed integer linear program (MILP) by Schouwenaars [31], Richards [24], Culligan [32], and Hao [33]. Mellinger et al. add a quadratic cost function to turn the formulation into a mixed-integer quadratic program (MIQP) [30]. However, this approach requires separate integer variables for every face of every obstacle, which causes the mixed-integer formulation to become intractable with more than a handful of simple obstacles.

Instead, we again use IRIS, a greedy tool for finding large convex regions of obstacle-free space [7], to create a small number of convex regions to cover all or part of the free space. These regions can be seeded automatically based on heuristic methods or by an expert human operator. We then need only one integer variable for each region, rather than for each face of every obstacle, and we show in Section 3.3 that it allows us to handle environments with tens or even hundreds of obstacle faces.

3.1.1 Quadrotor Dynamics

While the dynamics of the quadrotor are complex, we can rely on the work Mellinger and Kumar, who demonstrated that the quadrotor system is *differentially flat* with respect to the 3D position and yaw of the vehicle's center of mass [34]. That is, the entire state of the system can be expressed as a function of the instantaneous value of the x, y, z positions of the CoM and the yaw of the vehicle, along with their derivatives. As a result, any smooth trajectory with sufficiently bounded derivatives can be executed by the quadrotor. This means that we are free to design smooth trajectories of the position and yaw of the vehicle's center of mass without explicitly considering the dynamics.

We define a trajectory as a piecewise polynomial function in time with vector-valued coefficients, mapping time to position in 2D or 3D space. We choose the degree of the polynomials and the number of pieces offline. Our optimization problem is a matter of choosing the coefficients of each polynomial in order to ensure that the trajectory reaches a desired goal state, avoids collisions, and satisfies an objective function of our choosing.

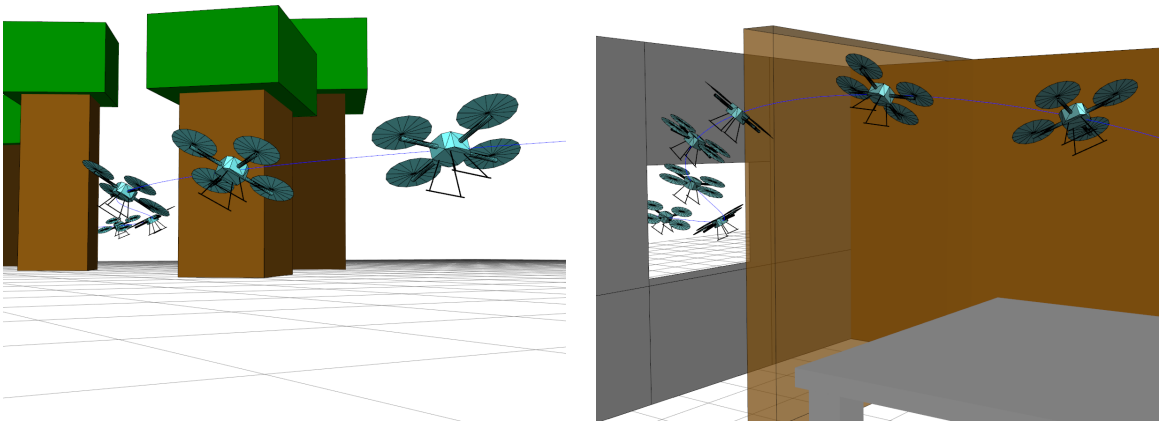


Figure 3-1: Fully collision-free trajectories for a quadrotor UAV in a simulated forest and office environment, showing 1.6s of planned trajectory execution.

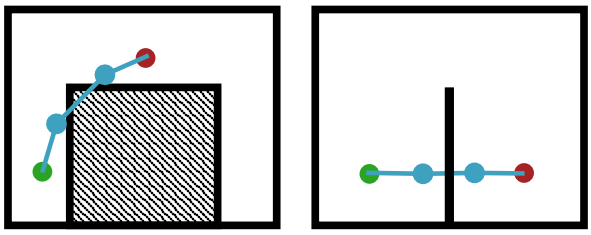


Figure 3-2: A piecewise linear trajectory between two points, with obstacle avoidance enforced only at 4 points along each trajectory. The continuous trajectory through those points may cut corners or pass through thin obstacles.

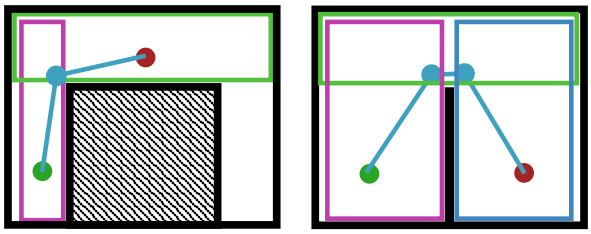


Figure 3-3: A trajectory in which each linear segment is required to remain entirely within one of the convex obstacle-free regions indicated by the colored boxes. This requirement ensures that the entire trajectory is collision-free.

3.1.2 Safety of the Entire Trajectory

A further advantage of our convex region approach is the ability to ensure that the polynomial trajectory for the UAV is obstacle-free over its entire length, rather than at a series of sample points. Existing mixed-integer formulations have chosen only to enforce the obstacle-avoidance constraint at a finite set of points [30, 31, 24]. This can result in the path between those sample points cutting the corners of obstacles, or, more dangerously, passing through very thin obstacles, as shown in Figure 3-2. As noted by Bellingham [35], the severity of the corner cuts can be reduced by increasing the number of sample points and limiting the total distance between adjacent samples, but this also increases the complexity of the optimization problem. Mellinger approaches this problem by requiring that the bounding boxes of the UAV at adjacent sample points must overlap [30]. This is sufficient to ensure that the UAV never passes entirely through an obstacle, but it does not necessarily prevent corner cutting.

Representing the environment with convex safe regions allows us to completely elimi-

nate the cutting of corners. If we treat the problem as one of assigning entire segments of trajectories, rather than points, to the safe regions, then we can create a fully collision-free trajectory. For piecewise linear trajectories, this is simply a matter of ensuring that, for each linear segment, both endpoints must be contained within *the same* convex safe region, as shown in Figure 3-3. This decision weakens our claim of optimality, since it requires the breakpoints between trajectory segments to occur in the intersection of two convex regions, but it results in a formulation that can be solved exactly with mixed-integer programming. We enforce the constraint that each polynomial lie within a convex region using a sums-of-squares (SOS) approach. In this way, collision-free trajectories can be generated using piecewise polynomials of arbitrary degree. Here we show that, for trajectory segments defined by polynomials, we can exactly enforce the constraint that each segment lies inside a convex region by solving a small semi-definite program (SDP).

This is somewhat similar to work by Flores, who uses non-uniform rational B-splines to generate trajectories which are completely contained within convex polytopes [36]. These polytopes must be given as an ordered union of only pairwise adjacent sets, but the trajectories are guaranteed to be contained within the polytopes by construction. Since the polytopes must be pairwise adjacent, they must be laid out along a single path from start to goal by some other planning procedure, and the resulting trajectories may not leave this path. On the other hand, by performing our mixed-integer optimization, we are able to explore arbitrarily connected polytopes which may admit many different possible paths through them.

3.2 Technical Approach

The trajectory planning problem we propose has three components: (1) generating convex safe regions, (2) assigning trajectory segments to those convex regions and (3) optimizing the trajectories while ensuring that they remain fully within the safe regions. We perform step (1) as a pre-processing stage, then solve (2) and (3) *simultaneously* in a single mixed-integer convex optimization, which can be solved to global optimality.

3.2.1 Generating Convex Regions of Safe Space

In the same way that we segmented terrain into convex safe regions in Chapter 2, we can use the IRIS (Iterative Regional Inflation by Semidefinite programming) algorithm to construct convex 3D volumes of free space. While the footstep planner IRIS segmentation operated in a configuration space and avoided regions of steep terrain, the IRIS segmentation in this work operates directly in 3D Cartesian space and simply avoids whatever objects are defined in the environment. Each run of IRIS produces a single convex safe volume from a single seed point, and additional runs of IRIS with different seed points produce additional obstacle-free regions.

The applicatiosn of IRIS to the footstep planning problem in Chapter 2 required a human operator to manually place the seed points. Human input was valuable for that problem, since the choice of seed locations allowed an expert operator to provide high-level input such as which surfaces should be used for stepping. Manually seeded regions are, of course, also possible in the case of an aerial vehicle, and we expect that having an expert operator choose the location of the seeds might be beneficial when the environment is largely static and known beforehand. However, this requirement is overly restrictive in the general case.

For that reason, we have developed a simple heuristic for automatically seeding IRIS regions with no human input. First, the space is discretized into a coarse grid. We then choose the grid cell which maximizes the distance to the nearest obstacle and seed one IRIS region at that point. We then repeat, choosing a new seed which maximizes the distance to the nearest obstacle *or* existing IRIS region. Currently, we terminate the algorithm when a pre-defined number of regions have been generated, but we intend to explore other stopping criteria, such as a threshold on the volume of the space which is currently unoccupied by obstacles or regions. Automatic seeding of IRIS regions is shown in Section 3.3.

3.2.2 Searching over Assignments of Polynomials to Regions

We encode the assignment of each polynomial piece of the trajectory to a safe region using a matrix of binary integer variables $H \in \{0,1\}^{R \times N}$, where R is the number of regions and N

is the number of polynomial trajectory pieces. The polynomial trajectory pieces are labeled as $P_j(t)$ and the convex regions as G_r. Thus, we have:

$$H_{r,j} \implies P_j(t) \in G_r \quad \forall t \in [0,1] \tag{3.1}$$

We arbitrarily choose the range of $[0,1]$ for simplicity in this discussion, but any desired time span can be chosen when constructing the problem. The actual time spent executing each trajectory segment can also be adjusted as a post-processing step by appropriately scaling the coefficients.

Ensuring that polynomial j is collision-free is expressed with a linear constraint on H:

$$\sum_{r=1}^{R} H_{r,j} = 1 \tag{3.2}$$

Note that we allow the regions to overlap, so it is possible for a polynomial to simultaneously exist within multiple regions G_r. Such a case is allowed by our formulation, since the implication in (3.1) is one-directional (so polynomial $P_j(t)$ being contained in G_r does not necessarily require that $H_{r,j} = 1$).

We show in Section 3.2.3 that the constraint $P_j(t) \in G_r \quad \forall t \in [0,1]$ is convex, and we can use a standard big-M formulation [25] to convert the implication in (3.1) to a linear form.

3.2.3 Restricting a Polynomial to a Polytope

We represent our trajectories in n dimensions as piecewise polynomials of degree d in a single variable, t. Each segment j of the trajectory is parameterized by $d+1$ vectors of coefficients $C_{j,k} \in \mathbb{R}^n$ of a set of polynomial basis functions, $\Phi_1(t), \ldots, \Phi_{d+1}(t)$. For each segment j, the trajectory can be evaluated as

$$P_j(t) = \sum_{k=1}^{d+1} C_{j,k}\Phi_k(t) \quad t \in [0,1] \tag{3.3}$$

We restrict the R convex regions of safe space to be polytopes, so for each region $r \in 1, \ldots, R$ we have some $A_r \in \mathbb{R}^{m \times n}$ and $b_r \in \mathbb{R}^m$ and the constraint that

$$A_r \, P_j(t) \leq b_r \tag{3.4}$$

if $H_{r,j}$ is set to 1. To ensure that the trajectory remains entirely within the safe region, we require that (3.4) hold for all $t \in [0, 1]$

$$A_r \sum_{k=1}^{d+1} C_{j,k} \Phi_k(t) \leq b_r \quad \forall t \in [0, 1]. \tag{3.5}$$

Equation 3.5 consists of m constraints of the form

$$a_{r,\ell}^\top \sum_{k=1}^{d+1} C_{j,k} \Phi_k(t) \leq b_{r,\ell} \quad \forall t \in [0, 1] \tag{3.6}$$

where

$$A_r = \begin{bmatrix} a_{r,1}^\top \\ a_{r,2}^\top \\ \vdots \\ a_{r,m}^\top \end{bmatrix} \text{ and } b = \begin{bmatrix} b_{r,1} \\ b_{r,2} \\ \vdots \\ b_{r,m} \end{bmatrix}. \tag{3.7}$$

We can redistribute the terms in (3.6) to get

$$\sum_{k=1}^{d+1} (a_{r,\ell}^\top C_{j,k}) \Phi_k(t) \leq b_{r,\ell} \quad \forall t \in [0, 1] \tag{3.8}$$

and thus

$$q(t) := b_{r,\ell} - \sum_{k=1}^{d+1} (a_{r,\ell}^\top C_{j,k}) \Phi_k(t) \geq 0 \quad \forall t \in [0, 1]. \tag{3.9}$$

The condition that $q(t) \geq 0 \ \forall t \in [0,1]$ holds if and only if $q(t)$ can be written as

$$q(t) = \begin{cases} t\sigma_1(t) + (1-t)\sigma_2(t) & \text{if } d \text{ is odd} \\ \sigma_1(t) + t(1-t)\sigma_2(t) & \text{if } d \text{ is even} \end{cases} \tag{3.10}$$

$$\sigma_1(t),\ \sigma_2(t) \text{ are sums of squares} \tag{3.11}$$

where $\sigma_1(t)$ and $\sigma_2(t)$ are polynomials of degree $d-1$ if d is odd and of degree d and $d-2$ if d is even [37]. The condition that $\sigma_1(t), \sigma_2(t)$ are sums of squares requires that each can be decomposed into a sum of squared terms, which is a necessary and sufficient condition for nonnegativity for polynomials of a single variable [37]. The coefficients of the polynomials σ_1 and σ_2 are additional decision variables in our optimization, subject to linear constraints to enforce (3.10). The sum-of-squares constraints in (3.11) can be represented in general with a semidefinite program [38]. The problem of assigning the trajectories to safe regions is thus a mixed-integer semidefinite program (MISDP). This class of problems can be solved to global optimality using, for example, the Yalmip branch-and-bound solver [39] combined with a semidefinite programming solver like Mosek [22] or using the dedicated SDP package developed by Mars and Schewe [40]. We have successfully applied this formulation to polynomials of degree 1, 3, 5, and 7. For polynomials of degree 7 and higher, we experienced numerical difficulties which often prevented Mosek from solving the semidefinite program. As a result, we have developed more numerically stable exact reductions for lower degree polynomials.

For polynomials of degree 1, σ_1 and σ_2 are constants, and the condition in (3.11) reduces to linear constraints

$$\sigma_1 \geq 0,\ \sigma_2 \geq 0, \tag{3.12}$$

which reduces the problem to a mixed-integer quadratic program (MIQP), given our quadratic objective function.

If the polynomials are of degree 3, then $\sigma_i(t)$ is a quadratic polynomial:

$$\sigma_i(t) = \beta_1 + \beta_2 t + \beta_3 t^2. \tag{3.13}$$

Using the standard sum-of-squares approach, we rewrite $\sigma_i(t)$ as

$$\sigma_i(t) = \begin{bmatrix} 1 & t \end{bmatrix} \begin{bmatrix} \beta_1 & \frac{\beta_2}{2} \\ \frac{\beta_2}{2} & \beta_3 \end{bmatrix} \begin{bmatrix} 1 \\ t \end{bmatrix}. \tag{3.14}$$

The condition that $\sigma(t)$ is SOS is equivalent to the matrix of coefficients in (3.14) being positive semi-definite:

$$\begin{bmatrix} \beta_1 & \frac{\beta_2}{2} \\ \frac{\beta_2}{2} & \beta_3 \end{bmatrix} \succeq 0, \tag{3.15}$$

which is in turn equivalent to the following rotated second-order cone constraint:

$$\beta_2^2 - 4\beta_1\beta_3 \leq 0 \tag{3.16}$$

$$\beta_1, \beta_3 \geq 0 \tag{3.17}$$

We can thus write the problem of assigning degree-3 polynomials to convex regions as a mixed-integer second-order cone problem (MISOCP), which we can solve effectively with Mosek [22], Gurobi [21], and other tools.

3.2.4 Choosing an Objective Function

Mellinger et al. relate the snap (that is, the fourth derivative of position) to the control inputs of a quadrotor, and thus choose an objective of the form:

$$\text{minimize} \sum_{j=1}^{N} \int_0^1 \left\| \frac{d^4}{dt^4} P_j(t) \right\|^2 dt \tag{3.18}$$

If $p_j(t)$ is of degree $d \geq 4$ then we may do likewise, resulting in a convex quadratic objective on the coefficients of the P_j. We demonstrate this objective function in operation with 5^{th}-degree polynomials in Figure 3-4.

However, to reduce our problem to a MISOCP and improve the numerical stability of the solver, we found it beneficial to restrict ourselves to 3^{rd}-degree polynomials and thus piecewise constant jerk. Our objective is

$$\text{minimize} \sum_{j=1}^{N} \left\| \frac{d^3}{dt^3} P_j(t) \right\|^2 . \tag{3.19}$$

which is likewise convex and quadratic in the coefficients of the P_j. We also add linear constraints on the position, velocity, and acceleration of each trajectory piece to ensure that they are continuous from one polynomial piece to the next. Additional linear equality constraints require that the position, velocity, and acceleration of the beginning of the first trajectory piece and the end of the last piece match our desired initial and final states.

3.2.5 Handling Lower-Degree Trajectories

Even if the mixed-integer optimization is done over the numerically easier degree 3 polynomials, we can post-process the resulting trajectories in order to successfully use the differential flatness of the system to derive the full state and input. A piecewise degree-3 trajectory has a piecewise constant 3^{rd} derivative. It thus has delta functions for its 4^{th} derivative, which Mellinger relates directly to the rotor thrusts of the UAV [34]. Since this is clearly undesirable, we proceed as follows: First, we run the MISOCP to optimize our degree-3 polynomials and assign them to convex safe regions. Next, we fix the resulting assignment of trajectories to safe regions and then re-run the optimization for polynomials of degree 5 or higher while minimizing the squared norm of the snap. Since all of the integer variables are fixed, we no longer have a mixed-integer problem but instead a single semidefinite program, which can be solved very efficiently. For example, in the office environment shown in Figure 3-1, computing the smooth 5^{th}-degree polynomial trajectory required only 1.5 s in Mosek.

3.2.6 Complete Formulation

Our optimization problem can be written as follows for a trajectory of N piecewise 3^{rd}-degree polynomials:

$$\underset{P,H,\sigma}{\text{minimize}} \sum_{j=1}^{N} \left\| \frac{d^3}{dt^3} P_j(t) \right\|^2 \tag{3.20}$$

subject to:

$$\begin{aligned} &P_1(0) = X_0, && \dot{P}_1(0) = \dot{X}_0, && \ddot{P}_1(0) = \ddot{X}_0 \\ &P_N(1) = X_f, && \dot{P}_N(1) = \dot{X}_f, && \ddot{P}_N(1) = \ddot{X}_f \\ &P_j(1) = P_{j+1}(0), && \dot{P}_j(1) = \dot{P}_{j+1}(0), && \ddot{P}_j(1) = \ddot{P}_{j+1}(0) \end{aligned} \tag{3.21}$$

$$\begin{aligned} &H_{r,j} \implies b_{r,\ell} - a_{r,\ell}^\top P_j(j) = t\sigma_{\ell,j,1}(t) + (1-t)\sigma_{\ell,j,2}(t) \\ &\quad \forall j \in \{1, \ldots, N\} \quad \forall r \in \{1, \ldots, R\} \\ &\quad \text{where } \sigma_{\ell,j,1}(t),\ \sigma_{\ell,j,2}(t) \text{ are sums of squares} \end{aligned} \tag{3.22}$$

$$\sum_{r=1}^{R} H_{r,j} = 1 \quad \forall j \in \{1, \ldots, N\} \tag{3.23}$$

$$H_{r,j} \in \{0, 1\} \tag{3.24}$$

where $X_0, \dot{X}_0, \ddot{X}_0$ are the initial position, velocity, and acceleration of the vehicle and $X_f, \dot{X}_f, \ddot{X}_f$ are the final values. All of the above conditions are linear constraints on the coefficients C and β and the matrix H, except the condition that σ_1 and σ_2 are sums of squares, which is a rotated second-order cone constraint.

3.2.7 Trajectories Without Convex Segmentation

For the sake of comparison, we have included numerical experiments for the proposed mixed-integer optimization using the faces of obstacles instead of convex regions from IRIS, just as Bellingham [35], Mellinger [30], and others have done. Instead of a single matrix of binary

variables H, we have a matrix H_o for each obstacle. The m linear constraints in (3.5) are replaced with a linear constraint for each face r of obstacle o, and we constrain that

$$\sum_{r=1}^{N_{\text{faces}}} H_{o,r,j} = 1 \quad \forall j \in \{1, \ldots, N\} \tag{3.25}$$

for every obstacle $o \in \{1, \ldots, N_{\text{obstacles}}\}$. The disadvantage of this formulation is the rapid increase in the number of binary variables as the numbers of obstacles and faces increase. This increases the time required to find the global optimum, as shown in Figure 3-6.

3.3 Results

3.3.1 Simulation

We demonstrate the mixed-integer trajectory planning in a variety of two- and three-dimensional environments. Figure 3-4 shows a simple 2D environment, in which we find four convex safe regions with IRIS and then solve for several trajectories through those regions. The arrangement of the obstacles in the simple environment is such that only four convex regions are needed to completely fill the space. For more complex environments, such as that shown in Figure 3-5, more convex regions may be required, and those regions may not cover the entire space.

In Figure 3-5, we show a randomly generated environment with five obstacles, a starting pose in the upper right, and a goal pose in the lower left. We generate 7 convex safe regions with IRIS, which do not entirely fill the obstacle-free space. Within those 7 regions, we plan 6 polynomial segments of degree 3 to form a smooth trajectory from the start to the goal while minimizing the objective in (3.19). This trajectory is shown in Figure 3-5a. We can also avoid pre-computing convex regions and use the faces of the obstacles directly, as shown in Figure 3-5b. This results in an optimal solution with a 15% lower (i.e. better) objective function value, but requires nearly double the solver time for the example shown. The solutions in Figures 3-5a and 3-5b are both globally optimal with respect to the cost

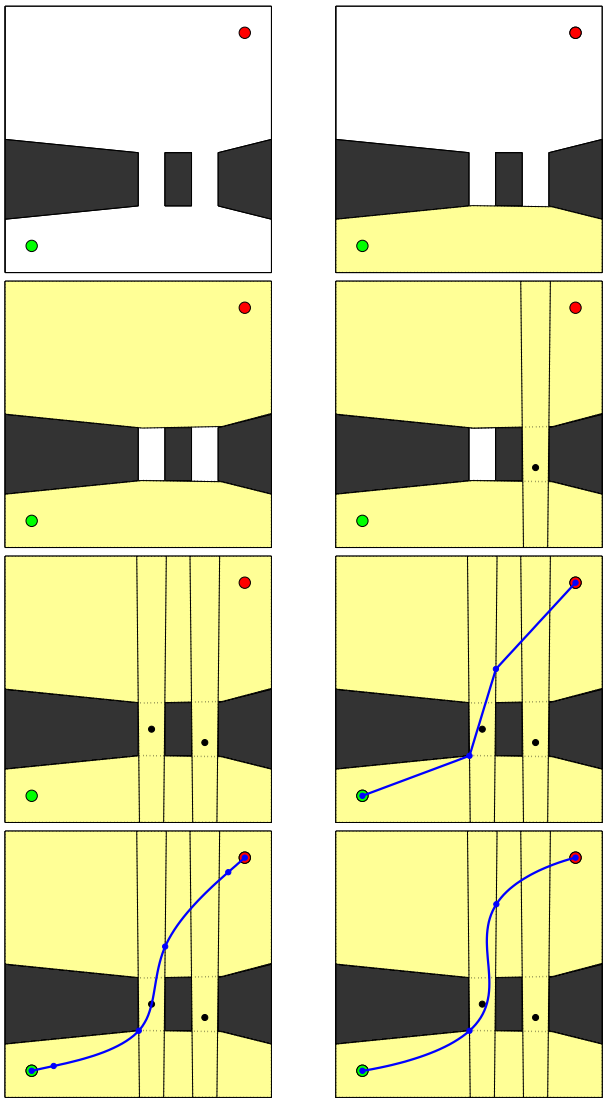
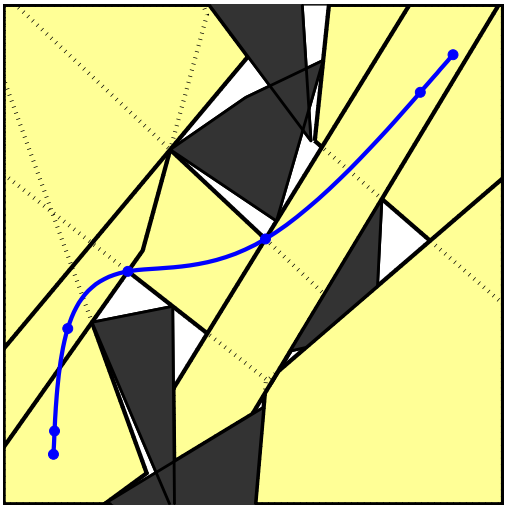
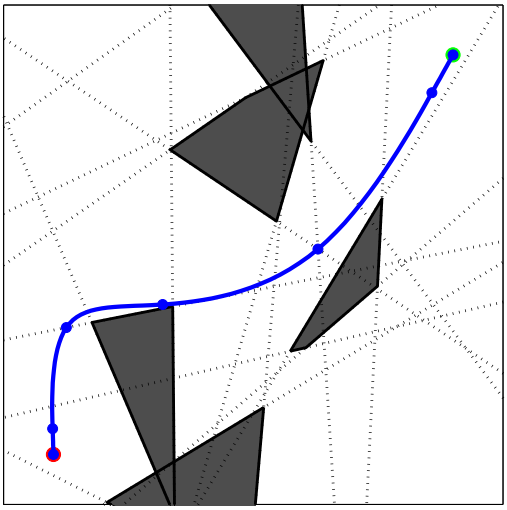


Figure 3-4: Solving a simple environment with IRIS regions. We construct an environment with obstacles and a start and goal pose (a), then generate IRIS regions around the start (b) and goal (c). Next, we identify a point far from the existing set of obstacles and IRIS regions and seed a new region there (d), and repeat until we have 4 regions (e). Finally, we solve for trajectories of 1^{st}-degree polynomials minimizing squared velocity in 0.1s (f), 3^{rd}-degree polynomials minimizing squared jerk in 1.3s (g), and 5^{th}-degree polynomials minimizing squared snap in 4.0s (h). All trajectories lie entirely within the convex regions shown.



(a) With IRIS convex segmentation.



(b) Without IRIS convex segmentation.

Figure 3-5: An environment consisting of 5 uniformly randomly placed convex obstacles. Above: We generate 7 convex regions of free space using IRIS, then solve for a piecewise 3^{rd}-degree trajectory which is entirely contained within those safe regions in 14.1s. Below: we also solve for a trajectory of the same degree without the convex segmentation step, which results in a 15% decrease in the optimal cost value but requires 27.5s to solve to optimality.

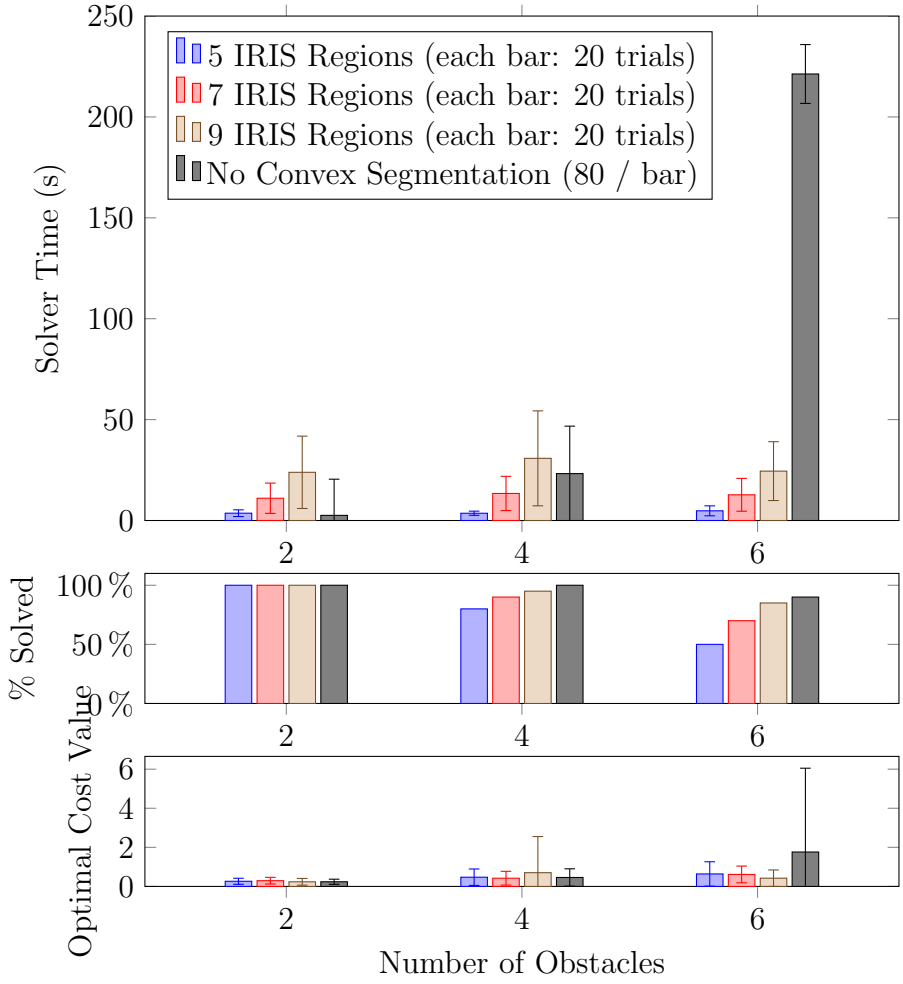


Figure 3-6: Comparison of our approach for various numbers of convex safe regions, as well as the approach described in Section 3.2.7, which does not require convex segmentation. We show results for randomly generated environments with 2, 4, or 6 obstacles. Above: the mean and std. dev. of time required to solve the problem to within 1% of global optimality using Mosek [22] on a 2.7 GHz 12-Core Intel Xeon E5 processor. For more than 4 obstacles, the solve time required increases dramatically if no convex segmentation is performed. Middle: the fraction of environments for which an optimal solution could be found. Reducing the number of IRIS regions improves the speed of optimization but, by covering less of the free space, it also decreases the likelihood of a feasible trajectory from start to goal being found. Below: the final value of the objective function at optimality in each case.

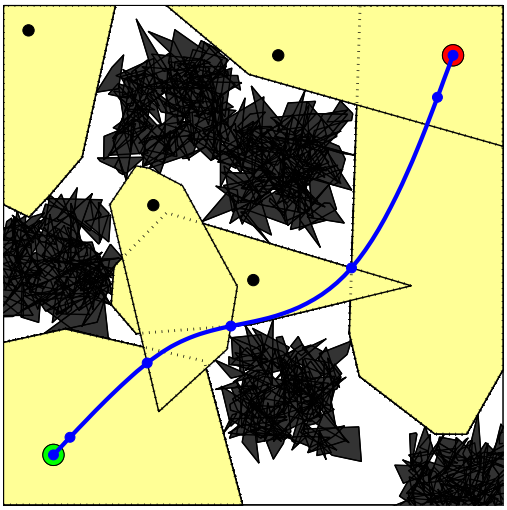


Figure 3-7: Collision-free trajectory with very many obstacles. Five clusters of 100 obstacles each were placed, and 6 convex regions were found with IRIS in the free space. Solve time for this trajectory was 20s. Generating IRIS regions required $<$ 1s.

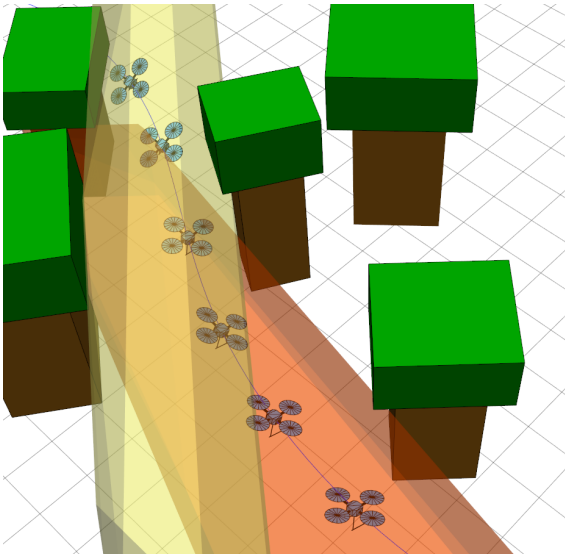


Figure 3-8: A virtual forest environment showing two obstacle-free IRIS regions and a trajectory which passes through those regions. The UAV itself is treated as a sphere to ensure that no part of the vehicle is allowed to exit the set of obstacle-free regions.

function in (3.19), but they differ from one another because they are subject to different safe region constraints. In Figure 3-5a, each polynomial must be contained entirely within one of the seven convex safe regions shown, while in Figure 3-5b, each polynomial must lie on the outside of one face of every obstacle.

We compare the approach of generating convex safe regions (which may fail to fill the entire free space) with the approach of using the obstacle faces directly as our safe regions (which may require a dramatically more complex integer program) in Figure 3-6. Environments consisting of 2, 4, or 6 obstacles were generated, and 5, 7, or 9 IRIS regions were automatically created using the heuristic described in Section 3.1. Time spent on IRIS segmentation is not included in the table, but was less than 1 s in all cases. For each environment, we optimized a trajectory of 6 polynomial pieces of degree 3, while minimizing the objective in (3.19). We also attempted to find a trajectory in each environment using the method of Section 3.2.7 with no IRIS regions, which tended to be much slower for more than 4 obstacles. One such environment is shown in Figure 3-5.

Substantially more complex environments can also be handled by the technique introduced here. In Figure 3-7, we generate an environment of 500 obstacles in 5 clusters. Although we cannot in general hope to fully explore all possible paths through all 500 obstacles, our ability to quickly find large open areas of space with IRIS allows us to find a collision-free trajectory even in this extremely cluttered space.

We are by no means limited to problems in two dimensions. In Figure 3-1, we show trajectories generated in two different 3D environments. For each environment, we generated 7 to 9 regions of safe space in 3D with IRIS, then planned a trajectory consisting of 7 degree-3 polynomials assigned to those safe regions. Each trajectory took approximately 200 s to solve to within 1% of the globally optimal objective value on an Intel i7 at 2.9GHz. The convex regions were generated for the configuration space of a bounding sphere for the UAV in order to ensure that the trajectory would be collision-free for the whole vehicle. Figure 3-8 shows two of the convex regions used in the forest environment, along with part of the trajectory through those regions.

3.4 Conclusion

In this chapter, we have presented a new method for optimal trajectory planning around obstacles which ensures that the entire path is collision-free, rather than enforcing obstacle avoidance only at a set of sample points. This method is formulated as a mixed-integer convex program and can be directly used with the mixed-integer obstacle avoidance approach which is already common in the field. Performance of our approach can be significantly improved by pre-computing convex regions of safe space with IRIS, a tool for greedy convex segmentation, which can allow us to solve for trajectories even in very cluttered environments.

3.4.1 Limitations

By requiring that each polynomial trajectory piece lie entirely within one convex safe region, we disallow trajectories which may not intersect the obstacles but which pass through several safe regions. Our claims of global optimality are also limited to trajectories which obey this restriction. This problem can be alleviated by increasing the number of trajectory segments so that each segment can be assigned to a single safe region, but doing so increases the complexity of the mixed-integer program. This limitation also exists in the footstep planning work on which this is based: the requirement that each footstep lie completely within some convex region eliminates potentially safe plans in which a footstep spans multiple convex regions.

Successful trajectory generation is also dependent on the particular set of convex regions which are generated. In the environments shown in Figures 3-4, 3-5, and 3-7 and in the forest environment shown in Figure 3-1, automatically finding regions at points far from the obstacles was sufficient, but as the environment becomes more complex, we may require a more intelligent method of selecting the seed points at which the IRIS algorithm begins. Input from a human operator can be extremely helpful in this case: in the office environment shown in Figure 3-1, a human operator indicated the interiors of the window and doorway as salient points at which to generate convex regions, which allowed a feasible trajectory to be found with less time spent blindly searching for good region locations.

Finally, in order to ensure a smooth control input, we may wish to constrain the derivatives of the snap of the trajectory, which will require polynomials of degree 5 or higher. This will require a more careful approach to ensure the numerical stability of the mixed-integer semidefinite program. We were not able to reliably solve these high-order problems using Yalmip and Mosek without encountering numerical difficulties. The choice of basis functions Φ in Equation 3.3 is likely to be a significant factor in the numerical stability of the solver [30]. So far, we have experimented with the Legendre basis suggested by Mellinger et al., but not with other polynomial bases.

Chapter 4

Analysis: Convex Segmentation and Mixed-Integer Planning

The approaches to locomotion planning presented in Chapters 2 and 3 are quite similar: we decompose a cluttered environment into convex save regions (using IRIS), then solve a hard mixed-integer optimization to find an optimal path, including assignments of parts of that path to their corresponding safe regions. This approach worked well for the controlled examples presented in those chapters and was even scaled up to the full Atlas hunanoid robot walking on completely unmapped terrain, but it still proved difficult to use outside of the lab. Unmodeled dynamics, perception difficulties, and a variety of other violated assumptions meant that the mixed-integer footstep planner could not be reliably used on the most difficult tasks seen at the finals of the DARPA Robotics Challenge.

In this chapter, I will discuss the successful applications of the mixed-integer optimization techniques beyond the initial results shown in Chapters 2 and 3, including Atlas walking on rough terrain and a real UAV executing optimally-planned trajectories. In Section 4.2, I will explore some of the specific challenges that made this kind of planning difficult to use reliably in the field. Fortunately, this is not the end of the story, as Section 4.2.3 will demonstrate how recent papers have built upon the work presented here to give better performance, autonomy, and extension to new domains.

4.1 Successes

The mixed-integer planning approaches for UAVs and footsteps were both successfully demonstrated in simulation and in hardware, despite the inevitable challenges of bringing theoretical work into practice. In the case of the footstep planner, this required implementing an entire pipeline to from raw perception data to optimal footstep locations, while in the UAV case it required generating and tracking trajectories in complex physical environments.

4.1.1 Simulating the Footstep Planning Pipeline

To evaluate the entire footstep planning pipeline proposed in Chapter 2, including the convex segmentation, footstep placement, and whole-body control, a set of self-contained examples was implemented in the Drake software framework [27]. The experiments consisted of the following steps:

1. An environment consisting of cinder blocks (represented as rectangular prisms) was created manually to match the terrains seen at the DARPA Robotics Challenge.

2. A heightmap was generated automatically by raycasting the simulated terrain from above.

3. Each cell of the heightmap was classified as safe or unsafe by applying a Sobel filter to estimate the local gradient [28]. The IRIS algorithm was used to find convex regions containing only safe cells, using the automated seeding procedure described in Section 3.2.1.

4. A footstep plan was generated, constrained to use only the convex safe regions, using the optimization from Section 2.2.4.

5. A center of mass trajectory was constructed using the method in [41] and executed in simulation using the controller from [6].

Figure 4-1 shows a complete execution of such an example, in which the Atlas robot crosses a terrain course consisting of multiple cinderblock obstacles in approximately 90 seconds of simulated walking.

The successful application of IRIS and the mixed-integer footstep planner, however, relied on a number of assumptions whose importance became clear with further experimentation:

1. The simulated heightmap was assumed to be perfect, allowing for very accurate classification into safe and unsafe cells.

2. The simulated robot model was assumed to be perfect, with no unmodeled friction or restrictions on the joint motions, and the joint actuators were assumed to be perfect torque sources.

3. The simulated state estimate was assumed to be perfect, so there was no uncertainty in the robot's position or velocity in the world.

The effects of these assumptions became clearer as we moved from a simulated terrain course to experiments with the real Atlas robot.

4.1.2 Walking on Unmapped Terrain

To evaluate the performance of the footstep planner in a real environment, we constructed a terrain course consisting of a long row of cinderblocks with a variety of upward and downward steps and some small gaps. The complete terrain course can be seen in Figure 4-2. With no prior model of the terrain, the robot walked across the terrain course, performing the following actions in a receding horizon manner at every footstep:

1. Depth data from the robot's stereo cameras was fused into a 3D world map using the Kintinuous algorithm [43].

2. Convex regions corresponding to the surfaces of the cinder blocks were generated.

3. The mixed-integer footstep planner generated a sequence of 2-6 footsteps among those safe regions in the direction of travel.

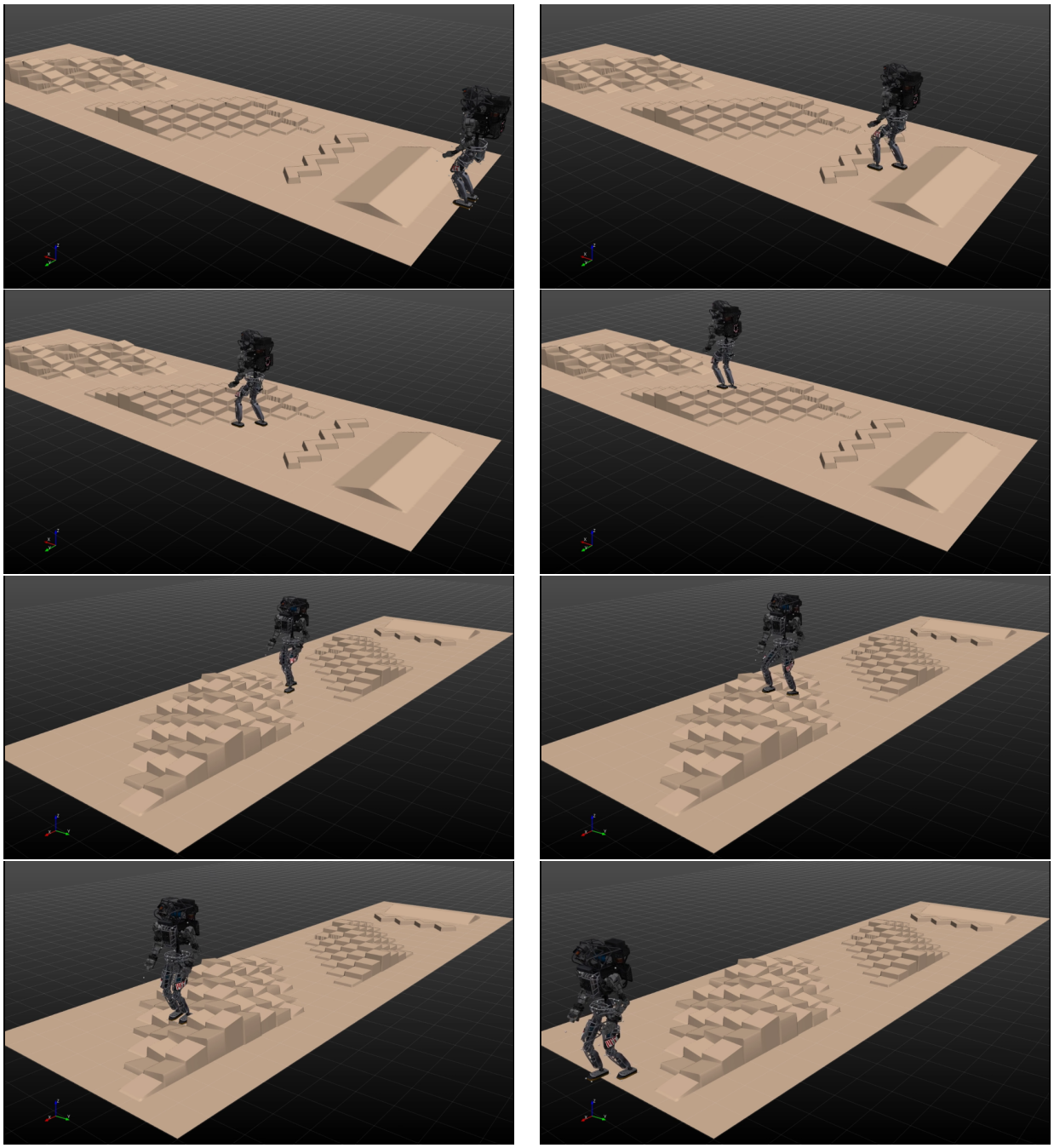


Figure 4-1: The Atlas robot autonomously navigating over simulated rough terrain. Given just the heightmap and a goal pose, IRIS regions were segmented automatically and the mixed-integer footstep planner produced sequences of safe footholds for the robot to follow. Each frame represents 12 seconds of simulated walking.

Figure 4-2: The cinderblock terrain used in the autonomous footstep planning experiment described in Section 4.1.2. The Atlas robot was able to navigate autonomously across this terrain using the mixed-integer footstep planning approach described in Chapter 2. Reproduced from [42].

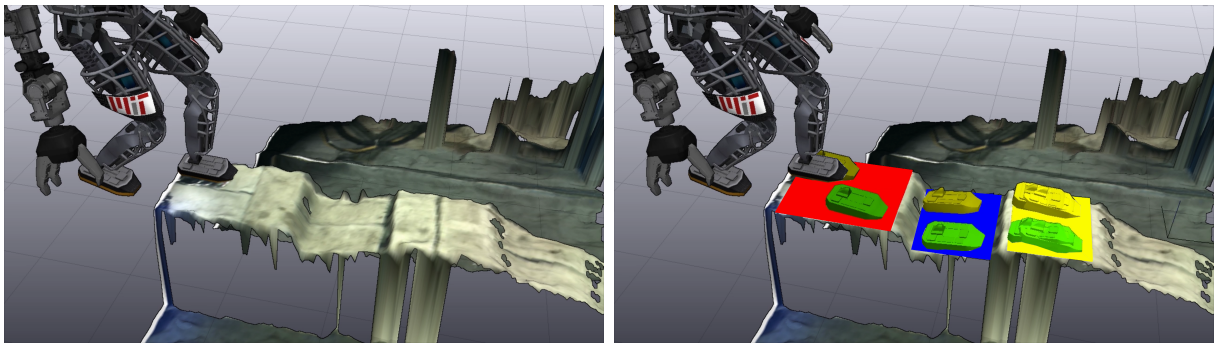


Figure 4-3: A sample footstep plan from the autonomous walking experiment. Left: the terrain map in front of the robot, reconstructed from fusion of stereo depth data. Right: the convex regions of safe terrain, computed via RANSAC, and the footstep placements chosen by the mixed-integer planner.

4. The center-of-mass plan and walking controller were re-computed from the new footstep plan and applied to the robot.

This entire sequence of steps was reliably performed in less than 0.5 s, allowing a new footstep plan to be generated each time the robot took a step.

A full description of the experiment and its components can be found in our earlier publication [42]. Over the course of this experiment, the robot successfully and autonomously navigated terrain for which it had no prior map, using only the mixed-integer footstep planner to choose its contact locations.

There were, however, a number of caveats to that success. In particular, the noisy sensor data made it difficult to apply the IRIS segmentation algorithm (see Section 4.2.1), so a simpler but more reliable RANSAC-based plane-fitting approach was used instead. Furthermore, the lack of any notion of dynamics in the footstep planner made the resulting plans difficult to reliably follow for the robot (see Section 4.2.2). While shorter segments of terrain could be reliably crossed, the full terrain course shown in Figure 4-2 was only completely traversed once during our experiments, with tens of attempts resulting in the robot falling before reaching the end. To generate more reliable behaviors and respond to perception issues, unexpected disturbances, and unmodeled dynamics, we need controllers that can actually reason about their future contact decisions in more detail (see Chapter 5).

4.1.3 UAV Experiments

In addition to the footstep planner from Chapter 2, the mixed-integer UAV planning approach from Chapter 3 was also successfully implemented in hardware. In a paper by Benoit Landry, a Crazyflie quadrotor was flown autonomously through a series of cluttered environments. The environments were constructed manually, and the IRIS algorithm was used to generate convex safe regions from a provided model of each environment. The mixed-integer planner then produced safe piecewise polynomial trajectories, avoiding the modeled obstacles. The trajectories were tracked using a time-varying LQR controller running on the vehicle. Examples of the environments and trajectories can be seen in Figure 4-4, and full

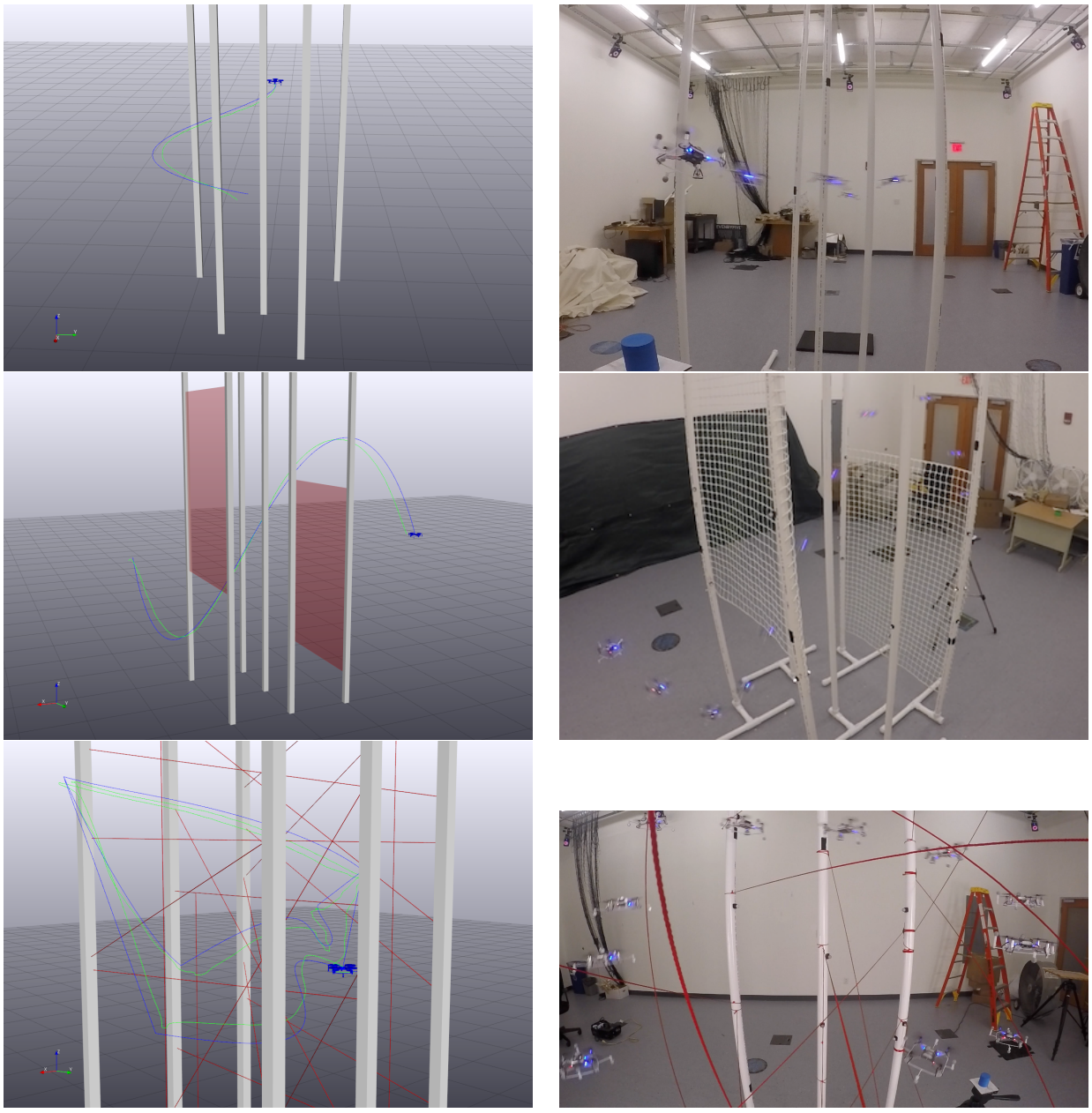


Figure 4-4: The Crazyflie quadrotor executing a piecewise polynomial trajectory generated by our mixed-integer optimization in three environments. Left: the planned trajectories in the modeled environments. Blue lines indicate the planned trajectory and green lines show the executed trajectory, measured with motion capture. Right: execution of the trajectories in hardware. *Figures reproduced from [44].*

details of the experiment can be found in [44].

4.2 Challenges

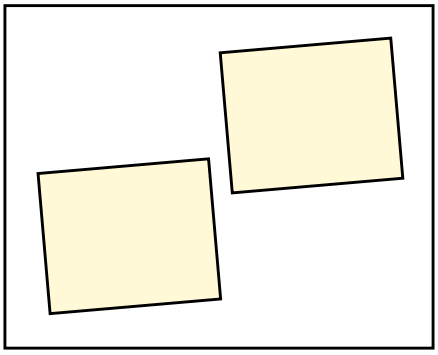
The Atlas robot and Crazyflie UAV experiments showed how effective the process of convex segmentation and optimal mixed-integer planning can be, even on complex physical robots. At the same time, they also showed some of the limitations of the approach, particularly when perfect information about the environment and the robot model are not available.

4.2.1 Perception Challenges and IRIS

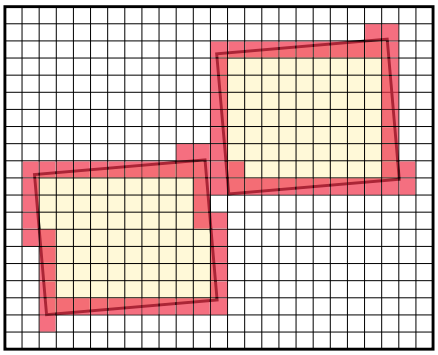
The idea of segmenting the environment into convex obstacle-free regions rests on the assumption that we actually know what the environment consists of, which in general requires some kind of perception system to measure the environment around the robot. In the quadrotor experiments from Section 4.1.3, the issues of perception were entirely avoided by providing a perfect model of the environment to the planner ahead of time, but perfect maps are not something we can generally assume.

In the footstep planning experiments with the Atlas humanoid robot, we assumed that the obstacles (regions of unsafe or steep terrain) could be identified precisely from measured LIDAR or stereo camera data. Failing to classify part of the terrain as unsafe could result in IRIS returning convex regions that are not entirely safe for stepping. Mis-classifying safe regions as unsafe, on the other hand, could result in many more convex regions being necessary to cover a particular terrain (see Figure 4-5), making the combinatorial choice of to which region to assign each footstep much more difficult.

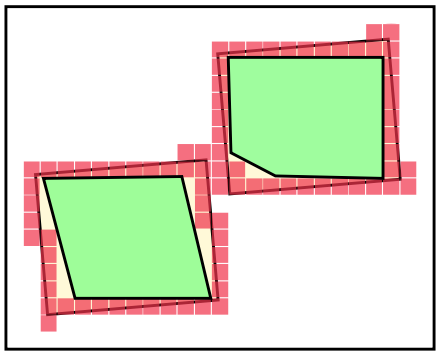
There was also a more subtle issue with the use of IRIS in the footstep planning experiments, in which the resolution of the terrain map significantly affected the success of the planner. The IRIS-based terrain segmentation assumed that the terrain was stored as a map consisting of a rectangular grid of cells. Each cell could be classified as either safe or unsafe for walking, and the IRIS algorithm would return a convex 2D polyhedron which avoided



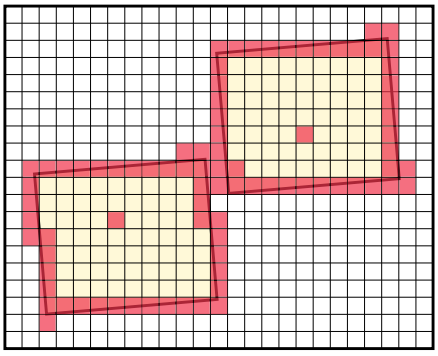
(a) Hypothetical terrain consisting of two rectangular stepping stones.



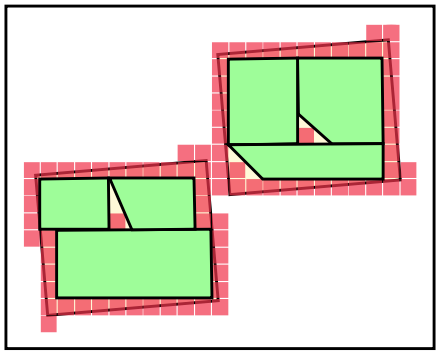
(b) Applying a discretized map grid to the given terrain and classifying cells as safe or unsafe.



(c) Running IRIS using the discretized grid cells as obstacles results in significant underestimation of the available safe terrain.



(d) An example of false detections of unsafe cells due to noise in the sensor data.



(e) The minor misclassifications of the terrain result in significantly more IRIS regions being required to achieve similar coverage.

Figure 4-5: Illustration of difficulties encountered when using IRIS segmentation on potentially noisy terrain data.

any overlap with unsafe cells. This design was intentionally conservative: any obstacle, even one much smaller than a single map cell, would cause the resulting safe regions to avoid the entire cell around that obstacle. The primary sources of unsafe classifications were the vertical edges of the rectangular cinder blocks. Although these unsafe regions were essentially zero-width (being predominantly perfectly vertical), they would result in entire rows and/or columns of cells in the map being labeled as unsafe. The discretization thus had the effect over over-estimating the size of the obstacles by, on average, half the width of a map cell. Consequently, the resulting IRIS convex regions tended to under-estimate the size of the safe terrain by the same amount, with the ultimate result that the robot was unable to plan footsteps that came close to the edges of the terrain. With a default map cell size of 5 cm, the under-estimation caused by the map grid could force the robot to step several centimeters farther than was actually required in order to ensure that the footstep landed within the computed IRIS region. Smaller grid cells were possible, but increasing the map resolution had additional computational costs.

Conservatively keeping the robot's feet farther from the unsafe edges of the cinder blocks would seem to be a reasonable choice, but it interacted poorly with the limitations of the controller (discussed in more detail in Section 4.2.2). The distance from one cinderblock to the next (approximately 25 cm in the experiment shown in Figure 4-2) was just on the boundary of the maximum achievable flat-ground forward step for the controller used in this experiment. Thus, a conservative terrain estimate which added just 5 cm (one map cell) to the length of a forward step could easily result in plans that the controller could not actually follow.

Instead, for the duration of the continuous walking experiment, the IRIS algorithm was shelved in favor of a simpler approach: points in the scene were filtered by their estimated surface normal to extract only the near-horizontal terrain surfaces, and those filtered points were clustered into rectangular patches which could be used as convex safe terrain regions. This simplified terrain classification approach is described in more detail in [42]. While this approach solved the immediate problem of IRIS returning overly conservative regions, it was

only useful in an environment consisting entirely of rectangular planar stepping stones.

4.2.2 Control Challenges

Although the footstep planner was able to create dynamically feasible plans which could be tracked reliably in simulation (see Figure 4-3), executing the planned footsteps on uneven terrain proved to be unreliable. Several cases were particularly challenging in execution and contributed to many of the robot's falls:

Changing Height

The linear inverted pendulum model used by the walking controller described in [41] assumes that the center of mass of the robot undergoes no acceleration in the vertical direction. In order to avoid significantly violating that assumption, any change in height had to be done slowly and smoothly to minimize vertical acceleration. In particular, this meant that any step up or down had to be done extremely slowly, with long periods during which only one foot could remain flat on the ground. While a human can step down quickly, using the impact of the leading foot to dissipate energy, such an impact causes precisely the kind of vertical acceleration that the linear inverted pendulum model forbids. Instead, the robot must use its leg actuators to support itself while slowly dissipating potential energy as it descends, resulting in long periods with the leg joints at or near their maximum rated torque production. The overall result of this was frequent falls as the robot spent long periods of time at its torque limits, even when descending as little as 20 cm at a time. A more human-like gait, exploiting impact to dissipate energy, would likely have avoided the torque limit issue, but simply was not possible while remaining close to the linear inverted pendulum dynamics.

Moreover, since the footstep planner of Chapter 2 had no notion of dynamics (only of whether a given foot position was reachable from some prior foot position), there was no clear way for the planner to account for this limitation of the controller. Heuristically limiting the amount of upward or downward travel the robot was able to make between footsteps

helped, but the limits proved very difficult to tune. A more complete solution would involve planning the timing and location of the robot's footsteps in order to exploit its full dynamics, rather than trying to ignore them at the footstep planning stage.

External Disturbances

A wide variety of unexpected and unmodeled disturbances contributed to the difficulty in executing planned footsteps on rough terrain. The Atlas robot's actuators, while modeled as perfect torque sources, were in fact biased and noisy due to the unmodeled dynamics of the hydraulic systems which powered them. Errors in the torque output had to be corrected with integral control, resulting in the robot's actions lagging behind the commanded signals. The kinematics of the robot were also difficult to model perfectly, as the complex linkages in the joints connecting the robot's legs to its feet resulted in coupled joint limits which were difficult to model in simulation. The coupled joint limits resulted in unexpected internal torques when the ankles were near the limits of their range of motion. Even wind in outdoor environments could create small but significant disturbances of the robot's motion.

None of these disturbances was generally catastrophic on its own, but their effects were magnified by the robot's total inability to correct its behavior by making or breaking contact. Stumbling, skipping, touching a wall, dropping to a knee, or even simply stopping in place would all require a new contact plan, a process which could take several seconds. Thus, the robot's only option was to continue executing its current plan, no matter the disturbance, a strategy which often resulted in falls.

On flat ground, our footstep plans could be made sufficiently conservative and our controllers were sufficiently robust that the robot could tolerate most disturbances without falling and without needing to modify its contact plan. But on rough ground, or when the robot received a significant push, a new strategy was needed to give the robot the ability to make contact decisions on the fly.

The quadrotor hardware experiments, by virtue of being run in a closed lab environment, were relatively free from external disturbances and consequently more reliable. But similar

effects would almost certainly have been seen had the UAV been moved out of the lab. Just as in the case of the footstep planner, the UAV planner produced optimal results too slowly to respond to disturbances online: a new plan is of little use if it is only returned a minute or two after the robot has already been blown off-course.

4.2.3 Extensions and Derivative Works

The convex segmentation and mixed-integer planning approaches presented above have also inspired extensions and modifications by other researchers, with a particular focus on exploring the tradeoffs between optimality, expressiveness, and computation time.

Convex Segmentation

The IRIS algorithm is effective at generating regions of safe terrain and volumes of safe space, but the implementation presented in Chapter 2 required manual intervention to choose the seed point for each region. In Section 3.2.1 we proposed a heuristic for seeding IRIS regions automatically using a coarse discretization of the configuration space. More recently, Sadhu et al. propose a similar heuristic, which they label *Extended IRIS*, to automatically choose seed locations [45]. On the other hand, Savin chooses to keep the seeding process, but introduces a completely different mechanism of generating convex regions, replacing the optimization process in IRIS with a computationally simpler stereographic projection approach and going on to apply this segmentation approach to simulated footstep and UAV planning problems [46]. In [47], Jatsun et al. propose yet another convex segmentation approach by casting rays out from the seed point and taking the convex hull of the intersection of those rays with the obstacle set. Since that convex hull will not be a perfectly obstacle-free region, they then intersect the computed region with additional half-space constraints to cut out any intersection points between the rays and obstacles which still lie within the computed region. The result is a potentially smaller safe region, but one computed without solving any numerical optimizations.

Bialkowki et al. do away completely with the notion of pre-computing obstacle free

regions, but end up with a related result in [48]. Their work is based on a standard sampling approach for motion planning, but each time a point is checked for its distance to the nearest obstacle, the authors construct a disc, centered at that point, whose radius is equal to the just-computed nearest obstacle distance. The interior of that disc is, by construction, entirely obstacle free (since its radius must just barely reach the nearest obstacle), and that disc is then added to a set of such discs. Future point queries can be cheaply checked against these discs rather than requiring an expensive check against every obstacle. In this way, the authors create a cluster of convex obstacle-free regions, although in this case the convex shapes are always circles and their placement is determined by random sampling rather than some seeding procedure.

Gait Generation and Dynamic Plans

The footstep planning problems presented so far have all assumed a strict gait pattern (alternating between the left and right foot), but this assumption breaks down for many walking robots. A bipedal robot may use one or both feet for support, but it may also use its hands to make contact with the world, and it may even briefly have no contact with the world at all when jumping or running. Robots with more than two legs likewise have many possible contact sequences to choose from, and restricting them to a single gait pattern limits their ability to traverse complex environments.

Instead, a more general approach is to expand the planning problem to allow each limb to independently be chosen as in contact with a particular surface or out of contact. In a mixed-integer optimization, this involves adding additional binary variables for each combination of a limb and a potential contact surface. This is the approach taken by Aceituno-Cabezas et al. in [49] and [50], in which the mixed-integer footstep planner from Chapter 2 is expanded with the addition of full gait planning for a quadruped robot. The authors further extend the footstep planner by explicitly reasoning about contact forces (using Dai's Contact Wrench Cone method from [51]), allowing the mixed-integer program to simultaneously choose the set of active contacts, their location, and their distribution of forces in order to create a

desired motion of the robot's center of mass.

In a similar vein, Valenzuela also adds gait planning and contact force optimization to the mixed-integer program in [52]. While Aceituno-Cabezas focused only on assigning footsteps to regions on the terrain [49, 50], Valenzuela also adds three-dimensional volumes of free space above the terrain in order to ensure that the robot's limbs are not in collision with the world while moving from one contact surface to another, allowing for more aggressive flight phases to be considered. Valenzuela also uses the McCormick envelope formulation [53] to allow approximate regulation of the robot's angular momentum while still maintaining the mixed-integer convex optimization structure.

Finally, Ponton et al. construct a related extension to the footstep planner of Chapter 2, adding support for hand contacts and introducing a new convex relaxation of the nonlinear angular momentum dynamics of the robot [54]. Their work results in a more complex, but also more expressive, mixed-integer convex optimization than the original footstep planner, one which is able to produce locomotion involving multiple different contacts between the robot and the world while still maintaining the basic structure of convex decomposition and mixed-integer planning. Furthermore, their more recent work in [55] explores extending the convex relaxation to approximate an optimization over the duration of each time step of the planned trajectory, a feature which has otherwise only been represented using a large number of very small time steps.

Trajectory Planning

The mixed-integer UAV trajectory planner of Chapter 3 has likewise seen some interesting extensions and modifications. While the trajectories generated in that work were chosen simply to optimize smoothness (minimizing the norm of some high-order derivative like jerk or snap), other objective functions could produce different useful results. For example, in [56], Press et al. introduce a trajectory optimization framework designed to improve the observability of certain self-calibration variables (such as the pose of a robot's IMU relative to its body). This optimization framework assumes that it is given a sequence of polytopic

regions of free space, information which is easily extracted from the solution to our mixed-integer optimization.

Miller et al. implement an approach which is quite similar to that of Chapter 3, but with an application to autonomous vehicles performing lane-change maneuvers [57]. A notable difference in their work is that simplified geometry of a multi-lane highway allows them to write down the convex safe regions by hand, rather than requiring some convex segmentation tool like IRIS, but the structure is otherwise comparable: by constructing convex obstacle-free regions, they are able to write down a mixed-integer convex optimization to choose a collision-free trajectory.

One notable drawback of our planner, however, is the relatively long computation time required to solve the mixed-integer optimization. Liu et al. [58] and Mohta et al. [59] reduce the computational burden by eliminating the mixed-integer component of the optimization entirely. Instead of simultaneously choosing a smooth trajectory and a sequence of convex safe regions, they instead use a more traditional search-based planner to first find a sequence of straight line segments from the robot's current position to its goal. They then use a method similar to IRIS to inflate a sequence of convex safe polyhedra around that sequence of line segments. This sequence of convex regions can then be used to optimize a smooth trajectory that lies within the given regions. Since the ordered sequence of convex safe regions is already known, no integer variables are required to assign each smooth trajectory segment to a safe region; this results in a much simpler online optimization problem, but sacrifices some notion of optimality, since the original sequence of line segments may not actually correspond to the optimal path for the smooth trajectory.

Alonso-Mora et al. also choose to eliminate the mixed-integer optimization in favor of a more traditional sampling-based motion planner, this time in order to support navigation of multiple robots flying in formation among moving obstacles [60]. While it would be possible to extend the planner from Chapter 3 to handle multiple vehicles, doing so would require adding binary variables for the safe region assignment of each vehicle, making an already hard mixed-integer optimization potentially much harder (in the worst case, exponentially

harder). Instead, Alonso-Mora chooses to simplify the optimization by constructing only a single convex region, using a modified version of IRIS, designed to grow in the direction of some local goal or waypoint. The entire formation of robots is then required to move within this single convex safe region, with nonlinear constraints used to enforce collision avoidance. Since this approach can only handle moving in a single direction, a higher level sampling-based motion planner is used to choose the appropriate waypoints for the optimization. Removing the mixed-integer component, as in [58], makes the optimization much easier to solve, but likewise allows the possibility of sub-optimal trajectories.

4.3 Conclusion

The approach of segmenting an environment into convex safe pieces in order to create a locomotion plan using those pieces has been fruitful, but it has also come with a substantial computational cost which has made it difficult to use when online re-planning is necessary or when perfect maps of the environment are not available. While other researchers have found interesting ways to build upon these ideas, offering improved performance or applicability to new domains, the fundamental question of how to control robots that make contact with the world is still open. In the next chapter, we will introduce a new method, unlike the mixed-integer optimizations presented here, which offers some hope for a more general solution to contact-aware control.

Part II

Learning Contact-Aware Controllers

The contact planning approach described in Part I simplified the problem of controlling a legged robot by separating it into tractable components, but its results left something to be desired. The work presented previously suffered from an inability to make contact plans which were aware of the robot's dynamics, and it offered no answers about how to rapidly make new contact plans in the fact of unexpected disturbances.

The problem of missing dynamics is, to some extent straightforward to solve: we can simply add a dynamics model to the footstep plan, adding additional variables to track the momentum of the robot and the contact forces that affect it. This is precisely the approach taken by Valenzuela in [52]. Given a simplified quadruped robot model, a mixed-integer optimization similar to the footstep planning problem in Chapter 2 is constructed, with additional variables and constraints to enforce a linearized model of the robot's centroidal dynamics. This enables much more aggressive plans to be generated: rather than alternating between two feet, the trajectories generated by [52] can involve multiple contacts or even airborne phases as the robot leaps from one contact to another. But this additional complexity comes at a cost: the optimizations take longer to solve, generally requiring more than a minute to solve a single problem and putting rapid online re-planning even farther out of reach.

Instead, it is interesting to focus on the second problem, the issue of how to make new contact plans online when the robot experiences some external disturbance. In particular, how could we build a controller that could decide when and how to make or break contact instantaneously? If such a controller were possible, then it might not be necessary to pre-plan every contact sequence, and the controller could naturally respond to disturbances by making or breaking contact. When the robot is pushed, such a controller could cause it to stumble or reach out for support rather than simply falling.

In Chapter 5 I present a new approach to contact-aware control. By learning a value function from mixed-integer optimization samples, we can design a controller that is able to make intelligent contact decisions online to stabilize the robot.

Chapter 5

Learning from Value Function Intervals for Contact-Aware Robot Controllers

This chapter has been submitted for independent publication as [3].

5.1 Introduction

While there are a variety of successful approaches for planning multi-contact behaviors (e.g. [52, 61, 62, 63]), it has proven to be difficult to apply these techniques quickly enough to be used in response to disturbances. Furthermore, most multi-contact trajectory optimizations are solved via non-convex optimizations, typically through sequential quadratic programming (e.g. [61, 62]) or differential dynamic programming (e.g. [63, 64, 65, 66]). These techniques can generally find locally optimal solutions, but make no guarantees of global optimality. While locally optimal solutions are often sufficient for planning purposes, they make training a policy from examples (as in [66] and [64]) more difficult, as the locally optimal samples may not describe a coherent global policy.

Mixed-integer optimization offers some hope for planning globally optimal multi-contact

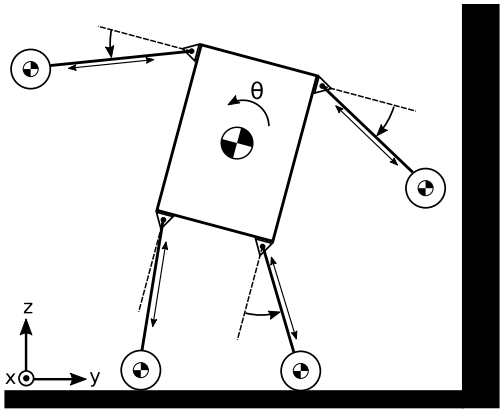


Figure 5-1: The simplified humanoid model, consisting of a central body and four limbs, each with mass and inertia. Each limb is connected to the body by a revolute and translational joint, and all motion is restricted to the frontal plane. Contact is possible between each limb and the rigid floor and wall.

behaviors: By explicitly representing the discrete changes in dynamics with discrete (i.e. integer) variables, we can create optimization problems which are solvable to global optimality using branch-and-bound [67]. Global optimality is possible even in the presence of nonlinear constraints [68, 69], but for this work we restrict ourselves to piecewise affine models, inspired by the long history of successful linearized dynamical models for humanoid robots (e.g. [70]). Unfortunately, global optimality comes at a cost, with typical trajectory optimizations taking seconds or minutes to solve [52]. Furthermore, there is no guarantee that these expensive optimizations will result in a consistent global policy, as the optimal policy itself may not be unique.

On the other hand, we do not necessarily need to completely solve a mixed-integer optimization to get some useful information from it. Mixed-integer convex problems are generally solved by branch-and-bound [67], a process which iteratively finds better candidate solutions and tighter bounds on the best possible solution. If we ensure that a candidate solution always exists, then we can terminate the branch-and-bound process at any time, retrieving the best solution and tightest bound found so far. Although we could attempt to train from these sub-optimal solutions, we would again be learning to imitate a sub-optimal controller. The *bounds themselves*, however, are extremely useful: In an MPC problem, bounds on the

optimal objective value are also bounds on the optimal cost-to-go[1] from a given state.[2] Having a model of the cost-to-go in turn enables fast online control by simply greedily descending that cost.

5.1.1 LVIS: Learning from Value Interval Sampling

In this work, we introduce LVIS, a new approach for the creation of contact-aware controllers. We model our robot's contact dynamics with complementarity constraints (Section 5.3.1). Offline, we set up a large number of trajectory optimizations in the form of mixed-integer quadratic programs (MIQPs) from a variety of initial states. We partially solve those optimizations, terminating early and extracting concrete intervals containing the optimal cost at the given robot state (Section 5.3.2). From these intervals, we train a small neural net to approximate the cost-to-go using a loss function which penalizes predicted values outside the known intervals (Section 5.3.3). Online, we run a simple one-step MPC controller to greedily descend the approximate cost-to-go as quickly as possible subject to the robot's dynamics (Section 5.3.4).

5.2 Related Work

This work is similar to that of Zhong et al. in [64], in which offline optimizations were also used to train an approximate cost-to-go used as the terminal cost of a shorter-horizon MPC problem. Zhong's work differs from ours in its use of iterative LQR (iLQR) to generate the cost-to-go samples. As iLQR is a local nonlinear optimization, it can only provide an estimate of the upper bound of the the cost-to-go (since a lower cost might exist in a space that was not explored by the local optimization). In our case, by constructing a mixed-integer optimization and solving it with branch-and-bound, we recover global upper and

[1] The cost-to-go, which we will also refer to as the value function $J(x)$, is the cost which will be accumulated by the optimal controller starting from state x [5].

[2] This assumes that the MPC horizon is long enough to reach a set of states with known cost-to-go. We will violate that assumption later, but attempt to demonstrate empirically that the objective bounds are still useful.

lower bounds on cost-to-go, using the interval spanned by those bounds during training. In Section 5.4.2, we specifically compare LVIS with an approach of learning only from upper bounds on the cost-to-go, and we demonstrate that the intervals produce a more effective MPC controller.

A major obstacle to solving MPC problems for system with contacts is the potentially vast number of possible mode sequences.[3] If an optimal mode sequence for a given state could be computed, then we could perform a cheap continuous optimization to choose the precise optimal input given that mode sequence. This is the approach taken by Hogan in [71], in which a neural net is trained to predict mode sequences from robot states. Marcucci takes a similar approach in [72] by creating a library of provably feasible stabilizing mode sequences and looking up a mode sequence for the robot's current state at run-time.

Mixed-integer optimizations are also no the only way to plan complex multi-contact behaviors. The contact-implicit trajectory optimizations of Posa [62] and Dai [61] can generate complex motions exploiting a variety of contacts in the environment. These approaches have so far been too slow to run online in an MPC fashion, but it might be possible to learn an approximate controller or value function, as with Zhong's work in [64], from the trajectory optimization. As with the iLQG approach above, the fact that these general nonlinear optimizations claim only local optimality may make learning from their results less effective.

Alternatively, the efficient sequential linear-quadratic methods from [65] do allow for locally optimal real-time MPC for systems with contact, avoiding the need for offline learning. These optimizations are still subject to local minima, but the ability to run them at real-time means that they do not need to be used to train a global policy.

Looking more broadly, reinforcement learning offers an alternative approach which does not require any explicit offline planning but instead simply the ability to roll out actions in simulation or hardware. For a general survey of reinforcement learning in robotics, see [73], and for a more up-to-date example for reinforcement learning with deep neural nets, see [74]. The potential advantages of reinforcement learning are tremendous, as it does not

[3] Our humanoid system has $1,679,616$ modes (Section 5.2.1), so a trajectory optimization with a horizon of 10 steps has $1679616^{10} \approx 1.8 \times 10^{62}$ possible mode sequences, most of which are infeasible.

require the expensive offline mixed-integer optimizations that our proposed method uses. On the other hand, directly measuring the cost-to-go from a given state, rather than trying to estimate a reward based on expected future actions, allows us to avoid careful initialization or reward shaping. In fact, we initialize our entire learning system randomly and use only a quadratic cost, centered on the robot's desired final configuration.

5.2.1 Robot Models

We demonstrate the LVIS controller on two models: a cart-pole system balancing between two walls and a simplified humanoid model.

Cart-Pole with Walls

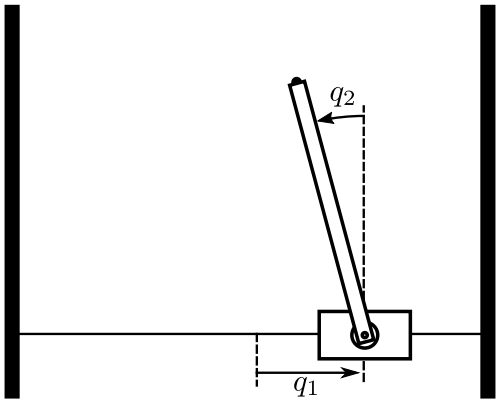


Figure 5-2: The cart pole system with two walls. The cart and pole both have mass and gravity acts downward. The pole pivots freely about its joint, and the only input to the system is the horizontal force applied to the cart. Contact occurs only between the tip of the pole and the two walls.

The cart-pole (Figure 5-2) is a canonical underactuated robotic model (and a common control task in Reinforcement Learning) in which a pole is stabilized to the upright position by accelerating the cart to which it is mounted. The cart-pole has 2 degrees of freedom (horizontal position of the cart and angular position of the pole) and 1 input (horizontal force applied to the cart). We modify the system by adding two walls on either side (contact is considered only between the tip of the pole and the walls). The cart-pole system has 4

continuous states, 1 input, and 7 discrete modes (out of contact, sticking, sliding up, and sliding down for each wall).

Simplified Humanoid Model

To model the application of our idea to a real walking robot, we introduce a simplified humanoid model shown in Figure 5-1. This model has 11 degrees of freedom (3 DoF for the planar translation and rotation of the central body, and 2 DoF for the rotation and extension of each limb in the plane). We model contact between each limb and a fixed floor and wall, including both slipping and sticking, and we add hard position limits for each of the 8 joints connecting the limbs to the body. The humanoid has 22 continuous states and 11 inputs. Since each of the 4 limbs can either be free, sticking, or sliding in one of two directions, and each of the 8 joints can either be free, at its upper limit, or at its lower limit, the system has a total of $4^4 \times 3^8 = 1,679,616$ discrete modes. Note that we do not explicitly model each of these 1.7 million modes, but instead model the contacts implicitly using complementarity conditions, as described in Section 5.3.1.

5.3 Technical Approach

The LVIS approach consists of the following components:

1. Modeling the robot's dynamics as a piecewise affine system, allowing us to write down optimal control problems with only linear and integer constraints.

2. Offline data collection by partially solving the mixed-integer optimizations and extracting bounds on the optimal cost-to-go.

3. Offline training of a neural net using a double-sided hinge loss for each pair of upper and lower cost-to-go bounds.

4. Online control of the robot using mixed-integer greedy descent on the learned cost-to-go, in the form of a short-horizon MPC controller.

5.3.1 Modeling

Following the formulation of Stewart and Trinkle in [75], we model the dynamics of our robot in a contact-implicit manner with complementarity conditions. In discrete time, these dynamics take the form:

$$\mathbf{M}\left(\mathbf{v}^{l+1} - \mathbf{v}^{l}\right) = h\mathbf{f}_{ext}^{l} + h\mathbf{C} + h\mathbf{B}\mathbf{u}^{l} \tag{5.1a}$$

$$\mathbf{q}^{l+1} - \mathbf{q}^{l} = h\mathbf{v}^{l+1} \tag{5.1b}$$

where h is the length of the time step, $\mathbf{q}^l$ and $\mathbf{v}^l$ represent the system's generalized configuration and velocity at time step l, $\mathbf{M}$ is the mass matrix, $h\mathbf{f}_{ext}$ is the external impulse due to contact and friction, $h\mathbf{C}$ is the impulse caused by Coriolis, and gravitational forces, and $h\mathbf{B}\mathbf{u}^l$ is the impulse caused by the generalized inputs $\mathbf{u}$. We further break down the contact impulse into tangential and frictional components:

$$\mathbf{f}_{ext} = \mathbf{n}c_n + \mathbf{D}\beta \tag{5.2}$$

where $\mathbf{n}$ is the contact normal vector, c_n is the contact force along that normal, and $\mathbf{D}$ and β are the basis vectors and weights, respectively, of the frictional force. Additional contacts can be added by introducing additional f_{ext} terms.

Complementarity conditions ensure that there is no force acting at a distance and no sliding frictional force without an accompanying velocity. For a single contact, we have:

$$\mathbf{n}^{\top}\mathbf{q}^{l+1} - \alpha_0 \perp c_n \tag{5.3a}$$

$$\lambda\mathbf{e} + \mathbf{D}^{\top}\mathbf{v}^{l+1} \perp \beta \tag{5.3b}$$

$$\mu c_n - \mathbf{e}^{\top}\beta \perp \lambda \tag{5.3c}$$

where α_0 is a scalar indicating the displacement at which collision occurs, μ is the coefficient of Coulomb friction, and $\mathbf{e}$ is an all-ones vector. We use the notation $\mathbf{x} \perp \mathbf{y}$ to mean $\mathbf{x} \geq 0$ and $\mathbf{y} \geq 0$ and $\mathbf{x}^{\top}\mathbf{y} = 0$. For a complete explanation of these constraints, see [75].

It is important to note that $\mathbf{M}$, $\mathbf{k}$, $\mathbf{n}$, and $\mathbf{D}$ all depend on the robot's current state (in general in a nonlinear way). When we write down a trajectory optimization as a mixed-integer quadratic problem, we cannot represent these nonlinear dependencies, so we linearize the dynamics around the current state, fixing $\mathbf{M}$ etc. This introduces a real drawback of our approach, as our results are only valid for this particular linearization. We hope to explore using the full nonlinear dynamics in future work (see Section 6.1).

5.3.2 Data Collection via Optimal Control

To generate a single sample, we first choose an initial configuration $\mathbf{q}_0$ and velocity $\mathbf{v}_0$, collectively referred to as the state $\mathbf{x}_0 = \begin{bmatrix} \mathbf{q}_0 \\ \mathbf{v}_0 \end{bmatrix}$. We then write down a trajectory optimization optimal control problem:

$$\begin{aligned} J^* = \underset{\substack{\mathbf{x}^1 \dots \mathbf{x}^N, \mathbf{u}^1 \dots \mathbf{u}^N, \\ \lambda^1 \dots \lambda^N, \beta^1 \dots \beta^N, c_n^1 \dots c_n^N}}{\text{minimize}} & \sum_{l=1}^{N} \left[\mathbf{x}^{l\top} \mathbf{Q} \mathbf{x}^l + \mathbf{u}^{l\top} \mathbf{R} \mathbf{u}^l \right] + \mathbf{x}^{N\top} \mathbf{S} \mathbf{x}^N \\ \text{subject to} \quad & \text{dynamics constraints (5.1a, 5.1b)} \\ & \text{complementarity conditions (5.3a, 5.3b, 5.3c)}\,. \end{aligned} \tag{5.4}$$

While specialized solvers such as PATH [76] can handle complementarity conditions of the form in (5.3a, 5.3b, 5.3c) directly, they do not generally support minimizing a quadratic objective. Instead, for each scalar complementarity condition $x_i \perp y_i$ we introduce a new binary variable z_i and constrain:

$$x_i \geq 0 \tag{5.5a}$$

$$y_i \geq 0 \tag{5.5b}$$

$$z_i = 1 \implies x_i = 0 \tag{5.5c}$$

$$z_i = 0 \implies y_i = 0\,. \tag{5.5d}$$

We formulate the implication constraints in (5.5c, 5.5d) as linear constraints using a big-M formulation [25]. The introduction of the binary variables z_i converts our optimization into a general mixed-integer quadratic program (i.e. a program with a quadratic objective, linear constraints, and some variables constrained to take integer values in $\{0, 1\}$). We solve the resulting MIQPs with the Gurobi optimization software [21].

Big-M Formulation

Given a binary decision variable z and an affine function f applied to a vector of decision variables $\mathbf{x}$, the big-M formulation transforms the following logical constraint:

$$z = 1 \implies f(\mathbf{x}) \leq 0 \tag{5.6}$$

into a linear inequality constraint:

$$f(\mathbf{x}) \leq M(1 - z). \tag{5.7}$$

When $z = 1$, (5.7) becomes:

$$f(\mathbf{x}) \leq 0, \tag{5.8}$$

satisfying the implication from (5.7). When $z = 0$, on the other hand, (5.7) becomes:

$$f(\mathbf{x}) \leq M, \tag{5.9}$$

which appears as if it might impose an unwanted constraint on the optimization. If, however, M is chosen to be big enough that no otherwise feasible values of $\mathbf{x}$ could produce a value of $f(\mathbf{x}) > M$, then (5.9) will essentially have no effect on the optimization, and we will have fully satisfied the original logical constraint from (5.6). Unfortunately, M cannot be chosen to be arbitrarily large, as doing so will make the resulting optimization more difficult to solve, both by creating poor numerical scaling and by making tight convex relaxations of the

mixed-integer program more difficult to find (see [25] for more on why this is the case).

In order to choose a reasonable big-M value without requiring human intervention, we developed the ConditionalJuMP.jl software package [77] which uses validated interval arithmetic (via the IntervalArithmetic.jl software package [78]) to compute a conservative interval containing all possible values of $f(\mathbf{x})$ given the existing bounds on the decision variables $\mathbf{x}$. The interval arithmetic upper bound on $f(\mathbf{x})$ is always a valid choice for M, and we found that such a choice produced reasonably efficient optimizations.

It is worth noting, however, that the upper bound on $f(\mathbf{x})$ produced by interval arithmetic may be arbitrarily higher than the true upper bound on $f(\mathbf{x})$ achievable in any feasible solution to the optimization problem. This can be due to (1) the interval arithemtic itself producing overly conservative (i.e. too wide) intervals and (2) additional constraints in the optimization problem limiting the possible values of $\mathbf{x}$. Such an over-estimate does not affect the correctness of the big-M formulation, as it simply results in choosing a big-M which is larger than would actually be required, but it can result in an optimization problem with worse numerical scaling or poor convex relaxations (again, as in [25]).

A tighter bound on $f(\mathbf{x})$ could be achieved by solving an optimization problem to maximize $f(\mathbf{x})$ subject to all of the constraints present in the original optimization problem. If the other variables in the optimization are all continuous and the constraints are all linear, then maximizing $f(\mathbf{x})$ is simply a linear program. If, however, there are other integer or binary variables (such as those created by other logical constraints with their own big-M formulations), then even maximizing $f(\mathbf{x})$ may require solving a mixed-integer linear program, which could be quite computationally expensive. There is thus a trade-off between the amount of work done to find an accurate value of $f(\mathbf{x})$ and the quality of the resulting big-M formulation.

In this work, we implement only the interval arithmetic approach, as it is computationally efficient and produces sufficiently tight values of M for our work. For other models, however, it may actually be worthwhile to pursue much tighter values of M. As an example, in [79], Tjeng et al. implement all of the above approaches for computing M in the context of the

mixed-integer optimization of the output of a neural net.

Generating Feasible Solutions as Warm-Starts

One notable feature of the mixed-integer trajectory optimization problem in (5.4) is that its constraints only enforce the dynamics and complementarity conditions of the robotic system. These constraints represent simply the underlying physics of the robot, so any desired behavior of the system, beyond what mechanics requires, must be expressed in the cost matrices $\mathbf{Q}$, $\mathbf{R}$, and $\mathbf{S}$. This is a significant restriction in the expressiveness of our optimization, but it comes with an advantage: we can generate *feasible* (but not generally optimal) solutions to the optimization (5.4) simply by simulating the system under any controller subject to the same physical constraints. Simulation is generally much easier than trajectory optimization, as only one time step must be solved for at a time, and any simulated trajectory will, by construction, obey the simulated physics. We ensure, by constructing our own simulation, that the simulated physics are exactly the constraints in (5.1a, 5.1b) and (5.3a, 5.3c, 5.3c) and we simulate the system for exactly N steps in order to collect a simulated trajectory whose length matches that of our trajectory optimization.

These simulated trajectories are then used as warm-starts in the mixed-integer optimization: From the simulated trajectory, we can look up the state of each complementarity condition at each time step and extract the values of the binary decision variables z_i. We can then provide those values as initial assignments to the z_i variables when passing the full trajectory optimization to the MIQP solver.

Warm-Start Quality

The quality of the feasible solutions we generate depends entirely on the choice of controller used during the simulation. Fortunately, since simulation is computationally cheap compared to a full MIQP solution, we can afford to warm-start with multiple controllers and pick whichever simulation result happens to have lower cost. In our case, we warm-start every optimization by simulating with an LQR controller designed around the robot's nominal

posture and with the learned controller described in this chapter. The LQR controller performs well near the robot's nominal state, so it can provide near-optimal warm-starts in these easy cases. The learned controller initially performs essentially randomly (as it starts out completely untrained), but as we train its terminal cost-to-go its performance improves and it tends to provide increasingly good warm-starts.

Early Termination

Solving the MIQP optimization (5.4) can be extremely expensive for a robot with as many states and modes as our planar humanoid: a trajectory with just $N = 10$ steps can take thousands of seconds to solve to near optimality[4]. The key insight of LVIS is that we do not need to solve all the way to optimality: We can easily generate feasible solutions by simulation, so the process of solving the optimization problem in (5.4) is a matter of running the branch-and-bound algorithm to iteratively find better solutions and tighter bounds on the optimal cost. At any point, we can simply terminate the optimization and extract the best solution and tightest bound found so far. We label the cost of the best solution found so far (which is an upper bound on J^* as J_{ub}, and we label the tightest lower bound on the optimal cost as J_{lb}.

The ability to terminate the optimization early raises a new question of when we choose to terminate. Typical options include: optimality gap (the difference between the best feasible solution and tightest lower bound, in absolute or relative units), nodes explored (a measure of how many convex relaxations have been constructed by the MIQP solver), or simply total elapsed time. We choose elapsed time for the simplified humanoid, and somewhat arbitrarily cease optimization after a fixed time horizon. For the results shown in this chapter, we allowed 3 seconds of work on each MIQP to balance the sub-optimality of each sample with the rate at which samples can be generated (limits of 5 or 10 seconds provided similar performance).

[4]For our humanoid robot model, solving to within 1% of the optimal cost required an average of 1160 seconds per trajectory optimization, with some cases taking several hours each. At a nominal control rate of 100 Hz, solving to full optimality would thus require the data collection to run at less than 0.0008% real-time.

5.3.3 Training the Neural Net

The neural net which will approximate our cost-to-go consists of a simple fully-connected feed-forward network with exponential linear unit (ELU) activations [80]. For the humanoid in Figure 5-1 we use two hidden layers with 48 units each, while for the cart-pole in Figure 5-2 we use two hidden layers with 24 units each. The neural net has a number of input dimensions equal to the number of states in the robot and an output dimension of one. We will label the predicted cost-to-go at a state $\mathbf{x}$ as $\hat{J}(\mathbf{x};\theta)$, where θ are the trainable parameters of the net.

Loss Function

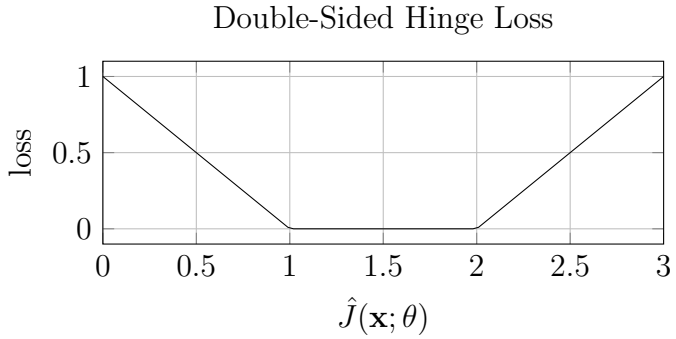


Figure 5-3: The double-sided hinge loss used to train the neural net, shown with $J_{\text{lb}} = 1$ and $J_{\text{ub}} = 2$.

We train the neural net from a set of training samples, where each sample consists of a tuple of (robot state $\mathbf{x}$, cost-to-go lower bound J_{lb}, and cost-to-go upper bound J_{ub}). We penalize the net for predicting values outside of the range $[J_{\text{lb}}, J_{\text{ub}}]$ using a double-sided hinge loss shown in Figure 5-3 and defined as:

$$h(\mathbf{x}, J_{\text{lb}}, J_{\text{ub}}; \theta) = \begin{cases} J_{\text{lb}} - \hat{J}(\mathbf{x};\theta) & \text{if } \hat{J}(\mathbf{x};\theta) < J_{\text{lb}} \\ 0 & \text{if } J_{\text{lb}} \leq \hat{J}(\mathbf{x};\theta) \leq J_{\text{ub}} \\ \hat{J}(\mathbf{x};\theta) - J_{\text{ub}} & \text{if } \hat{J}(\mathbf{x};\theta) > J_{\text{ub}}\,. \end{cases} \tag{5.10}$$

The total loss is simply the sum of the hinge losses over every sample $(\mathbf{x}, J_{\text{lb}}, J_{\text{ub}})$.

The use of the hinge loss provides a unique advantage: as training proceeds, we may revisit a particular state $\mathbf{x}$, and due to a better warm-start we may come up with tighter bounds J_{lb} and J_{ub} than we previously discovered. The hinge loss ensures that, so long as the net predicts a value $\hat{J}$ which falls within the new, tighter bounds, the old, looser bounds do not influence the total loss. If, instead, we were attempting to learn J by exactly matching the upper or lower bounds or their precise midpoint, then our older samples would tend to pull the net's output away from the newer samples. Figure 5-4 shows an example of a learned value function and the interval samples which were used to train it.

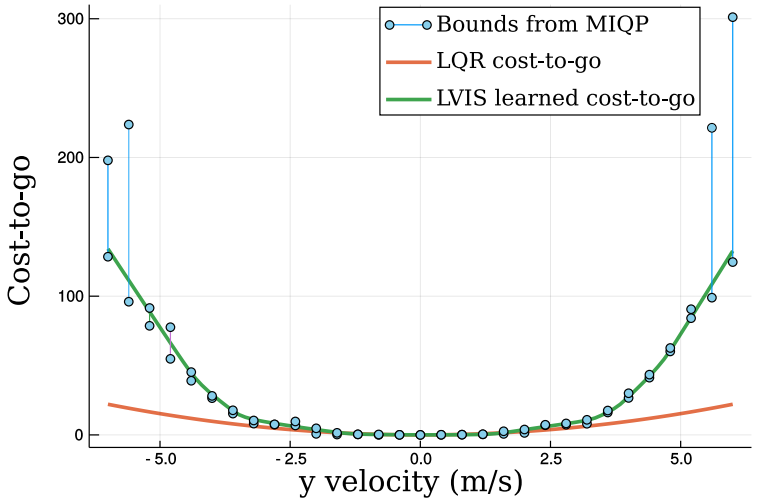


Figure 5-4: An example of an approximate value function learned from interval samples. Each connected pair of points represents the result of a single trajectory optimization sample: the upper point is the upper cost bound J_{ub} and the lower point is the lower bound J_{lb}. The upper and lower bound values are identical for the states in which the mixed-integer program was solved to optimality. The neural net (green) was trained using the hinge loss from Figure 5-3 and lies between the upper and lower bound values at each sample. Note that the size of the intervals increases as magnitude the robot's initial x velocity increases: for small initial velocities, the warm-start solution provided by the LQR controller (see Section 5.3.2) is excellent, and the solver is often able to prove optimality of that solution quickly. But for larger initial velocities, the warm-start solution is often highly suboptimal, and the solver often needs much longer to both find the optimal solution and prove its optimality. The LQR cost-to-go is also shown in red for reference. The LQR and learned cost-to-go functions match closely for small initial velocities, indicating that the LQR policy is nearly optimal for small disturbances.

Optimization

The parameters θ are trained using a stock Adam optimizer [81] with a batch size of 1. No dropout or regularization was performed during the training process, and all parameters of the Adam optimizer were set to the defaults suggested in [81].

5.3.4 Online Control Using the Learned Cost

The result of the training process described in Section 5.3.3 is a neural net whose forward pass approximates the cost-to-go of the original MPC problem. To turn this neural net into a control policy, we need a way to greedily descend that cost-to-go. Our procedure for this is simple: we construct a new MPC optimization with only one time step ($N = 1$) using the same step size h as in the offline optimizations, and we set as its terminal cost the learned neural net cost. The neural net's output is nonlinear and non-convex, so rather than directly trying to minimize its output, we minimize a local linear approximation of the neural net's output expanded about the robot's current state. This corresponds to a gradient descent on the cost-to-go, subject to the robot's physical dynamics constraints.

Even the one-step MPC optimization, however, still involves complementarity constraints and is thus a mixed-integer problem. While the only control decisions are the continuous forces applied to each limb, we must still resolve the boolean complementarity constraints to determine the physically consistent contact and frictional forces. The restriction to a single time step dramatically reduces the number of integer variables which must be solved, allowing near-real-time controller performance, but solving even these smaller MIQPs at control rates is still a challenge. For this work, we implemented an aggressively optimized online mixed-integer controller, using the RigidBodyDynamics.jl software package [82] to model the robot's dynamics, the Parametron.jl software package [83] to model the optimization problem, and the Gurobi solver [21] to solve the resulting problems.

5.3.5 Choosing Initial States with DAgger

Rather than sampling randomly across the robot's entire state space, we adopt the DAGGER approach from [84]. Put simply, DAGGER relies on simulating the system using the (initially poorly-trained) policy as its controller instead of the expert, iteratively collecting new training samples from the regions of state-space visited by the learned policy. Our training alternates between (a) letting the learned controller drive the robot for 25-100 time steps while running the mixed-integer optimization to produce new training samples and (b) using those new samples to further train the approximate cost-to-go.

One aspect we have not yet explored is whether an approach like DAGGER can result in even better warm-starts due to its frequent sampling of similar states along a given trajectory. We could potentially take advantage of this by attempting to warm-start the controller using the entire trajectory solved at the previous time step, but we have not yet done so.

5.3.6 Policy Net

Rather than trying to learn the value function, we could simply attempt to train a neural net to mimic the mapping from $\mathbf{x}$ to $\mathbf{u}$ using the same trajectory optimization samples. We label this approach the Policy Net, though we could also refer to it as behavioral cloning as in [85]. As discussed in Section 5.3.2, however, the fact that the trajectory optimizations are not generally solved to optimality means that the $\mathbf{u}$ samples are not generally optimal. Training a neural net to approximate these suboptimal samples is unlikely to result in a good approximation of the optimal policy, but we attempt to do so in order to evaluate that claim.

5.4 Results

The learned value function controller was tested on the bounded cart-pole system (Figure 5-2) and the simplified planar humanoid model (Figure 5-1). In both cases, value function samples were collected offline by solving mixed-integer trajectory optimizations as described

in Section 5.3.2 while training a neural net to mimic the optimal cost-to-go. Online, an MPC controller with a horizon of 1 timestep was used to control the robot, greedily descending the learned cost-to-go.

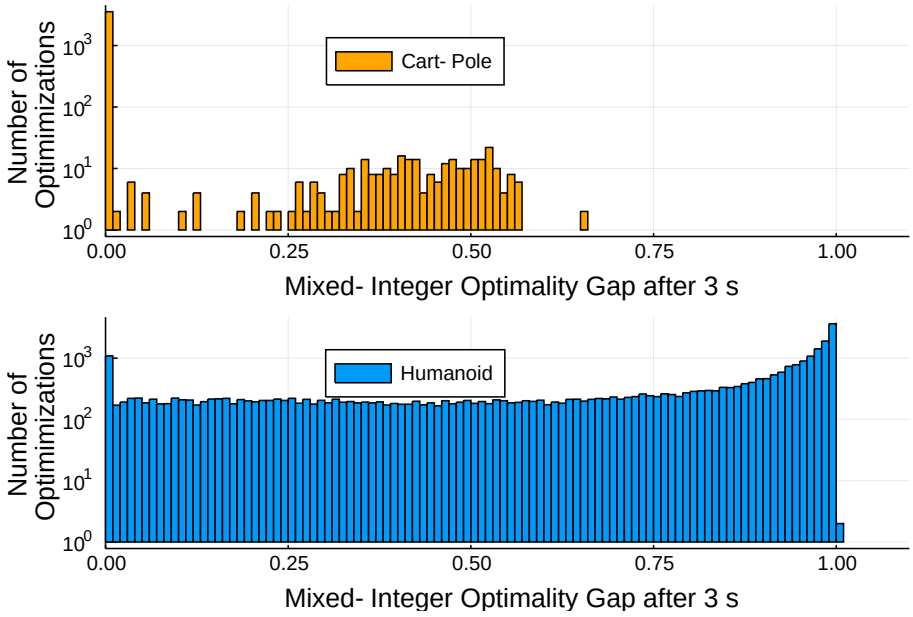


Figure 5-5: Comparing the optimality gap of the mixed-integer trajectory optimizations for the cart-pole and humanoid robot models. Optimizations were terminated after 3 seconds using the Gurobi solver. Optimality gap is defined as $\frac{J_{ub}-J_{lb}}{J_{ub}}$, where a value of 0 indicates convergence to the global optimum. The humanoid robot had an additional 932 samples with objective gap values ranging from 1.1 to 10^6 (not shown on this plot), but those accounted for less than 3% of samples. The cluster of results with an optimality of gap near 0 for the cart-pole indicates that most of the optimizations were solved to (near) global optimality, while the humanoid results show that full convergence to optimality was much less common for that more complex system.

5.4.1 Cart-Pole With Walls

The approximate cost-to-go for the cart pole was trained from 3862 mixed-integer trajectory optimization samples. Each trajectory optimization had a horizon of 20 steps and a time step of 25 ms, for a total lookahead time of 0.5 s. Trajectory optimizations were terminated after 3 seconds, which was sufficient for 92.0% of samples to converge to within 1% of the globally optimal cost, as shown in Figure 5-5.

Baseline LQR Policy

As a baseline policy, an LQR controller was constructed using a quadratic state cost:

$$Q = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{5.11}$$

and a quadratic input cost:

$$R = \begin{bmatrix} 0.1 \end{bmatrix} \tag{5.12}$$

centered around the upright fixed point of the cart and pole:

$$\mathbf{x} \equiv \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.13}$$

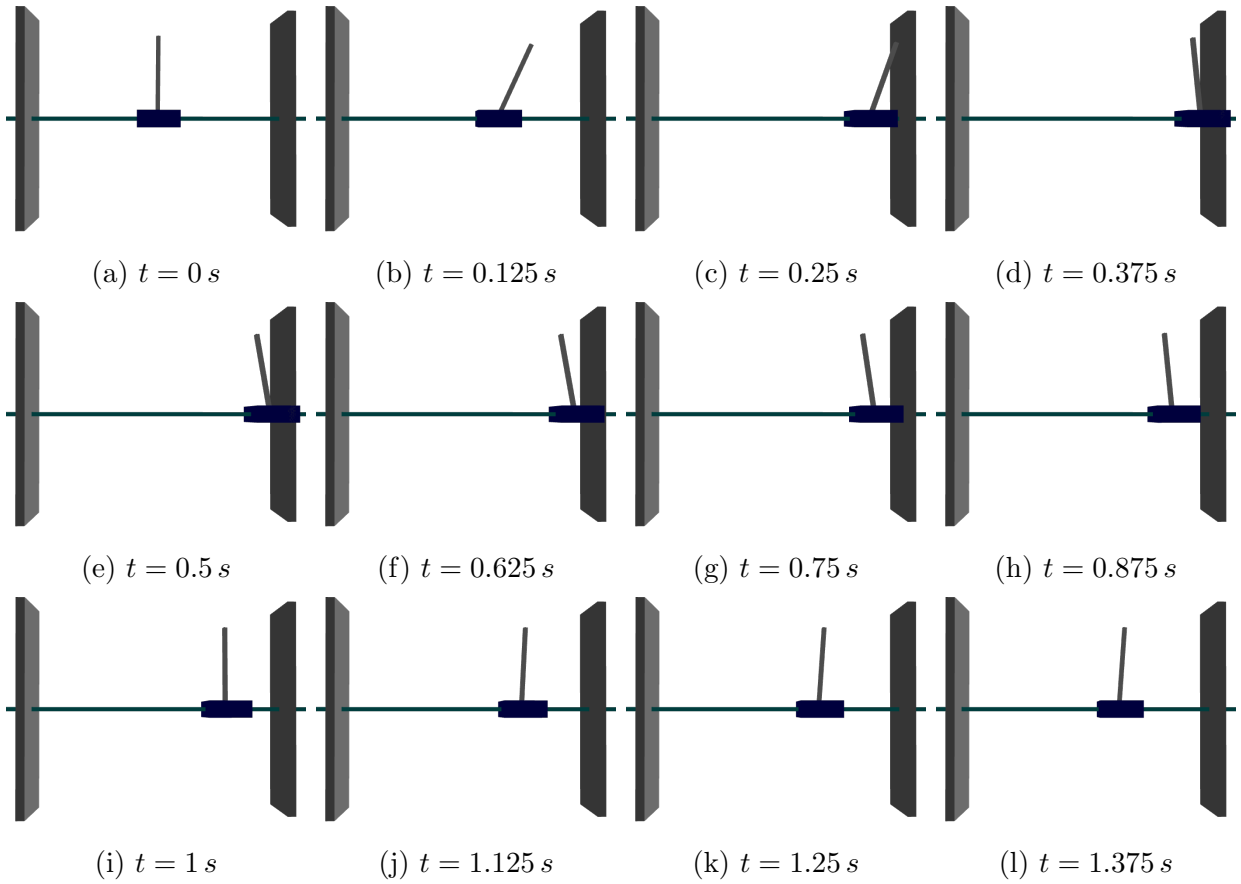
A discrete-time LQR controller was constructed from the given cost matrices, and the resulting cost-to-go was used as the terminal cost during the offline mixed-integer trajectory optimization.

Data Augmentation

Since the cart-pole system and cost matrices are perfectly symmetric, any trajectory optimization solution from state $\mathbf{x}$ with input $\mathbf{u}$ implied the existence of a mirrored solution from state $-\mathbf{x}$ with input $-\mathbf{u}$ and with the same bounds on the cost-to-go. Each optimization could thus contribute two samples to the training set: one at $\mathbf{x}$ and another at $-\mathbf{x}$.

Training

Training the cart-pole cost-to-go required approximately two hours on a single CPU, the majority of which was spent solving mixed-integer trajectory optimizations. A total of 500 rounds of training with the ADAM optimizer were performed. Convergence was estimated from 20% of the training samples held as a validation set.



(a) $t = 0\,s$ (b) $t = 0.125\,s$ (c) $t = 0.25\,s$ (d) $t = 0.375\,s$

(e) $t = 0.5\,s$ (f) $t = 0.625\,s$ (g) $t = 0.75\,s$ (h) $t = 0.875\,s$

(i) $t = 1\,s$ (j) $t = 1.125\,s$ (k) $t = 1.25\,s$ (l) $t = 1.375\,s$

Figure 5-6: The learned controller for the cart-pole recovering from an initial rotational velocity of 8 radians per second by using the contact between the pole and the right wall.

Evaluation

Three potential controllers were evaluated for the cart-pole in order to measure the effectiveness of the learned value function approach:

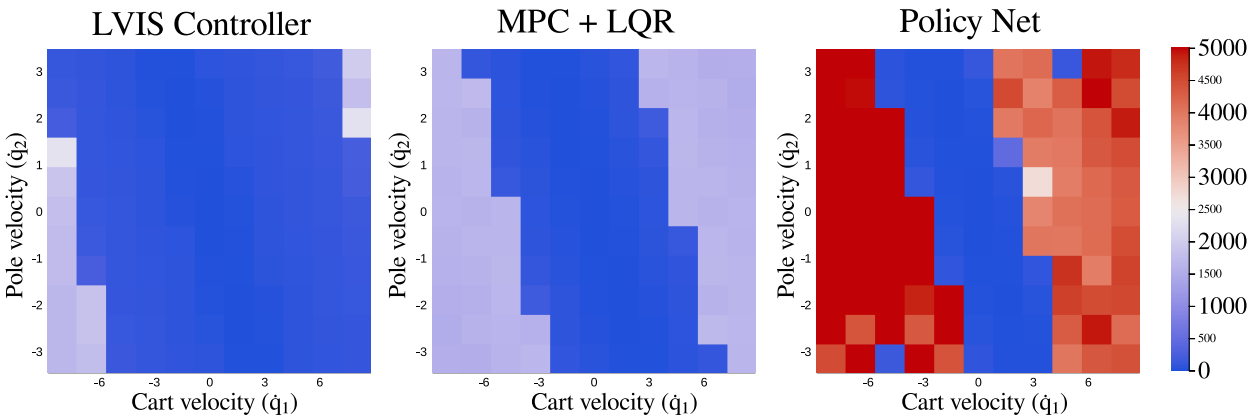


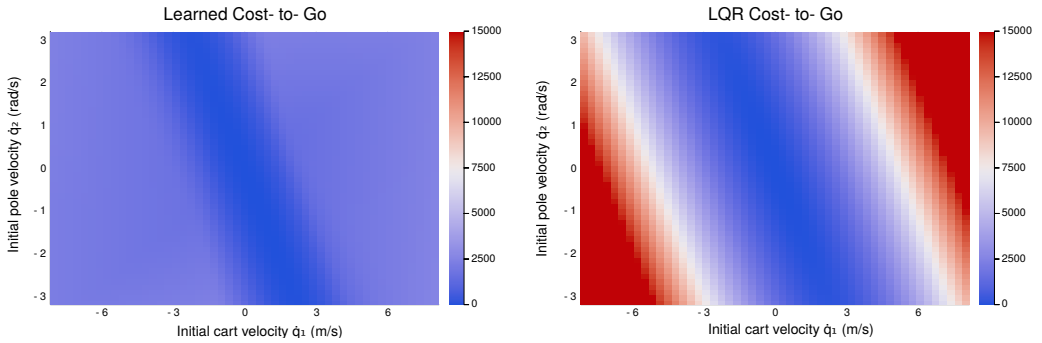
Figure 5-7: Accumulated cost of the cart-pole controllers. Each cell indicates the total accumulated cost over a 4-second simulation from the given initial cart and pole velocities, using the same cost matrices as the LQR controller. The regions of very low (dark blue) accumulated cost indicate simulations for which the pole was successfully balanced. The LVIS approach resulted in the lowest accumulated cost and successful stabilization from the widest variety of initial conditions.

1. LVIS: One-step mixed-integer MPC using the value function learned from the $[J_{lb}, J_{ub}]$ intervals as its terminal cost.

2. MPC + LQR: One-step mixed-integer MPC using the LQR cost-to-go as its terminal cost.

3. Policy Net: The neural net trained to mimic the optimal policy (Section 5.3.6).

Each controller was evaluated by simulating the cart-pole for 4 seconds from a range of initial velocities. Each simulation began with the cart centered ($q_1 = 0$) and the pole upright ($q_2 = 0$), with initial cart velocity ($\dot{q}_1$) ranging uniformly from -8 $\mathrm{m\,s^{-1}}$ to 8 $\mathrm{m\,s^{-1}}$ and initial pole rotational velocity ($\dot{q}_2$) ranging uniformly from $-\pi\,\mathrm{rad\,s^{-1}}$ to $\pi\,\mathrm{rad\,s^{-1}}$. Eleven samples of each initial velocity were collected, for a total of 121 simulations of each controller. An example simulation, showing the MPC + learned value controller recovering from $\dot{q} = \begin{bmatrix} 0 \\ 8 \end{bmatrix}$, can be seen in Figure 5-6. Performance of the controller was evaluated by measuring the total accumulated cost (using the same quadratic cost matrices Q and R that were used to design the LQR controller) over each simulation.



(a) Cost-to-go learned from offline trajectory optimizations.

(b) Cost-to-go of the LQR policy.

Figure 5-8: Comparing the learned cost-to-go with LQR as a function of the initial cart velocity ($\dot{q}_1$, horizontal axis) and initial pole velocity ($\dot{q}_2$, vertical axis). Note that the cost-to-go is a scalar function of 4 variables, so the plots show only a 2D slice corresponding to $q_1 = 0$, $q_2 = 0$. The learned cost is substantially lower in the regions of higher cart velocity, as the dissipative impact with the walls can be exploited when the cart is moving quickly.

Results of the cart-pole simulation are shown in Figure 5-7. The LVIS controller (using the learned value function) performed better than the MPC controller which used only the LQR value function, resulting in a lower accumulated cost and successful stabilization of the pole from a wider range of initial velocities. The policy net controller (see Section 5.3.6) was still able to stabilize the pole from a few initial velocities, but it accumulated more cost than either of the MPC approaches in nearly every case.

The learned cost-to-go is compared with the baseline LQR cost-to-go in Figure 5-8. The learned cost-to-go shows a much shallower slope in the regions of higher cart velocity, essentially indicating that high initial cart velocities are less costly than LQR would predict. That can be explained by the accumulated knowledge from the trajectory optimization samples: the presence of the walls allows the system to dissipate energy through contact, making control of the pole possible from a wider array of initial velocities. The lower cost-to-go of the learned policy reflects that knowledge.

5.4.2 Planar Humanoid

The training and evaluation process for the planar humanoid robot model was similar to that of the cart-pole, although substantially more data collection was required. Approximately 33,700 trajectory optimization samples were collected, and each trajectory optimization had a horizon of 10 and a time step of 50 ms, for a total lookahead of 0.5 s. As with the cart-pole, the trajectory optimizations were terminated after 3 seconds of optimization with Gurobi. The higher state dimension and larger number of discrete modes made the humanoid trajectory optimization problems substantially harder to solve within that time limit, resulting in a large fraction of suboptimal solutions, shown in Figure 5-5. For evidence of just how difficult the trajectory optimizations were to solve to full optimality, see Figure 5-9.

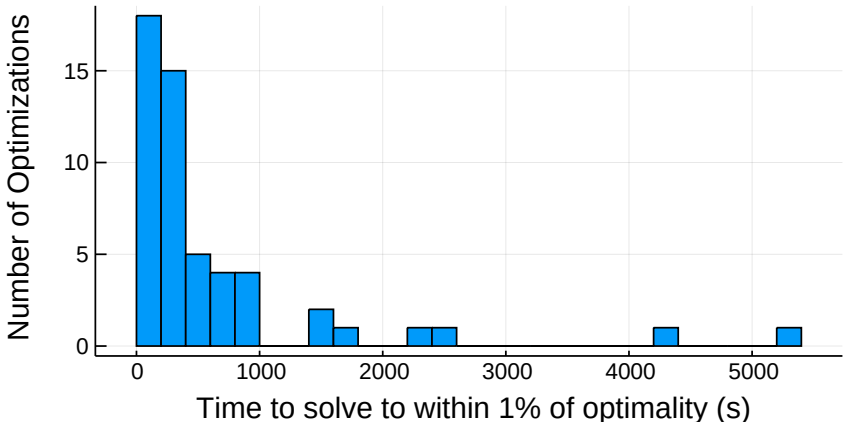


Figure 5-9: A demonstration of the difficulty of solving the trajectory optimization MIQPs for the humanoid model (Figure 5-1) to near global optimality. A total of 54 samples were collected over 18 hours of optimization with the Gurobi solver using the DAGGER technique (Section 5.3.5), with each optimization terminated upon reaching a 1% optimality gap (i.e. a solution with a cost within 1% of the provable global optimum). The minimum solve time was 0.24 s, and the maximum was 5367 s, with a mean of 664 s. In addition, one optimization was terminated after 27,775 seconds (over 7.7 hours) having still only reached 1.29% of optimality. Since that optimization was not completed, it is not shown on the plot, but its inclusion brings the mean solve time up to 1157 s. We do not have sufficient information to explain the variability in solution time, but it appeared to be closely related to the robot's initial state: states close to the robot's nominal state, for which the LQR warm-start produced an optimal or near-optimal mode sequence, tended to result in the fastest MIQP solutions. States with large initial velocities, requiring optimal behaviors very different from the LQR warm-start, tended to have the longest MIQP solution times.

Baseline LQR Policy

The method of Mason et al. [86] was used to generate an LQR policy consistent with the contact dynamics of the humanoid (the similar method of [87] could also be used). The LQR policy was designed for the nominal configuration of the robot, shown in the left-most column of Figure 5-10, with both feet in contact with the ground. As was the case for the cart-pole, the baseline controller comprised a one-step mixed-integer model-predictive controller with the LQR cost-to-go as its terminal state cost. The LQR policy was designed in the nullspace of the contact constraints, so it implicitly assumed that the robot's feet never move.

Training

Training the humanoid value function required approximately 36 hours, again with the majority spent collecting trajectory optimization samples. A total of 300 rounds of training with the ADAM optimizer were performed, and convergence was estimated from 20% of the training samples held as a validation set.

Policy Net

A policy net was also trained on the humanoid optimization samples in an attempt to directly learn the mapping from state to action. The policy net also had two hidden layers with 48 units each, but had 11 outputs, corresponding to the 11 input dimensions of the robot. The same DAGGER training process was run for the policy net, and the same 33,700 samples were provided for training.

Evaluation

The learned controller was evaluated by simulating the humanoid robot from a variety of initial velocities. From the nominal configuration, the robot's initial linear velocity (along the y axis of Figure 5-1) was varied from -1.5 $\mathrm{m\,s^{-1}}$ to 1.5 $\mathrm{m\,s^{-1}}$ and its initial angular velocity (about the x axis of Figure 5-1) was varied from $-\pi\,\mathrm{rad\,s^{-1}}$ to $\pi\,\mathrm{rad\,s^{-1}}$. The robot was then simulated under each control policy for 4 seconds using a simulated control rate of 100 Hz.

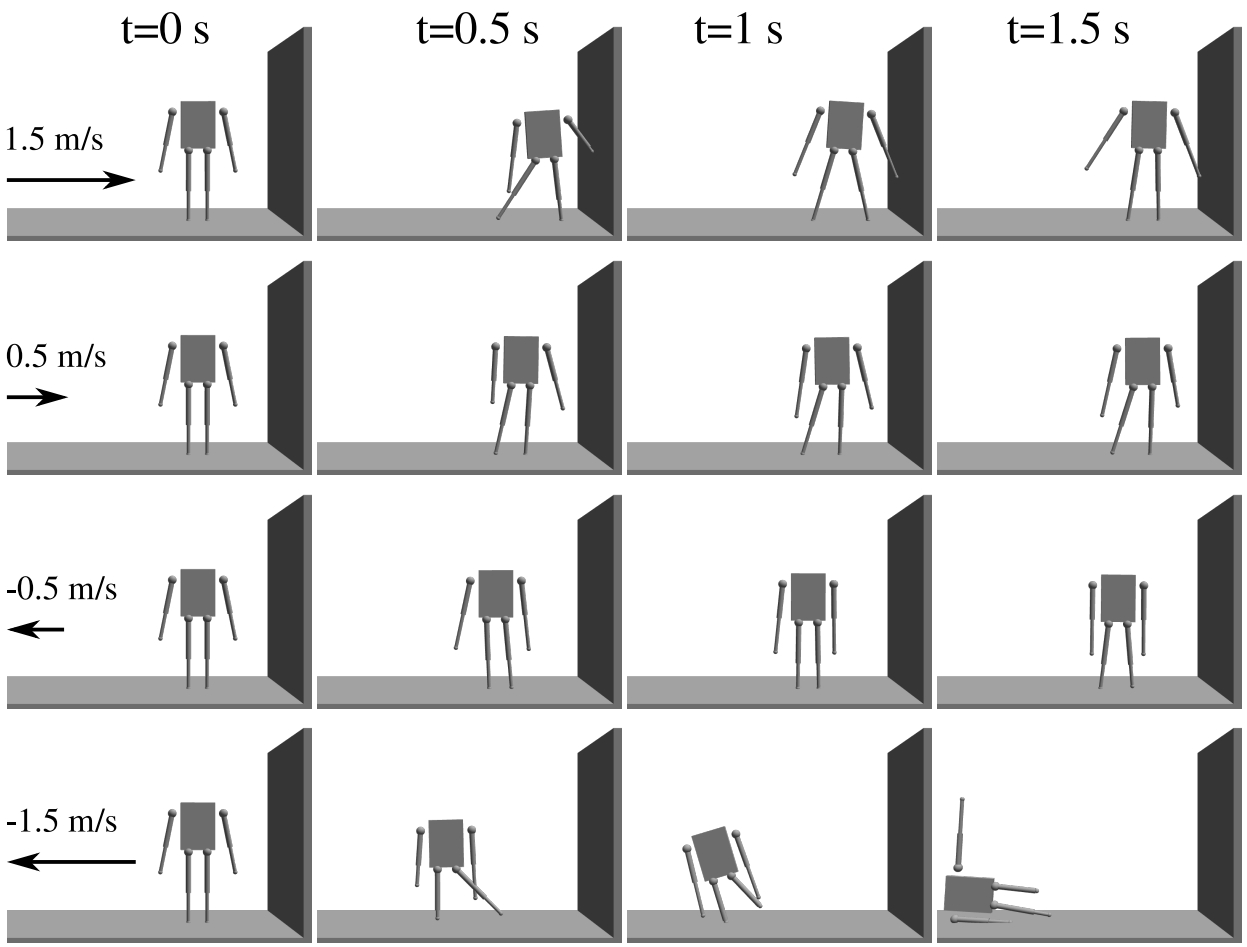


Figure 5-10: Animations of the planar humanoid recovering from pushes using the LVIS controller. Initial velocities refer to the velocity of the robot's torso along the y axis of Figure 5-1. Note that the robot can recover from a $1.5\,\mathrm{m\,s^{-1}}$ velocity to the right by using contact with the wall, but cannot recover from the same initial velocity to the left.

As was the case with the cart-pole, the controller using the learned cost-to-go (trained from the J_{lb} and J_{ub} samples as described in Section 5.3.3) generated substantially lower running cost than the baseline controller using the LQR cost-to-go. In particular, the learned controller performed especially well when the robot's initial velocity directed it towards the wall, since the learned controller was able to both step and reach for the wall in order to maintain balance. The behavior of the learned cost-to-go controller from a variety of initial velocities can be seen in Figure 5-10, and the LVIS controller is compared with LQR and the Policy Net in Figure 5-11.

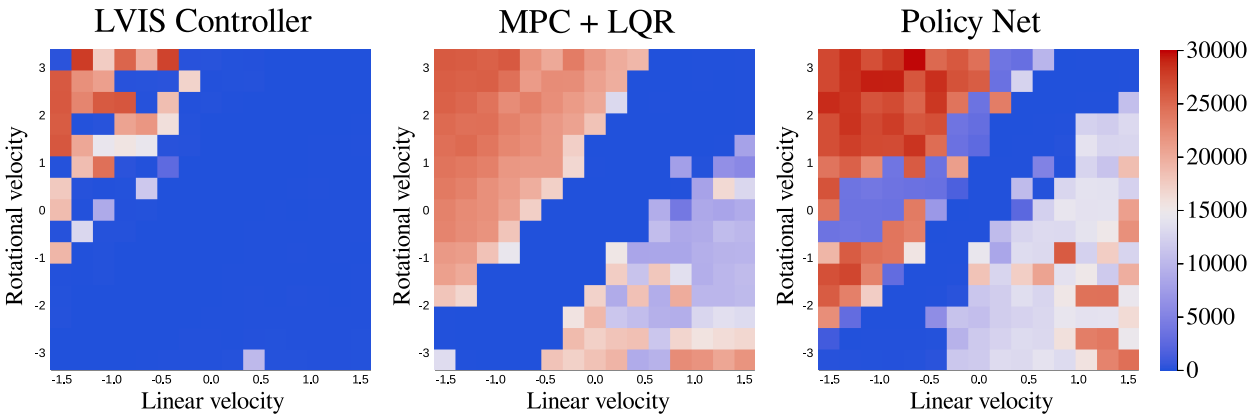


Figure 5-11: Accumulated cost of the humanoid controllers. Each cell indicates the total accumulated cost over a 4-second simulation from the given initial linear and angular velocity of the robot's body, using the same cost matrices as the LQR controller. LVIS achieved a low accumulated cost (dark blue) across a wide variety of initial conditions, performing particularly well in the bottom-right corner of the grid in which the initial velocity moved the robot towards the wall.

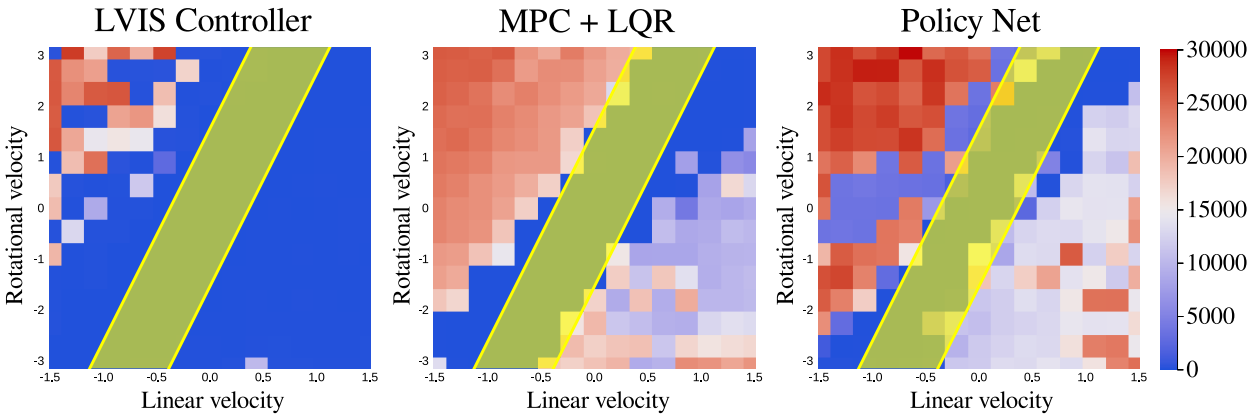


Figure 5-12: Comparing the performance of the humanoid controllers with the zero-step capturability region predicted by [88]. Each cell indicates the accumulated cost, as in Figure 5-11. For initial velocities in the yellow shaded region, the estimated Instantaneous Capture Point (ICP) lies between the robot's feet, so it should be possible for a controller to stabilize the center of mass without taking a step. The set of states stabilized by the LQR controller, indicated by the very low (dark blue) accumulated cost, approximately matches the region predicted by the ICP, while the LVIS controller stabilizes a much larger region.

Capturability Analysis

Figure 5-11 shows that the controller using the learned cost-to-go out-performs the baseline LQR controller, but it does not indicate whether the baseline LQR controller was particularly

effective. It could simply be the case that the baseline LQR controller performed very poorly, making it easy to beat. To evaluate the performance of both controllers with respect to an independent benchmark, we can apply the *capture point* work of Pratt et al. [88] to estimate the range of initial velocities for which the controller should be able to recover without taking a step.

The capture point approach relies on the assumption that the robot's center-of-mass does not accelerate in the z direction. It defines a single point, the Instantaneous Capture Point (ICP) on the ground, such that if the robot instantly places its center of pressure at that point, then its center of mass will come to rest directly above that point. For our humanoid model with both feet on the ground, the center of pressure can be placed anywhere between the two feet. Thus, as long as the Instantaneous Capture Point is also located between the two feet, the robot should be able to balance without taking a step. Figure 5-12 shows the set of velocities for which the ICP can be estimated to lie between the robot's feet and demonstrates that the baseline controller, using the LQR cost-to-go, can stabilize the robot from that entire range of initial velocities. The learned cost-to-go, on the other hand, stabilizes the entire region predicted by the ICP as well as a much larger set of initial velocities for which stepping or reaching out to the wall is necessary to maintain balance.

Note that the correspondence between the yellow region of stability predicted by the ICP in Figure 5-12 does not align perfectly with the set of states which are stabilized by the controller using the LQR cost-to-go. The ICP results from [88] assume that the robot consists of a single uniform flywheel, while the joints of our humanoid robot allow its moment of inertia to change. We can also expect the ICP results to somewhat underestimate the set of states for which the controller can recover: While the zero-step capturability region (the yellow region in Figure 5-12) requires that the robot never take a step, the LQR controller was observed to occasionally slide its foot in the direction of a fall, allowing it to recover from slightly larger pushes.

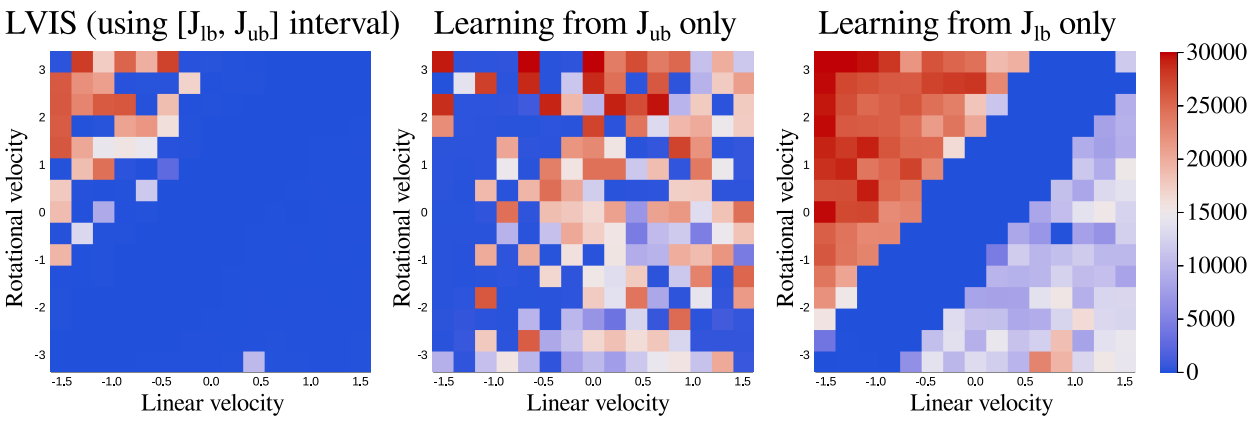


Figure 5-13: Comparing learning the cost-to-go from the bounded intervals (Section 5.3.3), just the upper bound samples J_{ub}, or just the lower bound samples J_{lb}. Accumulated cost was measured as in Figure 5-11. Neither the upper nor the lower bounds alone were sufficient to train an approximate cost-to-go which out-performed the LQR baseline or the LVIS approach (left).

The Importance of Intervals

Since solving the offline trajectory optimization problems could take minutes or hours for the full humanoid system, all trajectory optimizations were terminated after a fixed 3 second time limit, as discussed in Section 5.3.2. The optimizations were thus rarely solved to global optimality, and the actions u returned by the optimizations were often highly sub-optimal. This explains, to some extent, the very poor performance of the policy net, shown in Figure 5-11: Our training data consist of samples of sub-optimal actions, so there is a great deal of noise and bias that would have to be overcome to learn the optimal policy.

Since the actions u cannot be trusted at each sample, what about the cost-to-go? Each partial optimization produces two values: J_{ub}, the best optimal cost found so far, and J_{lb} the tightest lower bound on optimal cost so far. As described in Section 5.3.3, we only penalize the neural net for predicting a cost-to-go which is outside of the interval $[J_{lb}, J_{ub}]$. To test the validity of that approach, we trained the neural net to exactly mimic the best feasible value J_{ub} or the best lower bound J_{lb} as a different way of approximating the cost-to-go.

Two additional neural nets were trained using the same neural net structure, number of samples, and training process as in Section 5.4.2. Each net was penalized for the ℓ^1 error

between its prediction and J_{lb} or J_{ub}, respectively.

We evaluated both of these cost-to-go approximations using the same simulation procedure as in Figure 5-11. Neither the J_{ub} samples alone nor the J_{lb} samples alone produced a cost-to-go and controller which could out-perform even the LQR baseline, as shown in Figure 5-13. In practice, attempting to train from just J_{lb} or J_{ub} resulted in substantial under-fitting, as the upper and lower bound data both showed a great deal of noise from one sample to the next, influenced by the quality of the warm-start solutions, and the varying behavior of Gurobi's internal heuristics.

The lesson to be drawn from the failures of the policy net, the upper bound net, and the lower bound net, and from the success of the interval net, is that the cost-to-go bounds generated by the branch-and-bound procedure are valuable information. The interval $[J_{lb}, J_{ub}]$ can be trusted even when the individual upper and lower bounds are far from the optimal cost and even when the optimization is terminated with a highly sub-optimal solution.

5.5 Learning in Parameterized Environments

One drawback of the LVIS approach is that the offline training and trajectory optimization are performed in a fixed environment. This essentially bakes that environment into the learned cost-to-go, resulting in a controller which is only useful in the trained environment. To some extent, we can side-step the issue by varying the robot's initial state. For example, although the humanoid model in Figure 5-1 is only trained in a single environment with a wall and a floor, by just varying the initial y and z position of the robot relative to its modeled floor and walls, we can fit the learned controller to a wall at a variety of distances. To make this approach more broadly applicable, however, we need the ability to handle more diverse environments.

One approach we can take when handling a variety of environments it to create parameterized templates representing deformable environments. This idea is similar to the templated affordance fitting in [19]. By encoding the environment parameters into the input to the LVIS neural net (both in training and at run-time), we can create a learned cost-to-go

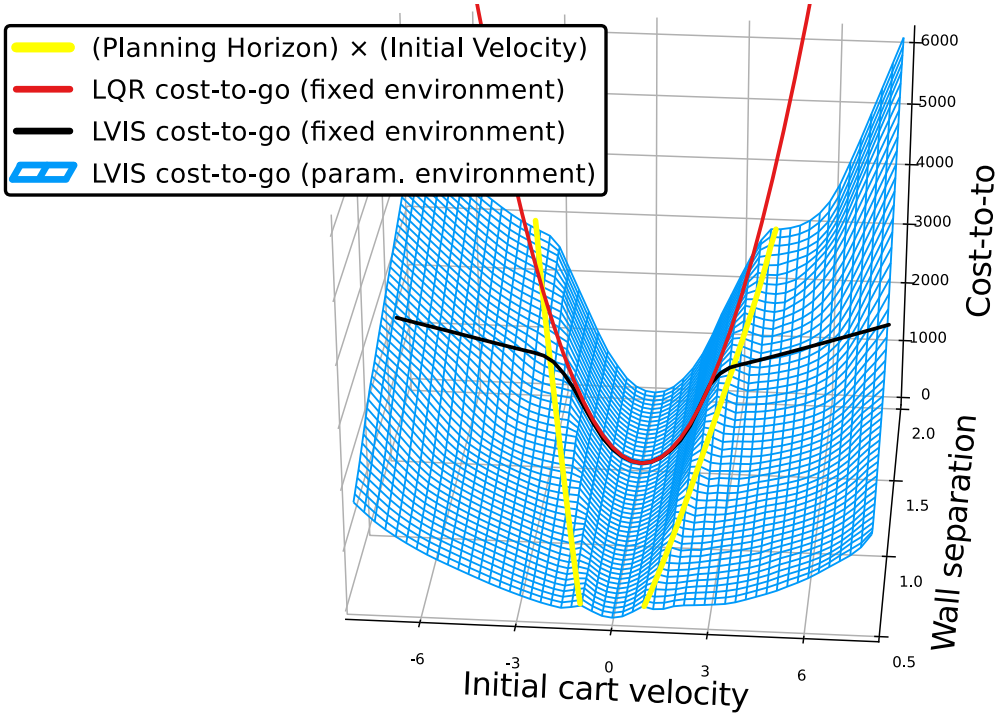


Figure 5-14: Learned cost-to-go (blue mesh) as a function of the initial cart velocity $\dot{q}_1$ and the parameter representing the distance from the center of the track to each wall. The parameterized cost-to-go closely matches LQR (in red) for states for which the cart will not reach the wall within the planning horizon (between the yellow lines), while it rises more gently elsewhere as the walls enable the robot to dissipate energy and reduce its cost. The black line shows the cost function learned in Section 5.4.1, for which the wall distance was fixed at 1.5 m.

which is a function both of the robot's state and the environment parameters.

As a very basic demonstration of this approach, we modified the cart-pole environment (Figure 5-2), adding a single parameter to represent the distance from the center of the track to the walls. The same training process as in Section 5.4.1 was run, with 20,982 trajectory optimizations samples collected over 18 hours. For each iteration of the DAGGER training, the distance to the walls was uniformly randomly varied from 0.5 m to 2.0 m. The resulting learned cost-to-go is shown in Figure 5-14.

It is particularly interesting to examine the sharp change in the parametric learned cost in Figure 5-14. This sharp change occurs quite close to the line, marked in yellow on both figures, corresponding to a wall distance which is exactly 0.5 times the initial cart velocity.

We can easily explain this change: the time horizon of the offline trajectory optimizations was precisely 0.5 s (see Section 5.4.1), so the yellow lines correspond to the combination of initial cart velocity and wall distance such that the cart will, at constant velocity, impact the walls precisely at the end of the trajectory optimization horizon. For lower initial velocities (or higher wall distances), the cart can be expected not to reach the wall within the planning horizon, so any further distance to the wall is irrelevant. Moreover, when the cart cannot reach the wall within the planning horizon, we expect the cost-to-go to match the (contact-unaware) LQR cost-to-go, as shown in red in Figure 5-14.

This cutoff shows both the effectiveness of learning the cost-to-go from trajectory optimizations and also one of its weaknesses. With a fixed trajectory optimization horizon and globally valid bounds on the cost-to-go from mixed-integer optimization, we can clearly see the effect of future contacts in the learned cost-to-go. But that effect is not infinite: any contacts that would occur outside of the planning horizon can not be reflected in the learned cost-to-go. In the future, it will be interesting to compare the trade-offs of longer planning horizons. We would expect longer horizons to give better cost-to-go samples at optimality, but also to be harder to solve within a given time limit. It is an open question as to whether it is worthwhile to try to solve harder optimizations which may produce looser bounds J_{lb} and J_{ub}.

One final area of interest which we have not yet explored is bootstrapping the entire LVIS process using a previously learned cost-to-go. Specifically, the offline trajectory optimization still requires a terminal cost, which we have so far limited to a quadratic term $\mathbf{x}^{N\top} S \mathbf{x}^N$ in (5.4). We have so far only used the baseline LQR cost-to-go as that terminal cost, which assumes that the trajectory optimization horizon is long enough to bring the system close enough to the nominal state that the LQR cost-to-go is accurate. That assumption is unlikely to hold for complex systems like our humanoid executing maneuvers far from the nominal state, so a more accurate terminal cost might prove valuable. We could instead use the learned cost-to-go from LVIS to replace that terminal cost and then run the entire data collection and training process again to produce a new learned cost-to-go function. This

procedure could even be repeated multiple times as a means of value iteration to hopefully converge to the true cost-to-go (although we cannot currently offer a proof of convergence for such a hypothetical approach). However, since the learned cost-to-go is not a convex quadratic function but rather the potentially complicated output of a neural net, its use as a terminal cost for trajectory optimization may prove difficult. While we were able to use a local linear approximation of the learned cost in our online one-step MPC, this was only possible because we could fit that local linear approximation to the robot's current state, on the assumption that the state would not change substantially over the course of just one time step. But for longer trajectory optimizations over many time steps, the robot's state is likely to change much more, and any linear approximation is likely to be too inaccurate. Instead, we would need to either try to compute a sufficiently accurate convex quadratic approximation of the neural net in the vicinity of the robot's initial state or just include the full neural net output in our optimization and switch to a fully nonlinear mixed-integer optimization (as discussed further in Section 6.1).

5.6 Conclusion

LVIS allows us to collect data from offline mixed-integer trajectory optimizations in order to train an approximation of the cost-to-go of a dynamical system. By solving the trajectory optimizations only to partial optimality, we can vastly increase the number of samples collected while still extracting useful intervals containing the true cost from the branch-and-bound algorithm. So far, however, LVIS has only been applied to relatively small systems with dynamics modeled as piecewise affine functions. In the next chapter, we will discuss opportunities to move beyond those limitations and bring LVIS to more complex robots.

Chapter 6

Future Work in Learned Control

6.1 Handling Nonlinear Dynamics

The results presented in Chapter 5 relied on the use of a system with piecewise affine dynamics (an approximation of the true dynamics of the robots shown) in order to allow the trajectory optimizations to be written as mixed-integer programs with linear constraints. This in turn was necessary in order for the standard branch-and-bound algorithms implemented by solvers like Gurobi [21] to provide rigorous bounds on the optimal cost-to-go without solving the full optimization. For LVIS to be useful for a wider range of systems, however, we need to remove this restriction.

For example, if we were to try to use an approach like LVIS to control the swing-up behavior of a pendulum, we would need to reason about dynamics involving sines and cosines of the pendulum angle. While the dynamics of the cart-pole and humanoid examples from the previous chapter also involved trigonometric functions of the state variables, those systems remained close enough to an upright operating point that a single linearization provided a sufficiently good model. This is not the case for a pendulum executing a trajectory that passes through a wide range of angles, as any single linearization of the trigonometric functions of that angle would be completely invalid for large deviations from the chose linearization point. We could imagine dividing up the space of angles into bins and creating a separate

linearization for each bin (just as in Figure 2-3), but this makes each mixed-integer program much more complex, as the discrete choice among bins of angles becomes yet another discrete mode in the optimization. Rotations about more than a single axis are even more challenging to effectively divide into appropriate bins, as explored in [89].

A related problem occurs when trying to stabilize more complex motions of a legged robot due to the coupling of contact force and contact position. The torque $\vec{\tau}$ produced by a force $\vec{f}$ acting at a displacement $\vec{r}$ is equal to $\vec{r} \times \vec{f}$. If, as in the trajectory optimizations used for LVIS, $\vec{r}$ and $\vec{f}$ are both decision variables, then $\vec{\tau}$ is a bilinear product of decision variables and therefore cannot be used in a linear constraint. Again in Chapter 5 we chose to select a particular operating point and linearize $\vec{\tau}$ about that point, but this prevents the trajectory optimization from being able to reason about the interactions between contact forces and their positions. This issue is discussed in detail in [52], in which a series of approximations to the $\vec{\tau} = \vec{r} \times \vec{f}$ constraint are constructed, with additional mixed-integer variables used to choose the appropriate approximation. The resulting optimizations, however, become harder to solve due to the larger number of integer variables, and the results are still only an approximation, controlled by the granularity of the approximate constraints.

It turns out, however, that the general ideas which allow branch-and-bound to find rigorous upper and lower bounds for mixed-integer programs with linear constraints can also be applied to a much wider category of optimizations. One particularly relevant technique is Spatial Branch-and-Bound, which expands the branch-and-bound algorithm to handle continuous, non-linear and non-convex constraints. In essence, the spatial branch-and-bound involves dividing up the space of continuous variables and using using rigorous lower bounds (produced by relaxing constraints) and upper bounds (produced by local optimization) on the optimal cost to eliminate useless regions of state space. Just as in standard branch-and-bound, this variant iterates between improving the upper and lower bounds until the bounds converge or the optimization is terminated. In this way, spatial branch-and-bound can actually provide globally optimal solutions to difficult non-convex optimization problems.

While this approach sounds extremely promising, it has a significant drawback which is

that performance of spatial branch-and-bound has been very poor for the kinds of trajectory optimizations studied here. Valenzuela attempted to use the COUENNE [68] solver for trajectory optimization of a planar robot model in [52], but found its performance too slow even for offline use. The BARON [69] solver promises better performance on many problems, but so far has still not been efficient enough to actually solve complex trajectory optimizations to global optimality.

However, the point of LVIS is that global optimality may not be necessary: only the ability to produce upper and lower bounds on the cost-to-go is required. This may alleviate some of the computational cost of solvers like COUENNE and BARON: rather than solving to optimality, which is quite expensive, we can simply terminate the solution process early and extract a model for the cost-to-go which can be used online. In this way, LVIS could be applied to arbitrary dynamical systems, even those which are not well represented by piecewise affine dynamics.

6.2 Scaling LVIS to a Full Humanoid

Beyond the issue of handling nonlinear dynamics there is also a question of how to apply LVIS to systems with many more degrees of freedom than the models used in Chapter 5. For example, the ATlas humanoid robot, used by several teams during the DARPA Robotics Challenge, has 36 degrees of freedom and (assuming a 6-DoF floating joint between the robot and the world) and 72 states [6], while the planar humanoid model of Chapter 5 has just 11 DoF and 22 states. The full Atlas robot also has more complex contact between its limbs and the world than the planar humanoid model, so even more complementarity conditions would be required in order to fully model its dynamics. The additional complexity of a robot like Atlas suggests that while we could certainly attempt to run LVIS exactly as proposed (perhaps also using the full nonlinear dynamics with a spatial branch-and-bound solver), we may find it difficult to gather enough data to train the approximate cost-to-go in a reasonable amount of time.

On the other hand, while the full dynamics of the robot must be accounted for at run-

time, we may be able to create effective behaviors using a simpler model and then apply those behaviors to the full dynamics at run time. This is the general idea behind the widely used linear inverted pendulum model (LIPM) controllers for humanoid robots (see [70, 6, 90, 91] for some examples). The LIPM approach treats the robot as if it is simply a mass on a stick moving in a horizontal plane when planning footsteps, and then uses a QP to choose appropriate contact forces and joint torques at each instant to drive the full dynamics of the robot to match the desired LIPM behavior.

The success of LIPM for humanoid robot walking suggests that we could apply a similar approach with LVIS. By creating a simplified humanoid (simlar to the one in Chapter 5 but without the planar constraint) whose mass distribution and kinematic reach are similar to those of the full Atlas, and running LVIS on that simple model, we should be able to train a cost-to-go which encodes appropriate behaviors like balancing, stepping in the direction of a fall, and using the arms to make contact with the world. Online, we can then use the LVIS cost-to-go to control the full robot's dynamics.

There are a challenges involved in actually making this approach work. First, from the current state of the full Atlas robot, we need a way to compute an equivalent state of the simplified model. This likely requires using the limited degrees of freedom of the simplified model to match the positions of the full Atlas robot's torso, hands, and feet, while also trying to keep the center of mass of the two models close together. Second, we need to decide how to map the behavior of the LVIS controller running on the simplified model up to the full robot's dynamics. Two interesting options for performing this mapping are: (1) trajectory matching and (2) cost-to-go transfer.

In the trajectory matching approach, we would simulate running the LVIS controller on the simplified model for one or more time steps (as many as our control rate allows) and extract a trajectory of motions of the simplified robot's body and limbs. Precisely which information we would extract is an open question, but it seems reasonable to use the same body poses which were used when matching the simplified model's configuration to the full robot state. From these extracted trajectories, we can create desired motions of the full

robot's limbs, which would then be tracked by a whole-body QP controller. In this way, the complex problem of planning which contacts to make is left to the simplified model while the full dynamics of the robot are handled at each time step by the QP.

The trajectory matching approach, while promising, also presents a number of challenges. Generating the trajectories from the reduced model requires simulating the behavior of that model under the LVIS controller. That real-time simulation may be computationally overwhelming and may also introduce artifacts in the motion of the full robot as it attempts to precisely mimic the simulated result. Furthermore, if the simulated simple model becomes unstable, the full robot will almost certainly lose stability as it attempts to track the unstable motion, so some mechanism would be required to detect such instabilities and avoid propagating them to the full model.

Instead, there is a second option: direct transfer of the cost-to-go. By locally linearizing the map from full robot state to simple model state, we can compute the gradient of the LVIS cost-to-go (which was constructed in terms of the simple model) with respect to the state of the full robot via the chain rule. We can then construct a controller for the full robot which attempts to descend the gradient of this mapped cost-to-go, using a short-horizon MPC controller just as we did when running LVIS online. This approach completely avoids the need for simulation and also creates opportunities for other inputs to the controller, as the cost function in its MPC optimization can involve additional terms beyond just the LVIS learned cost-to-go. We should not expect a cost-to-go which was learned from a simple model to lead to optimal behavior of the full robot's dynamics, but it is still likely to be an informative heuristic. In essence, this can be thought of as an extension of the control approach presented in [41], but with the LIPM replaced with the simplified humanoid model.

Bibliography

[1] Robin Deits and Russ Tedrake. Footstep Planning on Uneven Terrain with Mixed-Integer Convex Optimization. In *IEEE-RAS International Conference on Humanoid Robots*, November 2014.

[2] Robin Deits and Russ Tedrake. Efficient Mixed-Integer Planning for UAVs in Cluttered Environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, WA, May 2015.

[3] Robin Deits, Twan Koolen, and Russ Tedrake. LVIS: Learning from Value Function Intervals for Contact-Aware Robot Controllers. *Under Review*, September 2018.

[4] DARPA Tactical Technology Office. DARPA Robotics Challenge. https://www.darpa.mil/program/darpa-robotics-challenge, 2012.

[5] Russ Tedrake. Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation. http://underactuated.csail.mit.edu/underactuated.html, March 2018.

[6] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 40(3):429–455, March 2016.

[7] Robin Deits and Russ Tedrake. Computing Large Convex Regions of Obstacle-Free Space through Semidefinite Programming. In *Workshop on the Algorithmic Foundations of Robotics*, Istanbul, Turkey, 2014.

[8] A. Hornung, A. Dornbush, M. Likhachev, and M. Bennewitz. Anytime search-based footstep planning with suboptimality bounds. In *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 674–679, November 2012.

[9] Philipp Michel, Joel Chestnutt, James Kuffner, and Takeo Kanade. Vision-guided humanoid footstep planning for dynamic environments. In *IEEE-RAS International Conference on Humanoid Robots*, pages 13–18, 2005.

[10] Léo Baudouin, Nicolas Perrin, Thomas Moulard, Florent Lamiraux, Olivier Stasse, and Eiichi Yoshida. Real-time Replanning Using 3D Environment for Humanoid Robot. In *IEEE-RAS International Conference on Humanoid Robots*, pages 584–589, Bled, Slovénie, 2011.

[11] Joel E. Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning Biped Navigation Strategies in Complex Environments. In *IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, Germany, 2003.

[12] Joel E. Chestnutt, Koichi Nishiwaki, James Kuffner, and Satoshi Kagami. An adaptive action model for legged navigation planning. In *IEEE-RAS International Conference on Humanoid Robots*, pages 196–202, 2007.

[13] James J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. Footstep planning among obstacles for biped robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 500–505, Maui, Hawaii, 2001.

[14] James J. Kuffner, Koichi Nishiwaki, Satoshi Kagami, Masayuki Inaba, and Hirochika Inoue. Online footstep planning for humanoid robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 932–937, 2003.

[15] A Shkolnik, M Levashov, I R Manchester, and R Tedrake. Bounding on rough terrain with the LittleDog robot. *The International Journal of Robotics Research*, 30(2):192–215, 2011.

[16] Tim Bretl, Sanjay Lall, Jean-Claude Latombe, and Stephen Rock. Multi-Step Motion Planning for Free-Climbing Robots. In Michael Erdmann, Mark Overmars, David Hsu, and Frank van der Stappen, editors, *Algorithmic Foundations of Robotics VI*, number 17 in Springer Tracts in Advanced Robotics, pages 59–74. Springer Berlin Heidelberg, January 2005.

[17] Peter D. Neuhaus, Jerry E. Pratt, and Matthew J. Johnson. Comprehensive summary of the Institute for Human and Machine Cognition's experience with LittleDog. *The International Journal of Robotics Research*, 30(2):216–235, January 2011.

[18] Stephen P Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK; New York, 2004.

[19] Maurice Fallon, Scott Kuindersma, Sisir Karumanchi, Matthew Antone, Toby Schneider, Hongkai Dai, Claudia Perez D'Arpino, Robin Deits, Matt DiCicco, Dehann Fourie, Twan Koolen, Pat Marion, Michael Posa, Andres Valenzuela, Kuan-Ting Yu, Julie Shah, Karl Iagnemma, Russ Tedrake, and Seth Teller. An Architecture for Online Affordance-based Perception and Whole-body Planning. *Journal of Field Robotics*, 32(2):229–254, March 2015.

[20] Andrei Herdt, Holger Diedam, Pierre-Brice Wieber, Dimitar Dimitrov, Katja Mombaur, and Moritz Diehl. Online Walking Motion Generation with Automatic Foot Step Placement. *Advanced Robotics*, 24(5-6):719–737, 2010.

[21] Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual. http://www.gurobi.com/, 2014.

[22] Mosek ApS. The MOSEK optimization toolbox for MATLAB manual. Version 7.0 (Revision 141). https://docs.mosek.com/7.0/toolbox/index.html, 2014.

[23] IBM Corp. User's Manual for CPLEX. www.cplex.com, 2010.

[24] Arthur Richards, John Bellingham, Michael Tillerson, and Jonathan How. Coordination and Control of Multiple UAVs. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*. American Institute of Aeronautics and Astronautics, Monterey, CA, August 2002.

[25] Johan Lofberg. Big-M and Convex Hulls. http://users.isy.liu.se/johanl/yalmip/pmwiki.php, 2012.

[26] MATLAB. *Version 8.2.0.701 (R2013b)*. The MathWorks Inc., Natick, MA, 2013.

[27] Russ Tedrake. Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems. http://drake.mit.edu, 2014.

[28] Per-Erik Danielsson and Olle Seger. Generalized and Separable Sobel Operators. In Herbert Freeman, editor, *Machine Vision for Three-Dimensional Scenes*. Academic Press, Inc., Sand Diego, CA, 1990.

[29] Richard M. Karp. Reducibility Among Combinatorial Problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[30] D. Mellinger, A Kushleyev, and V. Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 477–483, May 2012.

[31] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How. Mixed Integer Programming for Multi-Vehicle Path Planning. In *European Control Conference*, Porto, Portugal, 2001.

[32] Kieran Forbes Culligan. *Online Trajectory Planning for UAVs Using Mixed Integer Linear Programming*. Thesis, Massachusetts Institute of Technology, 2006. Thesis

(S.M.)–Massachusetts Institute of Technology, Dept. of Aeronautics and Astronautics, 2006.

[33] Yongxing Hao, Asad Davari, and Ali Manesh. Differential flatness-based trajectory planning for multiple unmanned aerial vehicles using mixed-integer linear programming. In *Proceedings of the American Control Conference*, volume 1, page 104, 2005.

[34] D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2520–2525, May 2011.

[35] John Saunders Bellingham. *Coordination and Control of Uav Fleets Using Mixed-Integer Linear Programming*. PhD thesis, Citeseer, 2002.

[36] Melvin E. Flores. *Real-Time Trajectory Generation for Constrained Nonlinear Dynamical Systems Using Non-Uniform Rational B-Spline Basis Functions*. PhD thesis, California Institue of Technology, Pasadena, CA, 2007.

[37] Pablo A. Parrilo. Sums of Squares and Semidefinite Programming. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-972-algebraic-techniques-and-semidefinite-optimization-spring-2006/lecture-notes/lecture_10.pdf, March 2006.

[38] Victoria Powers and Thorsten Wörmann. An algorithm for sums of squares of real polynomials. *Journal of Pure and Applied Algebra*, 127(1):99–104, May 1998.

[39] Johan Lofberg. YALMIP Wiki. http://users.isy.liu.se/johanl/yalmip/, 2012.

[40] Sonja Mars and Lars Schewe. An SDP-package for SCIP. Technical report, Technical report, TU Darmstadt, 2012.

[41] Russ Tedrake, Scott Kuindersma, Robin Deits, and Kanako Miura. A closed-form solution for real-time ZMP gait generation and feedback stabilization. In *International Conference on Humanoid Robots*, Seoul, South Korea, November 2015.

[42] Maurice F. Fallon, Pat Marion, Robin Deits, Thomas Whelan, Matthew Antone, John McDonald, and Russ Tedrake. Continuous humanoid locomotion over uneven terrain using stereo fusion. In *IEEE-RAS International Conference on Humanoid Robots*, pages 881–888, Seoul, South Korea, November 2015. IEEE.

[43] Thomas Whelan, Michael Kaess, Hordur Johannsson, Maurice Fallon, John J. Leonard, and John McDonald. Real-time large-scale dense RGB-D SLAM with volumetric fusion. *The International Journal of Robotics Research*, 34(4-5):598–626, April 2015.

[44] B. Landry, R. Deits, P. R. Florence, and R. Tedrake. Aggressive quadrotor flight through cluttered environments using mixed integer programming. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1469–1475, May 2016.

[45] A. K. Sadhu, R. Dasgupta, and P. Balamuralidhar. EIRIS - An Extended Proposition Using Modified Occupancy Grid Map and Proper Seeding. In *2018 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8, September 2018.

[46] Sergei Savin. An Algorithm for Generating Convex Obstacle-free Regions Based on Stereographic Projection. In *International Siberian Conference on Control and Communications*, page 6, 2017.

[47] Sergey Jatsun, Sergei Savin, and Andrey Yatsun. Footstep Planner Algorithm for a Lower Limb Exoskeleton Climbing Stairs. In Andrey Ronzhin, Gerhard Rigoll, and Roman Meshcheryakov, editors, *Interactive Collaborative Robotics*, Lecture Notes in Computer Science, pages 75–82. Springer International Publishing, 2017.

[48] Joshua Bialkowski, Michael Otte, Sertac Karaman, and Emilio Frazzoli. Efficient collision checking in sampling-based motion planning via safety certificates. *The International Journal of Robotics Research*, 35(7):767–796, June 2016.

[49] B. Aceituno-Cabezas, H. Dai, J. Cappelletto, J. C. Grieco, and G. Fernández-López. A mixed-integer convex optimization framework for robust multilegged robot locomotion

planning over challenging terrain. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4467–4472, September 2017.

[50] B. Aceituno-Cabezas, C. Mastalli, H. Dai, M. Focchi, A. Radulescu, D. G. Caldwell, J. Cappelletto, J. C. Grieco, G. Fernández-López, and C. Semini. Simultaneous Contact, Gait, and Motion Planning for Robust Multilegged Locomotion via Mixed-Integer Convex Optimization. *IEEE Robotics and Automation Letters*, 3(3):2531–2538, July 2018.

[51] Hongkai Dai and Russ Tedrake. Planning robust walking motion on uneven terrain via convex optimization. In *IEEE-RAS International Conference on Humanoid Robots*, pages 579–586, Cancun, Mexico, November 2016. IEEE.

[52] Andrés Klee Valenzuela. *Mixed-Integer Convex Optimization for Planning Aggressive Motions of Legged Robots over Rough Terrain*. PhD thesis, Massachusetts Institute of Technology, 2016.

[53] Garth P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems. *Mathematical Programming*, 10(1):147–175, December 1976.

[54] Brahayam Ponton, Alexander Herzog, Stefan Schaal, and Ludovic Righetti. A Convex Model of Momentum Dynamics for Multi-Contact Motion Generation. *arXiv:1607.08644 [cs]*, July 2016.

[55] B. Ponton, A. Herzog, A. Del Prete, S. Schaal, and L. Righetti. On Time Optimization of Centroidal Momentum Dynamics. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–7, May 2018.

[56] James Preiss, Karol Hausman, Gaurav Sukhatme, and Stephan Weiss. Trajectory Optimization for Self-Calibration and Navigation. In *Robotics: Science and Systems XIII*, July 2017.

[57] C. Miller, C. Pek, and M. Althoff. Efficient Mixed-Integer Programming for Longitudinal and Lateral Motion Planning of Autonomous Vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1954–1961, June 2018.

[58] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar. Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments. *IEEE Robotics and Automation Letters*, 2(3):1688–1695, July 2017.

[59] Kartik Mohta, Michael Watterson, Yash Mulgaonkar, Sikang Liu, Chao Qu, Anurag Makineni, Kelsey Saulnier, Ke Sun, Alex Zhu, Jeffrey Delmerico, Konstantinos Karydis, Nikolay Atanasov, Giuseppe Loianno, Davide Scaramuzza, Kostas Daniilidis, Camillo Jose Taylor, and Vijay Kumar. Fast, autonomous flight in GPS-denied and cluttered environments. *Journal of Field Robotics*, 35(1):101–120, January 2018.

[60] Javier Alonso-Mora, Stuart Baker, and Daniela Rus. Multi-robot formation control and object transport in dynamic environments via constrained optimization. *The International Journal of Robotics Research*, 36(9):1000–1021, August 2017.

[61] Hongkai Dai, Andrés Valenzuela, and Russ Tedrake. Whole-body Motion Planning with Simple Dynamics and Full Kinematics. In *IEEE-RAS International Conference on Humanoid Robots*, Madrid, Spain, 2014.

[62] Michael Posa, Cecilia Cantu, and Russ Tedrake. A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research*, 33(1):69–81, January 2014.

[63] Igor Mordatch and Emo Todorov. Combining the benefits of function approximation and trajectory optimization. In *Robotics: Science and Systems*, 2014.

[64] M. Zhong, M. Johnson, Y. Tassa, T. Erez, and E. Todorov. Value function approximation and model predictive control. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 100–107, April 2013.

[65] Farbod Farshidian, Edo Jelavić, Asutosh Satapathy, Markus Giftthaler, and Jonas Buchli. Real-Time Motion Planning of Legged Robots: A Model Predictive Control Approach. *arXiv:1710.04029 [cs]*, October 2017.

[66] Sergey Levine and Vladlen Koltun. Guided Policy Search. In *ICML*, page 10, Atlanta, GA, USA, 2013.

[67] Christodoulos A. Floudas. *Nonlinear and Mixed Integer Optimization: Fundamentals and Applications*. Topics in chemical engineering. Oxford Univ. Press, New York, 1995. OCLC: 246760342.

[68] Pietro Belotti, Jon Lee, Leo Liberti, François Margot, and Andreas Wächter. Branching and bounds tighteningtechniques for non-convex MINLP. *Optimization Methods and Software*, 24(4-5):597–634, October 2009.

[69] Mohit Tawarmalani and Nikolaos V. Sahinidis. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103(2):225–249, June 2005.

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

[71] Francois Robert Hogan, Eudald Romo Grau, and Alberto Rodriguez. Reactive Planar Manipulation with Convex Hybrid MPC. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2018.

[72] Tobia Marcucci, Robin Deits, Marco Gabiccini, Antonio Bicchi, and Russ Tedrake. Approximate hybrid model predictive control for multi-contact push recovery in complex environments. In *IEEE-RAS International Conference on Humanoid Robotics*, Birmingham, UK, November 2017.

[73] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement Learning in Robotics: A Survey. *International Journal of Robotics Research*, July 2013.

[74] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates. *arXiv:1610.00633 [cs]*, October 2016.

[75] David Stewart and Jeffrey C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with coulomb friction. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 162–169. IEEE, 2000.

[76] Michael C. Ferris and Todd S. Munson. Complementarity problems in GAMS and the PATH solver. *Journal of Economic Dynamics and Control*, 24(2):165–188, February 2000.

[77] Robin Deits and contributors. ConditionalJuMP.jl. https://github.com/rdeits/ConditionalJuMP.jl, 2018.

[78] Luis Benet and David P. Saunders. IntervalArithmetic.jl. https://github.com/JuliaIntervals/IntervalArithmetic.jl, 2018.

[79] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. *arXiv:1711.07356 [cs]*, November 2017.

[80] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv:1511.07289 [cs]*, November 2015.

[81] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014.

[82] Twan Koolen and contributors. RigidBodyDynamics.jl. https://github.com/JuliaRobotics/RigidBodyDynamics.jl, 2016.

[83] Twan Koolen and Robin Deits. Parametron.jl. https://github.com/tkoolen/Parametron.jl, 2018.

[84] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.

[85] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to Fly. In Derek Sleeman and Peter Edwards, editors, *Machine Learning Proceedings 1992*, pages 385–393. Morgan Kaufmann, San Francisco (CA), January 1992.

[86] S. Mason, N. Rotella, S. Schaal, and L. Righetti. Balancing and walking using full dynamics LQR control with contact constraints. In *IEEE-RAS International Conference on Humanoid Robots*, Cancun, Mexico, November 2016.

[87] M. Posa, S. Kuindersma, and R. Tedrake. Optimization and stabilization of trajectories for constrained dynamical systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2016.

[88] J. Pratt, J. Carff, S. Drakunov, and A. Goswami. Capture Point: A Step toward Humanoid Push Recovery. In *IEEE-RAS International Conference on Humanoid Robots*, December 2006.

[89] Gregory Izatt and Russ Tedrake. Globally Optimal Object Pose Estimation in Point Clouds with Mixed-Integer Programming. *International Symposium on Robotics Research*, 2017.

[90] Siyuan Feng, Eric Whitman, X. Xinjilefu, and Christopher G. Atkeson. Optimization-based Full Body Control for the DARPA Robotics Challenge. *Journal of Field Robotics*, 32(2):293–312, March 2015.

[91] Matthew Johnson, Brandon Shrewsbury, Sylvain Bertrand, Tingfan Wu, Daniel Duran, Marshall Floyd, Peter Abeles, Douglas Stephen, Nathan Mertins, Alex Lesman, John Carff, William Rifenburgh, Pushyami Kaveti, Wessel Straatman, Jesper Smith, Maarten Griffioen, Brooke Layton, Tomas de Boer, Twan Koolen, Peter Neuhaus, and Jerry

Pratt. Team IHMC's Lessons Learned from the DARPA Robotics Challenge Trials. *Journal of Field Robotics*, 32(2):192–208, March 2015.