Machine Learning For Robot Motion Planning

Clark June Zhang
University of Pennsylvania

Machine Learning For Robot Motion Planning

Abstract

Robot motion planning is a field that encompasses many different problems and algorithms. From the traditional piano mover's problem to more complicated kinodynamic planning problems, motion planning requires a broad breadth of human expertise and time to design well functioning algorithms. A traditional motion planning pipeline consists of modeling a system and then designing a planner and planning heuristics. Each part of this pipeline can incorporate machine learning. Planners and planning heuristics can benefit from machine learned heuristics, while system modeling can benefit from model learning. Each aspect of the motion planning pipeline comes with trade offs between computational effort and human effort. This work explores algorithms that allow motion planning algorithms and frameworks to find a compromise between the two. First, a framework for learning heuristics for sampling-based planners is presented. The efficacy of the framework depends on human designed features and policy architecture. Next, a framework for learning system models is presented that incorporates human knowledge as constraints. The amount of human effort can be modulated by the quality of the constraints given. Lastly, semi-automatic constraint generation is explored to enable a larger range of trade-offs between human expert constraint generation and data driven constraint generation. We apply these techniques and show results in a variety of robotic systems.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Electrical & Systems Engineering

First Advisor

Alejandro Ribeiro

Keywords

Machine Learning, Motion Planning

Subject Categories

Computer Sciences | Robotics

MACHINE LEARNING FOR ROBOT MOTION PLANNING

Clark Zhang

A DISSERTATION

in

Electrical and Systems Engineering

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

Supervisor of Dissertation

Alejandro Ribeiro, Professor, Electrical and Systems Engineering

Graduate Group Chairperson

Victor Preciado, Associate Professor of Electrical and Systems Engineering

Dissertation Committee

Pratik Chaudhari, Assistant Professor, Electrical and Systems Engineering

Dinesh Jayaraman, Assistant Professor, Computer and Information Science

Szymon Jakubczak, Ph.D., Computer Science

ACKNOWLEDGEMENT

The road to this dissertation has had many twists and turns, and I am grateful for everyone who has supported me along the way. First, I would like to thank my advisor, Alejandro Ribeiro, for his wisdom and support over the years. He has encouraged my research even when things seemed tough. This dissertation could not have been written without him.

I would also like to thank my committee, Pratik Chaudhari, Dinesh Jayaraman, and Szymon Jakubczak for their advice and insight. They have taken time from their busy schedules to provide guidance in my research and the dissertation process.

I would also like to thank all my fellow collaborators that worked with me. Through conversations, conferences, meals, and more, they have made my time at Penn more enjoyable and productive. I would particularly like to thank Arbaaz Khan, Jinwook Huh, Heejin Jeong, Bhoram Lee, and Santiago Paternain for their work and friendship.

Lastly, but certainly not least, I would like to thank my family and friends for their love and support. My Ph.D. would not be possible without them.

ABSTRACT

MACHINE LEARNING FOR ROBOT MOTION PLANNING

Clark Zhang

Alejandro Ribeiro

Robot motion planning is a field that encompasses many different problems and algorithms. From the traditional piano mover's problem to more complicated kinodynamic planning problems, motion planning requires a broad breadth of human expertise and time to design well functioning algorithms. A traditional motion planning pipeline consists of modeling a system and then designing a planner and planning heuristics. Each part of this pipeline can incorporate machine learning. Planners and planning heuristics can benefit from machine learned heuristics, while system modeling can benefit from model learning. Each aspect of the motion planning pipeline comes with trade offs between computational effort and human effort. This work explores algorithms that allow motion planning algorithms and frameworks to find a compromise between the two. First, a framework for learning heuristics for sampling-based planners is presented. The efficacy of the framework depends on human designed features and policy architecture. Next, a framework for learning system models is presented that incorporates human knowledge as constraints. The amount of human effort can be modulated by the quality of the constraints given. Lastly, semi-automatic constraint generation is explored to enable a larger range of trade-offs between human expert constraint generation and data driven constraint generation. We apply these techniques and show results in a variety of robotic systems.

CONTENTS

1 INTRODUCTION

Robot motion planning is a technology that enables autonomous behaviors in robot systems as well as other fields. At its core, robot motion planning algorithms attempt to address the general problem of "How can we find a plan for a robotic system to achieve a goal?" These algorithms have seen success in autonomous cars [70, 142], humanoid robotics [66], autonomous underwater vehicles [11], robotic arms [166], and many other robotics systems. Additionally, robot motion planning has been applied in other fields such as video games [154] and protein folding [6]. While robot motion planning algorithms have made great strides, it is not a solved problem. The robot motion planning problems can often lead to large computation times or a compromise on solution quality. Machine learning provides many techniques that can provide heuristics to guide traditional planning problems to increase their computational efficiency or solution quality.

Robot motion planning encompasses a variety of problems. The most basic form of a motion planning problem can address finding a continuous path from one configuration state to another for a holonomic system [120]. This is known as the *Piano Mover's Problem*. It is often described as the problem a piano mover must solve to find a path for a piano to move in or out of a house. A motion planning algorithm designed to solve such problems will typically used a model of the robotic system that relates how the state of robot changes when actions are applied. This model then constrains how a search algorithm or optimization procedure can chart a motion plan to achieve its goal. Beyond the basic motion planning problem, there are many variations for different types of robotic systems or environmental scenarios. Motion planning problems can deal with deterministic systems [120, 72, 77, 27, 42] or stochastic systems with both aleatoric and epistemic uncertainty [85, 12, 140, 26]. Problems can involve finding a set of open-loop controls [72, 42] or a set of closed loop controllers [77, 26, 137]. Researchers can find the most optimal motion plan (in terms of a defined cost function) [57, 27] or simply a feasible path to save on computational time [81]. The robotic systems can be smooth and continuous [87] or contain discrete elements [54] or discontinuities [106].

Configuration Space and Task Space

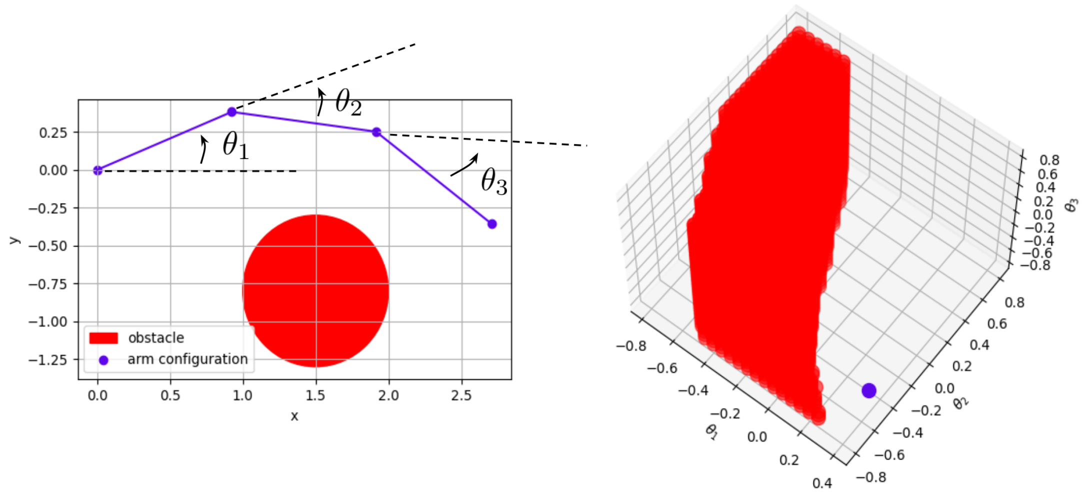


Figure 1.1: (Configuration and Task Space) Illustration of the configuration space of a 3 link planar robot arm. On the left, the task space is shown. The purple lines represent the physical links of the robot arm. The red circle is an example of a circular obstacle in the plane. On the right, the purple dot represents the configuration space coordinates of robot arm configuration shown in the left. On the right, the red shape represents the circle obstacle in configuration space coordinates.

Before discussing some of the specific problems that motion planning algorithms attempt to solve, we will define the idea of a *configuration space* and a *task space*. A particular *configuration* of a robotic system is the full description of the state of the robot relevant to planning. This is analogous to the state-space representation used in control theory [61, Chapter 1.1]. The configuration space used for defining a planning problem depends on the fidelity of planning that is required. For example, a simple model of a rocket might have a configuration state consisting of a location, a velocity, an attitude, and an angular velocity. For generating coarse motion plans, a simple model might be good enough. In a higher fidelity model of a rocket, the configuration state might consist additionally of the mass of the rocket fuel onboard and nozzle angles. A technical definition of the configuration space is also presented in [71, Chapter 4].

The *task space* for a robotic system is simply the space in which obstacles and goals are expressed. There exists a map from the configuration space to the task space, though this map may not necessarily be injective or surjective. For illustrative purposes, we will look at a 3 link planar robot arm (all three joints rotate around parallel axes). For this robot arm, the configuration space might be the space of the 3 angles of the joints of the arm. This is shown on the right in Figure 1.1. The task space can be the 2D euclidean plane in which the robot arm moves. An obstacle, such as the circle on the left in Figure 1.1 can be expressed as a set of 2D coordinates in the task space. This obstacle can be translated into a configuration space obstacle by mapping which points in configuration space collide with the obstacle.

1.1 EXAMPLES OF ROBOT MOTION PLANNING PROBLEMS

We are now ready to look at some concrete examples of motion planning problems.

Example 1.1. *The basic Piano Mover's Problem consists of a start, x_{start}, and goal, x_{goal} state both defined in the configuration space, $\mathcal{X}$. It also includes configuration space obstacles, C_{obs}, which is a set of configuration states for which the piano would be in collision with something in* *the house. The goal is to find a continuous path* $x(t) : [0, 1] \to \mathcal{X}$ *that avoids the obstacles and travels from the start to the goal that minimizes some cost. The cost can be something as simple as the length of path.*

$$\begin{aligned} \min_{x(t)} \quad & c(x(t)) \\ \textit{s.t.} \quad & x(0) = x_{start}, x(1) = x_{goal} \\ & x(t) \notin C_{obs}, \forall t \end{aligned} \tag{1.1}$$

Example 1.2. *A common problem for robot arm manipulation will be to guide the robot arm through a cluttered environment such that its end-effector will be at some pose. The robot arm starts in a configuration* $x_{start} \in \mathbb{R}^n$ *and the goal is expressed in the task space* $y_{goal} \in SE(3)$ *with a mapping from configuration to task space* $T(x) : \mathbb{R}^n \to SE(3)$.*Similar to the previous example, there exists configuration space obstacles,* C_{obs}.

$$\begin{aligned} \min_{x(t)} \quad & c(x(t)) \\ \textit{s.t.} \quad & x(0) = x_{start}, T(x(1)) = y_{goal} \\ & x(t) \notin C_{obs}, \forall t \end{aligned} \tag{1.2}$$

Example 1.3. *For kinodynamic systems we might look at solving for a sequence of actions rather than a continuous path. Given a discrete system model which involves elements of the configuration space* $x_t \in X$ *and the control space* $u_t \in U$

$$x_{t+1} = f(x_t, u_t), \tag{1.3}$$

we might define a problem that optimizes the controls over an allowable control set $U_{allowed}$ *as follows*

$$\begin{aligned} \min_{\{u_t\}_{t=1}^T} \quad & \sum_{t=1}^{T} c(x_t, u_t) \\ & x_t \notin C_{obs}, \quad \textit{for} \quad t = 1, \ldots T \\ & u_t \in U_{allowed} \end{aligned} \tag{1.4}$$

Example 1.4. *[85] describes an example of a probabilistic planning problem. A slight adaptation of that problem is shown here. Given a stochastic linear system model*

$$\begin{aligned} x_{t+1} &= Ax_t + Bu_t + \omega_t \\ x_0 &\sim \mathcal{N}(\hat{x}_0, P_{x_0}) \\ \omega_t &\sim \mathcal{N}(0, P_\omega), \end{aligned} \tag{1.5}$$

and a set of state space obstacles, C_{obs}, one goal can be to find a sequence of controls that obtains the minimum average cost path that has at most an ϵ chance of collision at every time-step The random variable that denotes the distribution of the state at time t is denoted as X_t.

$$\begin{aligned} \min_{\{u_t\}_{t=1}^T} \quad & \sum_{t=1}^{T} \mathbb{E}[c(X_t, u_t)] \\ & P(X_t \in C_{obs}) < \epsilon, \quad \textit{for} \quad t = 1, \ldots T \end{aligned} \tag{1.6}$$

1.2 CATEGORIES OF PLANNING ALGORITHMS

We have seen a selection of motion planning problems in Section 1.1, and many more exist in literature. Because there are many types of motion planning problems, there have also been a large number of motion planning algorithms that can be roughly grouped into three categories: graph-based planners, sampling-based planners, and optimization-based planners. Each of these categories are suited to different types of motion planning problems. There is not one algorithm that is superior to all others. Instead, each algorithm takes advantage of certain aspects of the problem they are designed to solve.

1.2.1 Graph-based Planners

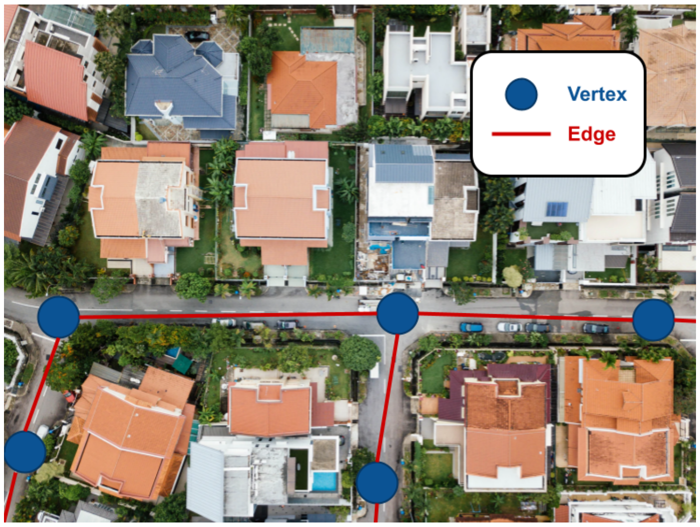


Figure 1.2: **(Motion Planning Graph)** Illustration of how a graph can model vehicle movement on a road.

The first category of planning algorithms can be described as graph-based planners. At their core, these algorithms take as input a graph $G = (V, E)$ with a set of vertices V and edges E, and perform a graph search. This type of motion planning algorithm may have been the earliest to be developed, starting with what is now known as *Dijkstra's algorithm* [27], published in 1959. The vertices of the graph, V, represent configurations in the configuration space while the edges, E, describe the cost of moving between them. For example, a car traveling on a road may be described with a set of vertices that represent a set of locations (longitude and latitude) and edges that describe the distance between those locations. In this graph context, many motion planning problems can now be described as a search for the minimum cost path from a start vertex to a goal vertex.

Graph Search

The search graph of an environment and robotic system contains a lot of the intricacies of the problem itself. Whether a system has dynamics or is holonomic will be realized in how vertices are connected as well as the weight of the edges. The obstacles in the environment are also represented by whether or not two vertices may be connected as well as the weight of connected edges. With a lot of the system complexity hidden in the graph itself, the algorithms can focus on searching the graph. Dijkstra's algorithm [27], finds the shortest path from a start vertex to a goal vertex by keeping track of a set of currently *expanded* vertices. Initially, the list contains only the start vertex. The search proceeds in iterations, and at each iteration, the set of all directly connected vertices to the *expanded* set is considered, and the one with the lowest total path cost is added to the list. When a vertex is added to the expanded list, the cost from the start vertex to that vertex is recorded as well as which parent vertex it came from. Thus, all vertices in the *expanded* set are guaranteed to have a lower cost from the start vertex than any vertex not in the expanded set. Additionally, the recorded cost for each expanded vertex is the lowest cost to reach it. The algorithm ends when the goal vertex is added to the expanded set.

An improvement to Dijkstra's algorithm is the A* algorithm [42]. A* follows the same principles of Dijkstra's algorithm with one difference. Instead of choosing the lowest cost (from the start) vertex to expand, the vertex with the lowest *cost* + *heuristic* is expanded. The heuristic is an estimate of the cost-to-go (cost from the vertex to the goal). Thus, *cost+heuristic* is an underestimate of the total cost of an optimal path from the start to goal containing the vertex. A* maintains the same guarantee of finding the minimum cost path, provided the heuristic is *admissible* (it never overestimates the true cost) and *consistent* (obeys the triangle inequality). The introduction of the heuristic can speed up the computational time of the graph search as the heuristic can guide the search towards more promising vertices first.

There are many different improvements to A* for different problems or systems with specific structures. One variation of the common motion planning problem is being able to provide a feasible path at any point during the computation. This is referred to as *anytime planning* and is useful in the case of robotics as there may be instances a feasible path is good enough. It is more important to execute a reasonable path than to spend extra computational effort finding the most optimal path. An algorithm to solve this problem is given by ARA* [81]. This algorithm works by running A* with different inflated heuristics. Inflating the heuristics gives more weight to it and can find a path faster as it could examine less vertices during the search. However, arbitrary inflation can cause the heuristic to no longer be admissible, which means the solution found might not be optimal. ARA* performs several searches by first using a high inflation value to find a possibly highly sub-optimal path but in a short amount of time. It then performs subsequent searches with lower and lower inflation values to find better paths. In this way, the algorithm can be stopped at (almost) anytime and provide a feasible solution.

Another variation of simply finding the minimum cost path is to find the minimum cost path given information about solutions of very similar graphs. The idea is when a robot is moving around in a dynamic environment, it may be solving a series of very similar path planning problems when small parts of the environment change. This idea is presented by LPA* (Life Long Planning A*) [62]. By saving the cost-to-go, and propagating changes in the graph only as needed, it can greatly reduce computational time on repeated runs of the algorithm. The ideas of anytime search as well as lifelong planning are combined in AD* [80].

Both Theta* [24] and Jump Point Search [41] work in very specific environments. They both assume uniform grids with uniform costs. This has applications in 2D navigation for holonomic ground vehicles as well as in video games By leveraging knowledge about the highly structured grid environment, both Theta* and Jump Point Search can provide better or faster solutions.

The advantages of graph-based planners are the strong theoretical guarantees that come with them. For many of the planners, optimal paths are guaranteed in finite time (for finite graphs). Even the non-optimal graph-based planners such as ARA* come with strong theoretical guarantees about the solution quality. There are, however, a few drawbacks to graph-based planners. When the planning domain is naturally in a continuous domain (which includes a large number of robot motion planning problems), using a graph-based planner necessitates discretizing the space. This discretization process can introduce sub-optimality in the continuous space, or turn feasible continuous problems into infeasible discrete graph search problems. For example, if a robot needs to fit into a tight entrance, but the discretization process is too coarse to contain a vertex in the small entrance, the graph problem will be unsolvable despite the fact that the underlying problem has a solution. Additionally, graph-based methods can suffer from the *curse of dimensionality* as uniform discretization of a configuration space scales exponentially with the number of dimensions.

1.2.2 Sampling-based Planners

Sampling-based planners are randomized algorithms that sample a continuous configuration space to obtain a discrete representation of it. Many popular sampling-based planners attempt to address the problem of both coarse discretization and the curse of dimensionality that plagues graph-based planners. In the case of a popular sampling-based

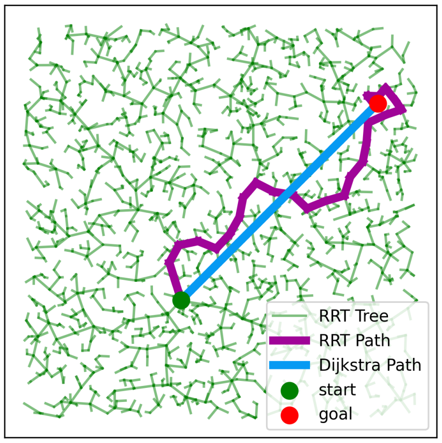


Figure 1.3: (Sampling-based vs Graph-based Planners) Comparison between paths found by Rapidly-Exploring Random Trees (RRT) and Dijkstra Graph Search on an empty 2D map with a holonomic robot.

planner, Rapidly-Exploring Random Trees (RRT) [73], the algorithm iteratively samples and searches the continuous configuration space, building a tree of connected configurations, until a path is found. Each new sample attempts to connect to the closest configuration in the existing tree. This allows the planner to operate in higher dimensions as the sampling allows for possibly more efficient exploration of the configuration space (new samples have a Voronoi Bias [73]). The planning process is not limited to exhaustively search through all the nodes of a chosen discretization level. Additionally, small areas but important areas of the configuration space which might be missed with a coarse predetermined discretization will be sampled with high probability as the number of samples increase. More concrete details about RRT will be given later in Section 2.1.1. Other popular sampling-based planners include Probabilistic Roadmap (PRM) [58] and Expansive-Space Trees (EST) [47]. Both RRT and EST sample and grow trees for one time path planning. PRM solves a slightly different problem of planning repeatedly in the same environment. It samples the configuration space, and tries to connect nearby configuration space samples if possible into a graph, and later performs graph search when a path in the environment is required.

PRM can be seen as a way to discretize a continuous space using random sampling and then running graph search algorithms. Because of the offline nature of PRM's sampling method, it does not address the issue of sampling coarseness.

While sampling-based planners do not have the strong theoretical guarantees of graph-based planners, there still exist some weaker guarantees. In particular, RRT is known to be *probabilistically complete* [65] in the case of holonomic systems. This is not always true for kinodynamic systems [67]. Probabilistic completeness is the notion that the probability of finding a feasible path converges to 1 as the number of samples goes to infinity. Note that this does not provide any guarantee on what happens in a finite amount of time.

The RRT algorithm has been modified by many researchers to solve different problems. RRT motion plans can be highly sub-optimal (see Figure 1.3), which the RRT* algorithm [57] attempts to correct. RRT* will *rewire* the tree, changing edges between existing tree nodes if new samples of the configuration space can provide more optimal paths. This allows RRT* to refine paths over time, and the algorithm can be run for as long as desired to obtain more cost efficient paths (in a similar method to ARA* [81]). It was also shown that as the number of samples tends towards infinity, the probability of RRT* finding the most optimal path converges almost surely to 1 [57, Theorem 38]. The asymptotic optimality of RRT* comes at the cost of being more computationally expensive as each iteration must find all the nearest neighbors in a radius (rather than the single nearest neighbor) and check if they must be rewired. Additionally, the rewiring process may not be easily extendable to non-holonomic systems. There exists some work to extend RRT* to some kinodynamic systems [78, 148, 100] but they require strong assumptions about properties of the system such as Chow's condition or linearity or lose the hard dynamics constraints between tree vertices. Another variation of RRT, denoted as Chance-Constrained RRT (CC-RRT) [85], looks at extending the piano mover's problem to linear systems with additive noise. It is able to reason about collision probabilities and choose best cost paths subject to chance constraints by propagating process noise throughout the tree.

Sampling-based planners are typically more efficient in high dimensional spaces and do not require explicit discretization schemes. However, they lose many of the strong theoretical guarantees that graph-based planners have. In practice, sampling-based planners may be used to find an initial solution and then smoothed or optimized over later (perhaps by using a method that will be discussed in Section 1.2.3).

1.2.3 Optimization-based Planners

Another major class of motion planning algorithms are those that use optimization solvers for specific types of optimization problems. The motion planning problem for a system is formulated in some canonical optimization form and solved using specific or generic optimization solvers. Quadratic Programs are solved in [87], Sequential Quadratic Programs are utilized in [117], and Gradient descent is employed in [166]. A specific solver, iLQR, is developed by [77] for nonlinear objectives with specific nonlinear dynamics constraints. Many motion planning problems can be written as generic nonlinear programs [117, 166, 77, 56].

The benefits of optimization-based planners are that they can find locally optimal solutions in high dimensional, continuous configuration spaces with complex system dynamics and without any discretization error or sampling artifacts. Additionally, they typically can run an order of magnitude faster than competing sampling-based planners in high dimensions [166]. The main disadvantage of optimization-based planners is that both general solvers and specific solvers do not guarantee convergence to globally optimal solution for generic nonlinear problems. In practice, many motion planning problems solved with nonlinear solvers must have a good initial guess at the solution to converge to a reasonable motion plan. While this can be alleviated by restarting the optimization problem with different initial conditions [166], problems with highly nonlinear dynamics or very high dimensionality can require an exponentially large number of restarts.

Recently, the lines between the field of Control Theory and Motion Planning have become more blurred. In older research works, when computational power was more limited with respect to the system and planning algorithms were less efficient, planning was seen as providing a one time input into a lower level controller [45, 87, 45]. Recently, planning algorithms are more often to be run in a closed loop [82, 70, 149]. While lower level controllers to track joint angles, velocities, robot attitude, etc. may still exist, planning algorithms are run at higher frequencies than before. A robot system consisting of a fast closed loop planner and an even faster low level controller stack appear similar to hierarchical control stacks [113] or cascaded controllers [32, 75]. Many of these optimization-based planners can be seen as doing Model Predictive Control (MPC) [14] when run in a closed loop.

1.3 MODEL LEARNING

All the algorithms discussed in Section 1.2 require a model of the robotic system and its environment. These models are commonly referred to as the *system model* or *dynamics model*. Graph-based algorithms utilize these models to build the search graph itself. Sampling-based algorithms use these models to check for valid samples and connections between them. Optimization-based planning use the models as hard constraints or as soft penalties in the objective function. Thus, the accuracy of a model is important to the performance of a planning algorithm. If there is a large mismatch between the real system and the model of the system, the computed motion plans can be useless. While certain algorithms try to account for model mismatch [44, 126], these methods cater to the worst case error and can lead to overly conservative motion plans or simply fail if the mismatch is too high. Thus, an accurate system model is still necessary for performance.

In *control theory*, the sub-field of System Identification [59] deals with using collected data from a system to estimate its parameters. For linear systems, there exist results on convergence along with the conditions in which convergence is possible [101, 95, 104]. For generic nonlinear systems, such strong theoretical results do not exist, though there has been work on nonlinear systems with specific structures [116].

In robotics, a variety of methods have been tested to learn nonlinear dynamics models from data: Gaussian Processes Regression [94, 109, 26], Locally Weighted Projection Regression [114], Gaussian Mixture Models [76], and Neural Networks [76, 91, 97, 159, 160]. The work by the robotics community has applied model learning to robot-arm trajectory tracking [94], multi-legged robot locomotion [91], and many other tasks.

1.4 REINFORCEMENT LEARNING AND PLANNING

A field that is closely related to motion planning is Reinforcement Learning (RL) [131]. Reinforcement Learning deals with the problem of selecting actions for a system to maximize the rewards (or minimize the costs). The problem setup is very similar to many motion planning problems, except for the fact that in Reinforcement Learning, usually the model and possibly the reward/cost function is unknown. There are two categories of Reinforcement Learning algorithms, model-based and model-free. A more rigorous introduction into this topic is given in Section 2.1.3.

Model-based Reinforcement Learning algorithms function much in the same way as learning a dynamics model and then planning with it. Many modern day model-based

Reinforcement Learning algorithms are based off of the Dyna [130] framework, which iterates between learning a dynamics model and using the model to plan. More modern approaches such as [76, 31] function in similar ways with neural network models and policies operating on more complicated state domains such as images. The DynaQ algorithm interleaves model learning with learning the action-value function through Bellman updates. Dijkstra's algorithm can be seen as performing Bellman updates with a specific known structure [127]. Thus, the Dyna framework is analogous to interleaving traditional System Identification with motion planning.

Model-free Reinforcement Learning algorithms decide how to actuate a system by trying actions many times and observing rewards to update a policy directly (or value function from which a policy is derived). Model free methods such as Q-learning [146] and REINFORCE [132, 102] as well as modern variations of them [39, 118, 119, 103] have been applied to robotic system simulations including ones from a suite of benchmarks know as the OpenAI gym [17]. When applying to robotic systems, a common strategy is to use a simulation of the real robot and apply these algorithms in simulation. Then, the control policy may be put on a real system, perhaps with some additional modifications [79, 98]. This scenario of using a simulation to apply Reinforcement Learning algorithms can be seen in the lens of a traditional motion planning algorithm. In this case, the system model is the simulator, which is used with a motion planning algorithm to generate closed loop controls. Just as in traditional motion planning, the difference between the model of the system (the simulation) and the real world can cause many problems. This is well known in the Reinforcement Learning community as the *simulation-reality gap* or *reality gap* [98, 134, 163].

1.5 GOALS OF THIS WORK

This work utilizes machine learning techniques to address some of the shortcomings of classical planning algorithms. First, it will address the computational time of planning in complicated domains. The basic *Piano Mover's Problem* is demonstrated to be p-space complete [120, Chapter 11] and the numerous changes in the problem can be even more difficult to solve. Obtaining reasonable motion plans for problems can be accelerated by relying on specific problem structure. This structure can come in the form of data. Learning heuristics for common motion planning problems from data can provide a way to increase the efficiency of motion planning algorithms in the face of p-space complete problems. This will be discussed in Chapter 2.

Another aspect of this work will address the problem of model learning. The model of a system is crucial to the functionality of motion planning algorithms. This work proposes using machine learning techniques to combine both data and human intuition to obtain models can provide better results during the planning process. This will be discussed in Chapters 3 and 4.

2 LEARNING PLANNING HEURISTICS

This chapter will discuss using machine learning techniques to speed up the computational time of motion planning algorithms. We will focus on developing methods to learn heuristics for sampling-based planners. To aid in the discussion, the RRT [73] algorithm will be used as the prototypical sampling-based planner, but the method works for all sampling-based planners. We cover various existing heuristics in sampling-based planners, and provide a unifying view in the form of rejection sampling. A reinforcement learning algorithm is used to optimize the rejection sampler to provide heuristics that suit a set of environments. It utilizes information from past searches in similar environments to generate better heuristics in novel environments, thus reducing overall computational cost.

Heuristics play an important role in motion planning algorithms as they guide the exploration of the configuration space. Without a heuristic, a brute force search will be necessary that can result in a large increase in the computation time of the motion plans. However, for complex problems, general purpose heuristics such as L1 or L2 distance can still be relatively uninformative. When a robot operates in a set of similar environments, say office spaces or warehouses, we can learn heuristics that are unique to that type of environment. The problem discussed here is sometimes known as the *Experienced Piano Mover's Problem*. The idea being, unlike the regular Piano Mover's Problem, the experienced piano mover has seen many houses and many pianos and has a good idea of how to move pianos in a new house they have never seen. They have developed a heuristic for moving pianos around houses. The input into a motion planning algorithm for this problem is not

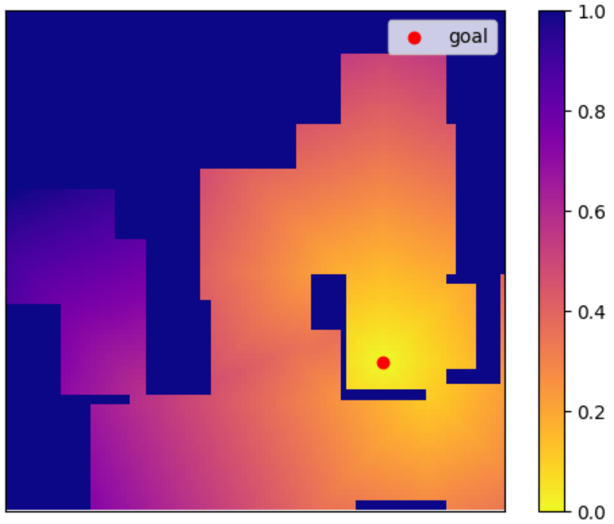


Figure 2.1: (Cost-to-go) Example of a cost-to-go function on a 2D grid world with rooms. The colorbar shows the cost-to-go normalized between to the range [0, 1].

simply a description of the environment and a desired start and goal, but also a set of previous problems and the solutions that were found. For a class of problems, it may be possible to use previous experience finding motions plans to guide and improve future instances of the motion planning problem.

The best heuristic for any motion planning problem environment is the true *cost-to-go*, which is a function $V(x) : \mathcal{X}_{\mathtt{config}} \rightarrow \mathbb{R}$ that maps states in the configuration space to the true optimal cost to reach the goal from that state (in Reinforcement Learning, this is also called the *value function*). An example of such a function is shown in Figure 2.1. Finding the cost-to-go is as difficult as solving the planning problem. Knowing the cost-to-go already yields the solution to a motion planning problem. For each state, the optimal action will be the one that minimizes the cost-to-go at the next state. Thus the goal of heuristic learning is to try to estimate the cost-to-go while taking into account a description of the environment and the goal state. A example of a heuristic that may work well in an office environments can be one that encourages exploration near doors and the corners of hallways. Thus even

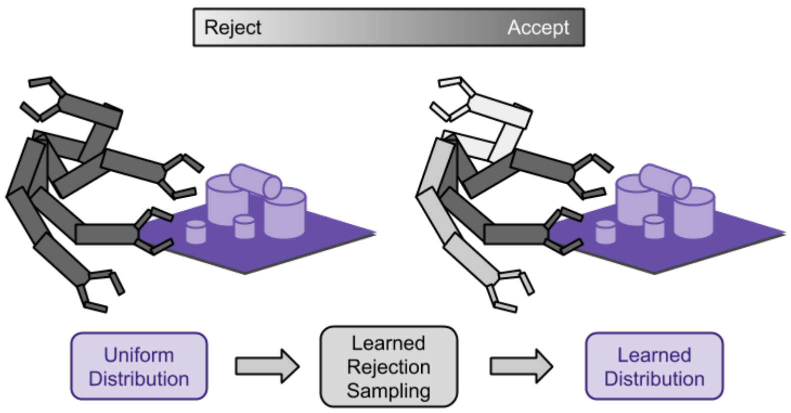


Figure 2.2: **(Rejection Sampling)** An example of a learned distribution for the task of a robot arm reaching for various objects on a tabletop. On the left, samples from a uniform distribution of the configuration space are displayed. Some of which may be rejected by a learned rejection sampling policy to form a new learned distribution over the configuration space.

when confronted with a completely new office space, a robot might efficiently plan in it knowning that good areas to explore during planning are doors and corners.

In sampling-based planners, the sampling distribution can be seen as a heuristic [158]. Traditionally, sampling-based planners draw random state samples from a uniform distribution (many times with a slight goal bias). However, for many classes of environments, a different probability distribution over the state space can speed up planning times. For example, in environments with sparse obstacles, it can be useful to heavily bias the samples towards the goal region as the path to the goal will be relatively straight. The natural questions to ask are "How heavily should the goal be biased?" or more generally "What is the best probability distribution to draw out of?" In previous literature, many researchers have found good heuristics [155, 122] to modify the probability distributions for specific environments. However, these heuristics do not work generally and may not apply to new environment types. In fact, a heuristic can increase the planning time dramatically if it is unsuited to the problem at hand.

In this work, we present a systematic way to generate effective probability distributions automatically for different types of environments. The first issue encountered is how to choose a good representation for probability distributions. The sampling distributions can be very complicated in shape and may not easily be representable as common distributions such as Gaussians or Mixtures of Gaussians. Instead, the distribution is represented with rejection sampling, a powerful method that can implicitly model intricate distributions. The process of accepting or rejecting samples is formulated as a Markov Decision Process. This way, policy gradient methods from traditional Reinforcement Learning literature can be used to optimize the sampling distribution for any planning costs such as the number of collision checks or the planning tree size. The method presented will use past searches in similar environments to learn the characteristics of good planning distributions. Then, the rejection sampling model will be applied to new instances of the environment. This process is shown pictorially in Figure 2.2. Section 2.1 will discuss the necessary background required for the learning method, Section 2.2 will formalize the problem of the experienced piano mover, Section 2.3 will introduce the learning method, and Section 2.4 will present experimental results and analysis.

2.1 BACKGROUND

Before discussing our method, we will present existing and relevant information and work. Section 2.1.1 will go over in detail the RRT algorithm that will be used throughout for explanation. Section 2.1.2 will discuss existing methods for heuristic learning. Section 2.1.3 will formally introduce the framework and algorithms of Reinforcement Learning as they are a crucial part of the methodology.

2.1.1 Rapidly-Exploring Random Trees

The RRT algorithm was first introduced in 1998 [73] and quickly gained popularity for its simplicity, effectiveness, and probabilistic completeness. The base algorithm is detailed in Algorithms 1 and 2.

Algorithm 1 RRT

```
1: procedure RRT($x_{start}, x_{goal}$)
2:     Initialize Empty(Directed) Graph $G = (V, E)$
3:     $V \leftarrow V \cup \{x_{start}\}$
4:     while $x_{goal} \notin V$ do
5:         $x_{rand} \leftarrow$ SAMPLE()
6:         EXTEND($x_{rand}, G$)
7:     end while
8: end procedure
```

Algorithm 2 EXTEND

```
1: procedure EXTEND($x_{rand}, G$)
2:     $x_{near} \leftarrow$ NEAREST($x_{rand}, G$)
3:     $x_{new}, path \leftarrow$ STEER($x_{near}, x_{rand}$)
4:     if COLLISION_FREE($path$) then
5:         $V \leftarrow V \cup \{x_{new}\}$
6:         $E \leftarrow E \cup \{(x_{near}, x_{new})\}$
7:     end if
8: end procedure
```

x_{start} is a start state, x_{goal} is the desired goal state. The algorithm starts with a tree that contains only the start state. It then iteratively samples a random state and extends the tree towards it. The extension is performed by choosing the closest vertex in the tree to the random state and then *steering* that vertex towards it, adding a new vertex. At the end of the algorithm, the tree G will contain a path from x_{start} to x_{goal}. This path can be found simply by finding the goal vertex and recursively following the parents of each vertex back to x_{start}. RRT is dependent on several robot specific functions. For any robotic system to use RRT planning, there must exist

- SAMPLE: Draws a random state from the state space, S.
- NEAREST(x, G): Finds the node nearest to x in the graph G based on some metric, p. For differentially constrained problems, p may not be a proper metric as it can be asymmetric.
- STEER(x_0, x_1): Applies a control to the system to generate a path (usually ignoring obstacles) that goes from x_0 to x_{new}, where $p(x_0, x_1) > p(x_{new}, x_1)$ This can be a nontrivial function for many systems. This function will return x_{new}, the state the path ends with, and some representation of the path (usually a collection of waypoints) taken. For general kinodynamic systems, a generic way to create this STEER function can be to randomly sample controls and a period of time to apply that control. Then, choose the control that moves the state closest to the x_1.
- COLLISION_FREE($path$): Checks if the path is in collision with the environment.

A common modification to the RRT algorithm is the RRT-Connect algorithm [65]. The RRT-Connect algorithm uses two trees, one rooted at the start and the other at the goal, and attempts to extend each one. When extended, the algorithm will attempt to connect the trees together by repeatedly extending one tree towards the other. This is detailed in Algorithms 3, 4, and 5.

Algorithm 3 RRT-CONNECT

```
1:  procedure RRT-CONNECT($x_{start}, x_{goal}$)
2:      Initialize Empty(Directed) Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$
3:      $V_1 \leftarrow V_1 \cup \{x_{start}\}$
4:      $V_2 \leftarrow V_2 \cup \{x_{goal}\}$
5:      while $x_{goal} \notin V$ do
6:          $x_{rand} \leftarrow$ SAMPLE()
7:          if EXTEND($x_{rand}$, $G_1$) $\neq$ Trapped then
8:              if CONNECT($x_{new}$, $G_2$) = Reached then
9:                  Break
10:             end if
11:         end if
12:         SWAP($G_1$, $G_2$)
13:     end while
14: end procedure
```

Algorithm 4 EXTEND (RRT-Connect)

```
1: procedure EXTEND($x_{rand}, G$)
2:     $x_{near} \leftarrow$ NEAREST($x_{rand}, G$)
3:     $x_{new}, path \leftarrow$ STEER($x_{near}, x_{rand}$)
4:     if COLLISION_FREE($path$) then
5:         $V \leftarrow V \cup \{x_{new}\}$
6:         $E \leftarrow E \cup \{(x_{near}, x_{new})\}$
7:         if $x_{new} = x_{rand}$ then
8:             Return Reached
9:         else
10:            Return Advanced
11:        end if
12:    else
13:        Return Trapped
14:    end if
15: end procedure
```

Algorithm 5 CONNECT

```
1: procedure CONNECT($x_{rand}, G$)
2:     $S \leftarrow$ EXTEND($x_{rand}, G$)
3:     while $S \neq Advanced$ do
4:         $S \leftarrow$ EXTEND($x_{rand}, G$)
5:     end while
6: end procedure
```

2.1.2 Heuristics and Heuristic Learning

Heuristic Learning in Graph-based Planners

In graph-based planning, there has been work in directly trying to learn a cost-to-go function as a heuristic. Since graph-based planners can find optimal paths, a supervised learning problem may be posed for a function approximator like a neural network to regress the cost-to-go values returned from an optimal planner [10, 23, 4]. While [10] uses the (generally inadmissible) heuristic as the sole determining factor to do a greedy search, [4] bounds the learned heuristic with an admissible heuristic and uses a traditional graph search algorithm. Similar to ARA* as discussed in Section 1.2.1, the inadmissible heuristic seems to provide large computational time improvements at the cost of optimality. The heuristic that is learned in [4] utilizes a convolutional neural network takes as input the start, goal, and a representation of all the obstacles. The heuristic is trained to regress the cost-to-go. Thus, one interpretation is that the convolutional network itself is attempting to solve the planning problem within the computational constraints of the network weights. There may exist a convolutional neural network large enough with specific weights such that all grid problems of a certain size might be fully solved by it without an explicit planning algorithm. However, the trade off here is between using a planning algorithm that is provably optimal versus a learned heuristic that may be more efficient at recognizing patterns for cost-to-go computations. [48] uses a similar approach in a realistic robot arm planning scenario in higher dimensions.

Sampling-based Planner Heuristics

While strong supervision exists for problems that can be solved with graph-based planners, more difficult problems may not have an optimal planner available to provide supervised cost-to-go values. There is no guarantee that sampling-based planners such as RRT (or the asymptotically optimal variants) will find an optimal path in any finite amount of time. In fact, it has been shown that the probability of sampling-based algorithms finding the optimal path in any finite time is zero [57] given some reasonable assumptions (optimal paths have zero-measure). Thus, the approach used to learn heuristics from data in graph-based planning algorithms are not easily translatable to sampling-based approaches.

Many researchers have used human intuition to develop heuristics that work for different sets of environments. There is a number of methods that use rejection sampling to bias the sampling distribution. [13] introduces a method to bias random samples towards obstacles for the PRM planner. For every sampled state, an addition state is generated from a Gaussian distribution around the first state. A sample is only accepted if exactly one point is in collision. [141] proposed a method to compute lower cost RRT paths. Each node in their tree is given a heuristic *quality value* that estimates how good a path passing through that node will be. Rejection sampling is used to sample points near high quality nodes. This method is mostly superseded by RRT* [57], but is a useful case of how rejection sampling has been used to improve path quality. Dynamic-Domain RRT [155] rejects samples that are too far from the tree. The idea is that drawing samples on the other side of an obstacle is wasteful since it will lead to a collision, so sampling is restricted to an area close to the tree. BallTree [122] uses a heuristic that is the opposite of Dynamic-Domain RRT and rejects samples that are too close to the tree. The idea is that many nodes in the tree are wasted in exploring areas that are close. [123] present a heuristic to improve RRT performance for differentially constrained systems by rejecting samples where the reachability region of the nearest neighbor is further from the random sample than the nearest neighbor itself, so that extending towards the sample will not actually encourage exploration. Informed RRT* [34] improves RRT* performance by restricting samples to an ellipsoid that contains all samples that could possible improve the path length after an initial path is found. This technique does not improve the speed at which the first path is found, but the speed at which the solution is further optimized. [68] improved the informed sampling technique to improve efficiency in high dimensions while [156] has extended the approach to kinodynamic systems.

There are also methods that do not utilize rejection sampling. [161] modifies random samples by moving points in the obstacle space to the nearest point in free space. The effect of this method is that small "tunnels" that are surrounded by obstacles will be sampled more frequently. As noted, this is effective for environments that have narrow passages which are particularly hard for traditional planners to solve due to the small probability of sampling within the narrow passage. [16] grows a backward tree in the task space and biases samples in the forward configuration space tree towards it. The backward task space tree can be much more easily found in manipulation tasks and can effectively guide the forward configuration space tree. [153] proposes a method to quickly compute an approximation to the medial axis of a workspace. Their goal is to generate PRM samples that are close to the medial axis, as it is a good heuristic to plan in environments with narrow tunnels. This has also been explored in [151].

Heuristic Learning in Sampling-based Planners

While the previous work has yielded good results for certain environments, they are not generally applicable. There has been some work in automating how to improve sampling for different environments. [165] introduced a method to optimize workspace sampling. The workspace is discretized and features such as visibility are computed for each discrete cell. The workspace sampling is improved using the REINFORCE algorithm [150]. This method performs well in the environment it is optimized in, but new environments can potentially have a high preprocessing cost to compute the features. In addition, discretizing the workspace may be infeasible for certain problem domains. More recently, [49] used a Conditional Variational Autoencoder [28] to learn an explicit sampling distribution for FMT* [53] by maximizing the likelihood of generating samples from previous successful motion plans or human demonstrations. Motion Planning Networks [107] also uses supervision from human demonstrations or *near-optimal* plans to learn a network that outputs a state sample which is used greedily. Neural Exploration-Exploitation Trees [20] proposes a similar method that uses supervision from previous successful plans to suggest samples which are used in combination with uniform random samples.

2.1.3 Reinforcement Learning

This section will provide a quick introduction into Reinforcement Learning (RL), which is a vital component in our method of heuristic learning. For a more detailed introduction, [131] is a good resource. Our method will formulate a rejection sampling process as a Markov Decision Process (MDP) in Section 2.3.2 and provide the Reinforcement Learning algorithm to solve it in Section 2.3.3. The relevant background to what will be discussed is presented here.

Markov Decision Processes

Reinforcement Learning operates in the framework of a MDP. A MDP is a general framework that can model fully observable systems with control inputs. Formally, a discrete MDP consists of a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r)$ where

1. $\mathcal{S}$ is a set of states that the system can be in. This is analogous to the configuration space defined in Chapter 1.
2. $\mathcal{A}$ is a set of actions the system may take.

3. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition function that is denoted as $\mathcal{P}(s_{t+1}|s_t, a_t)$ where $s_t, s_{t+1} \in \mathcal{S}$ and $a_t \in \mathcal{A}$. This function represents the probability of taking an action a_t at state s_t and ending up in state s_{t+1}. This should be a valid probability distribution such that $\sum_{s_{t+1}} \mathcal{P}(s_{t+1}|s_t, a_t) = 1, \forall (s_t, a_t)$.

4. $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function that represents the immediate *goodness* of taking an action at a specific state.

Often, the goal of *solving* a MDP is to find a *policy* to maximize the (possibly discounted) sum of rewards from the initial state. A policy is a function that maps a state to a desired action or distribution of actions. It is often denoted as $\pi(s), s \in \mathcal{S}$ for a deterministic policy or $\pi(a|s), a \in \mathcal{A}, s \in \mathcal{S}$ for a stochastic policy. Concretely, a common objective of solving a MDP in a finite time horizon setting is to find π to optimize the following

$$\max_{\pi} \mathbb{E}_{\{S_t, A_t\}_{t=0}^{T}} [\sum_{t=0}^{T} r(S_t, A_t)] \tag{2.1}$$

where S_t is a random variable that takes on values in $\mathcal{S}$ which represents the distribution of states at time t when following a policy π. Similarly, A_t is a random variable that takes on values in $\mathcal{A}$ which represents the distribution of actions at time t when following policy π. Even in the case of a deterministic policy, A_t can still be a distribution if the state transition function $\mathcal{P}$ is stochastic. The expectation is taken over all the states and actions at all times. Thus, (2.1) is a problem to find π to maximize the average sum of rewards over a finite time horizon.

Another common objective is to maximize the discounted sum of rewards over a infinite time horizon

$$\max_{\pi} \mathbb{E}_{\{S_t, A_t\}_{t=0}^{\infty}} [\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t)] \tag{2.2}$$

where $\gamma \in [0, 1)$ is the discount factor. γ plays two roles: 1) it determines how much more we value short term rewards over long term rewards and 2) it allows the infinite sum to converge to a finite number (provided that the rewards are bounded) so policies can be compared by a finite number. To see the latter point, consider a MDP where one action at any state will always give a reward of 1, and another action will always give a reward of 2. Without the discount factor, a policy that always chooses the action that obtains a reward of 1 would look just as appealing as any other policy. We would like to be able to formulate a problem where the action that chooses a reward of 2 is more preferable. Depending on the literature, the discount factor, γ, is sometimes included in the definition of the MDP tuple. To illustrate how a MDP might be used to model a system, we will give an example.

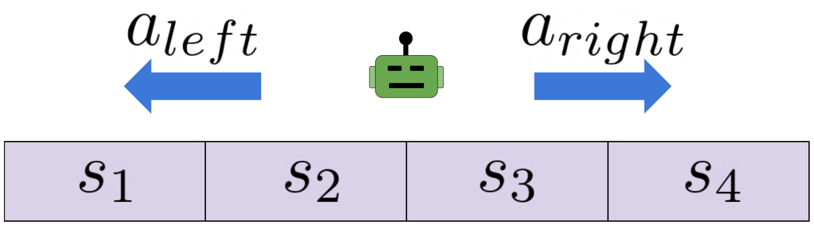


Figure 2.3: (MDP Example) A simple example of a Markov Decision Process with a robot traveling on a line.

Example 2.1. *We look at a lazy robot traveling on a line as shown in Figure 2.3. The state space* $\mathcal{S} = \{\ldots, s_1, s_2, s_3, s_4, \ldots\}$ *consists of the discrete locations that the robot can exist in. The action space* $\mathcal{A} = a_{\text{left}}, a_{\text{right}}$ *consists of the two actions that can be commanded to the robot. Since the* *robot is lazy, it will only do what it is commanded half the time, the other half it stays put. The transition function $\mathcal{P}$ is defined as*

$$\mathcal{P}(s'|s,a) = \begin{cases} 0.5, & \textit{if} \quad s = s' \\ 0.5, & \textit{if} \quad s \textit{ is to the left of } s' \textit{ and } a = a_{\mathrm{right}} \\ 0.5, & \textit{if} \quad s \textit{ is to the right of } s' \textit{ and } a = a_{\mathrm{left}} \\ 0, & \mathrm{otherwise} \end{cases} \tag{2.3}$$

We can also define a reward function that rewards the robot if it is at state s_3.

$$r(s,a) = \begin{cases} 1, & \textit{if} \quad s = s_3 \\ 0, \mathrm{otherwise} \end{cases} \tag{2.4}$$

Note that there are multiple ways of defining a MDP in literature. Some use a reward function that takes into account the state at the next timestep as well as the state and action at the current timestep. Different definitions can be notationally useful in specific circumstances.

Reinforcement Learning Algorithms

We have discussed the framework in which Reinforcement Learning algorithms are defined in. If all the ingredients in the discrete MDP tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r)$, are known, then there exists dynamic programming methods to find an optimal policy [131, Chapter 4]. Reinforcement Learning deals with how to solve the MDP when certain ingredients are unknown – most commonly the state transition function $\mathcal{P}$ and reward function r. While the exact functions are unknown, it is possible to sample trajectories from the MDP. For example, the transition function may be unknown for a novel robotic platform, but controls can be applied and the resulting states can be recorded. In these cases there are different categories of algorithms to solve the MDP. There exist model-based methods that attempt to learn $\mathcal{P}$ and r from data and use algorithms similar to dynamic programming to solve it [130, 91, 55]. There are different model-free algorithms as well, ranging from value-based methods [146, 138, 125] to policy gradient methods [150, 132, 38, 124] to actor-critic methods [152, 63, 103].

For sake of brevity, we will briefly describe only policy gradient methods as they are used in our heuristic learning method. The original policy gradient method, known as REINFORCE, was introduced by Williams [150] and later extended into function approximators by Sutton [132]. The idea behind policy gradient methods is to directly take the gradient of the Reinforcement Learning objective defined in (2.1) and (2.2) with respect to the parameters of a policy and apply gradient ascent to maximize it. The original policy gradient methods utilize stochastic policies, $\pi(a|s)$

We look at the finite horizon case with a policy π_θ that is parameterized by a vector of parameters θ, where the objective is

$$V^{\pi_\theta}(s_0) = \mathbb{E}_{\{S_t,A_t\}_{t=0}^{T}}[\sum_{t=0}^{T} r(S_t, A_t)] \tag{2.5}$$

which can be rewritten as

$$V^{\pi_\theta}(s_0) = \sum_{\tau} P_{\pi_\theta}(\tau) R(\tau) \tag{2.6}$$

where $\tau = s_0, a_0, s_1, a_1, \ldots$ is a trajectory sample of states and actions. $R(\tau) = \sum_{t=0}^{T} r(s_t, a_t)$ is the total discounted sum of rewards of that trajectory, and $P_{\pi_\theta}(\tau)$ is the probability of that trajectory sample happening under a policy π_θ. The expected sum of rewards is the same as the expected trajectory reward. We can now look at computing the gradient

$$\begin{aligned}\nabla_\theta V^{\pi_\theta}(s_0) &= \nabla \sum_\tau P_{\pi_\theta}(\tau)R(\tau) \\ &= \sum_\tau R(\tau)\nabla_\theta P_{\pi_\theta}(\tau) \\ &= \sum_\tau R(\tau)P_{\pi_\theta}(\tau)\nabla_\theta \log(P_{\pi_\theta}(\tau))\end{aligned} \tag{2.7}$$

where the expected discounted sum of rewards of a single trajectory sample does not depend on the policy parameters, thus it is constant with respect to θ. Additionally, the last line comes from the fact that $\nabla_\theta \log(f(\theta)) = \frac{1}{f_\theta}\nabla_\theta f(\theta)$. We can write the gradient of the log probability of a trajectory sample as

$$\begin{aligned}\nabla_\theta \log P_{\pi_\theta}(\tau) &= \nabla_\theta \log(\Pi_{t=0}^{T}\pi_\theta(a_t|s_t)\mathcal{P}(s_{t+1}|s_t, a_t)) \\ &= \nabla_\theta(\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{T} \log \mathcal{P}(s_{t+1}|s_t, a_t)) \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\end{aligned} \tag{2.8}$$

where $\mathcal{P}$ is the state transition probabilities and is constant with respect to policy parameters. Thus, the entire policy gradient can be written as

$$\begin{aligned}\nabla_\theta V^{\pi_\theta}(s_0) &= \sum_\tau R(\tau)P_{\pi_\theta}(\tau)\nabla_\theta \log(P_{\pi_\theta}(\tau)) \\ &= \mathbb{E}_{P_{\pi_\theta}(\tau)}[R(\tau)\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)].\end{aligned} \tag{2.9}$$

(2.9) is known as the Policy Gradient Theorem. It is useful, because the expectation on the right hand side can be estimated by running multiple trajectories and averaging the expression in the inside of the expectation. The gradient of the log policy can be computed analytically for most parameterized policies. Often, a baseline that is policy independent (but can be state dependent), $b(s)$, can be introduced to decrease the variance of the policy gradient [131, Chapter 13.4].

$$\nabla_\theta V^{\pi_\theta}(s_0) = \mathbb{E}_{P_{\pi_\theta}(\tau)}[(R(\tau) - b(\tau)) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)]. \tag{2.10}$$

With the ability to compute the gradient of the Reinforcement Learning objective, a stochastic gradient ascent algorithm can be formulated that iterates between 1) running trajectories on the MDP to gather samples to estimate (2.10) and 2) computing the gradient and applying gradient ascent to the policy parameters.

Policy gradient methods may not find the optimal policy, unlike many value-based methods. However, the advantages are that is that they converge to a local optimum speedily and have an explicit representation of the policy which can be fast to compute.

2.2 AN EXPERIENCED PIANO MOVER'S PROBLEM

Now that the appropriate background has been given, we mathematically define the statement of the *Experienced Piano Mover's Problem* that we seek to solve. We would like to reduce the computational cost of sampling-based planners in certain types of environments by modifying the sampling distributions. For clarity, let us consider planning trajectories for a robotic arm in typical tabletop environments.

Following the notation from [57], a configuration space for a planning problem is denoted as X. For a given environment, let X_{obs} denote the obstacle space, a subset of X that the robot can not move in. Thus a map is uniquely defined by its X_{obs}. A specific environment type, E, is a probability distribution over possible obstacle spaces, X_{obs}. For a 7DOF robotic arm, X is the 7 dimensional configuration space, and E will assign higher probability to environments that look like scattered objects on a table.

Let $\mathbf{Y}_k \sim \mu_k$ be a sequence of Random Variables that represents the i^{th} random sample of the state space drawn during the planning process (Note that the random variables do not need to be identical and can change during the planning process). In standard sampling-based planners, $\mathbf{Y}_k$ are independent and identically distributed. Now given a specific map, X_{obs}, and a sequence of random state space samples, let $\mathbf{Z}(X_{obs}, Y_1, Y_2, \ldots)$ be a random variable representing the computation effort of the planner which can be computed from number of collision checks, the size of the search tree, and the number of random samples drawn during the planning process. $\mathbf{Z}$ is a random variable due to its dependence on the random samples, $\mathbf{Y}_i$, that are drawn. The problem this work addresses is the following optimization problem:

$$\{\mu_k^*\} = \underset{\{\mu_k\}}{\arg\min}\, \mathbb{E}_{X_{obs},\{\mu_k\}}[\mathbf{Z}(X_{obs}, Y_1, Y_2, \ldots)]. \tag{2.11}$$

(2.11) succinctly describes the following: Given a distribution of maps, E, find the sequence of distributions $\{\mu_k^*\}$ that minimizes the expected computational cost of the search, $\mathbf{Z}$. These distributions can be a function of the map and planner. For a robotic arm, this amounts to finding the probability distribution that will minimize the number of collision checks, size of the search tree, and the number of random samples drawn in common tabletop environments.

2.3 LEARNING IMPLICIT SAMPLING DISTRIBUTIONS FOR PLANNING

Unlike in previous work that assumes an optimal planner [4, 10, 23, 48], this work addresses the problem when no such optimal planner is available. Thus, it is not possible to formulate a supervised problem. Instead, a weaker Reinforcement Learning signal can be utilized. While we can not assess the optimality of any given sample, we can attempt to assess the local *goodness* of a sample based on metrics we care about such as computation time. We will begin by introducing how to parameterize distributions and then discuss how to apply the Reinforcement Learning framework to this problem.

2.3.1 Representing Distributions Implicitly

It is difficult to represent the sequence of distributions, μ_k, from (2.11) explicitly. The distribution may be very complicated and not easily representable with simple distributions. In addition, for many problems, there may not be an easy explicit map available (often there is just an oracle that returns whether a collision has occurred or not). A way to implicitly represent a complicated distribution is with rejection sampling, similar to techniques presented in [13, 155, 122, 123]. In our method, random samples will be drawn from some explicitly given distribution, ν (usually the uniform distribution with a peak at the goal). For each random sample $x \in X$ drawn, a probability of rejection is computed. The sample is then either passed to the planner or rejected. The end result is that unfavorable samples are discarded with high probability so computation time is not wasted in attempting to add the node into the tree or in checking it for collisions. This can improve performance as the sampling operation is usually cheap, but collision checking and tree extension is much more expensive. For example, in the robotic arm experiments described later, the policy has learned that samples with large distances between joints and obstacles are unfavorable as it does not progress the search. The policy is learned offline, and is applied to new environments that are similar in nature (for example, in a grasping task, a desk with different objects in different locations).

More formally, the probability of rejecting a sample $x \in X$ is denoted as $\pi(a_{reject}|x)$ where a_{reject} is the action of rejecting a sample. The function, π is learned offline (discussed in Section 2.3.3). Thus, π can implicitly represent a probability measure μ, the distribution that is effectively being sampled when applying rejection sampling.

$$\mu(X_S \subset X) = \frac{\int_{X_S}(1-\pi(a_{reject}|x))d\nu(x)}{\int_X(1-\pi(a_{reject}|x))d\nu(x)} \tag{2.12}$$

This μ is valid as long as $\int_X(1-\pi(a_{reject}|x))d\nu(x)$ is some finite positive number. This will be easily satisfied if $\pi(a_{reject}|x) < 1, \forall x \in X$.

2.3.2 Rejection Sampling as a Markov Decision Process

The process of rejecting samples during the planning algorithm will be modeled as a Markov Decision Process as defined in Section 2.1.3. In the setting of sampling-based planners, a state $s_t \in \mathcal{S}$ consists of the environment, X_{obs}, the current state of the planner, and a randomly generated random sample $x_t \in X$ from the distribution ν. The action space is $\mathcal{A} = \{a_{accept}, a_{reject}\}$. Upon taking action a_{accept}, the sample $x_t \in X$ will be passed to the planner. Upon taking action a_{reject}, the sample will be rejected. In both cases, a new random sample, $x_{t+1} \in X$ will be included in the new state s_{t+1}. A reward, $r(s_t, a_t)$ is given based on $\mathbf{Z}(X_{obs}, Y_1, Y_2, \dots)$. The cost defined for Z will simply become the negative reward. A MDP model of the rejection sampling applied to RRT is described pictorially in

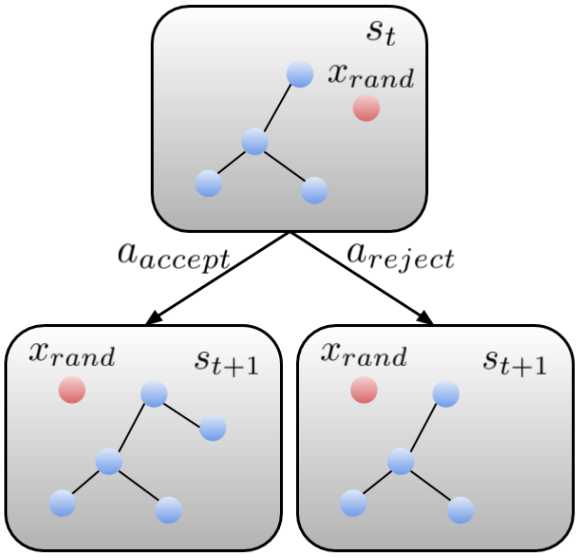


Figure 2.4: **(Rejection Sampling MDP)** MDP representing rejection sampling in a RRT. Blue circles represent nodes in the tree, while the lines represent edges connecting nodes. At a state s_t, you can transition to possible next states, s_{t+1}, by rejecting or accepting the random sample x_{rand}.

Figure 2.4. Note that algorithms that may use batches of samples such as PRM or BIT* can utilize this simply by drawing and rejecting samples until there is enough for a batch.

The policy will be defined as $\pi(a|s)$, the probability of taking action a in state s. Furthermore, π will be restricted to a class of functions with parameters θ and take in as input a feature vector $\phi(s)$ instead of the raw state s. The policy will be referred to as $\pi_\theta(a|\phi(s))$. In this work, the function is represented as a neural net where θ represents the weights in the network. By implicitly defining probabilities μ_k in (2.12) with policy π_θ, μ_k can be written as a function of θ. The optimization problem in (2.11) can be rewritten as

$$\theta^* = \arg\min_\theta \mathbb{E}_{X_{obs}}[\mathbf{Z}(X_{obs}, Y_1(\theta), Y_2(\theta), \dots)]. \tag{2.13}$$

where all μ_k share the same parameters θ but may be different distributions due to the different states the planner will be in. Furthermore, to keep notation with the reinforcement learning literature, the planning cost, $\mathbf{Z}$, will be redefined as

$$\mathbf{Z}(X_{obs}, P, Y_1, Y_2, \ldots) = -\sum_{t=0}^{T} r_{X_{obs}}(s_t, a_t) \tag{2.14}$$

where the rewards $r_{X_{obs}}(s_t, a_t)$ have been chosen to reflect the negative cost represented by $\mathbf{Z}(X_{obs}, Y_1, Y_2, \ldots)$. Specific reward functions for experiments are described in Section 2.4. Finally, the expectation can be approximated with some samples of typical environments that E contains.

$$\theta^* = \arg\max_{\theta} \frac{1}{|I_E|} \sum_{X_{obs} \in I_E} \mathbb{E}_{\{a_i\} \sim \pi_\theta}[r_{X_{obs}}(s_t, a_t)]. \tag{2.15}$$

where I_E is a set of X_{obs} that are representative of the environment E.

2.3.3 Solving the Markov Decision Process

There are many methods from reinforcement learning literature that has been developed to solve the optimization problem posed in (2.15). This work utilizes policy gradient methods as described in Section 2.1.3, specifically REINFORCE with a baseline. The rationale for choosing policy gradient methods over value-based methods is that the policy will have an explicit form that is fast to evaluate which is vital as the policy will be used in the innerloop of sampling-based planners. The baseline $V(\phi(s_t))$ will provide an estimate of the value function. Given an environment X_{obs} and policy π_θ, policy gradient can be estimated by running the planner N times with π_θ and collecting samples of $(\phi(s_t), a_t, r_{X_{obs}}(s_t, a_t), V(\phi(s_t)))$ tuples to calculate $\sum_{t=0}^{T} \nabla \log(\pi_\theta(a_t|\phi(s_t)))(R_t^{X_{obs}} - V(\phi(s_t)))$ for each rollout, then averaging over the N rollouts.

During training, another neural network is fitted to represent $V_w(\phi(s_t))$ with weights w. Utilizing the samples $(\phi(s_t), a_t, r_{X_{obs}}(s_t, a_t), V(\phi(s_t)))$ in each iteration of the policy gradient ascent, an iteration of gradient descent is run on w to minimize the loss function

$$L = \sum_{t=0}^{T} (V(\phi(s_t)) - R_t^{X_{obs}})^2. \tag{2.16}$$

to update the baseline $V(\phi(s_t))$. The steps of the algorithm are detailed in Algorithm 6.

One downside of policy gradient methods is that they are susceptible to local minima as the objective function is not convex. To mitigate this, several different policies are initialized and the best policy is chosen. Different features should also be tested. The performance depends on what information is available. An example what the rewards might look like while running the policy gradient algorithm is shown in Figure 2.5.

Algorithm 6 Learning Sample Distribution

```
1:  procedure LEARN($\{X_{obs}^{(i)}\}_{i=1}^{M}$)
2:      Initialize parameters $\theta_0$ for policy $\pi_{\theta_0}$
3:      Initialize parameters $w_0$ for value baseline, $V_{w_0}$
4:      Run planner with $\pi_{\theta_0}$ several times with each environment and collect data $D_0 =$
        $(\phi(s_t), a_t, r(s_t, a_t), V_{w_0}(\phi(s_t)))$
5:      Use $D_0$ to fit $V_{w_0}$ by running gradient descent on the loss function in (2.16)
6:      for i=1:NumIterations do
7:          for each environment in $I_E$ do
8:              Run $\pi_{\theta_{i-1}}$ N times and collect data $D_{i,j}$
9:              Use (2.10) to compute gradient and update $\theta_i = \theta_{i-1} + \eta_\theta \nabla V(s_0)$
10:             Compute gradient of (2.16) and update $w_i = w_{i-1} - \eta_w \nabla L$
11:         end for
12:     end for
13: end procedure
```

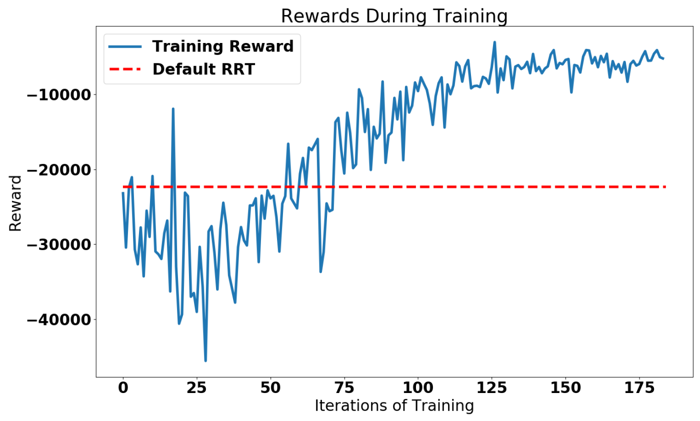


Figure 2.5: **(Policy Gradient Reward Curve)** Rewards while using the policy gradient to update a rejection sampling policy for RRT.

2.3.4 Probabilistic Completeness

The original RRT algorithm has the property of being probabilistically complete. That is, given that a solution exists for a motion planning problem, the probability that RRT will find a solution converges to 1 as the number of samples converges to infinity. It is intuitive that this process of rejection sampling will preserve probabilistic completeness for RRT. Following the original proof in [72], the existence of an attraction sequence of length K between the start and goal positions is assumed. $\{A_0, A_1, \ldots, A_k\}, A_k \subset X$ is an attraction sequence if $\forall A_k$, there exists a subset $B_k \subset X$ such that

1. $\forall x \in A_{k-1}, y \in A_k, z \in X \backslash B_k$, the distance $d(x,y) < d(x,z)$
2. $\forall x \in B_k$, it is possible for the EXTEND function (Algorithm 2) to extend into the set $A_k \subset B_k$

The proof then shows that there is a minimum probability of transitioning from one attraction set in the attraction sequence to the next. Treating the transition as a biased coinflip with success rate p, the question of whether a path is found in N steps turns into a question of whether or not out of N coinflips, K are successful. In [72], p is given as

$$p = \min_i \{\nu(A_i)/\nu(X_{free})\} \tag{2.17}$$

where A_i is the i^{th} element in the attraction sequence. The rejection sampling modifies $\nu(A_i)$ and not $\nu(X_{free})$. Setting a lower threshold for the probability of acceptance of a sample as ϵ, we can write

$$\begin{aligned}
\mu_k(A_i) &= \frac{\int_{A_i} \pi(a_{accept}|x) d\nu(x)}{\int_{A_i} \pi(a_{accept}|x) d\nu(x) + \int_{X \setminus A_i} \pi(a_{accept}|x) d\nu(x)} && (2.18)\\
&\geqslant \frac{\int_{A_i} \epsilon d\nu(x)}{\int_{A_i} \epsilon d\nu(x) + \int_{X \setminus A_i} 1 d\nu(x)} && (2.19)\\
&\geqslant \frac{\int_{A_i} \epsilon d\nu(x)}{\int_{A_i} 1 d\nu(x) + \int_{X \setminus A_i} 1 d\nu(x)} && (2.20)\\
&= \epsilon \nu(A_i) && (2.21)
\end{aligned}$$

Thus, when evaluating the modified p for the learned distribution

$$p = \min_i \{\mu_k(A_i)/\nu(X_{free})\} \geqslant \epsilon \min_i \{\nu(A_i)/\nu(X_{free})\}. \tag{2.22}$$

One key difference between the original proof and our method is that the samples drawn are no longer independent, as the acceptance or rejection of a sample can influence future samples. However, the probability of drawing a sample from A_i some K number of times is lower bounded by $(\epsilon p)^K$ since each sample has at least $\epsilon \nu(A_i)$ probability of being drawn. Thus, the probability that the modified distribution draws K successful samples from N tries is lower bounded by the probability of drawing K successful independent samples out of N from a biased coin flip with $p' = \epsilon p$.

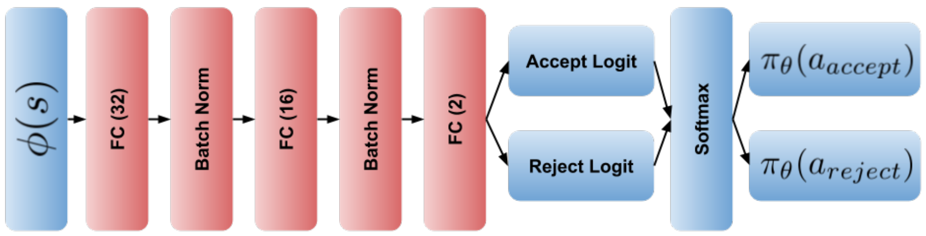


Figure 2.6: **(Policy Network Architecture)** Neural network architecture used for rejection sampling policy. FC(N) stands for a Fully Connected Layer with N neurons.

Thus, this method simply scales the probability p of the original proof by a constant factor, preserving probabilistic completeness.

2.4 EXPERIMENTS

This section will detail experiments to test the efficacy of our heuristic learning approach. Experiments were performed in three sets of environments. First, we tested the algorithm on three different planners in a simulated FlyTrap environment. This allowed us to analyze the learned policies and behavior in detail in a relatively simple environment. Next, the algorithm was tested on a pendulum environment to analyze its performance with dynamical systems. Then, we applied the algorithm to a more complicated 7 degree of freedom robotic arm to show performance on a real system.

2.4.1 Implementation Details

We begin by detailing the specific implementation details common to all experiments. This includes the details of the reward function and the policy and value neural networks.

Reward Function

The reward function $r(s, a)$ used is chosen to reflect the computation time of the planning algorithm.

$$r(s_t, a_t) = -(\lambda_1 1 + \lambda_2 n_{node,t} + \lambda_3 n_{collision,t}) \tag{2.23}$$

λ_1 is a small value that represents the cost of sampling. $n_{node,t}$ is the number of nodes added to the tree in iteration t and $n_{collision,t}$ is the number of collisions checks performed in iteration t. λ_2, λ_3 are simply scaling factors (the experiments use $\lambda_1 = 0.01, \lambda_2 = \lambda_3 = 1.$). Note that the total reward $\sum_{t=0}^{T} r(s_t, a_t)$ will simply be the scaled total number of nodes plus the scaled total number of collisions plus the scaled total number of samples drawn from ν. The reward function is designed to reflect the operations that take the majority of the time: extending the tree and collision checking. The reward function can be made more elaborate, or be nonlinear, but this form is used for simplicity. In practice, this method can be made more accurate by measuring the time of each operation (collision check, node expansion, etc.) to compute the weighting factors λ_i. In addition, the rewards are normalized by their running statistics so that all problem types can have similar reward ranges.

Policy and Value Networks

In this work, the policy π_θ is a neural network that outputs probabilities of acceptance and rejection. The choice in using a neural network to represent the policy is due to the flexibility of functions they can represent. Initial results showed that a simple model like logistic regression can be insufficient in complicated environments. In addition, with neural networks, there is no need to select basis functions to introduce nonlinearities.

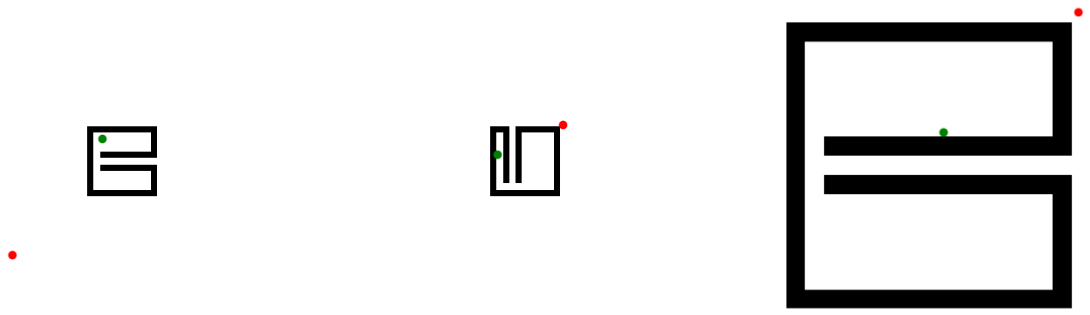
The network used is a relatively small two layer perceptron network (the inference must be fast as this function is run many times in the inner loop of the algorithm). For reference, the network evaluated a sample in around 3.59 microseconds using only a laptop CPU running at 3.5Ghz. The input $\phi(s)$ is passed through two hidden layers with 32 and 16 neurons and rectified linear activation. There is a batchnorm operation [51] after each hidden layer. The second batchnorm layer is passed to a final fully connected layer with 2 outputs that represent the logit for accepting or rejecting the sample. The logit is fed into a softmax operation to obtain the probabilities. Additionally, the logits are modified so that all probabilities lie between 0.05 and 0.95. This is so that $\pi_\theta(a_{reject}|s) < 1$ in order to guarantee that μ_k is a valid probability distribution. This also allows the policy to always have a small chance of accepting or rejecting, which is useful for exploration in the reinforcement learning algorithm. The policy network is shown in Figure 2.6.

The neural network for $V(\phi(s))$ is similar to the policy network. The only difference is that the output layer is a single neuron representing the value. All networks are trained with the Adam optimizer [60] with a learning rate of 0.001.

During execution, multiple samples may be batched together to run in parallel through the neural network. This slightly breaks assumptions of the heuristic learning algorithm proposed as each sample might be dependent on the one before it. However, empirically, batches of 64 samples do not meaningfully impact any metrics measured.

2.4.2 Flytrap Environments

The first experiment run is that of the 2D Flytrap. This environment is used as a benchmark in [122] and [155] as an example of a hard planning problem. It is difficult to solve because of the thin tunnel that must be sampled in order to find a path to the goal. The training and testing environments are shown in Figure 2.7. Three different planners are tested on the environment: RRT-Connect function with one tree [65], Bidirectional RRT-Connect (BiRRT) [65], and EST [47].



(a) Train (b) Test (c) BallTree

Figure 2.7: **(Various Flytrap Environments)** The green dot is an example starting location and the red dot is an example goal location. 2.7a is what the policy is trained on in the experiment. 2.7b is what the policy is tested on for the experiment. 2.7c shows the environment used by BallTree [122]

For RRT, the feature used is the distance to the nearest tree node minus the distance of that tree node to its nearest obstacle. For BiRRT, the feature used is the distance to the current tree being expanded minus the distance of that tree node to its nearest obstacle. For EST, there are a few choices for how to modify the sampling. In this experiment, we chose to modify the probability of picking nodes in the tree for expansion (the alternative being modifying the probability of how to pick nodes to expand to) since the choice of node has a larger effect on the algorithm's performance. The features used are two dimensional: the nearest obstacle to the node, as well as the number of nodes in a certain radius (this is the same as $w(x)$ used in the original EST work [47]).

For each planner, the original policy of always accepting samples is compared against the policy trained on the environment shown in Figure 2.7a. The results in Figure 2.8 show the statistics over 100 run. The average of each metric tracked for the planners is compared.

For all planners, the number of collision checks is reduced while the number of samples drawn is increased. In RRT, it is reduced around five times. The trade off between collision checks and number of samples saves overall execution time. In addition, the decreasing the tree size and reducing collision checks does not decrease the quality of the paths found.

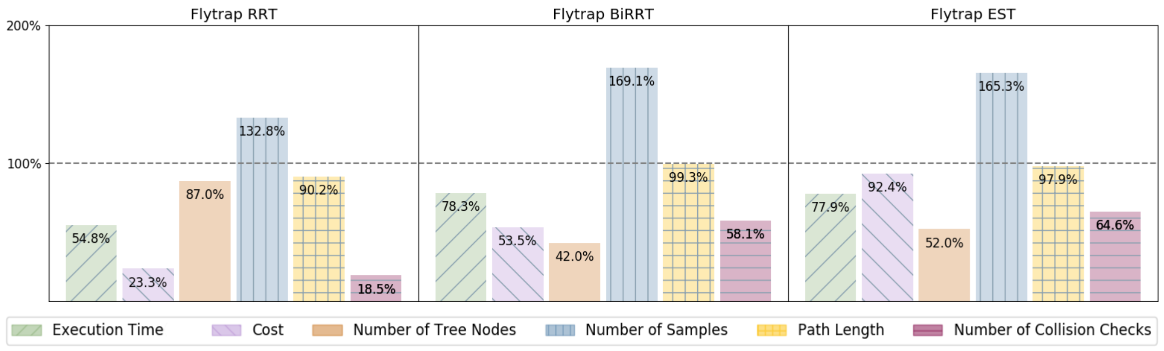


Figure 2.8: (Flytrap Experiment Results) Results of 100 runs of each planner on the test Flytrap environment. Each bar shows the ratio of the learned planner's metric to the unmodified planner (over 100% means more than the original planner).

For each planner, the path found by the trained policy is equivalent in length or sometimes shorter, despite not explicitly optimizing for path length.

Analysis

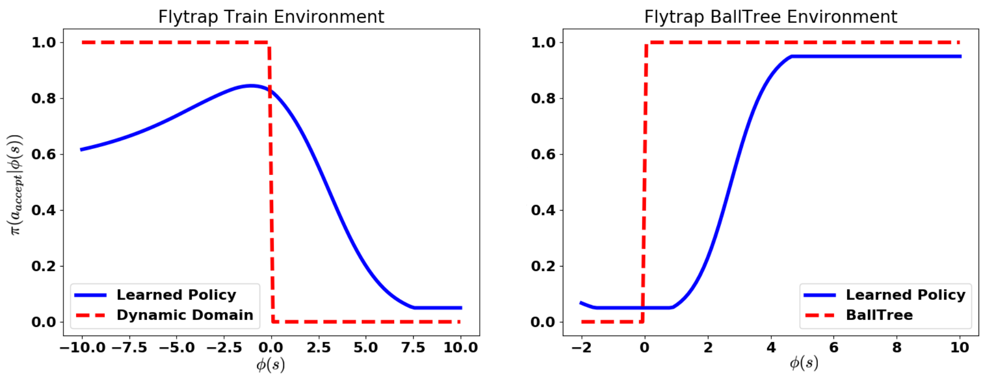


Figure 2.9: (Learned and Heuristic Rejection Policies) Comparison between learned policies and BallTree and Dynamic Domain RRT.

Next, the policies learned for RRT are analyzed. The learned policy rejects samples that are far away from the tree with higher probability. This is similar to the strategy that is suggest by Dynamic Domain RRT [155], in which the ideal version of it rejects all samples that are further away from the tree than the closest obstacle. However, for

Flytrap environments where the space outside of the Flytrap is not a large fraction of the space, the strategy suggested by BallTree [122] is more effective. BallTree rejects all samples that are closer to the tree than the nearest obstacle. It is curious that for very similar types of environments, the policies that work better for each are almost complete opposites! This shows a need for some environment types to use the data itself to tune a rejection sampling policy. When training on the different sized environment shown in Figure 2.7c, the policies learned to exhibit behavior similar to BallTree. The policy trained in the larger environment rejects samples further from the tree, and the policy trained in the smaller environment rejects samples that are closer to the tree as shown in Figure 2.9. The distributions encountered during the search process are visualized in Figure 2.10 by sampling a uniform grid in the state-space and using (2.12) to compute discretized probabilities for sampling each point.

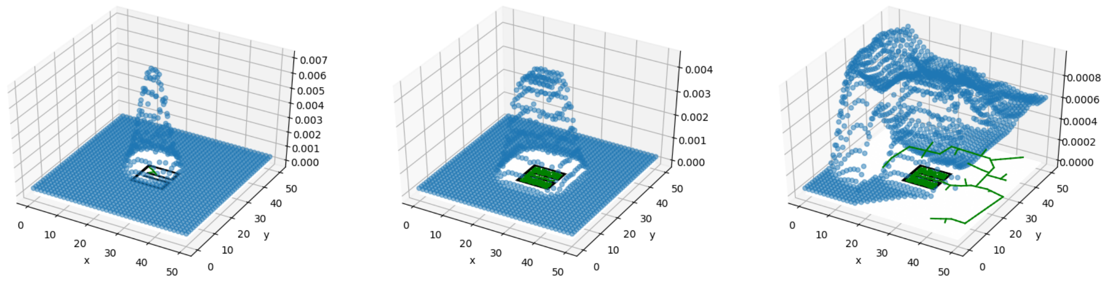


Figure 2.10: (Learned Sampling Distributions) Learned probability distributions for RRT. While the policy is the same, the distributions change as the RRT search progresses. For each figure, the bottom plane shows the environment with a green search tree, while the blue dots show sampled points representing the learned distribution.

2.4.3 Simulated Pendulum

In addition to the flytrap environment, experiments were done on a planar pendulum to test the effectiveness of it on a dynamical system. The pendulum starts at the bottom and needs to reach the top. It is control limited so it must plan a path that increases its energy until it can swing up. In this experiment, we used a steering function that randomly samples control actions and time durations. This is a common steering function that may be used in more complicated systems [78]. The results are shown in Figure 2.11. Number of collision checks is not included as for this particular experiments as there are no obstacles to collide with. The features used are 1) the difference between the goal angle and the current angle and 2) the difference in angular velocities. The policy learns to reject samples that are not likely to lead to the goal state, which saves the execution time otherwise spent computing the steering function.

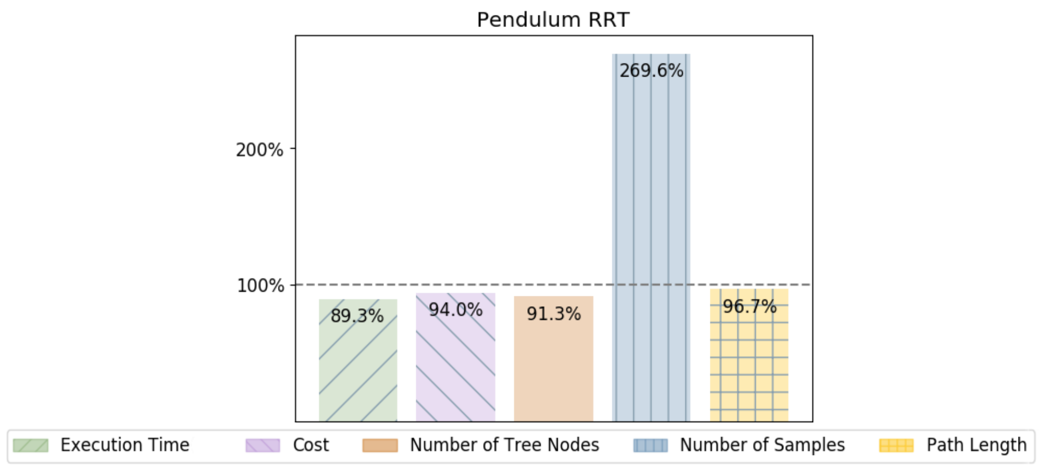


Figure 2.11: **(Pendulum Results)** Results of 100 runs of each planner on the Pendulum environment. Each bar shows the ratio of the learned planner's metric to the unmodified planner (over 100% means more than the original planner).

2.4.4 Real Robot Arm

The algorithm is also tested on the 7 degree of freedom arm of the Thor robot (Figure 2.12). This experiment is used to validate the method in a higher dimensional space and in a realistic environment. Thor is given tasks to move its arm to various difficult to reach

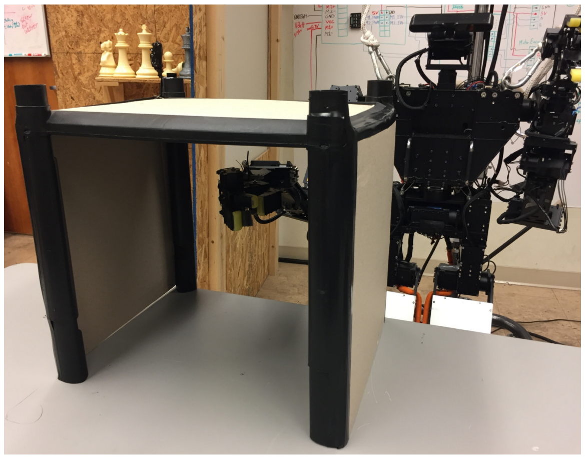


Figure 2.12: The Thor robot in a test of the tabletop environment.

places in assorted tabletop environments. The environments consists of crevices for Thor to reach into and obstacles to block passages. The base planner used is BiRRT, with a four dimensional feature space (EST and RRT were not used as the planning took too long). The first three features are the distances of various joints to the closest obstacle, and the last feature is the distance of the current configuration to the goal. Two very different environments were used for training, and a third environment distinct from the first two was used for testing.

The results of the arm experiments are shown in Figure 2.13. The figure details the statistics over successful plans over 100 runs of the planner. Our algorithm had 97% success rate in finding a path, while the original had 96% when the number of samples drawn is limited to 100,000 (this difference is too small to make any claims). Similar to the Flytrap experiments, a policy is learned that trades off extra samples for a vastly reduced number of collision checks and nodes in the tree. On the test environment, the number of nodes in the tree is more than 5 times less and uses 2.7 times less collision checks. In addition, the variance of the results is greatly reduced when using the learned distribution.

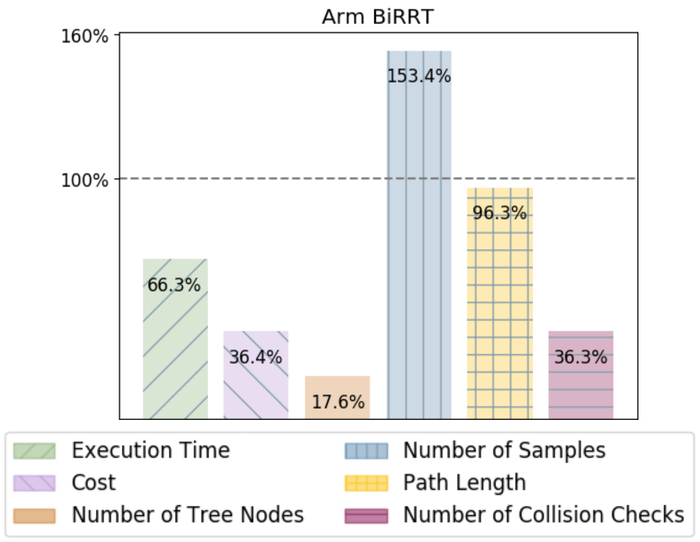


Figure 2.13: **(Robot Arm Results)** Comparison of results of BiRRT in 7 degree of freedom robot arm tabletop environments. Each bar shows the ratio of the learned planner's metric to the unmodified planner

Next, the policies learned for the Thor arm are examined to see what aspect of the environment it is exploiting. A visualization is shown in Figure 2.14. We note that the probability increases as 1) the distance of the configuration to the goal is lower, or 2) the workspace distance of the later joints is closer to an obstacle. This policy makes a lot of intuitive sense. Samples are concentrated near the surface of the table and objects, probing the surface for a good configuration.

2.5 CONCLUSIONS

This chapter described learning sampling distributions for sampling-based motion planners. Sampling distributions in sampling-based motion planners are a vital component of the algorithm that affects how many times computationally expensive subroutines such as collision checks are run. While the method presented can improve planning times by

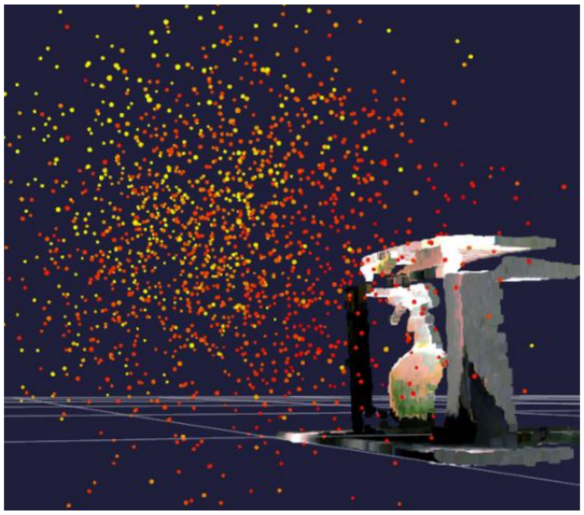


Figure 2.14: **(Learned Sampling Distribution)** Visualization of the learned rejection policy for the tabletop environment. On the bottom right is a point-cloud representation of a test environment. A cleaning spray is hidden within an open box. Each colored dot represents the position of the end effector for a configuration state. A yellow dot represents a state with high rejection probability while a red one represents one with low rejection probability.

modifying the sampling distribution, it is not the whole solution for all problem types. In maps where the thin tunnel issue is more pronounced, rejection sampling does not alleviate the main dilemma of how to sample the thin tunnel. However, this method can be easily combined with existing techniques such as [161, 153, 23] to improve performance.

When an optimal-planner is not available to provide direct supervision for heuristic learning, a Reinforcement Learning approach is feasible. The learned distributions can coincide with human intuition for which samples are beneficial. For environments tested in [122, 155], the learned distributions appeared to be soft approximations of human designed heuristics which validates the effectiveness of the learning process. We presented a general way to obtain good implicit sampling distributions in the absence of direct supervision. The process can be seen as a way of encoding the prior knowledge of the environments into the rejection policy by learning from previous searches in similar environments to solve the *Experienced Piano Mover's Problem*.

3 CONSTRAINED MODEL LEARNING

Chapter 2 discussed learning heuristics from data to improve the computational efficiency of motion planning algorithms. This section will discuss the other part of a motion planning algorithm: the system model. Many motion planning methods perform admirably when the models can approximate the dynamics of the system accurately. However, the performance of these planners can be degraded with model inaccuracy. While some planners try to explicitly account for model inaccuracy or uncertainty [44, 85, 126], they tend to yield overly conservative motion plans. In the field of control theory, *robust controllers* [162] have been developed to address the same problem. These methods often need to consider the worst case scenarios at each state which can lead to undesirable or overly conservative motion plans. Additionally, these methods have the assumption that user of the planning algorithm has an idea and model of the kinds of errors that can occur. For example, consider the problem of landing a quadrotor precisely at a target. There are complex aerodynamic effects associated when nearby surfaces cause disturbances to the airflow. This may result in large torques when the quadrotor is hovering close to the ground and hamper precise landings. This is known as the *ground effect* [112]. These aerodynamic effects can be hard to model from just prior knowledge and may show up as a highly correlated, state dependent, non-zero mean noise. A common method that has been suggested to model similar effects is to learn or adjust a dynamics model with real data taken from running the system.

In robotics, Gaussian Processes, Gaussian Mixture Models, or Neural Networks have been used to learn models of dynamics [26, 94, 76, 91, 109]. A typical process for learning these models involves selecting a parameterized model, such as a neural network with a fixed number of layers and neurons, and choosing a loss function that penalizes the output of the model for not matching the data gathered from running the real system. Then, one optimizes the parameters by minimizing the empirical risk using, for instance, stochastic gradient descent like algorithms. This formulation assumes that all transitions are equally important since it penalizes the mismatch between model and data uniformly on all portions of the collected data. While this formulation has shown success in some applications, prior knowledge about the task and system can inform better learning objectives. A control designer may know that a certain part of the state space requires a certain accuracy for a robust controller to work well, or that some part of the state space is more important and should have hard constraints on the model accuracy. For example, to precisely land a quadrotor, a designer may note that the accuracy of modeling the complex ground effect forces is more important near the landing site. Incorporating this prior knowledge can lead to better performing motion planners.

To address the problem of incorporating prior knowledge into model learning, we introduce the idea of *sufficiently accurate* model learning [159, 160]. This formulation is based on the inclusion of constraints in the optimization problem whose role is to introduce prior-knowledge about the importance of different state-control subsets. In the example of the quadrotor, notice that when the quadrotor is away from the surfaces, the ground effect is minor and thus, it is important to focus the learned model's expressiveness in the region of the state-space that is most heavily affected. This can be easily captured by a constraint that the average error in the important state-space regions is smaller than a desired value. These constraints will allow models with finite expressiveness concentrate on modeling important aspects of a system. One point to note is that this constrained objective can be used orthogonally to many existing methods. For example, the constrained objective can replace the unconstrained objective in [76, 91], and all other aspects of the methods can remain the same.

In its most generic form, the problem of model learning is an infinite dimensional non-convex optimization problem that involves the computation of expectations with respect to an unknown distribution over the state-action space. In addition, the formulation proposed here introduces constraints which seems to make the learning process even more challenging. However, in this work we show that solving this problem accurately is not more challenging than solving an unconstrained parametric learning problem. To reach this conclusion we solve a relaxation of the problem of interest with three common modifications: (i) function parameterization, (ii) empirical approximation, and (iii) dual problem solving. Function parameterization turns the infinite dimensional problem into one over finite function parameters. Empirical approximation allows for efficient computation of approximate expectations, and solving the dual problem leads to a convex unconstrained optimization problem. The three approximations introduced however may not yield solutions that are good approximations of the original problem. To that end, we establish a bound on the difference of the value of these solutions. This gap between the original and approximate problem depends on the number of samples of data as well as the expressiveness of the function approximation (Theorem 1). In particular, the bound can be made arbitrarily small with sufficient number of samples and with the selection of a rich enough function approximator. This implies that solving the functional constrained problem is nearly equivalent to solving a sequence of unconstrained approximate problems using primal-dual methods.

Section 3.1 will introduce the necessary background in model learning and Lagrangian duality. Then, Section 3.2 presents the *sufficiently accurate* model learning framework. Section 3.3 will examine the approximation error between the practical model learning problem and the general idealized model learning problem. One such algorithm to solve constrained problems is discussed in Section 3.4 and numerical experiments are presented in Section 3.5.

3.1 BACKGROUND

3.1.1 Types of Models and Model Learning

Before discussing our method of *Sufficiently Accurate Model Learning*, we will examine the types of models that are used as well as different model learning methods. We begin with examples of system models. The system model is a function that models how the configuration state of a robot changes under some control input. In this work, we will deal mainly with fully observable systems. Even among fully observable systems, there are a variety of ways to represent the dynamics. A natural way to model a system may be with a differential equation $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$

$$\dot{x} = f(x, u) \tag{3.1}$$

where $x \in \mathcal{X}$ is a state from the configuration space, and $u \in \mathcal{U}$ is a control from the control space. This is a natural way to describe a physical systems as differential equations are at the core of Newton's Laws of Motion or Lagrangian Mechanics. In fact, the standard form to describe the dynamics of an open-chain robotic manipulator is a second order differential equation [90, Chapter 3.2] of the form

$$M(x)\ddot{x} + C(x, \dot{x})\dot{x} + N(x, \dot{x}) = u \tag{3.2}$$

which can be described by (3.1) by replacing the second order system of differential equations with a first order system with a state space that contains both x as well as $\dot{x}$. For some systems, the differential equation can also change with time

$$\dot{x} = f(x, u, t). \tag{3.3}$$

A model changing over time can be used to model problems with dynamically moving obstacles or other agents. Another type of system model would be a discrete-time model.

$$x_{t+1} = f(x_t, u_t) \tag{3.4}$$

where x_t and x_{t+1} are states at a time index $t \in \mathbb{Z}$. This can more naturally fit some of motion planning algorithms described in Section 1.2. Graph-based planners look at how the state evolves at discrete time steps as well as many formulations of optimization-based planners. This does not mean a continuous time differential model will not work as it is possible to simply integrate the differential equation. In practice, the choice in model may be decided by how easy it is to represent or learn each model.

The problem of model learning reduces to finding the function ϕ in the class Φ that best fits the transition data. For example, Φ could be the space of all continuous functions. The figure of merit is a loss function $\ell : \mathcal{S} \times \Phi \to \mathbb{R}$. With these definitions, the problem of interest can be written as the following stochastic optimization problem

$$\phi^* = \arg\min_{\phi \in \Phi} \mathbb{E}_{s \sim \mathcal{S}_D}[\ell(x, \phi)] \tag{3.5}$$

where $s = (x_t, u_t, x_{t+1})$ are tuples of data samples from the sample space taken from observing the real system and the expectation is taken over a distribution of real system data $\mathcal{S}_D$ with a sample space $\mathcal{S} = \mathcal{X} \times \mathcal{U} \times \mathcal{X}$. The goal is to minimize some loss function,

$\ell : \mathcal{S} \times \Phi$, between the true observed next state x_{t+1} with the parameterized model's output. An example of a loss function is the p-norm,

$$\ell(s, \phi) = \|\phi(x_t, u_t) - x_{t+1}\|_p . \tag{3.6}$$

For $p = 1$, this reduces to a sum of absolute differences between each output dimension of ϕ and the true next state, x_{t+1}. When $p = 2$, we obtain the Euclidean loss. A combination of both $p = 1$ and $p = 2$ losses is known as an Elastic net loss [164]. Other common losses include the Huber loss [88] and, in discrete state settings, a 0-1 loss.

Often times a rough model $\hat{f}$ of the dynamical system of interest can be obtained. Depending on the complexity of the system, these models may be inaccurate since they may ignore hard to model dynamics or higher order effects. For instance, one can derive a model $\hat{f}$ for a quadrotor from rigid body dynamics where forces and torques are functions of each motor's propeller speed. However, the accuracy of this model will depend on other effects that are harder to model such as aerodynamic forces near the ground. In these cases, the system model can decomposed as the sum of an analytic model $\hat{f}$ and a *residual error model* ϕ [5, 30, 157]

$$f(x_t, u_t) \approx \hat{f}(x_t, u_t) + \phi(x_t, u_t). \tag{3.7}$$

The residual model can be easily incorporated into the loss defined in (3.5). For instance, the p norm loss can be modified to take the form

$$\ell(s, \phi) = \left\|\phi(x_t, u_t) - (x_{t+1} - \hat{f}(x_t, u_t))\right\|_p .$$

In general, many researchers fit a discrete-time model to collected data [91, 114, 50, 93, 115, 40, 31, 18, 147, 5, 30, 157, 52, 69] using a variation of optimization problem (3.5) . There has also been work on learning a differential equation model. In practice, it is easier to measure the state of a system rather than the time-derivative of the state. Thus, one approach has been compose a supervised learning problem that penalizes an Euler integration of the predicted derivative and the true next state [83, 108]. Neural ODE [21] looks at a loss that depends on black box ODE solvers that can be more sophisticated than Euler integration.

In addition to the deterministic models shown in (3.1) and (3.4), there are also stochastic models. Instead of directly modeling the next state, probabilistic models can represent a distribution over next states. An example of simple probabilistic system model is a state and control dependent Gaussian distribution

$$p(x_{t+1}|x_t, u_t) = \mathcal{N}(\mu(x_t, u_t), \Sigma(x_t, u_t)) \tag{3.8}$$

This type of model is usually used to represent *aleatoric* uncertainty or the intrinsic stochasticity of the system. The actual dynamics can be probabilistic in nature. There can be a philosophical argument whether or not systems can have aleatoric uncertainty, but a practical example can be systems where the full state representation is not observable. A car might have micro-abrasions on the tire which might hit tiny pebbles on the road. Relative to the state that is measurable, the system can be seen as *inherently* stochastic.

There also exists *epistemic* uncertainty – the uncertainty in the model's knowledge of the system. This type of uncertainty can exist in deterministic system models as it refers not to the underlying physical system but knowledge about how accurate the model is. This is usually described by a distribution over models ϕ. This can be realized, for example, by using a neural network with dropout to represent ϕ_θ as each parameter becomes a Bernoulli random variable [33]. Other methods have used an ensemble [52, 69] such that when multiple models closely agree, there is higher certainty in the model's knowledge of the system.

This work will primarily focus on deterministic, discrete-time residual models as shown in (3.7). However, the constrained objective that will be introduced can be used with all model types.

3.1.2 Lagrangian Duality

This section will briefly review the concept of Lagrangian duality that will be utilized and referenced in Sections 3.2.1 and 3.3. A more detailed discussion is presented in [15, Chapter 5]. We begin by defining a standard optimization problem

$$\begin{aligned} P^\star = \min_x \quad & l(x) \\ \text{s.t.} \quad & g(x) \leqslant 0 \end{aligned} \tag{3.9}$$

where $x \in \mathbb{R}^n$ is a vector of variables to optimize over, $l : \mathbb{R}^n \to \mathbb{R}$ is the objective function (or loss function) to minimize, and $g : \mathbb{R}^n \to \mathbb{R}^K$ is a vector-valued constraint function. (3.9) will be denoted as the *primal* problem.

Lagrangian duality explores the relationship between the *primal* problem and an alternative problem known as the *dual problem*. To define the *dual problem*, we first define a quantity known as the *Lagrangian*

$$\mathcal{L}(x, \lambda) = l(x) + \lambda^\top g(x), \tag{3.10}$$

where the *dual variable*(or Lagrangian multipliers), $\lambda \in \mathbb{R}^K$ is introduced. The k^{th} element of λ corresponds with the k^{th} element of the constraint function g as they are multiplied and summed with the other constraints indices. The *Lagrangian* is used in a function known as the *Lagrangian dual function*

$$d(\lambda) = \inf_{x} \mathcal{L}(x, \lambda). \tag{3.11}$$

The *Lagrangian dual function* is used to define the *dual problem*

$$D^{\star} = \max_{\lambda \geqslant 0} d(\lambda). \tag{3.12}$$

The *dual problem* is extremely useful in the analysis of optimization problems. For many optimization problems, the dual formulation can be more practical to solve than the primal. The dual problem optimizes over the dual variable λ which exists in $\mathbb{R}^K$ rather than primal variable x which exists in $\mathbb{R}^n$. For some problems where $K \ll n$, this can be a benefit. Though, note that an infimum still exists in (3.11), so this benefit mainly applies to problems where the dual problem can be reduced to an analytic form. Linear programs, for example, have an analytic dual [15, Chapter 5.2.1]. Additionally, the constraints in the dual formulation consist only of non-negativity constraints $\lambda \geqslant 0$ which can be easier to deal with than the generic constraint function g. Another useful property of the dual function is that (3.11) is concave without an assumption of convexity on the primal problem (3.9) [15, Chapter 5.1.2].

While the dual formulation might be easier to solve for some optimization problems, it may not have the same solution value as the optimal primal solution, $P^{\star}$. The dual function $d(\lambda)$ has an interesting feature in that for the domain considered, $\lambda \geqslant 0$, it is upper bounded by the P^{*} for all λ. This can be seen as for any given λ and feasible x,

$$l(x) + \lambda^{\top} g(x) \leqslant l(x) \tag{3.13}$$

as a feasible x would fulfill the condition $g(x) \leqslant 0$. Thus,

$$d(\lambda) = \inf_x [l(x) + \lambda^\top g(x)] \leqslant l(x^\star) + \lambda^\top g(x^\star) \leqslant l(x^\star) = P^\star \tag{3.14}$$

where $x^\star$ is a solution to (3.9). This implies that the solution to the dual problem (3.12),

$$D^\star = d(\lambda^\star) \leqslant P^\star \tag{3.15}$$

is also upperbounded by the solution to the primal problem. The difference $P^\star - D^\star$ is known as the *duality gap*. There are optimization problems where the duality gap is zero. The solution values to the primal and dual formulations are the same. When this happens, the problem is said to have *strong duality*. Convex problems that fulfill Slater's condition are well known to have strong duality. Convex problems are a specific instance of (3.9) where l and g are convex functions. Slater's condition is fulfilled when there exists x such that the constraint function is strictly satisfied, $g(x) < 0$. Convex problems are not the only class of optimization problems that exhibit strong duality. A relevant class of problems will that are strongly dual will be discussed in Section 3.3.1, Theorem 2.

3.2 SUFFICIENTLY ACCURATE MODEL LEARNING

We now present the framework of *sufficiently accurate* model learning. For clarity, we will focus on the discrete-time deterministic system model presented in (3.4)

$$x_{t+1} = f(x_t, u_t).$$

We consider a Euclidean configuration state space $\mathcal{X} = \mathbb{R}^n$ and Euclidean control space $\mathcal{U} = \mathbb{R}^p$. A characteristic of the classic model learning problem defined in (3.5)

$$\phi^* = \underset{\phi \in \Phi}{\arg\min}\, \mathbb{E}_{s \sim \mathcal{S}_{\mathcal{D}}}[\ell(x, \phi)]$$

is that errors are uniformly weighted across the sample distribution, $\mathcal{S}_{\mathcal{D}}$. In principle, one can craft the loss so to represent different properties on different subsets of the state-input space by weighting the lossses from different regions of the state-control space. However, this design is challenging, system dependent, and dependent on the the distribution of the transition data. In contrast, our approach aims to exploit prior knowledge about the errors and how they impact the control performance. For instance, based on the analysis of robust controllers one can have bounds on the error required for successful control. The model should be able to incorporate prior knowledge such as “The errors in this part of the state-control space should be at least ϵ-accurate." This information can be used to formulate the *sufficiently accurate* model learning problem, where we introduce the prior information in the form of constraints. Formally, we encode the prior information by introducing $K \in \mathbb{N}$ functions $g_k : \mathcal{S} \to \mathbb{R}$. Define in addition, a collection of subsets of transition tuples where this prior information is relevant, $\mathcal{S}_k \subset \mathcal{S}$ and corresponding indicator functions $\mathbb{I}_k(s) : \mathcal{S} \to \{0, 1\}$ taking the value one if $s \in \mathcal{S}_k$ and zero otherwise. With these definitions the sufficiently accurate model learning problem is defined as

$$\begin{aligned} P^\star = \min_{\phi \in \Phi}\, & \mathbb{E}_{s \sim \mathcal{S}_{\mathcal{D}}}[\ell(s, \phi)\mathbb{I}_0(s)] \\ \text{s.t. } & \mathbb{E}_{s \sim \mathcal{S}_{\mathcal{D}}}[g_k(s, \phi)\mathbb{I}_k(s)] \leqslant 0, k = 1, 2, \dots, K \end{aligned} \tag{3.16}$$

Note that the sets $\mathcal{S}_k$ that define the indicator variables $\mathbb{I}_k(s)$ are not necessarily disjoint. In fact, in practice, they are often not. The sets can be arbitrary and have no relation to each other.

Notice that the (3.16) is an infinite dimensional problem since the optimization variable is a function and it involves the computation of expectations with respect to a possibly unknown distribution. An approximation to this problem is presented in Section 3.2.1. For technical reasons, the functions g_k and l should be expectation-wise Lipschitz continuous.

Assumption 3.1. *The functions* $\ell_0(s,\phi) = \ell(s,\phi)\mathbb{I}_0(s)$ *and* $\mathbf{g}_k(s,\phi) = g_k(s,\phi)\mathbb{I}_k(s)$ *are* L*-expectation-wise Lipschitz continuous in* Φ*, i.e.,*

$$\mathbb{E}_s \left\|\ell_0(s,\phi_1) - \ell_0(s,\phi_2)\right\|_\infty \leqslant L\mathbb{E}_s \left\|\phi_1 - \phi_2\right\|_\infty, \forall \phi_1, \phi_2 \in \Phi \tag{3.17}$$

for some L*. Here,* $\|\phi_1 - \phi_2\|_\infty$ *is the infinity norm for functions which is defined as* $\|f\|_\infty = \sup_x |f(x)|$.

The expectation-wise Lipschitz assumption is a weaker assumption than Lipschitz-continuity, as any Lipschitz-continuous function with a Lipschitz constant L is also expectation-wise Lipschitz-continuous with a constant of L. In particular, the loss functions in Example 3.1 and 3.2 are expectation-wise Lipschitz-continuous with some constant (cf., Appendix 6.1). There is no assumption that the functions should be convex or continuous. Before we proceed, we present two examples of *sufficiently accurate* model learning. For notational brevity, when an expectation does not have a subscript, it is always taken over $s \sim \mathcal{S}_\mathcal{D}$.

Example 3.1. *Selective Accuracy*

$$\begin{aligned} &\min_{\phi\in\Phi} \mathbb{E}\left\|\phi(x_t,u_t) - x_{t+1}\right\|_2 \\ &\text{s.t. } \mathbb{E}\left\|\phi(x_t,u_t) - x_{t+1}\right\|_2 \mathbb{I}_A(s) \leqslant \epsilon_c \end{aligned} \tag{3.18}$$

This problem is a simple modification of (3.5)*. It has the same objective, but adds a constraint that a certain state-control subset, defined by a set* A*, should be within* ϵ_c *accuracy. The indicator variable* $\mathbb{I}_A(s)$ *will be 1 when* s *is in the set* A*. Here,* $g(s, \phi) = \|\phi(x_t, u_t) - x_{t+1}\|_2 - \frac{\epsilon_c}{\mathbb{E}\mathbb{I}_A(s)}$*. This formulation allows you to trade off the accuracy in one part of the state-control space with everything else as it may be more important to a task. Another use case can be to provide an error bound for robust controllers. This is the formulation used in the quadrotor precise landing experiments detailed later in Section* 3.5.3*, where the set* A *is defined to be all states close to the ground where the ground effect is more prominent.*

Example 3.2. ***Normalized Objective***

$$\begin{aligned} &\min_{\phi \in \Phi} \mathbb{E} \frac{\|\phi(x_t, u_t) - x_{t+1}\|_2}{\|x_{t+1}\|_2} \mathbb{I}_A(s) \\ &\text{s.t. } \mathbb{E}\, \|\phi(x_t, u_t) - x_{t+1}\|_2\, \mathbb{I}_{A^C}(s) \leqslant \epsilon_c \end{aligned} \tag{3.19}$$

where $\mathbb{I}_A$ *is the indicator variable for the subset* $A = \{s \in \mathcal{S} : \|x_{t+1}\|_2 \geqslant \delta_c\}$*, and* A^C *is the complement of the set* A*. This problem formulation looks at minimizing an objective such that the error term is normalized by the size of the next state. This can be useful in cases where the states can take values in a very large range. An error of 1 unit can be large if the true value is 0.1 units, but it is a small error if the true value is 100 units. The set* A *contains all data samples where the true next state is large enough for this to be significant. This can reduce numerical issues when the denominator is small. For all small state values, the error is simply bounded by* ϵ_c*. From a practical point of view, sensors will always have noise. When the state is small, the "true" measurement of the state can be dominated by noise, and the model can be better off just bounding the error rather than focusing on fitting the noise. This is the formulation used in the ball bouncing experiment in Section* 3.5.2*, where the we would like the errors in velocity prediction to be scaled to the speed, and all errors below a small speed can be constrained with a simple constant.*

3.2.1 Problem Approximation

The unconstrained problem in (3.5) and the constrained problem in (3.16) are functional optimization problems. In general, these are infinite dimensional and usually intractable. This section will present common approximations of (3.16). Instead of optimizing over the entire function space Φ, one may look at function spaces, $\Phi_\theta \subset \Phi$, parameterized by a d-dimensional vector $\theta \in \Theta = \mathbb{R}^d$. Examples of these classes of functions are linear functions of the form $\phi_\theta(x,u) = \theta_x^\top x + \theta_u^\top u$ where $\theta = [\theta_x, \theta_u]$ is a vector of weights for the state and control input. More complex function approximators, such as neural networks, may be used to express a richer class of functions [46, 91]. Restricting the function space poses a problem in that the optimal solution to (3.16) may no longer exist in the set Φ_θ. The goal under these circumstances should be to find the closest solution in Φ_θ to the true optimal solution $\phi^\star$. Additionally, the expectations of the loss and constraint functions are in general intractable. The distributions can be unknown or hard to compute in closed form. In practice, the expectation is approximated with a finite number of data samples $s_i \sim \mathcal{S}_\mathcal{D}$ with $i = 1, \ldots, N$. This yields the following empirical parameterized risk minimization problem

$$\begin{aligned} P_N^\star = &\min_{\theta\in\Theta} \frac{1}{N}\sum_{i=1}^{N} l(s_i, \phi_\theta)\mathbb{I}_0(s_i) \\ &\text{s.t. } \frac{1}{N}\sum_{i=1}^{N} g_k(s_i,\phi_\theta)\mathbb{I}_k(s_i) \leqslant 0, k = 1, 2, \ldots, K \end{aligned} \tag{3.20}$$

While both function and empirical approximations are common ways to simplify the problem, the approximate problem (3.20) is still a constrained optimization problem and can be difficult to solve in general as it can be nonconvex in the parameters θ. This is the case for instance when the function approximator is a neural network. One approach to solve this problem is to solve the dual problem associated with (3.20). To aid in the definition of the dual problem, we define the Lagrangian associated with (3.20)

$$\mathcal{L}_N(\theta, \lambda) = \frac{1}{N} \sum_{i=1}^{N} \ell_0(s_i, \phi_\theta) + \lambda^\top \frac{1}{N} \sum_{i=1}^{N} \mathbf{g}(s_i, \phi_\theta). \tag{3.21}$$

where $\lambda \in \mathbb{R}_+^K$ are the dual variables as discussed in Section 3.1.2.

Here, the symbol, $\ell_0(s_i, \phi_\theta)$, is defined as $l(s_i, \phi_\theta)\mathbb{I}_0(s)$ to condense the notation. Similarly, the bolded vector, $\mathbf{g}(s_i, \phi_\theta)$ is a vector where the k^{th} entry is defined as $g_k(s_i, \phi_\theta)\mathbb{I}_k(s)$. Similar to Section 3.1.2, the dual problem is now defined as

$$D_N^\star = \max_{\lambda \geqslant 0} \min_{\theta \in \Theta} \mathcal{L}_N(\theta, \lambda) \tag{3.22}$$

Notice that (3.22) is similar to a regularized form of (3.20) where each constraint is penalized by a coefficient ω_k

$$D_N^\star = \min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} \ell_0(s_i, \phi_\theta) + \boldsymbol{\omega}^\top \frac{1}{N} \sum_{i=1}^{N} \mathbf{g}(s_i, \phi_\theta). \tag{3.23}$$

Adding this type of regularization can weight certain state-action spaces more. In fact, if ω_k is chosen to be $\lambda_N^\star$ (the solution to (3.22)), solving (3.23) would be equivalent to solving (3.22). However, arbitrary choices of ω_k provide no guarantees on the constraint function values. By defining the constraint functions directly, constraint function values are determined independent of any tuning factor. For problems where strong guarantees are required or easier to define, the sufficiently accurate framework will satisfy them by design. An alternative interpretation is that (3.22) provides a principled way of selecting the regularization coefficients. In Section 3.4, we discuss an implementation of a primal dual algorithm to do so.

3.3 SURROGATE DUALITY GAP

Section 3.2.1 introduces an approximation (3.22) to the original problem statement (3.16). An important question to ask is whether solving the approximation will yield a good solution to the original, We are interested in the difference between the primal problem (3.16) and the (3.22).

$$|P^\star - D_N^\star|. \tag{3.24}$$

This is not quite the duality gap introduced in Section 3.1.2 as it is the absolute value of the difference between the primal and the dual of different but closely related problems. Hence, the quantity we are interested in bounding will be denoted as a *surrogate duality gap*.

To provide specific bounds for the difference in the previous expression (3.24), we consider the family of function classes Φ_θ, termed ϵ-universal function approximators. We define this notion next.

Definition 1. The function class Φ_θ is an ϵ-universal function approximator for Φ if, for any $\phi \in \Phi$, there exists a $\phi_\theta \in \Phi_\theta$ such that $\mathbb{E}_{s \sim \mathcal{S}_\mathcal{D}} \|\phi(s) - \phi_\theta(s)\|_\infty \leqslant \epsilon$.

To provide some intuition on the definition consider the case where Φ is the space of all continuous function, the above property is satisfied by some neural network architecture. That is, for any ϵ, there exists a class of neural network architectures, Φ_θ such that Φ_θ is an ϵ-universal approximator for the set of continuous functions [46, Corollary 2.1]. Thus, for any dynamical system with continuous dynamics, this assumption is mild. Other parameterizations, such as Reproducing Kernel Hilbert Spaces, are ϵ-universal as well [128]. Notice that the previous definition is an approximation on the total norm variation and hence it is a milder assumption than the universal approximation property that fully connected neural networks exhibit [46].

Next, we define an intermediate problem on which the surrogate duality gap depends: a perturbed version of problem (3.16) where the constraints are relaxed by $L\epsilon \geqslant 0$ where L is the constant defined in Assumption 3.1 and ϵ the universal approximation constant in Definition 1

$$\begin{aligned} P^{\star}_{L\epsilon} = \min_{\phi\in\Phi} \mathbb{E}_{s\sim\mathcal{S}_{\mathcal{D}}}[\ell_0(s,\phi)] \\ \text{s.t. } \mathbb{E}_{s\sim\mathcal{S}_{\mathcal{D}}}[\mathbf{g}(s,\phi)] + \mathbf{1}L\epsilon \leqslant 0. \end{aligned} \tag{3.25}$$

where $\mathbf{1}$ is a vector of ones. The perturbation results in a problem whose constraints are tighter as compared to (3.16). The set of feasible solutions for the perturbed problem (3.25) is a subset of the feasible solutions for the unperturbed problem (3.16) since $L\epsilon \geqslant 0$. The perturbed problem accounts for the approximation introduced by the parameterization. In the worst case scenario, if the problem (3.25) is infeasible, the parameterized approximation of (3.20) may turn infeasible as the number of samples increases. Let $\lambda^{\star}_{L\epsilon}$ be the solution to the dual of (3.25)

$$\lambda^{\star}_{L\epsilon} = \arg\max_{\lambda\geqslant 0} \min_{\phi\in\Phi} \mathbb{E}[\ell_0(s,\phi)] + \lambda^{\top}(\mathbb{E}[\mathbf{g}(s,\phi)] + \mathbf{1}L\epsilon) \tag{3.26}$$

With these definitions, we can present the main theorem that bounds the surrogate duality gap.

Theorem 1. *Let Φ be a compact class of functions over a compact space such that there exists $\phi \in \Phi$ for which (3.16) is feasible, and let Φ_θ be an ϵ-universal approximator of Φ as in Definition 1. Let the space of Lagrange multipliers, λ, be a compact set as in [92]. In addition, let Assumption 3.1 hold and let Φ_θ satisfy the following property*

$$\lim_{N\to\infty} \frac{H^{\Phi_\theta}(\delta - \epsilon L(\left\|\lambda^{\star}_{L\epsilon}\right\|_1 + 1), N)}{N} = 0 \tag{3.27}$$

where $\left\|\lambda^\star_{L\epsilon}\right\|_1$ *is the optimal dual variable for the problem* (3.26), L *is the Lipschitz constant for the loss function, and* H^{Φ_θ} *is the random VC-entropy [143, section II.B]. Note that both arguments for* H^{Φ_θ} *must be positive. Then* $P^\star$ *and* $D^\star_N$*, the values of* (3.16) *and* (3.22) *respectively, satisfy*

$$\lim_{N\to\infty} \mathbb{P}\left(|P^\star - D^\star_N| \leqslant \delta\right) = 1, \tag{3.28}$$

where the probability is over independent samples $\{s_1, s_2, \ldots, s_N\}$ *drawn from the distribution* $\mathcal{S}$ *as defined in problem* (3.20).

Proof. See Section 3.3.1 □

The intuition behind the theorem is that given some acceptable surrogate duality gap, δ, there exists a neural network architecture, Φ_θ, and a number of samples, N such that the probability that the solution to (3.22) is within δ to the solution to (3.16) is very high. The choice of neural network will influence the value of ϵ and $\lambda^\star_{L\epsilon}$. These in turn will decide the duality gap, δ, as the quantity $\delta - \epsilon L(\left\|\lambda^\star_{L\epsilon}\right\|_1 + 1)$ must be positive. A larger neural network will correspond to a smaller ϵ which will also has an impact on the perturbed problem (3.25). A smoother function and smaller ϵ will lead to smaller perturbations. Smaller perturbations can lead to a smaller dual variable, $\lambda^\star_{L\epsilon}$. Thus, larger neural networks and smoother dynamic systems will have smaller duality gaps. If $L\epsilon$ is large, then the perturbed problem may be infeasible. In theses cases, $\lambda^\star_{L\epsilon}$ will be infinite. This corresponds to problems where the function approximation simply can not satisfy the original constraint functions. For example, using constant functions to approximate a complicated system may violate the constraint functions g_k for all possible constant functions. Thus, no δ exists to bound the solution as the parameterized empirical problem (3.20) has no feasible solution. This theorem suggests that with a good enough function approximation and large enough N, solving (3.22) is a good approximation to solving (3.16) with large probability.

There are some details to point out in Theorem 1. First, the function H^{Φ_θ} a complicated function that will usually scale with the size of the neural network. A larger neural network will lead to a smaller ϵ, but may require a larger number of samples N to adequately converge to an acceptable solution. The assumption on the limiting behavior of H^{Φ_θ} is fufilled by some neural network architectures [9], but the general behavior of this function for all neural network architectures is still a topic of research. Additionally, we assume the space of Lagrange multipliers is a compact set. This will imply, along with compact state-action space, that $\mathcal{L}$ is bounded. A finite Lagrange multiplier is a reasonable assumption as the problem (3.16) is required to be feasible [92].

The bound established in Theorem 1 depends on quantities that are in general difficult to estimate, These include H^{Φ_θ}, $\left\|\lambda^\star_{L\epsilon}\right\|_1$, L, ϵ. Thus, while this theorem provides some insights on how these quantities influence the gap between solutions, it is mainly a statement of the existence of such values that can provide a desired result. In practice, this result can be achieved by choosing increasing the sizes of neural networks as well as data samples until the desired performance is reached. Note that the theorem follows our intuition that larger neural networks and more data will give us more accurate result. However, this theorem formalizes not only that it is more accurate, but that the error will tend to 0 as number of samples and number of parameters increase.

3.3.1 Proof of Surrogate Duality Gap Bound

This section will provide a proof of Theorem 1. To begin, we define an intermediate problem

$$\begin{aligned} P^\star_\theta = &\min_{\theta\in\Theta} \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\ell_0(s,\phi_\theta)] \\ &\text{s.t. } \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\mathbf{g}(s,\phi_\theta)] \leqslant 0. \end{aligned} \tag{3.29}$$

Note that this is the unperturbed version of (3.25). As a reminder, this problem uses a class of parameterized functions, but does not use data samples to approximate the expectation. Thus, it can be seen as a step in between (3.16) and (3.20). As with the dual problem to (3.20), we can define the Lagrangian associated with (3.29)

$$\mathcal{L}_\theta(\theta, \lambda) = \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\ell_0(s, \phi_\theta)] + \lambda^\top \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\mathbf{g}(s, \phi_\theta)] \tag{3.30}$$

and the dual problem

$$D^\star_\theta = \max_{\lambda\geqslant 0} \min_{\theta\in\Theta} \mathcal{L}_\theta(\theta, \lambda). \tag{3.31}$$

Using this intermediate problem, we can break the bound $|P^\star - D^\star_N|$ into two components.

$$\begin{aligned} |P^\star - D^\star_N| &= |(P^\star - D^\star_\theta) + (D^\star_\theta - D^\star_N)| \\ &\leqslant |P^\star - D^\star_\theta| + |D^\star_\theta - D^\star_N| \end{aligned} \tag{3.32}$$

As a reminder, $P^\star$ is the solution to the problem we want to solve in (3.16), $D^\star_N$ is the solution to the problem (3.22) we can feasibly solve, and $D^\star_\theta$ is the solution to an intermediate problem (3.31). The first half of this bound, $|P^\star - D^\star_\theta|$, is the error that arises from using a parameterized function and dual approximation. The second half of this bound, $D^\star_\theta - D^\star_N$, is the error that arises from using empirical data samples. It can be seen as a kind of generalization error. The proof will now be split into two parts that will find a bound for each of these errors.

Function Approximation Error

We first look at the quantity $|P^\star - D^\star_\theta|$. This can be further split as follows

$$\begin{aligned} |P^\star - D^\star_\theta| &= |(P^\star - D^\star) + (D^\star - D^\star_\theta)| \\ &\leqslant |P^\star - D^\star| + |D^\star - D^\star_\theta| \end{aligned} \tag{3.33}$$

where $D^\star$ is the solution to the dual problem associated with (3.16). This is defined with the Lagrangian

$$\mathcal{L}(\phi,\lambda) = \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\ell_0(s,\phi)] + \lambda^\top \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\mathbf{g}(x,\phi)] \tag{3.34}$$

and the dual problem

$$D^\star = \max_{\lambda\geqslant 0} \min_{\phi\in\Phi} \mathcal{L}(\phi,\lambda). \tag{3.35}$$

We note that the quantity $|P^\star - D^\star|$ is actually 0 due to a result from [110, Theorem 1]. The theorem is reproduced here using the notation of this work.

Theorem 2 ([110], Theorem 1)**.** *There is zero duality between the primal problem* (3.16) *and dual problem* (3.35)*, if*

1. *There exists a strictly feasible solution* (ϕ, λ) *to* (3.35)

2. *The distribution* S *is nonatomic.*

While the problem defined in [110] is different from the sufficiently accurate problem defined in 3.16, there is an equivalent problem formulation (see Appendix 6.2). Since Theorem 1 fulfills the assumptions of Theorem 2, we get $|P^\star - D^\star| = 0$.

For the second half of this approximation error, $|D^\star - D^\star_\theta|$, has also been previously studied in [29, Theorem 1] in the context of a slightly different problem formulation. The following theorem adapts [29, Theorem 1] to the *Sufficiently Accurate* problem formulation (3.16).

Theorem 3. *Given the primal problem* (3.16) *and the dual problem* (3.31)*, along with the following assumptions*

1. Φ_θ *is an* ϵ*-universal function approximator for* Φ*, and there exists a strictly feasible solution* ϕ_θ *for* (3.29).

2. *The loss and constraint functions are expectation-wise Lipschitz-continuous with constant* L.

3. *All assumptions of Theorem, 2*

The dual value, $D_\theta^\star$ *is bounded by*

$$D^\star \leqslant D_\theta^\star \leqslant D^\star + (\|\lambda_{L\epsilon}^\star\|_1 + 1)L\epsilon, \tag{3.36}$$

where $\lambda_{L\epsilon}^\star$ *is the dual variable that achieves the optimal solution to* (3.26).

Proof. See Appendix 6.3 □

Again, the assumptions of Theorem 1 fulfill the assumptions for Theorem 3. Due to notational differences, as well as a different way of framing the optimization problem, the proof has been adapted from [29] and is given in Appendix 6.3. With Theorem 2 and 3, the following can be stated

$$|P^\star - D_\theta^\star| \leqslant (\|\lambda_{L\epsilon}^\star\|_1 + 1)L\epsilon \tag{3.37}$$

Empirical Error

We now look at the empirical error, $|D_\theta^\star - D_N^\star|$. We first observe the following Lemma.

Lemma 1. *Let* $\Delta\mathcal{L}(\theta,\lambda) = |\mathcal{L}_\theta(\theta,\lambda) - \mathcal{L}_N(\theta,\lambda)|$. *Then under the assumption of Theorem 1 it follows that*

$$|D_\theta^\star - D_N^\star| \leqslant \sup_{\theta,\lambda} \Delta\mathcal{L}(\theta,\lambda). \tag{3.38}$$

Proof. See Appendix 6.4 □

Probabilistic Bound

Substituting the parameterized bound (3.37) and the empirical bound (3.38) in (3.32) yields the following implication

$$\sup_{\theta,\lambda} \Delta\mathcal{L}(\theta,\lambda) \leqslant \delta - \epsilon L(\|\lambda_{L\epsilon}^\star\|_1 + 1) \Rightarrow \|P^\star - D_N^\star\| \leqslant \delta. \tag{3.39}$$

Let $\mathbb{P}\left(|P^\star - D_N^\star| \leqslant \delta\right)$ be a probability over samples $\{s_1, s_2, \dots, s_N\}$ that are drawn to estimate the expectation in the primal problem (3.20). Using the implication (3.39) it follows that

$$\begin{aligned}\mathbb{P}\left(|P^\star - D_N^\star| \leqslant \delta\right) &\geqslant \mathbb{P}\left(\sup_{\theta,\lambda} \Delta\mathcal{L}(\theta,\lambda) \leqslant \delta - \epsilon L(\|\lambda^\star_{L\epsilon}\|_1 + 1)\right) \\ &= 1 - \mathbb{P}\left(\sup_{\theta,\lambda} \Delta\mathcal{L}(\theta,\lambda) > \delta - \epsilon L(\|\lambda^\star_{L\epsilon}\|_1 + 1)\right),\end{aligned} \tag{3.40}$$

where the equality follows directly from the fact that for any event A, $\mathbb{P}(A) = 1 - \mathbb{P}(A^c)$. The assumptions of Theorem 1 allows us to use the following result from Statistical Learning Theory [143, (Section II.B)],

$$\lim_{N\to\infty} \mathbb{P}\left(\sup_{\theta,\lambda} \Delta\mathcal{L}(\theta,\lambda) > \delta - \epsilon L(\|\lambda^\star_{L\epsilon}\|_1 + 1)\right) = 0. \tag{3.41}$$

Note that this theorem requires bounded loss functions. The assumptions for a bounded dual variable, and compact state-action space in Theorem 1 satisfies this constraint. Thus, this establishes that for any $\delta > 0$, we have $\lim_{N\to\infty} \mathbb{P}\left(|P^\star - D_N^\star| \leqslant \delta\right) = 1$. This concludes the proof of the theorem.

3.4 CONSTRAINED SOLUTION VIA PRIMAL-DUAL METHOD

Section 3.3 has shown that problem (3.22) can approximate (3.16) given a large enough neural network and enough samples. This section will discuss how to compute a solution (3.22). There are many primal-dual methods [37, 35, 36] in the literature to solve this exact problem, and Algorithm 7 is an example of a simple primal-dual algorithm. One way to

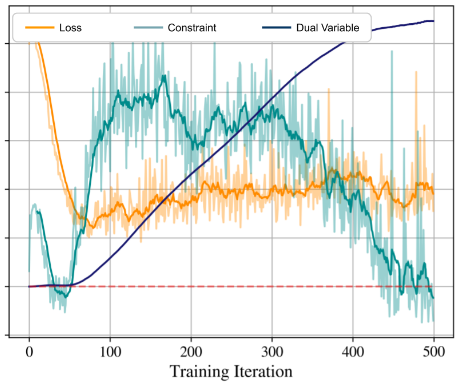


Figure 3.1: (Primal Dual Training Curve) Example of the evolution of the constraint, loss, and dual variables during training. The red dotted line shows 0. The curves have been scaled so that they fit on the same y-axis and they are of different magnitudes. The constraint function is unfeasible, which leads to a growing dual variable. This can cause the loss function to increase until the constraint is feasible again.

approach this problem is to consider the optimal dual variable, $\lambda_N^\star$. Given knowledge of $\lambda_N^\star$, the problem reduces to the following *unconstrained* minimization

$$D_N^* = \min_{\theta \in \Theta} \mathcal{L}_N(\phi, \lambda_N^\star) \tag{3.42}$$

A possible solution method is to start with an estimate of $\lambda_N^\star$, and solve the minimization problem. Then holding the primal variables fixed, update the dual variables by solving the outer maximization. This method can be seen as solving a sequence of unconstrained minimization problems. This method can be further approximated; instead of fully minimizing with respect to the primal variables, a gradient descent step can be taken. And instead of fully maximizing with respect to the dual variables, a gradient ascent step can be taken. This leads to Algorithm 7 where we iterate between the inner minimization step and the outer maximization step. At each iteration, dual variables are projected onto the positive orthant of $\mathbb{R}^K$, denoted by the projection operator, $[\lambda]_+$. This is to ensure non-negativity of the dual variables.

Algorithm 7 Primal Dual

```
1:  procedure PRIMAL-DUAL
2:      Input:
3:          Data Samples, $S = \{s_1, s_2, \dots, s_N\}$
4:          Initial Neural Network parameters, $\theta$
5:          Batch Size, $M$
6:          Primal Learning rate, $\alpha_\theta$
7:          Dual Learning rate, $\alpha_\lambda$
8:      $\lambda = 0$
9:      while Not Converged do
10:         Sample batch of data $\hat{S} = \{s_{i_1}, \dots, s_{i_M}\}$ from S
11:         Use $\hat{S}$ to compute estimates of $\nabla_\theta \mathcal{L}_N(\theta, \lambda)$ and $\nabla_\lambda \mathcal{L}_N(\theta, \lambda)$ (See (3.43) and (3.44))x
12:         $\theta \leftarrow \theta - \alpha_\theta \nabla_\theta \mathcal{L}_N(\theta, \lambda)$
13:         $\lambda \leftarrow \lambda + \alpha_\lambda \nabla_\lambda \mathcal{L}_N(\theta, \lambda)$
14:         $\lambda \leftarrow [\lambda]_+$
15:     end while
16: return $\theta$
17: end procedure
```

In many cases, the full gradient of $\nabla_\theta \mathcal{L}_N(\theta, \lambda)$ and $\nabla_\lambda \mathcal{L}_N(\theta, \lambda)$ can be too expensive to compute. This is due to the possibly large number of samples N. An alternative is to take stochastic gradient descent and ascent steps. The gradients can be approximated by taking M random samples of the whole dataset $S = s_1, \dots, s_N$. The samples will be denoted as $\hat{S} = s_{i_1}, \dots, s_{i_M}$ where i_1 is an integer index into whole dataset S. Using $\hat{S}$, we obtain

$$\begin{aligned}\nabla_\theta \mathcal{L}_N(\theta, \lambda) &= \nabla_\theta [\frac{1}{N}\sum_{i=1}^{N} \ell_0(s_i, \phi_\theta) + \lambda^\top \frac{1}{N}\sum_{i=1}^{N} \mathbf{g}(s_i, \phi_\theta)] \\ &\approx \nabla_\theta [\frac{1}{M}\sum_{j=1}^{M} \ell_0(s_{i_j}, \phi_\theta) + \lambda^\top \frac{1}{M}\sum_{j=1}^{M} \mathbf{g}(s_{i_j}, \phi_\theta)] \\ &= \frac{1}{M}\sum_{j=1}^{M} \nabla_\theta \ell_0(s_{i_j}, \phi_\theta) + \lambda^\top \frac{1}{M}\sum_{j=1}^{M} \nabla_\theta \mathbf{g}(s_{i_j}, \phi_\theta).\end{aligned} \tag{3.43}$$

The gradients $\nabla_\theta \ell_0(s_{i_j}, \phi_\theta)$ and $\nabla_\theta \mathbf{g}(s_{i_j}, \phi_\theta)$ can be computed easily using backpropogation. Similarly, for $\nabla_\lambda \mathcal{L}_N(\theta, \lambda)$,

$$\begin{aligned}\nabla_\lambda \mathcal{L}_N(\theta,\lambda) &\approx \nabla_\lambda [\frac{1}{M}\sum_{j=1}^{M} \ell_0(s_{i_j}, \phi_\theta) + \lambda^\top \frac{1}{M}\sum_{j=1}^{M} \mathbf{g}(s_{i_j}, \phi_\theta)] \\ &= \frac{1}{M}\sum_{j=1}^{M} \mathbf{g}(s_{i_j}, \phi_\theta).\end{aligned} \tag{3.44}$$

The dual gradient can be estimated as simply the average of the constraint functions over the sampled dataset.

In the simplest form of the primal-dual algorithm, the variables are updated with simple gradient ascent/descent steps. These updates can be replaced with more complicated update schemes, such as using momentum [129] or adaptive learning rates [60]. Higher order optimization methods such as Newton's method can be used to replace the gradient ascent and descent steps. For large neural networks, this can be unfeasible as it requires the computation of Hessians with respect to neural network weights. The memory complexity for the Hessian is quadratic with the number of neural network weights.

The primal-dual algorithm presented here is not guaranteed to converge to the global optimum. With proper choice of learning rate, it can converge to a local optimum or saddle point. This issue is present in unconstrained formulations like (3.5) as well. An example of the evolution of the loss and constraint functions is shown in Figure 3.1.

3.5 EXPERIMENTS

This section shows examples of the *sufficiently accurate* model learning problem. First, experiments are performed using a simple double integrator experiencing unknown dynamic friction. The simplicity of this problem along with the small state space allows us to explore and visualize some of the properties of the approximated solution. Next, two more interesting examples are shown. One example learns how a ball bounces on a paddle with unknown paddle orientation and coefficient of restitution. The other example mitigates ground effects which can disturb the landing sequence of a quadrotor. The experiments will compare the sufficiently accurate problem (3.16) with the unconstrained problem (3.5) which will be denoted as the *uniformly accurate* problem. Each experimental subsection will be broken down into three parts, 1) System and Task introduction, 2) Experimental details, and 3) Results.

3.5.1 Double Integrator with Friction

Introduction

To analyze aspects of the *Sufficiently Accurate* model learning formulation, simple experiments are performed on a simple double integrator system with dynamic friction. When trying to control the system, a simple base model to use is that of a double integrator without any friction

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \begin{bmatrix} 0 \\ \Delta t \end{bmatrix} u_t \tag{3.45}$$

where p is the position of the system, v is the velocity, u is the control input, and Δt is the sampling time. The state of the system is $x = [p, v]$.The true model of the system that is unknown to the controller is

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \begin{bmatrix} 0 \\ (u_t \Delta t) - c(v_t, u_t, b(p_t)) \end{bmatrix} \tag{3.46}$$

where $b(p_t)$ a position varying kinetic friction. c is a function that ensures that the friction cannot reverse the the direction of the speed (it is an artifact of the discrete time formulation)

$$c(v_t, u_t, b(p_t)) = \begin{cases} v_t + u_t\Delta t, \text{ if} \\ \quad \text{sign}(v_t + u_t\Delta t) \neq \\ \quad \text{sign}(v_t + u_t\Delta t + b(p_t)) \\ b(p_t), \text{ otherwise.} \end{cases} \tag{3.47}$$

If within a single time step, the friction force will change the sign of the velocity, c will set v_{t+1} to be 0. Otherwise, c will not modify the friction force in any way. The specific $b(p)$ used is shown in Figure 3.2 and the sampling time is set to $\Delta t = 0.1$. The task is to drive the system to the origin $\begin{bmatrix} p & v \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$.

Experimental Details

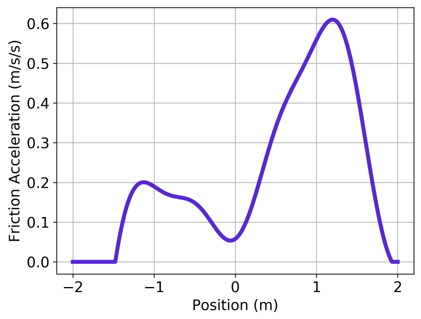


Figure 3.2: (Double Integrator Friction Force) The magnitude of the acceleration due to the position varying kinetic friction force.

The goal of model learning in this experiment is to learn $\phi_\theta(x, u)$ such that $f(x, u) \approx \hat{f}(x, u) + \phi_\theta(x, u)$ where f is (3.46) and $\hat{f}$ is (3.45). A *uniformly accurate* model will be learned using (3.5) along with a *sufficiently accurate* model using the problem defined in Example 3.1.

In the scenario defined by (3.18), $\mathbb{I}(s)$ is active in the region $\{(p,v) \in \mathbb{R}^2 : \left\|[p,v]^\top\right\|_\infty \leqslant 0.5\}$ and $\epsilon_c = 0.035$. The constraint, therefore, enforces a high accuracy in the state space near the origin.

The neural network, ϕ_θ, used to approximate the residual dynamics has two hidden layers. The first hidden layer has four neurons, while the second has two. Both hidden layers use a parametric rectified linear (PReLU) activation [43]. The input into the network is a concatenated vector of $[p_t, v_t, u_t]$. The output layer's weights are initially set to zero so before learning the residual error, the network will output zero. The dataset used to train both the sufficiently and uniformly accurate models is generated by uniformly sampling $15,000$ positions from [-2, 2], velocities from [-2.5, 2.5], and control inputs from [-10, 10]. The real model (3.46) is then used to obtain the true next state. Instead of simple gradient descent/ascent, ADAM [60] is used as an update rule with $\alpha_\theta = 1 \times 10^{-3}$ and $\alpha_\lambda = 1 \times 10^{-4}$. Both models were trained in 200 epochs.

The models are then evaluated on how well it performs within a MPC controller defined in (3.48). This controller seeks to drive the system to the origin while obeying control constraints. The controller is solved using a Sequential Quadratic Programming solver [64, 144] with a time horizon of $T = 10$. The models are evaluated in 200 different simulations where x_{start} is drawn uniformly from $[-2, 2]$.

$$
\begin{aligned}
\min_{\{x_t,u_t\}_{t=1}^{T}} \; & \sum_{t=1}^{T} |x_t| \\
\text{s.t. } & |u_t| \leqslant 10, t = 1, \dots, T \\
& x_{t+1} = \hat{f}(x_t, u_t) + \phi_\theta(x_t, u_t), t = 1, \dots, T-1 \\
& x_1 = x_{start} \\
& \dot{x}_1 = 0
\end{aligned}
\tag{3.48}
$$

Results

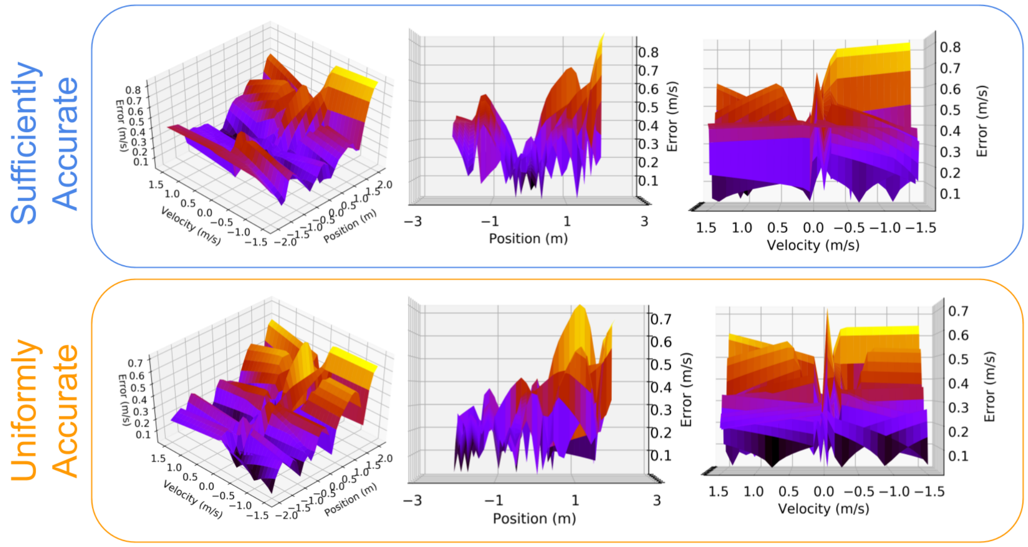


Figure 3.3: **(Double Integrator Model Errors)** The error in predicted velocity of the sufficiently accurate and the uniformly accurate models. There is a constant control input of $u = 1$ used to generate these plots. The top row contains three different views of the error for the sufficiently accurate model, while the bottom row contains the same three views for the uniformly accurate model. The z-axis on each plot is the error in the velocity for the difference $|f(x_t, u_t) - (\hat{f}(x_t, u_t) + \phi_\theta(x_t, u_t))|$. Best viewed in color.

The sufficiently accurate formulation utilizes the prior knowledge that the model should be more accurate near the goal in order to stop efficiently. While the system is far from the origin, the control is simple, regardless of the friction; the controller only needs to know what direction to push in. A plot of the accuracy of both models is shown in Figure 3.3 and summarized in Table 3.2. It is noticeable that the *sufficiently accurate* model has low average error near the origin, but suffers from higher average error outside of the region defined by $\mathbb{I}(s)$. This is the expected behavior.

The performance of the controllers are summarized in Table 3.1. Even though *sufficiently accurate* model has higher error outside of the constraint region and lower error within, it leads to lower costs when controlling the double integrator. The reason is shown in Figure

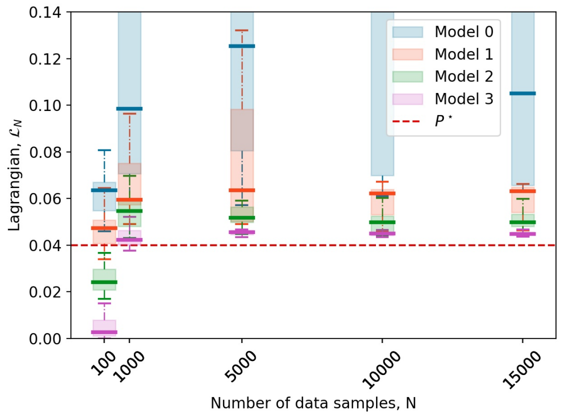


Figure 3.4: (Experimental Surrogate Duality Gaps) Duality gaps of learning the double integrator problem with different neural networks and sample sizes (best viewed in color). The y-axis shows the Lagrangian value at the end of training which approximates $D^\star_N$. The models numbers indicate how large the neural network is with Model 0 being the smallest network. For each N, 15 tests with each model were run, and a box plot is shown that indicates the median as a solid bolded line. The ends of each box are the 1st and 3rd quartile, while the whiskers on the plot are the minimum and maximum values. The red line is the optimal value to the original problem (3.16). Note that for $N = 10000$, the median is not shown for Model 0 as it is very large (0.21).

3.5, where the sufficiently accurate model may get to *steady state* a bit slower but is able to control the overshoot better and not have oscillations near the origin. This is because the model is purposefully more accurate near the origin as it is more important for this task.

Convergence Experiments

The double integrator is a simple system. This enables running more comprehensive tests to experimentally show some aspects of Theorem 1. For this particular system, we will run one more experiment where 4 different neural network architectures were used. Each network has two hidden layers with PReLU activation, where the only difference is in the number of neurons in each layer. Denoting a network as (number of neurons in first layer, number of neurons in second layer), the network sizes used are: (2, 1), (4, 2), (8, 4), (16, 8). A set of values of the number of samples, N, are also chosen: $\{100, 1000, 5000, 10000, 15000\}$.

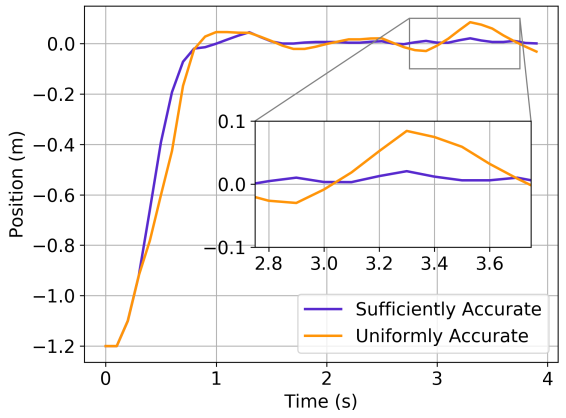


Figure 3.5: (Double Integrator Trajectory) This shows the evolution of the trajectory of the double integrator when controlled using MPC with both a sufficiently accurate and uniformly accurate model.

	$\sum_{t=1}^{T} \lvert x_t \rvert$	$\sum_{t=1}^{T} \lvert x_t \rvert / \lvert x_{start} \rvert$
Uniformly Accurate	5.51 ± 3.46	5.66 ± 0.89
Sufficiently Accurate	$\mathbf{4.73 \pm 3.38}$	$\mathbf{4.57 \pm 0.73}$

Table 3.1: (Double Integrator Controller Performance) This table shows the results of 200 trials of simulating the double integrator starting from different positions. Each entry shows the mean and one standard deviation. The first column shows the raw cost function of the MPC problem averaged over all trials. The second column shows an average normalized cost where each cost is normalized by the absolute value of the starting position. This is due to the fact that larger magnitude starting locations will have higher costs.

For each N, 15 random datasets are sampled, and each neural network is trained with each dataset using the sufficiently accurate objective described in Section 3.5.1. There is one minor difference in how the data is collected; a zero mean Gaussian noise with $\sigma = 0.2$ is added to v_{t+1}. With noisy observations of velocity, the optimal model that can be learned for (3.16) will have an objective value of $P^\star = 0.04$. The results of training each neural network model with each random dataset is shown in Figure 3.4. Each boxplot in the figure shows the distribution of the final value of the Lagrangian, $\mathcal{L}_N$, at the ending of training. This is an approximation of $D_N^\star$. The primal-dual algorithm may not be able to solve for the optimal $D_N^\star$, but the expectation is that for a simple problem like double

	All state space	$\mathcal{I}_K$	$\mathcal{I}_K^C$
Uniform	$\mathbf{0.110 \pm 0.098}$	0.083 ± 0.054	$\mathbf{0.113 \pm 0.10}$
Sufficient	0.136 ± 0.126	$\mathbf{0.048 \pm 0.040}$	0.146 ± 0.13

Table 3.2: **(Double Integrator Model Errors)** The error is the absolute difference between the true model and the learned models. Each entry shows the mean and one standard deviation. The first column shows the average error for the whole state space. The second column shows the average error for the state space near the origin, while the third column shows the average error for the complement of that set (states far from the origin). The errors are evaluated on a test set not seen during training.

integrator, the solution is somewhat close. In fact, Figure 3.4 shows that with increasing model sizes and larger N, the distribution of the solutions appear to be converging to $P^\star$. Note that the figure shows the value of the Lagrangian with training data. Thus for small N, networks can over-fit and have a near zero Lagrangian value. When increasing N, the networks have less of a chance to over-fit to the training data.

3.5.2 Ball Paddle System

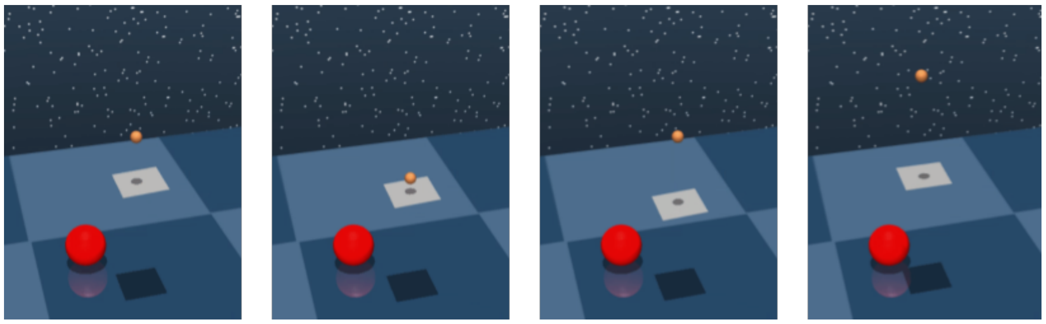


Figure 3.6: **(Ball Bouncing Simulation)** The paddle seeks to bounce the orange ball above a target position on the xy plane, represented by the red ball. This figure shows a time sequence of a bounce from left to right.

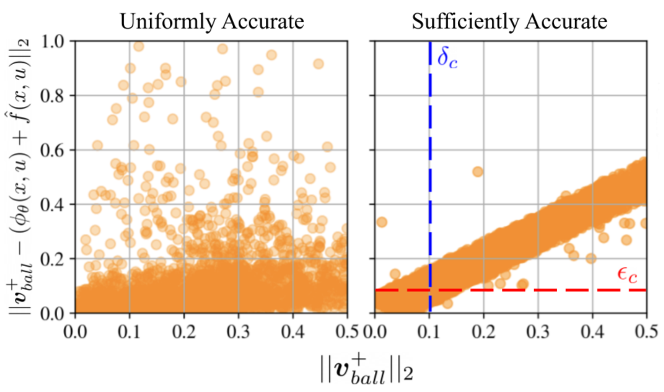


Figure 3.7: (Model Errors vs. Velocity Magnitude) The scatter plot shows the distribution of model errors versus the magnitude of the velocity of the true result. Both the sufficiently and uniformly accurate models are evaluated using a validation set that is not used during training. The blue dotted line represents the boundary of where the constraint set is and the red dotted line represents the boundary of the constraint function.

Introduction

This experiment involves bouncing a ball on a paddle as in Figure 3.6. The ball has the state space $x = [\mathbf{p}_{\text{ball}}, \mathbf{v}_{\text{ball}}]$, where $\mathbf{p}_{\text{ball}}$ is the three-dimensional position of the ball, $\mathbf{v}_{\text{ball}}$ is the three-dimensional velocity of the ball. The control input is $u = [\mathbf{v}_{\text{paddle}}, \mathbf{n}]$ where $\mathbf{v}_{\text{paddle}}$ is the velocity of the paddle at the moment of impact with the ball and $\mathbf{n}$ is the normal vector of the paddle, representing its orientation. This control input is a high level action and is realized by lower level controllers that attempt to match the velocity and orientation desired for the paddle. A basic model of how the velocity of the ball changes during collision is

$$\begin{aligned} \mathbf{v}^+_{\text{ball}} &= \alpha_r(\mathbf{v}^-_{\text{rel}} - 2\mathbf{n}(\mathbf{n}\cdot\mathbf{v}^-_{\text{rel}})) + \mathbf{v}_{\text{paddle}} \\ \mathbf{v}^-_{\text{rel}} &= (\mathbf{v}^-_{\text{ball}} - \mathbf{v}_{\text{paddle}}) \end{aligned} \tag{3.49}$$

where the superscript $-$ refers to quantities before the paddle-ball collision and the superscript $+$ refers to quantities after the paddle-ball collision (the paddle velocity and orientation are assumed to be unchanged during and directly after collision). α_r is the

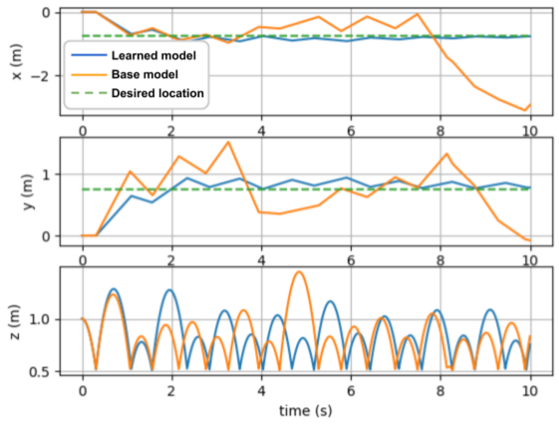


Figure 3.8: (Ball Bouncing Trajectory) The (x, y z) trajectory of the ball is plotted for both the base model (with wrong parameters) and a learned model using the sufficiently accurate objective. This plot shows that the base model is not sufficient by itself to bounce the ball at a desired location.

coefficient of restitution. In this experiment, a neural network is tasked to learn the model of how the ball velocity changes, i.e. (3.49).

Experimental Details

First, a neural network is trained without knowledge of any base model of how the ball bounces. This will be denoted as learning a *full* model as opposed to a *residual* model. This network is trained two ways, with the *uniformly accurate* problem (3.5) as well as the *sufficiently accurate* problem realized in Example 3.2. The constants used in Example 3.2 are defined here as $\epsilon_c = 0.1$ and $\delta_c = 0.1$.

A second neural network is trained for both the uniformly and sufficiently accurate formulations that utilizes the base model, $\hat{f}(x, u)$ given in (3.49) to learn a residual error. In the base model, the coefficient of restitution, α_r, is wrong and the control $\mathbf{n}$ has a constant bias where a rotation of 0.2 radians is applied to the *y-axis*. This is to simulate a robot arm picking up the paddle and not observing the rotation from the hand to the paddle correctly.

The neural network used for all models has 2 hidden layers with 128 neurons in each using the PReLU activation. The input into the network is the the state of the ball and the control input at time of collision, $[x^-, u]$, and it outputs the ball velocity after the collision, $\boldsymbol{v}^+_{ball}$. The network was trained using the ADAM optimizer with an initial learning rate of 10^{-3} for both the primal and dual variables. The data used for all model training was gathered by simulating random ball bounces in MuJoCo for the equivalent of 42 minutes in real life.

All learned models are then evaluated with how well a controller utilizes them. The controller will attempt to bounce the ball at a specific xy location. This is represented through the following optimization problem that the controller solves

$$\begin{aligned} &\min_u |loc(\phi_\theta(x, u)) - loc_{desired}| \\ &\text{s.t. } roll_{min} \leqslant roll \leqslant roll_{max} \\ &\quad pitch_{min} \leqslant pitch \leqslant pitch_{max} \\ &\quad \boldsymbol{v}_{min} \leqslant |\boldsymbol{v}_{rel}| \leqslant \boldsymbol{v}_{max} \end{aligned} \tag{3.50}$$

where $loc(\cdot)$ is a function that maps the velocity of the ball to the xy location it will be in when it falls back to its current height. $roll$ and $pitch$ are both derived from the paddle normal $\boldsymbol{n}$. $[loc_{desired}, roll_{min}, roll_{max}, pitch_{min}, pitch_{max}, v_{min}, v_{max}]$ are parameters of the controller that can be chosen. The system and controller is then simulated in MuJoCo [139] using libraries from the DeepMind Control Suite [136].

Each model is evaluated 500 different times for varying controller parameters. $loc_{desired}$ is uniformly distributed in the region $\{(x, y)| - 1m \leqslant x \leqslant 1m, -1m \leqslant y \leqslant 1m\}$, $\boldsymbol{v}_{min}$ uniformly sampled from the interval $[3m/s, 4m/s)$, and $\boldsymbol{v}_{max}$ is selected to be above $\boldsymbol{v}_{min}$ by between $1m/s$ to $2m/s$.

Results

A plot of the model errors are shown in Figure 3.7. While the uniformly accurate model has errors that are distributed more or less uniformly across all magnitudes of ball velocity, the sufficiently accurate model has a clear linear relationship. This is expected from the normalized objective that is used which penalizes errors based on large the velocity of the ball is. Therefore, larger velocities can have larger errors with the same penalty as smaller velocities with small errors.

The results of running each model with the controller 500 times is shown in Table 3.3. The error characteristics of the *sufficiently accurate* model (Figure 3.7) allow it to out perform its *uniformly accurate* counterpart with both a full model and a residual model. For the full model, the uniformly accurate problem yields a failure rate of over 20% while the sufficiently accurate problem yields a failure rate of under 1%. Here, failure means the paddle fails to keep the ball bouncing. For the residual model, neither model failed because the base model provides a decent guess (though the base model by itself is not good enough for control, see Figure 3.8). The sufficiently accurate model still provided better mean errors.

We hypothesize that the large errors spread randomly across the uniform model leads to high variance estimates of the output given small changes in the input. For optimizers that use gradient information, this leads to a poor estimate of the gradient. For optimizers that are gradient free, this still causes problems due to the high variance of the values themselves.

		Uniform	Sufficient
Full Model	Failure	20.8%	0.8%
	Mean error	0.3136	**0.2124**
Residual Model	Failure	0%	0%
	Mean error	0.164	**0.156**

Table 3.3: **(Ball Bouncing Controller Performance)** Results of 500 trials of ball bouncing for each model. There is a full and residual model trained for both the uniformly accurate and sufficiently accurate model learning problems.

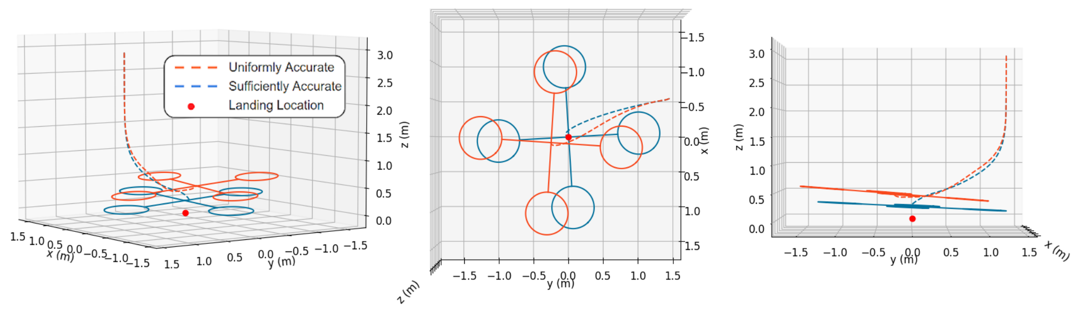


Figure 3.9: **(3D Quadrotor Simulation)** Trajectory of the Sufficiently Accurate and Uniformly Accurate models used in a MPC controller (best viewed in color). The goal is to land at a precise point represented by the red dot. Each plot is a different view of the same data. The dotted lines represent the trajectory of the center of mass of the vehicle for 50 time steps. The state of the quadrotor at the last time step is drawn.

3.5.3 Quadrotor

Introduction

The last experiment deals landing a quadrotor while undergoing disturbances from ground effect. This disturbance occurs when a quadrotor operates near surfaces which can change the airflow [112]. The state for the quadrotor model is a 12 degree of freedom model which consists of $x = [\mathbf{p}, \boldsymbol{\nu}, \mathbf{q}, \boldsymbol{\omega}]$ where $\mathbf{p} \in \mathbb{R}^3$ is position of the center of mass, $\boldsymbol{\nu} \in \mathbb{R}^3$ is the center of mass velocity, $\mathbf{q} \in SO(3)$ is a quaternion that represents the orientation the quadrotor, and $\boldsymbol{w} \in \mathbb{R}^3$ is the angular velocity expressed in body frame. The control input is $u = [u^{(1)}, u^{(2)}, u^{(3)}, u^{(4)}]$ where $u^{(i)}$ is the force from the i^{th} motor. The base model of the quadrotor, $\hat{f}(x, u)$ is as follows

$$\begin{bmatrix} \mathbf{p}_{t+1} \\ \boldsymbol{v}_{t+1} \\ \mathbf{q}_{t+1} \\ \boldsymbol{\omega}_{t+1} \end{bmatrix} = \begin{bmatrix} \mathbf{p}_t + \dot{\mathbf{p}}_t \Delta t \\ \boldsymbol{v}_t + \dot{\boldsymbol{v}}_t \Delta_t \\ \frac{\mathbf{q}_t + \dot{\mathbf{q}}_t \Delta_t}{\|\mathbf{q}_t + \dot{\mathbf{q}}_t \Delta_t\|_2} \\ \boldsymbol{\omega}_t + \dot{\boldsymbol{\omega}}_t \Delta_t \end{bmatrix}$$

$$\begin{bmatrix} \dot{\mathbf{p}} \\ \dot{\boldsymbol{v}} \\ \dot{\mathbf{q}} \\ \dot{\boldsymbol{\omega}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{v} \\ \mathbf{q} \otimes [0, 0, \sum_{i=1}^{4} u^{(i)}/m]^\top \otimes \mathbf{q}^{-1} - [0, 0, 9.81]^\top \\ \frac{1}{2} \boldsymbol{\omega} \otimes \mathbf{q} \\ \mathcal{J}^{-1}(\mathbf{T} - \boldsymbol{\omega} \times (\mathcal{J}\boldsymbol{\omega})) \end{bmatrix} \tag{3.51}$$

$$\mathbf{T} = \begin{bmatrix} u^{(4)} - u^{(2)} \\ u^{(3)} - u^{(1)} \\ (u^{(1)} + u^{(3)}) - (u^{(2)} + u^{(4)}) \end{bmatrix}$$

where m is the total mass of the quadrotor (set to be 1kg for all experiments) and $\mathcal{J}$ is inertia matrix around the principle axis (set to be identity for all experiments). The $\times$ symbol represents cross product, and $\otimes$ represents quaternion multiplication. When using $\otimes$ between a vector and a quaternion, the vector components are treated as the imaginary components of a quaternion with a real component of 0. The discrete model normalizes the quaternion for each state update so that it remains a unit quaternion. The body frame of the quadrotor is such that the x axis aligns with one of the quadrotor arms, and the z axis points "up."

The true model used in simulation adds disturbances to the force on each propeller, but is otherwise the same as the base model:

$$f(x,u) = \hat{f}(x, h(x,u)) \tag{3.52}$$

where $h : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^p$ is the ground effect model. In this experiment we provide a simplified model of ground effects where each motor has independence disturbances. The i^{th} output of the ground effect model, $h_i(x,u)$ is

$$\begin{aligned} h_i(x,u) &= u^{(i)}(1 + K_{ground}) \\ K_{ground} &= ([1 - \frac{h_{prop}}{h_{max}}]_+)(\frac{4[\theta_{ground} - \frac{\pi}{2}]_+^2}{\pi^2})\alpha \end{aligned} \tag{3.53}$$

where h_{prop} is height of the propeller above the ground (not the height of the center of mass), h_{max} is a constant that determines the height at which the ground effect is no longer in effect. θ_{ground} is the angle between the unit vector aligned with the negative z axis of the quadrotor and the unit vector $[0,0,-1]$. α is a number in the set $[0,1]$ that represents the maximum fraction of the propeller's generated force that can be added as a result of ground effect. As a reminder, the $[\cdot]_+$ operator projects its arguments onto the positive orthant. A visualization of K_{ground} is shown in Figure 3.10. In the experiments, $h_{max} = 1.5$, $\alpha = 0.5$.

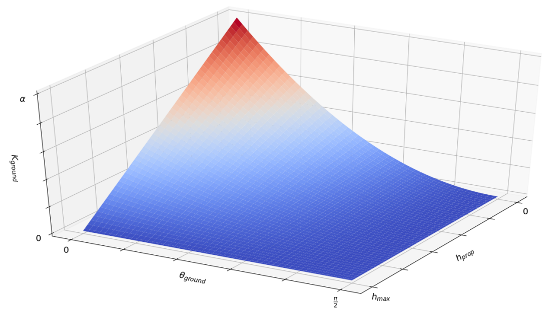


Figure 3.10: (Ground effect) A visualization of K_{ground} as a function of θ_{ground} and h_{prop}.

Experimental Details

The *Sufficiently Accurate* model trained using the problem presented in Example 3.1, where $\epsilon_c = 0.001$ and the indicator $\mathbb{I}(s)$ is active when the height of the quadrotor is less than 1.5. A *uniformly accurate* and *sufficiently accurate* model are trained to learn the residual error between $f(x, u)$ and $\hat{f}(x, u)$. Both models use a neural network with 2 hidden layers of 16 and 8 neurons each with PReLU activation. The update for primal and dual variables used ADAM with $\alpha_\theta = 1 \times 10^{-3}$ and $\alpha_\lambda = 1 \times 10^{-4}$, and both models trained using $3,000$ epochs. The training data consists of $10,000$ randomly sampled quadrotor states. The (x, y) positions of the quadrotor were uniformly sampled from $[-10, 10]$. The z positions of the quadrotor were uniformly sampled from $[0.1, 5]$. Linear velocities components were uniformly sampled from $[-6, 6]$. Angular velocities components were uniformly sampled from $[-2.5, 2.5]$. Control inputs for each motor are sampled from $[0, 10]$. Quaternions are sampled by sampling random unit vectors along with a random angle in $[-0.4rad, 0.4rad]$. This angle-axis rotation is transformed into a quaternion.

Both models are tested by sampling a random starting location and asking the quadrotor to land at the origin. The controller used for landing is an MPC controller that repeatedly solves the following problem

$$
\begin{aligned}
&\min_{\{u_t, x_t\}_{t=1}^{T}} \sum_{t=1}^{T} \|\mathbf{p}_t - \mathbf{p}_{target}\|_2 \\
&\text{s.t. } 0 \leqslant u_t \leqslant 10, \forall t \\
&\quad x_{t+1} = \hat{f}(x_t, u_t) + \phi_\theta(x_t, u_t), t = 1, \dots, T-1 \\
&\quad h_{t,com} \geqslant 0.2, \forall t \\
&\quad |q_w - 1| \leqslant 0.05
\end{aligned}
\tag{3.54}
$$

	Sufficiently Accurate	Uniformly Accurate
MPC cost	**4.21 ± 1.88**	4.55 ± 1.90
$\mathbb{E}[l(s, \phi_\theta)]$	6.52×10^{-4}	**6.16×10^{-4}**
$\mathbb{E}[g(s, \phi_\theta)\mathbb{I}(s)]$	**1.00×10^{-4}**	8.28×10^{-4}

Table 3.4: **(Quadrotor Results)** The first row shows the MPC cost average (3.54) and one standard deviation over 5 runs. The second row shows the training loss, while the third row shows the constraint function. The expected errors are evaluated on a test set not seen during training.

where $\mathbf{p}$ is the position of the quadrotor, $h_{t,com}$ is the height of the center of mass, and q_w is the real component of the quaternion at the last time step. This problem encourages reaching a target, subject to control and dynamics constraint. It also has a constraint on the height of the quadrotor so it is always above a certain small altitude, and an orientation constraint on the last time step so it is mostly upright when it lands.

Results

The results of running this controller over several different starting locations is shown in Table 3.4. Similar to previous experiments, the *sufficiently accurate* model has a higher loss overall, but better accuracy in the constrained area which is more important to the task. This allows the controller to utilize the higher accuracy to land the quadrotor precisely. An example of one of the landing trajectories is shown in Figure 3.9. It can be seen that the sufficiently accurate model can more precisely land at the origin $(0, 0)$. It also is able to reach the ground faster, as it can more accurately compensate for the extra force caused by the ground surface. The ground effect can also disturb roll and pitch maneuvers which can offset the center of mass as well.

3.6 conclusion

This chapter presented *sufficiently accurate* model learning as a way to incorporate prior information about a system in the form of constraints. In addition, it proves that this constrained learning formulation can have arbitrarily small duality gap. This means that existing methods like primal-dual algorithms can find decent solutions to the problem.

With good constraints, the model and learning method can focus on important aspects of the problem to improve the performance of the system even if the overall accuracy of the model is lower. These constraints can come from robust control formulations or from knowledge of sensor noise. Some important questions to consider when using this method is how to choose good constraints. For some systems and tasks, it can be simple while for others, it can be quite difficult. This objective is not useful for all tasks and systems but rather for a subset of tasks and systems where prior knowledge is more easily expressed as a constraint.

4 SEMI-AUTOMATIC CONSTRAINT GENERATION

Chapter 3 introduced the idea of *sufficiently accurate* model learning where introducing constraints in the model learning formulation can improve the performance of the planning algorithms that utilize it. These constraints come from human prior knowledge and can require expert insight. This chapter seeks to lower the barrier of entry for generating constraints by introducing a method to semi-automatically generate them. Human prior knowledge will still be required, but will not be in the form of specific model constraints. Instead, the prior comes in the form of knowing when a system has failed or is close to failure as well as how to create a constraint from a failure mode. Often times, this is easier to define that explicit constraints.

There is a trade off between using expert knowledge and performance. If a human expert is infinitely knowledgeable and had infinite time, they might be able to choose the most optimal constraints for a model learning problem for a specific system. This can result in great performance at the cost of large amount of human labor and expertise. On the other hand, if an automatic constraint generation method uses very general principles to learn constraints, it will not be able to exploit system specific information. This can lead to inefficiency in the learning process and result in poor models, large data requirements, or large training times but can save human labor. There is a spectrum that trades off computational time, effort, and performance with human time and effort. For many problems, a middle ground between purely human designed heuristics and no human intervention exists, and a method for semi-automatic constraint generation provides a method to find tradeoff point.

Another aspect to consider in purely human designed priors is the possibility of mistakes. An invalid constraint can reduce performance of a system rather than enhance it. A semi-automatic constraint learning system can require less knowledge which might in turn lead to less mistakes.

An analogue to directly human created constraints for model learning are *expert systems* [84], a framework for automatic decision making based on human given knowledge with computer inference. The performance of the system is very dependent on the strength of the expert human knowledge. On the other end of the spectrum, a completely unconstrained model learning problem is akin to early fully connected neural networks. It had very little prior knowledge about the task desired. All knowledge is derived from the data observed. The goal of semi-automatic constraint learning is to provide a framework such as a Convolutional Neural Network. This introduces some prior (convolutions) for tasks like image categorization but still relies on data. This approach can outperform a purely data driven approach [74] and may require less human effort than an expert system.

The method we propose requires a motion planning algorithm, the ability to test motion plans with a learned model, and a variable amount of human knowledge. A learned model will be trained from gathered data and tested on the real system. Human determined failure points will stop the system and generate constraints with the failure data. Then, this process will repeat. The iterative generation process allows constraints to be refined with more data and adapted as the model itself changes.

Section 4.1 will discuss previous work in machine learning and robotics for constraint learning. Section 4.2 will formulate the problem to be solved and Section 4.2.1 will present a framework to solve it. Experiments on two types of systems will be presented in Section 4.3.

4.1 BACKGROUND

This section will review methods with a similar goal to semi-automatic constraint learning for models. We will discuss constraint learning techniques in general machine learning algorithms as well as more specific work in motion planning.

4.1.1 Constraint Learning

The underpinning of many machine learning algorithms utilizes constrained optimization. Constrained problems can be formulated for classification [145], segmentation [96], regression [121] and appears in many classical machine learning techniques such as Support Vector Machines [133]. It is of interest in many situations to learn the constraints in such problems from a labeled dataset of inputs and outputs of an unknown optimization problem. This is known as *constraint learning* [25] or *inverse optimization* [3, 19]. Constraints can induce bias in the solution, and learning the constraints can hopefully introduce useful biases that can lead to more general solutions [89]. The literature in this field generally focuses on learning the parameters of an objective and constraint function [1, 135]. For example, instead of the optimization problem presented in (3.16).

$$\begin{aligned} P^\star = \min_{\phi \in \Phi} \; & \mathbb{E}_{s \sim \mathcal{S}_\mathcal{D}}[\ell(s, \phi)\mathbb{I}_0(s)] \\ \text{s.t. } & \mathbb{E}_{s \sim \mathcal{S}_\mathcal{D}}[g_k(s, \phi)\mathbb{I}_k(s)] \leqslant 0, k = 1, 2, \ldots, K, \end{aligned}$$

the loss function ℓ and constraint functions g_k might have additional parameters $\omega_\ell, \omega_{g_k}$. An example of a parameterized loss function is a Mahalanobis distance [86] $\ell(s, \phi, \omega_\ell) = (x_{t+1} - \phi(x_t, u_t))^\top \Sigma_{\omega_\ell}(x_{t+1} - \phi(x_t, u_t))$. Then presented, with a set of N pairs of distributions and solution models, $(\mathcal{S}_{\mathcal{D},i}, \phi_i)_{i=1}^{N}$, the inverse optimization problem can be written as the following bi-level optimization problem [135]

$$
\begin{aligned}
\min_{\omega_l, \omega_{g_k}} \quad & \frac{1}{N} \sum_{i=1}^{N} L(\phi_i, \phi_i^*) \\
\text{s.t.} \quad & \mathbb{E}_{\mathcal{S}_{\mathcal{D},i}}[g_k(s, \phi_i, \omega_{g_k}) \mathbb{I}_k(s)] \leqslant 0, k = 1, 2, \dots K, i = 1, 2, \dots, N \\
& \phi_i^* = \begin{cases} \arg\min_{\phi} \mathbb{E}_{s \sim \mathcal{S}_{\mathcal{D},i}}[\ell(s, \phi, \omega_\ell) \mathbb{I}_0(s)] \\ \text{s.t.} \quad \mathbb{E}_{s \sim \mathcal{S}_{\mathcal{D},i}}[g_k(s, \phi, \omega_{g_k}) \mathbb{I}_k(s)] \leqslant 0, k = 1, 2, \dots, K. \end{cases}
\end{aligned}
\tag{4.1}
$$

where L is a function that penalizes the case when $\phi_i \neq \phi_i^*$. (4.1) describes a problem to find the parameters $\omega_\ell, \omega_{g_k}$ such that the solution of the inner optimization problem matches closely to the given dataset solutions, ϕ_i, while also ensuring that the dataset solutions match the constraint functions that are optimized over. When the problem to optimize over is a linear program, there are various solutions [3, 135]. [135] uses Sequential Quadratic Programming to optimize the bi-level optimization. The inner optimization problem is solved using any typical linear program solver, and the gradients are obtained by differentiating through the Karush–Kuhn–Tucker (KKT) conditions [15] at the solution [7]. A similar approach to learning general convex problems is given in [1].

4.1.2 Learning Trajectory Constraints

Section 4.1.1 discussed constraint learning in general machine learning context, however there has also been work on constraint learning specifically for trajectory generation or motion planning. [2, 8] uses the a similar approach as [7] but applies it to the specific problem of Model Predictive Control. The goal is to be able to learn dynamic constraints or cost functions with respect to some function of the MPC solution. This can be used, for example, to learn the dynamics model of a robot given trajectories that follow some presumable optimal path or to learn the parameters of cost and constraint functions to match the style of some demonstrated motion.

[99] learns constraints for robotic arm grasping tasks by using a database of demonstrations. Keyframes of the demonstrations are clustered, and human prior knowledge is used to infer geometric constraints from the clustered keyframes. Constraints on pose or orientation are established based on human tuned heuristics that depend on geometric quantities in a keyframe cluster. This allows certain tasks to learn constraints from a handful of demonstrations which are then used for planning. The goal is that these constraints more effectively capture the essential quantities of importance from the demonstrations. [22] also learns constraints from a set of demonstrations. The method assumes the demonstrations come from an optimal planner with some known constraints and hopes to discover unknown constraints. A planner is then used to generate lower cost solutions than the demonstrations. These low cost solutions are assumed to violate the unknown constraints as the assumption is the demonstrations should obtain the lowest costs already. These low cost solutions can be utilized to detect the unknown constraints.

4.2 CONSTRAINT GENERATION

The goal of semi-automatic constraint generation is given 1) a motion planner, 2) the ability to test motion plans on the system, and 3) some human prior information, find constraints $g_k : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$ such that a model can perform well on a set of N planning tasks. This can be expressed as a tri-level optimization problem

$$
\begin{aligned}
\min_{g_k} \quad & \frac{1}{N}\sum_{i=1}^{N} c_i \\
\text{s.t.} \quad & c_i = \begin{cases} \min_{u_{1:T}} & \mathtt{planning\_cost}_i(x_{1:T}, u_{1:T}) \\ \text{s.t.} & \mathtt{planning\_constraints}_i(x_{1:T}, u_{1:T}) \leqslant 0 \\ & x_{t+1} = \phi_\theta(x_t, u_t), t = 1, \dots, T-1 \end{cases} \\
& \theta = \begin{cases} \arg\min_\theta & \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[\ell(s,\phi)\mathbb{I}_0(s)] \\ \text{s.t.} & \mathbb{E}_{s\sim\mathcal{S}_\mathcal{D}}[g_k(s,\phi)\mathbb{I}_k(s)] \leqslant 0, k = 1, 2, \dots, K \end{cases}
\end{aligned}
\tag{4.2}
$$

where $\mathtt{planning\_cost}_i$ and $\mathtt{planning\_constraints}_i$ are the cost function and constraints for the i^{th} planning task. The goal of this problem is to find good *model constraints*, g_k such that a *sufficiently accurate* model from Section 3.2 will do well on a set of tasks. This can be seen as learning a set of useful biases for the model to improve its functionality and bears some resemblance to the *experienced piano mover's problem* discussed in Chapter 2. As with Chapter 3, the constraints can empower resource constrained function approximators to trade off where their expressive power can be utilized best for the set of tasks.

Consider the concrete example of planning and controlling a vehicle on an unevenly icy surface. Learning a dynamics model in this environment can help predict the hard to model friction. A set of tasks in this environment might take the form of traveling from one corner of the map to the other from various starting and goal locations. The problem of constraint generation is to identify constraints for model learning such that the task of traveling across the map is better fulfilled.

While (4.2) looks similar to a inverse optimization problem (4.1), there are a few key distinctions. First, there is an additional level of indirection. In (4.1), a planning problem can be formulated, and the *planning constraints* are optimized [1–3, 8, 135]. In (4.2), a model learning problem is formulated and *model constraints* are optimized to best suit an inner set of planning problems. Second, the planning problems considered in [2] are convex and known to have zero duality gap, thus the inner problems can be implicitly differentiated through the KKT conditions. For general model learning, the convexity of the inner problem does not hold.

4.2.1 Methodology

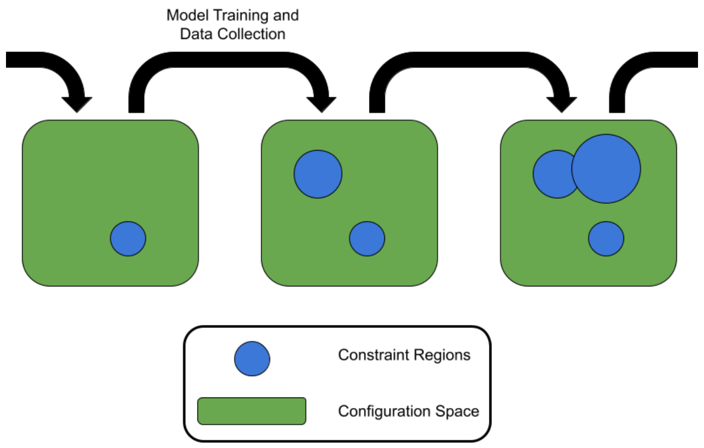


Figure 4.1: **(Semi-automatic Constraint Generation)** The semi-automatic constraint generation process iteratively adds and updates constraints with new data collected.

Instead of directly solving (4.2), the approach will be to incorporate human heuristics into finding good constraints. We incorporate these human heuristics into a standard model learning pipeline.

Model learning in robotics is often done in an iterative way [149]. Data is collected from the system, the a model is learned from the data, then the new model is used to collect data. This iterates until a desired level of performance is achieved. The reason for this iterative process rather than a one time data collection is due to *distribution shift*. Often times, the distribution of states and actions that the loss function optimizes over, $\mathcal{S}_{\mathcal{D}}$ is dependent on the model itself. $\mathcal{S}_{\mathcal{D}}$ can represent the distribution of state, action, next state tuples induced by using the learned model to plan on a set of tasks. The objective is to have low error on the dataset that is actually experienced by the model. Thus, a one time data collection may not accurately reflect the real distribution of data that is dependent on the model itself. A form of this problem is addressed by [111], which suggests that adding new data to the dataset collected from the model can converge to a steady state distribution. Thus, model learning for a robotic task generally appears in the form of Algorithm 8 where the red lines are modified or added for the constrained version that this work proposes.

Algorithm 8 Constrained Model Learning with Semi-automatic Constraint Generation

```
 1: procedure LEARN-CONSTRAINED-MODEL
 2:     Initialize $\theta \in \Theta$
 3:     Initialize dataset $D = \emptyset$
 4:     while Not Converged do
 5:         Use model $\phi_\theta$ to collect data tuples, $D_i = (x_t, u_t, x_{t+1})$
 6:         $D \leftarrow D \cup D_i$
 7:         GENERATE-CONSTRAINTS$(D_i, \phi_\theta)$ to formulate a *sufficiently accurate* model
    learning problem (3.16).
 8:         $\theta \leftarrow$ PRIMAL-DUAL$(\theta, D)$
 9:     end while
10: end procedure
```

Algorithm 9 Constraint Generation

```
1: procedure GENERATE-CONSTRAINTS
2:     for $(x_t, u_t, x_{t+1})$ in D do
3:         if FAILURE$(\phi_\theta(x_t, u_t))$or FAILURE$(x_{t+1})$ then
4:             CREATE-CONSTRAINTS()
5:             UPDATE-CONSTRAINTS()
6:         end if
7:     end for
8: end procedure
```

This work proposes using the iteratively generated data to create task specific constraints and then solve the constrained problem that arises as shown in Figure 4.1. Similar to how the iterative data collection and training process can fix the distributional shift, the iterative constraint generation can also adapt the constraints to be useful to the task at hand. As the model is improved, constraints are added to only the areas that further improve the planner's capabilities. The GENERATE-CONSTRAINTS function is task specific and will output both a constraint function, g_k as well as a set over the state-action space denoted by the indicator variable $\mathbb{I}_k$. It is where human knowledge is brought in to heuristically solve (4.2). In this work, the constraints considered are of the form

$$\mathbb{E}[\|x_{t+1} - \phi_\theta(x_t, u_t)\|_p \, \mathbb{I}_k(x_t, u_t)] \leqslant \epsilon_k \tag{4.3}$$

which constrains the expected norm of the error in a region defined by $\mathbb{I}_k$ to be less than ϵ_k. The two tunable parameters of this constraint will be $\mathbb{I}_k$ and ϵ_k. The constraints can be created by Algorithm 9 which require several task and system specific functions to be defined: FAILURE, CREATE-CONSTRAINTS, and UPDATE-CONSTRAINTS.

1. FAILURE(x) is a function that takes the state of a system and decides if it has failed or is close to failure. This is what triggers a constraint region to be generated.
2. CREATE-CONSTRAINTS is a function that will generate the constraint based on which state failed and any additional human knowledge that can be incorporated in.
3. UPDATE-CONSTRAINTS is a function that updates the constraints given new data. For example, if a the planner keeps creating paths that go through a constraint region, but the trajectory keeps failing, it could indicate that the constraint needs to be tightened. This function may be less system specific but require choosing

the correct heuristics for the task at hand. Common heuristics will be discussed in Section 4.2.2.

4.2.2 Update Constraint Heuristics

This section will discuss two heuristics that can be used in the UPDATE-CONSTRAINTS function from Algorithm 9. The goal of this function is to adapt the constraint parameters in (4.3) from already created constraint regions to new data.

Constraint Tightening

The first heuristic is simple. It relies on a counter for each constraint region. This counter will be incremented whenever a path that goes through the constraint region ends up triggering the FAILURE function later on. When the counter is high enough, the counter will be reset and the error bound, ϵ_k will be scaled down to $\alpha\epsilon_k, \alpha \in (0, 1)$. The goal of this heuristic is to tighten constraints that were initially decided by the CREATE-CONSTRAINTS function if too many paths that go through the constraint region ultimately end up failing. The scaling parameter α should be chosen conservatively as decreasing the constraint bound too fast may give too much importance to some regions. It should also be noted that the counter increment should take place only if the constraint $\mathbb{E}[\|x_{t+1} - \phi_\theta(x_t, u_t)\|_p \, \mathbb{I}_k(x_t, u_t)]$ is close to the constraint bound, ϵ_k. The constraint should not be adjusted if the model learning has not yet achieved the constraint desired, as the failure counter does not properly reflect the status of current constraint parameters.

Region Merging

The second heuristic will merge constraint regions that are close. When two constraint regions, $\mathbb{I}_k$ and $\mathbb{I}_{k+1}$, have large overlap and similar constraint bounds, ϵ_k, a new region that encompasses both can be created to replace them. The new region will use the lower constraint bound. This process allows the number of constraints to remain low if a large number of small constraint regions are created. Checking overlap between regions can be a nontrivial computation. It depends on how the regions are defined. Thus, this heuristic is only applicable in cases where computing region overlap and the new region shape is computationally efficient.

4.3 EXPERIMENTS

This section will present results on experiments utilizing model learning with semi-automatic constraint generation as described in Algorithm 8. All experiments will compare the constrained model against a model learned without using any constraints. There are two systems tested: a 2D simulated rocket and a kinematic bicycle model. All constraints used will be of the form

$$\|x_{t+1} - \phi_\theta(x_t, u_t)\|_2^2 \, \mathbb{I}_k(x_t, u_t) \leqslant \epsilon_k \tag{4.4}$$

Each experimental section will consist of three components: a description of the system and problem setup, a description of the experimental details including the system specific functions as described in Section 4.2.1, and a description of the results and analysis.

4.3.1 Simple Rocket Model

System Description

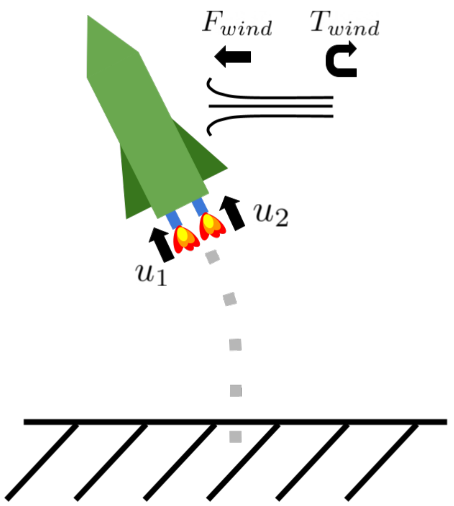


Figure 4.2: (Rocket System) Simple 2D model of a rocket that has two thrusters and is buffeted by winds that apply lateral force and torque.

This system simulates a simple rocket in a plane with two thrusters that allows the rocket to control its lift and attitude as shown in Figure 4.2. There is also an external disturbance of the wind which will cause a lateral force to the rocket as well as a torque. The strength of the wind depends on the height of the rocket.

The state of the rocket system is a 6 dimensional vector $x = [px, py, \theta, \dot{px}, \dot{py}, \dot{\theta}]^\top$, where (px, py) represents the coordinates of the location of the rocket and θ is the heading. $\theta = 0$ represents the rocket pointing straight up, and a positive theta corresponds with counter-clockwise rotation. The two thrusters output forces $u = [u1, u2]^\top$ as the control inputs in the range $[0, 1]$. The dynamics model of the rocket with the external wind disturbance (in red) is

$$\dot{x} = \hat{f}(x, u) + \text{disturbance}(x, u)$$
$$= \begin{bmatrix} \dot{px} \\ \dot{py} \\ \dot{\theta} \\ -\frac{u1+u2}{m}\sin(\theta) \\ \frac{u1+u2}{m}\cos(\theta) - g \\ \frac{B(u2-u1)}{I_{moment}} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{F_{wind}}{m} \\ 0 \\ \frac{T_{wind}}{I_{moment}} \end{bmatrix} \tag{4.5}$$

where $m = 0.1$ kg is the mass, $B = 0.25$ m is half the base width, and $I_{moment} = 0.417\text{kg m}^2$ is the moment of inertia. A discretized model using Euler integration with time step $\Delta t = 0.1$ s is used for simulation. The disturbance functions, F_{wind} and T_{wind} are shown in Figure 4.3.

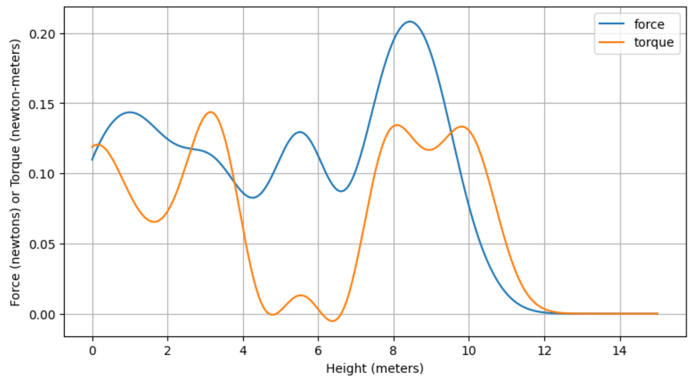


Figure 4.3: **(Rocket Wind Disturbance)** Force and Torque caused by wind disturbance as a function of the height of the rocket.

The task for the rocket is to land safely despite the buffeting winds. A optimization based controller is formulated

$$\begin{aligned}
\min_{\{x_t\}_{t=1}^{T},\{u_t\}_{t=1}^{T-1}} \quad & \alpha_{goal}(x_T - x_{goal})^\top C(x_T - x_{goal}) + \alpha_{control}\sum_{t=1}^{T-1} \|(\| u_t) \\
\text{s.t.} \quad & \|u_t\|_\infty \leqslant u_{max}, \forall t \\
& py_t \geqslant 0, \forall t \\
& |\theta_t| \leqslant \frac{\pi}{4} - \epsilon_\theta, \forall t \\
& x_1 = x_{start}, (px_T, py_T) = p_{goal}
\end{aligned} \tag{4.6}$$

which tries to land the rocket at the goal with minimal control effort and avoid crashing or tilting at too large an angle. p_{goal} is simply the location component of x_{goal}.

Experimental Details

The experiments compare two neural network models. Both are residual error models that attempt to learn the $\mathtt{disturbance}$ component of the dynamics model in (4.5). One model is learned using Algorithm 8, while the other learned using the same iterative process but with no constraints. Both models are neural networks with two hidden layers of 16 and 8 neurons. Each hidden layer is followed by a PReLU activation [43]. All weights and biases of the output layers are initialized to zeros. The models are all trained using the ADAM optimizer with a learning rate of $1e-4$. The constrained model updated the dual variables with ADAM as well with a learning rate of $5e-4$.

The system specific functions for the rocket system are as follows:

1. FAILURE: the system is in a failure mode if the height is below 0 or the absolute value of angle of the rocket exceeds $\pi/4$

2. CREATE-CONSTRAINTS: A constraint region is generated around the failure state that includes all $(px, \dot{px}, \dot{py})$ values. $(py, \theta, \dot{\theta})$ values within a radius of $(1.5, \pi/4, 3)$ are included. The error bound ϵ_k is set to be half of the current error in that region.

3. UPDATE-CONSTRAINTS: The rocket system uses the *constraint tightening* heuristic discussed in Section 4.2.2 with $\alpha = 0.5$.

The training procedure for both the constrained and unconstrained models consists of using an initial dataset of 60,000 data tuples to train an initial unconstrained model. Then, 4 iterations of collecting data and training the model. Each data collection consisted of obtaining 32 trajectories using a receding time horizon controller that solves (4.6) with $T = 30$, $x_{goal} = 0$, $C = diag(0.1, 0.1, 0.05, 1, 1, 1)$, $\alpha_{goal} = 1$, $\alpha_{control} = 0.01$, $u_{max} = [1, 1]^{\top}$. Each model is trained for 150 epochs between data gathering. After all the constraints are gathered, the models are finetuned for a further 700 epochs to facilitate constraint satisfaction.

Results and Analysis

The constraint generation procedure generated two constraints centered around a mean with error ϵ_k shown in Table 4.1. Using a validation dataset, the results of the trained model are shown in Table 4.2. The first three rows show the squared model error over the entire dataset as well as on each constraint region. Notice that though the constrained model has worse error overall, it is more accurate in the constraint regions. The error in the constraint regions in Table 4.3 do not perfectly satisfy the constraints bounds given in Table 4.1 because of two reasons: 1) a validation dataset is used to evaluate the errors and 2) the primal-dual training process does not guarantee convergence to a feasible solution.

After both a constrained and unconstrained model is trained, the models are tested by running the planner described in (4.6) over 200 novel starting states. The planner is used to generate controls in the style of a Model Predictive Controller; A plan is generated and several time steps of the control are used, and then the planner replans. The starting x location is uniformly sampled from $[-4, 4]$, the starting y location is sampled from $[4, 10]$, and the starting angle is sampled from $[-\pi/6, \pi/6]$. The starting velocities are all set to 0. The failure rate of the constrained and unconstrained model are shown in the last row of Table 4.2. Despite having higher overall model error, the constrained model obtains a slightly lower failure rate.

	Mean $(py, \theta, \dot{\theta})$	ϵ_k
Constraint 0	$(2.312, 0.731, 0.359)$	5.7244e-7
Constraint 1	$(0.243, 0.918, 0.639)$	2.5523e-7

Table 4.1: **(Rocket Constraint Regions)** The two constraint regions generated for the rocket system through Algorithm 8. The second column shows the mean of constraint region, while the third column shows the constraint bound.

Metric	Unconstrained Model	Constrained Model
Entire state-action space ℓ	2.534e-6	3.369e-6
Constraint 0: $\mathbb{E}[g_0 \mathbb{I}_0]$	2.284e-6	5.995e-7
Constraint 1: $\mathbb{E}[g_1 \mathbb{I}_1]$	2.084e-6	3.173e-7
Failure Rate	0.875	0.840

Table 4.2: **(Rocket Experiment Results)** This table records the error of the model in various subsets of the dataset as well as the failure rate of running a closed loop planner 200 times on problems with new starting states.

4.3.2 Bicycle Model

System Description

The kinematic bicycle model, as shown in Figure 4.4, is a common model used to simulate higher fidelity car models as it can closely approximate them [105]. The state consists of $x = [px, py, \theta, v]^\top$ where (px, py) is the location of the vehicle on a 2D plane, v is the velocity, and θ is the heading. The control inputs are $u = [a, \delta]$ where a is an acceleration and δ is a steering angle.

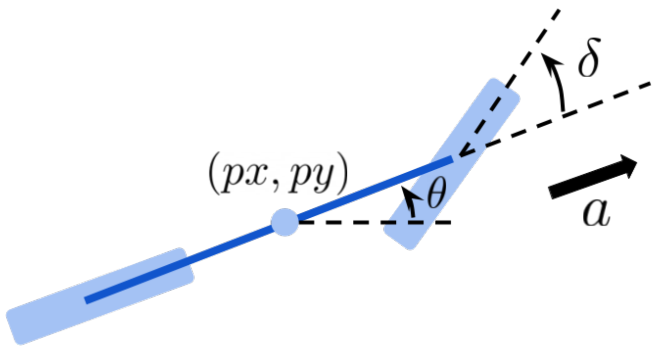


Figure 4.4: **(Bicycle Model)** Kinematic bicycle model that is commonly used for vehicle motion planning.

This experiment deals with a vehicle traveling on an uneven icy surface. The differential equation that governs the slippery bicycle model is

$$\dot{x} = \begin{bmatrix} v\cos(\theta+\beta) \\ v\sin(\theta+\beta) \\ \frac{v}{l_r}\sin(\beta) \\ a * \text{coeff}_a \end{bmatrix}, \beta = \arctan(\frac{l_r}{l_r + l_f}\tan(\delta))\text{coeff}_\delta \tag{4.7}$$

where l_r, l_f is the distance from the center of the bike to the rear and front wheel respectively. coeff_a and coeff_δ are slip coefficients in the range $(0, 1]$ where 1 contains no slip and corresponds to the normal bicycle model. A discrete system model is created using Euler integration with a time step of $\Delta t = 0.1$.

The task for the bicycle system is to navigate from a start state to a goal position while avoiding circular obstacles. An receding horizon optimization based planner is formulated as

$$\begin{aligned}
\min_{\{x_t\}_{t=1}^{T},\{u_t\}_{t=1}^{T-1}} \quad & \left\| [px, py]^\top - p_{goal} \right\|_2^2 + \sum_{t=1}^{T} \sum_{k=1}^{K} [1.1 * r_k - \left\| [px, py]^\top - c_k \right\|_2]_+ \\
\text{s.t.} \quad & \|u_t\|_\infty \leqslant u_{max}, t = 1, \dots, T-1 \\
& v_{min} \leqslant v_t \leqslant v_{max}, t = 1, \dots, T \\
& \left\| [px_t, py_t]^\top - c_k \right\|_2 \geqslant r_k, t = 1, \dots, T, k = 1, \dots, K,
\end{aligned} \tag{4.8}$$

where obstacles are represented as a set of circles, $(c_k, r_k)_{k=1}^{K}$, where c_k is the circle center and r_k is the radius. This controller hopes to keep the vehicle away from the obstacles and encourages the trajectory to keep a small safety margin.

Experimental Details

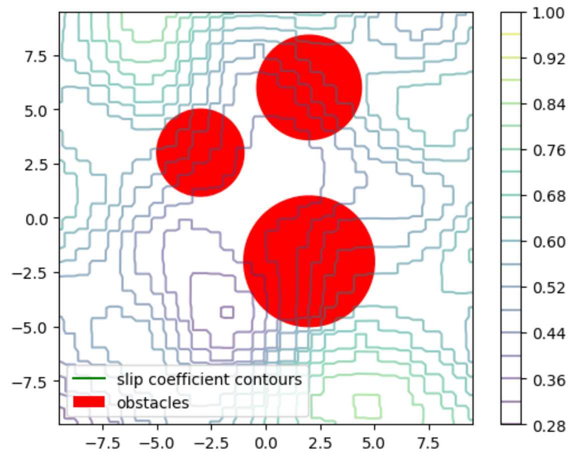


Figure 4.5: (Bicycle Slip Coefficient) The true map of the environment the bicycle system is simulated in. The slip coefficients in (4.7) are shown as contour lines.

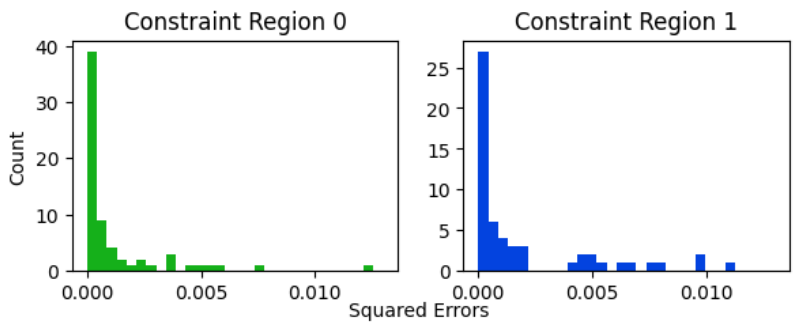


Figure 4.6: (Bicycle Square Errors) Squared errors in two constraint regions on a validation dataset. The errors approximately follow a folded Gaussian distribution.

Similar to the rocket experiment, a constrained and unconstrained model is trained. Both models try to learn the error between the *real* system model in (4.7) with slip coefficients varying over location as shown in Figure 4.5. Both the acceleration and turning slip coefficients are the same at any given location. Both models use neural networks identical to those used in the rocket experiment (Section 4.3.1) except that the hidden layers have 8 and 4 neurons.

The system specific functions for the bicycle system are as follows:

1. FAILURE: the system is in a failure mode if the bicycle system crashes into an obstacle

2. CREATE-CONSTRAINTS: A constraint region is generated around the state right before a crash that includes all (θ, v) values. (px, py) values within a radius of 0.5 are included. The (px, py) squared prediction error in a constraint region is modeled as a *folded normal distribution*. This assumption is approximate and can be seen in Figure 4.6. To maintain safety, the squared prediction error can be constrained such the distance from the predicted state to the nearest obstacles is greater than 3 standard deviations.

3. UPDATE-CONSTRAINTS: The bicycle system uses the *region merging* heuristic discussed in Section 4.2.2 when the overlap volume between two constraints is 40% of any of the two separate constraints.

The training procedure for both the constrained and unconstrained models consists of using an initial dataset of 40,000 data tuples to train an initial unconstrained model. Then 3 iterations of collecting data and training the model. Each data collection consisted of obtaining 32 trajectories using a receding time horizon controller that solves (4.8) with $T = 60$, $u_{max} = [1, 0.6]^{\top}$, $v_{min} = -1$, $v_{max} = 5$. There are three circular obstacles in the map shown in Figure 4.5. The starting (px, py) locations are sampled uniformly from the range $[-6, -4] \times [-6, -4]$. The goal (px, py) locations are sampled uniformly from the range $[4, 6] \times [4, 6]$. Each model is trained for 200 epochs between data gathering. After all the constraints are gathered, the models are finetuned for a further 500 epochs to facilitate constraint satisfaction.

Results and Analysis

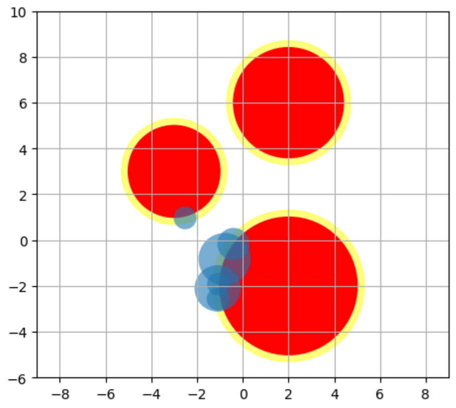


Figure 4.7: (Bicycle Constraint Regions) The generated constraint regions are shown in blue.

The constraint generation process created 6 constraint regions. The errors for the final constrained and unconstrained models are shown in Table 4.3 and their locations are

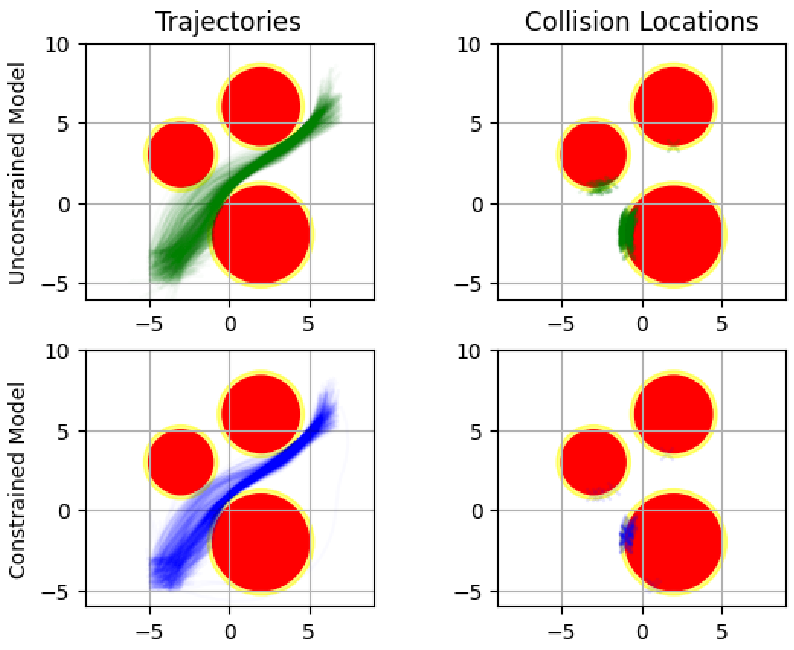


Figure 4.8: (Bicycle Trajectories and Failures) This plot shows the trajectories of 1000 test runs of the unconstrained and constrained models using the planner described in (4.8). The first row displays results for the unconstrained model in green. The second row displays results for the constrained model in blue. The first column shows the actual trajectories taken.

shown in Figure 4.7. The overall model error for both the constrained and unconstrained model are similar. However, the constrained model has less error in its constraint regions. This is traded off for worst accuracy elsewhere.

The last row of Table 4.3 displays the failure rate of the models when tested on 1000 validation start and goal locations for the planner (4.8) to navigate. Similar to the rocket system, this planner is operated in a receding time horizon loop. The constrained model obtains fails almost 3 times less than the unconstrained model. The 1000 trajectories and the failure locations are shown in Figure 4.8. While the constrained model still occasionally fails inside the constraint regions, the number is greatly reduced. There are 5 constraint regions generated for the bottom right obstacles, and Figure 4.8 shows that the vast majority of the collisions for the unconstrained model occurs in those constraint regions. This is alleviated in the constrained model.

Metric	Unconstrained Model	Constrained Model
Entire state-action space	7.937e-4	7.460e-4
Constraint 0: $\mathbb{E}[g_0 \mathbb{I}_0]$	3.966e-4	2.969e-4
Constraint 1: $\mathbb{E}[g_1 \mathbb{I}_1]$	9.947e-4	7.322e-4
Constraint 2: $\mathbb{E}[g_2 \mathbb{I}_2]$	5.269e-5	2.957e-5
Constraint 3: $\mathbb{E}[g_3 \mathbb{I}_3]$	1.760e-3	1.011e-3
Constraint 4: $\mathbb{E}[g_4 \mathbb{I}_4]$	3.672e-4	3.506e-4
Constraint 5: $\mathbb{E}[g_5 \mathbb{I}_5]$	1.000e-3	7.786e-4
Failure Rate	0.133	0.050

Table 4.3: **(Bicycle Experiment Results)** This table records the error of the model in various subsets of the dataset as well as the failure rate of running a closed loop planner 1000 times on problems with random start and goal states.

4.3.3 Conclusion

This chapter provides a method to alleviate some of the human effort involved in creating constraints for a *sufficiently accurate* model as presented in Chapter 3. By using an iterative process that requires a few system specific functions that may be easier to define than the constraints themselves, constraints may be automatically generated to improve the planning model.

The constrained rocket model achieved small performance gains compared to the unconstrained model, the constrained bicycle model improved upon its unconstrained counterpart by a larger margin. We believe this is due to the nature of the CREATE-CONSTRAINTS functions used. The rocket system used a fairly generic function. It simply created constraints that were some fraction of the current models errors. The bicycle model used a more principled approach that constrained the errors by a physically meaningful value. One interpretation is that the constrained bicycle model required more human effort to craft a more specific CREATE-CONSTRAINTS function. There is a trade off between crafting more specialized functions to enable the model learning algorithms to take advantage of human prior knowledge and performance and computational effort.

5 CONCLUSIONS

Incorporating machine learning algorithms into robot motion planning is about evaluating tradeoffs. Machine learning by itself is a tool that can be extremely powerful to enable pattern recognition across large datasets. However, it does not contain many of the human heuristics that might be hard to infer from data alone.

Chapter 2 discussed learning heuristics for planning algorithms. In particular, one experiment showed that learning a sampling distribution for a flytrap environment yields extremely similar results to some human designed heuristics (Figure 2.9). This illustrates the trade off on a small scale. Learning algorithms may be able to replace some human expertise but at the cost the computational effort required to search for the solution. Different problems may require a different mix of human expertise and machine learned data processing.

Chapters 3 and 4 explored a constrained framework for model learning. The results display a range of trade offs from explicitly designing constraints with human intuition (Chapter 3) to using less human expertise but more computational power to search for constraints (Chapter 4). Even within the semi-automatic process, there is a range of solutions that can incorporate less human knowledge (the rocket system) or more human knowledge (the bicycle system).

This work shows various methods that can shift the effort of finding efficient and low cost motion plans between humans and computers. There is no *free lunch* as each method comes with costs and benefits. At this time, specific robot systems and tasks should be evaluated on which aspects should be automated and which aspects should be designed with human expertise. Currently, the intersection of machine learning and robot motion planning is still in its infancy and we hope that future research can lessen the burden on the human expertise required to utilize the algorithms in this space.

6 APPENDIX

6.1 EXPECTATION-WISE LIPSCHITZ-CONTINUITY OF LOSS FUNCTIONS

This Appendix section shows that the loss functions in Examples 3.1 and 3.2 are expectation-wise Lipschitz-continuous.

Selective Accuracy, Example 3.1

The loss function $l(s, \phi) = \|\phi(x_t, u_t) - x_{t+1}\|_2$ is expectation-wise Lipschitz-continuous in ϕ. Using the reverse triangle inequality, we get

$$\begin{aligned} \|l(s,\phi_1) - l(s,\phi_2)\|_\infty &= \|\|\phi_1(x_t,u_t) - x_{t+1}\|_2 - \|\phi_2(x_t,u_t) - x_{t+1}\|_2\|_\infty \\ &\leqslant \|\|\phi_1(x_t,u_t) - \phi_2(x_t,u_t)\|_2\|_\infty . \end{aligned} \tag{6.1}$$

By, the equivalence of norms, there exists L such that $\|\phi_1(x,u) - \phi_2(x,u)\|_2 \leqslant L|\phi_1(x,u) - \phi_2(x,u)|$, for all (x,u). Thus,

$$\|l(s,\phi_1) - l(s,\phi_2)\|_\infty \leqslant L\,\|\phi_1 - \phi_2\|_\infty \tag{6.2}$$

Since this is true for any s, it is also true in expectation.

$$\mathbb{E}_s\,\|l(s,\phi_1) - l(s,\phi_2)\|_\infty \leqslant L\mathbb{E}_s\,\|\phi_1 - \phi_2\|_\infty \tag{6.3}$$

Normalized Objective, Example 3.2

The loss function $l(s, \phi) = \frac{\|\phi(x_t, u_t) - x_{t+1}\|_2}{\|x_{t+1}\|_2}\mathbb{I}_A(s)$ is also expectation-wise Lipschitz-continuous in ϕ. Following the same logic as for the euclidean norm, we get

$$\|l(s, \phi_1) - l(s, \phi_2)\|_\infty \leqslant L \left\| \frac{\mathbb{I}_A(s)}{\|x_{t+1}\|_2}(\phi_1 - \phi_2) \right\|_\infty \tag{6.4}$$

This reduces to

$$\|l(s, \phi_1) - l(s, \phi_2)\|_\infty \leqslant L \left\| \frac{1}{\min_{x_{t+1}} \|x_{t+1}\|_2}(\phi_1 - \phi_2) \right\|_\infty \tag{6.5}$$

when considering the case where $s \in A$. The largest that $\frac{\mathbb{I}_A(s)}{\|x_{t+1}\|_2}$ can be is 1 over the smallest value of $\|x_{t+1}\|_2$. If $s \notin A$, both sides reduce to 0, as the indicator variable is 0. This leads to

$$\begin{aligned} \|l(s, \phi_1) - l(s, \phi_2)\|_\infty &\leqslant L \left\| \frac{1}{\epsilon}(\phi_1 - \phi_2) \right\|_\infty \\ \mathbb{E}\,\|l(s, \phi_1) - l(s, \phi_2)\|_\infty &\leqslant \frac{L}{\epsilon}\mathbb{E}\,\|\phi_1 - \phi_2\|_\infty \end{aligned} \tag{6.6}$$

6.2 EQUIVALENT PROBLEM FORMULATION

Using the notation from Section 3.2, the problem defined in [110] is

$$\begin{aligned} P^\star = &\max_{\mathbf{x}, \phi} f_0(\mathbf{x}) \\ &\text{s.t. } \mathbf{x} \leqslant \mathbb{E}[\mathbf{f_1}(s, \phi(s))] \\ &\quad \mathbf{f_2}(\mathbf{x}) \geqslant 0, \mathbf{x} \in \mathcal{X}, \phi \in \Phi \end{aligned} \tag{6.7}$$

where f_0, and $\mathbf{f_2}$ are concave functions. $\mathbf{f_1}$ is not necessarily convex with respect to ϕ. $\mathcal{X}$ is a convex set and Φ is compact. Note that f_0, $\mathbf{f_1}$, $\mathbf{f_2}$, $\mathbf{x}$, and $\mathcal{X}$ are not directly present in the *Sufficiently Accurate* problem.

To translate the problem, let us assume that x is $K+1$ dimensional, where K is the number of constraints in (3.16). Let the last element of $\mathbf{f_1}(s,\phi)$ be equal to $-l(s,\phi)\mathbb{I}_0(s)$, the negative of the objective function in the *Sufficiently Accurate* problem. Let the first k^{th} element of $\mathbf{f_1}(s,\phi)$ be equal to $-g_k(s,\phi)\mathbb{I}_k(s)$, the negative of a constraint function in (3.16). Set the objective function to be $f_0(\mathbf{x}) = \mathbf{x}_{K+1}$, where $\mathbf{x}_{K+1}$ is the $(K+1)^{\text{th}}$ element of $\mathbf{x}$. $\mathbf{f_2}$ can be ignored by setting it to be the zero function. Under these assumptions, (6.7) is equivalent to the following

$$\begin{aligned} P^\star =& \max_{\mathbf{x},\phi} \mathbf{x}_{K+1} \\ &\text{s.t. } \mathbf{x}_k \leqslant -\mathbb{E}[g_k(s,\phi)\mathbb{I}_k(s)], k=1,\dots,K \\ &\quad \mathbf{x}_{K+1} \leqslant -\mathbb{E}[l(s,\phi)\mathbb{I}_0(s)] \\ &\quad \mathbf{x} \in \mathcal{X}, \phi \in \Phi. \end{aligned} \tag{6.8}$$

Now, define the set $\mathcal{X} = \{(x_1, x_2, \dots, x_{K+1}) : x_k = 0, k = 1, \dots, K, x_{K+1} \in \mathcal{X}_{K+1}\}$, where $\mathcal{X}_{K+1}$ is an arbitrary compact set in one dimension. This set of vectors, $\mathcal{X}$, is a set that is 0 in the first K components, and is compact in the last component. This, will further simplify (6.8) to the following

$$\begin{aligned} P^\star =& \max_{\phi} -\mathbb{E}[l(s,\phi)\mathbb{I}_0(s)] \\ &\text{s.t. } 0 \leqslant -\mathbb{E}[g_k(s,\phi)\mathbb{I}_k(s)], k=1,\dots,K \\ &\quad \phi \in \Phi \end{aligned} \tag{6.9}$$

as long as l takes on values in a compact set. Finally, flipping all the negatives in (6.9),

$$\begin{aligned} P^\star = -\min_{\phi\in\Phi} & \, \mathbb{E}[l(s,\phi)\mathbb{I}_0(s)] \\ \text{s.t. } & \, \mathbb{E}[g_k(s,\phi)\mathbb{I}_k(s)] \leqslant 0, k = 1,\dots,K \end{aligned} \tag{6.10}$$

This completes the translation of problem (6.7) to (3.16).

6.3 PROOF OF THEOREM 3

This proof follows some of the steps of the proof for [29, Theorem 1]. Let $(\phi^\star, \lambda^\star)$ be the primal and dual variables that attains the solution value of $P^\star = D^\star$ in problems (3.16) and (3.35). Similarly, let $(\theta^\star, \lambda^\star_\theta)$ be the primal and dual variables that attain the solution value $D^\star_\theta$ in problem (3.31). $\phi_{\theta^\star}$ is the function that $\theta^\star$ induces. Note that the optimal dual variables for (3.35), $\lambda^\star$, are not necessarily the same as the optimal dual variables for (3.31), $\lambda^\star_\theta$.

Lower Bound

We first show the lower bound for $D^\star_\theta$. Writing out the dual problem (3.35), we obtain

$$D^\star = \max_{\lambda\geqslant 0}\min_{\phi\in\Phi} \mathcal{L}(\phi,\lambda) = \mathcal{L}(\phi^\star,\lambda^\star) \tag{6.11}$$

Since $\lambda^\star$ is the optimal dual variable that achieves the maximal value for the maximization and minimization for the Lagrangian, it is true that

$$\phi^\star = \arg\min_{\phi} \mathcal{L}(\phi,\lambda^\star). \tag{6.12}$$

Thus for any ϕ,

$$\mathcal{L}(\phi^\star, \lambda^\star) \leqslant \mathcal{L}(\phi, \lambda^\star), \forall \phi \in \Phi. \tag{6.13}$$

We now look at the parameterized dual problem (3.31).

$$D_\theta^\star = \max_{\lambda \geqslant 0} \min_{\theta \in \Theta} \mathcal{L}_\theta(\theta, \lambda) = \max_{\lambda \geqslant 0} \min_{\theta \in \Theta} \mathcal{L}(\phi_\theta, \lambda) \tag{6.14}$$

This simply redefines $\mathcal{L}_\theta$ in terms of $\mathcal{L}$ as the only difference is that $\mathcal{L}_\theta$ is only defined for a subset of the primal variables that $\mathcal{L}$ is defined for. By definition, $\lambda_\theta^\star$ maximizes the minimization of $\mathcal{L}_\theta$ over θ. That is to say for the dual solution $(\theta^\star, \lambda_\theta^\star)$, $\lambda_\theta^\star$ minimizes $\mathcal{L}(\phi_{\theta^\star}, \cdot)$

$$\begin{aligned} \lambda_\theta^\star &= \arg\max_{\lambda \geqslant 0} \min_{\theta \in \Theta} \mathcal{L}(\phi_\theta, \lambda) \\ &= \arg\max_{\lambda \geqslant 0} \mathcal{L}(\phi_{\theta^\star}, \lambda). \end{aligned} \tag{6.15}$$

Thus, for all λ, it is the case that

$$\mathcal{L}(\phi_{\theta^\star}, \lambda_\theta^\star) \geqslant \mathcal{L}(\phi_{\theta^\star}, \lambda) \tag{6.16}$$

Putting together (6.13) and (6.16), we obtain

$$D_\theta^\star = \mathcal{L}(\phi_{\theta^\star}, \lambda_\theta^\star) \geqslant \mathcal{L}(\phi_{\theta^\star}, \lambda^\star) \geqslant \mathcal{L}(\phi^\star, \lambda^\star) = D^\star \tag{6.17}$$

Upper Bound

Next, we show the upper bound for $D_\theta^\star$. We begin by writing the Lagrangian (3.30)

$$D_\theta^\star = \max_{\lambda \geqslant 0} \min_{\phi \in \Phi_\theta} \mathcal{L}(\phi_\theta, \lambda) \tag{6.18}$$

as previously written in (6.14). By adding and subtracting $\mathcal{L}(\phi,\lambda)$, we obtain

$$\begin{aligned} D_\theta^\star &= \max_{\lambda \geqslant 0} \min_{\theta \in \Theta} \mathcal{L}(\phi,\lambda) + \mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda) \\ &= \max_{\lambda \geqslant 0} [\mathcal{L}(\phi,\lambda) + \min_{\theta \in \Theta} [\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)]] \\ &\leqslant \max_{\lambda \geqslant 0} [\mathcal{L}(\phi,\lambda) + \min_{\theta \in \Theta} |\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)|] \end{aligned} \tag{6.19}$$

where the last line comes from the fact that the absolute value of an expression is always at least as large as the original expression, i.e. $x \leqslant |x|$. Looking just at the quantity $|\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)|$, we can expand it as

$$\begin{aligned} |\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)| = |&\mathbb{E}[\ell_0(s,\phi_\theta) + \lambda^\top \mathbf{g}(s,\phi_\theta)] - \\ &\mathbb{E}[\ell_0(s,\phi) + \lambda^\top \mathbf{g}(s,\phi)]| \end{aligned} \tag{6.20}$$

Using the triangle inequality, this is upper bounded as

$$\begin{aligned} |\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)| \leqslant &|\mathbb{E}[\ell_0(s,\phi_\theta) - \ell_0(s,\phi)]| + \\ &|\mathbb{E}[\lambda^\top(\mathbf{g}(s,\phi_\theta) - \mathbf{g}(s,\phi))]| \end{aligned} \tag{6.21}$$

Using Hölder's inequality, we can create a further upper bound

$$\begin{aligned} |\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)| \leqslant &\|\mathbb{E}[\ell_0(s,\phi_\theta) - \ell_0(s,\phi)]\|_\infty + \\ &\|\lambda\|_1 \, \|\mathbb{E}[\mathbf{g}(s,\phi_\theta) - \mathbf{g}(s,\phi)]\|_\infty \end{aligned} \tag{6.22}$$

where the infinity norm of the scalar value $\mathbb{E}[\ell_0(s,\phi_\theta) - \ell_0(s,\phi)]$ is the same as its absolute value. Using the fact that the infinity norm is convex and Jensen's inequality, we can move the norm inside of the expectation.

$$|\mathcal{L}(\phi_\theta,\lambda) - \mathcal{L}(\phi,\lambda)| \leqslant \mathbb{E}\,\|\ell_0(s,\phi_\theta) - \ell_0(s,\phi)\|_\infty + \|\lambda\|_1\,\mathbb{E}\,\|\mathbf{g}(s,\phi_\theta) - \mathbf{g}(s,\phi)\|_\infty \tag{6.23}$$

By expectation-wise Lipschitz-continuity of both the loss and constraint functions,

$$\begin{aligned}|\mathcal{L}(\phi_\theta,\lambda)-\mathcal{L}(\phi,\lambda)| &\leqslant L\mathbb{E}\left\|\phi_\theta-\phi\right\|_\infty+\|\lambda\|_1\, L\mathbb{E}\left\|\phi_\theta-\phi\right\|_\infty \\ &=(1+\|\lambda\|_1)L\mathbb{E}\left\|\phi_\theta-\phi\right\|_\infty .\end{aligned} \tag{6.24}$$

Combining (6.19) with (6.24), we obtain

$$D^\star_\theta \leqslant \quad = \max_{\lambda\geqslant 0}[\mathcal{L}(\phi,\lambda)+(\|\lambda\|_1+1)L\min_{\theta\in\Theta}\mathbb{E}\left\|\phi_\theta-\phi\right\|_\infty] \tag{6.25}$$

Since, Φ_θ is an ϵ-universal approximation for Φ, we can write $\min_{\theta\in\Theta}\mathbb{E}\left\|\phi_\theta-\phi\right\|_\infty \leqslant \epsilon$. This further reduces (6.25) to

$$D^\star_\theta \leqslant \max_{\lambda\geqslant 0}[\mathcal{L}(\phi,\lambda)+(\|\lambda\|_1+1)L\epsilon]. \tag{6.26}$$

Note that (6.26) is true for all ϕ. In particular it must be also true for the λ that minimizes the inner value, i.e.

$$\begin{aligned}D^\star_\theta &\leqslant \max_{\lambda\geqslant 0}\min_{\phi\in\Phi}[\mathcal{L}(\phi,\lambda)+(\|\lambda\|_1+1)L\epsilon] \\ &= L\epsilon+\max_{\lambda\geqslant 0}\min_{\phi\in\Phi}[\mathbb{E}[\ell_0(s,\phi)]+\lambda^\top(\mathbb{E}[\mathbf{g}(s,\phi)]+\mathbf{1}L\epsilon)]\end{aligned} \tag{6.27}$$

The second half of (6.27) is actually the solution to the dual problem (3.26). The primal problem is reproduced here for reference,

$$\begin{aligned}P^\star_{L\epsilon} &= \min_{\phi\in\Phi}\mathbb{E}[\ell_0(s,\phi)] \\ &\text{s.t. } \mathbb{E}[\mathbf{g}(s,\phi)]+\mathbf{1}L\epsilon\leqslant 0.\end{aligned}$$

That is to say, $D^\star_\theta \leqslant L\epsilon + D^\star_{L\epsilon}$. The primal problem (3.25) is a perturbed version of (3.16), where all the constraints are tighter by $L\epsilon$. There exists a relationship between the solution of (3.25) and (3.16) from [15, Eq. 5.57]. Treating (3.16) as the perturbed version of (3.25) (that tightens the constraints by $-L\epsilon$), the relationship between the two solutions is

$$P^\star \geqslant P^\star_{L\epsilon} - {\lambda^\star_{L\epsilon}}^\top \mathbf{1} L\epsilon. \tag{6.28}$$

Since both (3.16) and (3.25) have zero duality gap by Theorem 2, this is the same as

$$D^\star \geqslant D^\star_{L\epsilon} - \|\lambda^\star_{L\epsilon}\|_1 \, L\epsilon. \tag{6.29}$$

Combining (6.29) with the fact that $D^\star_\theta \leqslant L\epsilon + D^\star_{L\epsilon}$, the following bound is obtained.

$$D^\star_\theta \leqslant D^\star + L\epsilon(\|\lambda^\star_{L\epsilon}\|_1 + 1) \tag{6.30}$$

This gives us the desired upper bound.

6.4 PROOF OF LEMMA 1

We start by establishing an upper bound on the difference $|D^\star_\theta - D^\star_N|$. By definition of the Dual Problems (3.31) and (3.22), it follows that $D^\star_\theta = \min_\theta \mathcal{L}_\theta(\theta, \lambda^\star_\theta)$ and $D^\star_N = \min_\theta \mathcal{L}_N(\theta, \lambda^\star_N)$. Hence we have that

$$\begin{aligned} D^\star_\theta - D^\star_N &= \min_\theta \mathcal{L}_\theta(\theta, \lambda^\star_\theta) - \min_\theta \mathcal{L}_N(\theta, \lambda^\star_N) \\ &\leqslant \min_\theta \mathcal{L}_\theta(\theta, \lambda^\star_\theta) - \min_\theta \mathcal{L}_N(\theta, \lambda^\star_\theta), \end{aligned} \tag{6.31}$$

where the inequality follows from the fact that $\lambda^\star_N$ maximizes the function $d_N(\lambda) = \min_\theta \mathcal{L}_N(\theta, \lambda)$. Thus, any other λ, in particular $\lambda^\star_\theta$ results in a value that is less than or equal to $D_N^\star$. Let $\theta_N^\star(\lambda) = \arg\min_{\theta\in\Theta} \mathcal{L}_N(\theta,\lambda)$. Substituting by this definition and using the definition of minimum, (6.31) can be further upper bounded by

$$D_\theta^\star - D_N^\star \leqslant \mathcal{L}_\theta(\theta_N^\star(\lambda_\theta^\star),\lambda_\theta^\star) - \mathcal{L}_N(\theta_N^\star(\lambda_\theta^\star),\lambda_\theta^\star). \tag{6.32}$$

We set now to establish a similar lower bound. Analogous to the step for the upper bound, we can use the definition of the Dual problem to lower bound $D_\theta^\star - D_N^\star$

$$D_\theta^\star - D_N^\star \geqslant \min_\theta \mathcal{L}_\theta(\theta,\lambda_N^\star) - \min_\theta \mathcal{L}_N(\theta,\lambda_N^\star). \tag{6.33}$$

Likewise, define $\theta^\star(\lambda) = \arg\min_{\theta\in\Theta} \mathcal{L}_\theta(\theta,\lambda)$ and further upper bound (6.33) as

$$D_\theta^\star - D_N^\star \geqslant \mathcal{L}_\theta(\theta^\star(\lambda_N^\star),\lambda_N^\star) - \mathcal{L}_N(\theta^\star(\lambda_N^\star),\lambda_N^\star). \tag{6.34}$$

Using the upper and lower bounds for $D_\theta^\star - D_N^\star$ derived in (6.32) and (6.34), we can bound the absolute value of the difference by the maximum of the absolute values of the lower and upper bounds

$$|D_\theta^\star - D_N^\star| \leqslant \max \begin{cases} |\mathcal{L}_\theta(\theta^\star(\lambda_N^\star),\lambda_N^\star) - \mathcal{L}_N(\theta^\star(\lambda_N^\star),\lambda_N^\star)| \\ |\mathcal{L}_\theta(\theta_N^\star(\lambda_\theta^\star),\lambda_\theta^\star) - \mathcal{L}_N(\theta_N^\star(\lambda_\theta^\star),\lambda_\theta^\star)| \end{cases} \tag{6.35}$$

A conservative upper bound for the previous expression is

$$|D_\theta^\star - D_N^\star| \leqslant |\sup_{\theta\in\Theta,\lambda\in\mathbb{R}_+^K} \mathcal{L}_\theta(\theta,\lambda) - \mathcal{L}_N(\theta,\lambda)| \tag{6.36}$$

This completes the proof of the Lemma.

LIST OF TABLES

LIST OF ILLUSTRATIONS

BIBLIOGRAPHY

[1] Akshay Agrawal, Shane Barratt, and Stephen Boyd. "Learning Convex Optimization Models." In: *arXiv preprint arXiv:2006.04248* (2020).

[2] Akshay Agrawal, Shane Barratt, Stephen Boyd, and Bartolomeo Stellato. "Learning Convex Optimization Control Policies." In: *Learning for Dynamics and Control*. PMLR. 2020, pp. 361–373.

[3] Ravindra K Ahuja and James B Orlin. "Inverse Optimization." In: *Operations Research* 49.5 (2001), pp. 771–783.

[4] Zlatan Ajanovic, Halil Beglerovic, and Bakir Lacevic. "A Novel Approach to Model Exploration for Value Function Learning." In: *arXiv preprint arXiv:1906.02789* (2019).

[5] Anurag Ajay, Maria Bauza, Jiajun Wu, Nima Fazeli, Joshua B. Tenenbaum, Alberto Rodriguez, and Leslie P. Kaelbling. "Combining Physical Simulators and Object-Based Networks for Control." In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 3217–3223. DOI: 10.1109/ICRA.2019.8794358.

[6] Nancy M Amato and Guang Song. "Using motion planning to study protein folding pathways." In: *Journal of Computational Biology* 9.2 (2002), pp. 149–168.

[7] Brandon Amos and J Zico Kolter. "OptNet: Differentiable Optimization as a Layer in Neural Networks." In: *International Conference on Machine Learning*. PMLR. 2017, pp. 136–145.

[8] Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J Zico Kolter. "Differentiable MPC for End-to-end Planning and Control." In: *arXiv preprint arXiv:1810.13400* (2018).

[9] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. "Spectrally-normalized margin bounds for neural networks." In: *Advances in Neural Information Processing Systems*. 2017, pp. 6240–6249.

[10] Mohak Bhardwaj, Sanjiban Choudhury, and Sebastian Scherer. "Learning Heuristic Search via Imitation." In: *Conference on Robot Learning*. PMLR. 2017, pp. 271–280.

[11] Jonathan Binney, Andreas Krause, and Gaurav S. Sukhatme. "Informative path planning for an autonomous underwater vehicle." In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 4791–4796. DOI: 10.1109/ROBOT.2010.5509714.

[12] Lars Blackmore and Masahiro Ono. "Convex Chance Constrained Predictive Control Without Sampling." In: *AIAA Guidance, Navigation, and Control Conference*, p. 5876.

[13] V. Boor, M.H. Overmars, and A.F. van der Stappen. "The Gaussian sampling strategy for probabilistic roadmap planners." In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. Vol. 2. 1999, 1018–1023 vol.2. DOI: 10.1109/ROBOT.1999.772447.

[14] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[15] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex Optimization*. Cambridge university press, 2004.

[16] Oliver Brock, Jeff Trinkle, and Fabio Ramos. "BiSpace Planning: Concurrent Multi-Space Exploration." In: *Robotics: Science and Systems IV*. 2009, pp. 159–166.

[17] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "Openai Gym." In: *arXiv preprint arXiv:1606.01540* (2016).

[18] Arunkumar Byravan, Felix Leeb, Franziska Meier, and Dieter Fox. "SE3-Pose-Nets: Structured Deep Dynamics Models for Visuomotor Control." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 3339–3346. DOI: 10.1109/ICRA.2018.8461184.

[19] Timothy CY Chan, Taewoo Lee, and Daria Terekhov. "Inverse Optimization: Closed-form Solutions, Geometry and Goodness of fit." In: *Management Science* 65.3 (2019), pp. 1115–1135.

[20] Binghong Chen, Bo Dai, Qinjie Lin, Guo Ye, Han Liu, and Le Song. "Learning to Plan in High Dimensions via Neural Exploration-Exploitation Trees." In: *International Conference on Learning Representations*. 2019.

[21] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. "Neural Ordinary Differential Equations." In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 6572–6583.

[22] Glen Chou, Dmitry Berenson, and Necmiye Ozay. "Learning Constraints from Demonstrations." In: *International Workshop on the Algorithmic Foundations of Robotics*. Springer. 2018, pp. 228–245.

[23] Sanjiban Choudhury, Mohak Bhardwaj, Sankalp Arora, Ashish Kapoor, Gireeja Ranade, Sebastian Scherer, and Debadeepta Dey. "Data-driven Planning via Imitation Learning." In: *The International Journal of Robotics Research* 37.13-14 (2018), pp. 1632–1672.

[24] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. "Theta*: Any-angle Path Planning on Grids." In: *Journal of Artificial Intelligence Research* 39 (2010), pp. 533–579.

[25] Luc De Raedt, Andrea Passerini, and Stefano Teso. "Learning Constraints from Examples." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[26] Marc Deisenroth and Carl E Rasmussen. "PILCO: A Model-based and Data-efficient Approach to Policy Search." In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. Citeseer. 2011, pp. 465–472.

[27] EW Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik*. 1959, pp. 269–271.

[28] Carl Doersch. "Tutorial on Variational Autoencoders." In: *arXiv preprint arXiv:1606.05908* (2016).

[29] Mark Eisen, Clark Zhang, Luiz FO Chamon, Daniel D Lee, and Alejandro Ribeiro. "Learning Optimal Resource Allocations in Wireless Systems." In: *arXiv preprint arXiv:1807.08088* (2018).

[30] Nima Fazeli, Samuel Zapolsky, Evan Drumwright, and Alberto Rodriguez. "Learning Data-efficient Rigid-body Contact Models: Case Study of Planar Impact." In: *Conference on Robot Learning*. PMLR. 2017, pp. 388–397.

[31] Chelsea Finn and Sergey Levine. "Deep visual foresight for planning robot motion." In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 2786–2793. DOI: 10.1109/ICRA.2017.7989324.

[32] RG Franks and CW Worley. "Quantitative Analysis of Cascade Control." In: *Industrial & Engineering Chemistry* 48.6 (1956), pp. 1074–1079.

[33] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." In: *International Conference on Machine Learning*. PMLR. 2016, pp. 1050–1059.

[34] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic." In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 2997–3004. DOI: 10.1109/IROS.2014.6942976.

[35] David M Gay, Michael L Overton, and Margaret H Wright. "A Primal-dual Interior Method for Nonconvex Nonlinear Programming." In: *Advances in Nonlinear Programming*. Springer, 1998, pp. 31–56.

[36] Philip E Gill and Daniel P Robinson. "A primal-dual augmented Lagrangian." In: *Computational Optimization and Applications* 51.1 (2012), pp. 1–25.

[37] Tom Goldstein, Brendan O'Donoghue, Simon Setzer, and Richard Baraniuk. "Fast Alternating Direction Optimization Methods." In: *SIAM Journal on Imaging Sciences* 7.3 (2014), pp. 1588–1623.

[38] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. "Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning." In: *Journal of Machine Learning Research* 5.9 (2004).

[39] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. "Continuous Deep Q-Learning with Model-based Acceleration." In: *International Conference on Machine Learning*. PMLR. 2016, pp. 2829–2838.

[40] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. "Learning Latent Dynamics for Planning from Pixels." In: *International Conference on Machine Learning*. PMLR. 2019, pp. 2555–2565.

[41] Daniel Harabor and Alban Grastien. "Online Graph Pruning for Pathfinding on Grid Maps." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 25. 1. 2011.

[42] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[44] Sylvia L. Herbert, Mo Chen, SooJean Han, Somil Bansal, Jaime F. Fisac, and Claire J. Tomlin. "FaSTrack: A modular framework for fast and guaranteed safe motion planning." In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. 2017, pp. 1517–1522. DOI: 10.1109/CDC.2017.8263867.

[45] Gabriel Hoffmann, Steven Waslander, and Claire Tomlin. "Quadrotor Helicopter Trajectory Tracking Control." In: *AIAA Guidance, Navigation and Control Conference and Exhibit*. 2008, p. 7410.

[46] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks are Universal Approximators." In: *Neural networks* 2.5 (1989), pp. 359–366.

[47] D. Hsu, J.-C. Latombe, and R. Motwani. "Path planning in expansive configuration spaces." In: *Proceedings of International Conference on Robotics and Automation*. Vol. 3. 1997, 2719–2726 vol.3. DOI: 10.1109/ROBOT.1997.619371.

[48] Jinwook Huh, Galen Xing, Ziyun Wang, Volkan Isler, and Daniel D Lee. "Learning to Generate Cost-to-Go Functions for Efficient Motion Planning." In: *arXiv preprint arXiv:2010.14597* (2020).

[49] Brian Ichter, James Harrison, and Marco Pavone. "Learning Sampling Distributions for Robot Motion Planning." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 7087–7094. DOI: 10.1109/ICRA.2018.8460730.

[50] Brian Ichter and Marco Pavone. "Robot Motion Planning in Learned Latent Spaces." In: *IEEE Robotics and Automation Letters* 4.3 (2019), pp. 2407–2414. DOI: 10.1109/LRA.2019.2901898.

[51] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *International Conference on Machine Learning*. 2015, pp. 448–456.

[52] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. "When to Trust Your Model: Model-Based Policy Optimization." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[53] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. "Fast Marching Tree: a Fast Marching Sampling-based Method for Optimal Motion Planning in Many Dimensions." In: *The International Journal of Robotics Research* 34.7 (2015), pp. 883–921.

[54] Leslie Pack Kaelbling and Tomás Lozano-Pérez. "Hierarchical task and motion planning in the now." In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1470–1477. DOI: 10.1109/ICRA.2011.5980391.

[55] Łukasz Kaiser, Mohammad Babaeizadeh, Piotr Miłos, Błażej Osiński, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. "Model Based Reinforcement Learning for Atari." In: *International Conference on Learning Representations*. 2020.

[56] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. "STOMP: Stochastic trajectory optimization for motion planning." In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 4569–4574. DOI: 10.1109/ICRA.2011.5980280.

[57] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning." In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894.

[58] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[59] Karel J Keesman. *System Identification: an Introduction*. Springer Science & Business Media, 2011.

[60] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[61] Donald E Kirk. *Optimal Control Theory: An Introduction*. Courier Corporation, 2004.

[62] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A*." In: *Artificial Intelligence* 155.1-2 (2004), pp. 93–146.

[63] Vijay R Konda and John N Tsitsiklis. "Actor-Critic Algorithms." In: *Advances in neural information processing systems*. Citeseer. 2000, pp. 1008–1014.

[64] Donald H. Kraft. "A Software Package for Sequential Quadratic Programming." In: 1988.

[65] J.J. Kuffner and S.M. LaValle. "RRT-connect: An efficient approach to single-query path planning." In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 2. 2000, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

[66] 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." In: *Autonomous Robots* 40.3 (2016), pp. 429–455.

[67] Tobias Kunz and Mike Stilman. "Kinodynamic RRTs with Fixed Time Step and Best-Input Extension Are Not Probabilistically Complete." In: *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 233–244.

[68] Tobias Kunz, Andrea Thomaz, and Henrik Christensen. "Hierarchical rejection sampling for informed kinodynamic planning in high-dimensional spaces." In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 89–96. DOI: 10.1109/ICRA.2016.7487120.

[69] Thanard Kurutach, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel. "Model-Ensemble Trust-Region Policy Optimization." In: *International Conference on Learning Representations*. 2018.

[70] Yoshiaki Kuwata, Gaston A. Fiore, Justin Teo, Emilio Frazzoli, and Jonathan P. How. "Motion planning for urban driving using RRT." In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008, pp. 1681–1686. DOI: 10.1109/IROS.2008.4651075.

[71] Steven M LaValle. *Planning Algorithms*. Cambridge university press, 2006.

[72] Steven M LaValle and James J Kuffner Jr. "Randomized Kinodynamic Planning." In: *The International Journal of Robotics Research* 20.5 (2001), pp. 378–400.

[73] Steven M LaValle et al. "Rapidly-exploring Random Trees: A New Tool for Path Planning." In: (1998).

[74] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[75] Yongho Lee, Sunwon Park, and Moonyong Lee. "PID Controller Tuning to Obtain Desired Closed Loop Responses for Cascade Control Systems." In: *Industrial & engineering chemistry research* 37.5 (1998), pp. 1859–1865.

[76] Sergey Levine and Vladlen Koltun. "Guided Policy Search." In: *International conference on machine learning*. PMLR. 2013, pp. 1–9.

[77] Weiwei Li and Emanuel Todorov. "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems." In: Citeseer.

[78] Yanbo Li, Zakary Littlefield, and Kostas E Bekris. "Asymptotically Optimal Sampling-based Kinodynamic Planning." In: *The International Journal of Robotics Research* 35.5 (2016), pp. 528–564.

[79] Zhongyu Li, Xuxin Cheng, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. "Reinforcement Learning for Robust Parameterized Locomotion Control of Bipedal Robots." In: *arXiv preprint arXiv:2103.14295* (2021).

[80] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. "Anytime Search in Dynamic Graphs." In: *Artificial Intelligence* 172.14 (2008), pp. 1613–1643.

[81] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. "ARA*: Anytime A* with Provable Bounds on Sub-optimality." In: *Advances in Neural Information Processing Systems* 16 (2003), pp. 767–774.

[82] Chang Liu, Seungho Lee, Scott Varnhagen, and H. Eric Tseng. "Path planning for autonomous vehicles using model predictive control." In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 174–179. DOI: 10.1109/IVS.2017.7995716.

[83] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. "PDE-Net: Learning PDEs from Data." In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3208–3216.

[84] Peter Lucas and Linda Van Der Gaag. "Principles of Expert Systems." In: (1991).

[85] Brandon J Luders, Mangal Kothariyand, and Jonathan P How. "Chance Constrained RRT for Probabilistic Robustness to Environmental Uncertainty." In: *Proceedings of the AIAA Guidance, Navigation, and Control Conference*. 2010.

[86] Prasanta Chandra Mahalanobis. "On the generalized distance in statistics." In: National Institute of Science of India. 1936.

[87] Daniel Mellinger and Vijay Kumar. "Minimum snap trajectory generation and control for quadrotors." In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 2520–2525. DOI: 10.1109/ICRA.2011.5980409.

[88] Gregory P Meyer. "An Alternative Probabilistic Interpretation of the Huber Loss." In: *arXiv preprint arXiv:1911.02088* (2019).

[89] Tom M Mitchell. *The Need for Biases in Learning Generalizations*. Department of Computer Science, Laboratory for Computer Science Research . . ., 1980.

[90] Richard M Murray, Zexiang Li, and S Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC press, 2017.

[91] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 7559–7566. DOI: 10.1109/ICRA.2018.8463189.

[92] Angelia Nedić and Asuman Ozdaglar. "Subgradient Methods for Saddle-Point Problems." In: *Journal of optimization theory and applications* 142.1 (2009), pp. 205–228.

[93] Duy Nguyen-Tuong and Jan Peters. "Model Learning for Robot Control: A Survey." In: *Cognitive processing* 12.4 (2011), pp. 319–340.

[94] Duy Nguyen-Tuong, Jan Peters, and Matthias Seeger. "Local Gaussian Process Regression for Real Time Online Model Learning and Control." In: *Proceedings of the 21st International Conference on Neural Information Processing Systems*. 2008, pp. 1193–1200.

[95] Samet Oymak and Necmiye Ozay. "Non-asymptotic Identification of LTI Systems from a Single Trajectory." In: *2019 American Control Conference (ACC)*. 2019, pp. 5655–5661. DOI: 10.23919/ACC.2019.8814438.

[96] Deepak Pathak, Philipp Krähenbühl, and Trevor Darrell. "Constrained Convolutional Neural Networks for Weakly Supervised Segmentation." In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1796–1804. DOI: 10.1109/ICCV.2015.209.

[97] H.D. Patino, R. Carelli, and B.R. Kuchen. "Neural networks for advanced control of robot manipulators." In: *IEEE Transactions on Neural Networks* 13.2 (2002), pp. 343–354. DOI: 10.1109/72.991420.

[98] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 3803–3810. DOI: 10.1109/ICRA.2018.8460528.

[99] Claudia Pérez-D'Arpino and Julie A Shah. "C-learn: Learning geometric constraints from demonstrations for multi-step manipulation in shared autonomy." In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 4058–4065.

[100] Alejandro Perez, Robert Platt, George Konidaris, Leslie Kaelbling, and Tomas Lozano-Perez. "LQR-RRT*: Optimal sampling-based motion planning with automatically derived extension heuristics." In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 2537–2542. DOI: 10.1109/ICRA.2012.6225177.

[101] Klaus Peternell, Wolfgang Scherrer, and Manfred Deistler. "Statistical Analysis of Novel Subspace Identification Methods." In: *Signal Processing* 52.2 (1996), pp. 161–177.

[102] Jan Peters and Stefan Schaal. “Policy Gradient Methods for Robotics.” In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2006, pp. 2219–2225. DOI: 10.1109/IROS.2006.282564.

[103] Jan Peters and Stefan Schaal. “Natural Actor-Critic.” In: *Neurocomputing* 71.7-9 (2008), pp. 1180–1190.

[104] Gianluigi Pillonetto and Giuseppe De Nicolao. “A New Kernel-based Approach for Linear System Identification.” In: *Automatica* 46.1 (2010), pp. 81–93.

[105] Philip Polack, Florent Altché, Brigitte d’Andréa Novel, and Arnaud de La Fortelle. “The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?” In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[106] Michael Posa, Cecilia Cantu, and Russ Tedrake. “A Direct Method for Trajectory Optimization of Rigid Bodies Through Contact.” In: *The International Journal of Robotics Research* 33.1 (2014), pp. 69–81.

[107] Ahmed H. Qureshi, Anthony Simeonov, Mayur J. Bency, and Michael C. Yip. “Motion Planning Networks.” In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 2118–2124. DOI: 10.1109/ICRA.2019.8793889.

[108] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. “Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems.” In: *arXiv preprint arXiv:1801.01236* (2018).

[109] Harish chaandar Ravichandar, Iman Salehi, Brian P. Baillie, George M. Bollas, and Ashwin Dani. “Learning Stable Nonlinear Dynamical Systems with External Inputs using Gaussian Mixture Models.” In: *2018 Annual American Control Conference (ACC)*. 2018, pp. 4825–4830. DOI: 10.23919/ACC.2018.8431461.

[110] Alejandro Ribeiro. “Optimal Resource Allocation in Wireless Communication and Networking.” In: *EURASIP Journal on Wireless Communications and Networking* 2012.1 (2012), p. 272.

[111] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning.” In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 627–635.

[112] Pedro Sanchez-Cuevas, Guillermo Heredia, and Anibal Ollero. “Characterization of the Aerodynamic Ground Effect and its Influence in Multirotor Control.” In: *International Journal of Aerospace Engineering* 2017 (2017).

[113] Riccardo Scattolini. “Architectures for Distributed and Hierarchical Model Predictive Control–A Review.” In: *Journal of Process Control* 19.5 (2009), pp. 723–731.

[114] Stefan Schaal, Christopher G Atkeson, and Sethu Vijayakumar. "Scalable Techniques from Nonparametric Statistics for Real Time Robot Learning." In: *Applied Intelligence* 17.1 (2002), pp. 49–60.

[115] Karl Schmeckpeper, Annie Xie, Oleh Rybkin, Stephen Tian, Kostas Daniilidis, Sergey Levine, and Chelsea Finn. "Learning Predictive Models from Observation and Interaction." In: *arXiv preprint arXiv:1912.12773* (2019).

[116] Maarten Schoukens and Koen Tiels. "Identification of Block-oriented Nonlinear Systems Starting from Linear Approximations: A Survey." In: *Automatica* 85 (2017), pp. 272–292.

[117] John Schulman, Jonathan Ho, Alex X Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. "Finding Locally Optimal, Collision-free Trajectories with Sequential Convex Optimization." In: *Robotics: Science and Systems*. Vol. 9. 1. Citeseer. 2013, pp. 1–10.

[118] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust Region Policy Optimization." In: *International Conference on Machine Learning*. PMLR. 2015, pp. 1889–1897.

[119] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms." In: *arXiv preprint arXiv:1707.06347* (2017).

[120] Jacob T Schwartz, Micha Sharir, and John E Hopcroft. *Planning, Geometry, and Complexity of Robot Motion*. NJ, United States: Ablex Publishing Corp., 1987.

[121] Xiaotong Shen, Wei Pan, Yunzhang Zhu, and Hui Zhou. "On constrained and regularized high-dimensional regression." In: *Annals of the Institute of Statistical Mathematics* 65.5 (2013), pp. 807–832.

[122] Alexander Shkolnik and Russ Tedrake. "Sample-based Planning with Volumes in Configuration Space." In: *arXiv preprint arXiv:1109.3145* (2011).

[123] Alexander Shkolnik, Matthew Walter, and Russ Tedrake. "Reachability-guided sampling for planning under differential constraints." In: *2009 IEEE International Conference on Robotics and Automation*. 2009, pp. 2859–2865. DOI: 10.1109/ROBOT.2009.5152874.

[124] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. "Deterministic Policy Gradient Algorithms." In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.

[125] Satinder P Singh and Richard S Sutton. "Reinforcement Learning with Replacing Eligibility Traces." In: *Machine learning* 22.1 (1996), pp. 123–158.

[126] Sumeet Singh, Mo Chen, Sylvia L Herbert, Claire J Tomlin, and Marco Pavone. "Robust Tracking with Model Mismatch for Fast and Safe Planning: An SOS Optimization Approach." In: *International Workshop on the Algorithmic Foundations of Robotics*. Springer. 2018, pp. 545–564.

[127] Moshe Sniedovich. "Dijkstra's Algorithm Revisited: The Dynamic Programming Connexion." In: *Control and cybernetics* 35.3 (2006), pp. 599–620.

[128] Bharath Sriperumbudur, Kenji Fukumizu, and Gert Lanckriet. "On the relation between universality, characteristic kernels and RKHS embedding of measures." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 773–780.

[129] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. "On the importance of initialization and momentum in deep learning." In: *International Conference on Machine Learning*. 2013, pp. 1139–1147.

[130] Richard S Sutton. "Dyna, an Integrated Architecture for Learning, Planning, and Reacting." In: *ACM Sigart Bulletin* 2.4 (1991), pp. 160–163.

[131] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[132] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." In: *NIPs*. Vol. 99. Citeseer. 1999, pp. 1057–1063.

[133] Johan AK Suykens and Joos Vandewalle. "Least Squares Support Vector Machine Classifiers." In: *Neural Processing Letters* 9.3 (1999), pp. 293–300.

[134] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. "Sim-to-Real: Learning Agile Locomotion For Quadruped Robots." In: *Robotics: Science and Systems*. 2018.

[135] Yingcong Tan, Daria Terekhov, and Andrew Delong. "Learning Linear Programs from Optimal Decisions." In: *arXiv preprint arXiv:2006.08923* (2020).

[136] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. "DeepMind Control Suite." In: *arXiv preprint arXiv:1801.00690* (2018).

[137] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. "LQR-trees: Feedback Motion Planning via Sums-of-Squares Verification." In: *The International Journal of Robotics Research* 29.8 (2010), pp. 1038–1052.

[138] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon." In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[139] Emanuel Todorov, Tom Erez, and Yuval Tassa. "Mujoco: A Physics Engine for Model-based Control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033.

[140] Emanuel Todorov and Weiwei Li. "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems." In: *Proceedings of the 2005, American Control Conference, 2005.* IEEE. 2005, pp. 300–306.

[141] C. Urmson and R. Simmons. "Approaches for heuristically biasing RRT growth." In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453).* Vol. 2. 2003, 1178–1183 vol.2. DOI: 10.1109/IROS.2003.1248805.

[142] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. "Autonomous Driving in Urban Environments: Boss and the Urban Challenge." In: *Journal of Field Robotics* 25.8 (2008), pp. 425–466.

[143] V.N. Vapnik. "An overview of statistical learning theory." In: *IEEE Transactions on Neural Networks* 10.5 (1999), pp. 988–999. DOI: 10.1109/72.788640.

[144] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[145] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas Huang, and Yihong Gong. "Locality-constrained Linear Coding for image classification." In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* 2010, pp. 3360–3367. DOI: 10.1109/CVPR.2010.5540018.

[146] Christopher JCH Watkins and Peter Dayan. "Q-learning." In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[147] Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin Riedmiller. "Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images." In: *arXiv preprint arXiv:1506.07365* (2015).

[148] Dustin J Webb and Jur van den Berg. "Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints." In: *arXiv preprint arXiv:1205.5088* (2012).

[149] Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M. Rehg, Byron Boots, and Evangelos A. Theodorou. "Information theoretic MPC for model-based reinforcement learning." In: *2017 IEEE International Conference on Robotics and Automation (ICRA).* 2017, pp. 1714–1721. DOI: 10.1109/ICRA.2017.7989202.

[150] Ronald J Williams. “Simple Statistical Gradient-following Algorithms for Connectionist Reinforcement Learning.” In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[151] S.A. Wilmarth, N.M. Amato, and P.F. Stiller. “MAPRM: a probabilistic roadmap planner with sampling on the medial axis of the free space.” In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. Vol. 2. 1999, 1024–1031 vol.2. DOI: 10.1109/ROBOT.1999.772448.

[152] Ian H Witten. “An Adaptive Optimal Controller for Discrete-time Markov Environments.” In: *Information and control* 34.4 (1977), pp. 286–295.

[153] Yuandong Yang and O. Brock. “Adapting the sampling distribution in PRM planners based on an approximated medial axis.” In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*. Vol. 5. 2004, 4405–4410 Vol.5. DOI: 10.1109/ROBOT.2004.1302411.

[154] Georgios N. Yannakakis and Julian Togelius. “A Panorama of Artificial and Computational Intelligence in Games.” In: *IEEE Transactions on Computational Intelligence and AI in Games* 7.4 (2015), pp. 317–335. DOI: 10.1109/TCIAIG.2014.2339221.

[155] A. Yershova, L. Jaillet, T. Simeon, and S.M. LaValle. “Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain.” In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 3856–3861. DOI: 10.1109/ROBOT.2005.1570709.

[156] Daqing Yi, Rohan Thakker, Cole Gulino, Oren Salzman, and Siddhartha Srinivasa. “Generalizing Informed Sampling for Asymptotically-Optimal Sampling-Based Kinodynamic Planning via Markov Chain Monte Carlo.” In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 7063–7070. DOI: 10.1109/ICRA.2018.8460188.

[157] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. “TossingBot: Learning to Throw Arbitrary Objects With Residual Physics.” In: *IEEE Transactions on Robotics* 36.4 (2020), pp. 1307–1319. DOI: 10.1109/TRO.2020.2988642.

[158] Clark Zhang, Jinwook Huh, and Daniel D. Lee. “Learning Implicit Sampling Distributions for Motion Planning.” In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 3654–3661. DOI: 10.1109/IROS.2018.8594028.

[159] Clark Zhang, Arbaaz Khan, Santiago Paternain, and Alejandro Ribeiro. “Sufficiently Accurate Model Learning.” In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 10991–10997. DOI: 10.1109/ICRA40945.2020.9197502.

[160] Clark Zhang, Santiago Paternain, and Alejandro Ribeiro. “Sufficiently Accurate Model Learning for Planning.” In: *arXiv preprint arXiv:2102.06099* (2021).

[161] Liangjun Zhang and Dinesh Manocha. “An efficient retraction-based RRT planner.” In: *2008 IEEE International Conference on Robotics and Automation*. 2008, pp. 3743–3750. DOI: 10.1109/ROBOT.2008.4543785.

[162] Kemin Zhou and John Comstock Doyle. *Essentials of Robust Control*. Vol. 104. Prentice hall Upper Saddle River, NJ, 1998.

[163] Yuke Zhu, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, et al. “Reinforcement and Imitation Learning for Diverse Visuomotor Skills.” In: *arXiv preprint arXiv:1802.09564* (2018).

[164] Hui Zou and Trevor Hastie. “Regularization and Variable Selection via the Elastic Net.” In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2 (2005), pp. 301–320.

[165] Matt Zucker, James Kuffner, and J. Andrew Bagnell. “Adaptive workspace biasing for sampling-based planners.” In: *2008 IEEE International Conference on Robotics and Automation*. 2008, pp. 3757–3762. DOI: 10.1109/ROBOT.2008.4543787.

[166] Matt Zucker, Nathan Ratliff, Anca D Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher M Dellin, J Andrew Bagnell, and Siddhartha S Srinivasa. “Chomp: Covariant Hamiltonian Optimization for Motion Planning.” In: *The International Journal of Robotics Research* 32.9-10 (2013), pp. 1164–1193.