SAMPLING-BASED PLANNING FOR HYBRID SYSTEMS

by

JOSHUA AARON LEVINE

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Thesis Advisor: Dr. Michael S. Branicky

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2004

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to first thank my family and friends for letting me explain to them every minute detail of my thesis ad nauseam (they asked first). My father especially deserves thanks for actually sitting down and proofreading the first few chapters. They are all intelligent people, and while they may not have understood everything I told them; they were smart enough to at least pretend they did. And they all have better grammar.

In addition, I would like to thank my advisor, Dr. Michael Branicky, for guiding me through my thesis. He has provided expert direction and created a relaxed, intellectual atmosphere that stimulates critical thinking. Also, Stuart Morgan deserves credit for numerous team discussions (peer advising) regarding the RRT as well as suggesting the third and fourth stair climber examples and the idea of a contour map for the RRT.

David Carlin's generous loan of computer resources allowed me to run many experiments that I would not have had the CPU power to run otherwise. Finally, Shaun Edwards deserves thanks for help with using Pro/E to create the 3d models used in my examples.

Sampling-Based Planning for Hybrid Systems

Abstact

by

JOSHUA AARON LEVINE

This thesis presents research involving sampling-based planning to study hybrid systems. This work has provided experimental success in understanding hybrid systems and their reachability properties. We approach this problem from a sampling-based planning perspective by applying the rapidly-exploring random tree (RRT) algorithm previously used in motion-planning problems. To this end, we have extended a current graphical tool for sampling-based planning, the Motion Strategy Library (MSL), to accept problems that are hybrid in nature. In addition, this thesis also furthers the study of the RRT algorithm by presenting new visual and statistical results.

Chapter 1

Introduction

1.1 Problem Overview and Motivation

The focus of this thesis is to study hybrid systems by means of sampling-based planning techniques. Informally, **hybrid systems** are those systems that contain both continuous and discrete elements. The values of the discrete variables represent different modes (or states) of the system, and the continuous variables change in different ways based on the current mode. A simple example of a hybrid system is that of a manual transmission car. The car has a continuous variable of velocity and a discrete variable of what gear the car is in. The acceleration of the car is dependent on which gear the car is in: position and velocity (the continuous variables) change based on the gear (the discrete variable).

Previous research has shown that hybrid systems are a convenient, succinct description for many real life applications. For example, hybrid systems have been used to model air traffic management [34, 56] and airplane safety [22], audio protocols [9, 40], automated vehicle highway systems [49], automotive suspension control [33], billiards [1, 47], bouncing balls [55], a cat and mouse problem [47], distributed controllers [30], a gas burner [1, 2, 30], a manual transmission car [40], mutual exclusion

protocols [1, 2, 3, 9, 23, 30, 40], mixing tanks [22], pendulums [50], predator-prey systems [33, 36], railroad management [3], robotic assembly [13, 14, 20], scheduling [3], a steam boiler [33], temperature control/thermostat [1, 3, 32, 33, 34, 47], a two-robot conveyor belt [3], a two-tank water system [1, 2, 34, 48, 55], uninhabited aerial vehicles (primarily helicopters) [24, 25, 26], the weather [50], and the author's favorite, a person walking in a multi-story building—a "stair climber" [15, 16, 17, 21].

This list is by no means an exhaustive one. Applications of hybrid systems fall into the general categories of robotics, chemistry, physics, the automotive and airline industries, signal processing, safety, and control. Hybrid systems provide a very general framework for modeling that is concise, convenient, and powerful. They are an ideal choice for an innumerable number of applications.

The PRIMARY PROBLEM we study in this thesis is the **reachability problem** for hybrid systems. Informally, the reachability problem asks "Starting with an initial configuration, is it possible for this system to reach a state with a given (different) configuration?" Returning to our manual transmission car, say we are interested in designing a controller to shift the gears. We could develop a scheme to shift based on particular velocities and then propose a reachability question such as "Starting in gear 1 with velocity 0 mph, is it possible to reach gear 4 with velocity 10 mph?" Obviously, in this case, we would like to show the answer to this question is "no", as most engines do not tolerate high gears with low velocities.

Since hybrid systems are used to model systems found in every day life, thorough analysis is required to make sure they are safe, complete representations. We can apply the reachability question to study safety-critical properties of hybrid systems. When given a hybrid system representation of the dynamics and control of a system, we phrase the reachability question as "Can this system ever reach an undesirable state?" In the formal model for hybrid systems used in this thesis, hybrid automata [2, 1, 47], the answer to the question is limited by the complexity of the system itself.

Currently, research in modeling hybrid systems has shown limitations, specifically in the scope of the types of hybrid systems that can be modeled. One tool, HyTech, uses polyhedral computations to study linear hybrid automaton; however, these tools can only be applied to systems with nonlinear dynamics if one approximates the system as a linear one [30, 33]. There are methods for doing this to arbitrary precision, such as epsilon-approximation of hybrid systems with Lipschitz differential inclusions [50, 49]. However, at the heart of this problem is finding the balance between a simple method for describing systems and finding an easy way to examine these systems. By approximating a system, we in essence break down our convenient description for the sake of examination. Many of the approximation techniques require adding additional variables, thus increasing the complexity of the system. Our goal is to apply sampling-based planning as an alternative—a means to study hybrid systems without redefining the system in terms of simpler ones. Consequently, sampling-based planning offers a potential computational advantage over current techniques, since we examine only a sample of the *original* system, which has fewer variables than the appoximation.

In the context of this thesis, **sampling-based planning** refers to methods by which a system can be studied by only examining a representative sample of that system. The underlying assumption here is that it is possible to identify the properties of the system as a whole from a subset of its state space consisting of a number of samples. If we can find such a subset efficiently, our next assumption is that we can analyze the subset in a more efficient way than we could the original system. As a toy example of sampling-based techniques, consider trying to figure out the proportion of red to blue marbles in a bag that contains 1000 marbles that are either red or blue. We could divide all 1000 marbles into two groups and count each of the marbles. Or we could select ten marbles from the bag and check the proportion, assuming that the ten marbles will be representative of the bag as a whole. Alternatively, select 100 marbles and determine the proportion. This example, while unrelated to hybrid

systems, illustrates the balance between finding an appropriately sized sample set and maintaining an accurate study of the system.

We intend for sampling-based planning to apply to hybrid systems by building upon the notion of a **trajectory** within a hybrid system. Informally, a trajectory can be thought of as follows. Given an agent that has a hybrid system description, the agent will experience changes in its continuous elements while in a particular mode. At some point, the agent will encounter a condition causing it to switch modes where it will now experience a different set of changes on its continuous elements. Returning to our manual transmission car, the velocity of the car will accelerate for a time in gear 1 until it reaches a critical velocity where the velocity of the car necessitates switching to another gear. This process will continue (indefinitely) for the car, and one can keep track of the continuous and discrete changes. Consequently, a sequence of continuous change, discrete change, continuous change, discrete change, etc. will form. This sequence is a trajectory.

For many hybrid systems, the number of possible trajectories is infinite as a result of the infinite number of trajectories through the continuous space. Thus, we need a way of studying the set of all trajectories in the hybrid system by examining a sample set of them. Our PRIMARY GOAL is to use sampling-based techniques to accomplish this task. We take a sample of all the possible trajectories through the system and use this subset to answer the reachability question. Of consequent importance is how we choose our sample to maintain the properties of the original system. Hence, in addition to developing a methodology for applying sampling-based techniques, we have a SECONDARY GOAL of improving the techniques themselves so that they are more applicable to problems that are hybrid in nature.

1.2 Contributions and Outline of the Thesis

This thesis presents the results of research involving sampling-based planning to study hybrid systems. The goal of this research is to further study the reachability problem for hybrid systems. We approach this problem from a sampling-based planning perspective by applying the rapidly-exploring random tree algorithm (RRT) [41] previously used in motion-planning problems. In other words, we would like to use sampling-based techniques to analyze hybrid systems and determine safety-critical properties.

1.2.1 Thesis Contributions

This thesis demonstrates that hybrid systems can be modeled in a new way—using sampling-based planning. In particular, it shows how the RRT algorithm can be applied to a hybrid automaton to provide a sense of "walking" through the system. Our methodology does not provide a complete decision algorithm for the reachability problem; however, it does provide a semi-decision algorithm. If the paths grown by the RRT reach a given state, then we can provide a "yes" result, i.e. "yes, the hybrid automata can reach a given state." Because of the approach that sampling-based planning techniques use, if the growth of the RRT has not reached an undesirable state, there is no guarantee that it never will. However, since the technique is probabilistic in nature, if repeated iterations of the algorithm continue to produce a "no", then the probability of ever reaching an undesirable state decreases.

We demonstrate these results by developing an extension of the Motion Strategy Library (MSL) developed by Steve LaValle et al. [42]. This tool has been previously used to provide a generalized framework for development and testing of motion planning algorithms. We extend it to allow motion planning on hybrid systems, and hence use it to apply the RRT to hybrid systems. The MSL is a visual tool; consequently our

extension of the MSL for hybrid systems is a tool used for visualizing the reachability of hybrid systems.

In addition, we also present new results regarding the RRT algorithm. First, we use a modified version of the convex hull (similar to that of a minimal-area hull [4]) to demonstrate general reachability properties of the RRT. We use this hull to investigate how the RRT converges to the reachable area of a state space. Also, we study the optimality of the RRT by making a comparison between the path chosen by the RRT to the optimal (straight-line) path. Finally, we present a synthesis of the concept of a metric tree [57], specifically that of the vantage-point tree (vp-tree) [58] and that of the RRT. This combination provides an improvement to the nearest neighbor query step of the RRT algorithm.

1.2.2 Thesis Outline

This thesis is organized as follows:

Chapter 1 introduces the goals of this thesis and presents its contributions to the academic community. It also provides a quick outline for the reader.

Chapter 2 summarizes background information regarding current research into hybrid systems as well as current research for sampling-based planning. In this section we also provide many useful definitions used throughout this thesis.

Chapter 3 presents our initial experimental research as well as the development of a target application based on the MSL.

Chapter 4 describes our experimentation with our extension of the MSL. In addition to providing different example systems devised by us or based on those in literature, this section also presents the development (based on need) of new features for our software package.

Chapter 5 presents our results regarding RRTs, their potential improvements as well as justification of their reachability properties

Chapter 6 concludes our thesis. Here we review our results and present open issues requiring future work.

Note that some of the work in this thesis has also been published in [16, 17].

Chapter 2

Background

Investigating the reachability problem for hybrid systems involves researching two main areas: hybrid systems and sampling-based planning. We must first have a clear understanding of hybrid systems before we can devise a strategy to study them. Afterwards, we need to understand the techniques involved with sampling-based planning. We plan to combine research in both areas to create a general methodology for applying sampling-based planning techniques to hybrid systems. The following chapter summarizes necessary research from both areas and provides the theoretical groundwork for the solution developed in later chapters.

2.1 Hybrid Systems

Hybrid systems are powerful, flexible tools for describing systems in both real world industry applications as well as theoretical problems. Hybrid systems, and their counterpart model, hybrid automata, can be used to model an innumerable number of different applications. Before we begin discussing them in the context of sampling-based planning, we present a few formal notions and syntax regarding them.

2.1.1 Syntax

In this thesis, we will use the following definitions:

Definition 2.1.1 (Hybrid System). *The term* **hybrid system** *refers to a system that contain both continuous dynamics as well discrete control states such that each control state is associated with a set of continuous dynamics.*

Definition 2.1.2 (Hybrid Automaton). *Given a hybrid system (see Definition 2.1.1), we define a* **hybrid automaton** *to be a formal model of that system. Our model is similar to those presented in [2, 1, 47]. Specifically, for each instance of a hybrid system, our hybrid automaton is* $H = (X, Q, A, E)$*, with each component defined as:*

- *The* **continuous state space**, X, *is the set of continuous variables on which the dynamics of the system occur. A continuous state is a valuation of all variables in* X*. The set of all possible valuations is* Σ_X*. Note, for convenience, we will often use* $\mathbf{x}$ *to refer to the vector of all variables in* X *and* σ *to be a particular valuation of* Σ_X*.*

- *The* **discrete state space** *(sometimes called the set of locations),* Q*, is a finite set of discrete states the hybrid automaton can be in.* $Q \simeq \{1, 2, \ldots, N\}, N \in \mathbb{N}$*.*
  Remark. *At this point note that we can define the concept of a hybrid state space simply by set* $\Sigma_X \times Q$*. Hence, a* **state** *(or* **hybrid state***) in our hybrid system is a tuple* $(\sigma, q) \in \Sigma_X \times Q$*.*

- *The set of* **activity functions** *(also called behaviors [12]),* $A = \{a_i | a_i : \Sigma_X \longrightarrow \Sigma_X \ \forall i \in Q\}$*. The function* $a_i \in A$ *defines how the continuous state changes when in discrete state* $i \in Q$*; hence, they are differential equations comprised of the variables in* X*, which we will notate as* $\dot{\mathbf{x}} = a_i$ *when in discrete state* i*.*

- *A set of* **edges**, $E \subset Q \times Q \times G \times J$ *where* G *is the set of* **guards** *(sometimes invariants) for the edge and* J *is the set of* **jump functions** *(sometimes jump resets, edge resets, or resets). The set* G *is defined as* $G \subseteq 2^{\Sigma_X}$ *where* 2^S *represents the power set of an arbitrary set* S*.* J *is defined as* $J \subseteq \Sigma_X$*.*

  Remark. *Hence, given an edge* $e = (q_1, q_2, g, j) \in E$*,* e *represents the edge from state* q_1 *to* q_2 *that is followed when the continuous state,* σ*, is such that* $\sigma \in g$ *while in state* q_1 *and* j *captures the assignment that occurs to "reset" the continuous variables when following the edge* e*.*

The above definition is a variant of the numerous models used in the academic community for describing hybrid systems. In addition to the sources above the reader will find [12, 55] useful. We justify using our model of hybrid automata because it conveniently provides all of the notions we will use later in this thesis.

Within this model of hybrid automata, we choose to classify instances of hybrid automaton by their different forms of dynamics. Specifically, we are most interested in three types of hybrid automata:

- Rectangular
- Linear
- Nonlinear

Before we can define rectangular hybrid automata we need to define the notion of a differential inclusion.

Definition 2.1.3 (Differential Inclusion). *A* **differential inclusion** *is a set of constant differential equations that are represented by the upper and lower bounds of the set. The common syntax used to represent a differential inclusion on the variable* y *is* $\dot{y} \in [L, U]$ *where* $L, U \in \mathbb{N}$ *are the lower and upper bounds, respectively, by which*

y changes. When $\dot{y}$ is notated this way, it means y may be changing at any rate within the set $[L, U]$.

Remark. *When $\mathbf{y}$ is a vector (notated in* **bold***), we denote a differential inclusion on all rows of $\mathbf{y}$ as $\dot{\mathbf{x}} \in [L_1, U_1] \times \ldots \times [L_n, U_n]$ where $L_i, U_i \in \mathbb{N}$, for all i, and $n = |\mathbf{y}|$, the size of $\mathbf{y}$. Each $[L_i, U_i]$ represents the upper and lower bounds, respectively, by which the i^{th} element of $\mathbf{y}$ may change.*

Given this definition we can now build up to the idea of rectangular hybrid automata. These are hybrid automata that have differential inclusions as their activity functions.

Definition 2.1.4 (Rectangular Hybrid Automata). *A* **rectangular hybrid automata (RHA)** *[35, 51] (also bounded-rate automata [3]) models a hybrid systems where the activity functions are* **differential inclusions** *of the form $\dot{\mathbf{x}} \in [L_1, U_1] \times \ldots \times [L_n, U_n]$ where $L_i, U_i \in \mathbb{N}$, for all i, and $n = |X|$ is the number of different continuous variables in X.*

Remark. *Hence, $A \subset \{a_j | a_j = \dot{\mathbf{x}} \in [L_1, U_1] \times \ldots \times [L_n, U_n]$, for all $j \in Q\}$ where $L_i, U_i \in \mathbb{N}$, for all i, and n is the number of different continuous variables in X.*

For example, if $X = \{x_1, x_2, x_3\}$ and $\Sigma_X = \mathbb{R}^3$ then $n = |X| = 3$ and a sample a_q could be:

$$a_q = \begin{pmatrix} [1,3] \\ [2,4] \\ [-3,0] \end{pmatrix}$$

Thus, in state q we could say

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \text{ is changing by } \dot{\mathbf{x}} = \begin{pmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{pmatrix} = a_q = \begin{pmatrix} [1,3] \\ [2,4] \\ [-3,0] \end{pmatrix}$$

That is, in state q we say x_1 is changing somewhere in the range $[1, 3]$, x_2 is changing in the range $[2, 4]$, and x_3 is changing somewhere in the range $[-3, 0]$.

Definition 2.1.5 (Linear Hybrid Automata). *We use a* **linear hybrid automata (LHA)** *[1, 29, 39] to model a hybrid system where all activity functions are differential equations such that the change of each variable in X is a constant k_i. Each activity function is vector function of the form $\dot{\mathbf{x}} = \mathbf{k}$ where $\mathbf{k}$ is a vector of size n and each $k_i \in \mathbb{R}$. Hence A is a subset of the functions that are linear combinations of the variables in X.*

Remark. *It is important to note that despite their syntactical differences, all rectangular hybrid automata can be described as linear hybrid automata [29]. Given a rectangular hybrid automata, M_{RHA}, we can construct a linear hybrid automata, M_{LHA}, such that M_{LHA} has two variables for each in variable in M_{RHA}. Half of these variables in M_{LHA} will have activity functions that are equal to the set of all L_i in M_{RHA} and the other half with have activity functions that are equal to the set of all U_i in M_{RHA}. This process is discussed in greater detail in [35].*

And finally, we define the notion of nonlinear hybrid automaton. Note that the following definition lacks in formality as opposed to the above. It is important to remember that the union of the set of nonlinear hybrid automata and linear hybrid automata is all hybrid automata. Thus, another way to define nonlinear hybrid automata is by taking the set of all hybrid automata and removing the set of linear hybrid automata.

Definition 2.1.6 (Nonlinear Hybrid Automata). *A* **nonlinear hybrid automata (NLHA)** *[33, 36, 50] models a hybrid system where all of the activity functions model differential equations that can vary with the values of the variables in X. Hence, the activity functions are of the form $\dot{\mathbf{x}} = f(\mathbf{x})$ where $f(\mathbf{x})$ is any nonconstant function.*

Given the above framework for describing and classifying hybrid systems, we are left with a few unresolved issues. Particularly, what properties of hybrid systems can (or should) one study given this description? We formalize these notions in the following sections.

2.1.2 Trajectories

The most important aspect from the above syntax is that a hybrid system is composed of discrete modes that affect how the continuous variables of the system change. From this there is a clear picture of the sequential nature of a hybrid system. The continuous variables, $\mathbf{x} \in X$, change based on the activity function, a_i, while in discrete state i. When a guard condition, $g_{i,j}$, is encountered, the discrete state changes from i to j and the continuous variables $\mathbf{x}$ are reset. This process then repeats, creating a sequence of continuous change, discrete change, continuous change, discrete change, etc. Thus we are given an abstract notion of a trajectory through a hybrid system. We can more formally define this notion now.

Definition 2.1.7 (Trajectory (through a Hybrid Automaton)). *A* **trajectory** *through a hybrid automaton, $H = (X, Q, A, E)$, is a sequence (not necessarily finite) of states (σ_i, q_i), (σ_{i+1}, q_{i+1}), (σ_{i+2}, q_{i+2}), ..., (σ_n, q_n), ... determined by the activity functions A. That is, (σ_{i+1}, q_{i+1}) is determined by using A to change the values of σ_i until a guard condition, $g \in G$ is encountered. When σ_i satisfies g, the edge $e \in E$ is followed from discrete state q_i to q_{i+1}.*

Figure 2.1 is provided to give the reader a visual sense of a trajectory in a hybrid system. This diagram shows three discrete states, $\{i, j, k\}$, and indicates that the continuous variables are changing based on the activity functions, $\{a_i, a_j, a_k\}$, in their respective states. The top diagram shows a plot where one can see the continuous changes occurring in each state based on the activity functions. Dotted arrows are

shown to indicate when an edge reset occurs once a given guard condition is satisfied. The corresponding diagram for the discrete state transitions is shown below. Each oval indicates a discrete state and the activity functions that are occurring within it. Also shown are the edges from states i to j and j to k, which are followed when their guard conditions $g_{i,j}$ and $g_{j,k}$ are satisfied.

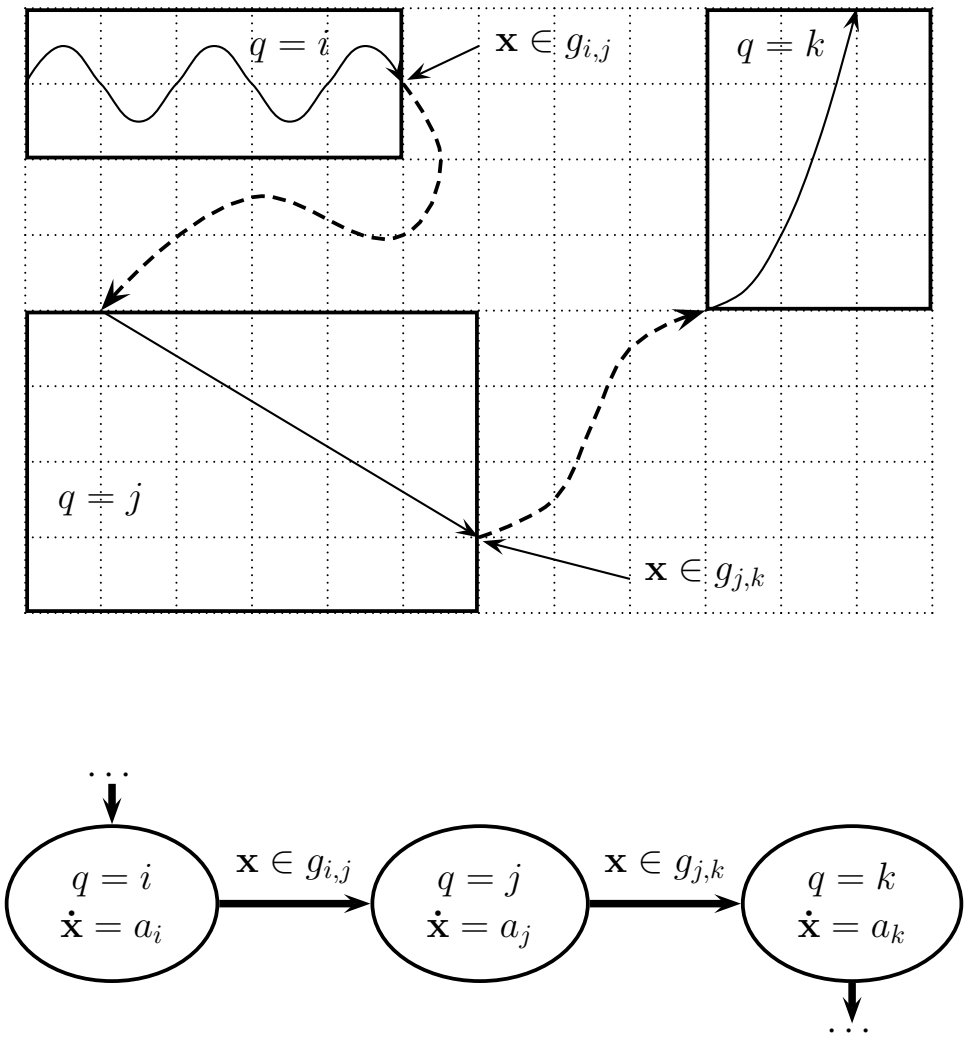

Figure 2.1: A Hybrid System Trajectory, Continuous (top) and Discrete (bottom)

Based on the two diagrams in Figure 2.1, one can see how we can model a hybrid

system with a sequnce of continuous and discrete changes. This sequence is a trajectory. Given a better understand of hybrid systems and their trajectories, we next begin to study the reachability problem in terms of a hybrid system. The following section introduces this concept.

2.1.3 Reachability

One important notion with hybrid automata is that of reachability: can we ever make it to a particular hybrid state (σ, q) starting from the initial state (σ_i, q_i). We say a hybrid system is **decidable** if by analyzing the hybrid automaton of that system, we can answer the reachability question with a definitive yes or no.

Definition 2.1.8 (Reachability Question). *Given a hybrid automaton, H, and initial state (σ_i, q_i) and a destination state (σ_d, q_d), can we construct a Turing machine M_H such that on input H, (σ_i, q_i), and (σ_d, q_d) M_H determines if H will ever reach state (σ_d, q_d) starting from (σ_i, q_i). If H can reach (σ_d, q_d) from initial state (σ_i, q_i), M_H outputs yes, otherwise M_H outputs no. We say a hybrid system is* **decidable** *if such a machine exists for each destination state.*

Remark. *This definition is equivalent to saying there exists a trajectory (σ_i, q_i), (σ_{i+1}, q_{i+1}), (σ_{i+2}, q_{i+2}), ..., (σ_d, q_d) by following the activity functions and/or edges of H to progress from (σ_i, q_i) to (σ_{i+1}, q_{i+1}).*

The reachability question for hybrid automata has been of particular interest in academia over the last ten years. We study reachability because hybrid systems are frequently used to describe real world systems. Often, people are interested showing that the hybrid system is "safe", that is, cannot ever reach undesirable states. Hence, if we can show (by proposing the reachability question) that from a start state, it is impossible to reach any undesirable state, we have verified the system.

Particularly important results regarding decidability of hybrid systems are summarized in [29, 33, 35, 39, 50, 51]. Many different approaches exist to studying

this problem. For example, one method used by various researchers is to calculate a set propagation of the constraints on the locations. Examples of this include ϵ-approximation [50, 49], unions of ellipses [38] or in general of convex polyhedra [1, 22, 48], and level set techniques [18]. To this end, a significant number of tools have been developed that automate these techniques, such as HyTech [34, 30, 31, 32], Kronos [23], Checkmate [52, 53], UPPAAL [9, 40], and d/dt [5]. A description of each of these tools would be beyond the scope of this thesis; however, an excellent overview of each of them can be found at [54].

The above approaches are algorithmic in nature, and hence can fall short in certain aspects of the reachability problem. In particularly, many of the tools limit the description of the input to be of a particular type. For example, HyTech is limited to only accept automata that can be expressed with activity functions that are linear expressions. Also, in some cases, they only will provide a semi-decision algorithm for the reachability problem. In particular, nonlinear systems have been difficult to study, and current research is beginning to focus on effective means to study them. Two popular techniques, clock translation and linear phase-portrait approximation [33, 36] approximate nonlinear hybrid systems to linear hybrid automata. Then these approximations are analyzed by "traditional" means such as the tools above. Clock translation replaces the constraints on nonlinear variables with constraints on clock variable constraints, and linear phase-portrait approximation is a conservative over-approximation of the nonlinear dynamics into a set of piecewise-constant differential inclusions. Both are sound for safety properties. Other techniques already mentioned above for nonlinear systems include ϵ-approximation and level set methods, particularly those demonstrated in [56]. However, all of these approximation techniques suffer from an increase in the number of the states (and thus the complexity of the system) as the nonlinear system is approximated more closely.

In addition to the algorithmic approaches, there exist several other less devel-

oped methods. Some of these involve deductive reasoning instead of algorithmically checking the reachability problem. One particular instance involves using the tool STeP [10, 45] to perform deductive verification. This tool was shown initially successful at some small examples, but the authors note that algorithmic approaches may be preferable because they are fully automatic. Their theorem prover is only semi-automated since verification of more complicated decisions requires use of an interactive Gentzen-style theorem prover. However, the tradeoff for the automatic approaches is they lose generality of application. While deductive approaches have been studied somewhat extensively, the work has been mostly theoretical since a computational approach would lack full automation.

One final approach we found unique was that of a verification tool based on modeling hybrid systems of (2-dimensional) polygonal differential inclusions (or SPDI). Asarin et al. describes a tool SPeeDI that was constructed to study the reachability properties of these hybrid systems in [6]. The methods of this tool attempt to do an exploration of a finite number of trajectories through the hybrid system, as opposed to directly calculating the reach set.

The technique SPeeDI uses is based very much on the input definition of an SPDI. An SPDI defines a finite partition of regions of the 2-dimensional plane where in each region you have a differential inclusion specified by a minimum and maximum angle (direction) you can move while in that region. The tool uses this input information to calculate a sequence of trajectories from the initial state based on the directions you can move. It uses this trajectory to determine when an edge of the next polygon region will be encountered, and then calculates a new trajectory. This simple technique is optimized in cases where the trajectories cause a loop in the regions visited. The idea of following trajectories, instead of reach sets, is like the one presented below, and we are excited to see a second tool using an approach similar to our own.

2.1.4 Conclusions About Hybrid Systems

It is clear from the above description that an extensive amount of research has been done regarding formalizing and classifying hybrid systems as well as studying the reachability problem. Decidability has been shown for various types but is limited on both theoretical and algorithmic fronts. Provable decidability often requires reduction of a hybrid system to either a simple model or making strict assumptions regarding the dynamics of the system. For example, it widely known that rectangular hybrid automata are decidable [35] as well as linear hybrid automata [39]; however, decidability for nonlinear hybrid systems is limited to approximation into linear or rectangular hybrid automata. In terms of the algorithmic fronts, we are met with the same issues; the tools that we have are limited to performing verification on timed, rectangular, or linear hybrid automata.

Hence we are left with the unresolved issue of finding a means to decide the reachability problem for nonlinear hybrid systems. We will return to this idea in Chapter 3.

2.2 Sampling-based Planning

Sampling-based planning aims to determine specific properties of a space based on a sampling of the points within that space. A parallel way of understanding this technique is that we would like to study the whole space by selecting a group of representative elements of the space (either deterministically or randomly). The underlying assumption here is that we can select a sample of the space that maintains the properties of the entire space. Following this belief, one reason for applying a sampling-based approach is to break the "curse of dimensionality" that brute-force approaches (analyzing every element in the space) often suffer from.

The thrust of this thesis is to demonstrate the applicability of sampling-based

planning, particular that of the rapidly-exploring random tree (RRT) algorithm, to hybrid systems. While sampling-based planning can be applied to many types of queries, often sampling-based planning is used in reference to the context of motion or path planning. Motion planning involves determining feasible trajectories throughout a state space. In terms of hybrid systems, we attempt to use sampling-based planning to determine feasible trajectories through a state space that has a hybrid definition. We feel that such techniques are highly applicable, and furthermore we are motivated by one connection of sampled-based planning to hybrid systems in that of Emilio Frazzoli and his colleague's "maneuver automaton". The following section will provide additional background regarding both the RRT algorithm and motion planning.

2.2.1 RRT Algorithm

The RRT algorithm was first introduced in [41, 43] for use in searching high-dimensional spaces. We feel that the properties of the RRT, particularly that of uniform distribution of sampling and expansion biased towards unexplored space [41], lend towards its successful exploration properties. The general idea behind the RRT is that by sampling the space, one can bias the search of the space towards the largest unexplored region, and hence achieve a more efficient search. The way it accomplishes this task is by selecting a random point and taking a small step from the search tree towards that point. In this manner the tree pulls itself towards the largest search unexplored region. We present the actual algorithm below:

Algorithm 2.1 (Build_RRT). *The following algorithm constructs an RRT, T, with K nodes.*

```
1  Build_RRT(x_init) {
2      //initialize tree, T
3      T.init(x_init);
4      FOR k = 1 TO K {
5          x_rand ← Random_State();
```

```
6         //extend T, see Algorithm 2.2
7         Extend_RRT(T,x_rand);
8      }
9      RETURN T;
10  }
```

The above algorithm constructs an RRT by picking K random nodes and each iteration extending the tree T based on the randomly picked node. The sub-algorithm Extend_RRT() (see Algorithm 2.2) is used to extend the RRT at each iteration of the construction loop. It first determines the closest node in the tree to x_{rand} (passed in as input x) using Nearest_Neighbor(). Next, it uses New_State() to create a new node, x_{new}, with input u_{new} to add to T as a child of x_{near}. Upon New_State()'s successful return, it adds a vertex and an edge to T.

Algorithm 2.2 (Extend_RRT). *The following algorithm extends a tree, T, towards x by taking a fixed step from the closest node in T to x towards x.*

```
1  Extend_RRT(T, x) {
2     //find node in T nearest to x
3     x_near ← Nearest_Neighbor(x, T);
4     //construct a new node, x_new
5     IF New_State(x_near, x_new, u_new, Δt) = TRUE {
6        T.add_vertex(x_new);
7        T.add_edge(x_near, x_new, u_new);
8     }
9  }
```

In Extend_RRT(), x_{new} is determined by taking a fixed step of size Δt from x_{near} towards x. The variable u_{new} is returned containing the input (direction) that will take you from x_{near} to x_{new}. In the holonomic case, one can think of u_{new} as the direction of movement from x_{near} to x (as well as x_{new}). In the nonholonomic case, u_{new} is selected from a limited set of potential movement directions (based on the system specifications), and it does not always correspond to a direct path to x.

Instead, a best input will be returned as u_{new}, and x_{new} will be constructed by moving from x_{near} based on u_{new}.

Figure 2.2 shows a sample RRT growth. The image depicts an RRT growing in a disc of radius 50, with a Δt of 2. The images from left to right are at 10, 100, 500, and 1000 nodes, respectively. As is clear in this picture, the RRT grows very quickly and expands to fill the space. This is demonstrative of how the incremental construction of the RRT biases it to explore the largest unexplored region, and in [44] it is shown that the region explored by the RRT converges to a uniform coverage of the entire space. In this sense, even though the RRT could fail to generate a feasible path, it will find a path with a high probability. Hence, the RRT is complete in a probabilistic sense [37].

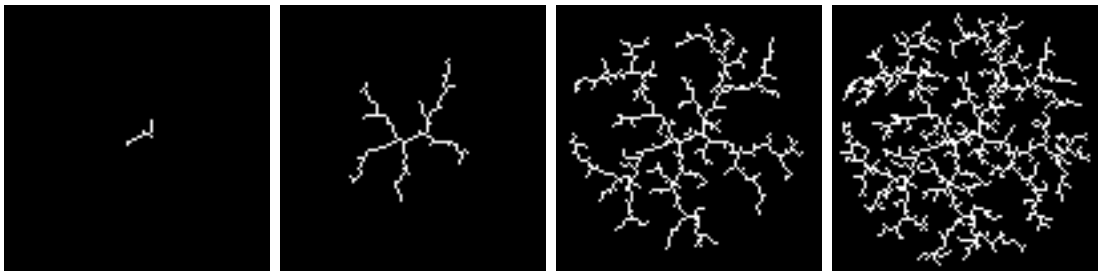

Figure 2.2: RRT Growth on a Disc of Radius 50

There are also some other facts to be noted regarding this algorithm. The New_State() function (implicitly) implements both collision detection as well as the dynamics of the system that the RRT is being grown through. Depending on the system, this may not be the most efficient method; however, in general this calculation can be performed quickly [15]. Second, nearest neighbor queries are one of the bottlenecks to this algorithm. Some solutions to this problem using kd-tree based variants are proposed in [7]; however, we will also return to this problem near the end of this thesis in Section 5.4.

2.2.2 Maneuver Automata

One important instance of sampling-based planning in use is that of motion planning against a hybrid control architecture. The idea is based on a maneuver automaton used to describe the motion of a complex vehicle proposed in [25]. The maneuver automaton attempts to break down a vehicle's motion into a set of different maneuvers. The example used by Frazzoli et al. [25] is that of a helicopter whose dynamics are both in a high dimensional space and have significant nonlinearities. As a result, instead of traditional approaches, they choose a set of maneuvers that the helicopter can perform and describe its motion as this set of primitives. This decomposition of a nonlinear system into a set of fixed motions is reminiscent of linear-phase portrait analysis applied in [33, 36] to discretize a nonlinear system.

Given this approach, the resulting set of maneuvers can then be synthesized to form a hybrid automata, referred to by the authors as a Robust Hybrid Automaton [24, 26]. In this model, states represent the different maneuvering modes that the helicopter could be in (e.g., flying forward, hovering, or turning) while edges execute maneuvers that alter the motion. The key here is that a previously infinite state space has been reduced to a smaller, quantized one. This is a direct extension of the generalization of a hybrid system as a finite number of dynamic behaviors and the rules for switching between them [12]. In particular, this is a powerful extension because we have now gained a concise way of representing motion through the system.

The authors' approach to motion planning for their system was to apply single-query exploration in the form of an RRT. Particularly, they showed the success of this approach in [19, 27]. From this work, we gain insight of how sampled-based techniques could in theory be applied to a hybrid state space to do trajectory planning. This is an important preliminary step to applying sampling-based techniques to hybrid systems. By maintaining the flavor of a motion planning problem, but applying it to a hybrid context, we see the potential for the techniques described in this thesis.

Chapter 3

Development

In the earlier chapters we presented an unsolved question of the generalized reachability problem for hybrid systems. We also presented research regarding sampling-based planning that was one possible solution to this problem. This chapter presents our initial research and resulting choice to develop a tool by extending the Motion Strategy Library (MSL).

3.1 Initial Research

Motivated by the work of Frazzoli et al. [25, 24, 26], we were encouraged to develop a generalized framework for running the RRT on the hybrid automata. In particular, this idea builds on the concept of an hybrid RRT described in [15, 21].

3.1.1 Original Framework

Our first task was to build a framework for RRTs that would allow a transparent level of access to each particular state. Initially, this framework was developed to allow an RRT to grow in an arbitrary state space. To do this, we needed to be able to run the RRT algorithm without knowing any particular information about the state

space itself; hence, we needed to encapsulate two particular aspects for this—that of the state itself and a means to create new states or "take steps".

The first is more straightforward: within a `State` object we need to be able to store location information (particularly that of an individual valuation of the state variables, i.e. a $v = (\sigma, q) \in \Sigma_X \times Q$, see Definition 2.1.2). In addition to storing this information, we needed a transparent interface to provide a metric, a means of determining the "distance" between two states. This is essential for the Nearest_Neighbor() query used in the Extend_RRT() algorithm. Discovering different metrics can in some ways be the most difficult aspect of this process, and in each of our examples we will spend some time discussing the metric used.

The second aspect we encapsulate separately; we use a `Step` object with simply one function to take a step in the given two instances of `State`. In this function we intend to implement the system dynamics; essentially we answer the question "How should one move from this state towards the next?" That is, we want to know how to construct new states given only the source and destination states.

Pseudocode for both objects is included below. In addition, we provide a revised version of the RRT algorithm that indicates how these two objects are used in our framework.

Algorithm 3.1 (Build_RRT (Revised)). *The following algorithm is a revised version of the original RRT construction in Algorithm 2.1 and 2.2.*

```
1  CLASS State {
2     //the particular value of this state
3     valuation v;
4     //determine the distance from this state to s2
5     //the distance from this.v to s2.v
6     FLOAT metric(State s2);
7  };
8
9  CLASS Step {
10    //construct a state, dest, a distance Δt from src
```

```
11      State takeStep(State src, State dest, Input u, Δt);
12  };
13
14  Build_RRT(State x_init) {
15      // initialize tree, T
16      T.init(x_init);
17      FOR k = 1 TO K {
18          x_rand ← Random_State();
19          Extend_RRT(T,x_rand);
20      }
21      RETURN T;
22  }
23
24  Extend_RRT(T, x) {
25      // find node in T nearest to x
26      // careless algorithm checks all nodes
27      FLOAT minDist = MAX_DIST;
28      FOR EACH s IN T {
29          dist = s.metric(x);
30          IF dist < minDist {
31              x_near = s;
32              minDist = dist;
33          }
34      }
35
36      // construct a new node, x_new
37      x_new = Step.takeStep(x_near, x_new, u_new, Δt)
38      IF x_new ≠ NULL {
39          T.add_vertex(x_new);
40          T.add_edge(x_near, x_new, u_new);
41      }
42  }
```

This framework allowed development of a variety of sample experiments. Our first example modeled a 2d RRT which was used to make the screen shots shown in Figure 2.2 above. This code, and all remaining code described in this section was created using C++ and OpenGL and can run on both Windows and Unix platforms. Code was primarily tested on a Linux machine running Mandrake 8.1 Linux with a Pentium II-366 processor and 128 MB of ram. The metric used was a simple Euclidean metric.

3.1.2 Examples

After developing this simple example, we chose to next develop a system with a hybrid state. Opting to follow in the research of Curtiss and Branicky [15, 21], we developed a stair climber example. This example is unique in that it provides a chance to do motion planning with switching based on continuous state and homogeneous dynamics. The stair climber models an agent walking through a four story building at a constant rate (homogeneous dynamics), but who can go up and down the stairs at will (switching based on continuous state). Thus, this situation is a simple example of a hybrid RRT problem.

Below we have pictured the hybrid RRT (see Figure 3.1) for the stair climber in a four floor layout, floors numbering 1–4, left to right and top to bottom. There are two images here, the left shows the initial configuration of the system and the right shows the system after 2000 nodes of the RRT has been planned. The hybrid state in this model is $s = (x, y, q) \in [0, 100] \times [0, 100] \times \{1, 2, 3, 4\}$. Here we take a distance metric of $\rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + 100|q_1 - q_2|$. Δt is given as 5. This is reminiscent of the work done by Curtiss and Branicky [15, 21]; however, the reasoning behind this metric is as yet unexplained. In this metric, we try to balance the Euclidean distance on the x–y plane with that of the actual floor. In essence, we make points picked on the same floor to be weighted as closer to each other (i.e. the factor of $100|q_1 - q_2| = 0$). What we are attempting to do is add a gauge for how much additional distance would need to be covered to get from floor q_1 to q_2. This concept of weighting a metric to weight out the discrete space of the hybrid system is key in developing useful metrics from hybrid RRT problems. We will come back to stair climbers and have a more detailed discussion of them in Section 4.1 below.

After modeling the stair climber, we opted to model another similar system within our framework, that of the peg-in-hole problem. This model is a simplified version of the one studied in [13, 15, 20]. In this problem, we are given a set of layers that

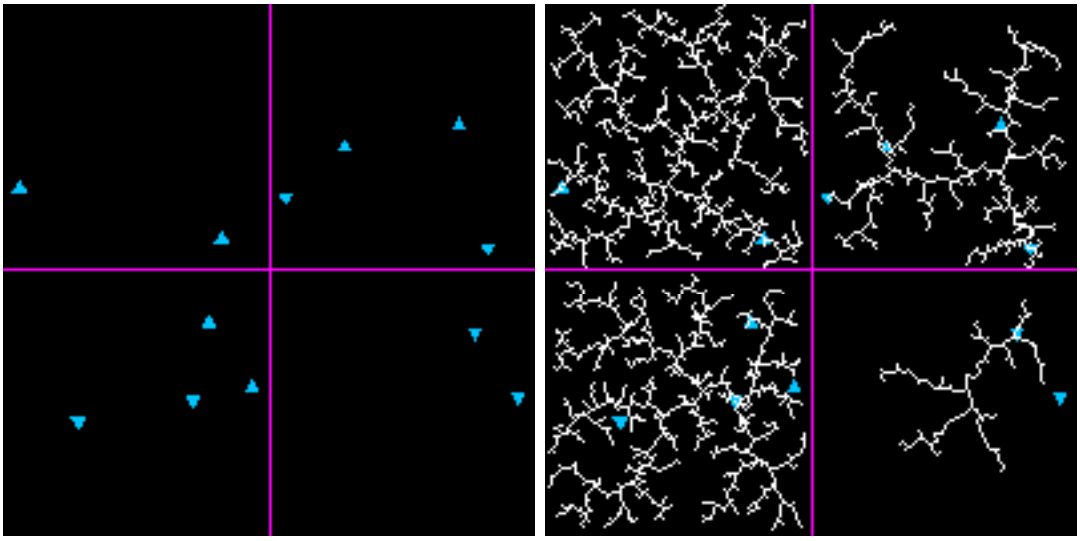

Figure 3.1: Stair Climber Growth

have holes in them that are not in line with each other. By moving a peg left and right, we can slide the layers to align them as we move the peg downward. Here the hybrid state is $s = (x, y, q) \in [0, 200] \times [0, 300] \times \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The discrete state, q, is representative of how many layers deep we have traveled. Here we take a distance metric of $\rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. In our example we chose Δt as 3 and picked random points as integers pairs from the set $[0, 200] \times [-600, 300]$. The y component of these random values was chosen to force the peg to be biased to grow downward.

The images shown in Figure 3.2 depict the initial and final configurations of an example peg-in-hole problem. The peg is green in these images, and the path found is shown in red. The final configuration is the result of 943 nodes being grown in the RRT (shown in white). In this particular example, there are ten layers (depicted as cross sections in blue) that the peg must find a path through. The height of each layer is fixed at 10 while the horizontal specifications of each layer (the offset from the left side, the hole width, and the widths of the left and right halves) are picked at random.

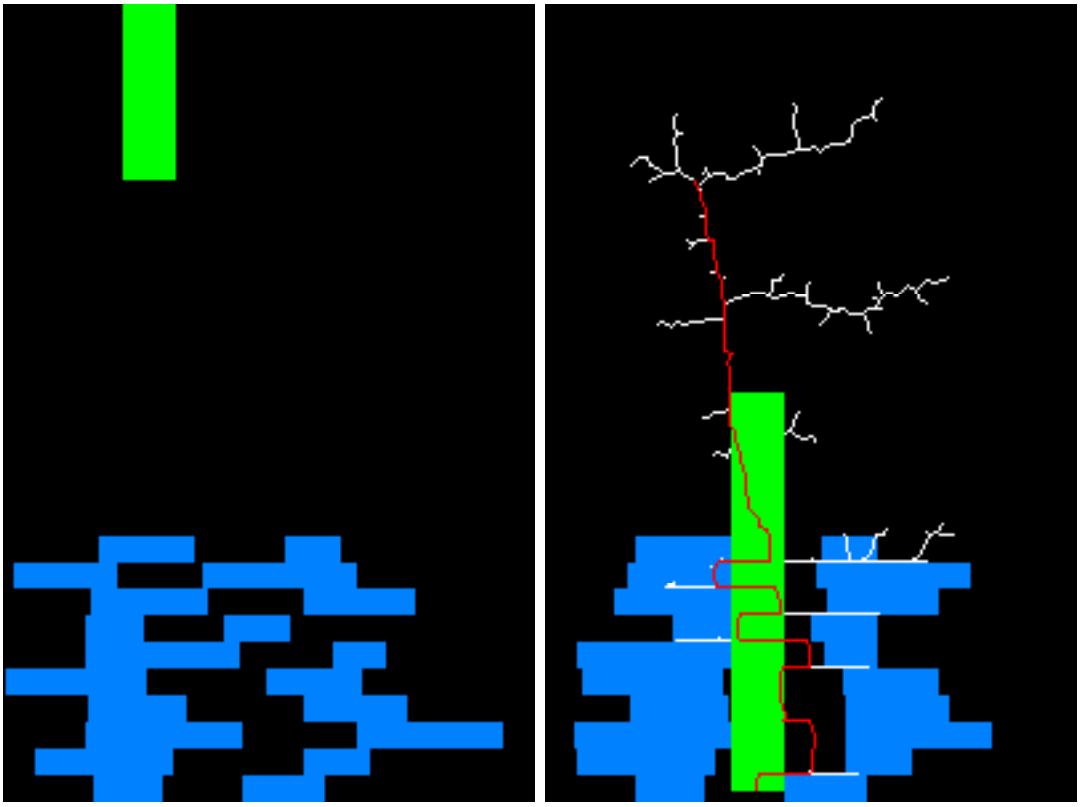

Figure 3.2: 2d Peg-In-Hole Problem, Initial and Final Configurations

This example is similar to the stair climber in that it can only transition to the next lowest layer if we have reached a particular (x, y) value that allows the motion of the peg to continue. However, there are a few subtle differences. The object moving around is no longer just a point in space, but in this model it is a peg of dimensions 20×150. In addition, as the peg move left and right within the maze of layers, the location of the previously entered layers must shift as well. Hence, the x–y motion of the peg is now bounded by the width of all of the layers.

3.1.3 Limitations

Given the initial success of the framework developed, we were encouraged to pursue further work with hybrid RRTs. However, we realized quickly that our framework was limited by the lack of development time and number of people to create it. Basically, it

was limited to a tree structure that wrapped the RRT algorithm and a set of interfaces to create examples. Similarly, designing a user interface was complicated because inputting each new model required programming new classes as well as writing a new interface in OpenGL for representation of the problem. We also realized doing collision detection would be difficult if we wrote our own libraries. (This issue became particularly apparent when working with the peg-in-hole problem.)

One possible choice of a tool was to follow Frazzoli et al. [19, 27] on modeling maneuver automaton. In particular, they made use of the LEDA platform [46] to implement their strategies in C++; however, the LEDA package is no longer freely available for academic use. Instead of their approach, we opted to develop a tool that was based on the sampling-based planning techniques themselves, in particular, the RRT. The next section discusses our development of such a tool.

3.2 Motion Strategy Library (MSL)

The Motion Strategy Library (MSL) provides a backbone for our development of a visual tool to study the reachability problem for hybrid systems. LaValle et al. developed the MSL with the purpose of providing a "general-purpose C++ library for implementing and comparing motion planning algorithms" [42]. In this section, we will explain our choice for using the MSL as well as a brief overview of the MSL itself. (To our knowledge, there is no documentation which presents the MSL other than the informal presentation in [42].) Afterwards, we will explain our extension of the MSL to deal with hybrid systems.

3.2.1 Rationalization

The selection of the MSL was justified in the previous paragraphs; however we will summarize our key points here. In the examples above, our framework lacked:

- Resources to fully develop
- Ease of problem definition input
- User interface
- Generality of application

To this end, we sought a means to apply sampling-based planning algorithms in a general way to many problem definitions. In addition, we wanted a tool that was freely available and straightforward to use. And finally, we wanted a tool that could be applied to hybrid systems that have complex dynamics with a minimum of coding required.

One other key aspect of the MSL is visualization. Allowing the user of the tool to view the problems and their subsequent solutions offers a better understanding of the system as a whole while minimizing time spent learning the system.

3.2.2 Overview of MSL Class Heirarchy

The MSL is composed of three aspects: an interface to input problems of arbitrary dimensions and geometries, as well as the dynamics of these problems; a set of planning algorithms, ranging from probabilistic road maps (PRMs) to numerous RRT variants; and a graphical means for a user to study how effectively the planning algorithms solve these problems. A typical session involves running the MSL against a particular problem, selecting a planner, and using the interface to plan and view paths through the system. Figure 3.3 shows how the objects are contained and the interactions between them. This diagram has been highlighted to show the three main subsystems of the class hierarchies.

Despite differences between our problem definition and the goals of the MSL, this framework is applicable to our needs. We want to use planning algorithms,

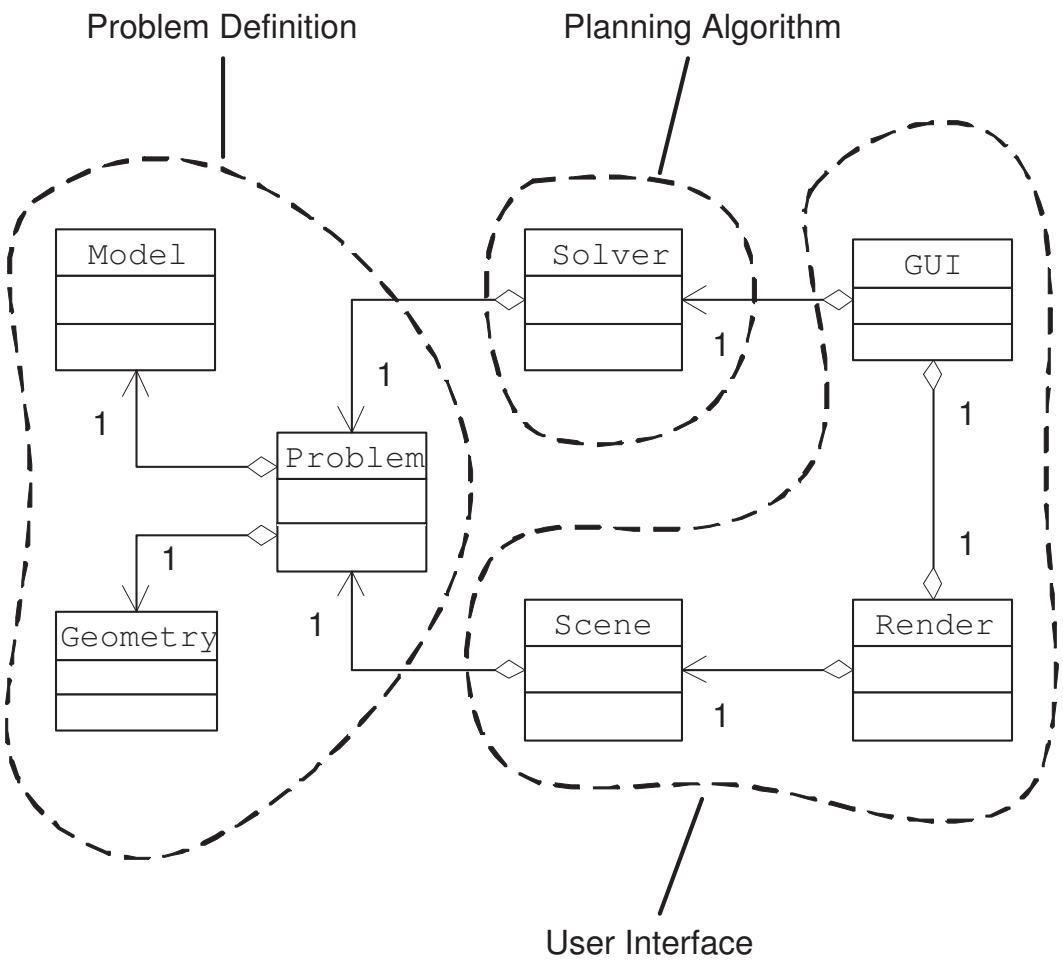

Figure 3.3: Motion Strategy Library Class Hierarchy, Derived from [42]

particularly RRTs, to "walk" through hybrid systems, a similar task to planning paths through a motion strategy problem. The MSL provides straightforward extensibility for all aspects of its design, and serves as the basis for our tool. Our needs require extending all of the three major subsystems of the MSL listed above.

The interface to a problem definition in the MSL is divided into two main objects—a `Geometry` object and a `Model` object. The `Geometry` object contains physical representations of objects in the state space; in a motion-planning problem these include the agent and the obstacles among which it travels. Primarily, its role is to do collision detection checks, to keep potential paths for the agent from intersecting the

space of the obstacles. The `Model` object encapsulates the dynamics of the system, including a metric function for determining distance in the space as well as algorithms to determine future states for the planner given a current state, a time increment and a control input. Given these two objects, a `Problem` object provides an encapsulation of both; it provides an interface for the planning algorithm and user interface subsystems.

The `Solver` hierarchy is a collection of different planning objects that will be used to find motion planning solutions to problems. Under the `Solver` branch there are two main parent objects, `RoadmapPlanner` and `IncrementalPlanner`. `RoadmapPlanner` refers to the class of solvers that are PRM variants and `IncrementPlanner` includes a dynamic programming variant, FDP, and the RRT hierarchy of planners. The RRT branch is the largest in this tree of planners, and it includes variants to grow two trees together as well as ones biased towards the goal. The important thing to note regarding the planners is that there are a significant number of variants; hence in the MSL we are provided a large amount of extensibility with how we actually do sampling-based planning. In addition, we are given a large set of planners to initially work from in designing our own.

The third main portion of the MSL is the user interface. This part is composed of three main objects, the `Scene`, `Render`, and `GUI` objects and their respective hierarchies. The `Scene` object computes the physical locations of objects based on information provided by the `Problem` object. The `Render` object provides a hierarchy of different ways to draw objects based on different graphical libraries. The MSL is provided with `Render` subclasses to work on SGI Iris Performer, Open Inventor, or OpenGL, to allow flexibility of platform. A `Render` object receives most of its information from the `Scene` and `Problem` objects, and given this input it does both the drawing and the animation.

The final part of the user interface is the `GUI` hierarchy of objects that provide

a graphical control window for the user. In the hierarchy there is currently only one `GUI` subclass, `GUIPlanner`, which is used for planning against RRT variants. In this window, users can select different types of planners, modify properties of the planning algorithm, and execute motion planning. In addition, the `GUI` also provides an interface for the user to the commands of the `Render` object by providing a set of animation controls that allow the user to start and stop animation, change the speed of the animation, and the viewpoint. Also included in this is the mouse interface that allows a user to rotate and translate the viewpoint in the rendering window. One last feature of the `GUI` object is to provide the user a way to graph the trees that result from the planning algorithms.

Given the above description, we will provide a brief summary of each of the above seven class hierarchies:

`Problem` is a container class for the `Model` and `Geometry` objects. It provides an interface to the `Scene` and `Solver` classes.

`Model` is an encapsulation of the dynamics of a system, such as a metric for the `Solver` object and a way to compute future states.

`Geometry` is an encapsulation of the physical world. This object is used for collision detection and to provide geometric representations of the agent and obstacles.

`Solver` (and its hierarchy) contain the different types of planners used by the system.

`Scene` is used to compute configurations of objects for the `Render` object by processing information from the problem.

`Render` is used to draw and animate objects based on information it receives from the `Problem` object via the `Scene` object. Its hierarchy is based on different drawing libraries.

`GUI` provides a user interface and control window to the user to set parameters of the `Solver` class as well as view animations and display images from the `Render` object.

3.2.3 Using the MSL

Outside of knowing the underlying structure of the MSL, it is equally important to get a feel for how the MSL is used to solve planning problems. The input to the MSL is highly parameterized, and this section will present a brief overview of its use. Please see [42] for a more complete description.

One should first understand the MSL is used for motion planning, that is, planning of different agents to find paths through a set of obstacle objects given a set of motion constraints. This is slightly different than the traditional path planning problem of finding a path from here to there. A significant portion of the customization that the user has control with is in these objects. All inputs to the MSL are done with ASCII files, and the user is allowed to enter in either a list of polygon vertices or a list of triangles in a simple ASCII format.

Generally, for each motion planning problem, the user will create a directory to store the problem information. In this directory one will store the agent and obstacle representations as well as empty files that are named for the `Model` and `Geometry` objects you want to run the problem against. In addition, there are files containing the initial state, goal state, as well as the lowest and highest possible coordinates. Finally, there are input files for global values such as view position, the planner's Δt, and the number of nodes to plan.

Given all of this input information, one can then run the MSL, and use the GUI control window to do planning. From the GUI window the user can select which planning algorithm they would like to use as well as enter in specific parameters, including the number of nodes to plan and what Δt to use. Next, the user presses the

Plan button and waits for the planning algorithm to finish. Provided the planning algorithm finishes successfully, the user will then be able to use the animation controls to adjust the view and watch the planned path. If the planning algorithm fails, the user is allowed to continue planning additional nodes or switch to a different planning algorithm.

3.3 Extending the MSL

Implementing an extension to the MSL requires extending all three of the main areas discussed above—the problem definition, the planning algorithm, and the user interface. In addition we also needed to extend the input language to provide some additional parameters for hybrid systems.

3.3.1 Problem Core

In terms of the Problem core, we had to implement both a new `Geometry` object and a new `Model` object. Our `Geometry` object contained the same information as a regular `Geometry` object, but also includes geometries for the state transitions of the hybrid system. Each transition was modeled as a pair of (P, q) tuples, (P_s, q_s) and (P_f, q_f), where P_i is a polygon in continuous space and q_i is its discrete state. In addition to collision detection, we included algorithms for "state transition detection". That is, we used the same geometric algorithms for detecting if the planner intersects with an obstacle to detect if we intersected with a state transition (or guard region). However, instead of preventing transition, we use this information to determine when a jump will be taken through the hybrid system.

The implementation of the `Model` object for hybrid systems was accomplished similarly. We created a subclass of `Model` for hybrid systems and overrode the functions for determining the metric to include some metric between two elements of the hybrid

state. In some cases the metric used the discrete information, but was not required to do so. In an example we show later, we chose a metric that was dependent on the continuous as well as the hybrid state. Also, the `Model` object is used for determining future states or "taking steps" throughout the continuous state. Since the `Model` is independent of the `Geometry`, the planning algorithm itself determines when state transitions occur, and reacts accordingly.

In addition to implementing a metric, the `Model` object also contains the counterpart information to the state transitions that are detected by the `Geometry` object. For this, we include information in the `Model` to perform different edge resets when it reaches a guard transition.

One final element of the `Model` object is that of maintaining the dynamics of the system and determining new states based on source and destination states. This information is also encapsulated by our `Model` object, primarily in two functions, Integrate() and InterpolateState(). These two functions are used by the `Solver` object to create new states, and hence were modified accordingly in our `Model` object. Integrate() is used by the Solver in calculating a new state that is a fixed distance along a given input vector from some start state. InterpolateState() is used to interpolate two states and returns the input vector from one to the other based on the topology of the system.

One should note that the `Model` object, and in some respects the `Geometry`, are very dependent on the problem definition. This is apparent in the hierarchies of `Geometry` and `Model` objects that are contained within the standard MSL. In our case, we will often end up developing a new `Model` object for each different problem we expect to solve (or at the very least class of problem). Part of the problem here is that we need to implement different metrics as well as problem dependent information.

3.3.2 Solver

In addition to the `Model` and `Geometry` objects, we also must implement a new type of planning class that plans against hybrid systems. As stated above, the MSL contains a `Solver` hierarchy of classes in which a major branch is a collection of RRT planners. Our hybrid RRT [15, 21] extends directly from the RRT branch, and plans in a similar manner to all other RRTs used in the MSL. However, one important difference is that at each iteration of the algorithm, the RRT also does a check to see if a state transition has occurred. In effect, it queries the `Geometry` object, and if the newly planned state for the RRT "collides" with a state transition polygon, then it adds an additional node to the RRT and follows the jump, performing any necessary resets. We present a modification of the Extend_RRT() below (for original Extend_RRT() see Algorithm 2.2):

Algorithm 3.2 (Extend_RRT (Revised)). *The following algorithm extends a tree, T, towards x by taking a fixed step from the closest node in T to x towards x.*

```
1  HybridRRT.Extend_RRT(T, x) {
2     //find node in T nearest to x
3     x_best ← Select_Node(x, T);
4     //construct a new node, x_new with input u_best
5     u_best ← Select_Input(x, x_best, x_new);
6
7     //is there a state transition?
8     IF Problem.StateTransFree(x_new, x_new2) = FALSE {
9        //node growth causes a state transition
10       //perform a reset, create x_new2 and u_best2
11       Problem.EdgeReset(x_new, x_new2, u_best2);
12       //now ok to add, insert
13       T.add_vertex(x_new2);
14       T.add_edge(x_best, x_new2, u_best2);
15    } ELSE {
16       //no transition, just insert
17       T.add_vertex(x_new);
18       T.add_edge(x_best, x_new, u_best);
19    }
```

```
20 }
```

Note a few minor differences in this algorithm. First, instead of calculating a near neighbor, the MSL uses Select_Node() to encapsulate this, and then uses Select_Input() to construct a new node and the input one would take from going from x_{best} to x_{new}. This is how the original MSL performed the Extend_RRT() algorithm. In addition, we added the **if** statement (line 7) to perform a check to see if a state transition should occur. Here, the hybrid RRT queries the `Problem` object that in turn queries the `Geometry` object for state transition information. Here, x_{new2} is returned with the resulting node from taking the transition from x_{new}. If a state transition has occurred (i.e. the new node is not transition free), then hybrid RRT queries the `Problem` object which queries the `Model` object to determine what should happen when the state transition jump occurs (i.e. when the system takes the edge from x_{new} to x_{new2}).

3.3.3 User Interface

The final addition to the MSL is developing the user interface to include information for hybrid systems. Our needs require extending both the `Render` and `GUI` object. The `Render` object provides an implementation independent interface for rendering the physical representations of objects in a 3d world. Hence, it works closely with the `Geometry` classes. The developers of the MSL include sample `Render` objects for a variety of platforms, but for our purposes we chose to work with the OpenGL library. The `GUI` object provides a windowed interface that allows the user to easily specify inputs for the planning algorithms as well as controlling how the `Render` object animates the paths that are planned; hence, we needed to modify it to allow for special controls related to hybrid automata.

Our `Render` object for hybrid systems involved several changes. Specifically, we

needed to modify the input language to include discrete state information, and hence needed to modify how the `Render` object accessed the input files for the MSL. Currently, the MSL uses a set of ASCII text files as a convenient means for input; extending the `Render` object to read our new input language involved reading an extra discrete value for state information. In terms of drawing bodies, we chose to draw them on a one-state-at-a-time basis; hence, displaying each body involves checking if that body's state is the current state, and drawing them. Similarly, animation involves storing state information for each frame, and only animating if the state is the one currently being viewed. A big addition to the `Render` object included drawing not only the path planned for the problem, but also drawing the RRT as it is being constructed as an optional way to view the problem.

Our modified GUI object as compared to the original GUI window is shown in Figures 3.4 and 3.5. New controls have been added to allow for enabling the new features listed above in our `Render` object description. In addition, a control has been added to cycle through which state you are currently displaying. One important aspect of the new `GUI` window is the modified planning controls at the top bar. We have improved the MSL to allow planning to occur in a threaded fashion so that the windows provided by the `GUI` and `Render` objects do not freeze while waiting for the `Solver` objects to finish planning. There are additional features that are shown in this window that have not been completely explained because more improvements to the MSL were made incrementally as more complex hybrid systems were attempted against the extended MSL. Hence, discussion regarding them is saved for later sections, where they become more significant from this development point of view.

3.3.4 Discussion

For the benefit of the reader, below we summarize a list of new features that have been discussed. This list does not include features to be described in the next chapter.

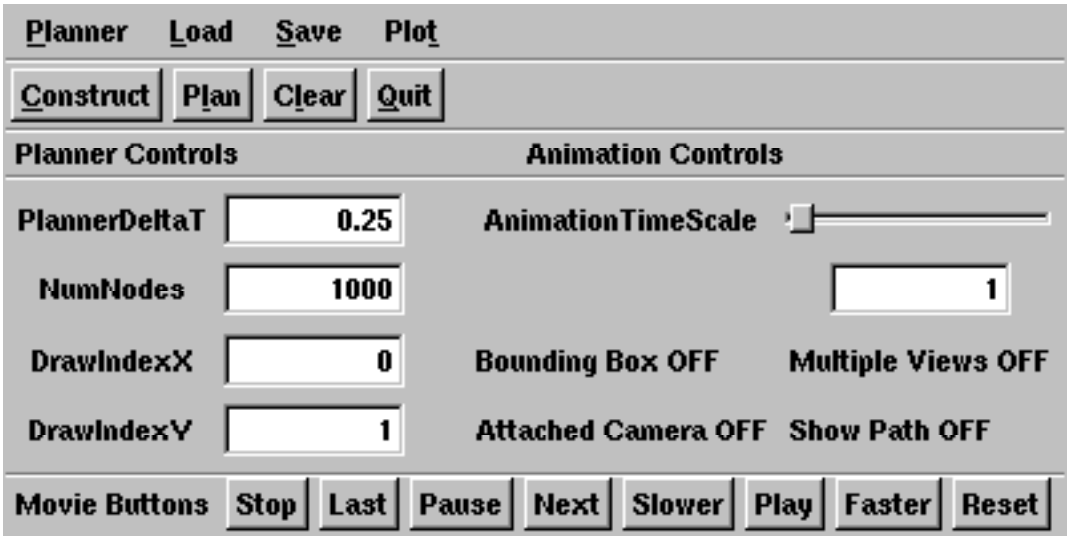

Figure 3.4: Original GUI Window

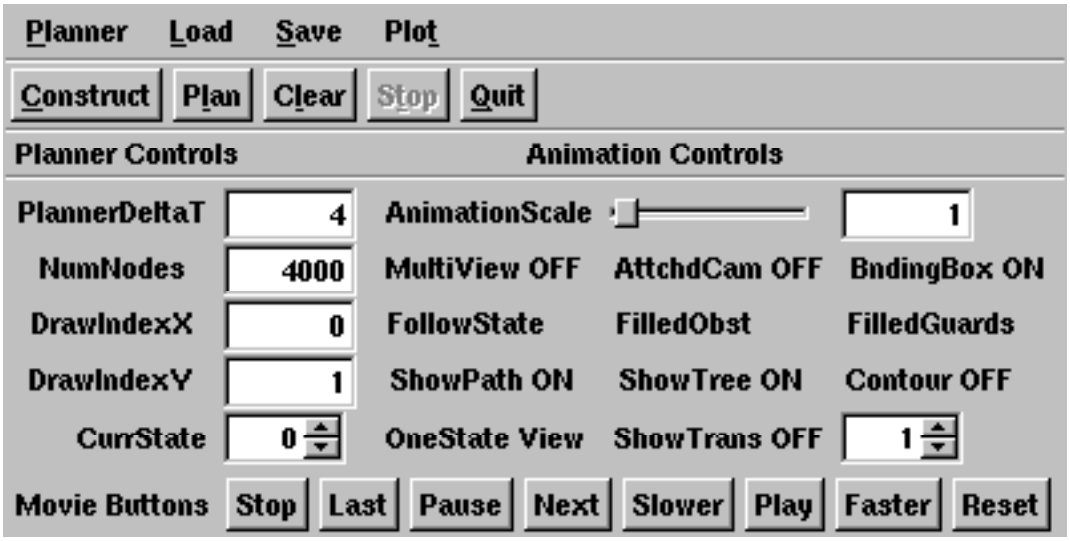

Figure 3.5: Extended GUI Window

- The `Geometry` object was extended to support hybrid states via a state transition free check.
- The `Model` object was extended to support hybrid states via resetting on the edges and new metric functions.
- The `Solver` object was extended to support hybrid RRT planning and include

state transitions.

- The `Render` object was extended to support state based drawing.
- The `Render` object was extended to accept state-based input.
- The `Render` object was extended to allow drawing of the RRT during construction in addition to the path found.
- The `GUI` object was extended provide state selection.
- The `GUI` object was extended to allow planning to occur in a separate thread.

One final note is that we did not want to lose the original functionality of the MSL or hide any of its original features. This is the motivation behind extending the library by subclasses of each of the above classes instead of just overwriting them. In each directory for the different hybrid system problems, we included a file named UseRenderHS to make sure that we use the extended versions of the `GUI` and `Render` objects when we are working with hybrid systems.

Chapter 4

Experimentation

To develop our tool and prove it useful, we generated some different experimental hybrid systems to study against the tool. The following sections describe those experiments as well as specific improvements to the MSL that became necessary to accommodate these examples. We discuss the problems in terms of scope of their complexity—beginning with those problems that have hybrid states, but with simple dynamics, and then we progressively increase the complexity of the dynamics of the systems.

4.1 Stair Climbers

To test our extended MSL for hybrid systems, we have done example work using a four-story building, similar to the one used in previous works devised by Curtiss and Branicky [15, 21]. This example is important because it bridges the gap between a motion planning problem and a hybrid systems problem by extending the state space of a continuous problem to that of a hybrid problem.

Our setup is as follows. Given a four story building ($Q \in \{1, 2, 3, 4\}$), we attempt to try to devise a path from the lowest floor to the highest traversing different sets of staircases. Here the guards are simply regions that represent a staircase up or

down a floor. The stair climber example parallels that of a holonomic planning problem; the climbing entity can move in any direction in the continuous space in an unrestrained manner. Hence the activity functions for this system is the set of all possible trajectories in the continuous state space.

Given the RRT algorithm, we approach this problem in the most direct manner. We set an initial state somewhere in the bottom floor, $q = 1$, and the goal state as a location in $q = 4$ and grow an RRT. When the RRT reaches any of the switching regions or "staircases", we transition and change the value of q accordingly—inserting a new node into the RRT that has the same continuous configuration but a new discrete state. Essentially, the RRT is used to fill up the continuous state in each floor, and is allowed to reach addition floors when it reaches switching regions.

In [15, 21], Curtiss suggests use of a metric function where we introduce a scaling factor k such that $\rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2} + k * |q_1 - q_2|, k \in \mathbb{R}^+$. The idea here is to take the Euclidean distance of the continuous space, but at the same time provide a constant factor modification based on the discrete space. This metric is an intelligent choice, and it works well provided two conditions exist:

1. k must be picked such that $k > \delta(x_{max}, x_{min}) - \varepsilon$ where δ here is the common Euclidean distance and ε is "small" compared to $\delta(x_{max}, x_{min})$.

2. x_{rand} returned by Random_State() in the Build_RRT() algorithm must be chosen to be an even distribution of both the continuous states as well as the discrete ones.

Condition 1 attempts to balance the metric to give a sense of how much further away a point is if it is located on a different floor. It essentially adds a factor of how much distance is necessary to cover to move between those continuous states as well as to find a state transition. The bound on k assumes the worst; that one must travel the maximum distance in the continuous space to find a state transition. The goal of

condition 2 is to ensure the RRT continues to grow once it has reached new states. If all points were picked with an uneven distribution, exploration of the continuous regions in particular discrete spaces would not occur. However, condition 2 can be relaxed depending on the specific need of the problem. For example, in the examples below, by picking points only with $q = 3$ always, we bias the RRT to grow up, and hence speed up growth towards that floor. The flaw in this is that once we have reached a specific floor, we limit the RRT from exploring lower floors.

4.1.1 Two Dimensions

Our first example is a direct translation from the experiment done with our initial framework above, but was necessary to ensure our extended MSL was capable of performing the experiments we needed. Specifically, states in our system consist of a two-dimensional coordinate combined with a discrete floor. Our hybrid state space is $s = (x, y, q) \in [0, 50] \times [0, 50] \times \{1, 2, 3, 4\}$. We use distance metric $\rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + 50|q_1 - q_2|$. Given these as inputs to our model, we grow an RRT via the MSL to get a result like the one shown in Figure 4.1.

In this figure, which shows only floor 3 of the state space, the red peg represents a simple point object (as a tiny square of width 1). The light blue squares are state transitions or "down stairs" in which q is decremented while the dark blue are "up stairs" where q is incremented. These squares are of width 6. The MSL draws all objects in 3d space, and consequently, 2d problems are drawn by translating the problem onto a 3d space. In this case, all objects have been translated to have a height of 5 units. The white lines represent the RRT that has been grown through the system and the red represents the path determined by the RRT. A representation of each floor of the problem is shown in Figure 4.2. Floors are numbered 1 to 4, left to right and top to bottom.

As described above (see Section 3.3.2), state transitioning is done using collision

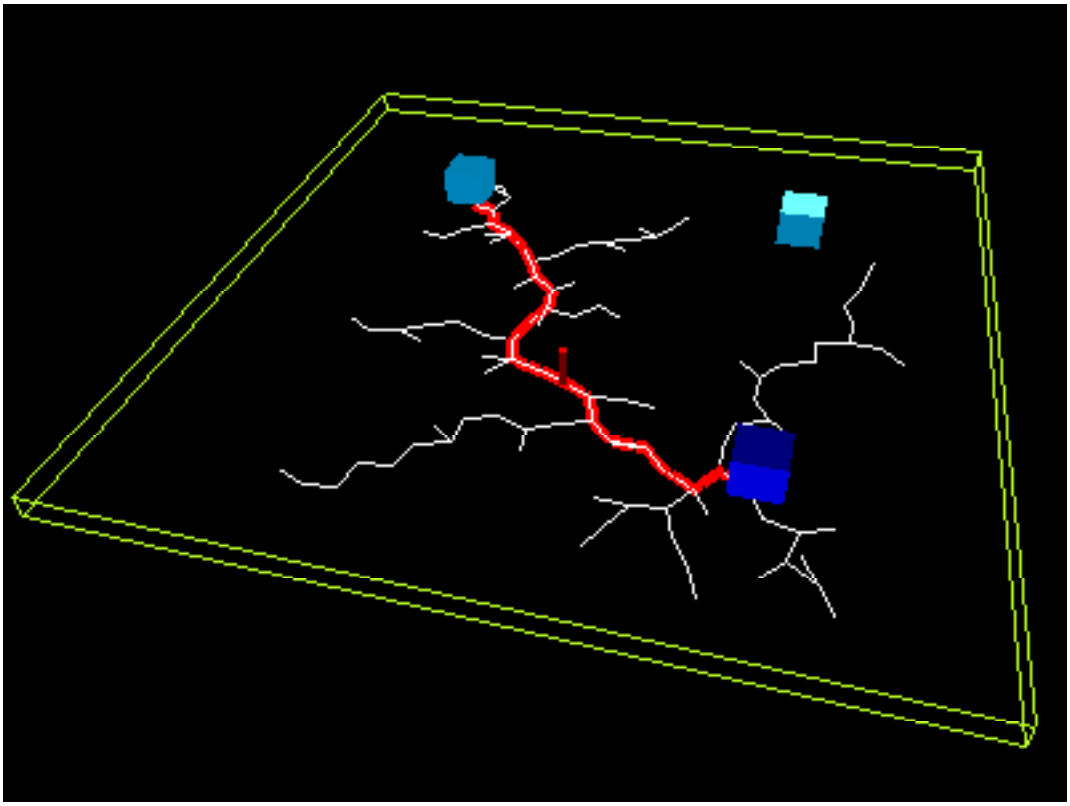

Figure 4.1: MSL Example of a 2d Stair Climber, Floor 3

detection between the agent and the squares of the transition regions. As the RRT plans its way through the floors of the example, if it encounters a staircase, it adds a node in the tree on the corresponding state where the other end of the staircase is. Staircases here are treated as regions which map two different floors together, acting like tunnels to different floors. In our examples, we make transitions to adjacent floors, but the implementation is open enough to allow a guard region to cause a transition from any floor (state) to any floor (state).

We also felt it was important to make use of the collision detection (for its original purpose) by trying out obstacles to plan around. Figure 4.3 demonstrates a more challenging placement of the switching regions as well as obstacles in the two dimensional stair climber problem. Again, floors are numbers 1 to 4, left to right and top to bottom. This example has two equidistant staircases on floor 1, but an L-shaped

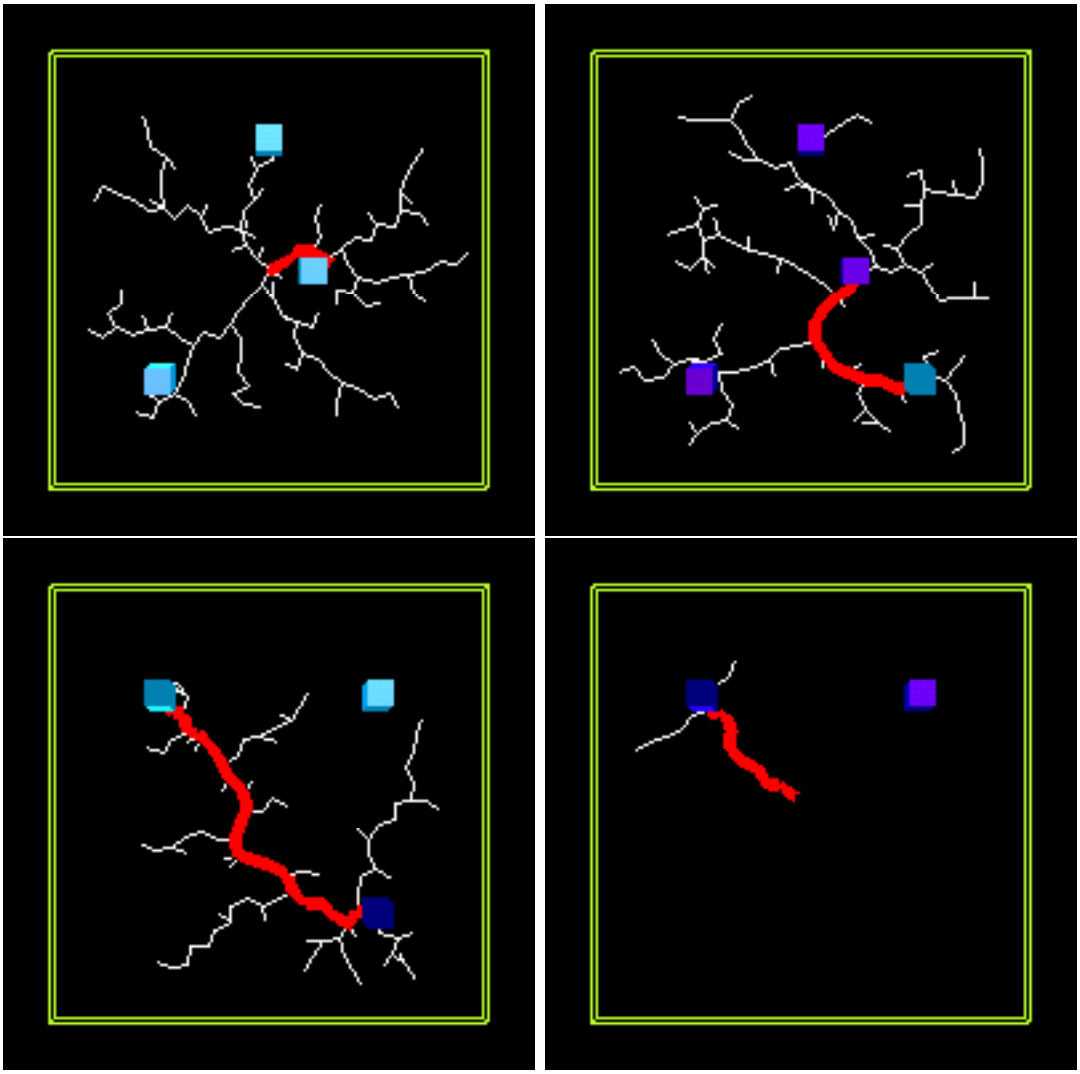

Figure 4.2: Four Floors of the 2d Stair Climber

obstacle — shown in magenta — mostly blocks one. However, if the RRT makes it around the obstacle, floors 2 and 3 will have stair cases located close to each other allowing the agent to quickly make it to the final floor, $q = 4$. These staircases have been walled off with additional obstacles to prevent access except from the staircase on floor $q = 1$. If the agent does not find the staircase in the lower right, it must traverse the entire width of the floor at each point to make it to the final floor.

In this example, choosing the Random_State() function correctly has a significant impact on the results of the planning. If we only choose random states with $q = 4$ the

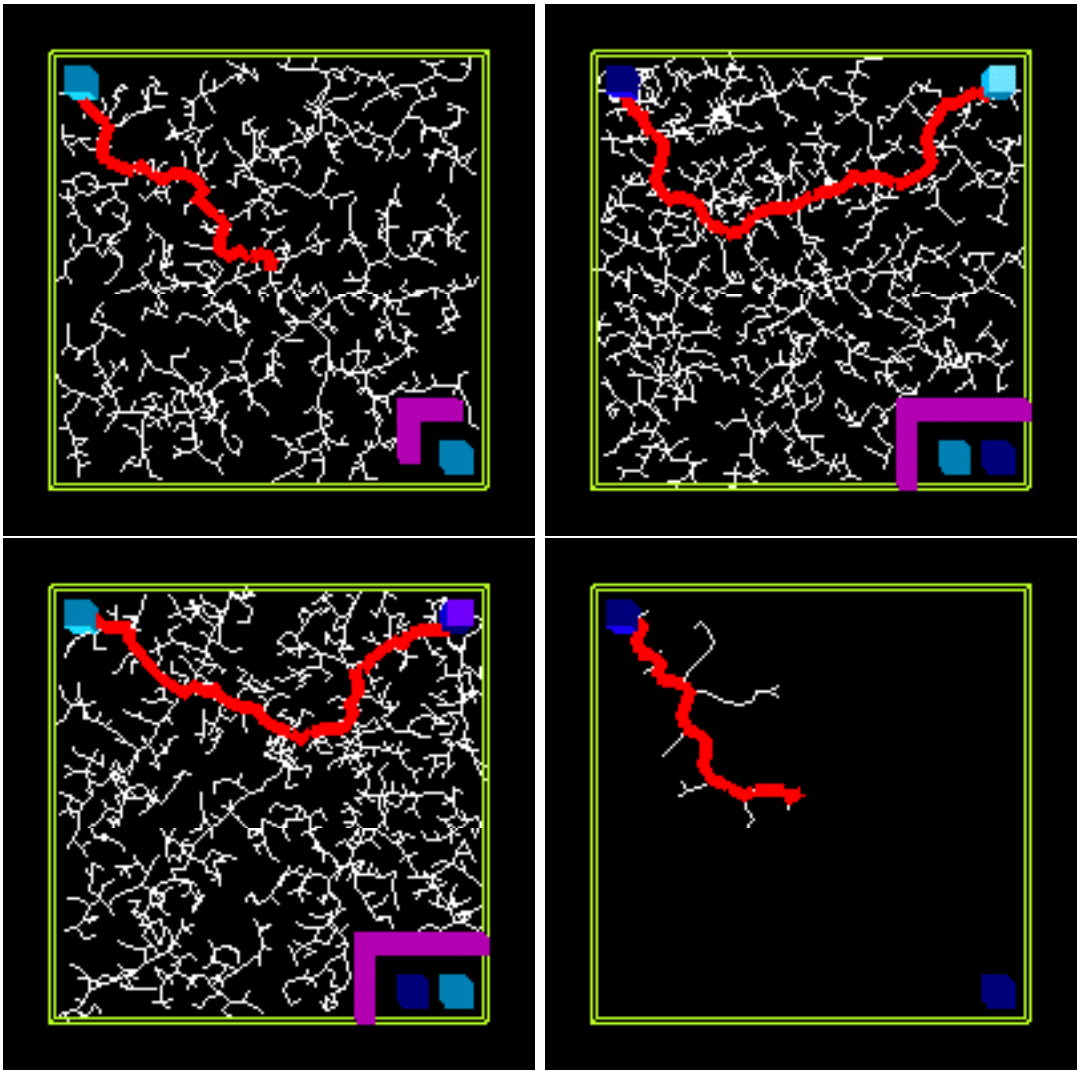

Figure 4.3: Four Floors of the Second 2d Stair Climber

RRT may quickly find the staircase in the upper left, and if so will never return to explore floor 1 and find the partially blocked one in the lower right. This is an example of how both conditions 1 and 2 discussed above (Section 4.1) present tradeoffs for the weighted Euclidean metric used by Curtiss. We present more discussion regarding these ideas in the next section.

4.1.2 Biasing Exploration

The MSL was originally designed with path planning in mind; consequently, the planners it uses are made to grow until a path to the goal state is found. This presents a conflict in goals: we would like to have a tool that can check if a particular trajectory exists, but we would also like to have a tool that achieves full coverage of all of the possible trajectories. In addition, a third goal is not only finding trajectories, but providing control over the optimality of the trajectories themselves. This section explores the balance between these goals. To this end, we have extended our RRT for hybrid systems in a few different ways as well as devised two more 2d stair climber problems.

One method we have already mentioned for biasing the growth is by adjusting how we pick states for our Random_State() function. Here, we adjust the distribution of the points fed into the RRT planner causing changes in the way it grows. This fine tuning provides delicate control over the way the planner fills out the space.

An example for biasing Random_State() that is intrinsic to the problem is by selecting random points such that $q = 4$ for every point. This biases the RRT to grow upward continuously, as the metric we use selects points by assuming that closer floors (smaller Δq) are better choices to grow from. Thus, depending on the value of k in our metric, as soon as the RRT finds a path from floor $q = 1$ to $q = 2$ it will explore $q = 2$ and limit continued exploration of floor $q = 1$.

Table 4.1 shows the results of running ten trials to compare picking random states where q is on every floor ($q \in \{1, 2, 3, 4\}$) or where q is on the top floor ($q \in \{4\}$). The first and third column list total nodes planned, while the second and fourth list the number of nodes in the path to goal (the path from state $x = (0, 0, 1)$ to $x = (0, 0, 4)$). As expected, the growth that picks points on only the top floor found a path to the top using fewer total nodes (compare the averages of 1455.4 nodes to 4679 nodes). Fewer nodes correlate directly with smaller planning times, but at a cost of both

$q \in \{1,2,3,4\}$		$q \in \{4\}$	
total nodes	nodes in path	total nodes	nodes in path
3169	133	1772	159
5288	154	1300	131
8061	156	962	141
4062	156	1927	147
3400	135	1503	150
5141	67	1769	157
3039	142	1216	139
2390	135	1418	137
7591	140	1489	139
4649	150	1198	127
Averages			
4679	136.8	1455.4	142.7
Standard Deviation			
1906.965	26.11428	301.2825	10.47802

Table 4.1: Random States Picked with $q \in \{1,2,3,4\}$ vs. $q \in \{4\}$

optimality and complete exploration of the system. Note that in row six, we see that the planner that picked nodes on every floor found a path to goal that was only 67 nodes. This small value is indicative of a trial where the RRT found the shortcut through the path, around the L-shaped obstacle on the first floor. The planner which picked nodes only on $q \in \{4\}$ never found this path. This point also contributes to the high standard deviation for the trials where $q \in \{1,2,3,4\}$.

Biasing based on the value of the discrete state presents an improvement towards finding if a single trajectory exists. When we pick $q = 4$ only, we pick so that we force to grow the RRT to the top floor, where the goal state is. Focusing on our second goal of exploring all possible trajectories, we change our bias methodology. Assuming that interesting trajectories happen at the state transitions, we bias our growth so that a proportion of the random states we select are points that go towards the guard regions. Or, more generally, we have a list of subgoals that the planner uses. Based on a bias percentage, our Random_State() function either picks a node at random or selects one of the subgoals to grow towards.

To accomplish this task, we provide an additional input to the planner that is a list of the subgoals (as points) as well as two additional parameters for each subgoal. The first is a probability to select that subgoal; the second is a radius to select around. This way, we can bias our planner to grow towards a particular area (within the radius of the subgoal) at a particular probability. Our Random_State() function is modified as follows. First we use the bias percentage to check if the point we return should be completely random or biased based on the subgoals. If we pick a bias, we then select one of the subgoals (based on their individual probabilities) to return. Based on this subgoal, we pick a point at random within the selected radius of the subgoal. Otherwise, if we are not returning a subgoal, we just return a point at random.

In Figure 4.4 we show two different plots where we have varied the bias percentage. Similar to the trials we ran above, we ran ten trials and took the averages for the total number of nodes planned as well as the path length (in nodes). We ran our tests using bias percentages in the set of $\{0, 0.05, 0.1, 0.25, 0.5, 0.75\}$. The plots above show how total nodes planned varied as well as path length, based on the bias percentages. For the set of bias percentages, we ran two sets of six trials in total, in the first set we returned a random state of $q \in \{1, 2, 3, 4\}$ and in the second we returned random states of $q \in \{4\}$. Where we did not return a random state, we selected a subgoal at equal probability, and returned a state that was in a radius of 0 of the subgoal. The list of subgoals we used were the points at the center of each guard region as well as the goal state itself.

These two plots show multiple results. First, if we use any bias, even a slight one, we gain a significant reduction in number of nodes needed to plan as well as a reduction in the path to goal. Consequently, we can interpret this as a more optimal path in less computation time. However, increasing the bias more than 25% did not significantly improve either of these factors. In fact, for the case of $q \in \{1, 2, 3, 4\}$, increasing the bias to 75% caused total number of nodes to increase. We can also

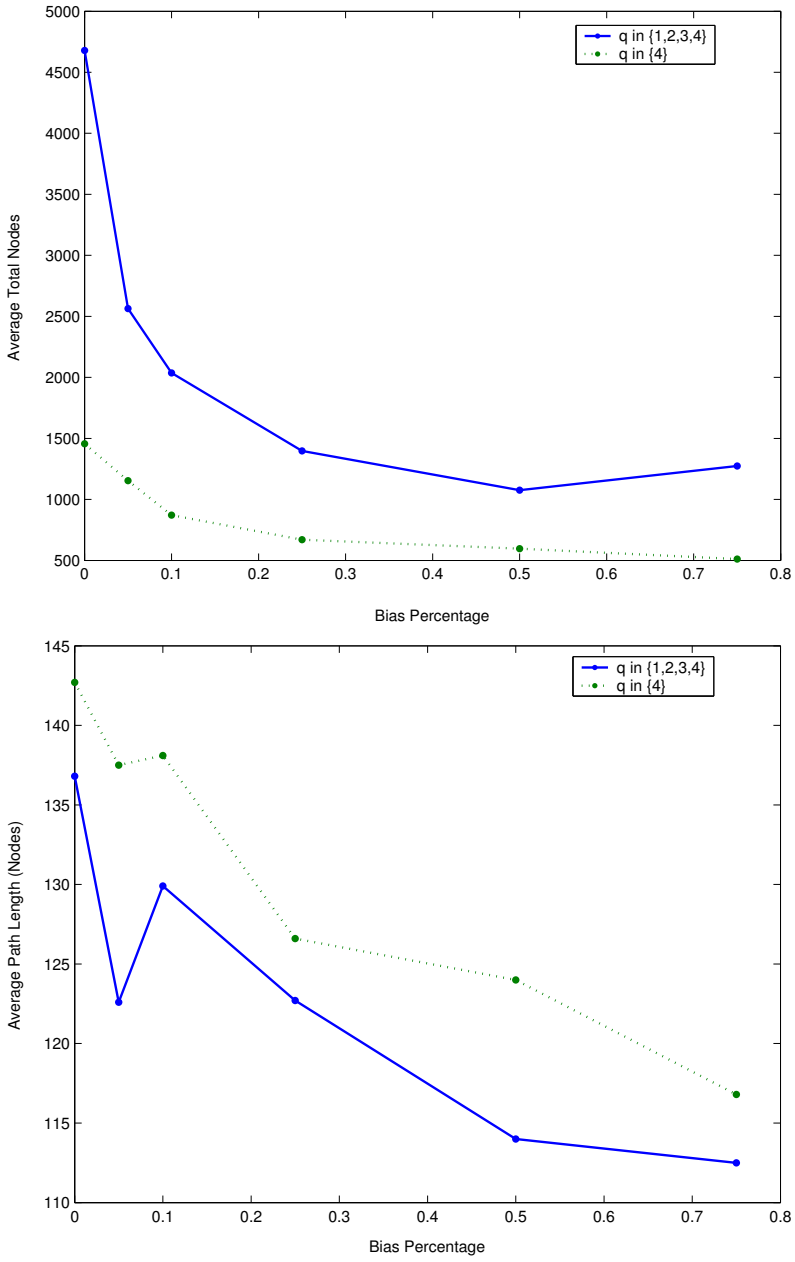

Figure 4.4: Plots of Total Nodes and Path Length vs. Bias Percentage

see here again that by selecting random states with $q \in \{4\}$ we gain a reduction in total number of nodes planner, but we end up growing slightly less optimal paths to the goal. For the less biased examples, this is a result based on averaging in the occasional shortcut. For the more biased examples, this is a result of having the incorrect subgoal pull the path away from the optimal. For example, on floors $q = 2$

and $q = 3$ the staircases in the lower right pull the path downward.

Another important result is that in only one case did any of the biased planners find the shortcut path ($q \in \{1, 2, 3, 4\}$ for a bias percentage of 0.05 — note the dip in the path to goal line). The reason here is that the bias caused the RRT to grow towards the guard regions; however when an obstacle was encountered, the RRT would walk right into it instead of around. By reducing the number of completely random points selected, we reduced our chances of making it around the L-shaped obstacle on the first floor. From this, it is apparent that while the biased planners performed both faster and with more optimal paths, they explore the region less completely.

In an effort to improve the exploration of the reachable area, one modification we made to the planner is for it to have reached every subgoal before it stops planning. The MSL itself plans until it reaches one goal state, so we modified this stopping condition to be when all subgoal states are reached. In addition to this, we also created two new two-dimensional stair climber obstacle layouts. Figures 4.5 and 4.6 show both of these layouts along with sample solutions. Note while both examples have goal states in the center of floor 4, there are no paths drawn because in these examples we grew to reach every subgoal, not a particular goal. Therefore, a single path to a goal state does not coincide with the planner used in these experiments.

The layout in Figure 4.5 has a narrow passageway across the diagonal of the first floor which opens up in the far corner. Across the other diagonal is an additional pair of stairs that lead to shorter paths up to the top floor. The initial placement of the agent is in the bottom right corner, so it must traverse the entire passageway before reaching a break where it can go to one of the three transitions. The steps in the top right lead to the shortest path up to the top floor, while the steps in the top left lead to a path where the agent must traverse the diagonal of each floor. The steps located in the bottom left are unique in that the agent must go up to floor 3 and then back

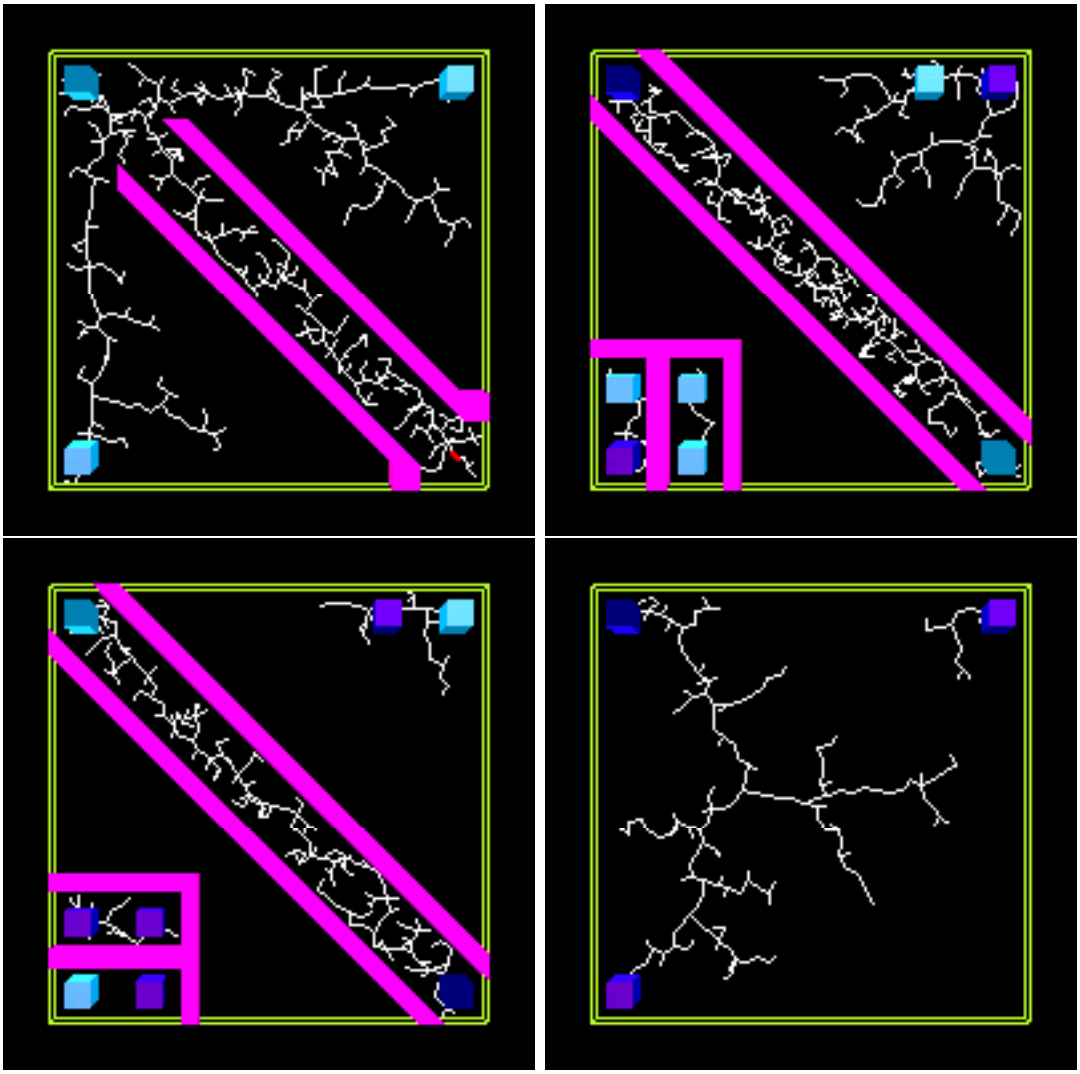

Figure 4.5: Four Floors of the Third 2d Stair Climber

down to floor 2 to continue up to the top floor. There are a total of 23 subgoals in this layout; each of the twelve steps has a pair of subgoals as well as the true goal state in the center of floor 4.

Figure 4.6 presents a layout where the agent is presented with a staircase almost immediately, and a large wall that it must navigate around to get to the next goal. The agent is initially located in the top right. The steps on the middle of the right wall lead to a path where the agent must navigate throughout the width of each floor, while the steps in the lower right lead to a short path up to the top floor. There are

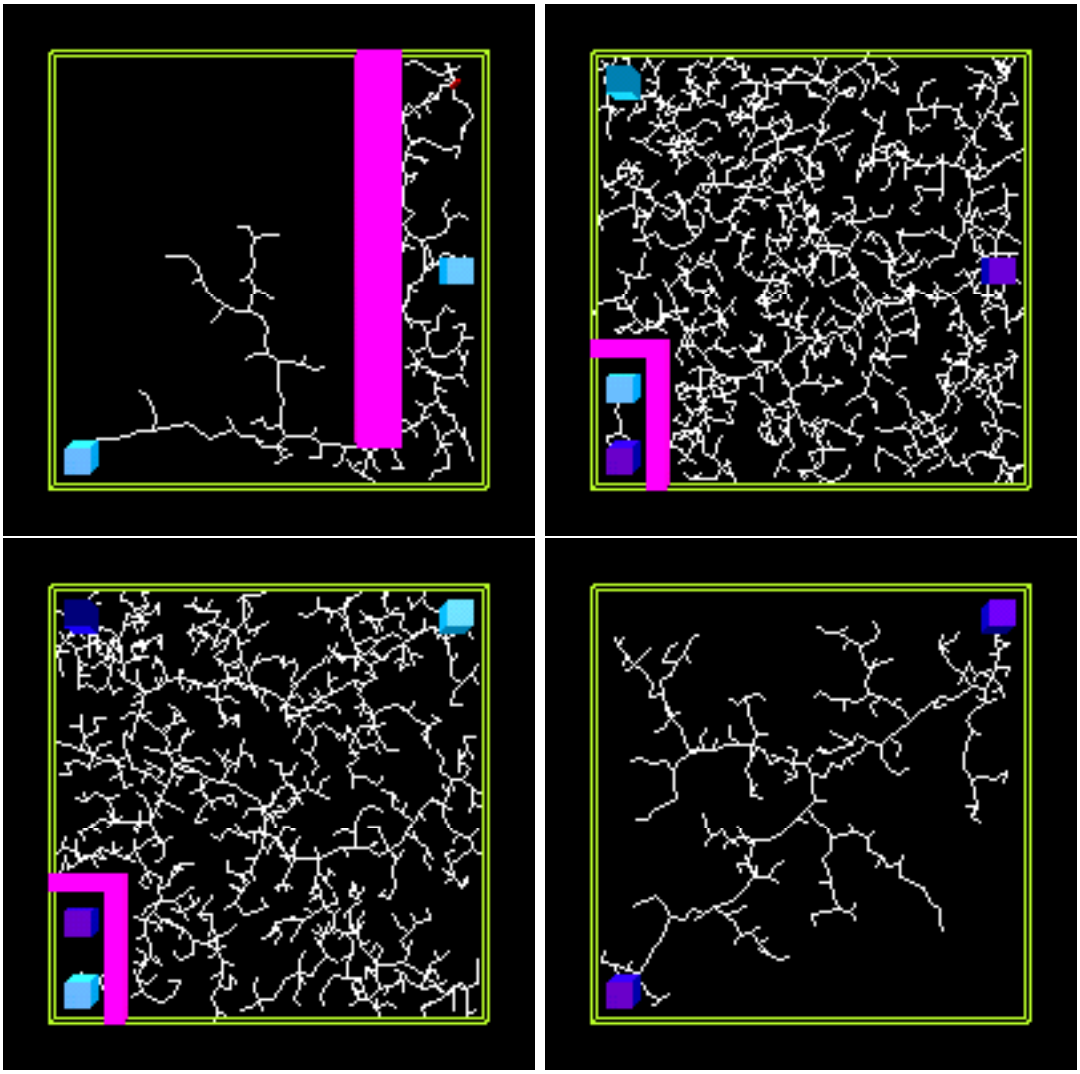

Figure 4.6: Four Floors of the Fourth 2d Stair Climber

a total of 13 subgoals in this layout; each of the 6 steps has a pair of subgoals as well as the true goal state in the center of floor 4.

We ran two sets of trials against these two layouts. The first was run with our original subgoal-biased planner (named SubGoal Bias), except we modified the stopping condition to be when every subgoal had been reached. The second (named TreeDist Bias) was a modified version of the SubGoal Bias planner so that the probability that subgoal i was picked, π_i, was related to the distance that subgoal was from the RRT itself. A list of minimum distances was maintained in the planner by calculating the

distance, d_i, of each newly added node to each of the subgoals that had not been met. If d_i was smaller than the previously stored value for subgoal i, we saved it. We then calculated the probability to pick a subgoal, k, by taking

$$\pi_k = \frac{e^{d_k}}{\sum_{i=1}^{n} e^{d_i}}$$

Thus, we weight the probabilities for each subgoal by their distance to the tree as compared to all of the other subgoals' distances to the tree. We exponentiate each of the distances so that the subgoals that are further away are even more likely to get picked.

For each of the two trials we ran ten iterations of the planner in question and took the average results for the total number of nodes planned. We varied the bias percentages over the set of $\{0, 0.05, 0.1, 0.25, 0.5\}$ (0.75 was removed because the results were similar to 0.5 in our first trials). Figures 4.7 and 4.8 show the resultant plots of the mean values for the total nodes planned in the third and fourth stair climber examples. Also shown are the standard deviations for each of these trials (notated by squares indicating the mean $\pm\frac{1}{2}$ of the standard deviation). We do not show plots of the path lengths for these two examples because when we were growing these two RRTs we were interested in reaching all of the subgoals, not finding a path to the goal state.

For the third stair climber example, Figure 4.7 indicates that the original planner (SubGoal Bias—shown in blue) was more efficient at reaching all of the goals than the one that took into account the tree distance of each subgoal (TreeDist Bias—shown in red). For all trials with any sort of bias, it took more nodes planned (and hence more time) for the second planner to finish. However, one should be careful to examine the standard deviation for this result. In most cases the standard deviation for the TreeDist Bias planner was significantly higher than the SubGoal bias. This indicates that the results were highly distributed, and since this is only the average

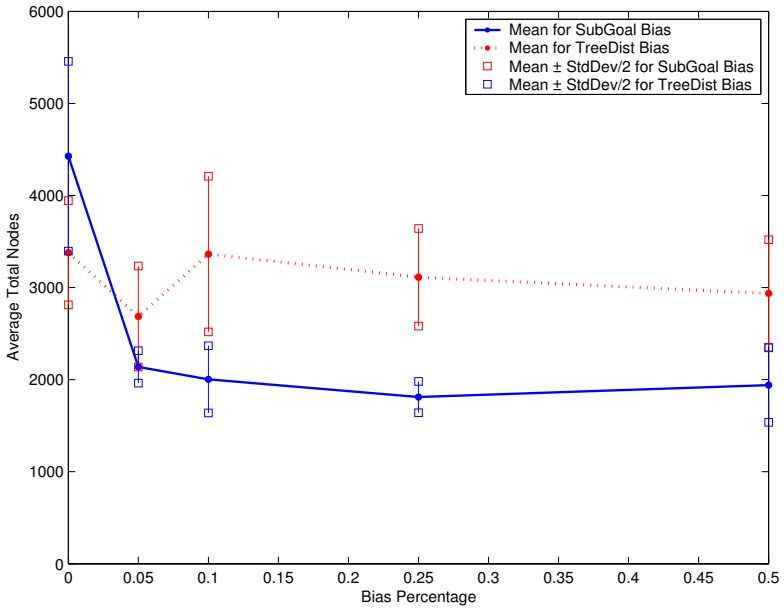

Figure 4.7: Plot of Total Nodes vs. Bias Percentage for Third Stair Climber

of ten trials, the TreeDist Bias averages might be overly skewed by only a few large values.

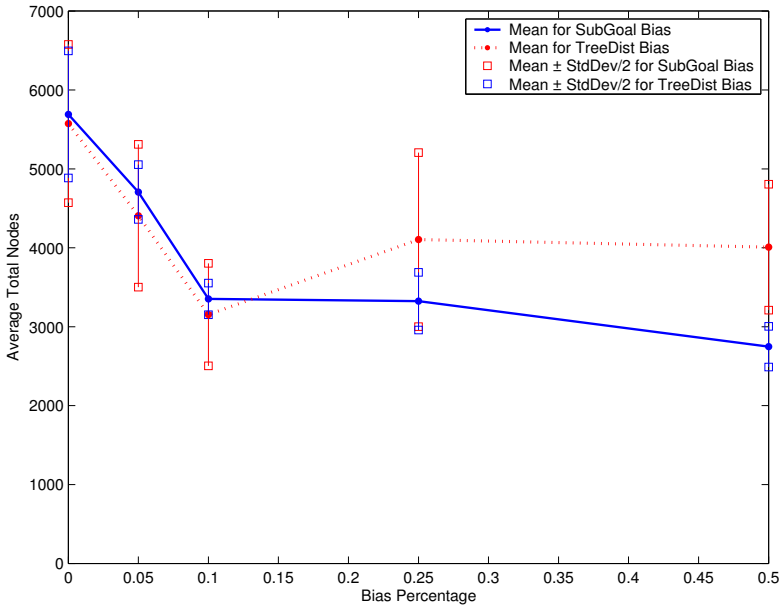

Figure 4.8: Plot of Total Nodes vs. Bias Percentage for Fourth Stair Climber

In the fourth stair climber, shown in Figure 4.8, we can see that the TreeDist

Bias was slightly better for low bias (≤ 0.1), but as soon as the bias was increased, it again was outperformed by the SubGoal Bias planner. Once again though, we also have higher standard deviations, indicative of a highly distributed result. For this experiment, the TreeDist Bias planner had the three lowest trials, with two at 752 and 859 nodes for a bias of 0.25, and one with 579 nodes at a bias of 0.5.

Consequently we have a highly mixed result for both experiments. The TreeDist Bias planner has the ability to outperform the SubGoal Bias planner in individual cases, but on the average the SubGoal Bias is an equal (and most of the time more efficient) way to determine paths to all subgoals. But note again that for each bias percentage we only ran ten trials, thus we might be experiencing issues based on a few poorly distributed samples.

One final interesting note is that we grew our tree until the tree reached each of the subgoals. However, we made no restrictions on what path it would take to reach the subgoals. Often, for the more difficult subgoals, the tree would go up to floor 4 and then work back down towards these subgoals. One potential method (that we did not pursue) to fix this is by specifying sequences of subgoals to accomplish, as opposed to an unordered list of subgoals.

4.1.3 Three Dimensions

We next chose to extend our work from two dimensions to a three dimension problem to take advantage of many more of the visual features of the MSL. Thus, we created a stair climber example where the configuration space was now $s = (x, y, z, q) \in [0, 50] \times [0, 50] \times [0, 10] \times \{1, 2, 3, 4\}$. Thus, the stair climber could move in the z direction within each floor as well. In terms of hybrid systems, this example presents little more significance than the two dimensional one above; however, in terms of application development having access to the third dimension is more powerful as well as more visually appealing.

We took our second two dimensional example (the one including obstacles) and translated it into the three dimensional state space. From a development standpoint; this involved writing additional features into the Render object to draw not in three dimensions, but in six (rotation about the each of the axes as well). In addition, the input specifications for the agent, Guard regions, and Obstacles were previously 2d polygons, but now had to be modified to be 3 dimensions. The designers of the MSL initially chose to use triangles to specify 3d objects, and we also decided to use a similar specification for our 3d objects.

Screen captures from this example are shown in Figures 4.9 and 4.10. For our agent, instead of using a point object, we chose to use a 3-dimensional stick figure. Instead of squares for the state transitions, we chose pyramid-like objects. Both of these models were designed using PTC Pro/ENGINEER and then output into Medusa .asc ASCII format. The RRT growth below took 99.9531 seconds on an AMD-K6 300 processor with 128 MB of RAM. 1415 nodes were grown with a step size of 3 and the path to root (shown in red) had 74 nodes.

The formula $\rho(s_1, s_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} + 200|q_1 - q_2|$ was used as our distance metric in this example. Again, we reached a similar problem with choosing random states, finding a balance between forcing the RRT to grow up while allowing it to continue to explore lower floors. The large factor of $k = 200$ causes only points with a $\Delta q = 0$ to always be selected first in the nearest neighbor computation. Thus, we experienced the same results as in the 2d case by picking random points with $q \in \{1, 2, 3, 4\}$ versus those with $q \in \{4\}$.

We also ran our biased RRT for the 3d examples. In Figure 4.11 we see two additional plots that were created the same way as the 2d version. We again ran two sets of six trials, using the same set of bias percentages ($\{0, 0.05, 0.1, 0.25, 0.5, 0.75\}$). In the first set we returned random states with $q \in \{1, 2, 3, 4\}$ and in the second $q \in \{4\}$. Again, each trial was run ten iterations and averaged.

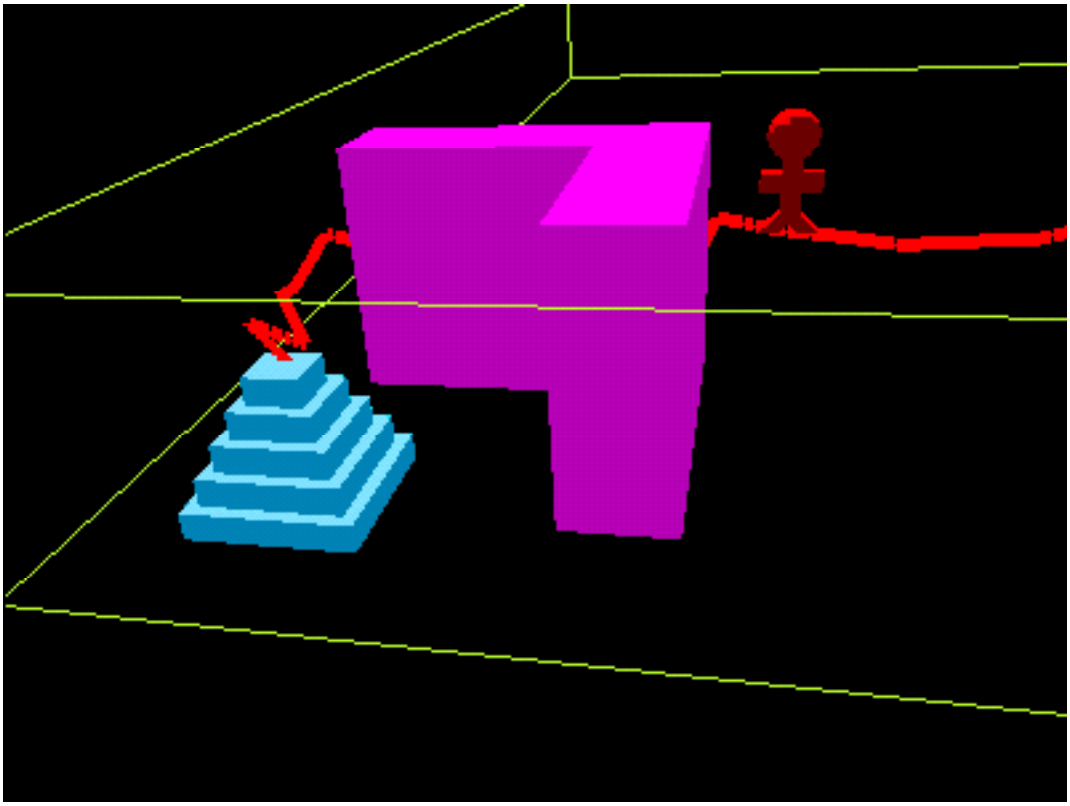

Figure 4.9: MSL example of a 3d Stair Climber, Floor 1

The results here are comparable to those in the 2d case. However, some features are important to note. First, since we selected the subgoals to have a z value of 0, the planner was pulled down toward $z = 0$ when it grew. Since the transitions here are tiny pyramids, which are wider at the base, the closer we stayed to the floor, the more likely we were to hit a guard region. Therefore, having these subgoals caused state transitions to be found more rapidly. In addition, our 3d agent is a tiny stick figure, instead of a point object, so the L-shaped obstacle on floor $q = 1$ was made with larger gaps than in the 2d case. Consequently, the shortcut path was taken more often. When we selected random states with $q \in \{1, 2, 3, 4\}$, the shortcut was taken four times when the bias was 0, and once when the bias was 0.05. When we selected random states with $q \in \{4\}$, the shortcut was taken once when the bias was 0. Therefore, the average path length was lowered for lower biases, causing the initial

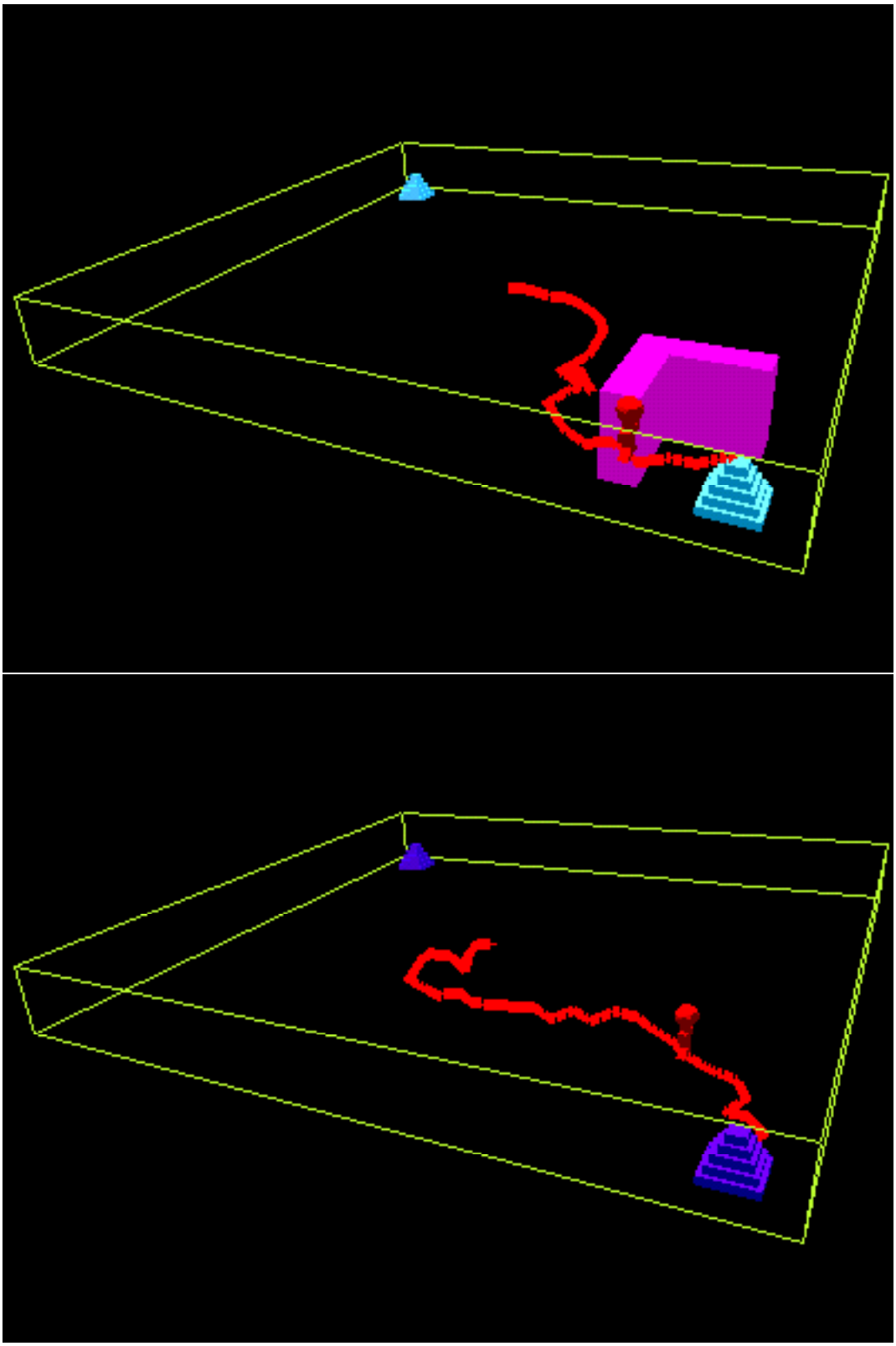

Figure 4.10: MSL Example of a 3d Stair Climber, Floors 1 and 4

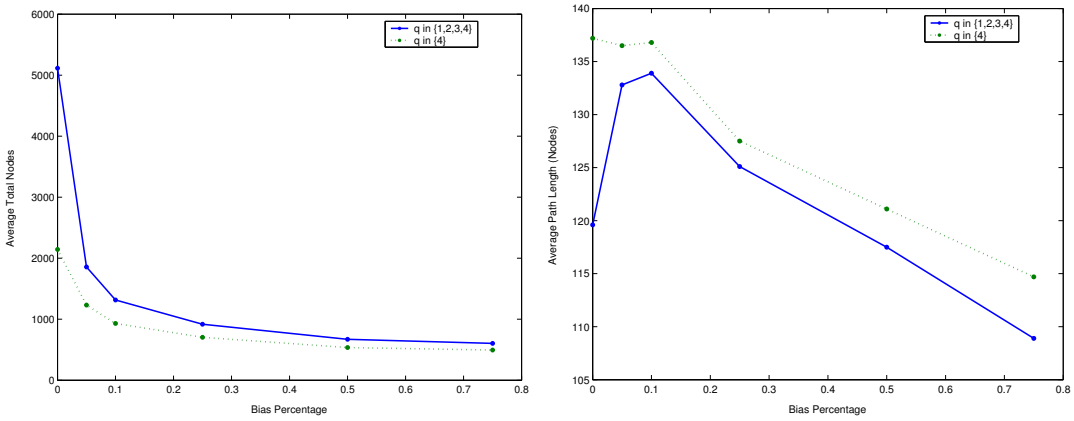

Figure 4.11: Plots of Total Nodes and Path Length vs. Bias Percentage for 3d

dips on the two lines.

4.1.4 Dynamics

The examples above demonstrate systems with constant, holonomic dynamics. That is, each step the RRT takes is governed by an unconstrained function of a fixed step size. Specifically, the RRT may move in any direction. However, our future work will be implementing examples that use more complicated differential inclusions. Specifically, hybrid dynamics can be included in the `Model` class for a given example problem. In the functions that determine the next state given a current state, a time step, and a control input, we include information regarding the explicit dynamics of the system. The differential inclusions of a hybrid system are applied given this input information to determine the future state, instead of just using the constant dynamics as shown above.

Our first steps involved systems with different speeds on each "floor" in the stair-climbing problem. To this end, we returned to the 2d example, but adjusted the way steps were taken. This involved designed a new `Model` object for the 2d example where the Select_Input() call within the Extend_RRT() function did a simple check for what floor the RRT was on. Based on this discrete state information, steps were

taken of different lengths. This is an example of nonhomogeneous dynamics, where the activity functions are now a function of q, the discrete space.

To provide additional clarity for the reader, we will recap where we stand based on both the work done by Curtiss [15, 21] and in the MSL. The stair climber examples all fall under the category of restricted, nonhomogeneous switching, since state changes occur at particular regions of each floor that are in different locations on floor-by-floor basis. With the exception of the example presented in this section, all of the stair climbers make use of holonomic, homogeneous dynamics as well since the dynamics allow the agent to move in any direction, at the same speed, and do not change on a floor-by-floor basis. Altering the Select_Input() method to return a different step length dependent on q creates nonhomogeneous (but still holonomic) dynamics. In the next sections, we explore additional systems with nonhomogeneous and nonholonomic examples.

4.2 Bouncing Balls

Motivated by the need to find more examples with non-constant dynamics we further pursued different hybrid systems examples. The next one we decided to test against the MSL was that of a bouncing ball motivated by Simić et al. in [55]. The basic idea here is to model a ball with a velocity that is affected by a constant acceleration. We model a bouncing ball as follows in the state diagram in Figure 4.12.

As the state diagram demonstrates, there is one single state for the bouncing ball with a constant acceleration due to gravity. Of more importance is the jump function. Here we choose to model the ball as having potentially all possible elasticities ($0 \leq c \leq 1$), so the jump function allows the velocity to be set assuming a fully elastic collision ($c = 1$), a fully inelastic collision ($c = 0$), or any possible value in-between.

It became apparent that some sort of jump reset functionality would be necessary

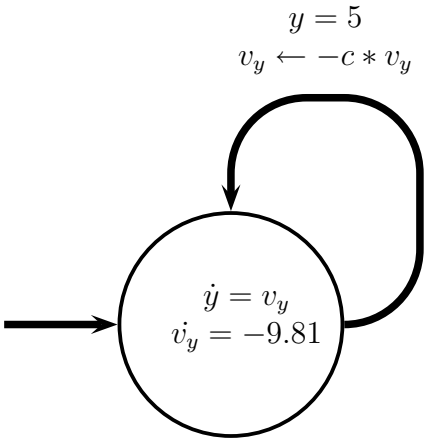

Figure 4.12: Bouncing Ball State Diagram

for our extended MSL to be able to work with bouncing balls. The version used for the stair climber examples above did not have a need for such functionality with particular edge resets, since following an edge only implied a discrete state configuration change. Here, we required modifying only the continuous configuration when we followed an edge, and not the discrete configuration at all. Consequently, in revised versions of the MSL, line 10 of Algorithm 3.2 (the call to `Problem.EdgeReset`$(x_{new}, x_{new2}, u_{new2})$ became an essential aspect of the RRT code. This function call directs the `Problem` object to query the `Model` object to determine the jump reset that will occur when following the edge from x to x_{new} and then returns the node to add as x_{new2}. Thus x_{new2} is inserted as a child of x following the input u_{new2}.

4.2.1 Ball with Staircase

With the bouncing ball example in mind, we devised a more visually appealing bouncing ball example modeling a ball bouncing down a set of stairs. This example was also motivated by [55], but is slightly different in that our model again only has one state. We essentially just add a constant horizontal velocity to the state diagram, and our guard condition is changed to incorporate the different step heights. Figure 4.13

shows a state diagram of the hybrid automaton we use to model this system.

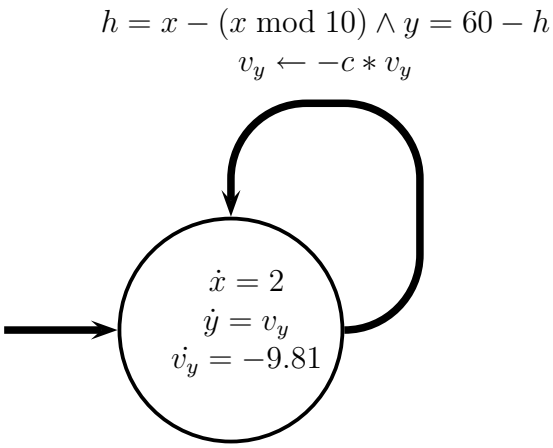

Figure 4.13: Bouncing Ball with Staircase State Diagram

This model incorporates a variable h to keep track of the x value of what step we are located above horizontally. The guard condition causes state transitions when the value of y is equal to the top of the step. This model assumes a set of six steps, each of width and height 10. Hence the term $60 - h$ maintains the top of the step, since the first step is of height 60 between x values of $[0, 10]$, the second step is height 50 located at x values of $[10, 20]$, and the steps continue downward as x increases.

For this example we model the ball as a sphere of radius 2.5. We created a set of guard blocks that are 10 units wide and 60 units deep. In addition, we made these guards 4 units high, to accommodate the step size and approximate ball collisions. Pictured in Figure 4.14 is the result of planning against this particular example. This particular path was grown in 4.36719 seconds on a Pentium 4, 2.4 GHz machine with 512 MB of RAM. The tree has 539 nodes and was grown with a Δt of 0.1. Here since our hybrid automaton only has one state, we again chose to use the Euclidean metric.

In this particular example, we grow to when the ball has reached a continuous configuration where its z value is less than zero. The continuous configuration for this

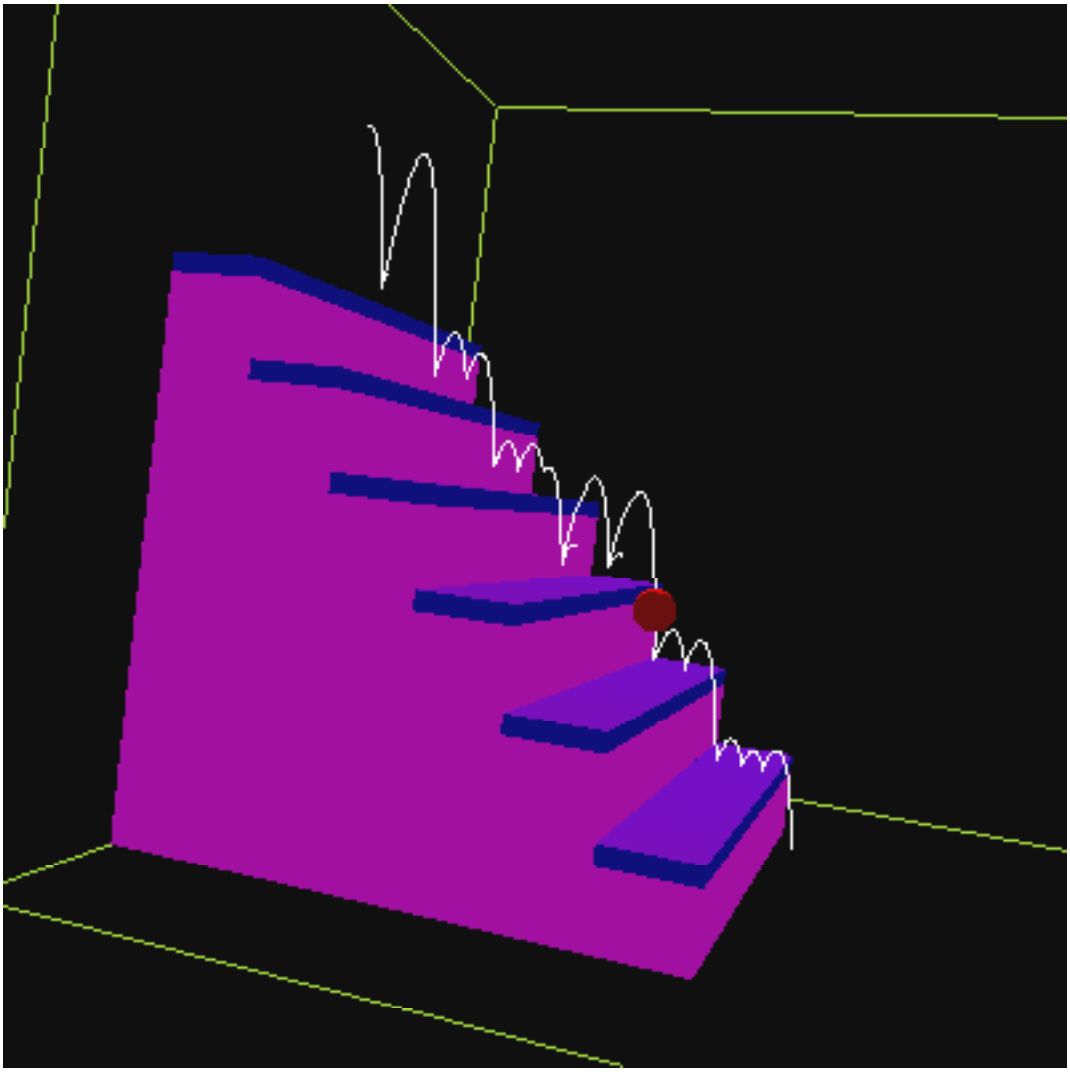

Figure 4.14: MSL Example of a Ball with Staircase

example is nine dimensional, including the x, y, and z components, rotations on the three axes, and velocities in the x, y, and z directions. Here we use z as the height instead of y in our state diagram, but the change of variable has no significant impact on the results.

We are interested here in discovering not a single trajectory but catering the example to observe multiple trajectories. The difficulty here is picking random points so that the point of state transition occurs multiple times; causing splits in the path when the ball bounces off of the step. In this sense, we need to revise the metric

function and the random sampling of points so that these points are more likely to be picked and grown from. The example shown in Figure 4.14 is typical; most samples grown did not have many branches that were taken more than a step or two deep (note the limited branching visible on the fourth step).

4.3 Rectangular Hybrid Automata (RHA)

We chose to next test our extension of the MSL against a rectangular hybrid automaton (RHA) similar to those discussed in [3, 35, 51] as well as above in Definition 2.1.4. To review briefly, rectangular hybrid automata model those systems where the behaviors bound the rate of change of each of the continuous variables. Hence, the activity functions are differential inclusions of the form $\dot{x} \in [L, U]$, $L, U \in \mathbb{N}$ for each of the continuous variables. They are often used to approximate more complicated system dynamics, and they are equivalent in power to linear hybrid automata [29]. Our tool is not designed specifically for them; however, by studying RHAs, we show our tool is capable of modeling many of the examples currently researched. In addition, they are important systems because they present another nonholonomic example that has more complexity than the bouncing ball.

To control state transitions, we make use of the collision detection algorithms similar to way we did with the stair climber example. For each guarding condition, we construct a polygon region to represent the guard. Our RRT_Extend() algorithm is modified so that before the new state is added, a check is done to see if the new state will collide with this region. If so, we perform a reset using the EdgeReset() function, and add this reset state instead of the original new state calculated previously.

An interesting feature necessary to implement here was the concept of a multi-state view. As opposed to the stair climber example, in each state, the values that the continuous variables take on are disjoint regions of the same continuous space.

Consequently, while viewing each state individually, the need to view multiple states simultaneously increased dramatically. To this end, our Render object was modified to be able to draw all states at the same time, on top of each other. We do this by simply drawing the tree and path as colored line segments instead of white ones as in the images above. Each segment is colored differently depending on which state it is in. We also disable drawing the obstacles and state transition regions, as their locations will be apparent by the color changes in the tree. To enable or disable this feature, the user toggles the **AllState View/OneState View** switch located at the bottom of the center column in Figure 3.5.

4.3.1 Sample RHA

Given this, we propose to study the hybrid automata diagramed in Figure 4.15. We choose to take this model instead of one directly from the literature because we wanted to devise an example system that included sufficient branching and jump resets to be interesting as a case study.

This model has two continuous variables x and y as well as the discrete variable $q \in \{0, 1, 2, 3, 4\}$. Hence the hybrid state here is (x, y, q). To grow RRTs in this context we require redefining the notion of creating a new state or "taking a forward step" in this system. As we pick random points in the space and find their nearest neighbor, r, we construct a new state by determining random values for $\dot{x}$ and $\dot{y}$ within the bounds based on the differential equations for state r. The nearest neighbor is selected by using the Euclidean metric as a distance measure. We extend forward based on these values as well as the value for Δt, the size of the step to be taken. Hence, we construct a new state incrementally by $x_{new} = x_r + \dot{x} * \Delta t$ and $y_{new} = y_r + \dot{y} * \Delta t$.

The results that we achieved using this strategy are positive. Figure 4.16 shows an example of a run on the above system. The region plotted extends in the lower left from $(x, y) = (-10, 0)$ to $(x, y) = (20, 20)$ in the upper right.

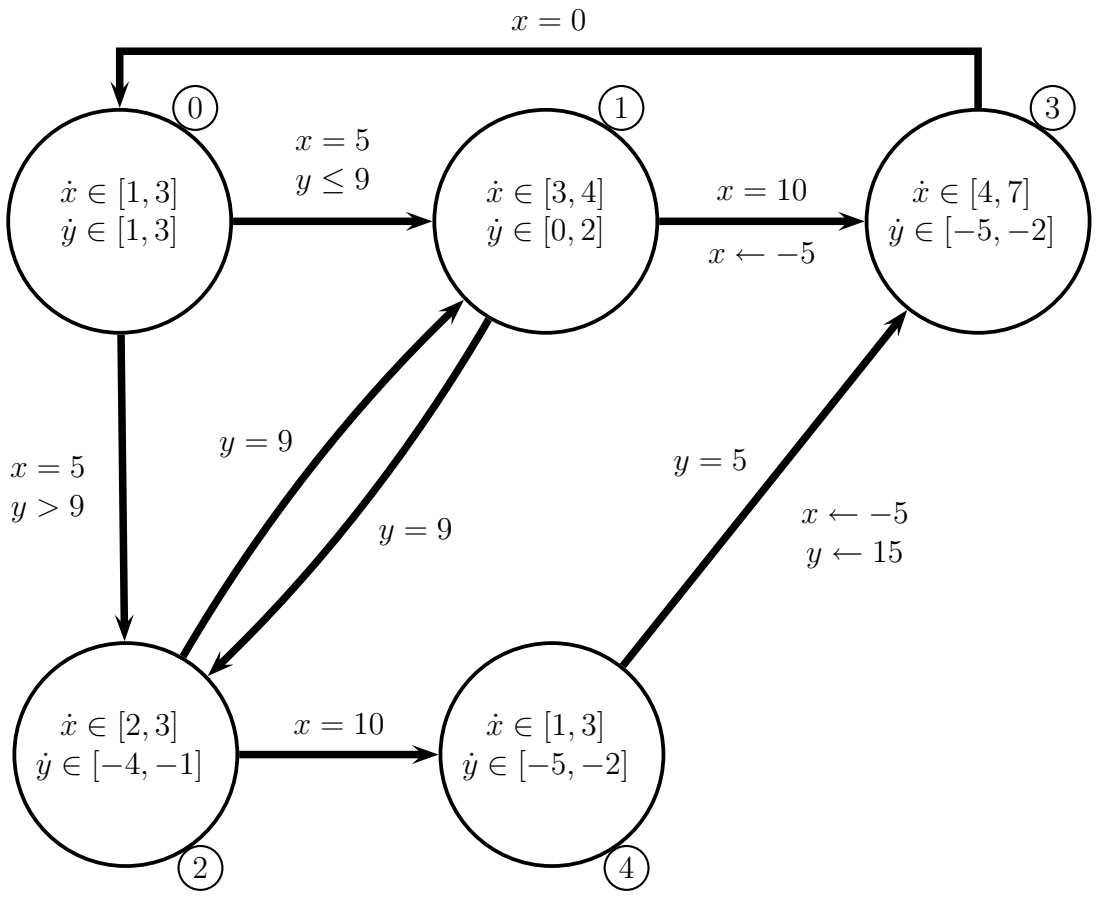

Figure 4.15: A Rectangular Hybrid Automata

For this particular plot, the planner Δt is 0.3, and we have constructed 1795 nodes in our RRT. Planning took 26.45 seconds on a Pentium II 366 with 128 MB of RAM. We plan using an initial configuration of $(x_{init}, y_{init}, q_{init}) = (1, 1, 0)$ and a goal configuration of $(x_{goal}, y_{goal}, q_{goal}) = (2.5, 15, 0)$. We use the goal configuration to represent a stopping point for the growth algorithm; that is, we are interesting in determining if the goal state is reachable from the initial state. This is similar in context to a motion planning problem, the planner will stop if the RRT gets to the goal state or if the RRT algorithm grows a fixed number of iterations.

Figure 4.16 represents the growth of the RRT throughout the hybrid system. Each color is indicative of a different state the RRT is in, specifically they translate to 0

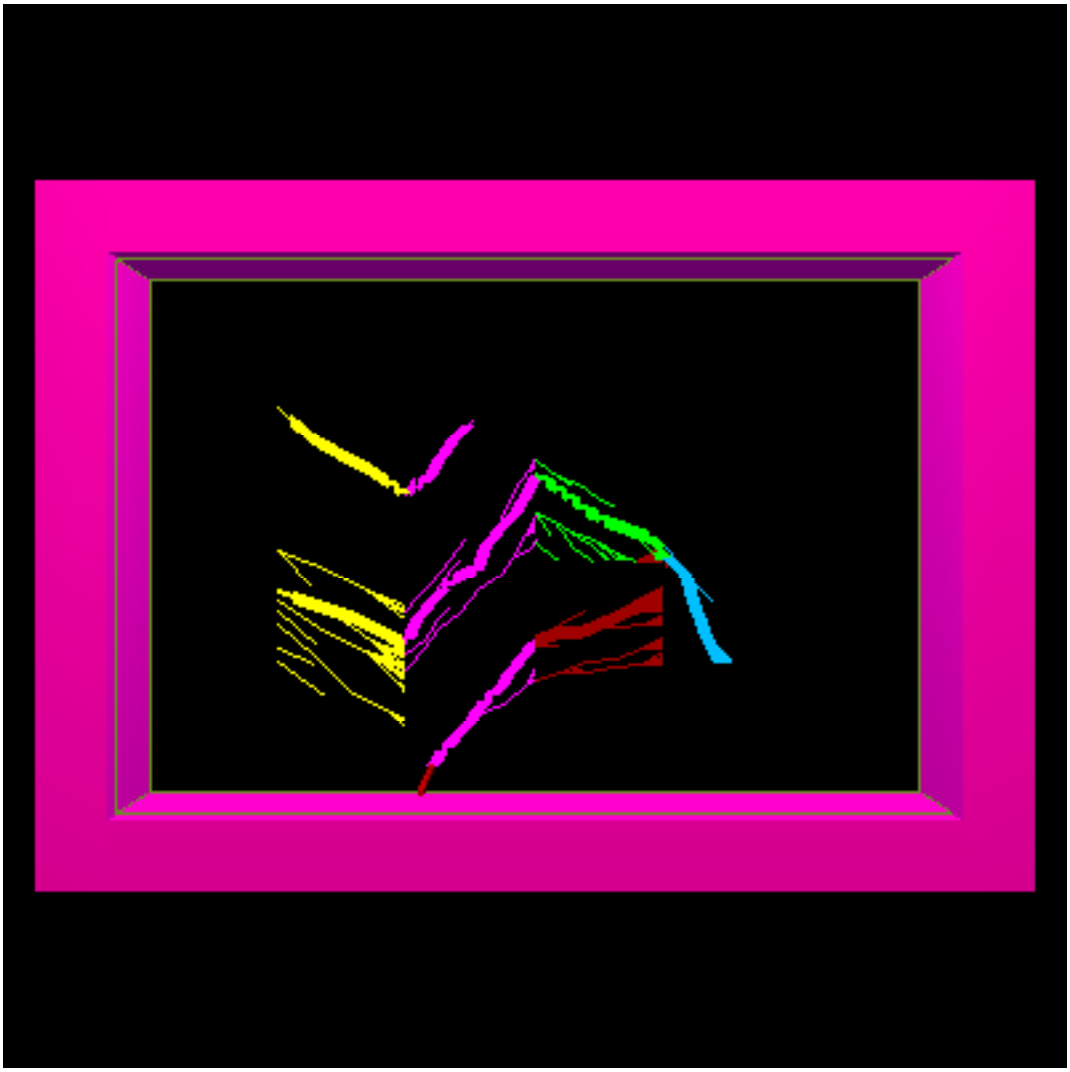

Figure 4.16: MSL Output for the RHA in 4.15

= pink, 1 = red, 2 = green, 3 = yellow, and 4 = blue. From this picture we gain a sense of both the vector fields in each state as well as the regions that are reachable within this hybrid system. Another way of representing this can be depicted in the graph shown in Figure 4.17. Note here that arrow lengths are *not* proportional to the actual rates in each state (which may vary anyway), and this arrows are just shown to represent general direction. Grid size in this figure is in increments of 10.

Is it important to note in both pictures where the jump resets occur. In state 1 (red), when $x = 10$ a state transition occurs that shifts $x = -5$, corresponding to a

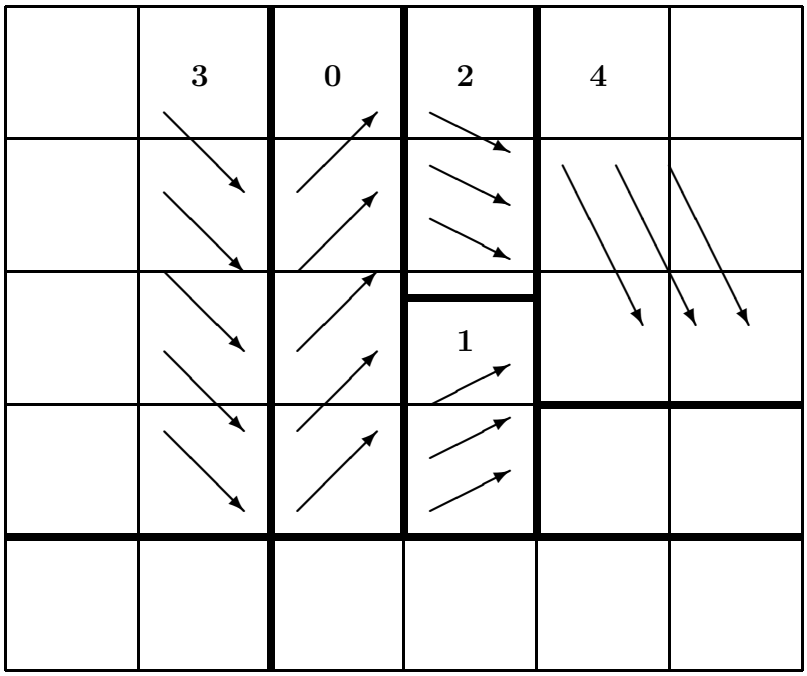

Figure 4.17: A Plot of the RHA in 4.15

change into state 3 (yellow). Similarly, when the state is 4 (blue) and $y = 5$, a state transition occurs again into state 3 (yellow) and the reset $x = -5, y = 15$ occurs. Both of these account for the discontinuities in the graph pictures shown, and given this the MSL is performing as expected.

To accommodate these discontinuities, we also added the option in the MSL to show the state transitions where they occur. It may be rather complicated to view all of the jump resets when showing the entire tree; however, it becomes more clear when viewing the path planned as shown in Figure 4.18. As you can see in this image, the path is colored as shown, but transitions are colored using white lines. They allow the reset conditions to stand out and conveniently show where disjoint parts of the path connect. Referring back to our extended control panel in Figure 3.5, we can enable or disable viewing transitions using the **ShowTrans On/ShowTrans Off** toggle switch located at the bottom of the second column from the left.

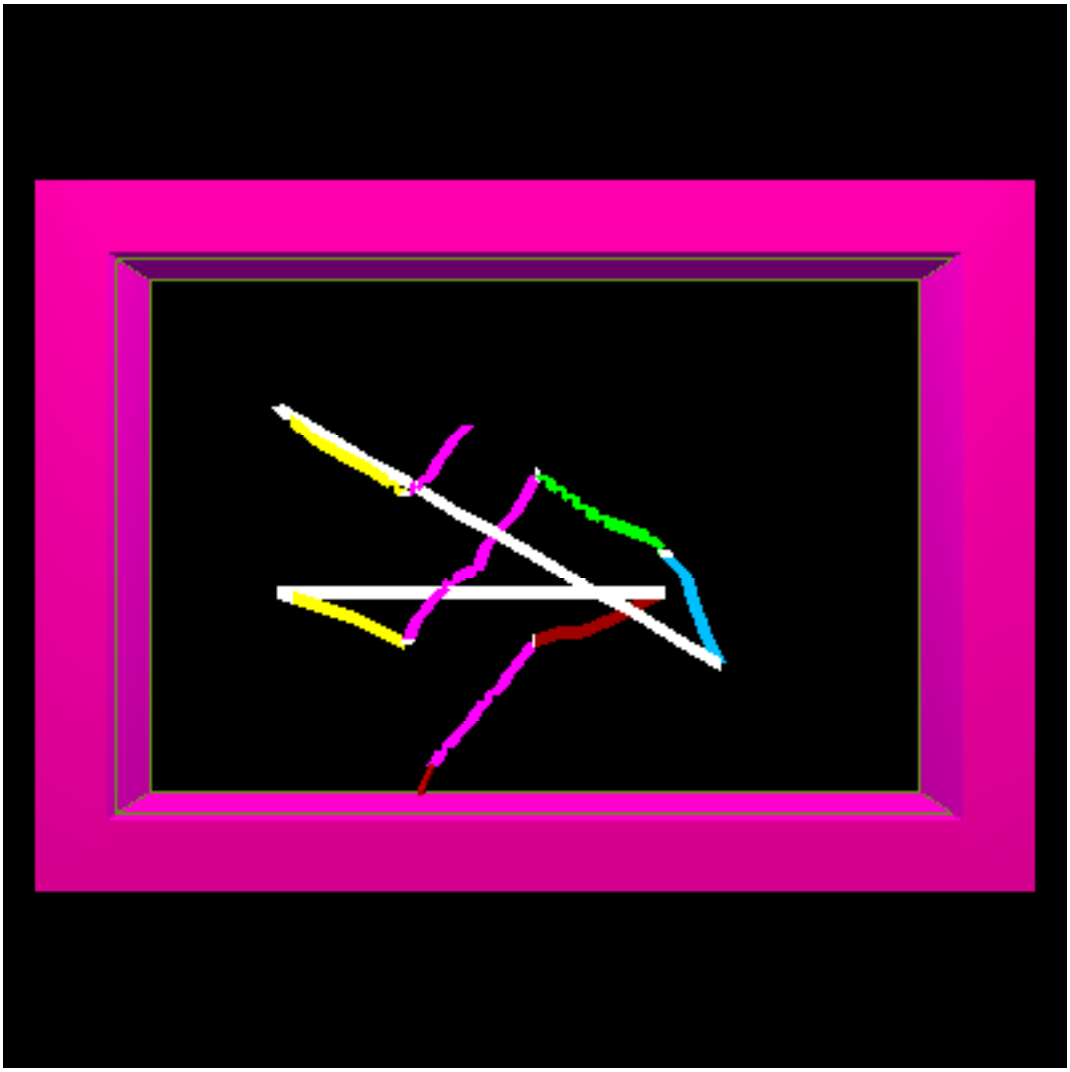

Figure 4.18: Output for the RHA in 4.15 with Transitions

The results from above demonstrate that using our extended MSL we can gain a successful visual representation of the vector fields as well as the reachable regions for an arbitrary rectangular hybrid automaton.

4.3.2 Multi-Action Growth

The traditional approach to deciding rectangular hybrid automata is to double the dimensionality by introducing a new variable for each in the continuous space such that one variable changes with the upper bound and the second changes with the lower

bound. This effectively reduces the hybrid automaton to a multi-rate automaton. For more information on this approach, the author suggests reviewing [35, 51].

Continuing with this theme, we have modified our RRT `Planner` and our `Model` objects to grow multiple points for each RRT_Extend() call. In this sense, we grow a unit instead of just one node, where each unit represents the upper and lower bound reachable from our nearest neighbor. This involved modifying both the Select_Input() function as well as the Extend_RRT() function. Select_Input() previously returned one new node and one input to get to that node, instead we have it return a set of nodes as well as a set of inputs. RRT_Extend() is then modified to insert each of these nodes into the tree as well shown in Algorithm 4.1.

Algorithm 4.1 (Extend_RRT (Revised for Multi-Action Growth)). *The following algorithm extends a tree, T, towards x by taking a fixed step from the closest node in T to x towards x.*

```
1  HybridRRT.Extend_RRT(T, x) {
2     //find node in T nearest to x
3     x_best ← Select_Node(x, T);
4     //construct a new node set, X_new, input set U_best
5     U_best ← Select_Inputs(x, x_best, X_new);
6
7     //insert all new nodes
8     FOR ALL x_new ∈ X_new {
9        //is there a state transition?
10       IF Problem.StateTransFree(x_new, x_new2) = FALSE {
11          //node growth causes a state transition
12          //perform a reset, create x_new2 and u_best2
13          Problem.EdgeReset(x_new, x_new2, u_best2);
14          //now ok to add, insert
15          T.add_vertex(x_new2);
16          T.add_edge(x_best, x_new2, u_best2);
17       } ELSE {
18          //no transition, just insert
19          T.add_vertex(x_new);
20          T.add_edge(x_best, x_new, u_best);
21       }
22    }
```

```
23 }
```

Outside of being close in theory to how to approach rectangular hybrid automata, this idea is close to the nonholonomic cases studied by LaValle et al. with their MSL. The Select_Input() function in the nonholonomic case asks the Model for a list of potential inputs instead of doing an integration along the line from x_{best} to x (x being the random state chosen). Select_Input() next iterates through and finds the best one to use and constructs x_{new} accordingly. Instead, our algorithm just returns all possible inputs and then grows from all of them.

We show a resultant tree grown using the concept of multi-action growth in Figure 4.19. This example had an RRT grown with a Δt of 0.6 in 819 nodes. The path was planned in 0.98 seconds on a Pentium II 366 with 128 MB of RAM. One can see from this figure that for each node the tree will grow out four new nodes representing the unit for this rectangular system. Given the discrete state, we know that $\dot{x}$ and $\dot{y}$ are bounded such that $\dot{x} \in [L_x, U_x]$ and similarly $\dot{y} \in [L_y, U_y]$. Hence we can create four inputs for $(\dot{x}, \dot{y})$ from the set of $(L_x, L_y), (L_x, U_y), (U_x, L_y), (U_x, U_y)$.

One potential problem that we noted with this system is that by using the modified Euclidean metric for extending our tree, we often caused parts of the tree to be left without extension. This is apparent in Figure 4.19, as many of the branches grown in states 0 and 2 are left childless, and only a few are selected to continue growing from. In some cases, this is a result of the fact that we add four nodes for each one we select, leaving more leaves. However, by selecting nodes more judiciously, we might be able improve this as well and create more uniform expansion.

Despite this potential problem, the results have been positive as well, providing significant improvements over the original, single-action growth versions of the same system. Often (not always, since the algorithm is still probabilistic), the planner will complete paths more quickly. Furthermore, we were able to identify two possible trajectories to the goal state that were not as likely to be found by the traditional

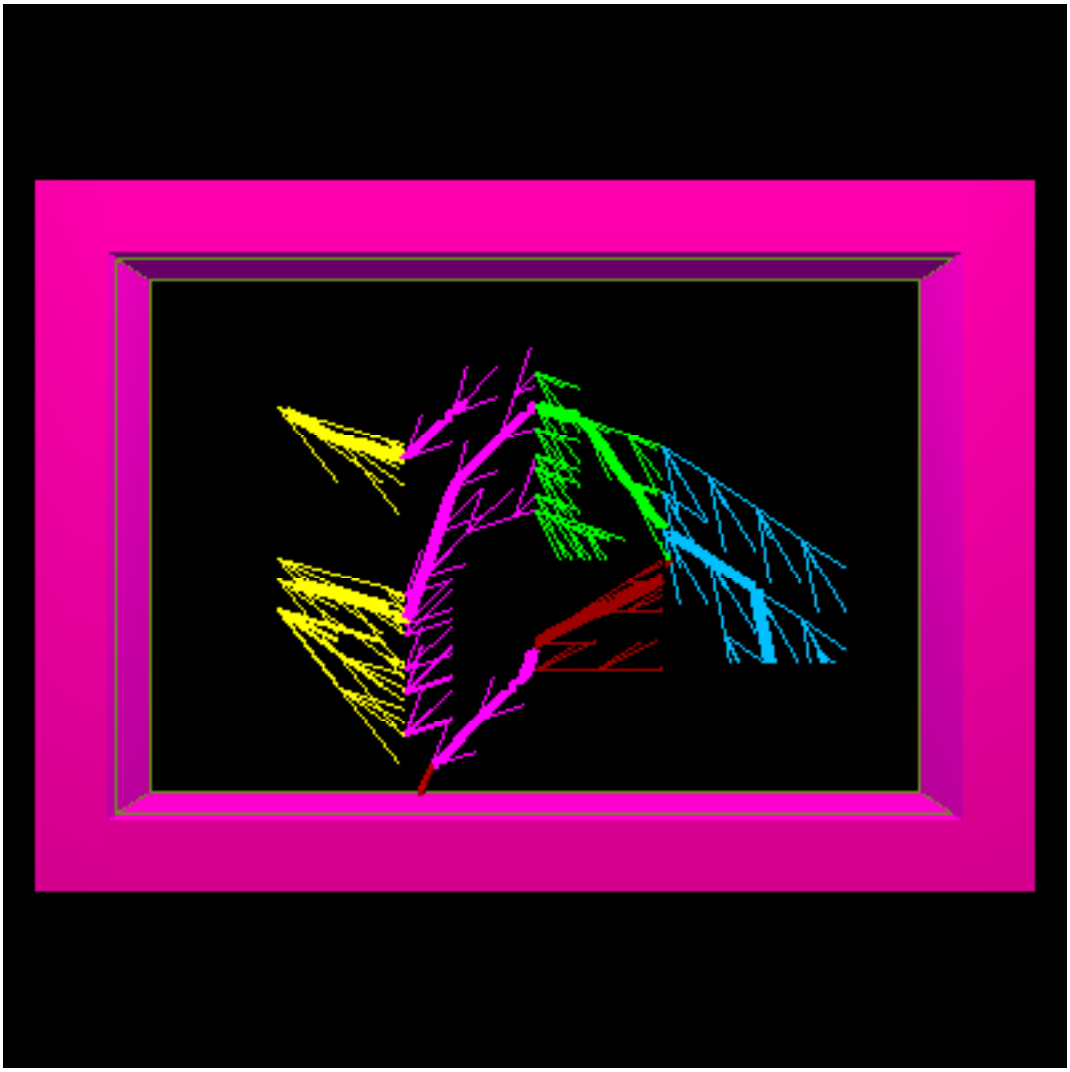

Figure 4.19: RHA with Multi-Action growth

algorithm. In Figure 4.20 we show two such instances.

The first demonstrates being able to reach the goal state without planning into discrete states 2 and 4. This example was grown using a Δt of 0.3 and had 1742 nodes in the RRT. It took 3.57 seconds to plan the path on a Pentium II 366 with 128 MB of RAM. The second example shows the planner finding a trajectory to the goal state without planning into state 1. In this example the Δt has been increased to 1.0 and the RRT had 288 nodes. Planning time was 0.13 seconds on a Pentium II 366 with 128 MB of RAM.

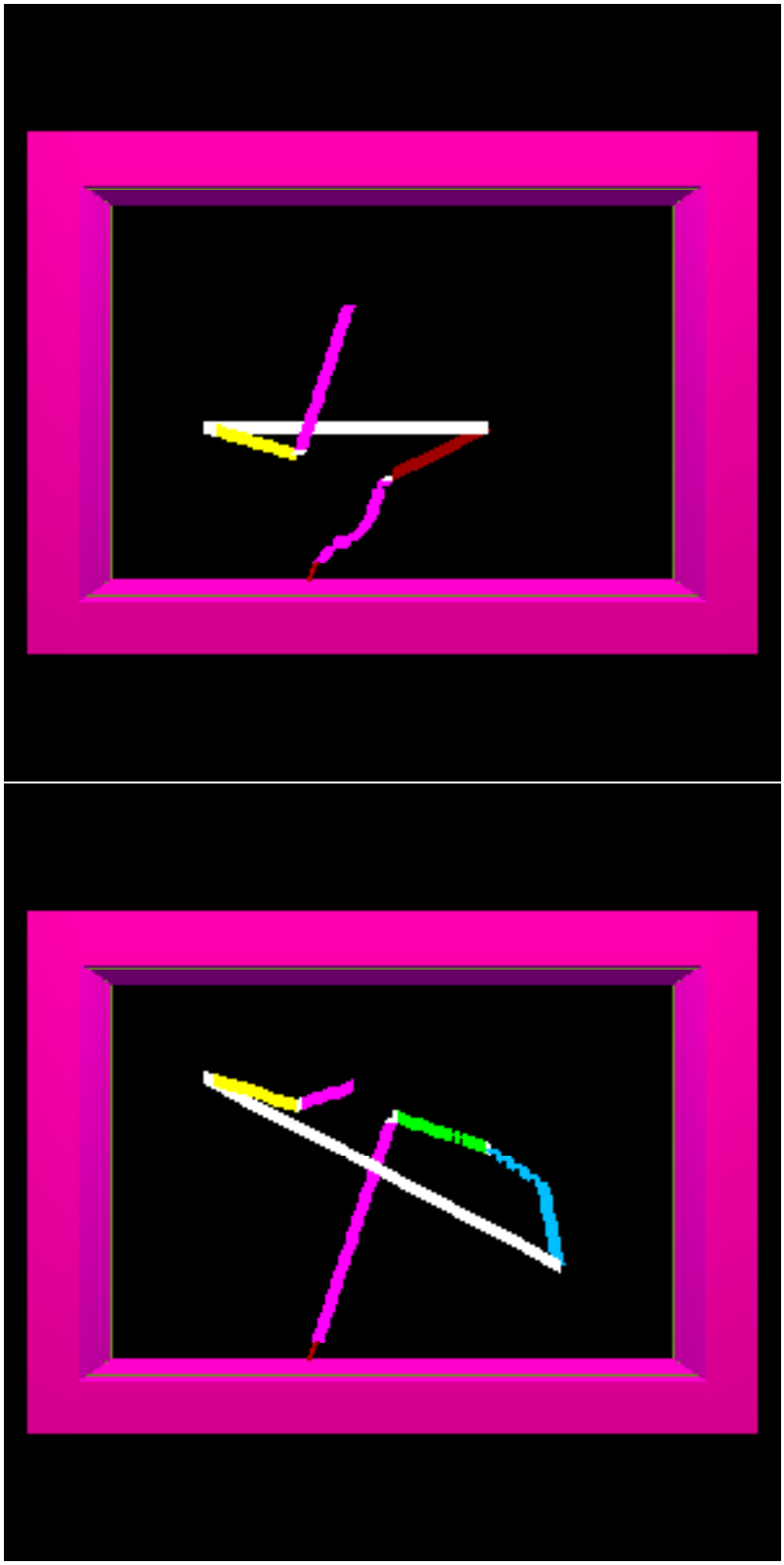

Figure 4.20: Multi-Action Growth without States 2 and 4 (top) and without State 1 (bottom)

Of course, the fact that these paths exist is simply a function of the specifications of the system itself. However, the system was devised so that all states would be likely to be traveled to reach the goal state. Hence, the existence of both of these simpler ones came as a pleasant surprise that demonstrates the success of this technique.

Since many hybrid systems may be approximated as rectangular hybrid systems, demonstrating that our extended MSL functions on rectangular hybrid systems is an important step to performing verification. Coupled with the concept of multi-action growth, which is simply an extension of the techniques used to study rectangular systems we have increased the scope of our extension considerably.

Chapter 5

Investigating and Improving RRTs

In addition to the study of the applications of RRTs to hybrid systems, we conducted a significant number of experiments to further the research of the RRT algorithm by studying its properties and making improvements. This section presents the results of that research as divided into two major categories of investigating the RRT and improving the RRT. We investigated the RRT by trying to gain a sense of how completely it covers the space and how quickly this coverage occurs. Similarly, an experiment was proposed to study how close to the optimal path the path the RRT grows is. Finally, we propose improvements to the RRT by increasing the efficiency of the nearest neighbor calculations through the use of a vantage-point data structure.

5.1 Contour Maps

Our first experiments present a means for visualizing the RRT and the reachable region exactly. To this end we created a "contour map" of the RRT indicating the area reachable in the next steps of the RRT. The construction is based on drawing discs at each node that are of radius equal to a multiple of the step size for the RRT. We color the circles differently (with a gradient of colors) to indicate the range out from each node of the tree. The resulting images provide a significant visualization

of both how far out the RRT can reach in a fixed number of steps and how many steps are required to reach a particular region. Figures 5.1 and 5.2 show two such examples.

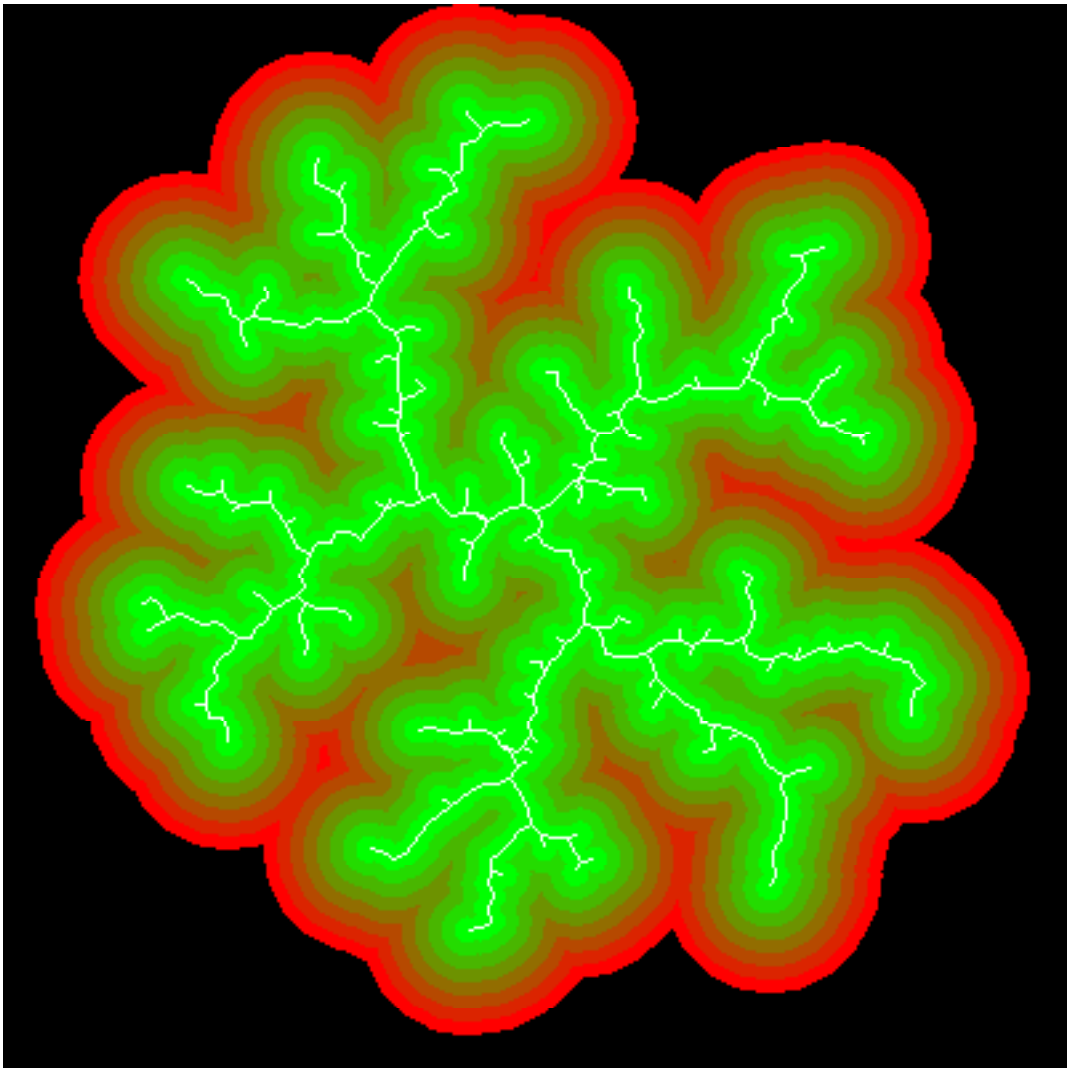

Figure 5.1: A Contour Map of the RRT

Figure 5.1 has the RRT growing out with eight different layers indicating the region reachable by the RRT in one through eight steps. This image was created by growing 450 nodes of an RRT on a disc of radius 200. The Δt for the RRT was 5. Figure 5.2 shows a similar picture with a progression of the contour map of an RRT growth. Here the RRT is grown on a disc of radius 50 with a Δt of 3. Only three

layers of the contour are drawn, and the images from left to right contain 10, 50, 100, and 300 nodes, respectively.

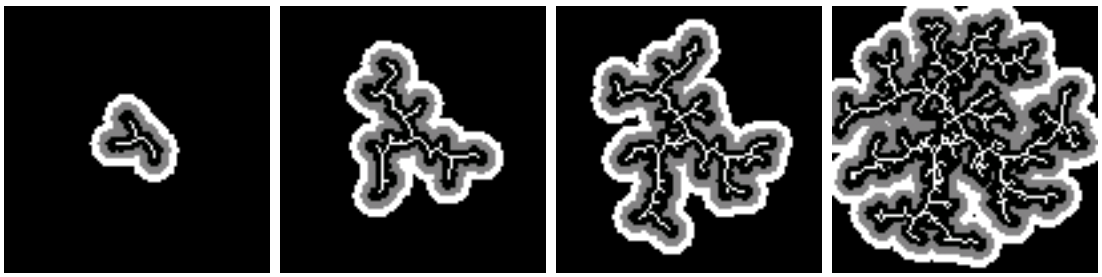

Figure 5.2: Contour Map of RRT Growth

In addition to drawing this contour in our own private examples, we opted to add this feature into our extension of the MSL. In two dimensions, the drawing is done exactly the same, but the user is allowed to select how many layers are drawn outward. For three dimensions our visualization is done using spheres drawn as point clouds with a randomly picked number of slices so that the viewer can differentiate them. This effect produces a fog around the RRT that achieves the desired purpose of visualizing where the RRT can reach next. Figure 5.3 depicts this three dimensional example done in our stair climber example ($\Delta t = 1$).

In terms of added controls to the MSL, referring back to Figure 3.5, the reader will now take note of the two controls in the bottom right of the window. The first is a toggle switch (the **Contour On/Contour Off** switch) to turn the contour map on or off. The second control (the cycling numeric control directly below) allows the user to cycle through contours of depth 1 up to 5.

Unfortunately, while an interesting visualization, we chose not to pursue calculating the exact area of the contour map of the RRT. This visualization shows exactly where the RRT can reach in n steps; however calculating the area of these regions would require the complex calculation of the intersecting area of all of the circles. Our focus in this experiment is a simple, accurate visualization.

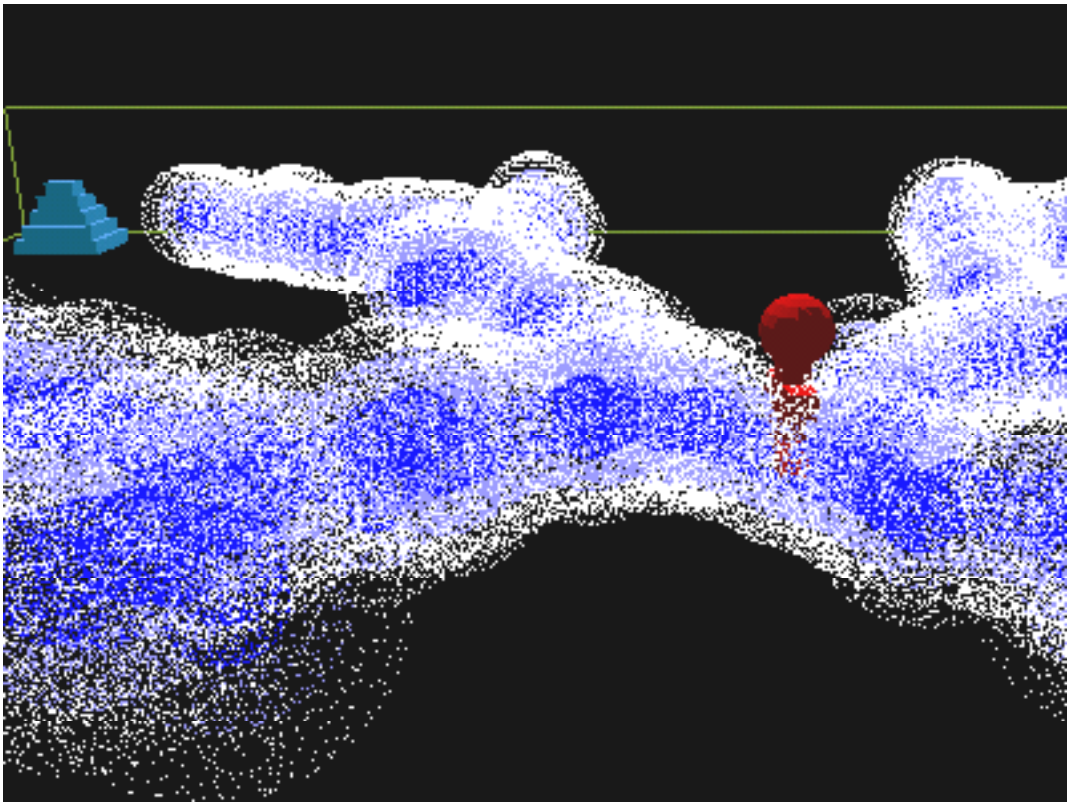

Figure 5.3: Stair Climber with 3d Contour

5.2 Convex Hulls

Motivated by the visualization work described above, we were interested in how to efficiently calculate the area reached by the RRT. To this end we began studying the convex hull of the points in the tree itself. However, this region incorrectly approximates the reachable area because of the large gaps of space between some points in the tree. Particularly in the case of obstacles this can provide a poor approximation. In Figure 5.4 we show the hull of an RRT that is grown in a disc of radius 200. 1000 nodes were grown using a Δt of 5. This hull was calculated using a variation of the Graham Scan algorithm [28]. It is apparent from this image that the area enclosed by the convex hull is not the area the RRT has reached.

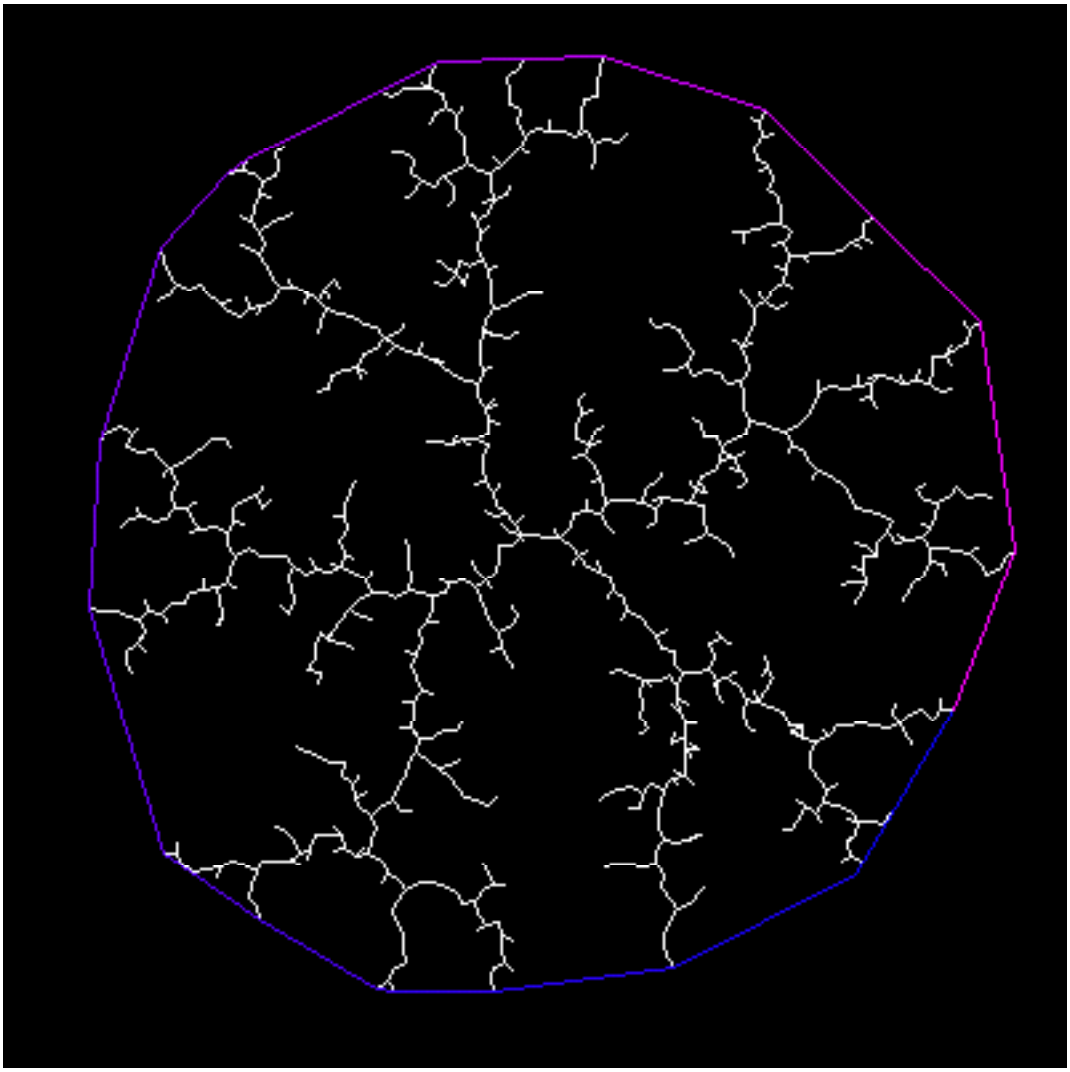

Figure 5.4: Convex Hull of an Entire RRT

5.2.1 Improving Our Hull

In attempt to better approximate the area, we needed a hull that was not convex, since the region explored by the RRT is itself not convex. Similar to the minimal-area hulls computed in [4] we wanted a tool to visualize the RRTs hull in a non-convex manner.

The first idea was to group the major divisions of the RRT and compute their hulls. In [44], it is noted that on a large disc the number of major branches of the RRT is $n + 1$, where n is the dimension of the space. Thus we aimed to group these

branches by calculating the hulls of the three of them. We do this by inserting the nodes based on which grandchild of the root they are under. The assumption here is that the grandchildren of the root have already been divided into their three branches. Generally, this assumption is correct; however, in a few cases the divisions could occur further down the tree and cause this to have fewer hulls than the number of major branches. Figure 5.5 shows the growth of this convex hull. Here we show our hull grown on a disc of radius 100 with a Δt of 5. The two images show the hull at 50 and 750 nodes. We can see clearly the three major branches; however, we can also see two flaws with this method. The hulls still are not an effective approximation: they contain large areas of empty space not reached by the RRT. Also, the hulls begin to intersect causing errors in the area calculated.

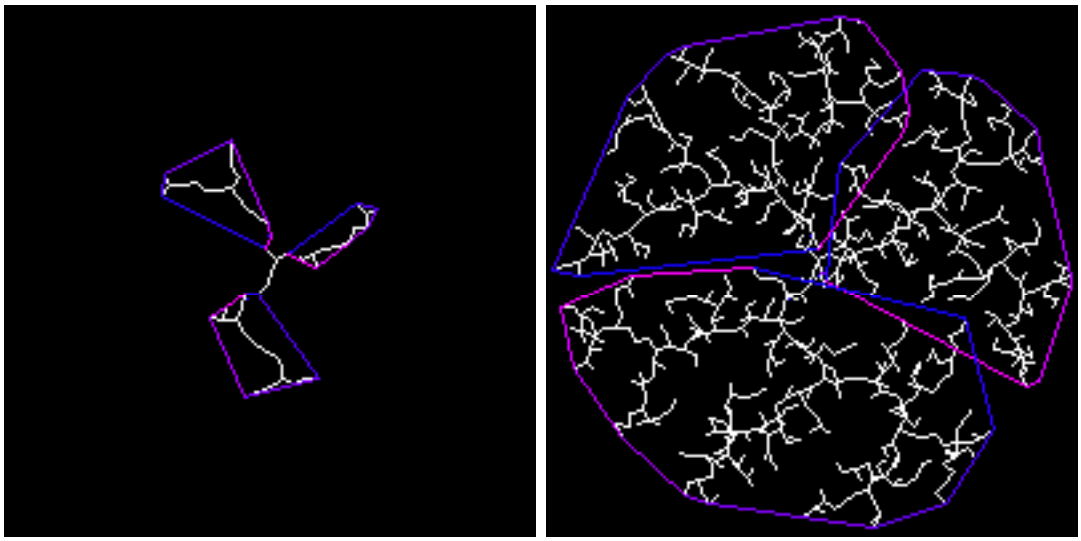

Figure 5.5: Branched Convex Hull at 50 and 750 Nodes

Finally, the last method we used to calculate the convex hull of an RRT was done in a manner where we could calculate the regions to arbitrary precision. We group the points based on their parent and depth in the tree. Basically, we fix a constant value D as our division of the depth of the nodes. We create multiple convex hulls such that their parent has a depth, d, of $d \equiv 0 \pmod{D}$. This creates a set of regions

that approximate the region that the RRT has reached to a precision ranging from the exact RRT ($D = 1$) to the height of the RRT (this would return the convex hull of all of the points of the RRT).

The results of this calculation show a powerful way of expressing the area explored. Figures 5.6 and 5.7 show two examples of the growth of the hull. Figure 5.6 has been grown to 210 nodes on a disc of radius 200 with a Δt of 10. The depth is divided by 2 and is drawn hollow without the tree itself. The low value for the depth division causes a very tight approximation of the tree itself.

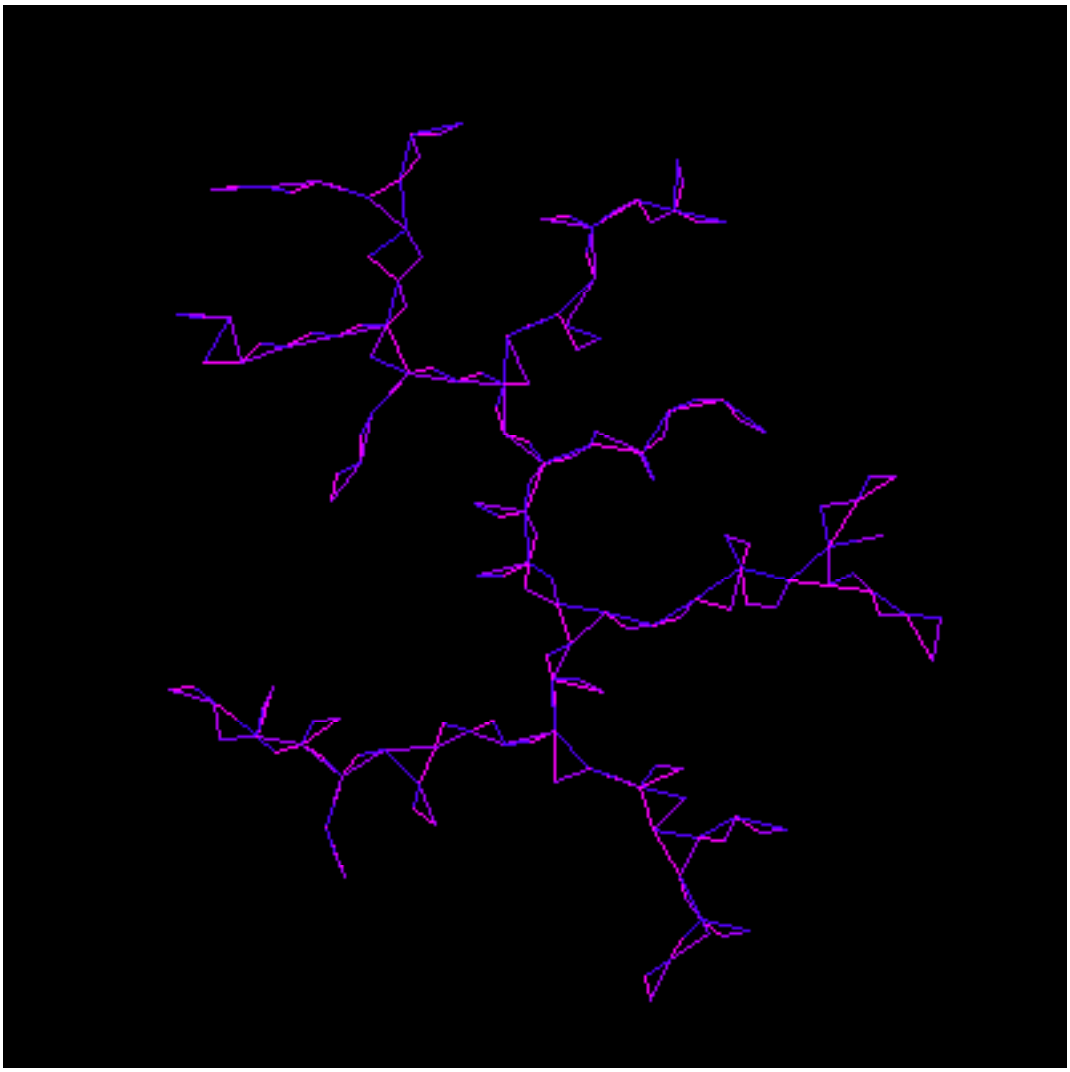

Figure 5.6: Convex Hull with $D = 2$

Figure 5.7 shows a progression of growth on a disc of radius 100 with a Δt of 5 at 100 and 910 nodes. Here the depth is divided every 10 children deep and it is drawn filled. We can see how the approximation of the tree is less tight, but how the reachable area is more accurately portayed.

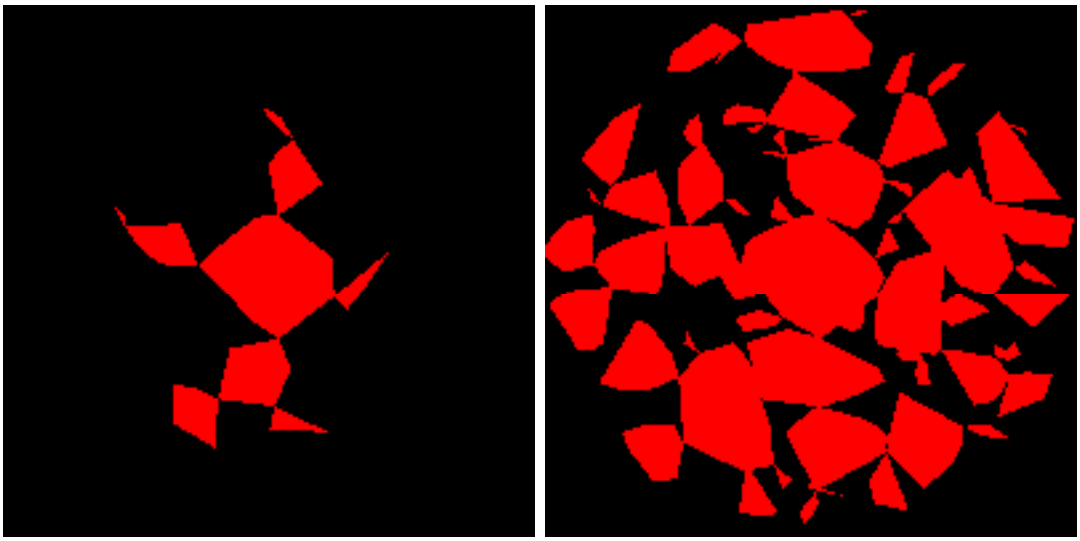

Figure 5.7: Convex Hull with $D = 10$ at 100 and 910 Nodes

5.2.2 Hausdorff Distance

The examples above lead to the question of how well this technique approximates the area, and furthermore at which depth we get a best approximation of the area. The Hausdorff distance is used to calculate the distance between two sets of points. This section presents results using the Hausdorff distance to measure the distance from the vertices in the convex hull to the points in the RRT.

Barnsley defines the Hausdorff distance between two sets of points, A and B as $h(A, B) = d(A, B) \vee d(B, A)$ in [8]. Here the $\vee$ operator indicates select the higher value of the left and right operands. $d(A, B)$ is defined as $d(A, B) = \text{Max}\{d(x, B)|x \in A\}$, and $d(x, B)$ is defined as $d(x, B) = \text{Min}\{d(x, y)|y \in B\}$. Thus the Hausdorff distance is built up from the notion of distance between two points $(d(x, y))$ to distance

between a point and a set ($d(x, B)$) to finally the distance between two sets ($d(A, B)$). It should be clear that $d(A, B) \neq d(B, A)$; the distance between two sets (in this definition) is not a symmetric calculation.

We would like to use the Hausdorff distance to gain a feel for the distance between two sets of points, the set of vertices in the RRT (we will denote them as T) and the set of vertices in the convex hull (denoted as C). However, $C \subseteq T$, since all vertices in the convex hull are points in the RRT itself. Consequently, $d(C, T) = 0$ for all $c \in C$, and to compute the Hausdorff distance we simply compute $d(T, C)$ since $h(C, T) = d(C, T) \vee d(T, C) = d(C, T) \vee 0 = d(T, C)$.

By using this metric we can get a measure of how close the convex hull that we have is to the actual tree itself. We ran 16 trials where we grew an RRT on a disc of radius 200 with step size of 5. After growing 3000 nodes of the tree, we calculated the convex hull of depths varying from division size of 1 up to the height of the tree (this maximum ranged from 62 to 73). For each we calculated the Hausdorff distance between the hull and the tree itself. In Figure 5.8 we show the plot of all 16 trials.

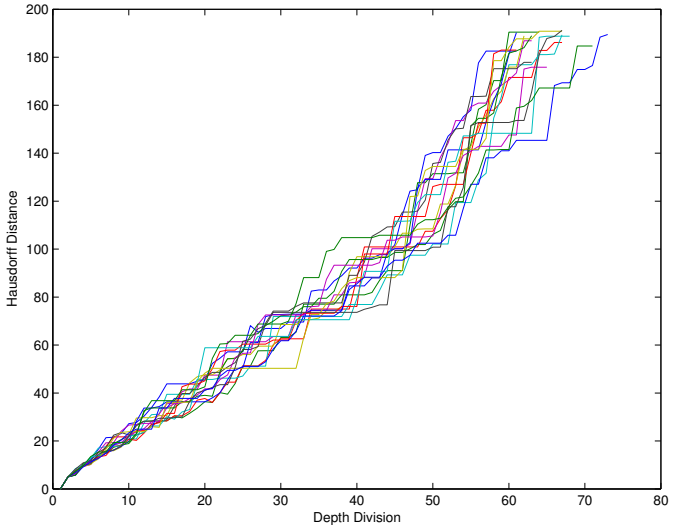

Figure 5.8: Plot of 16 Trials

This plot indicates that there is a nearly linear relationship between the division

of the depth we use to approximate the hull and the Hausdorff distance. Its positive slope indicates that as we increase the division, we get a worse approximation of the tree itself. Consequently, as we take fewer points from the tree we get a worse sense of the feel of the area covered. In Figure 5.9 we have a plot of the average of all 16 trials on the same axis. Here we see more clearly the linear relationship, except around a division of 45 nodes, the relationship breaks down and increases in slope—again indicating a worse approximation at higher depth divisions.

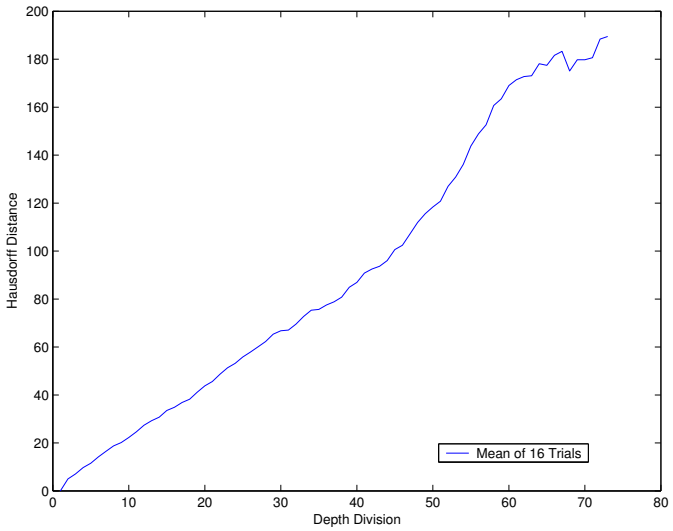

Figure 5.9: Mean of 16 Trials

From this data we gain a feel for what depth we should choose to make a best approximation of the reachable area. The graph suggests that in the range of $[7, 10]$ for the depth division we have a Hausdorff distance of $[14, 20]$. These numbers make sense in terms of the maximum Hausdorff distance we could achieve of 200—the radius of the disc we are growing in. We get a distance of 200 when the convex hull only contains points on the circle of radius 200, since here the distance calculate will be the distance from the center of the tree (at (0,0)) to the disc. Hence the Hausdorff distance of $[14, 20]$ indicates that we have a distance of $[7\%, 10\%]$ of the maximum distance possible.

5.2.3 Fractals

The RRT is often characterized as looking like a fractal pattern; however, since its growth is random it does not possess the property of strict self-similarity that is necessary for fractal patterns. The convex hull patterns, particular those that are filled, exhibit this same visual appeal, but yet again are not exactly self-similar and consequently are not fractals in the traditional sense of having a self-similar unit. Barnsley discusses the calculating of a fractal dimension in [8] by counting the number of units needed to cover a pattern. Fractal dimensions give a notion of how densely a space is covered by a particular pattern, and are a particularly useful concept in comparing fractals to each other [8].

By using the hulls to represent units covering the RRT, we can gain an approximation of the fractal dimension of this hull. By maintaining a measurement of the average area of the hulls, we can determine a dataset where we have the number of hulls necessary to cover the RRT as well as their average size. Essentially, we can run a trial similar to the one used in the previous section. We grow 3000 nodes of an RRT on a disc of radius 200 and step size of 5. We next iterate through all possible divisions of the depth and keep track of the number of hulls needed to cover as well as their average area. Figure 5.10 shows this process in action at depths of 2, 10, 20, 30, 40, and 67. Note here that the images have been resized 100 pixels by 100 pixels (i.e. a disc of radius 100) to fit here.

From this we can make a plot of the $\log(N)$ and $\log(A)$ where N is the number of hulls needed to cover the area and A is their average area. From this we can take the negative slope of the fitted linear plot and we get a value of 1.22. In Figure 5.11 we show a figure with two such trials plotted and the linear fit for them. It is important to note the jagged part of the line near the center. This is the result of the way the data was calculated and plotted. The points were plotted in order of decreasing depth division; however, sometimes when depth division was decreased the number of hulls

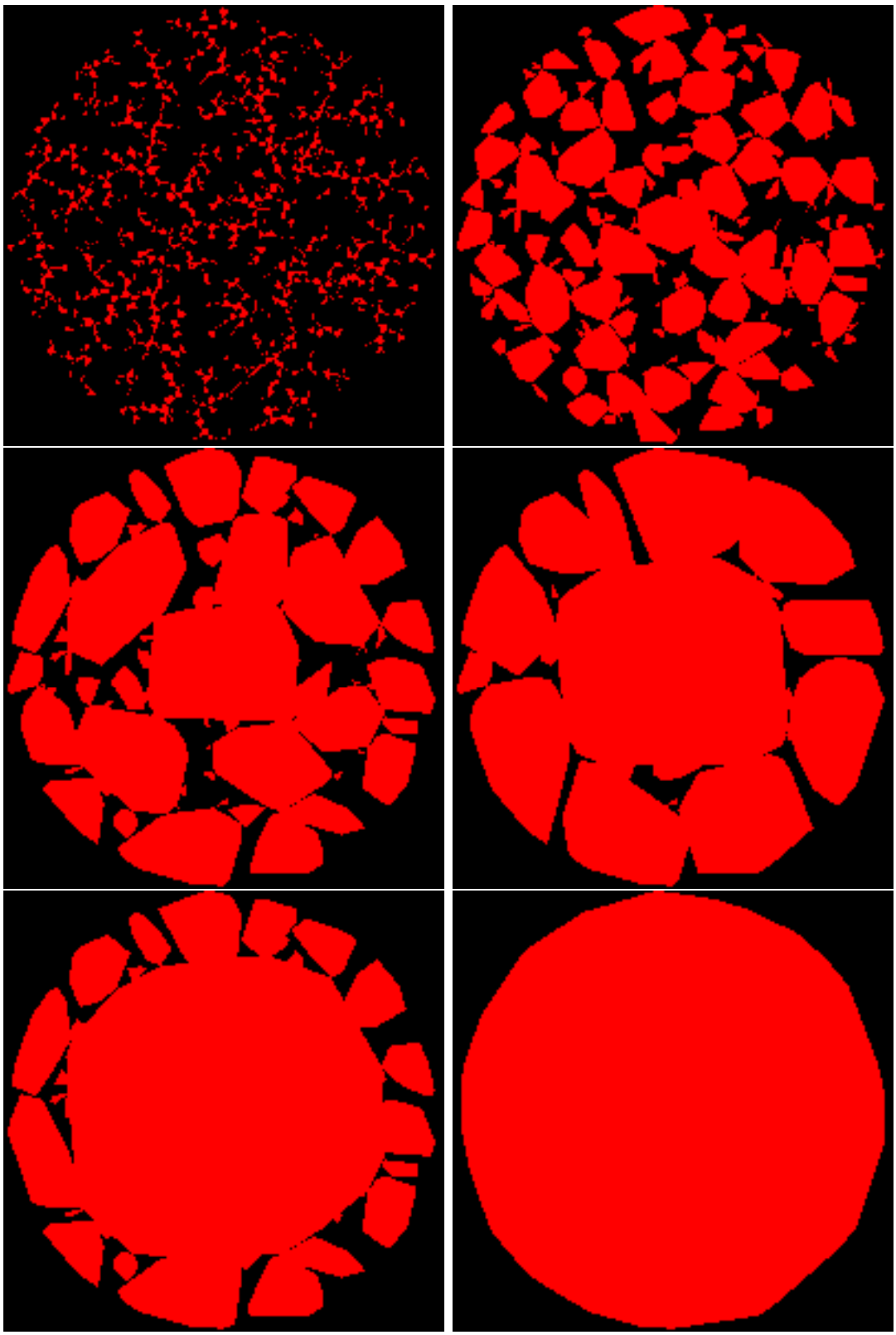

Figure 5.10: Convex Hull of an RRT at Depth Division 2, 10, 20, 30, 40, and 67

needed to cover the area increased (instead of the expected decreasing). Consequently, a small jiggle in the line occurred.

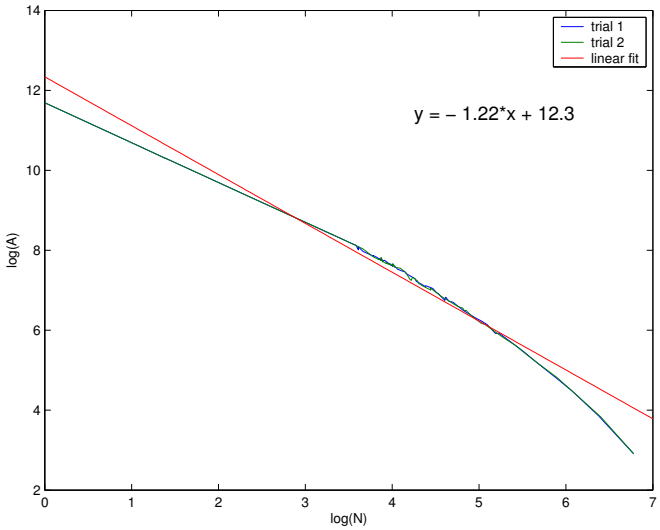

Figure 5.11: Plot of Fractal Dimension

As one can see from the plot, it is close to being linear, but not exactly linear (it begins to curve off when $\log(N)$ equals 4). Part of the problem here is the data only approximates the values that an actual fractal with a self-similar unit would have. Even with the expected error from approximating a real world data set, we can conclude that the hull of the RRT (and hence the RRT itself) falls somewhere between a plane and a line since the fractal dimension is in the range $[1, 2]$.

5.2.4 Hull Area versus Reachable Area

One question that still remains is how well the RRT is actually covering the reachable area of the region. Given that we can compute the convex hull area, we can gain an experimental result as to how quickly (as in how many nodes) the RRT covers the total reachable area. This data definitely corresponds to what our division of depth is as well as the step size and the size of the reachable region. In addition, it is also a function of the area itself—obstacles in the space of the RRT affect how it fills the

space. It is stated in [44] that the RRT is probabilistically complete (i.e. it will fill up the space eventually), but gaining a way to measure how well it fills the space is an important, unanswered question. Previously, it has been studied what happens when you change the step size, and above we examined how the depth division affects our measurements. In this section we study four examples where we vary the state space to control how the reachable area gets explored.

Our first example is simply the same disc of radius 200 we have been using. For this example as well as the remaining three we use a step size of 5 and a depth division of 7. The depth division was chosen as rationalized above in Section 5.2.2. We ran ten trials with these parameters, and a plot of the percentage of hull area to reachable area versus the number of nodes is shown in Figure 5.12. We took data points every 100 iterations of the RRT_Extend() algorithm. The initial state in the tree was the point at (0,0), and we grew the tree until the percentage of the hull area was 100% of the reachable area. In this plot we see all ten trials were very similar in their results.

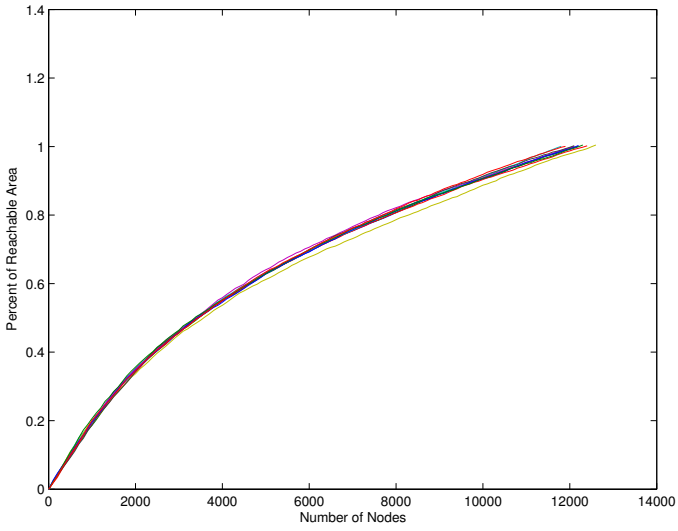

Figure 5.12: Plot of Hull Area vs. Reachable Area, No Obstacles

Our next example involved random shapes placed throughout the disc. These obstacles are not particularly inhibiting to the growth of the RRT, with the exception

of the corner on the left side of the bottom obstacle that the RRT was forced to navigate around. This particular obstacle is placed so that the left side creates a narrow gap with the circle of radius 200, and the right side intersects with the circle, blocking growth of the RRT completely. In Figure 5.13 we show both the initial obstacle layout as well as the plot (the white + represents the root of the RRT) of the resulting ten trials. Note here that the total reachable area is a disc of radius 200 minus the area of the obstacles.

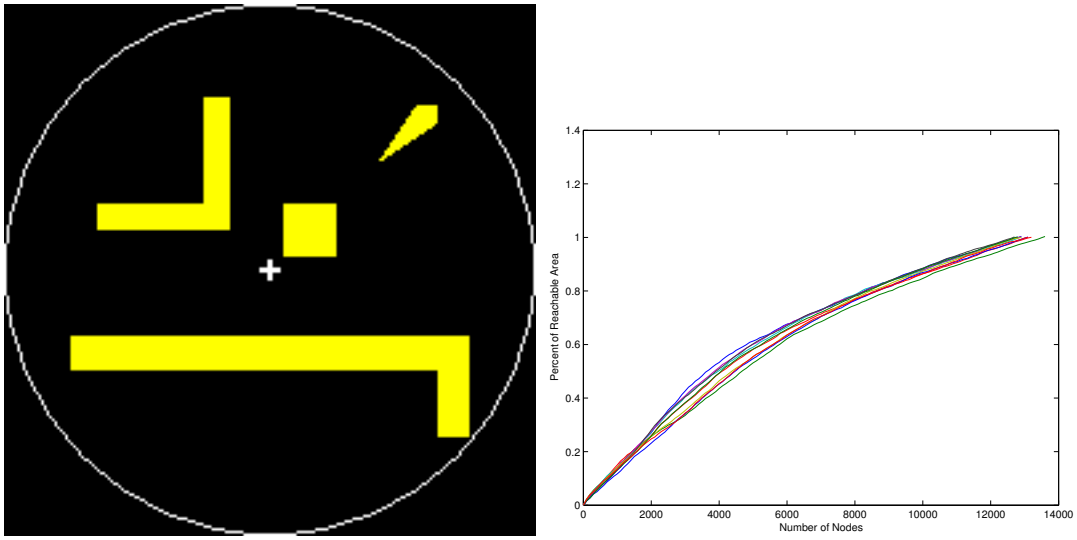

Figure 5.13: Hull vs. Area Initial Configuration and Plot, First Obstacle Set

Again the ten trials are relatively similar in growth. However, there is some slight separation in the trials between the ranges of 3000 to 5000 nodes. This is an indication of the variations in time required to get around the one obstacle.

Our third example attempts to control the plot of this to a finer degree. We created another trial where we moved the initial state to (100,0) and created an obstacle forcing the tree to grow through a narrow passageway at (0,0). Again we have a figure of the obstacle layout as well as the resulting plot, shown in Figure 5.14. Note here that the white + indicates how the initial point in the tree has been moved to (100,0).

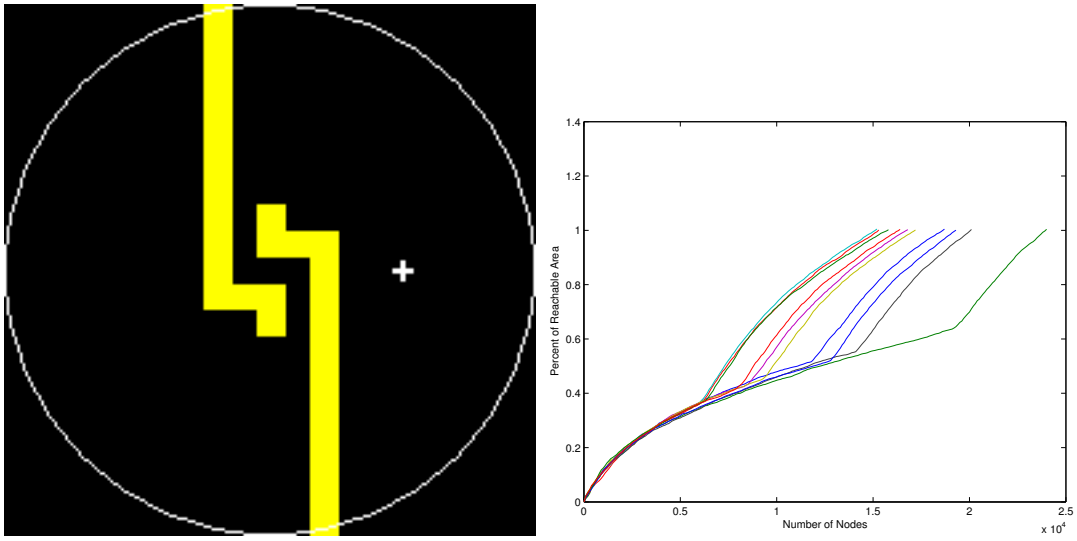

Figure 5.14: Hull vs. Area Initial Configuration and Plot, Second Obstacle Set

The results here are more interesting. For each line in the plot one can see a clear break where the RRT slowed down to try to make its way through the obstacle. After it found a path it again rapidly explored the new open region. However, while it did always find this path, it did not find this path at the same point in growth—indicative of the probabilistic completeness of the RRT.

Finally, we increased the "difficulty" of our third example by adding two C-shaped obstacles around the initial state of (100,0) and the point (-100,0). These obstacles and the plot of the ten trials are shown in Figure 5.15 (the initial state is shown as a white +). Again we have stalls where the RRT had to find its way around a narrow passage. The first stall is to get out of the C-shaped obstacle on the left and the second is to make it through the narrow passage in the center.

These four experiments demonstrate the power of our hull and are useful in exploring the area reached by the RRT. One caveat of our hull is that the area of the summed convex hulls includes a small amount of error. When hulls overlap (which they do eventually as the RRT grows) we sum the overlapping areas twice. In addition, in the examples that had obstacles, when the hull had points that cause the

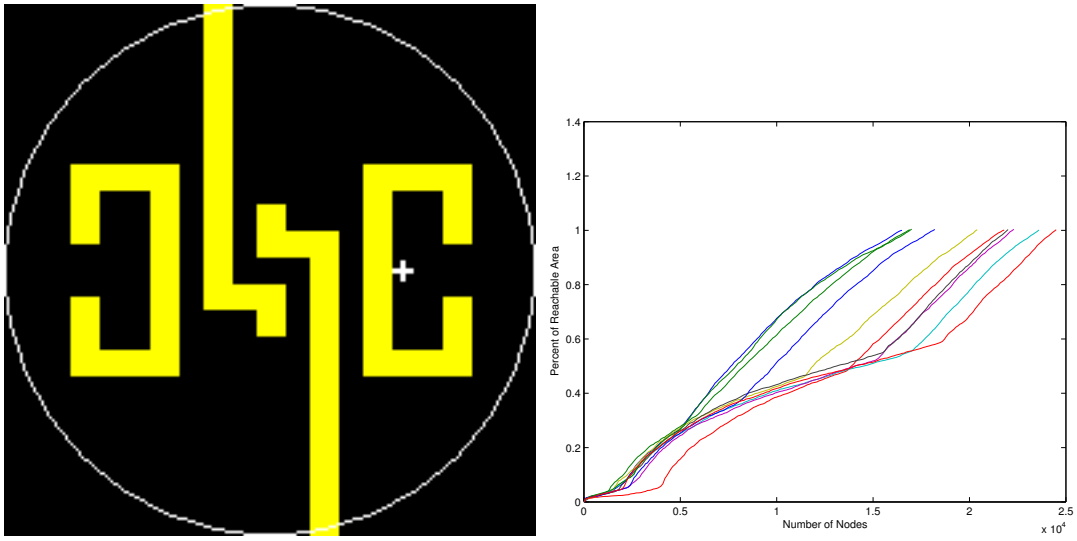

Figure 5.15: Hull vs. Area Initial Configuration and Plot, Third Obstacle Set

lines of the hull to span over the corner of an obstacle, this additional error is added in. However, for our purposes this slight error in calculation is acceptable.

5.3 Optimal Solutions

In addition to studying the area reachable by the RRT we were also interested in gaining a measure for the path planned from the RRT as compared to the optimal path. Here we present the results of a series experiments that were conducted to gain a sense of how close to optimal the paths grown by the RRT are. The results of these experiments present a distribution for the ratio of the RRT path versus the optimal (straight-line) path.

All experiments were conducted by growing 1000 separate RRTs rooted at the center of a disc (or in three dimensions, a sphere of radius 200). The step size used in these experiments was varied, and each RRT was grown until a node was created in the tree that was within one step of the goal. Afterwards, by following the parent links to the root of the tree, we calculated the RRT path by summing the distance from goal to root. Then the number of nodes, the optimal distance to goal (i.e.

the straight line path), and the RRT distance to goal was saved. No obstacles were present for any of these experiments.

5.3.1 Fixed Goal State

The first set of these experiments involved having the goal state fixed at the point (50,50) for each of the 1000 iterations. In Figure 5.16 we show a histogram of 20 bins of the resulting distribution for a step size of 5. On the x-axis we have a ratio of RRT path to optimal path and the y-axis is the number of points in that particular bin. Note here in this graph that we have clipped off the x values greater than 5, of which there were very few.

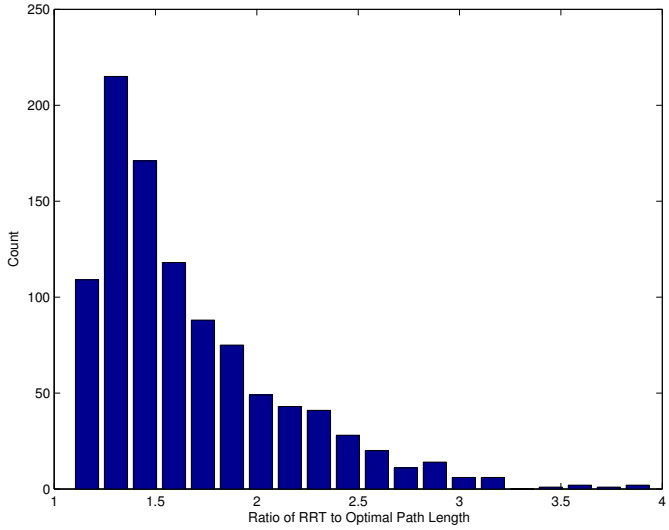

Figure 5.16: Histogram of Fixed Data Set for Step Size 5, 20 Bins

This distribution above is indicative of a lognormal distribution. The mean is $\mu = 1.6678$ and the standard deviation is $\sigma = 0.4646$. The distribution we have is however shifted right by 1. This is the result of the smallest ratio of RRT path to optimal path being 1 (in the case where the RRT path was exactly the straight line path). By transforming our histogram by taking $\log(x - 1)$ we gain a bell-shaped normal curve as shown in Figure 5.17. This histogram is also drawn with 20 bins.

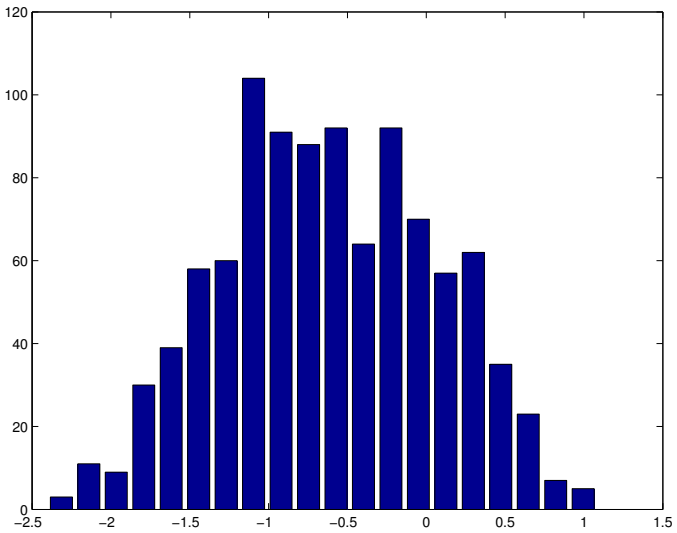

Figure 5.17: Transformation of Fixed Data Set, 20 Bins

We also ran the same experiment by varying the step size in the range of $[6, 50]$. In Figure 5.18 we show a summary histogram of all of the step sizes for both the original lognormal distribution (left) and the transformed distribution (right).

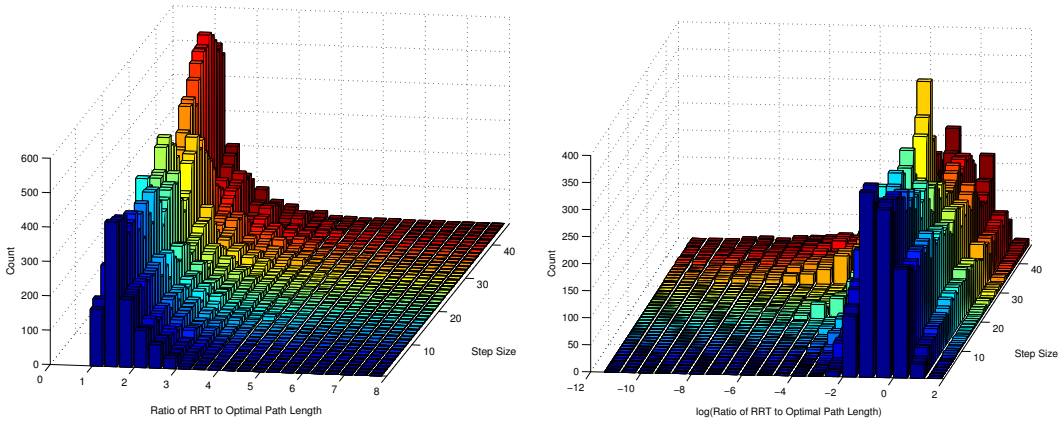

Figure 5.18: Summary of Fixed Goal State for Step Sizes $[5, 50]$

Not surprisingly, the distribution for every step size was indeed a lognormal distribution, and similarly, after taking the logarithm of the data set, we gain a set of normal curves.

5.3.2 Random Goal State

Our second experiment involved choosing a random goal state at each of the 1000 iterations. Here all of the specifics are the same as in the last experiment except for the goal state, which was chosen as integer located anywhere within a disc of radius 200 centered at $(0, 0)$. Again we plotted a histogram of the ratio of RRT path to the optimal path for a step size of 5 shown in Figure 5.19 with 20 bins.

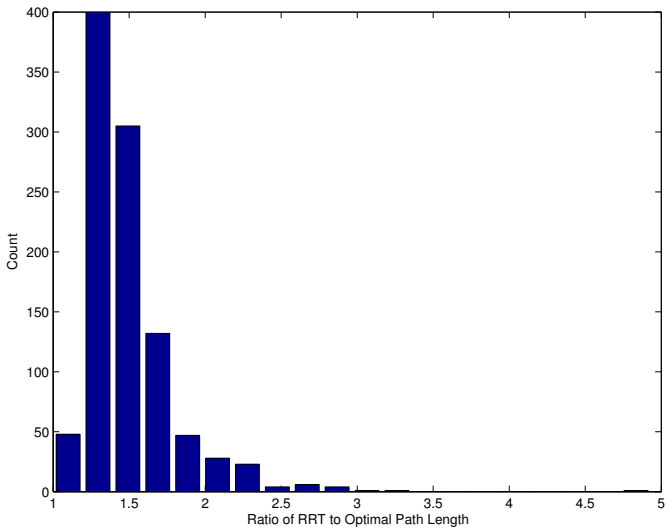

Figure 5.19: Histogram of Random Data Set for Step Size 5, 20 Bins

Here we again obtain a lognormal distribution with mean $\mu = 1.4990$ and standard deviation $\sigma = 0.3048$. We can take the same transformation $(\log(x - 1))$ to get the resulting normal curve in Figure 5.20.

The results here are the same as the fixed data set, we have a resultant normal curve. However, we have a large negative outlier point on the left hand side of the plot. Since we are picking random points anywhere within a disc of radius 200, occasionally we will pick a point that is within a disc of radius 5. Consequently, we will reach our random goal point within 1 step. In this case, the random point selected was $(-2, 1)$ which is a distance $2.23607 < 5$ from the root of the tree. However, since we take $\log(x - 1)$ of the data, we cannot calculate the log of $1 - 1 = 0$. So we approximate

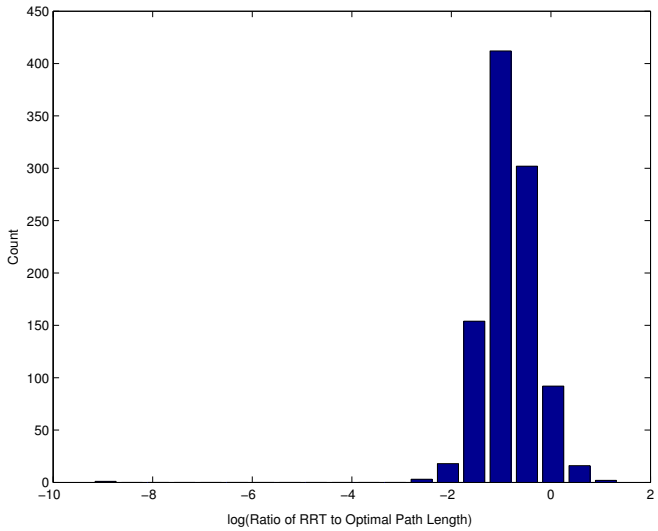

Figure 5.20: Transformation of Random Data Set, 20 Bins

the value by replacing all instances of 1 with 1.0001, and $\log(0.0001) = -9.2013$, explaining the outlier.

For completeness, we show the resulting histograms of the data (left) and its transformation (right) in Figure 5.21. One can see here that as step size increased, we encountered more and more instances where we had to convert 1 to 1.0001. As the step size increased, a larger portion of the disc of radius 200 was reachable in 1 step.

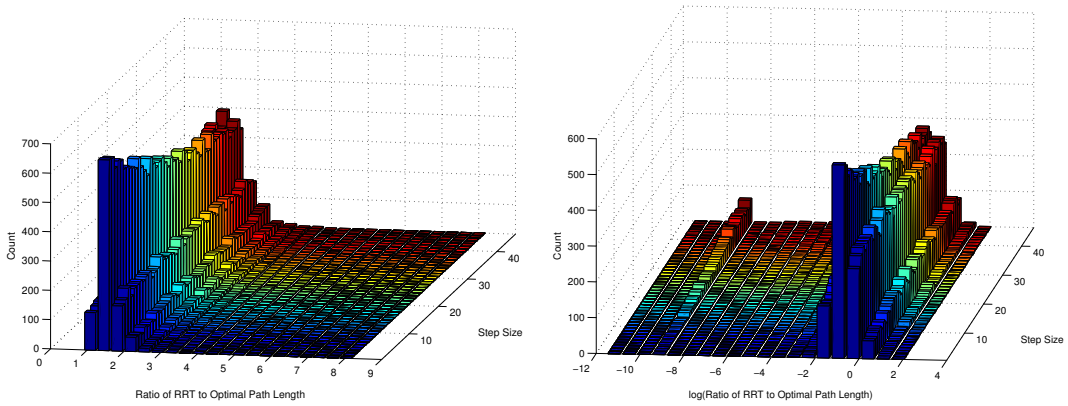

Figure 5.21: Summary of Fixed Goal State for Step Sizes $[5, 50]$

5.3.3 Three Dimensions

We were also interested in varying the dimensionality of the problem as well as the step size. So we chose to repeat our experiments for both fixed and random data points in a three dimensional space. For a fixed point we chose $(50, 50, 50)$ and we grew our RRT within a sphere of radius 200. Again we grew 1000 iterations of the RRT, and we varied the step size from 10 to 50 in increments of 5. The results of this series of experiments again presented us with a lognormal distribution.

More interesting are the results we obtained when we plotted the mean values of the ratio of RRT to optimal path lengths. In Figure 5.22 we show a plot of means for all of the 3d trials as well as the original 2d trials with a fixed goal state. Figure 5.23 shows a similar plot of means except with a random goal state.

These plots clearly show that in three dimensions the mean values of the ratios are much higher than the two dimensional cases. It also shows that as step size changes, the mean value of the ratio does not change significantly for fixed or randomly selected goal points. However, in the fixed case, there are two distinct dips in the mean ratios, in the 2d case around 35 and in the 3d case around 40. This is the result of the step size approaching 50% of the distance to the fixed goal point. As step size increases past this threshold, it becomes increasingly more likely that the RRT path will be closer to the optimal path because we are more likely to find paths in one or two steps to the goal.

One final comparison can be made from the fixed goal states to the random goal states. Both of the fixed trials had consistently higher mean ratios than the random goal state trials. This result can be attributed to the fact that in the random goal state trials, we occasionally chose points where the goal was in one step of the root of the tree, causing more optimal paths on average (since we had more ratios of value 1).

We show the standard deviations of all of these trials in both plots by drawing

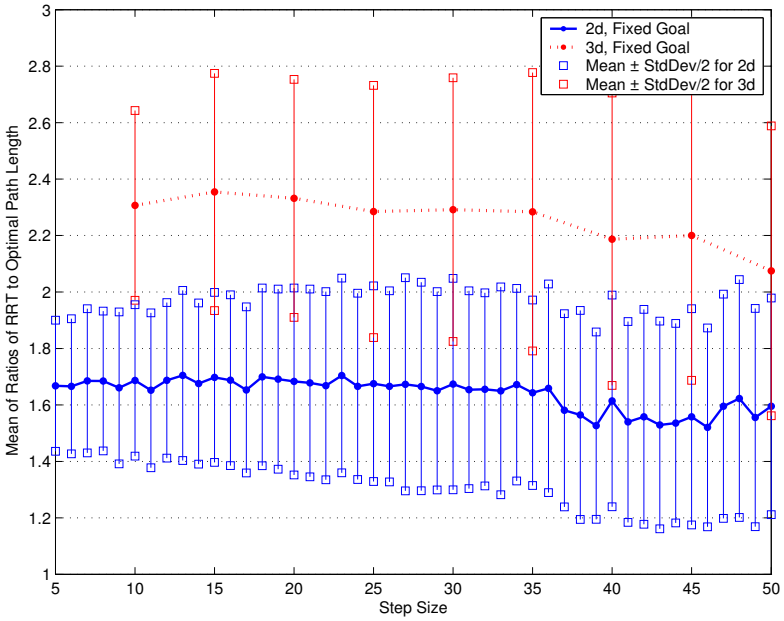

Figure 5.22: Mean Ratios for 2d and 3d Experiments, Fixed Goal

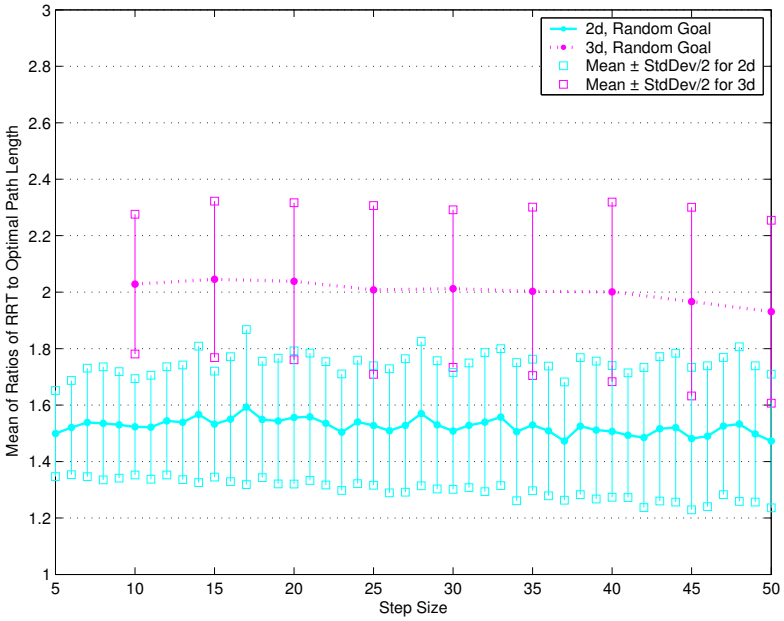

Figure 5.23: Mean Ratios for 2d and 3d Experiments, Random Goal

squares which stem from each value indicating the mean $\pm\frac{1}{2}$ of the standard deviation. In all cases the standard deviations increased as a function of step size, the expected result as larger step sizes cause the extremes in the ratio of RRT path to optimal

path to be more likely. Since we are taking fewer, larger steps, we are more likely to either hit an optimal path or completely miss one.

5.4 Bounded RRTs

One of the downfalls of the RRT algorithm is the heavy reliance on nearest neighbor calculations. At each iteration of the RRT_Extend() algorithm, one must calculate the nearest neighbor to the random point picked. Consquently, as the RRT gets larger, this becomes increasingly less efficient as more and more nodes must be searched for the nearest neighbor. Some solutions have been proposed in [7] using different data structures such as the kd-tree to store to the nodes of the RRT as it grows.

In this section we would like to suggest and investigate a second structure based on the metric tree, a concept originally introduced in [57]. This idea was later generalized to the concept of the vantage-point tree or vp-tree [58]. Broadly, metric trees work by storing a distance value with each node of the tree at construction. By saving this distance information we are able to trade distance calculations for comparisons, and using these comparisons to determine nearest neighbors. The assumption here is that metric functions are often expensive to calculate, and simple comparisons can be done more rapidly.

The basic algorithm for the vp-tree appears in [11, 58], but we will reproduce it here briefly. Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of data points, we select an arbitrary vantage-point $s_{vp} \in S$. Next determine the median distance by calculating the value $M =$ median of $\{\rho(s_i, s_{vp}) | \ \forall s_i \in S\}$ where ρ is the distance metric being used. Divide S into two subsets, S_l and S_r where $S_l = \{s_i | \rho(s_i, s_{vp}) \leq M\}$ and $S_r = \{s_i | \rho(s_i, s_{vp}) > M\}$. Now recursively divide S_l and S_r. Thus, we can search our vantage-point structure with a query point q by only needing to calculate the distance from q to each s_{vp} that we chose, and from their only needing to search either in S_l or S_r.

The vp-tree has already been studied as a powerful structure, particularly in high dimensional spaces, for improving nearest neighbor queries. However, it is not immediately applicable to the RRT algorithm. One major problem is that the vp-tree structure is not iterative. All points are known at construction time, and in fact there is a large, expensive construction step (inherent in all metric trees), which cannot be performed since the point set of the RRT is not known yet. However, we can take the same idea of exchanging distance calculations for comparisons and apply that to our RRT.

5.4.1 Maximum-Minimum Bounds

Our first idea was to use the origin of the RRT as the vantage point. Our construction of a vp-tree is modified as follows. We store the distance to the root as we add each node into the tree. We also store a value for the tree representing the maximum distance of any node to the root. Instead of maintaining a median M we use this maximum distance to subdivide our points. Also, instead of maintaining a recursive search structure, we keep the original RRT hierarchy.

Our nearest neighbor algorithm is modified from the basic idea of iterating through each point and calculating the distance to query point. Instead we calculate a median M based on the maximum distance to select only particular points to calculate the distance to. We use a constant $k \in [0, 1]$ specifiable on input to determine a nearest neighbor as shown in Algorithm 5.1. In each node the data field $distToRoot$ stores the distance to the root of that particular node. In the tree, the data field $maxDist$ stores the current maximum distance of any node to the root, and the field k stores the constant by which we determine M.

Algorithm 5.1 (Nearest Neighbor). *The following algorithm determines the nearest neighbor in a tree T to a point x.*

```
1  Nearest_Neighbor(T, x) {
2     //the distance from x to the root.
3     dist_x ← x.metric(T.root);
4     //stores the minimum distance of any node to x
5     //initially set to largest possible value
6     dist_min ← T.maxDist;
7     //the calculated median
8     M = T.maxDist * T.k;
9
10    //is x outside or inside of the bounds?
11    IF dist_x ≤ M {
12       //search the nodes inside of radius M
13       FOR EACH n ∈ T {
14          IF n.distToRoot ≤ M {
15             dist_tmp = x.metric(n);
16             IF tmp < dist_min {
17                dist_min ← dist_tmp;
18                n_nearest ← n;
19             }
20          }
21       }
22    } ELSE {
23       //search the nodes outside of radius M
24       FOR EACH n ∈ T {
25          IF n.distToRoot > M {
26             dist_tmp = x.metric(n);
27             IF tmp < dist_min {
28                dist_min ← dist_tmp;
29                n_nearest ← n;
30             }
31          }
32       }
33    }
34
35    RETURN n_nearest;
36 }
```

The idea here is that we have a moving partition with which we use to prevent unnecessary calculations of the metric. However, we do have to make a pass through the entire tree because we are limited in construction of our data structure (our tree does not possess the recursive subdivision of a metric tree). If we had tried

to subdivide the points into a hierarchy earlier, we would have to rebuild our tree structure as the bounds move outwards.

In Figure 5.24 we show the progression of growth of a bounded RRT. We have illustrated the maximum bound by a blue circle and the minimum by a red circle. In this example we have set $k = 0.75$. Here the RRT is grown on a disc of radius 100 with a Δt of 5. The image on the left has 50 nodes and the image on the right has 500. One already sees a potential problem with this idea in that the RRT is biased to grow in only the direction that maintains the furthest distance. This result occurs because we only query nodes in the outer ring for their nearest neighbor when the random state selected is outside of the blue ring. However, despite this limitation the RRT does eventually cover the entire region, by branching around the blue circle.

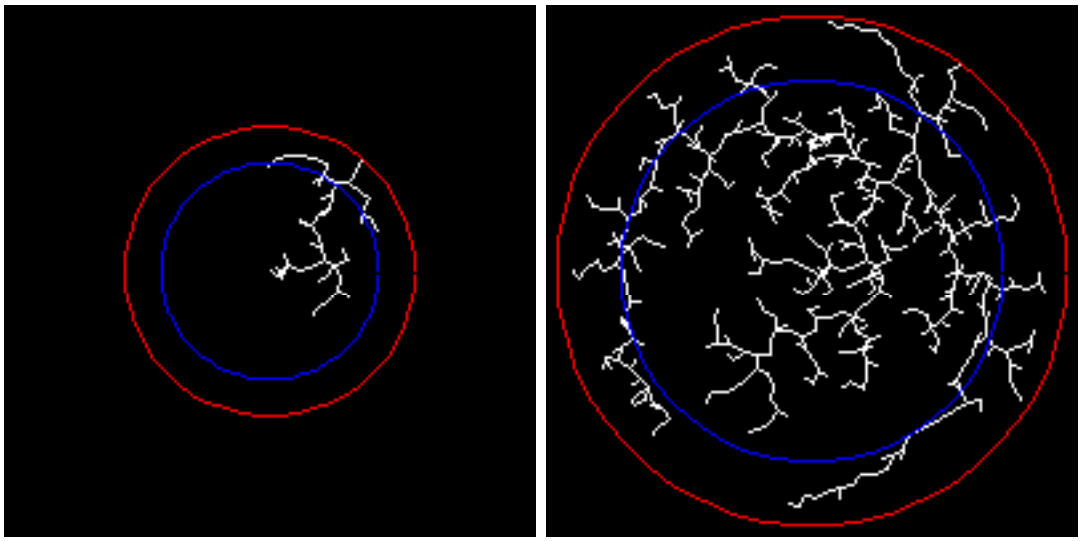

Figure 5.24: Bounded RRT with $k = 0.75$

5.4.2 Multi-way Bounded RRT

Significant improvements to the vp-tree are a current topic of research. Particularly, ideas exist such as the mvp-tree tree (multiple vantage points) and the multi-way vp-tree [11]. The former uses a set of vantage points to divide the region up into sets

of regions that are intersection of spheres, and the second uses a set of k medians for one vantage point. In this section we explore an application of multi-way vp-trees to our bounded RRT construction.

The idea follows immediately from our previous discussion. Instead of storing a maximum bound and using a constant k to modify the set of points we query, we use k as an integer to divide our searchable region as a set of concentric circles. Figure 5.25 illustrates the growth of an RRT in this manner as well as showing the set of concentric circles. Here we have chosen $k = 5$. Again the RRT was grown on disc of radius 100 with a Δt of 5. However, the images shown are at 50 nodes and 1000 nodes—demonstrating how extra exploration is needed if the bounds become too restrictive.

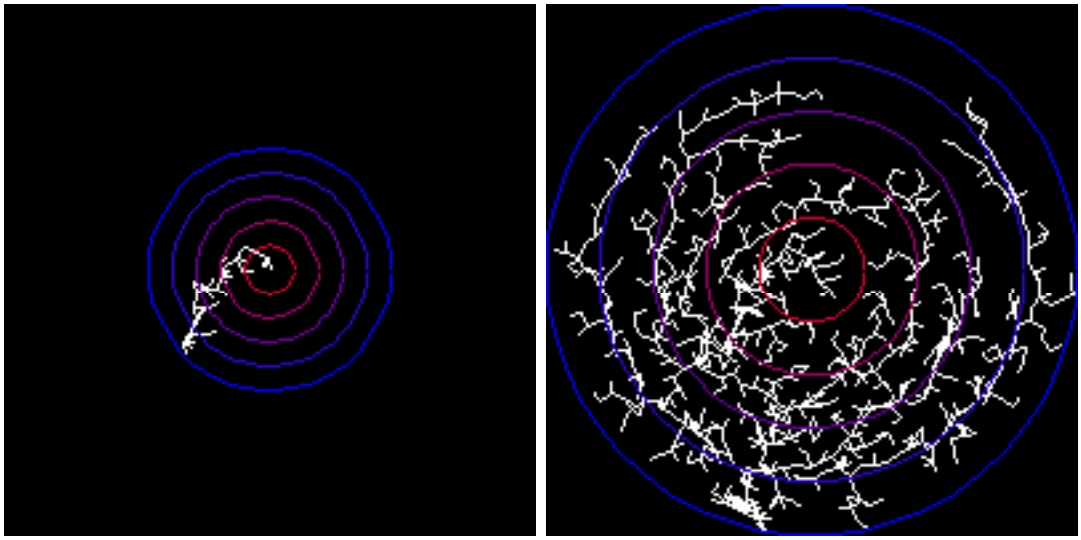

Figure 5.25: Bounded RRT with $k = 5$

One can see in this example that the RRT does eventually fill the space, but suffers from a similar flaw as in the maximum-minimum bounded RRT. The path is biased to grow only in one direction, and then curls around the boundaries to fill up the space. This issue becomes more apparent if we increase k. In Figure 5.26 we show the results of an RRT growing where $k = 15$. Here the RRT is grown on a disc of

radius 100 with $\Delta t = 5$. The image on the left has 50 nodes and the image on the right has 2000 nodes, indicative of 15 being too restrictive in this example.

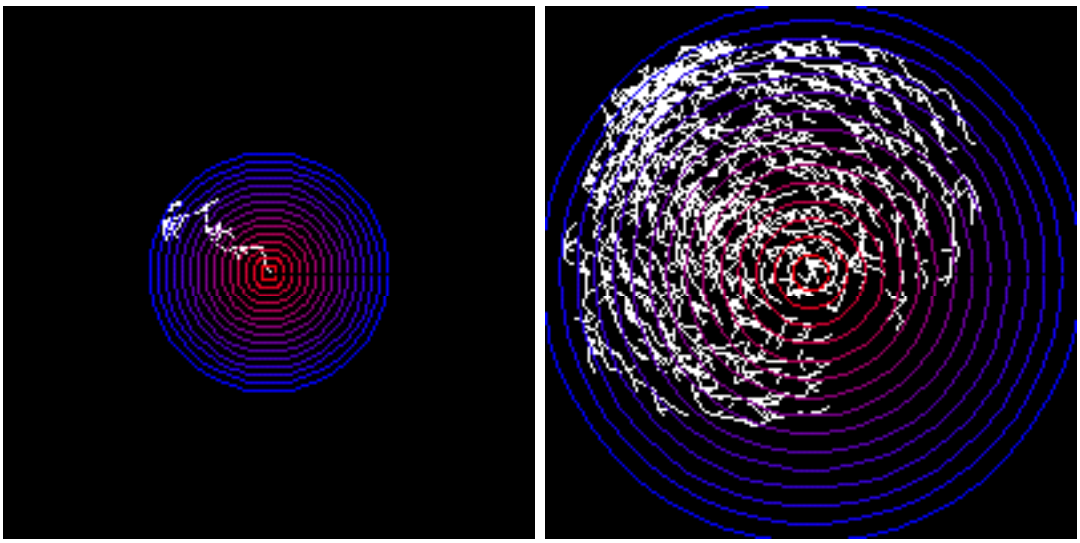

Figure 5.26: Bounded RRT with $k = 15$

It becomes more than apparent by these visuals that by bounding the RRT we can control and limit the regions the RRT grows in. However, we do save nearest neighbor calculations significantly. In the case where $k = 5$, on average we only queried 20% $(= 1/5)$ of the points calculated. This trend continues with other values of k; hence, when $k = 15$, we only queried $6.67\% = 1/15$ of the nodes (on average). Thus, we make an exchange of uniform growth for speed of querying—the original idea behind the vp-tree. The assumption in both cases however is the same: the points that we query are most likely to be the points we want to grow from.

An issue noted in [11] is that at higher dimensions the volume of the spherical shells becomes increasingly thin. Thus, at higher k values in high dimensions, the number of points queried is reduced so far that often nearest neighbor calculations will be increasingly inaccurate. Thus we have a mixed result; the ideal solution will have a balance of reducing the number of nodes queried but preventing the thinning of the shells to allow for uniform growth and exploration.

Chapter 6

Conclusions

6.1 Results Summary

In this thesis we presented the development of a visual tool using RRTs to study the reachability of hybrid systems. In addition we further explored the RRT and provided different visual, experimental, and statistical results related to them.

In **Chapter 3** we presented an overview of our extension to the Motion Strategy Library (MSL) [42]. First we explained the need for such a tool as compared to our initial, coded examples. We followed this with an explanation of the hierarchy behind the MSL as well as a brief discussion of using the MSL. We then described our extension to the MSL, detailing the major objects that we modified, and what features we added. We also described how the MSL was adapted to allow a hybrid system description input as well as how we extended the user interface of the MSL to contain hybrid state information

Chapter 4 followed up with a presentation of the experiments that we performed with the extended MSL. We explored the stair climber problem first discussed by Curtiss in [21]. Focusing first on the two dimensional case, we examined biasing the RRT to grow towards switching regions as a set of subgoals. We explained how we took

advantage of the features of the MSL and extended this problem to three continuous dimensions. Hybrid system examples with nonconstant dynamics (e.g., bouncing balls) were also studied. Finally, in a big step toward making our tool comparable to other hybrid automata tools, we created a rectangular hybrid automata example to study in our extension. Related to rectangular hybrid automata, we explored an improvement to the RRT by using the concept of multi-action growth.

Our last major chapter, **Chapter 5**, presents new results regarding the RRT algorithm itself. We discuss visual representations of the reachable region of the RRT through use of a contour map visualization. Next we attempted to study a convex hull of an RRT and proposed a scheme for estimating the area reached by the RRT to arbitrary precision. We also made a statistical analysis of the paths grown by RRTs as compared to the optimal solutions. Finally, we discussed an improvement to the RRT by modifying the vantage-point tree data structure and synthesizing it with the RRT's iterative construction.

6.2 Open Issues and Future Work

RRTs continue to provide positive results, and there are many areas that remain to be studied in relation to them. In addition, hybrid systems are also a current topic of much research, leaving many areas to further explore:

Nonlinear Hybrid Automata: We have taken our experimental examples to the level that other researchers in hybrid systems have as well. It is well known that many hybrid automata can be approximated as rectangular hybrid automata, which are decidable [35]. However, we believe our approach is applicable to nonlinear descriptions as well since RRTs have been used for nonholonomic problems with great success. In developing our extension to the MSL we left the implementation sufficiently open that by changing the `Model` object, one

could easily incorporate nonlinear dynamics. Our bouncing ball example is a simple nonlinear one, but more nonlinear problems should be pursued in the MSL.

Maneuver Automata: Suggested in [25] and later developed into the idea of a Robust Hybrid Automaton in [24, 26] these descriptions provide a powerful link between sampling-based planning and hybrid systems. Their examples, which model the motion of complex vehicles using RRTs and other sampling-techniques use similar ideas as the experiments we have performed. Multi-action growth is also similar to the idea of maneuver automata since it pursues a set of input directions. Consequently, we feel pursuing one of their examples and applying it to our extended MSL is highly applicable.

MSL Development: We believe that there is a significant amount of work left with the MSL from a development end as well. Potential features to be added include:

Multi-State Visualizations: Our current solutions to visualizing hybrid systems include drawing only one state or drawing all states on top of each other in different colors. A different visualization that could be more flexible would allow the user to select which states they wanted to view (either on top of each other or in separate windows simultaneously).

Problem Description Interface: Currently all problems are specified as text files but code must still be written to model the dynamics (a `Model` object) and the physical representations (a `Geometry` object) of the system. There could be other more flexible solutions to enter in this portion of the problem definition.

Metric Functions: Unexplored issues remain regarding different metrics. We are left with unanswered questions studying what different metrics we could use to describe the state space. We discussed the work of Curtiss [15, 21] in Section 4.1

and explained how his particular metric works well in terms of a stair climber. Additional work is necessary to determine what other metric functions will also work well with our hybrid RRTs.

Random State Selection: We experimented some in Section 4.1.2 with adjusting how we picked random states and using the random state selection to bias our planners to grow RRTs towards switching regions. There is still more work to do in creating more intelligent planners that are biased to grow towards switching regions in optimal ways. Ideally we would like a planner that provides a proper balance between exploring both the continuous and discrete state spaces. In addition, for examples such as the rectangular hybrid automata, it would be better to sample from only the set of points in the space that are in the "forward" direction (defined by the dynamics), as opposed to the set of all points.

Multiple RRTs: In [21] the idea of growing many RRTs for multi-agent planning is discussed. Similarly, would it be feasible/useful to grow a tree in multiple states to approach RRTs for hybrid systems? We discussed multi-action growth for rectangular hybrid automata in Section 4.3.2, which is growing a single RRT in multiple directions simultaneously. What about growing multiple trees, one from each switching region? If so, how would the intersections of n-trees be calculated and how would one know when to stop or balance the growth of the forest of trees? What are the ramifications of growing multiple trees in a nonholonomic problem space?

Despite the large list of remaining areas to research, we are pleased with the success of applying sampling-based techniques to study hybrid systems. We have created the foundation for a tool by which we can continue studying hybrid systems using RRT variants. Also, we have further researched the RRT itself, creating a better understanding of sampling-based techniques as a whole.

Works Cited

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. -H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. In *Theoretical Computer Science*, **138**:3–34, 1995.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. -H. Ho. Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems. In *Hybrid Systems I*, Lecture Notes in Computer Science, **736**:209–229, Springer-Verlag, 1993.

[3] R. Alur, T. A. Henzinger, and P. -H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pp. 2–11, IEEE Computer Society Press, 1993.

[4] E. M. Arkin, Y. -J. Chiang, M. Held, J. S. B. Mitchel, V. Sacristan, S. S. Skienna, and T. -C. Yang. On minimal-area hulls. In *Proceedings of the 1996 European Symposium on Algorithms*, Lecture Notes in Computer Science, **136**:334–348, Spinger-Verlag, Berlin, 1996.

[5] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear hybrid systems. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1790**:20–31, Springer, Berlin, 2000.

[6] E. Asarin, G. Pace, G. Schneider, and S. Yovine. SPeeDI - a verification tool for polygonal hybrid systems. In *Conference on Computer-Aided Verification*, Lecture Notes in Computer Science **2404**:354–358, Copenhagen, Denmark, July 27–31, 2002.

[7] A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 632–637, 2002.

[8] M. F. Barnsley. *Fractals Everywhere*. Academic Press, Inc. Boston, MA, 1988.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for the automatic verification of real-time systems. In *Proceedings of Hybrid Systems III*, Lecture Notes in Computer Science, **1066**:232–243, Springer-Verlag, 1996.

[10] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: The stanford temporal prover, user's manual. Tech Rep. STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Nov. 1995.

[11] T. Bozkaya and Z. M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, **24(3)**, 1999.

[12] M. S. Branicky. *Studies in Hybrid Systems: Modeling, Analysis, and Control*. Sc.D. thesis, Electrical Engineering and Computer Science Dept., Massachusetts Institute of Technology, Cambridge, MA, June 1995.

[13] M. S. Branicky and S. R. Chhatpar. Hybrid systems: Learning, planning, and control. In *Proceedings of the Eleventh Yale Workshop on Adaptive and Learning Systems*, pp. 196–200, New Haven, CT, June 2001.

[14] M. S. Branicky and S. R. Chhatpar. A computational framework for the simulation, verification, and synthesis of force-guided robotic assembly strategies. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1471–1476, Maui, HI, October 2001.

[15] M. S. Branicky and M. M. Curtiss. Nonlinear and hybrid control via RRTs. In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems*, South Bend, IN, August 2002.

[16] M. S. Branicky, M. M. Curtiss, J. Levine, and S. Morgan. RRTs for nonlinear, discrete, and hybrid planning and control. In *Proceedings of the IEEE Conf. Decision and Control*, 2003. Accepted.

[17] M. S. Branicky, M. M. Curtiss, J. A. Levine, and S. B. Morgan. Sampling-based planning and control. In *Proceedings of the Twelfth Yale Workshop on Adaptive and Learning Systems*, New Haven, CT, May 28-30, 2003. To appear.

[18] M. S. Branicky, R. Hebbar, and G. Zhang. A fast marching algorithm for hybrid systems. In *Proceedings of the IEEE Conference on Decision and Control*, pp. 4897–4902, Phoenix, AZ, December 7–10, 1999.

[19] M. S. Branicky, T. A. Johansen, I. Peterson, and E. Frazzoli. On-line techniques for behavioral programming. In *Proceedings of the IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[20] S. R. Chhatpar and M. S. Branicky. A hybrid systems approach to force-guided robotic assemblies. In *Proceedings of the IEEE International Symposium on Assembly and Task Planning*, pp. 301–306, Porto, Portugal, 21–24 July 1999.

[21] M. M. Curtiss. *Motion Planning and Control using RRTs*. M.S. thesis, Electrical Engineering and Computer Science Dept, Case Western Reserve University, Cleveland, OH, May 2002. http://dora.cwru.edu/msb/pubs/mmcMS.pdf.

[22] T. Dang and O. Maler. Reachabilitiy analysis via face lifting. In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1386**:96–109, Springer-Verlag, 1998.

[23] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, Lecture Notes in Computer Science, **1066**:208–219, Springer, 1996.

[24] E. Frazzoli. *Robust Hybrid Control for Autonomous Vehicle Motion Planning*. Ph.D. thesis, Aero/Astro Dept., Massachusetts Institute of Technology, Cambridge, MA, June 2001.

[25] E. Frazzoli, M. A. Dahleh, and E. Feron. A hybrid control architecture for aggressive maneuvering of autonomous helicopters. In *Proceedings of the IEEE Conference On Decision and Control*, December, 1999.

[26] E. Frazzoli, M. A. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicles motion planning. Technical report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[27] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. In *AIAA Journal of Guidance, Control, and Dynamics*, **25(1)**:116–129, May 2002.

[28] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. In *Information Processing Letters*, **1**:132–133, 1972.

[29] T. A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science*, pp. 278–292, 1996.

[30] T. A. Henzinger, P. -H. Ho, and H. Wong-Toi. A user guide to HyTech. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1995.

[31] T. A. Henzinger, P. -H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proceedings of the 16th Annual Real-Time System Symposium*, pp. 56–65, IEEE Computer Society Press, 1995.

[32] T. A. Henzinger, P. -H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. In *Software Tools for Technology Transfer*, **1(1)**, Springer, 1997.

[33] T. A. Henzinger, P. -H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. In *IEEE Transactions on Automatic Control*, **43(4)**:540–554, April 1998.

[34] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HyTech: hybrid systems analysis using interval numerical methods. In Gautam Biswas and Sheila McIlraith, editors, *Proceedings of the AAAI 1999 Spring Symposium on Hybrid Systems and AI*, 1999.

[35] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pp. 373–383, 1995.

[36] T. A. Henzinger and H. Wong-Toi. Linear phase-portrait approximations for nonlinear hybrid systems. In *Hybrid Systems III*, Lecture Notes in Computer Science, **1066**:377–388, Springer-Verlag, 1996.

[37] J. J. Kuffner and S. M. LaValle. RRT-Connect: an efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 995–1001, 2000.

[38] A. B. Kurzhanski, P. Varaiya. Ellipsodial techniques for reachability analysis. In *Proceedings of the Pittsburgh Conference on "Hybrid Systems - 2000"*, Lecture Notes in Computer Science, **170**:202–214, Springer, 2000.

[39] G. Lafferriere, G. J. Pappas, and S. Yovine. A new class of decidable hybrid systems. In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1569**:137–151, Springer-Verlag, 1999.

[40] K. G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a nutshell. In *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, September 1997.

[41] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.

[42] S. M. LaValle *et al.* Motion Strategy Library. http://msl.cs.uiuc.edu/msl/.

[43] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference On Robotics and Automation*, pp. 473–479, 1999.

[44] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: progress and prospects. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pp. 293–308. A K Peters, Wellesley, MA, 2001.

[45] Z. Manna and H. B. Sipma. Deductive verification of hybrid systems using STeP. In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1386**:305–318, Springer-Verlag, 1998.

[46] K. Melhorn and St. Näher. The LEDA platform of combinatorial and geometric computing. Cambridge University Press, 1999.

[47] X. Nicollin, A. Olivero, K. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems I*, Lecture Notes in Computer Science, **736**:149–178, Springer-Verlag, 1993.

[48] J. Preußig, S. Kowalewski, H. Wong-Toi, and T. Henzinger. An algorithm for the approximative analysis of rectangular automata. In *Proceedings of Formal Techniques for Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science, **1486**:228–240, Springer-Verlag, 1998.

[49] A. Puri. *Theory of Hybrid Systems and Discrete Event Systems*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1995.

[50] A. Puri, V. Borkar, and P. Varaiya. ϵ-approximation of differential inclusions. In *Proceedings of the 34th Conference on Decision and Control*, 1996.

[51] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *Computer-Aided Verification*, Lecture Notes in Computer Science, **818**:95–104, Spinger-Verlag, 1994.

[52] B. Silva and B. Krogh. Modeling and verification of sampled-data hybrid systems. In *Proceedings of the 4th International Conference On Automation of Mixed Processes: Hybrid Dynamic Systems*, pp. 237–242, 2000.

[53] B. Silva, K. Richeson, B. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamic systems using checkmate. In *Proceedings of the 4th International Conference On Automation of Mixed Processes: Hybrid Dynamic Systems*, pp. 323–328, 2000.

[54] B. I. Silva, O. Stursberg, B. H. Krogh, and S. Engell. An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In *Proceedings of the 40th Conference on Decision and Control*, Dec. 2001.

[55] S. Simić, K. Johansson, S. Sastry, and J. Lygeros. Towards a geometry theory of hybrid systems. In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1790**, Springer, 2000.

[56] C. J. Tomlin. *Hybrid Control of Air Traffic Management Systems*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1998.

[57] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, **40**:175–179, 1991.

[58] P. N. Yiannilos. Data structures and algorithms for nearest neighbor search in general metric spaces. *ACM-SIAM Symposium on Discrete Algorithms*, pp. 311–321, 1993.

Bibliography

Alur, R.; Courcoubetis, C.; Halbwachs, N.; Henzinger, T. A.; Ho, P. -H.; Nicollin, X.; Olivero, A.; Sifakis, J.; Yovine, S. "The algorithmic analysis of hybrid systems," In *Theoretical Computer Science*, **138** (1995) 3–34.

Alur, R.; Courcoubetis, C.; Henzinger, T. A.; Ho, P. -H. "Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems," In *Hybrid Systems I*, Lecture Notes in Computer Science, **736** (1993) 209–229.

Alur, R.; Henzinger, T. A.; Ho, P. -H. "Automatic symbolic verification of embedded systems," In *Proceedings of the 14th Annual Real-time Systems Symposium*, (1993) 2–11.

Arkin, E. M.; Chiang, Y. -J.; Held, M.; Mitchel, J. S. B.; Sacristan, V.; Skienna, S. S.; Yang, T. -C. "On minimal-area hulls," In *Proceedings of the 1996 European Symposium on Algorithms*, Lecture Notes in Computer Science, **136** (1996) 334–348.

Asarin, E.; Bournez, O.; Dang, T.; Maler, P. "Approximate reachability analysis of piecewise-linear hybrid systems," In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1790** (2000) 20–31.

Asarin, E.; Pace, G.; Schneider, G.; Yovine, S. "SPeeDI - a verification tool for polygonal hybrid systems," In *CAV'2002*. Lecture Notes in Computer Science, **2404** (2002) 354–358.

Atramentov, A.; LaValle, S. M. "Efficient nearest neighbor searching for motion planning," In *Proceedings IEEE International Conference on Robotics and Automation*, (2002) 632–637.

Barnsley, M. F. ; *Fractals Everywhere*, Academic Press, Inc. Boston, MA, 1988.

Bengtsson, J.; Larsen, K. G.; Larsson, F.; Pettersson, P.; Yi, W. "UPPAAL - a tool suite for the automatic verification of real-time systems," In *Proceedings of Hybrid Systems III*, Lecture Notes in Computer Science, **1066** (1996) 232–243.

Bjørner, N.; Browne, A.; Chang, E.; Colón, M.; Kapur, A.; Manna, Z.; Sipma, H. B.; Uribe T. E. "STeP: The stanford temporal prover, user's manual," Tech Rep. STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Nov. 1995.

Bozkaya, T.; Ozsoyoglu, Z. M. "Indexing large metric spaces for similarity search queries," In *ACM Transactions on Database Systems*, **24 (3)** (1999).

Branicky, M. S. *Studies in Hybrid Systems: Modeling, Analysis, and Control*, Sc.D. thesis, Electrical Engineering and Computer Science Dept., Massachusetts Institute of Technology, Cambridge, MA, June 1995.

Branicky, M. S.; Chhatpar, S. R. "Hybrid systems: Learning, planning, and control," In *Proceedings of the Eleventh Yale Workshop on Adaptive and Learning Systems*, (2001) 196–200.

Branicky, M. S.; Chhatpar, S. R. "A computational framework for the simulation, verification, and synthesis of force-guided robotic assembly strategies," In *Proceedings of the IEEE/RSJ International Conference for Intelligent Robots and Systems*, (2001) 1471–1476.

Branicky, M. S.; Curtiss, M. M. "Nonlinear and hybrid control via RRTs," In *Proceedings of the International Symposium on Mathematical Theory of Networks and Systems*, (2002).

Branicky, M. S.; Curtiss, M. M.; Levine, J.; Morgan, S. "RRTs for nonlinear, discrete, and hybrid planning and control," In *Proceedings of the IEEE Conference on Decision and Control*, (2003) Accepted.

Branicky, M. S.; Curtiss, M. M.; Levine, J. A.; Morgan, S. B. "Sampling-based planning and control," In *Proceedings of the Twelfth Yale Workshop on Adaptive and Learning Systems*, (2003) To appear.

Branicky, M. S.; Hebbar, R.; Zhang, G. "A fast marching algorithm for hybrid systems," In *Proceedings of the IEEE Conference on Decision and Control*, (1999) 4897–4902.

Branicky, M. S.; Johansen, T. A.; Peterson, I.; Frazzoli, E. "On-line techniques for behavioral programming," In *Proceedings of the IEEE Conference on Decision and Control*, (2000).

Chhatpar, S. R.; Branicky, M. S. "A hybrid systems approach to force-guided robotic assemblies," In *Proceedings of the IEEE International Symposium on Assembly and Task Planning*, (1999) 301–306.

Curtiss, M. M. *Motion Planning and Control using RRTs*, M.S. thesis, Electrical Engineering and Computer Science Dept, Case Western Reserve University, Cleveland, OH, May 2002. http://dora.cwru.edu/msb/pubs/mmcMS.pdf.

Dang, T.; Maler, O. "Reachabilitiy analysis via face lifting," In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1386** (1998) 96–109.

Daws, C.; Olivero, A.; Tripakis, S.; Yovine, S. "The tool KRONOS," In *Hybrid Systems III*, Lecture Notes in Computer Science, **1066** (1996) 208–219.

Frazzoli, E. *Robust Hybrid Control for Autonomous Vehicle Motion Planning*, Ph.D. thesis, Aero/Astro Dept., Massachusetts Institute of Technology, Cambridge, MA, June 2001.

Frazzoli, E.; Dahleh, M. A.; Feron, E. "A hybrid control architecture for aggressive maneuvering of autonomous helicopters," In *Proceedings of the IEEE Conference On Decision and Control*, (1999).

Frazzoli, E.; Dahleh, M. A.; Feron, E. "Robust hybrid control for autonomous vehicles motion planning," Technical report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, (1999).

Frazzoli, E.; Dahleh, M. A.; Feron, E. "Real-time motion planning for agile autonomous vehicles," In *AIAA Journal of Guidance, Control, and Dynamics*, **25(1)** (2002) 116–129.

Graham, R. L. "An efficient algorithm for determining the convex hull of a finite planar set," In *Information Processing Letters*, **1** (1972) 132–133.

Henzinger, T. A. "The theory of hybrid automata," In *Logic in Computer Science*, (1996) 278–292.

Henzinger, T. A.; Ho, P. -H.; Wong-Toi, H. "A user guide to HyTech," In *Tools and Algorithms for the Construction and Analysis of Systems*, (1995).

Henzinger, T. A.; Ho, P. -H.; Wong-Toi, H. "HyTech: the next generation," In *Proceedings of the 16th Annual Real-Time System Symposium*, (1995) 56–65.

Henzinger, T. A.; Ho, P. -H.; Wong-Toi, H. "HyTech: a model checker for hybrid systems," In *Software Tools for Technology Transfer*, **1(1)** (1997).

Henzinger, T. A.; Ho, P. -H.; Wong-Toi, H. "Algorithmic analysis of nonlinear hybrid systems," In *IEEE Transactions on Automatic Control*, **43(4)** (1998) 540–554.

Henzinger, T. A.; Horowitz, B.; Majumdar, R.; Wong-Toi, H. "Beyond HyTech: hybrid systems analysis using interval numerical methods," In Gautam Biswas and Sheila McIlraith, editors, *Proceedings of the AAAI 1999 Spring Symposium on Hybrid Systems and AI*, (1999).

Henzinger, T. A.; Kopke, P. W.; Puri, A.; Varaiya, P. "What's decidable about hybrid automata?," In *Proceedings Of 27th ACM Symposium on Theory of Computing*, (1995) 373–383.

Henzinger, T. A.; Wong-Toi, H. "Linear phase-portrait approximations for nonlinear hybrid systems," In *Hybrid Systems III*, Lecture Notes in Computer Science **1066** (1996) 377–388.

Kuffner, J. J.; LaValle, S. M. "RRT-Connect: an efficient approach to single-query path planning," In *Proceedings of the IEEE International Conference on Robotics and Automation*, (2000) 995–1001.

Kurzhanski, A. B.; Varaiya, P. "Ellipsodial techniques for reachability analysis," In *Proceedings of the Pittsburgh Conference on "Hybrid Systems - 2000"*, Lecture Notes in Computer Science, **170** (2000) 202–214.

Lafferriere, G.; Pappas, G. J.; Yovine, S. "A new class of decidable hybrid systems," In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1569** (1999) 137–151.

Larsen, K. G.; Pettersson, P.; Wang, Y. "UPPAAL in a nutshell," In *International Journal on Software Tools for Technology Transfer*, (1997).

LaValle, S. M. "Rapidly-exploring random trees: A new tool for path planning," TR 98-11, Computer Science Dept., Iowa State University, (1998).

LaValle, S. M. *et al*, Motion Strategy Library, http://msl.cs.uiuc.edu/msl/.

LaValle, S. M.; Kuffner, J. J. "Randomized kinodynamic planning," In *Proceedings of the IEEE International Conference On Robotics and Automation*, (1999) 473–479.

LaValle, S. M.; Kuffner, J. J. "Rapidly-exploring random trees: progress and prospects," In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, (2001) 293–308.

Manna, Z.; Sipma, H. B. "Deductive verification of hybrid systems using STeP," In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1386** (1998) 305–318.

Melhorn, K.; Näher, St. "The LEDA platform of combinatorial and geometric computing," Cambridge University Press, (1999).

Nicollin, X.; Olivero, A.; Sifakis, K.; Yovine, S. "An approach to the description and analysis of hybrid systems," In *Hybrid Systems I*, Lecture Notes in Computer Science, **736** (1993) 149–178.

Preußig, J.; Kowalewski, S.; Wong-Toi. H.; Henzinger, T. "An algorithm for the approximative analysis of rectangular automata," In *Proceedings of Formal Techniques for Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science **1486** (1998) 228–240.

Puri, A. *Theory of Hybrid Systems and Discrete Event Systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1995.

Puri, A.; Borkar, V.; Varaiya, P. "ϵ-approximation of differential inclusions," In *Proceedings of the 34th Conference on Decision and Control*, (1996).

Puri, A.; Varaiya, P. "Decidability of hybrid systems with rectangular differential inclusions," In *Computer-Aided Verification*, Lecture Notes in Computer Science, **818** (1994) 95–104.

Silva. B.; Krogh, B. "Modeling and verification of sampled-data hybrid systems," In *Proceedings of the 4th International Conference On Automation of Mixed Processes: Hybrid Dynamic Systems*, (2000) 237–242.

Silva, B.; Richeson, K.; Krogh, B.; Chutinan, A. "Modeling and verifying hybrid dynamic systems using checkmate," In *Proceedings of the 4th International Conference On Automation of Mixed Processes: Hybrid Dynamic Systems*, (2000) 323–328.

Silva, B. I.; Stursberg, O.; Krogh, B. H.; Engell, S. "An assessment of the current status of algorithmic approaches to the verification of hybrid systems," In *Proceedings of the 40th Conference on Decision and Control*, (2001).

Simić, S.; Johansson, K.; Sastry, S.; Lygeros, J. "Towards a geometry theory of hybrid systems," In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, **1790** (2000).

Tomlin, C. J. *Hybrid Control of Air Traffic Management Systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1998.

Uhlmann, J. K. "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters*, **40** (1991) 175–179.

Yiannilos, P. N. "Data structures and algorithms for nearest neighbor search in general metric spaces", *ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.