Toward More Efficient Motion Planning with Differential Constraints

by

Maciej Kalisiak

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2008 by Maciej Kalisiak

Abstract

Toward More Efficient Motion Planning with Differential Constraints

Maciej Kalisiak
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2008

Agents with differential constraints, although common in the real world, pose a particular difficulty for motion planning algorithms. Methods for solving such problems are still relatively slow and inefficient. In particular, current motion planners generally can neither "see" the world around them, nor generalize from experience. That is, their reliance on collision tests as the only means of sensing the environment yields a tactile, myopic perception of the world. Such short-sightedness greatly limits any potential for detection, learning, or reasoning about frequently encountered situations. In result these methods solve each problem in exactly the same way, whether the first or the hundredth time they attempt it, each time none the wiser. The key component of this thesis proposes a general approach for motion planning in which local sensory information, in conjunction with prior accumulated experience, are exploited to improve planner performance. The approach relies on learning viability models for the agent's "perceptual space", and the use thereof to direct planning effort. In addition, a method is presented for improving runtimes of the RRT motion planning algorithm in heavily constrained search-spaces, a common feature for agents with differential constraints. Finally, the thesis explores the use of viability models for maintaing safe operation of user-controlled agents, a related application which could be harnessed to yield additional, more "natural" experience data for further improving motion planning.

Dedication

To my son William, may he learn the lessons of will and persistence early in his life.

Acknowledgements

This dissertation would have never materialized without my wife Cathy, who always remained supportive and level-headed, an emotional ballast when the waters got choppy. Her patience, support, and handling of most day-to-day affairs were invaluable in seeing the thesis through. I am also grateful to my parents who have been a great help in juggling my own parenthood, and have in general aided in clearing any material impediments that otherwise stood in the way of completing the thesis.

A great many thanks also to Michiel van de Panne, my supervisor, who was an ever-ready font of great ideas, many of which lay the foundation for much of this thesis. I can only hope that his energetic drive and optimism have rubbed off on me. Likewise, I would like to thank the various committee members, and Professor Lydia Kavraki in particular; without their suggestions and ideas this dissertation would certainly suffer.

Finally, to plagiarise—ahem, paraphrase—an astute acknowledgement I have once seen (the source escapes me now), I would like to thank those who, at the outset, were kind to ask how things are going, and as the degree wore on, were kind not to.

Contents

1 Introduction **1**

1.1 Kinodynamic motion planning . . . 2

1.2 A sample kinodynamic problem . . . 3

1.3 Types of problems studied . . . 4

1.4 Current state of motion planning field . . . 5

1.5 Macro-primitives: a way forward? . . . 6

1.6 Contributions of thesis . . . 7

2 Previous Work **9**

2.1 Motion planning . . . 9

2.1.1 Real-world problems . . . 10

2.1.2 Relevance to Computer Graphics and Animation . . . 11

2.1.3 Useful distinctions . . . 11

2.1.4 Kinematic motion planning . . . 12

2.1.5 Kinodynamic motion planning . . . 20

2.1.6 General notes . . . 24

2.2 Viability . . . 28

2.2.1 Formal description . . . 30

2.2.2 Goal-viability . . . 31

2.2.3 Ambiguity of symbol x . . . 32

2.2.4 Related viability work . . . 32

2.2.5 Inevitable Collision States (ICS) . . . 32

2.3 User-control . . . 35

3 RRT-Blossom: sustained, non-redundant exploration **37**

3.0.1 RRT . . . 38

3.0.2 RRT-CT and “receding edges” . . . 39

3.1 RRT-Blossom . . . 41

3.1.1 Regression avoidance . . . 41

3.1.2 Node “blossoming” . . . 42
3.1.3 Bottleneck obstruction issue . . . 42
3.2 Experiments . . . 46
3.2.1 Agents . . . 46
3.2.2 Environments . . . 50
3.2.3 Test platform . . . 50
3.2.4 Results . . . 51
3.3 Discussion . . . 54
3.3.1 Benefits . . . 54
3.3.2 How improved operation is achieved . . . 58
3.3.3 Drawbacks . . . 59
3.3.4 Receding edge inclusion versus blossoming . . . 60
3.3.5 Control considerations . . . 61
3.4 Conclusion and future work . . . 62

4 Motion planning with local viability models 65
4.1 Filtering nonviable states . . . 66
4.2 Modeling viability . . . 66
4.2.1 Combined system state . . . 66
4.2.2 Sensors . . . 67
4.2.3 Locally situated state . . . 68
4.2.4 Local viability . . . 68
4.3 Acquiring training data . . . 70
4.4 Modeling . . . 70
4.4.1 Reachability viewpoint . . . 71
4.5 Exploiting viability . . . 71
4.6 Implementation details . . . 72
4.6.1 Training trajectories . . . 72
4.6.2 Deciding a sample’s viability . . . 72
4.6.3 Modeling . . . 73
4.7 Experiments . . . 73
4.7.1 Agents . . . 73
4.7.2 Environments . . . 75
4.7.3 Implementation details . . . 76
4.7.4 Learned models . . . 76
4.7.5 Performance evaluation . . . 79
4.8 Discussion . . . 81

4.8.1 Nonviable goal states 81
4.8.2 Choice of sensors 87
4.8.3 Oracle error 88
4.8.4 Oracle transferability 88
4.8.5 Scarcity of samples and bootstrapping 89
4.8.6 Expected reduction in planning effort 90
4.8.7 Completeness 91
4.9 Conclusion and future work 91
4.9.1 Automation 91
4.9.2 Reinforcement Learning 92
4.9.3 λ histories 93
4.9.4 Exploiting control action data 93
4.9.5 Extending applicability 93
4.9.6 Other 93

5 Safety enforcement with viability models 95

5.1 Introduction 95
5.1.1 Collision checking vs. viability checking 95
5.1.2 Theoretical framework vs. sample implementation 97
5.2 Framework 98
5.2.1 Single-step containment 98
5.2.2 Multi-step containment 99
5.3 Sample implementation 103
5.3.1 Modeling viability 104
5.3.2 Grace period 105
5.3.3 Threat level response 106
5.4 Experiments 107
5.4.1 Testing platform 108
5.4.2 Lunar lander 108
5.4.3 Bike 111
5.4.4 Car 114
5.4.5 Haptic feedback experiments 120
5.5 Discussion 122
5.5.1 Computational load and complexity 122
5.5.2 Viability model 123
5.5.3 “Constant u across T_h” assumption 125
5.5.4 Envelope margin of safety 126

5.5.5 Drawbacks and limitations . . . 127
5.6 Future work . . . 127

6 Conclusion and future work **129**
6.1 Motion planning with macro-primitives . . . 129
6.2 Exploiting human expertise . . . 131
6.2.1 Interactive motion planning . . . 131
6.2.2 Harnessing motion capture data . . . 132
6.3 Optimal planning . . . 134
6.4 Robust execution of offline plans . . . 135
6.5 Duality between motion planning and control . . . 136

Bibliography **138**

List of Tables

3.1 Results for holonomic point 52
3.2 Results for nonholonomic car 52
3.3 Results for kinodynamic bike 52

4.1 Performance comparison for **inertial point** 82
4.2 Performance comparison for **car** 82
4.3 Performance comparison for **bike** 82

5.1 Summary of relevant scenario parameters 108

List of Figures

1.1 A maze: one of the simplest motion planning problems. 2
1.2 A sample kinodynamic system 3
1.3 Problem space 7

2.1 Operation of a potential field planner 13
2.2 Operation of a potential field planner 14
2.3 Operation of Randomized Path Planner 15
2.4 Operation of a Probabilistic Roadmap planner 16
2.5 Steps of an RRT iteration 17
2.6 RRT vs. randomly grown tree 17
2.7 RRT commonly employs two trees 19
2.8 Seminal kinodynamic motion planner 21
2.9 Toy problem: "Lunar Lander" 29
2.10 The viable and reachable regions for the Lunar Lander 30

3.1 Example of RRT's poor performance - search trees 38
3.2 Sample histories of edge creation for RRTExtExt and RRT-CT 39
3.3 Empirically-derived "true cost to go" metric for car 40
3.4 "Receding edges" 41
3.5 Testing for regression 42
3.6 Interplay of viability and regression avoidance 44
3.7 FSM for node viability status 45
3.8 "Dormant deadlock" 45
3.9 The 8-way "holonomic point" agent. 46
3.10 The "nonholonomic car" agent. 49
3.11 The "kinodynamic bike" agent. 50
3.12 Test environments used 51
3.13 Boxplots of algorithm runtimes 53
3.14 Comparison of evolved tree structures 55

3.15 Comparison of typical edge creation histories for car agent. 56
3.16 Dual-tree RRT as a simple Finite State Machine 59

4.1 Virtual sensors for various agents 67
4.2 Judging viability using the local neighbourhood 69
4.3 Agent: inertial point 74
4.4 Environments used 75
4.5 Viability model for inertial point 77
4.6 Viability model for car 78
4.7 Viability model for bike 80
4.8 Boxplots of results for inertial point 83
4.9 Boxplots of results for car 84
4.10 Boxplots of results for bike 85
4.11 Visual comparison of tree density and structure 86
4.12 Detailed view of tree structure 87
4.13 Amount of viability filtering as a function of model error 89
4.14 Effect of excessive mismatch between training and test environments 90

5.1 A car constrained to stay on a track 96
5.2 Plot of desired and resultant steering angles for a car 97
5.3 A larger time horizon allows milder corrections. 100
5.4 Effect of using multiple look-ahead steps 100
5.5 Multi-step look-ahead tree structure 101
5.6 T_{eb} behaviour and viability of $\widehat{\mathcal{U}}$ for a fixed-velocity car 102
5.7 Threat levels 103
5.8 The lunar lander's viability envelope 105
5.9 "Grace period" 106
5.10 Lunar lander agent 108
5.11 Lunar lander envelope 109
5.12 Lunar lander altitude under safety enforcement 110
5.13 The effect of T_h on lunar lander trajectory 110
5.14 The effect of $|\widehat{\mathcal{U}}|$ on lunar lander trajectory 111
5.15 Bike envelope 113
5.16 An example of a bike ride with safety enforcement engaged. 113
5.17 The "track" environment. 115
5.19 Source of the "notch" in the envelopes 116
5.18 Car envelope for environment "track" 117
5.20 The "circles" environment. 118

5.21 Car envelope for environment “circles” 119
5.22 The “triangles” environment. 120
5.23 Car envelope for environment “triangles” 121
5.24 Minor obstacle motion can easily alter topology, viability. 123

Nomenclature

Below is a brief summary of common notation used throughout the thesis.

q	configuration of the agent
x	state of the agent; often $x = (q, \dot{q})$
$\vec{x}$	state of the agent, when x denotes distance along x-axis in surrounding text
e	state (and geometry) of the environment
x^+	combined system state; $x^+ = (x, e)$
$\mathcal{X}$	agent's state-space; $x \in \mathcal{X}$
$\mathcal{X}_{free}$	free-space; the set of states where agent is not in collision ($\mathcal{X}_{free} \subseteq \mathcal{X}$)
$\mathcal{X}_{viab}$	viable part of the state-space; $\mathcal{X}_{viab} = \{x \mid x \in Viab(\mathcal{X}_{free})\}$
$\mathcal{X}_{ric}$	"region of inevitable collision"; $\mathcal{X}_{ric} = \mathcal{X}_{free} \cap \neg\mathcal{X}_{viab}$;
$Viab(K)$	viability kernel (i.e., set of viable states) under constraint set K
σ	a virtual sensor
s	sensory state; $s = (\sigma_1, \sigma_2, \dots)$
λ	locally situated state
Λ	locally situated state-space; $\lambda \in \Lambda$
Ω_v	viability "oracle" (i.e., model)
u	a particular value for a control action
$\mathcal{U}$	set of all possible control actions
$\widehat{\mathcal{U}}$	a discrete subset of $\mathcal{U}$, usually sampled uniformly
$\mathcal{U}_v$	subset of control actions which maintain agent viability
v_k	control action selected by user at time-step k
T_h	time horizon; the duration of a look-ahead trajectory
$T_{eb}(x, u)$	time to (viability) envelope breach
T_{gr}	grace period
$F(x, u)$	system dynamics; in discrete-time case: $x_{k+1} = F(x_k, u_k)$ in continuous-time case: $\dot{x} = F(x, u)$

Chapter 1

Introduction

In simplest terms, *motion planning* is the problem of pathfinding, the piloting of an *agent*—a subject under our control, such as a car or a robot—from a given starting point to a destination. More formally, motion planning consists of discovering a trajectory for the agent from some initial state, x_{init}, to a goal state, x_{goal}, subject to the agent's laws of motion and any other applicable constraints, such as collision avoidance, balance, or joint limits.

Although pathfinding is a core application, motion planning has a far wider scope. For example, an incrementally more advanced application is the "Piano Mover's Problem", a canonical example in motion planning which generally deals with the problem of how to manoeuvre a piano through narrow corridors, stairwells and doorways, all the while avoiding walls and furniture, so that the piano is delivered to its destination unharmed. Motion planning also addresses problems such as how to parallel-park a car, or back a triple-trailer truck into a loading bay. Manipulation tasks, where the aim of the manipulator's (i.e., agent's) motion is to move an external object to a specified location and orientation, entail additional constraints, such as the avoidance of object-environment collisions, manipulator-environment collisions, and manipulator self-intersections. The general problem of planning, sequencing and coordinating movement of multiple agents also falls within the purview of this field.

With such a wide scope it is not surprising that motion planning has many diverse and far-reaching applications in the real world, and thus attracts considerable research attention. Aside from the most obvious applications to autonomous robot locomotion, robot control, robot-aided assembly, and the general problem of navigation, motion planning is also used in many virtual environments: computer games, computer generated animation in movies, and virtual prototyping. It even finds application in more remote domains, such as protein folding and drug design.

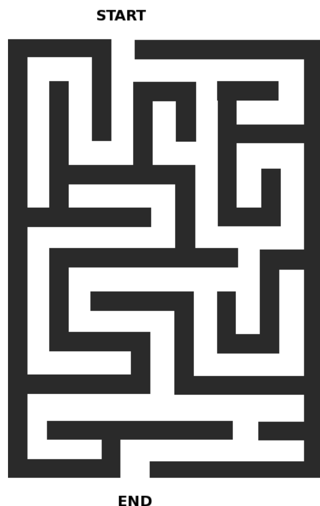


Figure 1.1: A maze: one of the simplest motion planning problems.

1.1 Kinodynamic motion planning

A more difficult class of motion planning problems involves agents whose equations of motion place restrictions, called *differential constraints*, on the agent's instantaneous displacement. The motions of a toy car and a typical bike, for example, exhibit differential constraints. The latter example, the bike, belongs to an even more challenging subclass of problems, called *kinodynamic* systems. Most typical motion planning problems are *kinematic*, in that they deal only with the agent's *configuration*—its position, orientation, and the internal arrangement of agent parts. Kinodynamic motion planning, on the other hand, deals with the agent's *state*, x, which consists of the agent's configuration, q, and its first time-derivative; that is, $x = (q, \dot{q})$. What makes kinodynamic planning difficult, aside from the larger search space[1], is the frequent abundance of deep "dead-ends" in this search space, regions that tend to ensnare the planner yet provide little prospect of advancing it toward its goal, much like local minima in gradient descent methods. These difficult regions are generally the result of the "cross product" of the agent's laws of motion with additional constraints imposed on the motion, such as the environment's geometry.

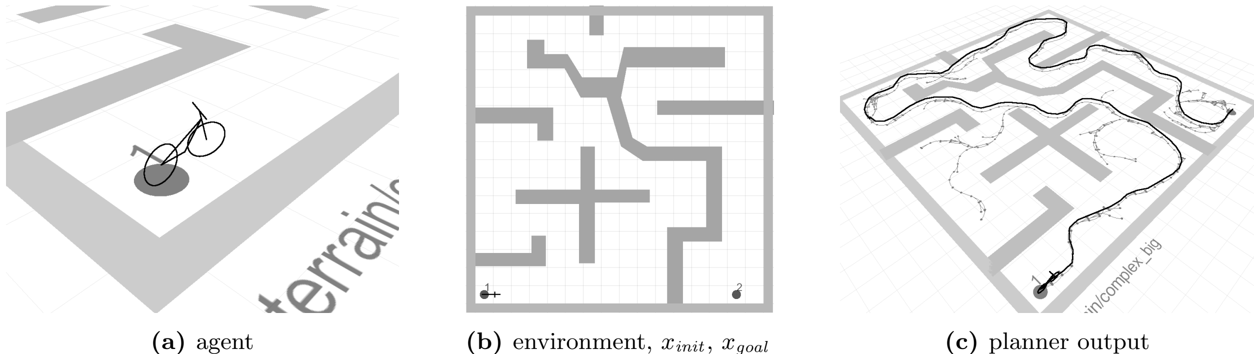
Kinodynamic systems are common in the real world. Some typical examples include driving a car on snow and ice, riding a bike, flying a helicopter, and even human motion, when it involves dynamic or acrobatic moves. Generally, any system in which inertia or dynamic balance play a significant role is kinodynamic, although the absence of these traits does not necessarily indicate the contrary.

[1]The search space has up to twice the number of dimensions compared to the kinematic-only case, when the full velocity vector $\dot{q}$ is employed.

1.2 A sample kinodynamic problem

Throughout this thesis we make repeated use of a particular kinodynamic system, namely the riding of a bike, to illustrate points or present results. The system is introduced here so that the reader is already familiar with it when it is employed. This also serves to illustrate the precise problem parameters being presented to a motion planner in a typical query.

In this example system the bike has a fixed forward velocity, and the only mode of agent control is the direct manipulation of the bike's steering angle, which is generally constrained to lie within some reasonable interval (e.g., $[-\frac{\pi}{4}, +\frac{\pi}{4}]$). What makes this system particularly difficult and interesting is the multi-purpose function of this one control input, which simultaneously acts to effect progress, avoid obstacles, and keep balance. A further hindrance is the counter-intuitive nature of bike motion, such as the execution of turns by first steering *away* from the intended direction of travel in order to set up and "lean into" the turn.



(a) agent **(b)** environment, x_{init}, x_{goal} **(c)** planner output

Figure 1.2: A sample kinodynamic system used throughout the thesis: a fixed-velocity bike. "1" and "2" indicate the locations of x_{init} and x_{goal}, respectively; at both these states the bike is fully upright, facing right, and has zero lean velocity.

Figure 1.2 illustrates a particular motion planning query for the bike. In general, a motion planning problem specifies:

- the starting state x_{init}
- the goal state x_{goal}
- the geometry of the environment
- a description of the agent's laws of motion

For actuated systems, the last is usually provided in the form of a function, $F(x, u)$, where x is the agent's current state, and u is the control action being applied (e.g., the bike's steering angle). In continuous-time systems the function computes the resultant derivative of the state vector:

$$\dot{x} = F(x, u). \tag{1.1}$$

Future states of the agent are thus obtained through integration of $F(x, u)$. In the case of a

discrete-time system, the function directly computes the subsequent state of the system:

$$x_{k+1} = F(x_k, u_k). \tag{1.2}$$

Also, for motion planners that do not perform any pre-computation step on the environment, a collision detection routine is usually provided in lieu of the complete description of the environment itself.

Given this description of the problem, the task of the motion planner is to find a trajectory from x_{init} to x_{goal} that conforms to the agent's laws of motion and all other imposed constraints. This trajectory is the planner's output, although for actuated systems the solution usually also includes the exact sequence of control actions that were applied to achieve the trajectory. The latter is useful when planning motion for physical robots, since this control sequence can then be fed to the agent, which then directly executes it to achieve the discovered solution trajectory.

1.3 Types of problems studied

There is much breadth and variety in the types of problems addressed by motion planning. In this thesis we are primarily interested in the types of problems one may encounter in applications of computer graphics and animation. In particular, this work has been motivated by an interest in automating the animation of secondary characters[2] and objects in various media, such as CG-enhanced movies, virtual worlds, and computer games. Currently, animation of such subjects is achieved using labour intensive methods, such as manual key-framing or motion capture sessions.

Most interesting animation subjects can be modeled adequately as kinodynamic systems, and hence this class is the primary focus of this thesis. Experiments have shown that the methods developed also work well with nonholonomic systems, hence the scope has been later widened to encompass systems with differential constraints in general. Furthermore, we have gravitated toward actuated agents, ones whose motion is controlled using *control inputs*, rather than through direct manipulation of the agent's degrees of freedom (DOFs), as this is slightly more general (e.g., a kinematic point can be recast as an actuated agent, where the the point's direction of motion and speed are the control inputs). Finally, most of this work has centred around first- and second-order systems with first-order control (e.g., integral control of car's velocity and steering), although in a few cases second-order control was used as well (e.g., the inertial point and the Lunar Lander).

[2] *Primary* characters—a protagonist of a story, for example—often need to portray subtle nuances and emotions, things which even a skilled CG animator might find challenging, let alone a fully automated tool.

1.4 Current state of motion planning field

Motion planning with kinematic agents, ones that are able to move along all DOFs independently—and this includes generic pathfinding—is largely a "solved" problem, and adequate methods exist to solve such queries. Unfortunately this is not the case with more complex agents, for example kinodynamic ones. Kinematic approaches either do not extend to these problems, or scale very poorly, becoming intractable for agents with even a handful of DOFs. Kinodynamic motion planning thus remains an active area of research, and is one of the key motivations behind this thesis.

There are naturally other challenges which motion planning research is actively pursuing, but as many of these have limited overlap with the work in this thesis, we only mention them in passing. For example, in the recent past there has been significant interest in the difficulty presented by the presence of narrow passages in the environment, which for many algorithms prove difficult to find and traverse. There has also been interest in systems with dynamic constraints and dynamic environments (we touch on these later), which present novel challenges. Manipulation planning (through grasping or using compliance) is also popular due to obvious industry applications. For the most part, many of these topics simply amount to a focus on a different (difficult) sub-class of motion planning problems.

A more basic and somewhat orthogonal issue that plagues motion planning, and either causes or at least exacerbates the above problems, is the *curse of dimensionality*: the search space—and thus also the runtime—grow exponentially with the agent's complexity, or more specifically, as the number of DOFs of the agent increases. What makes the problem acute is that many real-world problems can be highly dimensional. Although it is sometimes possible to reduce a problem to an approximation of sufficiently lower dimensionality, this is not always possible, as when a large number of the DOFs are each instrumental to finding a solution.

This is interesting because this behaviour seems sharply disproportionate to the human-perceived difficulty of the problems. Once sufficiently familiarized (i.e., after a period of experimentation and learning), humans are capable of controlling very sophisticated dynamical systems with only a moderate increase in planning time and effort. Furthermore, many motion planning problems are relatively underconstrained, in that the set of possible solutions is relatively large. Clearly finding an optimal solution, in the general case, requires an exhaustive search and hence warrants exponential growth. But optimality is frequently not required, and finding just a single solution should not be that difficult. To abuse the metaphor, when the number of needles is a linear function of the size of the haystack, the difficulty of finding a single needle should never grow out of control, no matter how big the haystack.

Part of the problem is the minute scale on which current methods operate. Tree-based planners solve problems essentially by brute-force, exploring the search space through incremental agent simulation, by applying fixed control actions over small time-steps dt. This is a convenient approach,

and perhaps even necessary for agents with system dynamics that are acutely sensitive to the agent state (e.g., many kinodynamic problems), and which therefore do not readily lend themselves to broad approximations or pre-processing. More importantly, this minute discretization of time is often also a necessary trait for the completeness proofs of the algorithms in question. Yet the dt-sized discretization of the problem is the main factor in the poor scaling of the methods. In a way, this represents a trade-off of speed for completeness guarantees. What if we chose a different balance in the trade-off?

The second issue is that most current methods are extremely inobservant, memoryless, and blind. That is, they neither "see" the world around them, nor learn from experience. In general, current motion planners rely solely on collision detection tests for sensing the surrounding environment, which leads to very tactile and myopic perception. As a result, such planners cannot detect, anticipate, learn, nor reason about commonly occurring patterns or scenarios, and instead end up solving all problem instances "from scratch" each time. For example, there is usually no substantial difference between the first and the hundredth time in how a typical planner solves a parallel parking problem for a car, or one that requires a three-point turn.

1.5 Macro-primitives: a way forward?

It seems clear that, in order to get a better handle on the curse of dimensionality, future planners will have to work on a scale larger than the current canonical dt time step. Since humans and other biological systems are capable of handling highly dimensional systems significantly better, it might be useful to try to imitate the mechanisms they use to do this. Recent research in neuroscience, such as [MIB00, MIS04], indicates that motion is realized through the sequencing and blending of motion primitives or control programs. This, in effect, acts as a means to reduce the dimensionality of the input problem, allowing the organism to successfully execute much more complex tasks.

In our context of motion planning, a similar effect could be achieved through the use of motion *macro primitives*, the building blocks of motion from which trajectories and solutions are constructed. These could be derived atomically or through chunking of lower operations. For example, in the case of the bike system introduced earlier, a simple macro-primitive could consist of a "turn" operation (likely parameterized by the turn radius), which would encapsulate the idea that one first needs to steer away from the desired direction of travel in order to setup a lean into the turn. More advanced primitives could then be built on top of this turning primitive, such as one for threading more smoothly between multiple obstacles, or for properly setting up the bike for narrow corridors which can only be entered from a particular direction. Even larger primitives could in turn be built on top of these, for example, for handling particular environment set pieces which might occur frequently.

A motion planner which works in terms of such macro-primitives would clearly scale significantly

better to more complex agents than current alternatives. As such, it makes an attractive research direction and goal. Alas, this has numerous open problems, and thus it is likely than any such general planner is still quite distant. In the next section, following the description of the key contributions of the thesis, a brief summary is presented of how the various contributions relate to this larger goal.

1.6 Contributions of thesis

The overall focus of this thesis is the enhancement of motion planning efficiency and speed. Even though the larger goal of a general motion planner capable of discovering and employing macro-primitives is still out of reach, this work also offers some ideas for a number of the open problems; these are discussed at end of this section.

The specific contributions of the thesis are as follows. In Chapter 3 we address the efficiency problem that a currently prominent motion planner has with more constrained environments, where it often wastes much time without making any progress. This problem becomes acute with more complex agents, especially kinodynamic ones. Viability plays a secondary role in this approach.

An even more substantial and general improvement is presented by Chapter 4, where local models of agent's viability are learned and exploited during motion planning to again reduce the amount of needlessly wasted effort. A *viable* state is one from which safe, collision-free agent operation can be maintained indefinitely; conversely, a *nonviable* state is one which will unavoidably lead to failure. For many complex agents each of the above improvements can often, on its own, reduce planning time by an order of magnitude or more.

Finally in Chapter 5 we look at using viability information to constrain user-control of agents, rather than planner exploration. This has many benefits, even for motion planning. Firstly, there are numerous dynamical systems where collision or failure carries prohibitive costs, and hence where

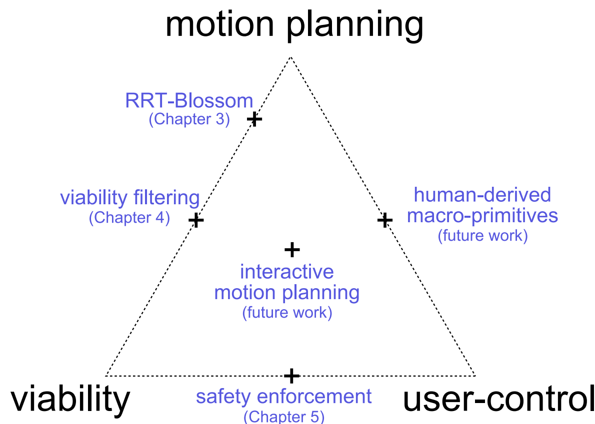


Figure 1.3: The problem space explored by the thesis.

such automatic safety enforcement is of value. More importantly, there is strong potential for a symbiotic relationship between motion planning and user control of the corresponding agent. For example, the lessons learned during motion planning (i.e., viability, other implicit characteristics of agent dynamics) can be exploited during computer-aided user control of the agent. At the same time, the user's control of the agent could be analyzed for common patterns and strategies which could then be exploited by the motion planner to yield faster and more "natural looking" solutions. Also, an interesting novel and completely unexplored research direction is the inclusion of a human during the planning process. This could be done for a number of reasons, such as human-aided computer motion planning, where the human's experience and intelligence is used to help with the particularly more difficult sections of the problem, or where the human is used to guide the solving process toward solutions of a particular topological or stylistic nature.

In terms of advancing toward motion planning with macro-primitives, the most notable is Chapter 4. Primarily, it demonstrates that using virtual sensors can be an effective way to capture the local context of the agent (i.e., local layout of the environment), a key requirement for representing macro-primitives so that they are environment neutral, and thus transferrable to environments other than in which they were learned. It also explores ideas on how learning can be employed to extract common motion primitives, using sequences or histories of such perceived local states. The other chapters relate on a more secondary level. Motion planning with macro-primitives will occasionally nonetheless need to resort to dt-sized exploration of particularly novel situations, hence a more robust planner is bound to still play a key role. The safety enforcement of Chapter 5, on the other hand, could significantly facilitate the acquisition of sample training data from human subject demonstrations, a topic explored further in Chapter 6 (Future Work).

Chapter 2

Previous Work

This chapter consists of three main parts, each providing an overview of previous work in the three key areas that most pertain to this thesis, as identified in the previous chapter (see Figure 1.3). Since the overarching goal of the thesis is faster, more efficient motion planning, previous work in this area is surveyed first, and more thoroughly. This is followed by a review of viability, which pervades every major chapter in this thesis. Finally, the last section gives only a cursory overview of user-control of an agent since this is a secondary consideration relative to the thesis' main goal.

It is also worth noting that each of the key chapters corresponds to an earlier publication. In particular, Chapters 3, 4, and 5 correspond respectively to [KvdP06], [KvdP07],and [KvdP04].

2.1 Motion planning

This section presents a brief review of motion planning. A thorough exposition of this topic can be found in a number of comprehensive references. [Lat91], one of the earlier noteworthy books, gives a thorough introduction to older methods, while [HA92a] provides a shorter survey. Their contemporary, [DW91] adopts a unified viewpoint between planning and control. In recent years a number of new texts have appeared, namely [CLH$^+$05] and [LaV06], which present an updated view of the field and discuss newer developments, such as sampling-based planners.

Motion planning, sometimes called *path planning* or *trajectory planning*[1], originates in the field of robotics. There is a distinct divide in motion planning research between what could be called "theoretical" vs. "real-world" approaches, a part of a larger trend in the robotics field (e.g., Field Robotics vs. work assuming "ideal conditions"), and to a lesser extent that in other fields (e.g., theoretical vs. applied science). A key difference between the two groups is one of objectives and criteria. Whereas theoretical motion planning approaches focus on issues such as path- or time-optimality and query speed, real-world problems often dictate greater focus on motion safety (i.e.,

[1]In this thesis "path planning" refers to kinematic problems, "trajectory planning" refers to kinodynamic ones, while "motion planning" is a general term representing both types of problems.

protecting the agent from the environment, and vice-versa), robustness to error, and are often subject to additional hard constraints. This disparity has led to the development of two bodies of knowledge that share little overlap, although recent research is starting to remedy this. Since the work in this thesis addresses issues mainly on the theoretical side of motion planning, the bulk of this section reflects this. We do however start off with a brief glimpse of the problems typically encountered in the real-world motion planning applications.

2.1.1 Real-world problems

Physical agents generally are subject to time pressures. It is dangerous for an agent to work out a full solution while remaining oblivious to its surroundings, especially in environments with moving obstacles. A logical way to deal with this is through Partial Motion Planning (PMP)[PF05], whereby time and agent operation is partitioned into slots, so that in time slot k the agent executes the partial plan derived in time slot $k-1$, and simultaneously plans motion to be executed in time slot $k+1$ (i.e., planning and the execution of corresponding plans are staggered by one time slot).

Another important trait of real-world problems is that one usually does not have full knowledge of the environment, but must instead work with a limited local model, obtained from agent-mounted sensors. A number of works address this problem, such as [FA04, BK07, BF95]. The "information space" approach ([BF95], and more recently [LaV06]) is particularly interesting. An *information space* $\mathcal{I}$ for a particular agent is the set of all its possible sensor readings i, where i is a tuple of length equal to the number of sensors. A noteworthy result in this area is that successful motion planning can often be achieved by working purely in $\mathcal{I}$, as opposed to using the sensory reading i to merely predict or estimate the agent's current state x, and then applying traditional $\mathcal{X}$-based motion planning methods.

A related problem is that the agent's exact state is often not fully known. A common source of uncertainty is the agent's less-than-perfect execution of prior motion plans, a consequence of using real-world sensors and actuators that are inherently subject to limitations and noise. Detecting and compensating for the resultant drift, between the expected and the achieved agent states, typically requires an accurate model of the surrounding environment, yet this often is not available. This is especially troublesome in applications where the agent's central purpose is to explore and map out the environment, since deriving an accurate model of the environment from agent sensor data usually assumes knowledge of exact agent state. Simultaneous Localization And Mapping (SLAM) [SC86, LDW91] research addresses this particularly troublesome chicken-and-egg problem.

Dynamic environments present a particular difficulty in real-world planning problems because, unlike under ideal conditions, the motions of the obstacles are not fully known. It is common in such cases to assume obstacle motion is linear (i.e., zero acceleration), but recent work has looked at predicting obstacle motion using statistical means [VF04].

2.1.2 Relevance to Computer Graphics and Animation

A long-standing goal of Computer Graphics and Animation (CGA), whether in film or interactive entertainment, has been the development of autonomous characters and objects, or at least ones which can be directed at a relatively abstract level (e.g., able to implement commands such as "go to garage", "get into car" or "drive to bank"). Motion planning is thus clearly relevant and of particular interest to CGA since such entities will need to independently solve various navigation and locomotion problems.

It is also worth noting that, in a large way, the task of animation is essentially a motion planning problem in disguise. Both problems consist of finding a suitable trajectory for the subject's state or configuration. Much like the motion planner, the animator must craft a trajectory that satisfies a number of constraints, such as the starting and the final pose, as well as a number critical intermediate waypoints or keyframes (e.g., establishing grasps, switching tasks, etc.) The only difference between the two is a disparity in constraints and objective functions: the animator generally focuses on the style and nuance of the motion whereas the motion planner is mainly concerned about avoiding collisions and, depending on application, finding the time-optimal solution. In fact, exploiting this apparent duality through borrowing and cross-applying ideas and techniques between these two fields seems like a promising future research direction.

Luckily, Computer Graphics and Animation are generally free from the real-world restrictions and the considerations discussed in the previous subsection. Neither safety nor complete realism are strictly necessary, and minor agent-obstacle interpenetrations are often permissible, aesthetic considerations aside. The main focus of this thesis will thus be on "pure" motion planning (i.e., under "ideal" conditions).

2.1.3 Useful distinctions

Before reviewing particular motion planning algorithms, it is worthwhile to first describe a number of categorizations which can be applied to them, to better understand their place, strengths, and applicability.

Kinematic vs. Kinodynamic

One of the most important distinctions in motion planning is between kinematic and kinodynamic methods. Kinematic planners work exclusively in terms of the agent's *configuration*, the set of values denoting the agent's position, orientation, and arrangement of internal parts. The symbol q is typically used to denote an agent's configuration, while $\mathcal{C}$ denotes the *configuration space*, the set of all possible configurations for an agent. Kinodynamic approaches, on the other hand, additionally take into consideration the first time derivative of the configuration, or some subset of it (i.e., kinodynamics = kinematics + dynamics). The combination of configuration and its time derivative

is called the agent's *state*, and denoted by x, while $\mathcal{X}$ is the agent's *state space* (also sometimes called *phase space*), the set of all possible states for the agent. Clearly kinematic approaches are simpler, but many problems that involve inertia or balance can only be addressed using kinodynamic methods.

Holonomic vs. Nonholonomic

Another important division is between holonomic and nonholonomic problems, and relates to the constraints placed on the system. In classical mechanics, *holonomic* constraints are ones that can be written in the form

$$f(x_1, x_2, x_3, \ldots, t) = 0. \tag{2.1}$$

That is, these are constraints which depend only on the agent's configuration parameters, and time. In contrast, *nonholonomic* constraints depend on additional quantities, such as agent velocity or momentum. Nonholonomic problems arise in the presence of rolling contacts (e.g., car) or velocity constraints (e.g., airplane). In general, nonholonomic systems are path-dependent, in that the agent's local configuration does not automatically determine the overall global configuration, but rather must be augmented with its full history. For example, knowing the net applied rotation to all the wheels of a car is insufficient to determine its final displacement; to do so one must also know in what sequence or combination the rotations were applied. Nonholonomic problems garner substantial attention due to the ubiquity of wheeled vehicles in every-day life.

Differential constraints

The above two classification schemes can be seen as special cases of a more general one. In particular, the kinodynamic and nonholonomic classes can be grouped together into problems with *differential constraints*. In essence, both classes deal with agents whose configurations may change through time only in a restricted manner: nonholonomic agents are limited by their nonholonomic constraints, whereas kinodynamic agents are limited by their velocity-dependent equations of motion. Such differential constraints problems are markedly more difficult, and are thus a focus of this thesis.

2.1.4 Kinematic motion planning

Kinematic problems were first to be studied, by virtue of being simpler, and thus have yielded a vastly more extensive body of research, to the point where the problem is being considered solved in general. Early work in motion planning generally revolved around three key approaches: roadmaps, cell decomposition, and potential fields.

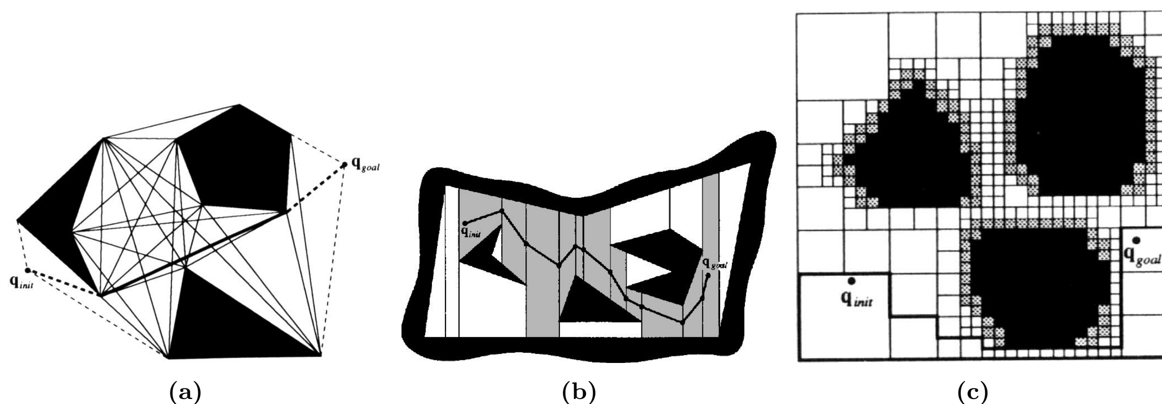


Figure 2.1: **left:** a visibility-graph roadmap; **middle:** exact cell decomposition (channel is shaded); **right:** approximate cell decomposition using a quadtree (channel is outlined with heavier line).
Figures adapted from [Lat91].

Early methods

The first planners used the notion of a *roadmap*. A roadmap captures the topology of the environment's free-space with a graph of canonical paths. Motion planning with roadmaps generally reduces to finding paths that connect the initial and goal configurations to the roadmap, and then finding the shortest path through the augmented graph. Numerous ways of generating such roadmaps have been studied. The seminal roadmap paper, [Nil69], used a visibility graph (also known as "shortest-path graph") of the agent's configuration space $\mathcal{C}$, where the graph nodes correspond to $\mathcal{C}$-obstacle corners, and edges are created between each pair of nodes which can "see" each other. The left diagram of Figure 2.1 shows a solved problem using this method. Roadmaps have seen a resurgence with the advent of probabilistic methods, and the Probabilistic Roadmap Planner (discussed later) remains in current usage.

A related approach is *cell decomposition* [Lat91, HA92a], where the agent's free-space $\mathcal{C}_{free}$ (i.e., subset of $\mathcal{C}$ in which agent is collision-free) is decomposed into a set of simpler fragments, termed *cells*. The connectivity of the cells is then captured using a graph, very similar to a roadmap. To plan a motion one simply finds a sequence of cells, called a *channel*, which connects the cell containing q_{init} to the one containing q_{goal}. The solution path is then obtained by plotting a path from q_{init} to q_{goal} by way of the midpoints of the cell walls that segment the channel. The middle diagram in Figure 2.1 demonstrates the approach.

Various cell shapes may be used, however they must make it easy to check for adjacency between two cells, find the portion of the cell boundary which two neighbouring cells share, and find a connecting path between any two points in a cell. These conditions ensure, respectively, the ease of constructing a connectivity graph of the cells, finding the midpoints, and constructing a continuous solution from the sequence of midpoints. Together they ensure the speed and robustness of the approach. In less regular environments, to comply with these constraints, approximate[Lat91] or probabilistic[Lin04] variants of the method are used.

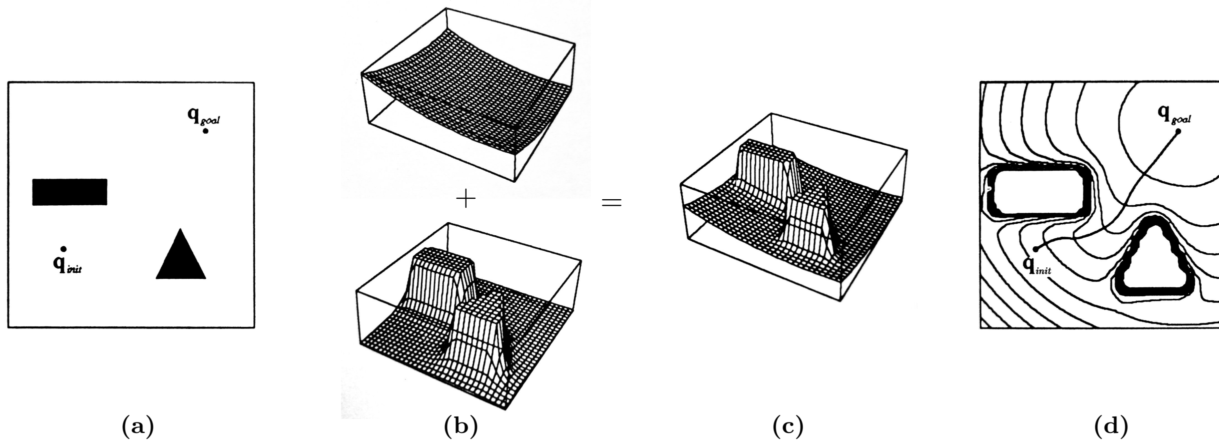


Figure 2.2: Potential field planner operation: **(a)** the query; **(b)** the q_{goal}-attracting potential (top) and obstacle-repulsing potential (bottom); **(c)** sum of potentials; **(d)** isoline plot of potential field, and gradient descent path. *Figures adapted from [Lat91].*

The last class of early planners employs *navigation functions* or *potential fields*, scalar functions erected over the search space that "lead" the agent toward the goal. These potential fields are usually the sum of a number of simpler fields; typically one field attracts the agent toward the goal, while the others repel it from individual obstacles. Solutions are obtained by performing gradient descent down the aggregate manifold. Figure 2.2 illustrates the key components of this approach.

A key weakness of such planners, typical of gradient descent methods, is that they frequently get trapped in local minima. The Randomized Path Planner (RPP)[BL91], a prominent algorithm at the time, proposes a number of mechanisms to mitigate this. Firstly, it uses specially constructed navigation functions, computed using a blend of medial axis and level-set methods, that guarantee absence of local minima for a kinematic point robot, and aim to reduce their incidence for more complex agents. Random-walks are used if the agent becomes trapped: that is, when the planner detects that a local minimum has been reached, the agent undergoes a series of random perturbations. Although there is no guarantee that any particular random-walk will escape the local minimum, the duration of the random-walks is chosen so that, based on the size and discretization of the workspace, escape is very likely. Should a random-walk fail to escape, the agent will return to the same local minimum and another escape attempt can be made. RPP operation is illustrated further in Figure 2.3.

An important feature of the RPP algorithm, one which allows it to produce good results for many problems, is the use of sampling to estimate the gradient of the potential field. Computing the gradient in high-dimensional spaces, whether analytically or through exhaustive inspection of a neighbourhood, is prohibitively expensive, in general; instead, a sufficiently accurate approximation is obtained by inspecting the potential values of a small set of randomly chosen neighbouring points. Such approximation-through-sampling allows the planner, in a way, to break the curse of dimensionality of the search space, and has since become a key component of most current

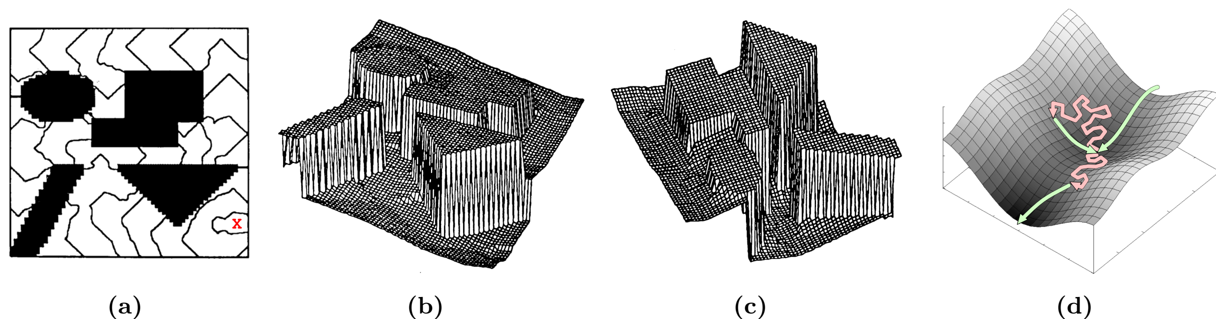


Figure 2.3: Randomized Path Planner (RPP) operation: the first step is computation of a "navigation function", using a combination of medial axis and level-set methods; (a) shows the function using isolines ("X" in lower right marks the goal), while (b) and (c) render it as a height-field from different angles. Planning consists of gradient descent down this manifold, and local minima are escaped using random-walks (d).
Figures (a)–(c) adapted from [Lat91].

planners (i.e., "sampling-based planners"). Interestingly, more recent work [LL03] argues that the randomness is not essential to the power of this technique, and that deterministic sampling strategies can be made to work equally well, thus avoiding the disadvantages of the former (e.g., non-repeatability, large variance in runtimes).

PRM

One popular sampling-based planner is the Probabilistic Roadmap method (PRM)[OS95]. This approach is usually limited to applications that perform multiple queries on a single static environment since a somewhat expensive pre-computation step is required. The pre-computation consists of first populating the workspace with *milestones*—a set of agent configurations chosen uniformly from $\mathcal{C}_{free}$—and then building a roadmap by connecting each milestone with a number of its closest neighbours, whenever this is possible without incurring a collision. For simple, holonomic agents straight lines are used to link the milestones, but in the general case the connections are attempted using an externally provided "local planner". Once this pre-processing step is done, queries are answered in the usual roadmap way: the initial and goal configurations are linked to the roadmap, the shortest path in the graph between the two points is found, and the solution is constructed by concatenating the trajectories corresponding to the edges in this path. Figure 2.4 illustrates the algorithm.

The key issues with PRM are the lack of robust local motion planners for most nontrivial dynamical systems, and the "narrow passage problem". The implicit premise behind PRM is one of "divide and conquer", where the planning task is explicitly split into global and local planning components, with the assumption that this results in two simpler subtasks. Alas, for more complex agents this is usually not true because the local planning problem is just as difficult as the global one. The "narrow passage problem", on the other hand, refers to the difficulty of capturing the topology of narrow passages with the roadmap, a side-effect of the stochastic milestone sampling.

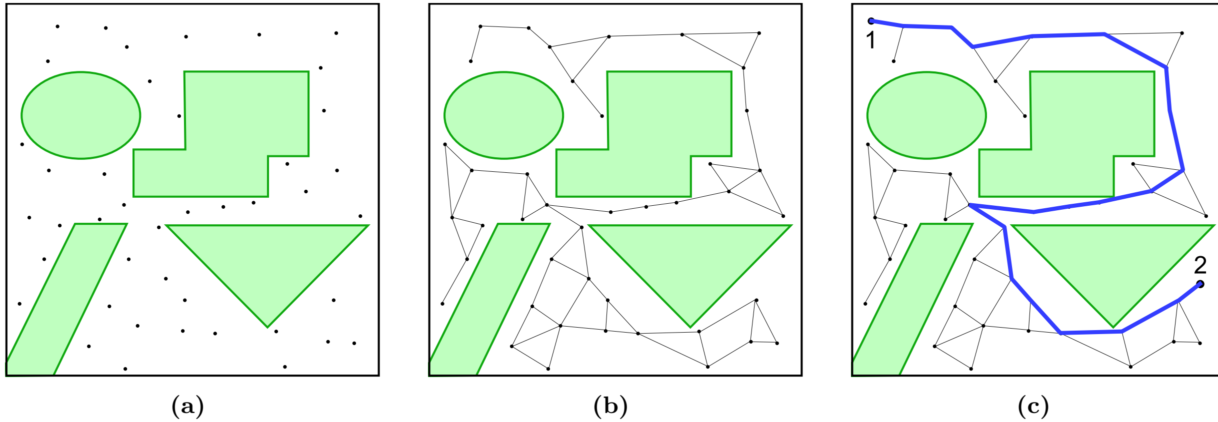


Figure 2.4: The Probabilistic Roadmap (PRM); (a) the environment is uniformly sampled with a set of random *milestones*; (b) each milestone is connected, where possible, with a number of its nearest neighbours, yielding a *roadmap*; (c) solution is found by connecting q_{init} ("1") and q_{goal} ("2") to the roadmap, and then finding the shortest path in the graph.

Even with a straight-line local planner a great many (random) milestones must be generated before a subset of them lines up in a way that allows the narrow passage to be "discovered" (i.e., traversed with a piecewise-linear curve).

RRT

The other predominant sampling-based approach to motion planning is that of the Rapidly-exploring Random Trees (RRT)[LaV98, LK00]. Rather than working globally like PRM, RRT works incrementally, building search trees with a strong bias for diffusion. Since RRT forms the basis of large portions of this thesis, a more extensive discussion of this algorithm and its variants follows.

The defining characteristic of RRTs, and the reason for the "rapidly-exploring" qualifier, is the tree's bias for growing toward unexplored space (see Figure 2.6). RRT's operation is best described using the most basic variant, where a single tree, rooted at the initial configuration q_{init}, is grown iteratively until a branch stumbles upon the goal configuration q_{goal}.[2] At each iteration the planner attempts to add a single edge to the tree. It first picks a random agent configuration, $q_{tgt} \in \mathcal{C}_{free}$, and identifies its nearest tree node, q_{near}.[3] The planner then attempts to form a collision-free edge that extends a short distance from q_{near} toward q_{tgt}. For holonomic agents (i.e., without any constraints on $\dot{q}$) the attempted edge extends directly toward the target q_{tgt}; for other agents the direction is determined indirectly: a small discrete subset of potential edges is selected (e.g., for actuated agents, a subset of possible control actions is applied over a small time-step), and from

[2]Or until the tree reaches some ϵ-neighbourhood of q_{goal}. In most tree-based planning approaches it is assumed that such minor discrepancies can be later corrected using suitable methods (e.g., "shooting method", trajectory deformation[LFV04], etc.)

[3]Note: we will often use the symbol q to refer to both, an agent configuration as well as the corresponding tree node. This node-configuration aliasing pervades the thesis, especially the pseudocode, and is extended to node-state aliasing when kinodynamic systems are considered.

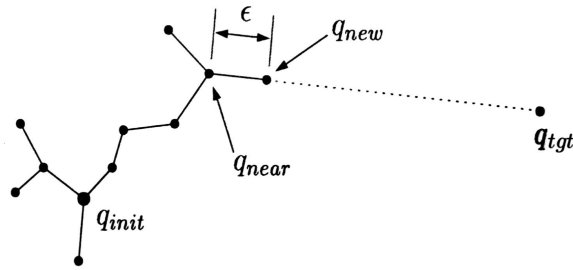


Figure 2.5: Steps of an RRT iteration: 1) randomly choose a target configuration, q_{tgt}; 2) find nearest tree node, q_{near}; 3) attempt to grow the tree from q_{near} toward q_{tgt}, for a single time-step ϵ, yielding new node at q_{new}.
Figure adapted from [KL00].

these the planner instantiates the candidate that advances the closest to q_{tgt} without incurring a collision. These steps are illustrated in Figure 2.5, while Algorithm 1 summarizes the planner with pseudocode.

The above basic scheme works well for quickly exploring the free-space, but finding a solution is relatively slow since the tree must blindly "stumble upon" q_{goal}. Hence a very common modification in single tree variants is to bias q_{tgt}. That is, in some portion of the iterations (e.g., 5%–10%) q_{tgt} is set to q_{goal} rather than to a random configuration, making the planner more goal-oriented.

RRT-Connect[KL00], another common variant typically used with holonomic agents, builds "maximal" tree edges. That is, once q_{near} has been found, the new tree edge is permitted to extend multiple time-steps, not just one, until it either hits an obstacle, reaches q_{tgt}, or reaches the point in its trajectory where it comes the closest to q_{tgt}. This serves to span large open spaces quickly in holonomic problems, but is often counter-productive when applied to most agents with differential constraints (e.g., leads to excessive weaving with cars, encourages loss of balance in bikes).

Despite these improvements, the single-tree RRT algorithm achieves only mediocre performance, and is thus rarely used. Much better results can be obtained by employing two trees[LK00]: the first rooted at q_{init} as before, and a second one at q_{goal}, which is grown in reverse-time (see Figure 2.7).

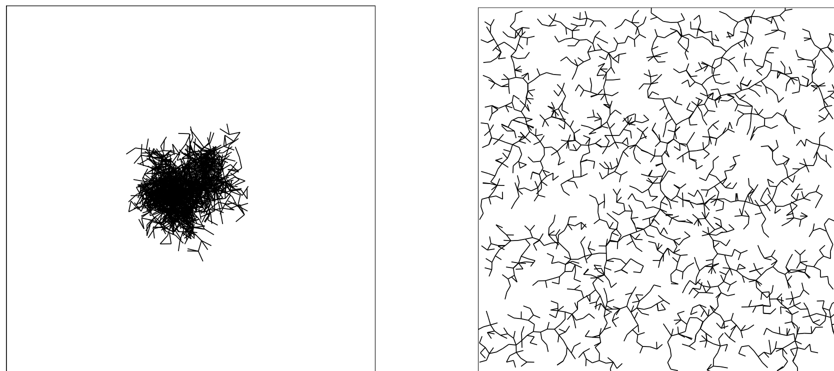


Figure 2.6: **left**: tree grown with new edge direction chosen uniformly over $(0, 2\pi)$; **right**: RRT, with each edge biased to grow toward unexplored space. Both trees have exactly 2000 nodes.
Figures taken from [LK99].

Algorithm 1 single-tree RRT for an actuated agent

```
 1: function QUERY(q_init, q_goal)
 2:     T ← tree(q_init)
 3:     while time_elapsed() < MAX_TIME do
 4:         q_tgt ← random_q()
 5:         q_new ← grow_tree(T, q_tgt)
 6:         if q_new ∧ ρ(q_new, q_goal) < ε then
 7:             return extract_soln(q_new)
 8:     return failed

 9: function GROW_TREE(T, q_tgt)
10:     q_near ← nearest_neighbour(T, q_tgt)
11:     u_best ← pick_ctrl(q_near, q_tgt)
12:     if u_best then
13:         T ← T + new_edge(q_near, u_best)
14:     return q_new

15: function PICK_CTRL(q, q_tgt)
16:     d_min, u_best ← ρ(q, q_tgt), ∅
17:     for u ∈ 𝒰 do
18:         q_new ← sim(q, u)
19:         if failure(q, u, q_new) then
20:             next u
21:         d ← ρ(q_new, q_tgt)
22:         if d < d_min then
23:             d_min, u_best ← d, u
24:     return u_best
```

where

- $\rho(x_1, x_2)$: distance metric
- `extract_soln(...)`: constructs solution by concatenating all edge trajectories on the graph path between q_{init} and q_{goal}.
- `new_edge`(x, u): create new edge from state x using control input u for a single (Extend) or maximal (Connect) number of time steps
- `failure`(x_1, u, x_2): test whether the transition from x_1 to x_2, using control input u, incurs a collision or violates other global constraints
- `sim`(x, u): compute state of agent after application of control input u from starting state x (paper assumes a constant time step)

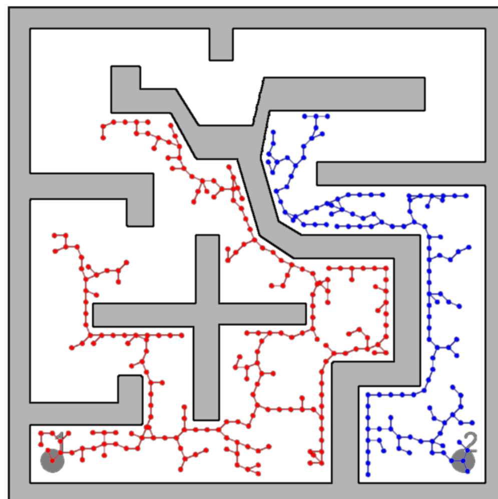


Figure 2.7: RRT commonly employs two trees, one rooted at x_{init}, and the other at x_{goal}.

Algorithm 2 dual-tree RRT (RRTExtExt, etc.)

```
1: function QUERY(q_init, q_goal)
2:     T_a, T_b ← tree(q_init), tree(q_goal)
3:     while time_elapsed() < MAX_TIME do
4:         q_tgt ← random_q()
5:         q_newA ← grow_tree(T_a, q_tgt)
6:         if q_newA then
7:             q_newB ← grow_tree(T_b, q_newA)
8:             if q_newB then
9:                 if ρ(q_newA, q_newB) < ϵ then
10:                    return extract_soln(q_newA , q_newB)
11:        T_b, T_a ← T_a, T_b
12:    return failed
```

(inherits `grow_tree()` & `pick_ctrl()` from single-tree RRT)

A solution is found when the two trees meet. The benefit of this approach is clear to see: as the trees grow, they provide an ever larger target for each other to connect to; the single tree, on the other hand, is always seeking a singular point (i.e., q_{goal}). Algorithm 2 outlines the operation of the dual-tree approach. In brief, on any given iteration one tree is treated in the same manner as before: a random q_{tgt} is chosen, q_{near} is found, and an edge is grown from q_{near} toward q_{tgt}. If this results in a new tree node, q_{new_A}, then the second tree is grown in a similar fashion, except that its target q_{tgt} is set to q_{new_A}. That is, the first tree is grown toward a randomly chosen q_{tgt}, yielding a new tree node q_{new_A}, while the second tree is grown toward q_{new_A}. At the end of each iteration the trees switch roles.

The RRT-Connect technique can be applied equally well to the dual-tree RRT algorithm, either to only one of the trees, or to both. This results in four possible variants, commonly named RRTConCon, RRTConExt, RRTExtCon, RRTExtExt. Here, “Con” (i.e., connect) refers to the RRT-Connect method of growing edges, while “Ext” (i.e., extend) refers to the simpler, single

time-step alternative. In each of the variant names, the first identifier pertains to the first, random-target-seeking part of the iteration, while the second identifier pertains to the second, tree-seeking part. RRTConCon is popular for holonomic agents, but in some cases RRTExtCon is preferred, as it emphasizes attempting to connect the trees over exploration, while RRTExtExt often makes more sense for agents with differential constraints.

The prospect of getting better performance through the use of even more trees has been explored in [Str04]. In this approach, whenever the planner encounters a q_{tgt} toward which extant trees cannot make any progress, the isolated configuration is used to seed a new, *local* tree, one which then takes equal part in the usual process of growth toward random targets and neighbouring trees. Trees are merged whenever they branches meet, while a connection between the primal, endpoint trees (i.e., the ones rooted at q_{init} and q_{goal}) signals the discovery of a solution. The benefit of the extra local trees is that difficult-to-reach areas of the free-space are explored earlier, from the inside out; this populates (with nodes) the ingress points to such poorly-accessible regions, thus increasing their exposure and likelihood of connection.

2.1.5 Kinodynamic motion planning

The first work to address kinodynamic motion planning, and in fact to coin the term, is [DX90, DXCR93], where (time) near-optimal trajectories are sought for a kinodynamic point mass with velocity and acceleration bounds. By limiting the agent's acceleration to $\{-a_{max}, 0, +a_{max}\}$ in each component, the method effectively discretizes the state-space into a regular grid (see Figure 2.8). A near-optimal trajectory is then found by computing the shortest path in the resulting graph. Although this approach has provably good time complexity and optimality, it suffers immensely from the curse of dimensionality, making it only applicable to simple agents with few degrees of freedom.

It was not until the advent of sampling-based planners that more complex agents could be handled adequately. Although these planners were introduced mainly for kinematic problems, refitting them for kinodynamic operation requires only a minor adjustment, namely the generalization of the search space to $\mathcal{X}$, the agent's state-space, in lieu of the purely kinematic configuration space $\mathcal{C}$. Alas, although useful solutions can be obtained with these algorithms, their runtimes can often be excessive (e.g., on the order of hours), even for relatively simple problems. For this reason there has been much attention recently in either optimizing these methods, or designing new algorithms specifically suited to kinodynamic motion planning.

The extension of the popular RRT algorithm to the kinodynamic domain is discussed in [LK99]. One important issue worth noting is that, with the transition from geometric paths to velocity-augmented agent trajectories, the trees are now necessarily directional. In particular, the x_{goal} tree must be grown using backward integration of the agent's dynamics (i.e., simulated using a

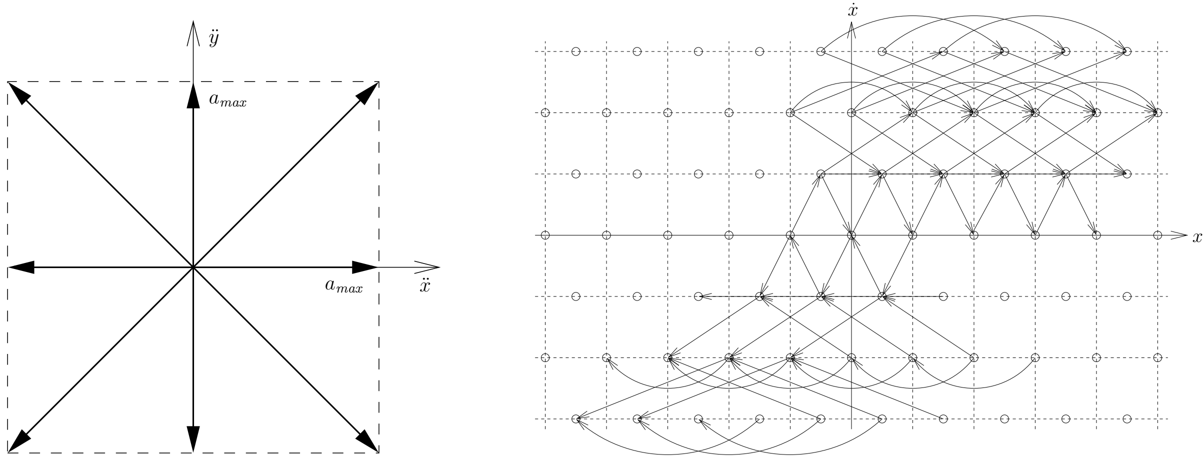


Figure 2.8: Seminal kinodynamic planner of [DX90, DXCR93]. **left:** the point agent is limited to a small set of acceleration vectors which are then applied for small, fixed durations τ. The use of such (a_{max}, τ)-*bang* motions results in a grid-like discretization of agent's state-space; **right:** cross-section of agent's state-space at $y = 0$. *Figures adapted from [DX90].*

negative time-step). This has significant repercussions, and even hinders the migration of some RRT extensions. For example, this presents problems for local trees[Str04]: since a single local tree can directly connect the x_{init} and x_{goal} trees, clearly it must consist of both, forward-time and reverse-time branches, but this leads to many unresolved questions and open problems with regards to implementation.

One of the key problems with RRTs is their sensitivity to the distance metric used to bias the exploration process: if the metric does not accurately reflect the true cost-to-go, as is often the case when the L_2 metric is used as a quick approximation for kinodynamic agents, RRT's performance suffers severely. [CL01b] proposes to mitigate this problem by introducing two key changes to the algorithm: 1) that the algorithm note which edge creation attempts result in collision, and subsequently disqualify them from future consideration; and 2) that the *collision tendency* of each node in the tree be tracked. We refer to this approach throughout the thesis as RRT with Collision Tendency (RRT-CT). Algorithm 3 outlines the planner in more detail. It is worth noting that the collision tendency value kept by the planner for a particular node is only a lower-bound: whenever an edge creation attempt incurs a collision, the parent node's collision tendency is suitably incremented, as well as that of all its ancestors, with the adjustment proportional to the edge's relative importance or "contribution" to each node. Thus a node's collision tendency asymptotically approaches its ultimate true value as the exploration of the subtree proceeds.

A number of alternative planners to RRT have also been proposed. [KHLR00, HKLR00, HKLR02] presents a PRM-inspired planner specifically designed for kinodynamic systems, initially dubbed *KDP*[HKLR00] but also referred to as Expansive Space Tree (EST) planner in later literature[LK05a, LK05b, PKV07]. It resembles RRT in that it is also a tree-oriented approach, but differs in its strategy for biasing exploration: rather than using a random target q_{tgt} to "attract"

new growth, the algorithm selects the node to grow (i.e., counterpart to RRT's q_{near}) based on the number of other nearby nodes, with preference given to candidates in more sparsely populated areas. An edge is then grown by picking a control action uniformly over the set of allowable control actions, $\mathcal{U}(q)$.

One drawback of the above planner, at least for actuated agents, is that proofs of probabilistic completeness for the algorithm assume uniform sampling of the agent's state-space $\mathcal{X}$, whereas most practical implementations sample uniformly only the agent's control space $\mathcal{U}$, which does not usually lead to the former. Frazzoli *et al*[FDF99, FFD00] address this issue with a new planner inspired by [KHLR00], although it is limited to agents capable of coming to a "stop", and further assumes knowledge of control policies that can bring the agent to a halt at a pre-specified location from an arbitrary starting state. The planner is used for real-time motion planning of a simulated autonomous helicopter among static and dynamic obstacles.

More recently, [LK05a] proposes the Path-Directed Subdivision Tree exploration planner (PDST-EXPLORE), a novel approach to tree-based planning, where the tree nodes represent motion segments rather than individual states, and branching can occur at an arbitrary point along an extant motion segment, rather than only at sampled states. The latter trait allows greater exploration freedom while keeping the number of tree nodes down. Like RRT, the planner iteratively grows additional tree edges, but in contrast uses a deterministic, greedy, metric-free node selection strategy that tends to focus tree growth to less populated areas and "younger" nodes. Steps are also taken to limit redundant exploration by enforcing density bounds on the creation of new nodes. PDST-EXPLORE aims to avoid the problems prior planners have with milestone placement, metrics, and coverage estimation. [LK05a] shows encouraging results for a kinodynamic point mass, a differential drive robot, and a blimp robot, while [LK05b] demonstrates the use of PDST-EXPLORE for motion planning in the game of Koules.

Finally, most recently [PKV07] insightfully notes that a key problem with existing tree-based kinodynamic planners is that they often get "stuck", and that this is due to their reliance on purely local information. A new algorithm is thus proposed, namely the Discrete Search Leading continuous eXploration (DSLX) planner, which directly addresses this weakness by guiding tree growth using global indicators. In particular, DSLX erects a coarse regular grid over the environment, and captures the adjacency of the resultant boxes using a weighted adjacency graph. The weights capture and reflect the likelihood of success (and thus desirability) of achieving the transition, and are influenced by node density, how much time has been already spent in attempting such a transition, and generally by the history of previous attempts. The adjacency graph is used to generate promising *leads*, favourably weighted "rough plans" which the underlying continuous tree then attempts to follow. The key benefit of this approach is that whenever the planner starts getting stuck, the graph weights automatically shift in response to reflect the fruitless directions, which in turn exposes more promising avenues for exploration that remain. The work goes on to show

Algorithm 3 RRT w/"Collision Tendency" (RRT-CT)

```
    (inherits query() & grow_tree() from single- or dual-tree RRT)
 1: function NEAREST_NEIGHBOUR(τ, x)
 2:     d_min, n_best ← ∞, ∅
 3:     for n ∈ τ do
 4:         if ∃ unexpanded input out of node n then
 5:             r ← random(),        r ∈ [0, 1]
 6:             if r > σ(n) then
 7:                 d ← ρ(n, x)
 8:                 if d < d_min then
 9:                     d_min, n_best ← d, n
10:     return n_best
11: function PICK_CTRL(x,x_tgt)
12:     d_min ← ∞
13:     for u ∈ U do
14:         if u has not been expanded for x then
15:             x_new ← sim(x,u)
16:             if failure(x,x_new) then
17:                 mark u as expanded
18:                 update_tendencies(x,τ)
19:             else
20:                 d ← ρ(x, x_new)
21:                 if d < d_min then
22:                     d_min, u_best ← d, u
23:     mark u_best as expanded
24:     return u_best
25: function UPDATE_TENDENCIES(x,τ)
26:     p ← 1
27:     while x do
28:         p ← p/||U||
29:         σ(x) ← σ(x) + p
30:         x ← parent(x)
```

where

- $\sigma(n)$: collision tendency of node n

significant runtime speedups with DSLX, sometimes of two orders of magnitude over the canonical, single-tree RRT algorithm.

2.1.6 General notes

Completeness & complexity

A *complete* planner is one that, in finite time, either returns a solution or (correctly) pronounces that no solution is possible. When designing a motion planner, or for that matter any algorithm for solving a challenging problem, it is typically desirable to construct it so that it is complete. Of course, it is even more desirable that an algorithm return its decision *quickly*, but typically it is very hard to characterize or bound an algorithm's rate of convergence to a solution. Thus the weaker condition of completeness is often the best achievable guarantee.

It turns out that motion planning is a difficult problem, and that complete motion planners do not scale well. Although runtime complexity can be expressed in terms of the dimensions of the environment, or the number and characteristics of the obstacles, the least scalable factor is typically the number of dimensions of the search space. In this respect motion planning has been shown by Reif [Rei79] to be PSPACE-hard, and later to be PSPACE-complete by Canny [Can88]. PSPACE denotes the class of problems which require polynomial space. More importantly, PSPACE $\supseteq$ NP. Reif's result in [Rei79] was demonstrated for the Generalized Mover's problem, a generalization of the Piano Mover's problem discussed earlier; clearly problems such as kinodynamic motion planning or planning amid moving obstacles are at least as hard, if not harder. More thorough review and discussion of the complexity of motion planning can be found, for example, in [LaV06]. Although specific subsets of motion planning problems may occasionally reduce to an easier class and admit more efficient algorithms, it is surprising how many "simpler"-looking problems are still complex. For example, even the game of Sokoban, which consists of moving solid tiles across a partially populated grid-like maze, has been shown to be PSPACE-hard[DZ99], and later PSPACE-complete[Cul99]. This PSPACE-completeness is very troublesome because many motion planning problems result in search spaces with surprisingly many dimensions.

One approach to overcoming this poor scalability is to partially abandon optimality. This mirrors similar strategies applied to solving other NP-hard problems, such as the Traveling Salesman Problem. Typically in such cases heuristics are used to instead arrive at approximate, near-optimal solutions. For example, near-optimal motion planners have been developed for kinematic points [CL93], curvature-bounded problems [JC89], and for acceleration bounded point robots [RW97]. Surveys of heuristic methods in motion planning can be found in [HA92b, FLS05, Eld01]. However, in general most such heuristic-based methods are of limited use as they are typically specific to a particular agent or agent class.

Worse still, for many problems (e.g., kinodynamic) it is even challenging to find a single feasible solution, let alone an optimal one. A pivotal point in motion planning research was therefore the advent of methods which traded-off completeness for improved scalability. By weakening the completeness guarantee, these methods were then able to solve far more complex problems. One example of this are the sampling-based algorithms, such as RRT, which ensure only *probabilistic completeness*. A planner is probabilistically complete when the probability of finding a solution approaches 1 as runtime approaches infinity. Other algorithms, on the other hand, ones which rely on some discretization of the search space (e.g., cell decomposition methods), instead provide *resolution completeness*, meaning that they are guaranteed to find a solution if one exists within the discretized image of the problem. Such approaches have the nice property that they allow the planner to first search for coarse solutions, before falling back to finer-granularity (but slower) searches. Although resolution completeness generally pertains to discretized, deterministic methods while probabilistic completeness is typically associated with stochastic sampling methods, it is worth noting that the two notions are closely related: a resolution complete method is usually also probabilistically complete since as runtime approaches infinity, ever-higher discretizations of the problem are used, in the limit arriving at the undiscretized problem, at which point they would necessarily find the solution due to the resolution completeness guarantee. Furthermore, the two classes also share the trait that they are unable to reliably detect when a problem simply does not have a solution, and will keep on searching indefinitely[4]. Implicit in their design is the assumption that a solution could always be “hiding around the next corner”, whether at a higher resolution or down a path not yet trodden.

Since the RRT algorithm plays a central role in this thesis, it is worthwhile to summarize its completeness results here. The basic RRT algorithm is shown to be probabilistically complete for kinematic systems in [KL00]. The crux of the proof lies in showing that as time progresses, the distribution of nodes in the RRT search tree approaches the distribution of the x_{tgt} states used to drive the tree growth; the latter are typically distributed uniformly over the free-space, hence this demonstrates that in the limit the search tree can explore any area with arbitrary density. Systems with differential constraints are more problematic. These constraints limit how the tree may grow, and thus in the general case the tree cannot match the distribution of x_{tgt} points. Instead, [CL02], [CL] and [Che05] extend the notion of resolution completeness to search-tree methods, with resolution pertaining here to discretizations in $\mathcal{X}, \mathcal{U}$, and time, and it is shown that the RRT algorithm can be made resolution complete with only minor alterations. The proof hinges on showing that, in the limit, the RRT search tree approaches the reachability graph (from x_{init}), within given tolerance parameters.

[4]Many implementations put in an artificial bound on runtime though, and instead err on the side of incorrectly reporting that a solution is impossible.

In defence of incomplete planners

Motion planning literature attaches much value to completeness guarantees of its algorithms, even when the guarantees are weak (e.g., probabilistic completeness). Clearly an algorithm that is guaranteed to be complete, even if in infinite time, is preferable to one without any guarantees whatsoever. However, from a practical standpoint, these affirmations are not as useful as one would hope. In particular, a guarantee of completeness does not say anything about the algorithm's speed or rate of convergence to a solution, while many applications are time constrained, and for them there is little difference between taking a very long time (e.g., a year) to find a solution, and not finding one at all. Hence a completeness guarantee has limited practical value in such cases.

In light of this, the pursuit of novel incomplete planners should not be disregarded out of hand. In fact, such planners can provide additional inroads in the battle against the inherent complexity of motion planning. Of particular value are motion planners that are incomplete but significantly faster than their complete counterparts. Any such planner can be made complete, or probabilistically complete, in a trivial manner: the incomplete planner should simply be run in parallel with a planner that supports the desired completeness guarantee. In cases where the incomplete planner would fail to terminate, the answer will be provided by the complete planner, while if the former does terminate, a solution will be typically available much faster. Clearly the combined planner will adhere to the desired completeness guarantee, and if the incomplete sub-planner terminates in a substantial portion of the runs, the average runtime will be significantly faster than that of the complete sub-planner alone.

In essence, this trivial "fix" mimics human behaviour, in that one often tries a number of simple or historically-proven potential solutions first, before resorting to more laborious, brute force methods. Also, in a way the fix embodies the canonical programming and hardware implementation principle of handling "the most common case first" (e.g., ordering within "if" and "switch" statements, virtual memory, caches, speculative execution, etc.) Finally, from a practical standpoint, this dual-planner approach is further supported by the current trend toward multi-core CPUs, which suggests such computing resources may soon be commonly available in embedded hardware and other physical implementation frameworks.

In light of this, limited effort has been made to make the methods presented in this thesis complete. Completeness guarantees could be obtained with further modifications, however it is likely that such changes would diminish the speed and power of said planners, while the above trivial "completeness fix" achieves an equivalent result. It bears pointing out though that, although this thesis takes a somewhat indifferent view of completeness, the pursuit of faster *complete* planners remains an important endeavour. Even in the above trivial "fix" a faster complete planner would result in material gains, in cases where the incomplete sub-planner fails to terminate. In essence,

the two classes ("complete" and "incomplete") complement each other, and the pursuit of each holds promise of improved motion planning.

Dimensionality of problems

A side-effect of the complexity of the general motion planning problem is that the dimensionality of the problems attempted in most literature may seem surprisingly low, at least when compared to the degrees of freedom (DOFs) typically encountered in Computer Graphics. This section provides a brief summary of the difficulty of problems attempted in the field.

In general, the majority of literature considers problems of at most 6D or 7D. For example, [LK00] considers holonomic motion of a piano, a Puma robotic arm, and the arm and hand of a virtual chess player, which respectively give rise to 3D, 6D, and 7D search-spaces. Nonholonomic agents included a car (3D) and car with 3 trailers (7D). The original PRM [OS95] works with planar articular robots (i.e., kinematic chains) in 7D configuration space, while subsequent work often focuses on kinematic motion for Puma arms (i.e., 6 DOFs). [HKLR00] addresses motion planning among moving obstacles for two joined nonholonomic carts (6+1D; last dimension is time), and for an air-cushioned disc (4+1D).

Occasionally the search-spaces reach 10D or 12D, but these usually involve fairly simple environments (i.e., sparse obstacles), and take much longer, typically in the range of 10–20 minutes. The kinodynamic subjects in [LK00] include a satellite docking amid a relatively sparse 3D field of medium-sized rectangular obstacles, yielding a 12D search-space, and solutions that required on the average over 8 minutes. [CL01b], on the other hand, considers lane-changing manoeuvres for a car with shock absorbers (i.e., model incorporates lateral car roll) as well as flying an under-actuated, rectangular, spacecraft-like agent out of a birdcage, yielding search-spaces of 9D and 12D, respectively, and average runtimes of 16.6 and 12 minutes.

Finally, a very few works consider even larger search-spaces. [KNK+01] and [JJKKN+02] look at motion planning for humanoid robots with 33 DOFs. This work takes a decoupled approach, common in kinodynamic methods prior to the seminal work [DXCR93], whereby the first stage works out a purely kinematic solution and a second stage then tries to implement it within the velocity constraints and dynamics of the robot. Although the obstacles in these problems were fairly trivial, typically consisting of two to three household furniture items, the runtimes varied in the range between 30 seconds to 11 minutes (on an SGI O2 / R12000), which is quite remarkable considering the 33D search-space. [SL01] on the other hand considers motion planning of 6 Puma arms in an car assembly setting, yielding a combined 36D search-space. Although this is a prodigious number of dimensions (for motion planning), the arms were used for spot welding and thus had minimal interaction with the car. The environment does not feature any other obstacles (i.e., only arm-car and arm-arm collisions had to be considered).

Boundary value problem

Many motion planning algorithms rely on the assumption that it is relatively easy to find motions for an agent—or the underlying control inputs, if the agent is actuated—that bring it to a particular, exact state or configuration. For example, the PRM algorithm assumes that its local sub-planner can work out a suitable trajectory to connect any two milestones (while ignoring the presence of obstacles, of course). Unfortunately for many agents, especially kinodynamic ones, solving this Boundary Value Problem (BVP) is quite difficult. This thus limits the applicability of these methods.

Under certain conditions (e.g., actuated agents with discrete controls), the BVP also surfaces in sampling-based planners. In single tree variants, the final branch of the tree must hit x_{goal} exactly. Since in most problems the neighbourhood around x_{goal} is an equally acceptable stopping point, the goal is often defined using a *region* of the search space. In fact, motion planning is typically stopped once a branch of the tree comes to within some tolerance of x_{goal}, which thus implicitly defines the region. In dual tree variants the BVP lies in connecting the two trees. The trees must meet in order for a solution to be found, yet in the case of many agents it is rare (or takes a very long time) for a node of one tree to coincide *exactly* with a node of the other. To remedy this, motion planning is again typically stopped once the trees approach each other to within some tolerance. If the remaining gap is small and the problem sufficiently forgiving (e.g., we are only interested in the rough shape of the solution because, for example, it is to be refined by a further stage), the gap can be ignored. Otherwise it must be eliminated. There has been some recent interest in such trajectory deformation, for example in [LFV04, CFL03].

The planners of Chapters 3 and 4 do contend with the BVP. In this work we ignore the gap between the two trees, making sure first that the connection tolerance is chosen small enough to be amenable to any trajectory deformation tools, should they be available, and so that the visual discontinuity of the solution is tolerable. It is worth noting that these planners could also be regressed into a single-tree variant form, in which case the typical "goal region" solution would be equally effective.

2.2 Viability

The concept of viability pervades this thesis. In plain terms, viability describes whether a dynamical system's operation is sustainable from a given starting point, whether the system can be kept from failure. In the context of controllable processes, a *viable* system state is guaranteed to have at least one corresponding sequence of control actions which, when applied from said state, will avert failure (i.e., "there is a way out", an "evasive action" exists). Conversely, a *nonviable* state is one in which the system has gone "past the point of no return", where failure is no longer avoidable (but

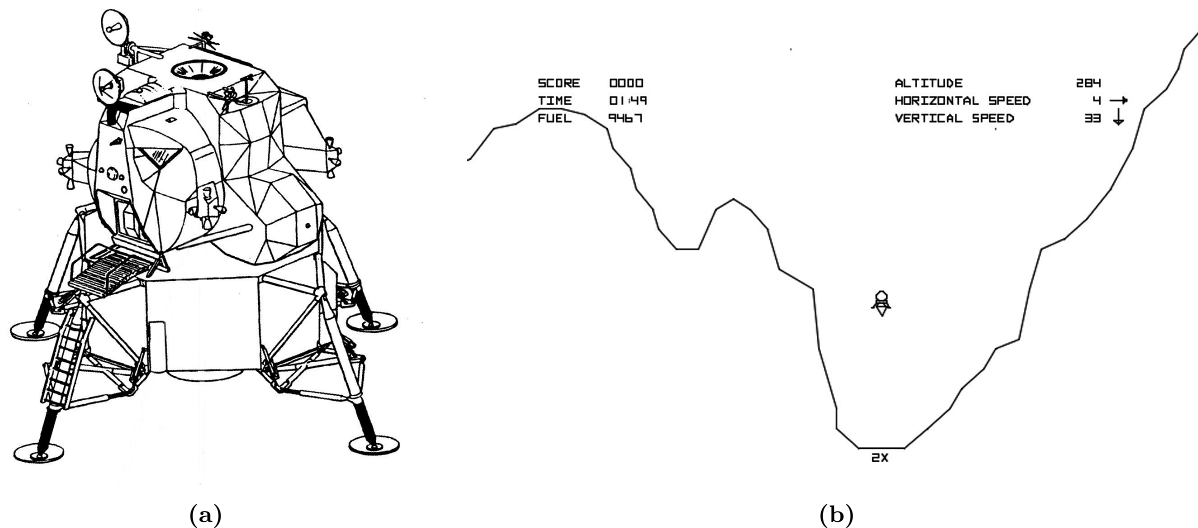


Figure 2.9: Toy problem for viability; **(a)** The Apollo Lunar Module; **(b)** the 1970s arcade game "Lunar Lander"

perhaps postponable for some finite time). It is important to note that viability merely indicates the existence of such sequences, but makes no claim as to their number—there could always be *only one* in any given case—nor does it help in discerning what the sequence is.

Reachability is a concept closely related to viability. Reachability plays an important role in safety analysis and similar fields, where safety guarantees are arrived at by showing that it is impossible for the dynamical system to reach a particular dangerous state. Pursuit-evasion games are a classical problem in reachability (e.g., the "homicidal chauffeur" problem). A thorough treatment of reachability is provided, for example, by [Mit02]. Since reachability plays a minor role in the thesis, only a minimal introduction is given here. We will refer to states as *reachable* if they can be reached from some canonical initial state, usually x_{init}, the starting point of a query. In the more formal terminology of the field, these reachable states comprise the *forward reachable set* of x_{init}.

These concepts can be best illustrated with a simple toy example. "Lunar lander" is the name of a popular video arcade game from the late 1970s (see Figure 2.9), a simple, 2D simulation of landing an Apollo Lunar Module craft on the moon. In this example we further simplify things by making the lunar surface an infinite plane, and by eliminating craft rotation and lateral motion, thus limiting the craft to one-dimensional displacement along the vertical axis. The object of this exercise then is to softly land on the moon surface; that is, to reach the state $(z, \dot{z}) = (0, 0)$ while ensuring that at all times $z \geq 0$, where z is the altitude, and subject to the additional constraint that the thruster, and thus the agent's acceleration, is bounded.

This last constraint has two important consequences: 1) if the lander's downward velocity grows too large, the limited amount of thrust will be unable to sufficiently decelerate the agent before it reaches the surface, leading to a crash; and 2) the lander cannot reach a substantial set of states

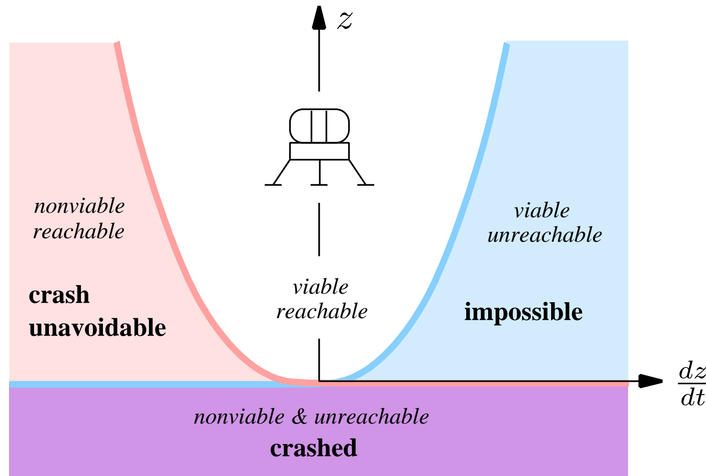


Figure 2.10: The viable and reachable regions of the lunar lander's state-space. The "crash unavoidable" region is nonviable since the downward velocity exceeds the braking power of the lander's (bounded) thrust. The "impossible" region is unreachable since the upward velocity exceeds what can be physically achieved, even if maximal thrust is applied from altitude $z = 0$.

since its limited thrust puts an upper bound on the achievable upward velocities for a given altitude (and the agent cannot descend below $z = 0$ to "wind up"). These two observations pertain to the concepts of viability and reachability. Figure 2.10 illustrates the lunar lander's state-space and its partitioning by these classifications.

It is worth pointing out that the work in this thesis does not make any use of theorems or results obtained in viability theory or reachability, other than the rudimentary term definitions, but merely makes heavy use of the concepts which they study. It is possible though that the theory could come into greater play in future work.

2.2.1 Formal description

In its most general form, viability theory makes statements about *evolutionary systems*, which are essentially nondeterministic dynamical systems resembling Markov processes, in that for each system state there exist a number of possible transitions, to be decided among through stochastic or simply unspecified means. The latter provision merely makes the theory more general; it is also applicable to systems where the transition policy may be deterministic (but perhaps hidden), or where the transitions are decided by an unpredictable agent (e.g., a human). In particular, the theory is applicable to controllable or actuated systems since they too permit multiple evolutions out of any particular system state, with the control variable $u(t)$ determining which one is taken.

Viability can only be assessed relative to some constraint. One generally defines a set $K \subset X$ of admissible states, called the *viability constraint set*. These are the states the system is allowed to assume. In motion planning and user-control problems, the most obvious choice is to set K to the agent's free-space; that is, $K = \mathcal{C}_{free}$ (kinematic problem) or $K = \mathcal{X}_{free}$ (kinodynamic problem). For example, for the kinodynamic bike K would be the set of states in which the bike is not in

collision with any obstacles, and has not fallen over.

Viability can be formalized as follows. Let the dynamical system be modeled by

$$\dot{x}(t) = F\big(x(t),\, u(t)\big), \tag{2.2}$$

$$u(t) \in U(t) \quad \text{where} \quad U(t) = g\big(x(t)\big). \tag{2.3}$$

Here, $x(t)$ is the system state at time t, $u(t)$ is the control action applied, and $U(t)$ is the set of allowable control actions. In the general case the set of allowable actions depends on the agent's state x, but in this thesis U is fixed for each agent. In this framework, a state is viable if there exists a sequence of control actions that allows the system to stay *forever* within the admissible region K. That is, x_0 is a viable state if

$$x(0) = x_0, \quad \text{and} \tag{2.4}$$

$$\exists u(t) \quad \text{such that} \quad \forall t \geq 0 \;\; x(t) \in K. \tag{2.5}$$

The *viability kernel*, denoted by $Viab(K)$, is then the set of all viable states under the constraint K:

$$Viab(K) = \big\{x_0 \;\mid\; x(0) = x_0 \;\;\wedge\;\; \exists u(t) \;\text{ such that }\; \forall t \geq 0 \;\; x(t) \in K\big\} \tag{2.6}$$

Thus $Viab(K) \subseteq K \subset X$.

The above definition can be easily reworked for discrete-time dynamical systems, in which

$$x_{k+1} = F(x_k, u_k), \tag{2.7}$$

$$u_k \in U_k \quad \text{where} \quad U_k = g(x_k). \tag{2.8}$$

The variables x_k, u_k, and U_k are the corresponding system state, control action applied, and the set of allowable actions at the kth time-step, respectively. The viability kernel then becomes

$$Viab(K) = \big\{x_0 \;\mid\; \exists\{u_0, u_1, u_2, \dots\} \;\text{ such that }\; \forall k \in \mathbb{N} \;\; x_k \in K\big\}. \tag{2.9}$$

2.2.2 Goal-viability

When discussing viability in the context of a motion planning problem, we have found it is often useful to use a slightly wider definition. In particular, we pronounce a state x to be *goal-viable* if x is viable by the standard definition, or if x_{goal} can be reached from x before the agent fails (this implies x_{goal} lies in nonviable space). Effectively, this concept allows us to cleanly separate those agent states which could lead to a solution, and those that surely will not. In the remainder of this thesis,"viable" and related terms will refer to this wider definition, unless explicitly noted otherwise.

2.2.3 Ambiguity of symbol x

We note an ambiguity of notation due to two traditional meanings for the symbol x; on the one hand it is the canonical variable for measuring distance along the x-axis, but also it is the traditional symbol used for agent state in literature, especially that of motion planning. Thus whenever the two meanings are used together, such as in Section 2.2.5, the agent state is written in vector notation ($\vec{x}$), while the distance along x-axis is written in plain scalar form (x); otherwise, the intended meaning of x should be clear from context, and almost always refers to the agent's state.

2.2.4 Related viability work

The above presents only the basic definitions. The most thorough and authoritative source on Viability Theory is the book[Aub91] by Jean-Pierre Aubin, while more concise and introductory overviews can be found in [Aub90, ASP04, Aub02b, Aub02a]. Viability theory goes on to prove a number of notable results. For example, it shows that all interesting features such as equilibria, trajectories of periodic solutions, limit sets and attractors, if any, are all contained in the viability kernel.

Alas, the theory does not, in general, provide explicit ways to compute the viability kernels, a matter of most immediate interest and import to the research presented here, although a number of other works have investigated this topic. Saint-Pierre[SP94] approaches the problem by discretizing the state-space $\mathcal{X}$ and control-space $\mathcal{U}$, and then in an iterative manner akin to cellular automata solving a maze, refines the discrete approximation by discarding states for which all the control actions in the discrete set lead to known nonviable space, until a steady-state is reached. The method naturally suffers from the curse of dimensionality, with exponential growth in samples as the dimensionality of $\mathcal{X}$ or $\mathcal{U}$ grows. [CD06], on the other hand, attempts to model the viability kernel with Support Vector Machines (SVMs). Roughly, this method learns a (continuous-space) SVM model in parallel with each iteration of the Saint-Pierre's algorithm, and then performs some additional refinements. The goal of this approach is to get an analytical approximation of the kernel, which can mitigate at least the curse of dimensionality in the control space $\mathcal{U}$. Another recent approach[BMMZ04] models the viability kernel using a value function of an optimal control problem, and proposes the use of *Ultra-Bee scheme* to mitigate the approximation's numerical diffusion due to the discontinuities of the function. Although the work shows good empirical results, Ultra-Bee scheme currently has no convergence proofs, and is currently limited only to 2D state-spaces.

2.2.5 Inevitable Collision States (ICS)

Recent motion planning literature has seen the introduction of the concept of *Inevitable Collision States* (ICS)[FA04]. An inevitable collision state for an actuated agent, as the name suggests, is

one where all possible future control action sequences (termed “control inputs” in [FA04]) lead to collision. Clearly this concept is strongly related to viability; in fact, a comparison of definitions shows that an ICS is equivalent to a nonviable state. That is,

$$x \in \text{ICS} \quad \textbf{iff} \quad x \notin \mathit{Viab}(\mathcal{X}_{\mathit{free}}).$$

[FA04] introduces the ICS concept, provides the basic definitions, and outlines a number of properties of ICS useful in its construction. [PF05] provides an extended ICS case study in the context of a velocity-controlled nonholonomic car, in addition to introducing the concept of Partial Motion Planning (PMP). The ICS computation is further extended in [PF07] to also support mobile obstacles through the use of “imitating maneuvers” (IM). The details of computing the ICS using IMs are further thoroughly detailed in a recent thesis [Par06]. [Fra07] argues for the superiority of ICS method in ensuring agent safety, relative to other approaches currently used in physical implementations. Finally, [BK07] combines a number of recent ideas and approaches, including ICS, in order to demonstrate fast kinodynamic motion planning for a nonholonomic car in partially observable terrains, although this implementation uses finite look-aheads and thus in the general case cannot ensure safety.

In comparing the ICS work to ours it is important to first note the difference in end goals of both approaches. The ICS approach was developed with real-world, physical implementations in mind, where the agent is subject to (live) partial motion planning. As such, the safety of the agent (and of the environment; e.g., human “obstacles”) is the primary concern, and in particular the need to guarantee avoidance of situations which would lead to unavoidable collision. This necessitates erring on the safe side, and hence leads to conservative models of ICS (i.e., model will yield ICS false positives). The work in this thesis, on the other hand, does not need to abide by this constraint, and can thus explore less rigourous approaches. In particular, the filtering of nonviable states during motion planning, from Chapter 4, can safely generate false negatives since the collision-checking routines in the planner will catch such errors, while the viability envelopes presented in Chapter 5, although also attempting to guard the agent from unavoidable collision states, are intended rather for observation of human control (i.e., virtual simulation) and general human-directed animation, where error is not a grave concern either. Because of this divergence of goals, each method is most suitable to and performs best in the application it was intended for, and their cross-application would yield degraded performance.

Overall, the ICS approach hinges on deriving a conservative model of $\text{ICO}(\mathcal{B})$, the *Inevitable Collision Obstacle* for the given environment. This is a boolean function over the agent's state-space that computes the “cross product” of the obstacles $\mathcal{B}$ and the agent's dynamics. That is, it captures the nonviable region: a state $\vec{x}$ is an ICS if, and only if $\vec{x} \in \text{ICO}(\mathcal{B})$. The model is captured exactly using geometric primitives. Deriving such exact representations in arbitrary

spaces is generally expensive, hence a key step in the approach is the reduction of the problem to a 2D slice of the space. Specifically, in [PF05, PF07, Par06], a car state $\vec{x} = (x, y, \theta, v)$ is tested for ICS by first deriving the explicit representation of the 2D ICO($\mathcal{B}$) slice that spans the agent's positional variables (x, y); the remaining state variables, orientation and velocity, are fixed at the values they have in $\vec{x}$. The slice is computed by first determining all possible ICO($\mathcal{B}_i, \phi_j$), the "cross products" of a particular obstacle and a particular evasive motion. These are then combined to yield ICO($\mathcal{B}$). The ICS check itself consists then of simply checking the value of the ICO($\mathcal{B}$) slice at (x, y). It is important to note that the shapes of all these Inevitable Collision Objects are captured *exactly* using generalized polygons, and then combined together using set operations. Such 2D polygon computation and manipulation (Minkowski Sums, unions, intersections) can be done efficiently and quickly, thus allowing the method to run at speeds required by PMP.

There are however a number of limitations to this approach. One of the key questions is how well this method will generalize. To date it has only been demonstrated with a car agent in relatively spacious environments. How will it hold up in higher-dimensional spaces? Can the 2D slice problem reduction technique always be applied? In particular, when migrating to a 3D workspace it would seem that the slice would now have to be 3D, spanning (x, y, z), or is the choice of slice dimensions arbitrary? Clearly computing exact ICO($\mathcal{B}$) representations in 3D would be vastly more expensive. The ICS method is also currently constrained to mobile obstacles whose motion the agent can fully imitate. In the demonstrations thus far the mobile obstacles were replicas of the agent itself; holonomic "human" obstacles (especially "suicidal pedestrians"), for example, cannot be handled. It seems doubtful that any future developments can amend the method to provide safety for fully general obstacles. Another limitation is the assumption that the agent is capable of braking; the method is currently unequipped to handle unstoppable agents, such as the fixed-velocity car and bike used in this thesis. It is also not clear how close the ICO($\mathcal{B}$) model can be made to resemble the true ICS region in general. Conservative models of nonviable space have significant repercussions for motion planning in constrained spaces (i.e., critical bottlenecks can be misclassified and, as a consequence, blocked). Finally, implementing this approach for a novel system seems involved and potentially difficult. The derivation of the trajectories or control sequences for the evasive actions could be significantly harder for more complex agents. Would the computation of ICO($\mathcal{B}_i, \phi_j$) be still manageable with the resultant more complex trajectories?

The approach does offer a number of key advantages though, the most important being a *guarantee* of safety, provided the problem meets the prerequisites (i.e., agent can imitate motion of obstacles). This is a crucial requirement of live PMP, one that the work in this thesis does not satisfy (but also does not pursue). The ICS work is also capable of handling dynamic environments, or at least a subclass of them, a problem the work in this thesis has not yet attempted to tackle. Finally, another notable benefit of the approach is its ability to handle partially observable environments.

Although our work does not demonstrate this, the local viability models of Chapter 4 should be capable of handling such cases too, with little or no modifications.

2.3 User-control

Significant research has been devoted to studying computer-assistance for human control of agents, with industry particularly focused on automotive applications. Today such assistance systems can be found in many places, from control-correcting systems on highly unstable stealth planes such as the B-2 bomber, down to anti-lock brake systems (ABS) in many modern cars. Perhaps the most actively studied domain for such systems is that for driving cars semi-autonomously on highways (e.g., lane-keeping, computer-assisted lane changing, "magic bumpers"). A predominant approach involves building potential fields which then serve to push the vehicle away from danger (e.g., to prevent leaving the roadway, avoid cars, and discourage driving astride multiple lanes). The approach was first introduced by [Kha86] and generalized by [Hog85] (under the term "impedance control"). A typical example of automotive application is [Ros03]. This work also provides a comprehensive review of further previous work in this area, while [VE03] looks at recent developments and trends in all facets of the general problem, including control strategies, human factors, and legal issues.

It's worth pointing out that such systems are generally agent- and task-specific. They also generally reduce to collision look-aheads. As seen in the earlier lunar lander example, such approaches always run the risk of using a look-ahead that is too short to detect and avert all onsets of unavoidable collisions. Viability-based approaches on the other hand do not suffer from this problem; even a single time-step into the future is sufficient to confirm and ensure agent safety. The only work that attempts to solve the safety problem through viability, other than our own, is that of Collision States (ICS)[FA04, PF05, PF07, Par06], which was discussed in Section 2.2.5.

Related safety boundary problems have also been investigated in [Mit02], where backward reachable states for an arbitrary target set are computed using a time-dependent Hamilton-Jacobi-Isaacs (HJI) partial differential equation (PDE). Such backward reachable sets are, for example, used to find the "capture region"[5] for two airplanes in the classical "game of two identical vehicles", where one is the evader and the other the pursuer. This has notable applications to real life: when a real plane finds itself within the capture region of another plane, there exists a sequence of control actions that the latter pilot could execute, potentially by error, that can lead to a collision, regardless of what the first pilot does; thus safety can only be guaranteed if the airplanes stay out of each other's capture regions (assuming the pilots can find and execute the appropriate evasive manoeuvres). Such capture regions thus play an important role in safety analysis.

[5] This is the set of states of the evader from which it can always be captured by the pursuer, regardless of what evasive manoeuvres it executes.

Chapter 3

RRT-Blossom: sustained, non-redundant exploration

This chapter looks at ways of improving the Rapidly-exploring Random Trees (RRT) algorithm for more complex agents. Of the current motion planners, RRT is the most promising motion planning algorithm for kinodynamic systems and those with differential constraints in general, because of its versatility and wide applicability. A comprehensive overview of RRT and its variants was presented earlier in Section 2.1.4. The other currently popular alternative, the Probabilistic Road-Maps (PRM) algorithm is not well suited to kinodynamic motion planning since it relies on having a local planner available, which is usually not the case for more complex agents due to the local planning problem being often no simpler than the global one.

RRT performance degrades rapidly as agent complexity increases, and especially when the environment becomes more constrained. A concrete example of this is illustrated in Figures 3.1 and 3.2; the first shows the progress made by the search trees of the RRT and RRT-CT (a particularly relevant variant) algorithms after 20 minutes of computation, while the second plots typical histories of edges creation for the same planners.[1] The agent used was the kinodynamic bicycle with $\mathcal{U}$ consisting of 5 control actions, sampled at regular intervals from the full range of possible steering angles. As can be seen from the figures, RRT spends most of its time on fruitless iterations, while RRT-CT, although prolific, does not take care to avoid redundancy.

In the following sections we analyze the reasons for this poor performance, and then undertake to rectify them, in order to attain a more robust low-level planner. It is worth pointing out that the scenario illustrated is not particularly special or contrived; the problems encountered here are systemic and appear, in some shape or form, for most other more complex agents.

[1] Note: the histories shown in Figure 3.2 are not from the same planner runs as those used for Figure 3.1.

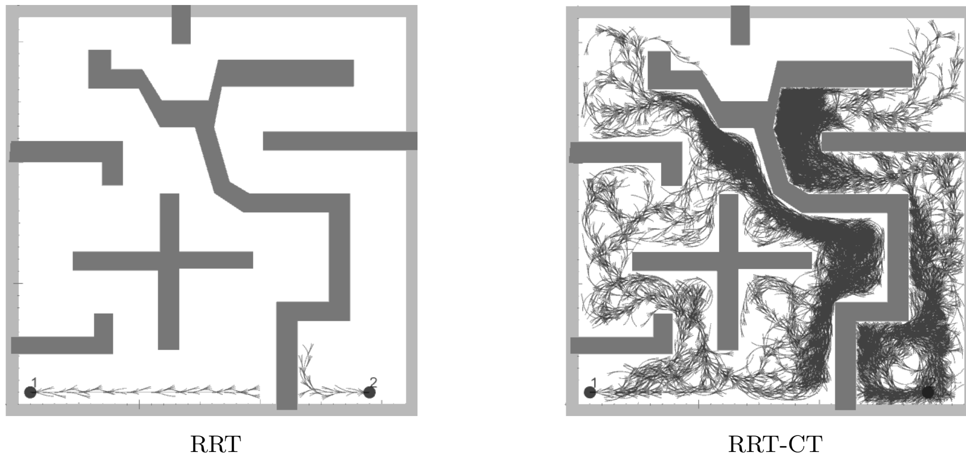


Figure 3.1: An example query in which RRT and RRT-CT (an improved variant) perform poorly. The agent is a kinodynamic bike (not shown) and it is to drive from point "1"(on left) to point "2"(on right). Plots illustrate progress made after 20 minutes of planning time, which should have been more than sufficient to find a solution.

3.0.1 RRT

The feature that immediately grabs one's eye in Figure 3.1 is RRT's startling lack of progress, despite the ample time allotted. To understand why this occurs one first needs to take note of a peculiar property of bikes: in order to execute a turn one must first steer the bike *away* from the intended direction of travel.[2] This is necessary for bringing about a bike lean which will prevent it from falling over during the desired turn, by balancing the centrifugal and gravity forces.

This steering behaviour is diametrically opposed to RRT's edge instantiation criterion, which favours edges that make immediate and direct progress toward the target. This results in search trees where the majority of created edges place the bike in an unrecoverable fall (i.e., large portion of the trees will be nonviable). Also, since RRT will never instantiate an edge that recedes from the target, it is therefore unable to "intentionally" first generate the preparatory lean. The only way for RRT to implement a "successful turn" is to have a radical switch in the desired direction of travel (i.e., a serendipitous sequence of x_{tgt} is needed) midway through a manoeuvre, thus turning the undesirable away-swing of the first turn into the requisite lean of the second.

The problem is greatly compounded by another RRT weakness: lack of memory. In the course of its planning, RRT encounters many potential edges that turn out to incur a collision or failure, yet by the end of the iteration the planner completely forgets about them. It then makes additional attempts on many of these edges in further iterations, each futile attempt incurring the full collision check and other costs. Worse, RRT does not remember nodes whose children it has "exhausted", either by instantiating or finding their inadmissibility. This is particularly troublesome because it quickly leads the search trees to develop a handful of prominent nonviable nodes which then

[2]This is the only way to execute a turn on a bike without a rider. When a rider is present, the lean required for a turn may also be achieved by shifting one's weight laterally. Even with a human rider, this counter-steering behaviour is sometimes still required (e.g., larger motorbikes).

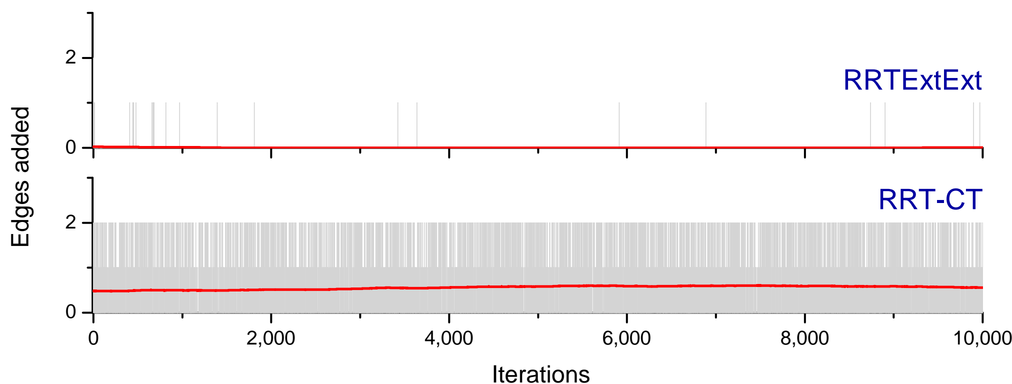


Figure 3.2: Sample histories of edge creation for typical RRTExtExt and RRT-CT runs (only 10,000 iterations shown). Red line indicates a moving average of edges created per iteration. The agent is a kinodynamic bike.

dominate the planner's attention. Unlike *viable* prominent nodes which quickly acquire children and lose their prominence, these nodes remain active for long durations, and by virtue of their protruding into unexplored space, act like lightning rods for the planner's n_{near} node selection mechanism. In short, the bulk of RRT's planning time is wasted on glaringly repetitive but futile attempts of growing the search trees from a handful of prominent yet nonviable nodes.

It bears pointing out that a root cause of this difficulty is that the RRT algorithm assumes that the distance metric used in its computations reflects the true "cost to go" from a particular state to the target. Alas, such metrics are difficult to come by for more complex agents, and therefore it is common practice to use the Euclidean distance (L_2 metric) instead, as a rough approximation. Figure 3.3, for example, shows an empirically derived approximation of the true "cost to go" for a nonholonomic car. The plot of the corresponding L_2 metric would naturally consist simply of concentric circles. Although at larger distances the L_2 metric makes a suitable approximation, at a smaller scale it is a poor model, and therefore leads the planner into making incorrect decisions, as seen above. With a proper metric, on the other hand, RRT would automatically implement the required preparatory "steering away" since the anticipatory steering behaviour would be inherent in the metric itself. Any such distance metric, after all, merely measures the distance of the optimal trajectory from a particular state to a desired destination; if said optimal trajectory involves a preparatory away-turn, as above, then this will be reflected in the gradient of the metric, as subsequent points along the optimal trajectory need to have monotonically decreasing metric values.

3.0.2 RRT-CT and "receding edges"

The RRT-CT algorithm aims to directly address this sensitivity to distance metrics. Most notably it remembers which attempted edges have been found to lead to failure. But as Figure 3.1 shows, RRT-CT introduces a new problem: redundant exploration. The (original) RRT algorithm forestalls

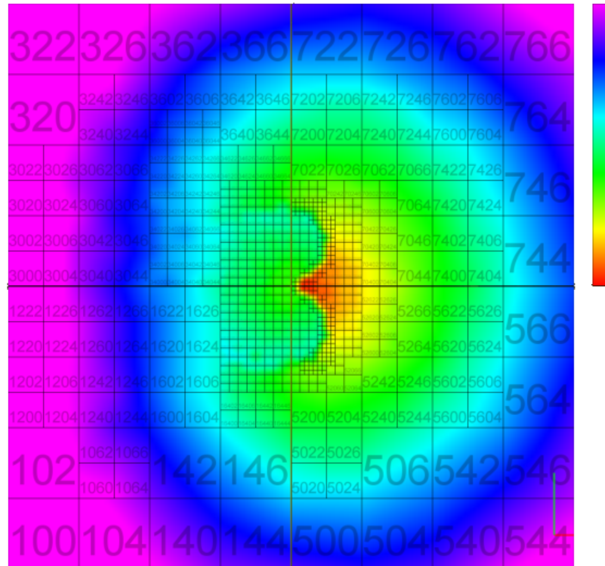


Figure 3.3: A cross-section of empirically-derived "true cost to go" metric for a nonholonomic Dubin's car (i.e., without a "reverse" gear). The value/colour at (x, y) indicates the minimal distance the car would have to travel to achieve its current orientation but at a relative displacement of (x, y). The square outlines and numbers are an artifact of the metric derivation tool.

this problem by design: by first choosing n_{near} to be the node closest to the target, and then only instantiating edges which approach the target further, absence of re-exploration is guaranteed—for if a node were to already exist where the new edge extends to, that node would have been chosen as n_{near} instead. But RRT-CT removes this important constraint[3], thus allowing new branches to regress into already explored space.

Yet such (target-)*receding edges* do not always result in re-exploration; just as often they may strike out into unexplored space, yielding progress when otherwise there would be none, in iterations where no suitable target-approaching edges exist. Figure 3.4 illustrates receding and *regressing* (i.e., re-exploring) edges, and their difference. We note that these edge types are not mutually exclusive: a receding edge can be also regressing, while a regressing edge is almost always a receding one, due to the way RRTs are constructed. The inclusion of receding edges is a major contributor to RRT-CT's improved performance since, on average, it is generally able to effect more progress per iteration.

In short then, receding edges can either be productive and yield progress on iterations which would otherwise be fruitless, or they can lead to re-exploration of known space and thus be wasteful. Neither RRT, which rejects both types, nor RRT-CT, which allows both types, makes the necessary distinction. Our proposed planner, outlined in the next section, does distinguish between good and bad receding edges, and is thus able to derive maximum benefit from them.

[3]Specifically, it does so by setting $d_{min} \leftarrow \infty$ rather than $d_{min} \leftarrow \rho(q, q_{tgt})$

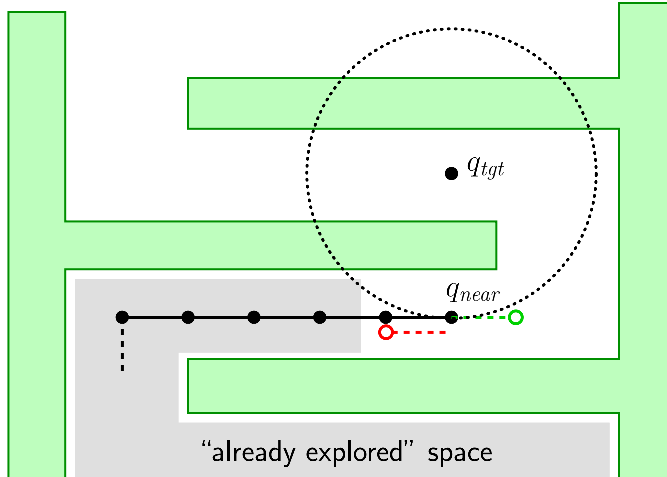


Figure 3.4: Receding edges; the diagram above shows a snapshot of an RRT iteration in heavily constrained environs. The agent is capable of motion only in the four cardinal directions. In this scenario only two collision-free edges are possible from q_{near} (dashed; leftward edge is offset downward to avoid superposition). The dotted circle passes through q_{near} and marks the locus of points which are equally distant from q_{near}; it demonstrates that both the above edges are *receding* from the target point. The green (rightward) receding edge is useful since it explores fresh space; the red (leftward) one is undesirable because it regresses into already explored space.

3.1 RRT-Blossom

3.1.1 Regression avoidance

The central novel feature of the proposed planner, which we call *RRT-Blossom*, is the inclusion of receding edges, but only those which do not regress into already explored space. Filtering is achieved in a direct manner: all potential new edges are simply tested to see if they intrude upon already-explored space; any found to be regressing are eliminated from further consideration. Implementing the regression test itself is problematic, however. Firstly, the problem is ill-defined: there is no obvious criterion (or even one that is marginally better than any others) for deciding the extent of the neighbourhood around the current search tree which one could consider to be "explored" or occupied by the tree. Mathematically the edges occupy zero volume by virtue of being lines. Also, even if this question were to be resolved, computing an explicit model of the explored space would likely be prohibitive, and subject to the curse of dimensionality. Luckily both these difficulties can be sidestepped by using an implicit approximation, one that captures the desired spirit of the term and that has been very effective in trials. This approximation decrees that an edge (x_{parent}, x_{leaf}) is regressing when x_{leaf} is closer to a tree node *other* than x_{parent}. That is, regression occurs when

$$\exists x \in T \;\mid\; \rho(x, x_{leaf}) < \rho(x_{parent},\, x_{leaf}), \tag{3.1}$$

where T is the search tree, and ρ is the distance metric (usually L_2) used throughout the rest of RRT. Figure 3.5 further illustrates the concept.

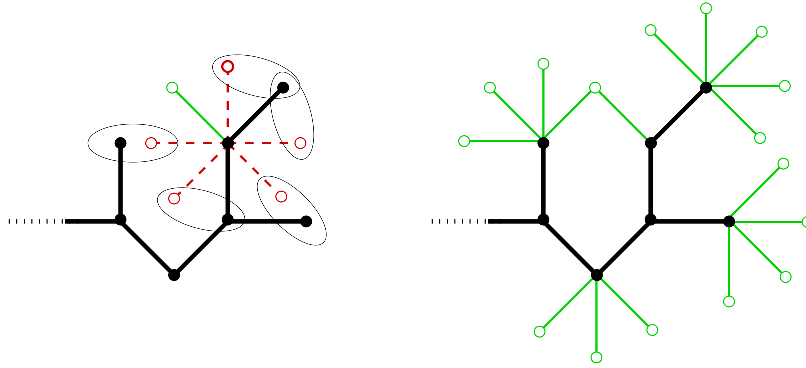


Figure 3.5: The "regression" test for an agent capable of 8-way motion; **left:** potential offspring for a node are shown. Edges in red (dashed) are *regressing* since the new endpoint lies closer to a node other than its parent (indicated with loops). Only the single edge in green is suitable for instantiation. **right:** all possible "regression"-free edges are shown in green, for the tree state shown. Naturally, instantiating any one of these edges will eliminate some of their neighbouring candidates. In the long run only a partial subset of these edges will be added to the tree, with the rest of the candidates discarded. The final structure of the tree will thus depend on the sequence and choice of edges added.

3.1.2 Node "blossoming"

At the same time, the larger, overarching goal of the planner is to be more efficient, to simply "do more per iteration". To this end the proposed planner makes an additional novel change: rather than creating only the single best edge out of n_{near}, it instantiates *all* such admissible edges. This gives the appearance of n_{near} "blossoming" with a spray of new edges, thus prompting the "RRT-Blossom" moniker. The key benefit of this alteration is better computational economy. Although RRT-CT remembers which attempted edges incurred a collision, it does not track collision-free edges which too have been attempted but remain uninstantiated (i.e., those that had a sibling edge that approached x_{tgt} closer, and was hence chosen instead). RRT-CT will unnecessarily recompute the particulars (edge endpoint, collision check, etc.) of such nodes the next time they are reconsidered. Instantiating all eligible edges thus avoids such re-computation, and has little negative cost: in more difficult regions where the planner needs to expend more effort, most of these edges would be eventually created anyhow, while expansive spaces are traversed quickly with few iterations, thus incurring negligible overhead. Finally, this behaviour is more consistent with RRTs "rapidly-exploring" spirit.

3.1.3 Bottleneck obstruction issue

For many simple agents the above two features, the regression avoidance and node blossoming, are all that is needed to convert RRT into RRT-Blossom. Algorithm 4 gives the pseudocode for this simplest variant.

Alas, with more complex agents this variant can run into trouble with narrow bottlenecks—regions of state-space which may be traversed only with a very limited set of possible trajectories. Such bottlenecks can be caused by corresponding choke-points in the environment itself, or by

Algorithm 4 RRT-Blossom for simple agents

```
(inherits query() from dual-tree RRT, page 19)
 1: function GROW_TREE(τ, x_tgt)
 2:     x_near ← nearest_neighbour(τ, x_tgt)
 3:     x_new ← node_blossom(x_near, x_tgt, τ)
 4:     return x_new
 5: function NODE_BLOSSOM(x, x_tgt, τ)
 6:     for u ∈ U do
 7:         x_new ← sim(x, u)
 8:         if failure(x, u, x_new) then
 9:             next u
10:         if regressionp(x, x_new, τ) then
11:             next u
12:         τ ← τ + new_edge(x, u)
13:     return the new node closest to x_tgt
14: function REGRESSIONP(x_parent, x_new, τ)
15:     for node n ∈ τ do
16:         if ρ(n, x_new) < ρ(x_parent, x_new) then
17:             return True
18:     return False
```

general agent instability, leading to situations where only a very few control actions can maintain system viability. The issue is that the no-regression constraint can inadvertently close such bottlenecks off, which is particularly detrimental if the obstructed choke-point happens to be critical to achieving a solution (e.g., the only bridge across a river) since the planner will then be unable to solve the problem. See Figure 3.6, for an example.

Such obstruction occurs when the area around a bottleneck becomes populated with branches that, for whatever reason, cannot directly traverse the choke-point (e.g., the agent is unfavourably oriented and no application of its laws of motion can negotiate the narrow therefrom), yet at the same time are collectively sufficient to completely mark off the area as "explored", thus barring any further branches from attempting passage, including ones that would otherwise succeed.

Ultimately, the planner's operation proceeds by the repeated sampling of the search-space, and the no-regression constraint merely attempts to put an upper bound on the resultant density of samples. This in essence limits the method's "resolution", in that it will have trouble detecting terrain features (e.g., bottlenecks) whose size is comparable to or smaller than the sampling density. Clearly increasing the resolution—by lowering the simulation time-step dt, which in turn will decrease length of edges, and hence distance between nodes—would work, but this is to be avoided, if possible, as it exponentially increases the size of the search trees. We thus focus on other ways to treat the problem, mostly by conditional relaxing of the no-regression rule.

Obstruction can be caused by either viable or nonviable branches; we consider treatment for the latter case first since the derived mechanism is a stepping stone toward addressing the former.

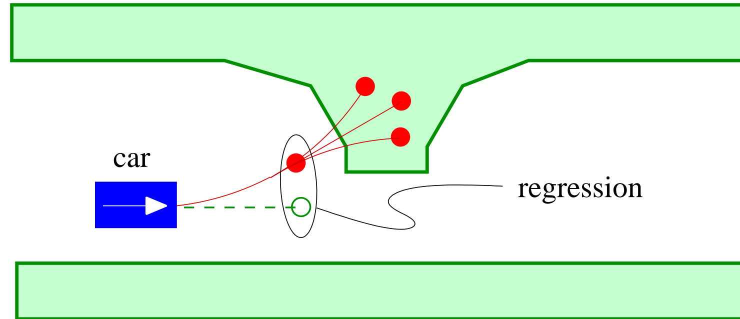


Figure 3.6: Interplay of viability and the no-regression constraint: the green (dashed) expansion is "blocked" by an extant nonviable edge (i.e., instantiating the edge would constitute a regression). Since the blocked edge is essential, the planner will not be able to find a solution.

Obstruction by nonviable branches

What is particularly vexing about this case is that critical edges are blocked by branches which, by definition, cannot lead to a solution and are thus useless. Figure 3.6 illustrates an example of this. This observation suggests one trivial fix: nonviable edges could be simply ignored when testing for regression. That is, nonviable edges would no longer "occupy" or "explore" their neighbourhoods. This effectively removes the sampling density restriction on the nonviable side of the state-space, which is a double-edged sword: although this immediately corrects the problem of blockage by nonviable branches, it also permits redundant exploration of nonviable space. In many motion planning problems the nonviable space comprises a small percentage of the search-space, hence the gains would still far outweigh this drawback. For more complex systems the drawback can be corrected by restricting planner search to viable space only; this topic is explored in the next chapter. Further fallout of this decision is discussed in Section 3.3.

Unfortunately the nonviability of edges is not known ahead of time, and must therefore be discovered during planning, and then incorporated retroactively. The discovery mechanism is implemented by annotating edges with a viability status, which is then updated as planning progresses. More specifically, each edge, instantiated or potential, carries a viability status, one of: `untried`, `live`, `dormant`, or `dead`. Edges that have not yet been considered are marked `untried`. Upon instantiation they become `live`. If the edge is currently disallowed by the regression test, it is marked `dormant`. Finally, edges that have been found to be nonviable are marked `dead`. Figure 3.7 shows the transitions in greater detail.

Since changing the status of an edge *may* precipitate a change in the parent (e.g., the last of the children of an edge turns `dead`, suggesting that the parent edge itself is now `dead`), status updates must be propagated up the tree. This is done by traveling up the parent hierarchy toward the root node, re-evaluating the status of each edge passed. The process stops when root node is reached, or when a re-evaluation results in no change of status.

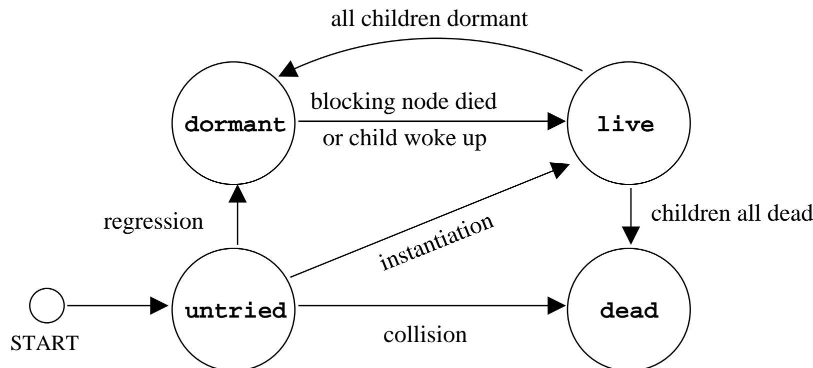


Figure 3.7: A Finite State Machine (FSM) used to track the viability status of nodes.

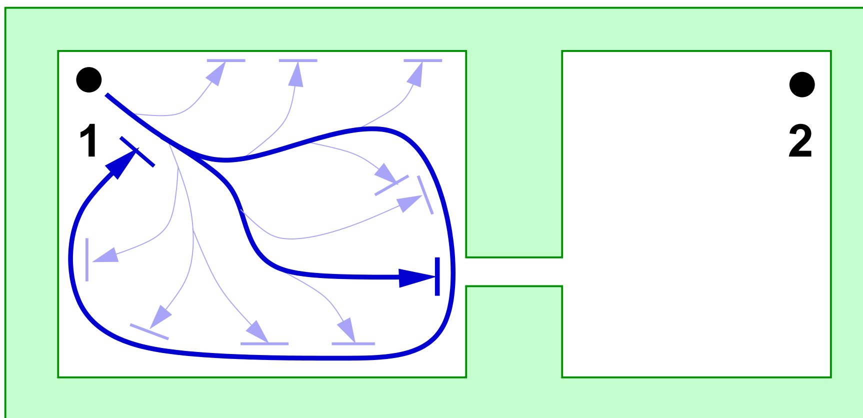


Figure 3.8: `dormant` deadlock: a viable branch may cutoff access to a critical passage without being able to explore it itself. This limits the planner's exploration to the "fenced off" area, and once this is exhausted, the remaining non-`dead` branches are locked in a cycle, mutually blocking each other's way.

Obstruction by viable branches

The analogous approach of unrestricting, or at least increasing the sampling density in the viable part of the search-space is less appealing in this case. Rather than simply detecting whether an edge is nonviable, one would instead have to ascertain whether one is near the border of viable space, a nontrivial task complicated further by the ill-defined nature of "near". We thus adopt a different tack, especially since in tests this case appears to be relatively rare.

When a fragment of the planner's search-space becomes isolated by blocked bottlenecks, it will eventually be exhausted by the planner. That is, in time more and more edges will become `dormant`, and this condition will slowly propagate up the tree until the root is reached. When the tree root becomes `dormant`, the accessible fragment of the search-space has been exhausted, and the tree is in "`dormant` deadlock". Figure 3.8 illustrates this condition. A dormant deadlock can occur either because all exits (i.e., bottlenecks) from the area have become blocked, or because x_{goal} is simply not reachable from x_{init}, given the problem parameters and simulation time-step.

A trivial fix for dormant deadlock is to simply ignore the no-regression constraint for the very next planner iteration. This is perhaps heavy-handed, inelegant, and not particularly efficient, but

considering the observed rarity of dormant deadlocks, and the fact that the fix works sufficiently well in tests, it would seem sufficient for most applications. Alas, this fix prevents the planner from terminating in queries where a solution is impossible. Interestingly, this also acts as an "escape valve" that allows any edge to be created in the long run, thereby restoring probabilistic completeness to the planner. Finally it is worth noting that this approach has no impact on queries for which dormant deadlock is not an issue, other than the single test per iteration to check whether the root node is `dormant` or not. The resultant complete RRT-Blossom algorithm is shown in Algorithm 5.

3.2 Experiments

3.2.1 Agents

Holonomic point

The simplest agent attempted is a kinematic point, capable of moving in 8 directions (i.e., $||\mathcal{U}|| = 8$), as shown in Figure 3.9. Traditionally, when solving problems for kinematic agents using RRT, the tree edges are grown directly toward the x_{tgt} samples, without pretending the agent is actuated, and without restring the agent's motion to a discrete set of directions. We have cast the kinematic point as an actuated agent as this is the class of problems we are interested in studying, and no other actuated agent illustrates the planner's operation as well, or gives as much insight into it. The discretization is then a natural consequence of treating direction of travel as the system control input.

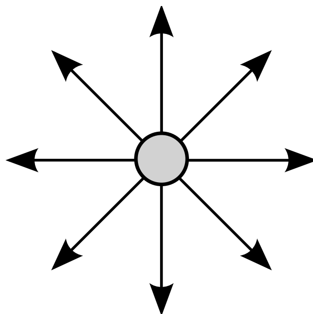


Figure 3.9: The 8-way "holonomic point" agent.

The state vector for this agent is:

$$\vec{x} = [x\ y] \tag{3.2}$$

Nonholonomic car

This first-order system is essentially a Dubin's car [Dub57]—a car restricted to forward-only motion—which is being modeled internally as having only one front and one back wheel, as shown

Algorithm 5 viability-aware RRT-Blossom

(inherits `query()` from dual-tree RRT, page 19)

```
 1: function GROW_TREE(τ, x_tgt)
 2:     x_near ← nearest_neighbour(τ, x_tgt)
 3:     x_new ← node_blossom(x_near, x_tgt, τ)
 4:     return x_new

 5: function NEAREST_NEIGHBOUR(τ, x_tgt)
 6:     d_min ← ∞
 7:     for node n ∈ τ do
 8:         if n is dead then
 9:             next n
10:         workable_states ← {untried, live}
11:         if deadlock then
12:             workable_states ← workable_states, dormant
13:         if n.edge_status ∩ workable_states = ∅ then
14:             next n
15:         d ← ρ(n, x_tgt)
16:         if d < d_min then
17:             d_min, n_min ← d, n
18:     return n_min

19: function NODE_BLOSSOM(n, x_tgt, τ)
20:     x ← n.x
21:     for u ∈ U do
22:         x_new ← sim(x, u)
23:         if failure(x, u, x_new) then
24:             n.edge_status[u] ← dead
25:             next u
26:         if not in deadlock then
27:             n_blk ← regressionp(x, x_new, τ)
28:             if n_blk then
29:                 n.edge_status[u] ← dormant
30:                 n_blk.blocked_edges ← n_blk.blocked_edges  +(n, u)
31:                 next u
32:         τ ← τ + new_edge(x, u)
33:         n.edge_status[u] ← live
34:     propagate_status(n)
35:     return new node closest to x_tgt

36: function REGRESSIONP(x_parent, x_new, τ)
37:     for node n ∈ τ do
38:         if n.status = dead then
39:             next n
40:         if ρ(n, x_new) < ρ(x_parent, x_new) then
41:             return n
42:     return ∅
```

Algorithm 6 the function `propagate_status`

```
1: function PROPAGATE_STATUS(n)
2:     while n do
3:         s_n ← dead
4:         for u ∈ U do
5:             s_e ← n.edge_status[u]
6:             if s_e = live then
7:                 s_e ← n.children[u].status
8:             if s_e ∈ {untried, live} then
9:                 s_n ← live
10:            if s_e = dormant then
11:                s_n ← dormant
12:        if n.status = s_n then
13:            return
14:        n.status ← s_n
15:        if s_n = dead then
16:            for (n_b, u) ∈ n.blocked_edges do
17:                if n_b.edge_status[u] = dormant then
18:                    n_b.edge_status[u] ← untried
19:                    propagate_status(n_b)
20:        n ← n.parent
```

in Figure 3.10 (sometimes called the "bicycle model" for a car). This gives a very simple nonholonomic model which, if needed, can later be retrofitted with the more common four wheels, provided that their turning radii are adjusted appropriately (to give a more natural visual representation). Furthermore, this car is assumed to move at a constant velocity and is restricted to a maximum steering angle $\psi_{max} = \pi/6$ radians, which yields a minimum turning radius that happens to be equal to the car's wheelbase, namely 1.275 metres.

The agent's state is

$$\vec{x} = [x\ y\ \theta] \tag{3.3}$$

where (x, y) is again the agent's position in the environment, while θ is the agent's orientation angle. The car has a fixed forward velocity, and control consists solely of steering, using one of three steering angles:

$$\mathcal{U} = \{-\psi_{max},\ 0,\ \psi_{max}\} \tag{3.4}$$

Kinodynamic bike

The bike, which is the most complex agent we have used, has a 5D state vector:

$$\vec{x} = [x\ y\ \theta\ \phi\ \dot{\phi}\,] \tag{3.5}$$

where ϕ is the bike's lateral lean, and the other variables are the same as for the car. Its control input is the steering angle, while the forward velocity is again fixed. Due to the more unstable

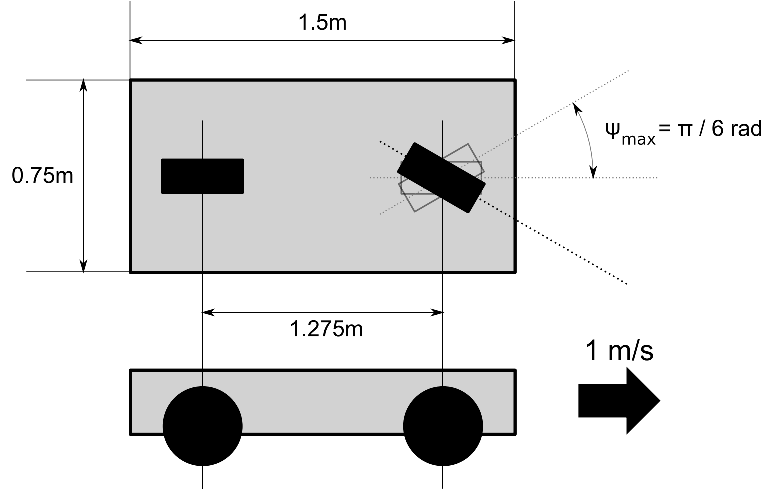


Figure 3.10: The "nonholonomic car" agent.

nature of bike control we found it necessary to use a finer discretization of the control action set:

$$\mathcal{U} = \{-\psi_{max},\ -\tfrac{1}{2}\psi_{max},\ 0,\ \tfrac{1}{2}\psi_{max},\ \psi_{max}\} \tag{3.6}$$

This steering angle is simultaneously used to maintain balance as well as effect progress. The maximum wheel deflection ψ_{max} was set to $\pi/4$ radians.

The maximum allowable lean was set to $\phi_{max} = {}^{\pi}/_{6}$ radians, and similarly the maximum lateral velocity was set to $\dot{\phi}_{max} = {}^{\pi}/_{6}$ radians per second. Any bike state which strayed beyond these bounds was flagged as failed (i.e., considered to have "fallen over").

The equations of motion for this second-order system were taken from [vdP94], and are based on modeling the bike as an inverted pendulum ([vdP94] provides a derivation). In summary:

$$\ddot{\phi} = \frac{lm(g\sin\phi - k)}{I_\phi + l^2 m} \tag{3.7}$$

where l is the "pendulum length", the (shortest) distance from centre of mass to the ground along the bike's plane, m is the mass of the bike, g is the gravitational constant, I_ϕ is the moment of inertia about the main axis of the bike, and

$$k = \frac{C_f {V_{cm}}^2 \cos\phi}{1 - C_f l \sin\phi} \tag{3.8}$$

Here V_{cm} is the forward velocity of the bike, implemented as 2 ${}^{\mathrm{m}}/_{\mathrm{s}}$, and C_f is the trajectory curvature resulting from the current steering control action, computed here as the reciprocal of the average turning radius of the two wheels:

$$C_f = \frac{2}{w\tan(\frac{\pi}{2} - \phi) + \frac{w}{\cos(\frac{\pi}{2} - \phi)}} \tag{3.9}$$

where w is the wheelbase of the bike, the distance between the wheel axles. Figure 3.11 illustrates the parameters.

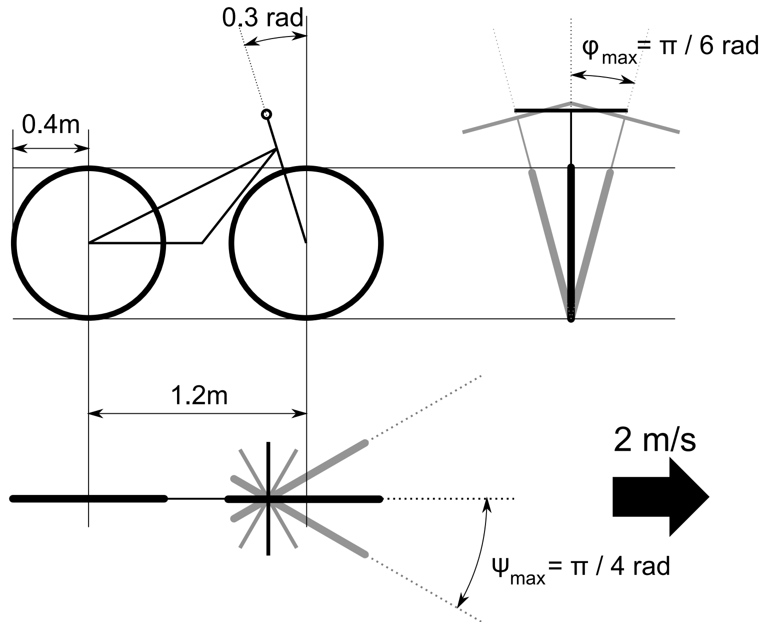


Figure 3.11: The "kinodynamic bike" agent.

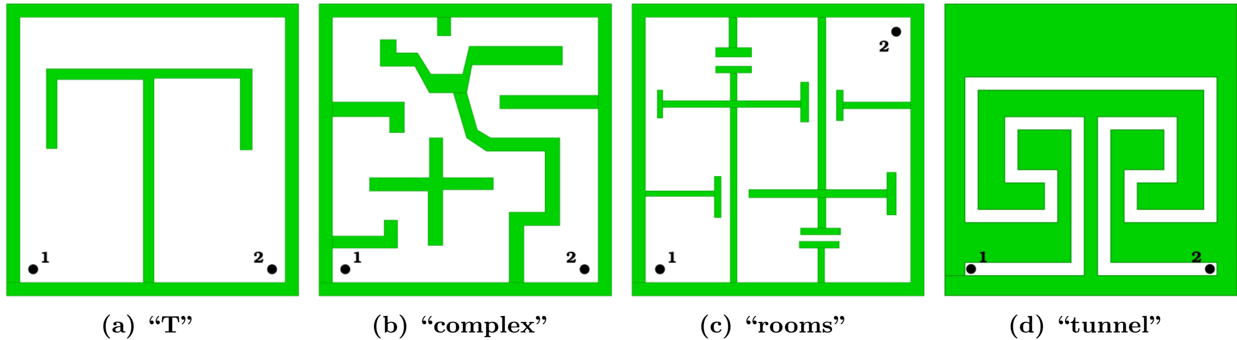
3.2.2 Environments

Figure 3.12 illustrate the problem environments used with the holonomic point. Nearly identical versions were used for the other agents, with a few minor alterations to allow for the reduced manoeuvrability, speed, and turning radius of the agents. In particular the "jambs" were removed from the doorways in "rooms", while "tunnel" was widened to accommodate the turning radii of the subjects. The problem environments were chosen to present deep local minima ("T"), to be highly constrained ("tunnel"), as well as mixes of these qualities ("complex" and "rooms")

For each environment the maximal query was used, one whose endpoints have been placed as far apart as the environment allows; the points labeled "1" and "2" mark x_{init} and x_{goal}, respectively. The orientation of the car and bike agents at both endpoints was specified to be "facing right, along the x-axis", and in the case of the bike, the lean angle ϕ and its velocity $\dot{\phi}$ were required to be zero.

3.2.3 Test platform

All algorithms were written in Python 2.3, running on Linux (Debian "sid", kernel 2.6), using Psyco (a JIT-like optimization for Python), on a Pentium IV 2.4 GHz machine. The algorithm implementations share the same component functions where feasible.



(a) "T" (b) "complex" (c) "rooms" (d) "tunnel"

Figure 3.12: Environments used in tests (kinematic agent variants shown).

3.2.4 Results

The obtained results are presented below in four forms: numerical tables, boxplots, a visual comparison of evolved tree structures, and a history plot of node production. The algorithms used are RRTExtCon for "RRT", RRTExtExt with Collision Tendency for "RRT-CT", and "RRT-Blossom" was based on RRTExtExt as well.

Tables 3.1–3.3 give the runtimes and other relevant statistics. The columns in the tables give, in order: algorithm runtimes, number of collision checks, number of nearest neighbour checks, and number of nodes created. These values are averages over the indicated number of runs. Furthermore, the planner runs were time-limited; the last column gives the number of runs which failed to find a solution within that period. Rows shown in *italics* indicate planning problems for which a significant portion ($> 10\%$) of the runs did not find a solution within the specified time limit; values in these rows will thus underestimate the true cost of solving the given problem. It should also be noted that data in the "NN" columns does *not* include the NN queries used in the no-regression constraint since the latter is computed using a different, cheaper method. Finally, Table 3.3 lacks data for "RRT" since the algorithm was unable to make significant progress (see Figure 3.14) in the allotted time (over 20 minutes).

Figure 3.13 provides a more visual and thorough representation of the runtimes, additionally depicting the variance in the samples. The figure is divided first into three plots, one for each agent; these are then subdivided by environment, and then once more by algorithm. In each of these boxplots (also known as "box-and-whisker" plots) the box extends from the first to the third quartile, with a vertical line dividing it at the median sample, while the small square represents the average. The "whiskers" extend from the box to the 10th and 90th percentiles; samples outside that range are marked with circles and are considered outliers. Finally, the minimal and maximal samples are marked with triangles. The boxplots for the bike have been plotted on a logarithmic scale to better show the variance of RRT-Blossom runtimes.

Figure 3.14 shows examples of evolved tree structures for the three algorithms. It should be noted that the pictures for RRT and RRT-CT in the bottom row portray incomplete trees (i.e.,

Table 3.1: agent: **holonomic point**; values averaged over 100 runs, `max_time` = $20s$

environment	algorithm	time	failure()	NN	nodes	time-outs
T	RRT	3.45	21,100	2628	410	—
	RRT-CT	*19.06*	*13,250*	*2870*	*2870*	97
	RRT-Blossom	0.90	2246	280	316	—
complex	RRT	2.75	10,048	1247	281	—
	RRT-CT	10.90	8858	1889	1889	6
	RRT-Blossom	0.85	1767	221	266	—
rooms	RRT	*13.10*	*39,398*	*4911*	*621*	48
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-Blossom	2.25	3276	409	499	—
tunnel	RRT	3.68	22,080	2754	122	1
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-Blossom	0.21	944	118	118	—

Table 3.2: agent: **nonholonomic car**; values averaged over 100 runs, `max_time` = $60s$

environment	algorithm	time	failure()	NN	nodes	time-outs
T	RRT	9.39	13,317	4407	486	—
	RRT-CT	35.13	8890	3848	3585	8
	RRT-Blossom	1.36	1343	451	448	—
complex	RRT	23.62	13,656	4542	294	9
	RRT-CT	11.42	4049	1677	1465	—
	RRT-Blossom	1.39	811	295	267	—
rooms	RRT	*32.62*	*27,119*	*9014*	*724*	42
	RRT-CT	9.59	4071	1717	1507	—
	RRT-Blossom	3.53	1967	644	649	—
tunnel	RRT	*51.27*	*24,917*	*8281*	*408*	77
	RRT-CT	N/A	N/A	N/A	N/A	100
	RRT-Blossom	1.43	806	277	266	—

Table 3.3: agent: **kinodynamic bike**; values averaged over 40 runs for RRT-Blossom, and over 3 runs for RRT-CT.

environment	algorithm	time	failure()	NN	nodes	time-outs
T	RRT-CT	1666.81	139,488	37,032	25556	—
	RRT-Blossom	103.02	34,054	8538	5808	—
complex	RRT-CT	1215.39	128,572	33,894	22,424	—
	RRT-Blossom	67.49	27,368	7088	4675	—
rooms	RRT-CT	345.85	64,570	17,034	11493	—
	RRT-Blossom	154.90	43,822	11,049	7461	—
tunnel	RRT-CT	1536.77	182,082	47,973	29,879	—
	RRT-Blossom	62.65	28,744	7476	4903	—

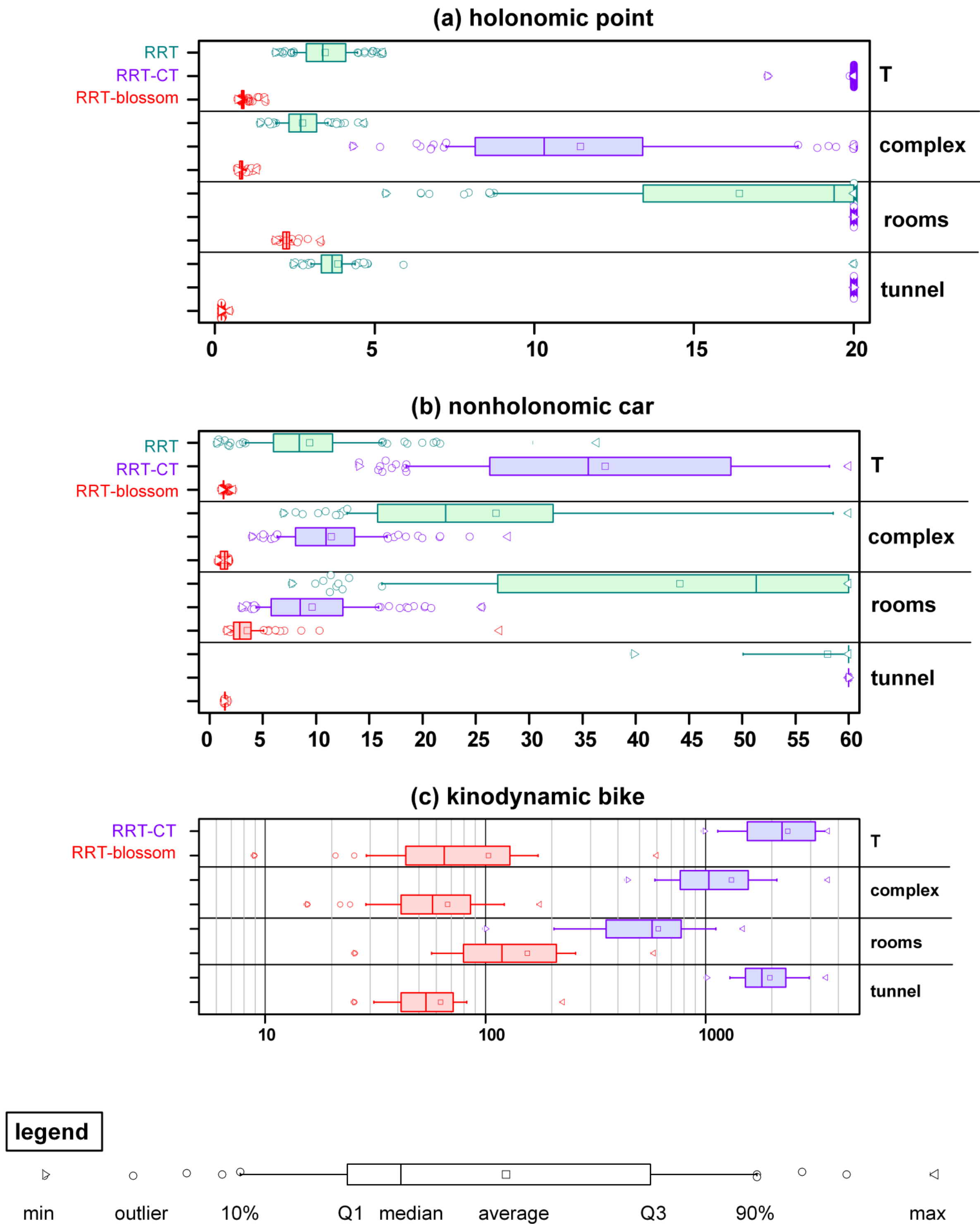


Figure 3.13: Algorithm runtimes, in seconds: **(a)** holonomic point (100 samples per boxplot); **(b)** nonholonomic car (100 samples per boxplot); **(c)** kinodynamic bike (40 samples per boxplot).

solution has not yet been found in those cases). The RRT-CT trees in particular would be much more profuse at time of solution discovery, but the resultant mass of edges would obliterate the visible detail of the trees, hence this premature stage has been illustrated instead. The RRT diagram is also incomplete because a solution could not be found with that algorithm in a feasible amount of time.

Finally, Figure 3.15 compares typical edge creation histories for the three algorithms. The queries from which these histories were derived were performed using the car agent in the "complex" environment. Similar patterns were observed with the holonomic point agent. The earlier Figure 3.2 shows histories for the bike.

3.3 Discussion

3.3.1 Benefits

Performance improvement

RRT-Blossom outperforms both RRT and RRT-CT in all of these scenarios, often by an order of magnitude, or more. It is interesting to note that it also generally tends to have a much smaller runtime variance, which is highly desirable[IMT03]. Some of the particularly poor showings by the other algorithms warrant an analysis.

In the holonomic point case RRT-CT's poor performance stems from the ease with which self-negating edge pairs are created ($\mathcal{U}$ often contains complementary pairs of control inputs, where one undoes the displacement of the other). Since it lacks regression-prevention, RRT-CT succumbs to a back-and-forth chase around local minima, and only a rare x_{tgt} choice can break this behaviour.

In the nonholonomic car tests, the deep local minima of "T" are again a problem for RRT-CT for similar reasons, but otherwise RRT-CT outperforms RRT as expected. "Tunnel" proves particularly difficult for RRT and RRT-CT due to the random target x_{tgt} distributions that are often directionally-biased[4].

The bike queries are effectively impossible for RRT, since it tends to quickly evolve prominent nonviable nodes, as discussed earlier, and these occupy the bulk of the planner's attention. Since not a single RRT query made any significant headway, we have not included it in the results table. RRT-CT fares better, but it still carries an exorbitant cost in time and number of nodes required, again due to redundant exploration.

In the comparison of the evolved tree structures it is interesting to note how RRT-Blossom's structure is strikingly regular in the kinematic case (i.e., holonomic point), which is a direct manifestation of its regression-less nature (although RRT generally does not have redundant exploration,

[4] x_{tgt} is chosen uniformly from the state-space, but for off-centre n_{near} nodes, especially ones closer to the edges of the terrain, this translates to a skewed distribution of growth directions, from the node's point of view.

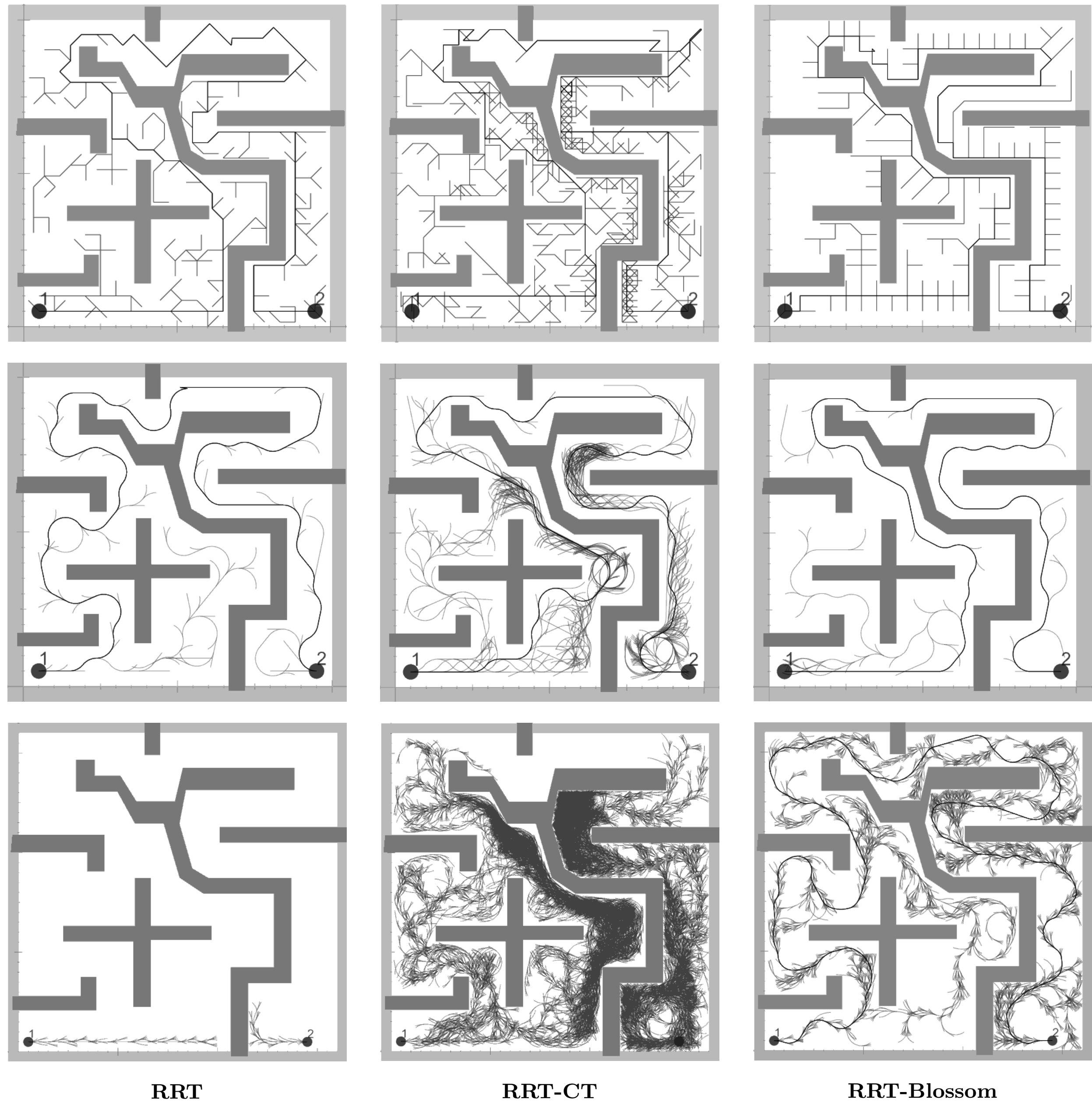


Figure 3.14: Comparison of evolved tree structure for various agents (rows) and algorithms (columns) in the "complex" environment; **top row:** holonomic point; **middle row:** nonholonomic car; **bottom row:** kinodynamic bike.

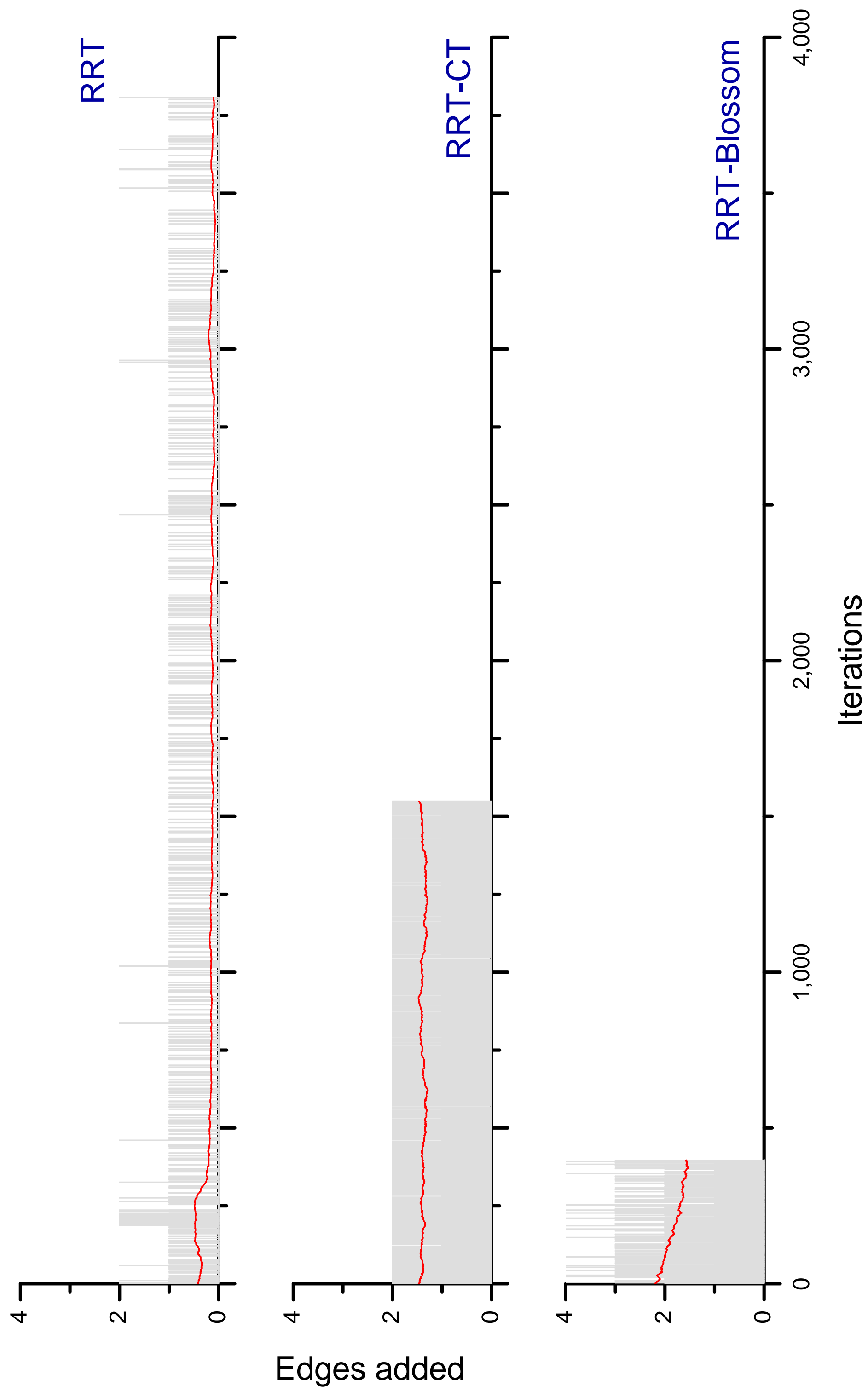


Figure 3.15: Comparison of typical edge creation histories for car agent.

it does allow edges which would be rejected by RRT-Blossom's regression test). In the car scenario, the RRT and RRT-Blossom trees are indistinguishable; the only difference lies in the time required to obtain the trees. In the bike scenario RRT makes clearly little progress in the allotted time (20 minutes), while RRT-CT is overwhelmed by massive redundant re-exploration. This RRT-CT behaviour is clearly visible in the previous scenarios as well, but to a lesser extent.

Problem difficulty vs. degree of environment constraint

In general, highly constrained environments should lead to simpler problems, given that adding constraints reduces the search-space. After all, in the extreme, an environment which supports only a single trajectory should be trivial as there is little choice in how to proceed. The most difficult motion planning problems should be the ones that are neither highly constrained nor underconstrained, but rather somewhere in the middle of these two extremes. A loose parallel can be drawn with observations about the satisfiability (SAT) problem and other NP-hard problems, where many such problems can be summarized by at least one order parameter, and that the hard problems occur at a critical value of such a parameter[CKT91, SML96], resulting in an "easy-hard-easy" progression of difficulty as a function of the number of constraints.

The runtime of current planners does not exhibit this behaviour; in fact, runtime generally increases monotonically as the amount of environmental constraint increases. Correcting this behaviour was one of the motivating goals of the RRT-Blossom design. The results seem to indicate moderate success in this regard. Clearly the "tunnel" is near the extreme case of constraint where very few possible trajectories remain, and indeed this turns out to be the easiest problem for RRT-Blossom, while being one of the toughest for RRT and RRT-CT. The amount of constraint in the "rooms" environment is probably the one closest to the critical value, and this too is reflected in runtimes, with this scenario taking the longest. One might expect "T" to be near the underconstrained extreme, but it is actually somewhat difficult due to the need to find the entrances to the upper corridor. A much better comparison for the underconstrained case would be an obstacle-free environment, or one with at most a handful of small obstacle "islands"; in those cases the runtimes are significantly shorter, with RRT-Blossom as with any of the other planners. Finally, the "complex" environment is a bit surprising as one would expect its behaviour to reflect more that of "rooms", to be closer to the critical value of constraint, yet it seems to perform much faster. One possible explanation for this is that the environment is still relatively underconstrained, and furthermore does not contain difficult-to-traverse portals, such as those found in "T" or "rooms".

Potential source of viability data

As a byproduct of its operation, RRT-Blossom can accumulate a wealth of reachability and viability information. Clearly any state reached by the search trees is reachable, and more importantly, any

node which attains `dead` status represents a nonviable state, at least when subject to the problem parameters used (i.e., the particular discretization of control space and the simulation time-step used). Such data can be exploited to "learn" further planner improvements; the work in the next chapter illustrates one example of this.

3.3.2 How improved operation is achieved

RRT-Blossom's improved performance comes from allowing creation of receding edges, and to a lesser extent, also from "blossoming". In more abstract terms, it comes from pressing on with tree growth in iterations where the regular RRT algorithm would falter. Although exactly accurate, neither of these descriptions is particularly enlightening. A more intuitively satisfying picture can be painted by first noting parallels between the operation of RRT and a potential field planner, and then using this analogy to convey the behaviour of the algorithm in more macroscopic and abstract terms.

The planner similarity is best seen in the case of the single-tree RRT algorithm, with x_{goal} bias. The iterations where the tree is grown toward x_{goal} amounts to a descent down a potential field, which in this case is the trivial $\rho(x, x_{goal})$ metric, the distance to goal. In the remaining iterations, on the other hand, the tree growth toward random targets x_{tgt} is reminiscent of random walks that potential field planners employ to escape local minima. The analogy is clearly imperfect though. In RRT the random walks are not really individual paths, but rather trees, or families of random walks that have been grown in a disjointed and desultory fashion. This is further obscured by the fact that the two types of iterations are multiplexed and tightly intertwined, so it is unusual to find substantial parts of the trees which represent either a continuous random walk, or a continuous descent (except when using RRT-Connect).

The potential field planner analogy can also be made in the case of a dual-tree RRT case. The growth of the first tree toward random targets again resembles (a tree of) random walks, while the growth of the first tree toward the second once again resembles gradient descent, although in this case the potential is some approximate measure of distance to the first tree. Figure 3.16 illustrates the parallel. Specifically, in Algorithm 2, lines 4–5 make up the EXPLORE mode, which behaves like an escape mechanism, while line 7 constitutes the SEEK mode, which corresponds to gradient descent.[5]

What makes RRT-CT and RRT-Blossom interesting in this light is that the minor concession of allowing receding edges essentially provides an additional strategy for escaping local minima, namely that of flood-filling. RRT typically gets stuck when its trees grow a number of prominent but "stunted" branches. These branches either probe dead-ends, or regions of the environment

[5] These two modes bear a strong resemblance to the EXPLORE and SEARCH components of the "Ariadne's clew" path planning algorithm[BATM93].

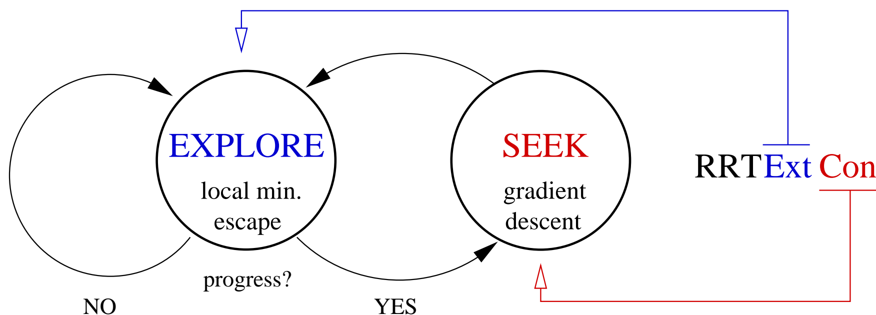


Figure 3.16: Dual-tree RRT as a simple Finite State Machine (FSM) which resembles the operation of a potential field planner.

where growth is only possible in unfavourable directions (e.g., away from x_{goal}, the other tree, or the central regions of the environment in general), and consequently they offer limited prospects for advancing the search. Yet their prominent nature—locally unrivaled jutting out into unexplored space—causes them to be repeatedly selected in the n_{near} selection step of most iterations. The confluence of these two traits results in the observed lack of progress. When receding edges are allowed, as in RRT-CT or RRT-Blossom, these problematic iterations are no longer wasted, but instead generate tree growth around said prominent branches. RRT-Blossom differs from RRT-CT in that it avoids redundant exploration, thereby making it possible to exhaust the local search space around the prominent branches, either finding an "escape route" that enables rapid further growth, or marking the whole area as a dead-end, and thus signaling to the planner to apply its efforts elsewhere. This flood-filling behaviour thus inexorably leads to a point where RRT gets "unstuck". Thus, in short, RRT-Blossom's improved performance comes from: 1) maintaining tree growth in iterations that RRT would otherwise waste; 2) in highly constrained neighbourhoods flood-filling tends to be a more effective strategy than a random walk for escaping local minima, and RRT's "random walks" are not even truly random—they are directionally biased by the distribution of x_{tgt} samples.

3.3.3 Drawbacks

Regression in nonviable space

The primary goal of RRT-Blossom is to maintain efficient and rapid rate of exploration, despite the difficulties presented by kinodynamic systems, but it only partially achieves this goal. Although it largely prevents re-exploration in the `live` and `dormant` parts of the tree, it fails to do so for the `dead` branches. In particularly difficult environments it is common for RRT-Blossom to explore and re-explore prominent nonviable regions with numerous (`dead`) branches. This is a direct consequence of the design decision to ignore `dead` branches when determining regression: as each attempt at exploration of such nonviable regions is blind to, and unhindered by the multitude of `dead` branches present, it proceeds along expected lines, and when shortly discovered to be `dead`

itself, it adds to the pile, to be likewise ignored by the next re-exploration attempt.

A number of stochastic remedies for this weakness are possible, but none of these provides a solid and conclusive solution, one that does not introduce further problems of its own. A foolproof solution would be to directly recognize the nonviable regions of state-space and then avoid exploring them, thus side-stepping the issue. This type of approach is the subject of the research presented in the following chapter.

Degeneration into depth-first search

RRT-Blossom is also sensitive to the distance metric used, like RRT, but in a different fashion. The difficulty crops up when the agent's dynamics result in sibling edges of a particular state being clumped close together, at least in the eyes of the metric. In such cases, when the first such edge is created out of a node, it automatically bars its siblings from being instantiated by virtue of the small inter-sibling distance triggering regression rule. If such clumping occurs throughout the whole of state-space then, in the extreme, RRT-Blossom will degenerate into a depth-first search, with each tree growing a single winding shoot. When the latter hits an obstacle or otherwise incurs failure, the tip of the shoot is terminated and another one springs forth a small distance back.

The culprit once again is the common L_2 metric, which misrepresents the true "cost to go": even though the clumped sibling edges are "close together", the cost to travel from one sibling endpoint to that of the other is actually quite large, and hence does not warrant being flagged as a regression. For example, even though two sibling edges might end up in very similar states, as would be common for a bike or car with limited steering, the actual "cost to go" between them is large because it will require the bike to first circle around.

It is worth noting that this does not necessarily lead to planner failure; in fact, solutions are still found, but it simply takes longer sometimes, and the solutions can be unnecessarily convoluted, because the single growing shoot is constantly tugged in random directions (i.e., by x_{tgt}). Since every other tree growth attempt comes about due to the SEEK stage of the planner, which pulls the shoot tips together, there is a natural tendency for them to converge, which eventually ensures a solution. No suitable resolutions to this issue present themselves at this time, other than using a more appropriate distance metric in the regression test.

3.3.4 Receding edge inclusion versus blossoming

RRT-Blossom consists of two key modifications to the RRT algorithm: 1) the blossoming operation, and 2) the inclusion of x_{tgt}-receding edges, combined with a mechanism to avoid tree regression. One might naturally wonder at this point whether these mechanisms are orthogonal, and what portion of the algorithm's gains is each of these features responsible for.

Blossoming can be made an orthogonal feature through a slight modification to its definition:[6] blossoming would be orthogonal if it instantiated only those edges (all of them) which reduce the distance to x_{tgt}, rather than instantiating any collision-free edges, as done currently. This variant could function independently of receding edge inclusion and regression filtering, but clearly such blossoming would instantiate fewer edges, making it less powerful. Considering the already minor role of blossoming to begin with (see discussion below), it seems this would be a feature of little value. On the other hand, the latter feature of receding edge inclusion and regression filtering could easily exist without blossoming, without any additional modifications.

The bulk of RRT-Blossom's gains actually come from the latter mechanism, the inclusion of receding edges, rather than from blossoming, despite the name of the algorithm[7]. It is difficult to quantitatively characterize the relative contributions as these are highly dependent on the agent and the environment. However, for many problems, such as the specific experiments shown in this chapter, the effect of blossoming is likely very minor (i.e., less than 5% of gains). Since blossoming essentially aids in "consuming" unexplored space *quicker*, its largest gains are seen in problems where finding a solution typically hinges on exhaustive exploration of large volumes of free-space. Put another way, blossoming essentially speeds up the "flood-fill" rate, thus it expedites planning most when there are deep, difficult to traverse "local minima" in the environment. For example, the inclusion of blossoming was observed to reduce runtimes for a single-tree planner (with receding edges and regression filtering) in the tunnel environment for the kinematic point agent by 25%. Since this feature occasionally produces such useful gains, without incurring a time penalty the rest of the time, it has thus been retained in the algorithm.

3.3.5 Control considerations

Discretization of $\mathcal{U}$

The blossoming operation as well as the inclusion of x_{tgt}-receding edges in RRT-Blossom both require the discretization of the control space $\mathcal{U}$[8]. The algorithm is thus naturally not applicable to systems where this is not feasible. Furthermore, the discretization can be problematic due to resolution issues. If $\mathcal{U}$ is poorly sampled, whether due to sparseness or poor distribution of the samples, it may be impossible for RRT-Blossom to find a solution, even though one exists. This is the fault and consequence of a poor choice for the discretization rather than a deficiency specific to

---

[6] Since the very definition of blossoming in this chapter mandates the presence of the second mechanism.

[7] One may thus wonder why the algorithm is named after the less powerful mechanism. This is due to historical accident: blossoming (albeit with receding edges) was the first idea to be explored, and it was the rampant re-exploration of the resultant trees that prompted the development of regression filtering. As the name has been already used in a publication, before the relative significance of the mechanisms has been fully investigated, the name became "frozen".

[8] Technically, the latter could be implemented without discretization, by randomly choosing $u \in \mathcal{U}$; in many cases u will result in a receding edge. Normally this strategy leads to gross re-exploration and poor progress, but the regression filtering would avert this.

the RRT-Blossom algorithm; in fact, any other planner, even a resolution complete one (cf. 2.1.6), would have the same problem given the same discretization. For example, if we were to discretize the kinematic point's direction of travel to the four cardinal directions, it would be impossible to solve problems which require the traversal of narrow *curved* corridors: there is no way for the agent to span such a corridor using axis-aligned motions (assuming the corridor is too narrow relative to the time-step to allow zig-zagging). The choosing of a sufficient yet not too fine a discretization, which would be unnecessarily costly, is a common problem in such discretized algorithms, although sometimes it is possible to sidestep the issue through adaptive means (e.g., [LL06]). Even the regular RRT algorithm must contend with this issue to a degree, in the time dimension, since too large a choice of time-step can make almost any problem unsolvable.

Scalability

The experiments in this chapter employed first-order (kinematic point, car) and second-order (bike) systems. Each was controlled using a single steering input, with the kinematic agent using zero-order (positional) control, and the car and bike using first-order (integral) control. The experiments in the next chapter demonstrate RRT-Blossom results, in a baseline role, for an inertial point agent which uses second-order (double integral) control. In general, higher-order control is expected to lead to better results with RRT-Blossom since such control schemes tend to lead to more complex and constrained search-spaces, where the planner's "flood-filling" can play a larger role.

RRT-Blossom is unlikely to scale well to problems with many-dimensional control spaces. The control inputs in this chapter have been one-dimensional: just steering control. Applying RRT-Blossom to systems with higher-dimensional control inputs presents two problems: 1) the resolution problem is exacerbated, making it even more important to get right the size and distribution of the discrete control input set, lest the problem becomes unsolvable; 2) this in turn is likely to cause the control set size to scale exponentially as dimensionality of $\mathcal{U}$ is increased, resulting in rapidly degrading performance of RRT-Blossom since on each iteration all the discrete control actions must be checked. It should be noted though that this behaviour is a consequence of discretization of $\mathcal{U}$, and thus would be shared by any RRT planner which uses the same strategy for handling actuated agents.

3.4 Conclusion and future work

The main contributions of this chapter are the introduction of a new RRT-based planner that has significantly improved performance in highly constrained search-spaces (due to bottlenecks in environment or state-space), and has planning times that decrease once the problem surpasses some critical amount of constraint.

One weakness of RRT-Blossom is how it handles critical bottlenecks in the environment that have become blocked by extant search tree edges. If the block is due to a viable, `dormant` tree edge, the planner will become aware of the problem only when the whole tree reaches `dormant`-deadlock. Although this situation was rare in the tests performed, this is clearly a weak, ad hoc way to handle the problem. Likewise, blocks due to nonviable edges are handled by simply ignoring all nonviable edges during regression tests. This solves the immediate problem, but at the cost of losing regression avoidance in nonviable space. Although the later addition of viability filtering largely eliminates this problem, one cannot help but wonder if the overall problem cannot be handled in a cleaner, more generic way, one that does not need to distinguish between viable and nonviable blocking edges.

Another issue worth investigating is the degeneration of RRT-Blossom into a depth-first search. This occurs when, for most states of the agent, all the possible trajectories out of a state are clumped close together (e.g., those for a car with a very limited steering range) The problem clearly lies with the distance metric used to test for regressions. Due to lack of better choices, one usually uses the L_2 distance metric (i.e., "Euler distance"), which leads to the above behaviour. A better metric would give better results, but it is not clear where to draw the line between regression and sufficiently separated siblings for such low-manoeuvrability systems, which in turn makes it difficult to arrive at a better metric.

It is likely that the regression test itself could be sped up in a number of ways. The obvious first step would be to employ *kd*-trees or a similar method to improve the Nearest Neighbour (NN) lookup for a potential new node; the current implementation uses an $O(n)$ naive linear search. A more aggressive optimization is possible if the regression test is performed within the context of an RRT iteration. According to the RRT-Blossom algorithm, the x_{near} node is chosen by performing a Nearest Neighbour search for the x_{tgt} state, and the regression test is typically applied to a potential new edge from this node. The regression search can be expedited by limiting it to just the set of nodes returned earlier, the nearest neighbours of x_{tgt}. Since this information has been computed earlier, there is no additional cost to simply retrieving it, and hence the regression check would then be $O(1)$.[9] Although this is not guaranteed to always find a regression if it exists, it should do so most of the time. This is because the set of NNs of x_{tgt} will usually include most NNs of x_{near} as well, by virtue of x_{near} being the node closest to x_{tgt}. At the same time, the potential new edge out of x_{near} is unlikely to stray far from x_{near}, hence the above set of nodes is likely to also include most NNs of the new proposed endpoint, the one being tested for regression. The approximation would break down if the NNs of x_{tgt} are equidistant from it, and distributed equally in all directions (e.g., consider the case where all k NNs are spread uniformly on a circle centred at x_{tgt}).

[9]It is assumed that the NN check always results in a constant, or nearly constant number k of nearest neighbours.

It would also be interesting to attempt combining RRT-Blossom approach with traits from other RRT variants. In particular, it should be possible to improve runtimes by including collision tendency tracking from [CL01b], as well as incorporating"local" trees from [Str04], once it has been properly extended to handle the directional nature of the trees in kinodynamic problems.

Finally, it might be desirable to work out more exactly the level of completeness of this algorithm. In the kinematic case, the current implementation is probabilistically complete since upon deadlock, when free-space has been exhausted under the regression prevention mechanism, the method degenerates into plain RRT. For agents with differential constraints, it is not clear whether RRT-Blossom could be made resolution complete, in the spirit of [CL].

Chapter 4

Motion planning with local viability models

In general, current motion planners rely solely on collision detection for sensing the surrounding environment, which leads to very tactile and myopic perception. As a result, such planners cannot detect, learn, nor reason about commonly occurring patterns or scenarios, and instead end up solving the same problem"from scratch" each time. For example, there is usually no substantial difference between the first and the hundredth time that a typical planner tries to parallel park a car. In contrast, the ideal motion planner would notice general patterns in the problem space, and through observing its own solutions, it would learn corresponding general motion strategies which could then be applied directly the next time a similar problem is encountered.

This chapter takes the first step toward this ideal. It proposes a simple way of exploiting prior experience to expedite the motion planning process. The approach relies on first observing and modeling of *undesirable* states, ones that have a low likelihood of leading to a solution. Since a key goal of this learning process is that the gained knowledge should be applicable to a wider range of problems (i.e., learned skills should be transferable), the models are parameterized using a local "perceptual space". The latter is achieved by equipping the agent with a bank of virtual sensors which then describe the agent's state in terms of its local context (i.e., obstacles). In this way we thus endow the motion planner with "sight" and "intelligence" (in the sense of having the ability to learn).

The idea itself is very general and can be exploited by many planners. Thus, although we show results using RRT-Blossom, similar improvements should be obtainable, for example, using RRT-CT; this remains an interesting direction for future work. It is also worth noting, however, that a complete treatment of a general and widely applicable idea such as this would require a more exhaustive study of the various facets and issues (e.g., alternate implementations and their relative merits). Since this lies outside the scope of the thesis, this work is perhaps best seen as a

feasibility study, demonstrating the workability of the approach and confirming its usefulness.

Our approach, then, consists of three key parts: 1) acquiring training examples of undesirable situations; 2) deriving a model from this data; and 3) exploiting this model to expedite the planning process by terminating search branches that stray outside the desirable space. These parts are described further in Sections 4.3–4.5.

4.1 Filtering nonviable states

There are many ways to define the desirability of a particular agent state. In the present context of expediting motion planning, it is useful to define a desirable state as one which has an attractive likelihood of leading to a solution; conversely, an undesirable state would then be one with a low likelihood of effecting such progress.

In this chapter we adopt a simpler but stronger definition that avoids probabilistic entanglements: here, an undesirable state is one that has *zero* likelihood of leading to a solution. A prominent set of states that fits this description is that which corresponds to nonviable[1] space. A nonviable state cannot lead to a viable one by definition (it would be viable if it did), hence if x_{goal} lies in viable space, a nonviable state cannot possibly contribute to a solution. The special case of a nonviable x_{goal} can be side-stepped, and is discussed in Section 4.8. In short then, we propose expediting the planning process by avoiding exploration of nonviable regions of the search space.

4.2 Modeling viability

Viability can be modeled in a number of ways. The next chapter, Chapter 5, looks at modeling viability globally, whereby the models are parameterized by the agent's global state x, and are therefore tightly bound to the environment on which they were trained. In this chapter we look at local viability models, which are essentially parameterized by the local context of the agent. Such local models are preferable since they are applicable to a wider range of similar-yet-different environments, not just the original one used to train the model. Local models also often capture the agent's viability in a more efficient and compact form.

4.2.1 Combined system state

Traditionally motion planners treat the agent's global state, x, and the state of the environment e, as separate entities, and only bring them together during collision checks.[2] But this split is superfluous, and even an impediment to describing the local context of the agent, where x is

[1] We are referring here to the standard definition of viability, rather than goal-viability (see §2.2.2).

[2] We construe this state to contain enough information about the environment's geometry that, together with x, it is sufficient to perform a collision check. For dynamic environments e is time-varying.

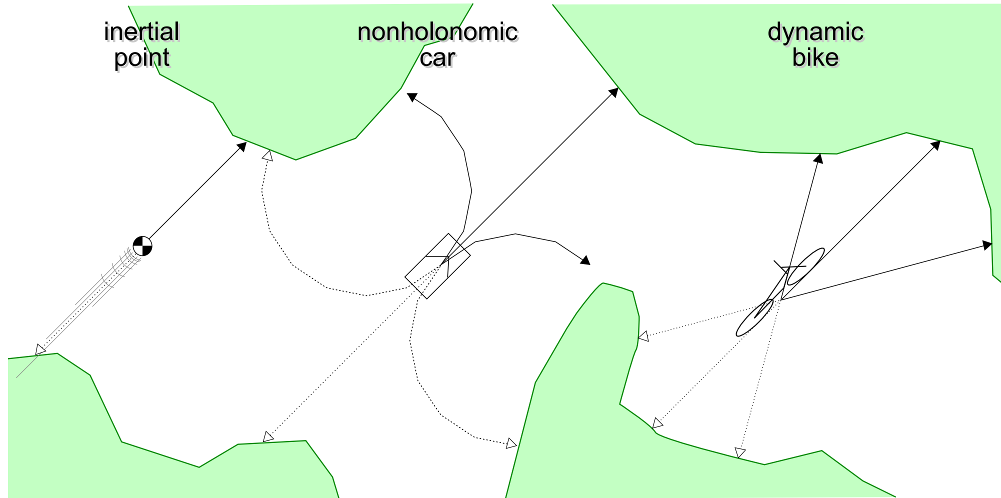


Figure 4.1: Virtual sensors for various agents. Lines indicate (virtual) agent-mounted range sensors used in planning (dotted lines indicate sensors used with the x_{goal} tree). They are discussed in Section 4.7.

meaningless without e, and vice versa. This leads us to define

$$x^+ = (x, e) \qquad \text{and} \qquad x^+ \in \mathcal{X}^+, \tag{4.1}$$

where x^+ is the full *system state*, and $\mathcal{X}^+ = \mathcal{X} \times \mathcal{E}$ is the system state space resulting from the Cartesian product of the agent and environment state spaces. The combined system state is also a more useful representation for problems with dynamic environments.

4.2.2 Sensors

Any local description of the agent's state must, in the end, somehow describe it "through the agent's eyes", in terms relative to the surrounding environment and features. A planner can be given such "sight" by fitting it with a number of virtual *sensors* that perceive the world relative to the subject. As an example, Figure 4.1 shows the sensors that were used in our experiments. More formally, a sensor σ is a function which maps the subject's state and the environment geometry to a scalar value:

$$\sigma(x^+) : \mathcal{X}^+ \to \mathbb{R}. \tag{4.2}$$

For a particular system state $x^+ \in \mathcal{X}^+$, one can compute all the sensor values and concatenate them into a single vector

$$s = \big(\sigma_1(x^+),\ \sigma_2(x^+),\ \ldots\big), \tag{4.3}$$

which we refer to as the *sensory state*. The set of all possible sensory states forms the *sensory space* $\mathcal{S}$ (also known as an *information space*), and thus $s \in \mathcal{S}$. The sensory state is a much better local descriptor of the agent's state, but is insufficient by itself. Often one must also include a number of elements from the state x, those that pertain to the agent's internal configuration (e.g.,

the lean angle for a bike), since these variables often have a profound effect on the importance of the surrounding environment features, emphasizing some while marginalizing others.

4.2.3 Locally situated state

We thus define the *locally situated state* of the agent

$$\lambda = (s, \hat{x}), \qquad \text{where} \qquad \lambda \in \Lambda \tag{4.4}$$

The vector $\hat{x}$, which is a subset of elements of $\vec{x}$, consists of relevant state variables of the agent that are otherwise not accounted for in the sensory state s, and Λ is the *locally situated state space*. At most, $\hat{x}$ is the position- and orientation-independent portion of the agent's state $\vec{x}$, but often smaller.[3]

For example, in the kinodynamic bike system,

$$\begin{aligned} \vec{x} &= (x, y, \theta, \phi, \dot{\phi}) \\ s &= (\sigma_L, \sigma_F, \sigma_R) \\ \lambda &= (\sigma_L, \sigma_F, \sigma_R, \phi, \dot{\phi}) \end{aligned} \tag{4.5}$$

where (x, y) is the bike's position, θ is its orientation, and ϕ is its lean angle, while $(\sigma_L, \sigma_F, \sigma_R)$ are the three sensors.

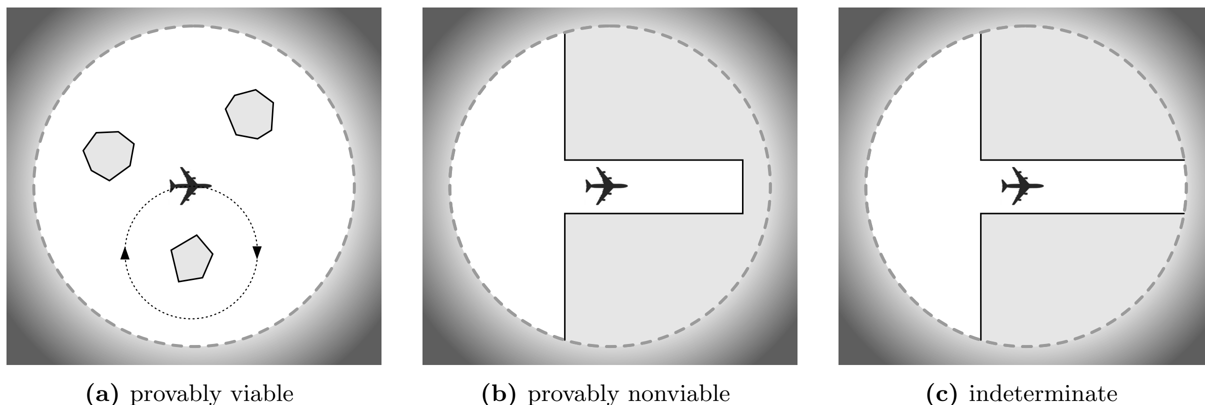
It is worth noting that such localized perception of the system state has been explored in control literature, yielding interesting and robust behaviour-based AI and control [Bra84, Bro86, Ark98].

4.2.4 Local viability

Our approach thus relies on constructing a model that, given a locally situated state λ, labels it as either viable or nonviable. This localization of viability carries important consequences however. The key side-effect is that the resultant model is only an approximation of true viability, because viability is not *locally decidable*. That is, in the general case it is not possible to conclusively assess the viability of a state based on limited, local knowledge of the environment. For example, consider an airplane flying down a very long, narrow corridor which recedes into darkness (i.e., outside sensing range), as shown in the rightmost case in Figure 4.2. It is not immediately known whether the airplane is viable or not, since it all depends on how the corridor terminates: if it exits onto a wide open area then the plane is viable; if it turns out to be a dead-end alley, then the plane is nonviable (the corridor's narrowness prevents the plane from turning around).

Theoretically, judging the viability of an agent's state based solely on local information leads

[3]For some agents it is useful to post-process $\hat{x}$, such that $\lambda = (s, f(\hat{x}))$, in order to reduce its dimensionality or make it more amenable to learning.



(a) provably viable **(b)** provably nonviable **(c)** indeterminate

Figure 4.2: The three possible cases when judging viability using the local neighbourhood.

to one of three cases, as illustrated in Figure 4.2:

1. x **is provable viable:** x is viable since there exists a trajectory within visible range which loops back to x without incurring a collision, thus allowing safe operation indefinitely.

2. x **is provably nonviable:** collision is unavoidable, with all subsequently reachable space fully within visible scope.

3. **indeterminate case:** collision-free trajectories exist which take the agent outside of the visible range; whether x is viable or not depends entirely on what happens to these trajectories outside the visible scope.

Since the oracle is a binary valued function, it must therefore label each state as "viable" or "nonviable", even in the theoretically indeterminate case. In practice the oracle thus ends up guessing at the true viability status of a state in such cases, based on prior experience. If the training environments featured similar situations and the exit paths tended to lead into open areas, then it will guess that the state is viable; conversely, if prior experience consists of many cases of long blind alleys, it will guess that the state is nonviable.

Of course, in practice the oracle does not discern between these three cases; as a classifier, it is only aware of feature vectors (i.e., the locally situated states) and the labels associated with them. In addition, its knowledge of the local environment is usually far more sparse than the ideal illustrated in Figure 4.2, typically consisting of only a handful of real values (i.e., the sensor readings). Thus even in cases where the agent state is in theory provably viable or nonviable, the oracle's operation still amounts to guessing, due to the sparseness of its local knowledge of the environment. The critical redeeming trait here is that, for most systems, the oracle's adeptness at guessing viability is often very good, assuming a well chosen set of sensors is used.

4.3 Acquiring training data

If an adequate external model of the agent's viability is available, one can completely forgo the collection and training stages, and use the model directly. Unfortunately most nontrivial dynamical systems do not have easily obtainable analytic viability kernels, nor prior empirically derived ones.

Viability data can be collected from prior motion planner runs, or can be provided by an external source. Specifically, we are interested in collecting samples of locally situated states λ that have been annotated with their viability status. This status is often nontrivial to derive. In order to prove a state is nonviable one must demonstrate that all progeny lead to collision or failure, while to prove a state is viable one must show that at least one failure-free trajectory exists out of the state. Both these tasks pose a challenge. The nonviable test requires an exhaustive search through all progeny of a state, which can be very expensive for problems with nontrivial $\mathcal{X}_{ric}$ ("region of inevitable collision", the nonviable subset of $\mathcal{X}_{free}$), and does not even terminate if state being tested is viable. Likewise, the viable test is problematic since demonstrating an infinite trajectory, whether viable or not, is generally not practical. Although in many cases one theoretically could demonstrate a trajectory that loops or reaches another known viable state (thus proving by inference that an infinite trajectory exists), in practice the equations of motion of most agents make it difficult to find a sequence of control actions that will hit a specific predetermined state exactly, or even a single state from some finite set of candidates. How best to resolve this problem is dependent on the nature of the training data; Section 4.6.2 describes our approach.

4.4 Modeling

The collected training data is used to derive a model of agent's local viability, the *viability oracle*

$$\Omega_v(\lambda) : \Lambda \rightarrow \{\texttt{viable}, \texttt{nonviable}\}. \tag{4.6}$$

In dual- or multi-tree planners one of the trees is usually grown in reverse-time (i.e., using reverse simulation; for example, x_{goal} tree). In such trees the agent generally moves backwards[4], so the sensory framework must likewise be flipped, since it is the obstacles behind the agent which now determine its viability. These flipped sensors are shown using dotted lines in Figure 4.1.

Since a different set of sensors is used under reverse-time simulation, a corresponding second viability model needs to be derived for filtering of such trees. In general, we thus train a second, reverse-time oracle

$$\Omega_{v_{rev}}(\lambda_{rev}) : \Lambda_{rev} \rightarrow \{\texttt{viable}, \texttt{nonviable}\}. \tag{4.7}$$

This second model captures *backward viability*; in contrast, Ω_v captures *forward viability.* These

[4]For agents which can normally move forwards and backwards, these directions take on the opposite meaning.

names were chosen to mirror similar terms in reachability, namely the forward and backward reachable sets. Backward viability of an agent captures the general reachability of the state, rather than predicting the avoidability of future collisions. An example of a state that is not backward viable is an airplane in flight, with a building immediately behind it: although the airplane can presumably maintain safe operation from this point onwards (i.e., forward viable), it is impossible for it to have gotten to this state, since this would have involved first flying through the building. It is the task of the backward-facing sensors to detect precisely such scenarios.

In summary then, the forward-simulated trees (e.g., x_{init}) thus use the standard oracle Ω_v, which is trained on λ derived from forward-facing sensors, while backward-simulated trees (e.g., x_{goal}) use the alternate, backward-viability oracle $\Omega_{v_{rev}}$, which is trained on λ_{rev} computed from the backward-facing sensors.

In some special cases Ω_v may be applied to reverse-time trees, thus obviating the need for a second model. This is true, for example, for systems such as a Dubins car, where any valid agent trajectory can be equally traversed in the opposite direction by merely switching which way the agent faces. In kinodynamic systems a state vector's velocity elements dictate the instantaneous agent direction (and thus the direction of any valid trajectory passing through the state), but usually some simple transformation of the viability model, such as an inversion along one of the axes, will render it valid and usable for the reverse-time trees.

4.4.1 Reachability viewpoint

Reachability, a concept strongly related to viability, can be helpful in better understanding the filtering process. In reachability terms, the basic RRT algorithm grows a tree from x_{init}, which corresponds to the *forward reachable set* of x_{init}, as well as a tree from x_{goal}, using reverse-time simulation, which corresponds to the *backward reachable set* of x_{goal}. The filtering provided by Ω_v and $\Omega_{v_{rev}}$ reduces the two trees to strict subsets of the corresponding initial sets.

In contrast to the above, the indirect goal of the proposed viability filtering is to ensure that all the states in both trees are simultaneously in the forward-reachable set of x_{init} and the backward-reachable set of x_{goal}. That is, the filtering intends to constrain the planner to only those states which could feasibly lie on a solution trajectory, in that they are reachable from x_{init}, and can themselves then reach x_{goal}. In some sense then, Ω_v attempts to capture the backward reachable set of x_{goal}, while $\Omega_{v_{rev}}$ attempts to capture the forward reachable set of x_{init}.

4.5 Exploiting viability

Given a trained viability oracle, it is trivial to instrument an arbitrary planner for viability filtering: one merely replaces the default call to a collision or failure checking routine, named `is_collision()` here, with a call to `is_nonviable()`:

```
1: function IS_NONVIABLE(x+)
2:     if is_collision(x+) then
3:         return True
4:     s ← σ1(x+), σ2(x+), ...
5:     x̂ ←extract_internal_state(x+)
6:     λ ← (s, x̂)
7:     return ¬ Ω_v(λ)
```

The above function queries `is_collision()` because the viability model may not be perfect, and thus could occasionally admit an in-collision state, if used unaided. In a dual-tree planner implementations one would generally need a second, analogous routine, `is_nonviable_backwards()`, that would be used with reverse-time trees (e.g., x_{goal} tree), and which would use the reversed sensor set as well as the $\Omega_{v_{rev}}$ model.

4.6 Implementation details

4.6.1 Training trajectories

Collection of viability-annotated training samples can be done in a number of ways. Ideally the planner should "bootstrap", starting out with no viability knowledge and progressively building it up by examining its own search trees and solutions for novel states and incorporating them into the oracle. However, such online bootstrapping is problematic as we describe in Section 4.8. Instead, we collect samples of viable states from very long random-walk trajectories. These are created by applying a random control action at each time step and backtracking whenever the agent encounters failure. Heuristic methods modulate the frequency and amount of backtracking since a naive brute-force approach, one which exhausts all possible control combinations from a particular state before backtracking, is often very inefficient.

4.6.2 Deciding a sample's viability

As stated earlier, to prove a state is viable one generally would have to demonstrate in some way an infinite collision-free trajectory emanating out of the state, which is usually not practical. We thus adopt an approximation which identifies states which are very likely to be viable. Specifically, we pronounce a state x as viable if we can demonstrate a *finite* failure-free trajectory emanating out of x, of duration T_h or longer. This effectively presupposes that the failure modes of the subject are confined to durations shorter than the above *time horizon* T_h (10 s in our implementation); that is, we assume that T_h exceeds the maximal "depth" of the region of inevitable collision $\mathcal{X}_{ric}$. Unless T_h is badly misjudged, any error introduced with this approximation (i.e., contamination of the experience pool with nonviable states) is insignificant when compared to other sources of error in our approach.

Using this convention, any search tree or solution trajectory in the training datasets is readily converted to a collection of sample viable states by simply discarding T_h-worth of motion from their tail ends. When collecting training samples for the reverse-time viability model, the trajectories must be trimmed from their front end instead. In both cases the resultant collections of states must then also be locally situated (i.e., mapped from x^+ to λ) before being used to train the (local) viability oracles.

4.6.3 Modeling

The random-walk trajectories described earlier offer little help in identifying nonviable states, but can be used to identify viable ones, as outlined above. This means that a one-class classifier must be used to derive the viability model since all the samples are of the same class (i.e., "viable"). We use the one-class Support Vector Machine (SVM) classifier in the libSVM[CL01a] library.

The SVM learning process used is straight-forward. Prior to learning, the training samples $\lambda \in \Lambda$ are first column-standardized and then additionally scaled to emphasize certain feature dimensions. That is, we preprocess each element λ_i of a training sample λ using

$$\lambda_i \leftarrow c_i \, \frac{\lambda_i - \mu_i}{\sigma_i}, \tag{4.8}$$

where λ_i is the i'th feature in vector λ, c_i is a scaling coefficient for said feature, while μ_i and σ_i are the corresponding mean and standard deviation, respectively, taken over all values of the feature.

4.7 Experiments

4.7.1 Agents

We apply the viability filtering method to three agents. Only the forward-time sensors are described here; for the reverse-time trees the obvious analogues are used. Both sensor sets are illustrated in Figure 4.1.

Inertial point

The "inertial point" robot is a point-mass with inertia that has four constant-force thrusters mounted in the four cardinal directions $\{\mathcal{N}, \mathcal{E}, \mathcal{S}, \mathcal{W}\}$, as shown in Figure 4.3. The agent thus has a fixed orientation and cannot rotate. During operation the agent is required to have at all times one, and only one of its thrusters on. Alternatively one can envision the agent as being equipped with only a single thruster that is always 'on', and which can be rotated to the four cardinal directions in a discrete and instantaneous manner. The thrusters exert a constant force which yields an acceleration of $0.5\,\mathrm{m/s^2}$.

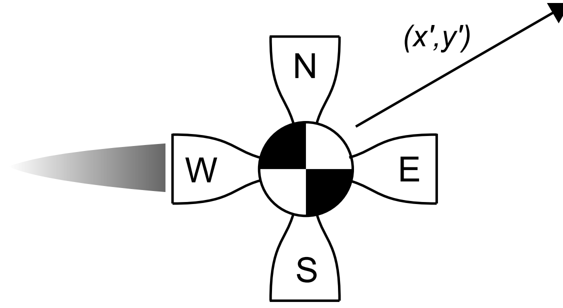


Figure 4.3: The "inertial point" agent: one, and only one of the four thrusters must be 'on' at all times.

The agent has upper and lower bounds on its velocity: $0.5\,\mathrm{m/s} \leq \|(\dot{x}, \dot{y})\| \leq 5\,\mathrm{m/s}$. A small lower bound was found useful in expediting the planning process with all planners; without it the planners spent a lot of time populating the free-space with very dense subtrees rife with near-zero velocity nodes, a consequence of the ease with which antagonistic thruster pairs can cancel out each other's accelerations.

The subject's state, control actions, sensory state, and locally situated state are:

$$\begin{aligned} \vec{x} &= (x, y, \dot{x}, \dot{y}) \\ \mathcal{U} &= \{\mathcal{N}, \mathcal{E}, \mathcal{S}, \mathcal{W}\} \\ s &= \sigma_v \\ \lambda &= (s, \|(\dot{x}, \dot{y})\|) \end{aligned} \tag{4.9}$$

where sensor σ_v measures the distance from the agent to the nearest obstacle along the velocity vector $(\dot{x}, \dot{y})$.

Nonholonomic car

The nonholonomic car used here is nearly identical to that of the previous chapter (see §4.7.1), but less manoeuvrable, with a smaller maximum steering angle $\psi_{max} = \sin^{-1}(1.275/5)$, giving a minimum turning radius of exactly 2.5 metres. The relevant parameters are:

$$\begin{aligned} \vec{x} &= (x, y, \theta) \\ \mathcal{U} &= \{-\psi_{max},\ 0,\ \psi_{max}\} \\ s &= (\sigma_{L_{wh}},\ \sigma_F,\ \sigma_{R_{wh}}) \\ \lambda &= s \end{aligned} \tag{4.10}$$

where θ is the car's orientation, σ_F is a forward-facing rangefinder, while $\sigma_{L_{wh}}$ and $\sigma_{R_{wh}}$ are the left and right "whiskers". A *whisker* returns the distance to the environment along a particular path. It can be useful to consider curved whisker-paths for robots when this reflects the nature of their motion. In practice, we approximate such curved paths using a small set ($n = 8$) of straight line

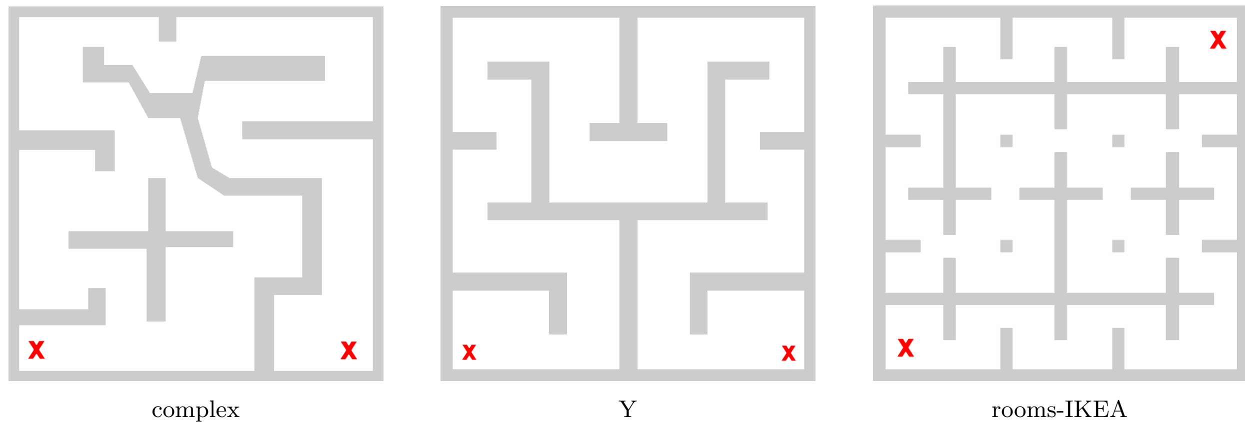


Figure 4.4: Example problem environments tested. In each case the agent is asked to navigate from the left 'X' to the one on the right.

segments for computational efficiency. The car's whiskers correspond to the two extremal steering actions applied for the duration of a 180° turn, as shown in Figure 4.1.

Kinodynamic bike

The least trivial agent used is the kinodynamic bike from the previous chapter (see §4.7.1), which has the following parametrization:

$$\begin{aligned} \vec{x} &= (x, y, \theta, \phi, \dot{\phi}) \\ \mathcal{U} &= \{-\psi_{max},\ -\tfrac{1}{2}\psi_{max},\ 0,\ \tfrac{1}{2}\psi_{max},\ \psi_{max}\} \\ s &= (\sigma_L,\ \sigma_F,\ \sigma_R) \\ \lambda &= (s, \phi, \dot{\phi}) \end{aligned} \tag{4.11}$$

where θ is the bike's orientation, ϕ its lean angle, σ_F is again the forward-facing rangefinder, while σ_L and σ_R are rangefinders deflected by 30° left and right, respectively.

4.7.2 Environments

Figure 4.4 depicts the environments used in the experiments. They all measure 30 m by 30 m. The "complex" environment was included to provide a common ground between these results and those of the previous chapter. They "Y" environment provides more twists and turns, but has very similar characteristics to "complex", especially its winding corridor nature and the general amount of room to move. Finally "rooms-IKEA" provides a more constrained and difficult environment, with some novel features not found in "complex", such as narrow doorways and small detached obstacles in the middle of the rooms. These novel features provide a greater challenge to the viability models which were trained only on the "complex" environment.

4.7.3 Implementation details

The implementation was done using a test platform written in Python 2.4, on an Intel Core 2 Duo E6600 (each core is a Pentium IV 2.4 GHz); although two cores were available, only one core was used when measuring runtimes. The operating system was Windows XP (SP2). We chose RRT-Blossom as the base algorithm to which viability filtering was applied. To partially offset the slower speed of an interpreted language a number of optimized modules were used: "psyco", the C-implemented libSVM library (through the accompanying Python bindings), "scipy", and other C-implemented modules of mathematical nature. Collision checking was done using a naive Python implementation.

4.7.4 Learned models

This section describes and discusses the derived models.

Inertial point

Figure 4.5 on page 77 illustrates training data and the derived model. The training data consists of a single random-walk in the "complex" environment. Its duration is 100,000 seconds, using a $1/2s$ time-step, yielding a little less than 200,000 samples (due to T_h-trimming). The SVM learning parameters are: kernel = RBF, $\gamma = 1$, $\nu = 0.005$, $c = (1, 1)$.

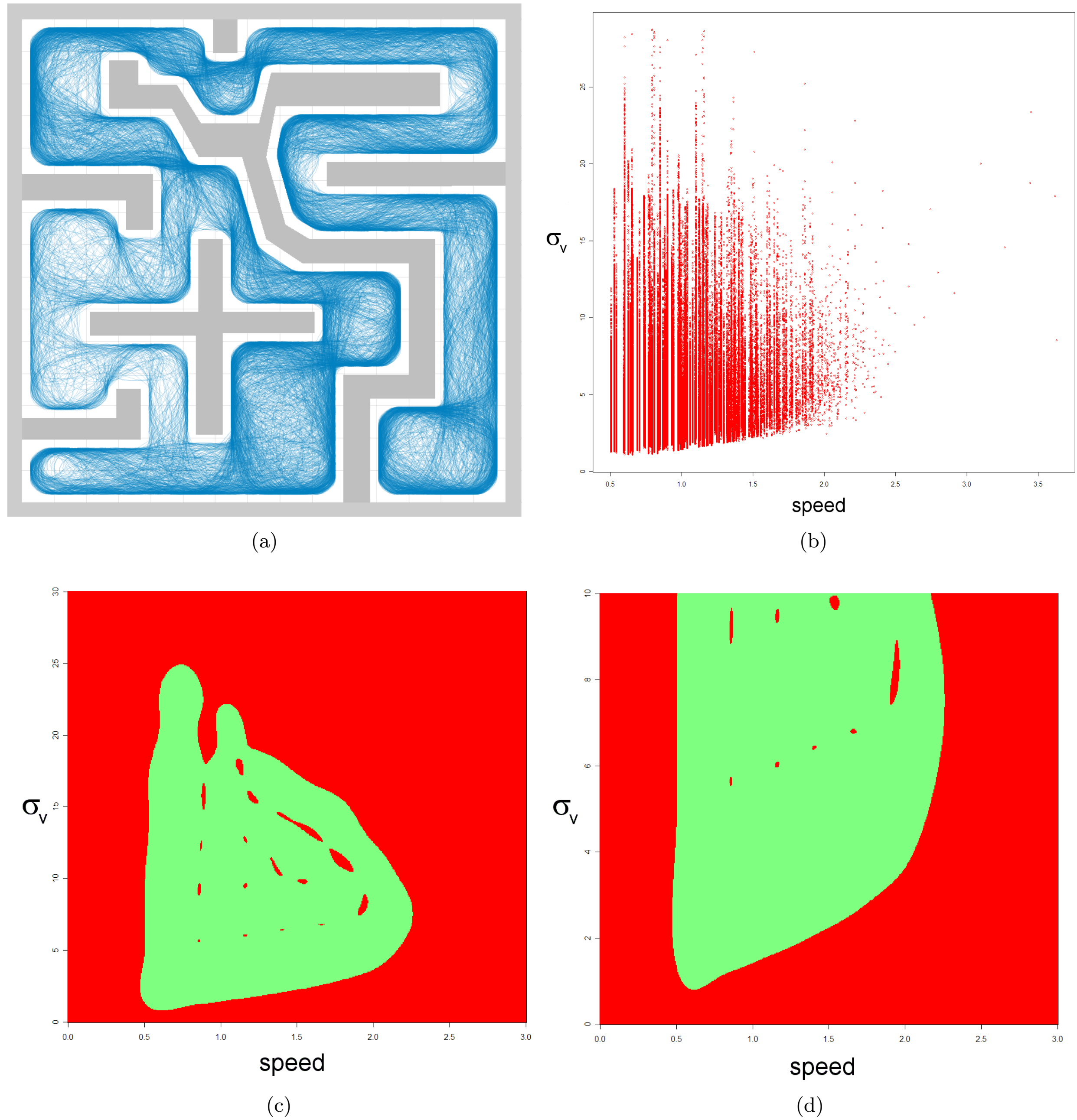
There are two general trends in the training data: 1) the hard lower bound that increases as velocity (x-axis) goes up, is an expected trend, indicating that the larger the velocity, the larger the forward clearance that is usually needed to decelerate if no other escape avenues are open; 2) the much softer upper bound which decreases with speed is a fictitious bound resulting from the general lack of large open straightaways in the environment, which limits the agent's ability to build up higher speeds.

The distribution of λ in Figure 4.5(b) appears striated. This is due to velocity being limited to a relatively small set of discrete values, a consequence of the discrete nature of allowable control actions and the velocity bounds imposed.

Car

Figure 4.6 on page 78 illustrates the training data and the derived model. The training data consists of a single random-walk in the "complex" environment. Its duration is 100,000 seconds, using a $1/2s$ time-step, thus yielding a little less than 200,000 samples (due to T_h-trimming). The SVM learning parameters are: kernel = RBF, $\gamma = 1$, $\nu = 0.01$, $c = (2, 1, 2)$.

It is interesting to note how certain areas admit little choice in navigation (e.g., the well defined loops in various nooks), and how, unlike for the other agents, it covers the free-space sparsely and unevenly. This suggests that the environment is surprisingly constraining for the agent, likely a



(a) (b) (c) (d)

Figure 4.5: Viability model for **inertial point** agent; **(a)**: random walk trajectory used for training; **(b)**: distribution of resultant locally situated states λ; **(c)**–**(d)**: learned viability model

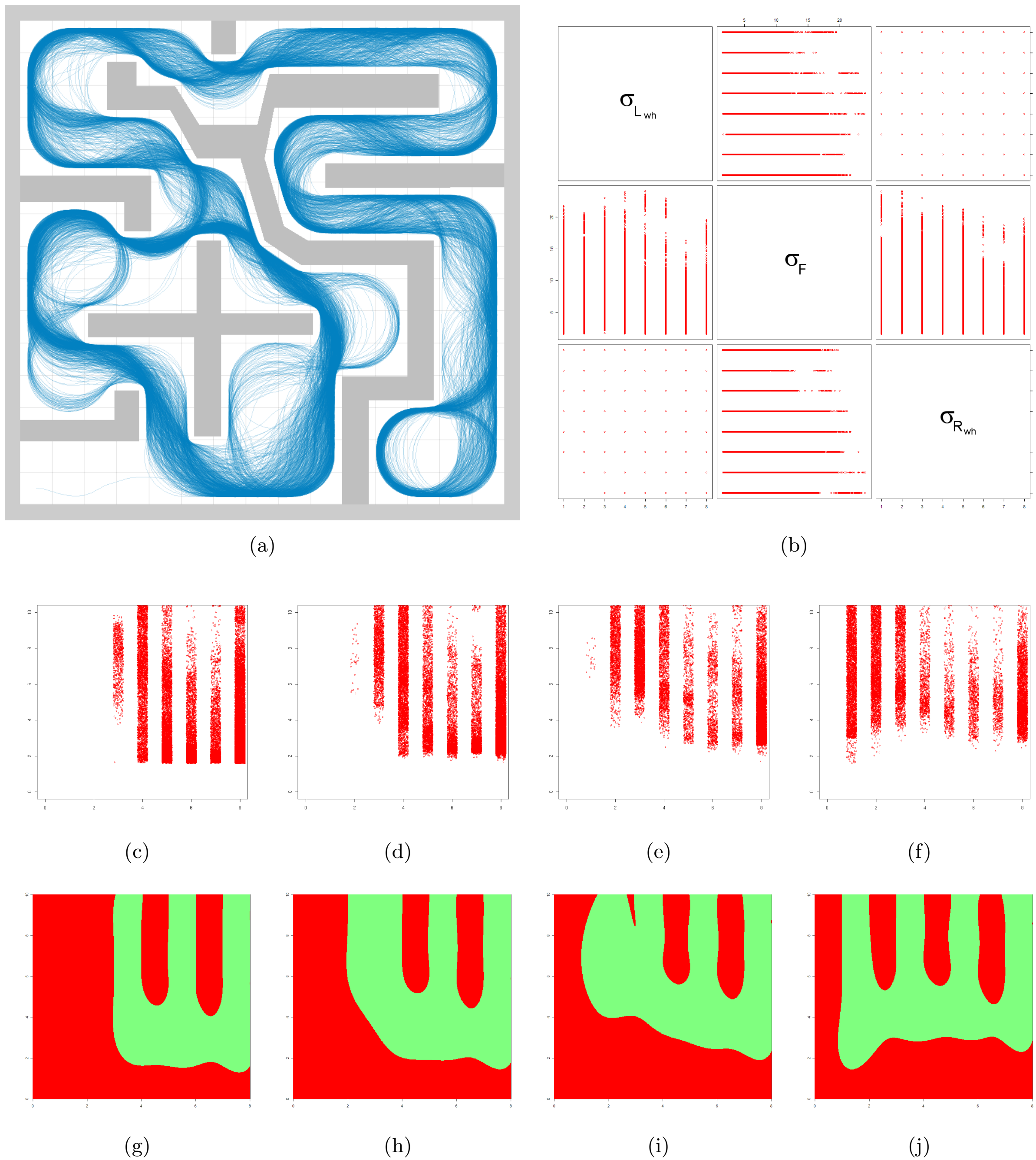


Figure 4.6: Viability model for car agent; **(a)**: random walk trajectories used for training; **(b)**: distribution of resultant locally situated states λ; **(c)–(f)**: cross-sections of the λ distribution around various values of the left whisker sensor ($\sigma_{L_{wh}} = \{1, 2, 3, 4\}$, respectively). The cross-sections plot σ_F on the y-axis versus $\sigma_{R_{wh}}$ on the x-axis. The whisker sensors can take on only integral values hence lateral jitter was applied for better clarity; **(g)–(j)**: equivalent cross-sections in learned model

consequence of the limited choice of control actions, the relatively small maximum turning angle, and the somewhat large discrete time-step used ($1/2s$).

The very discrete nature of the λ distributions in Figure 4.6(b) is expected since the two whisker sensors are discrete, and can only take on values in $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, due to their 8-segment piecewise-linear construction.

Bike

Figure 4.7 on page 80 illustrates the training data and the derived model. The training data consists of a single random-walk in the "complex" environment. Its duration is 100,000 seconds, using a $1/2s$ time-step, thus yielding a little less than 200,000 samples (due to T_h-trimming). The SVM learning parameters are: kernel = RBF, $\gamma = 1$, $\nu = 0.01$, $c = (2, 2, 1, 1, 1)$.

The scatter-plots of Figure 4.7(b) show many interesting trends. The diamond shape in the $\dot{\phi}$ vs ϕ plot in particular stands out, and captures the expected dependence of maximum safe lean angle on lean angle velocity. Whereas the scatter-plots show various projections of the full sample set, Figure 4.7(d) shows a cross-section of the learned model for $(\sigma_L,\ \sigma_F,\ \sigma_R) = (5\,\mathrm{m}, 3\,\mathrm{m}, 1/2\,\mathrm{m})$. It correctly indicates that the maximum safe lean angle can be achieved only when lean velocity is zero (i.e., the right-most tip of the region), and how increasing lean velocity limits the allowable lean angle. The viable space lies mostly in the right hand side of the diagram because a right lean (represented by negative values of ϕ), and consequently a right turn, is dangerous since there are obstacles to the right, as indicated by the sensor values. The viable space also lies mostly in the upper half of the diagram because negative $\dot{\phi}$ values (i.e., rightward lean velocity) are likely to lead to the dangerous right lean and turn. The model is nearly symmetrical (theoretically it should be exactly so) and hence an analogous situation exists for when obstacles appear on the left.

4.7.5 Performance evaluation

Tables 4.1–4.3 summarize the numerical findings. The columns specify: number of runs attempted, number of solutions found, average query time, median query time, median number of iterations taken, median number of nodes created, and median number of failure/collision checks performed. A discrepancy between number of runs attempted and solutions found indicates that some runs did not find a solution within the allotted time (usually 30 minutes for easier problems, and 60 minutes for more challenging ones). The average query time value was provided in addition to the median value as a convenience; the median arguably provides a better performance index since it is less susceptible to large-valued outliers and timeouts, and hence better represents a "typical" run.

Figures 4.8–4.10 give box-and-whisker[5] plots of key quantities; these have the advantage over

[5]The dot indicates the median, while the box extends from the first to the third quartile. The "whiskers" extend to the furthest sample at most 1.5 times the inter-quartile range from the box. Samples beyond this are marked with empty circles and commonly considered "outliers".

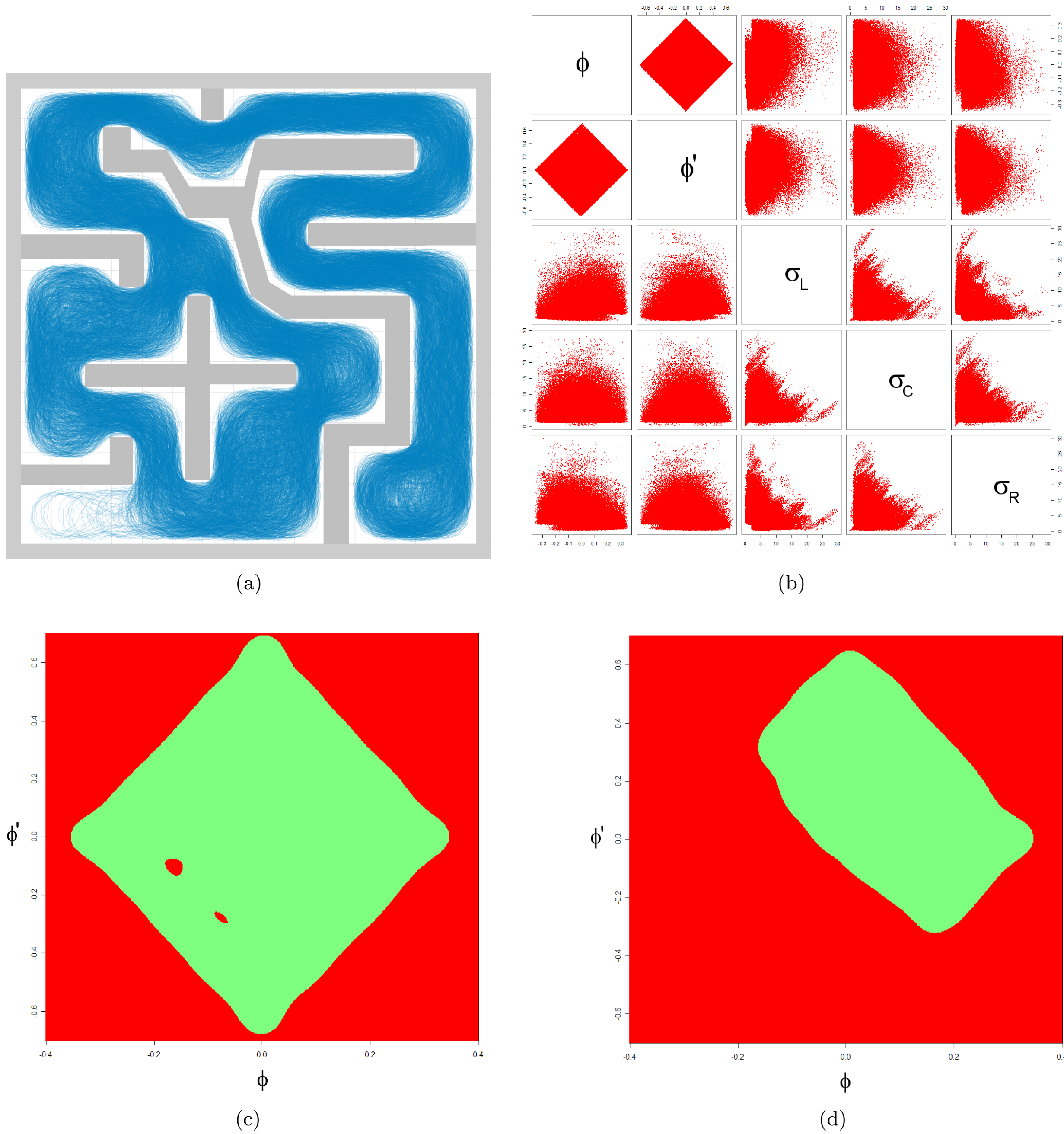


Figure 4.7: Viability model for bike agent; **(a)**: random walk trajectories used for training; **(b)**: distribution of resultant locally situated states λ; **(c)**: cross-sections of learned viability model for $(\sigma_L,\ \sigma_F,\ \sigma_R) = (5, 5, 5)$; **(d)**: cross-sections of learned viability model for $(\sigma_L,\ \sigma_F,\ \sigma_R) = (5, 3, 0.5)$

the tables in that they illustrate the variance of the variables.

Overall the planner with viability filtering ("RRTBlossomVF") performs very well, often improving runtimes by an order of magnitude. It is worth noting that, for each agent, the same "complex"-trained viability models were used in *all three* environments. This explains why the speed gains are sometimes not as dramatic in the other environments, and especially in "rooms-IKEA". The diminished gains are in general due to the sensors' perceptual limits which introduce environment-specific artifacts into the learned models, and such artifacts are not generally transferable (see §4.8.3). At the same time, it is encouraging to see how often the models are nonetheless surprisingly effective in such cases.

Interestingly, the plain RRT-Blossom algorithm found some scenarios troublesome, such as driving the car through "rooms-IKEA", and especially the "complex" environment, to the point of taking significantly longer than RRTCT. This is mostly due to the limited manoeuvrability of the car, the resulting degeneration of RRT-Blossom into a depth-first search (see §4.8), and the severely limiting nature of the environment, as apparent in the random-walk plot in Figure 4.6(b). It is thus surprising that viability filtering was able to reduce the planning time as it did.

It is also worth noting how the reduction in planning time is often surpassed by the reduction in the remaining columns, and the number of failure/collision checks in particular. The discrepancy is a side-effect of the additional computation time needed to compute sensor readings and to query the viability oracles. This suggests that optimized implementations of these two tasks could lead to additional runtime improvements.

Finally, Figures 4.11 and 4.12 illustrate the effect that viability filtering has on tree structure. The huge reduction in tree edges is clearly visible in the viability filtering variant. Also noteworthy is the complete lack of probing of corners under viability filtering, which is often pronounced with the other two planners. The other notable features are RRT-Blossom's localized huge concentrations of edges in various spots; this is a result of the lack of regression avoidance in nonviable regions of the state space.

4.8 Discussion

4.8.1 Nonviable goal states

Viability filtering presents a problem for single-tree planners since they will be unable to find a solution if x_{goal} is not generally viable (i.e., $x_{goal} \notin \mathcal{X}_{viab}$), where reaching x_{goal} places the agent in an unrecoverable position. The viability filtering mechanism will be actively deterring the planner from completing the last leg of any solution.

The dual-tree approach sidesteps this problem. Filtering ceases to be an issue with two trees since a solution now requires only a tree-tree connection, not a tree-node one. Let $\mathcal{T}_{init}$ and $\mathcal{T}_{goal}$

Table 4.1: Performance comparison for **inertial point**

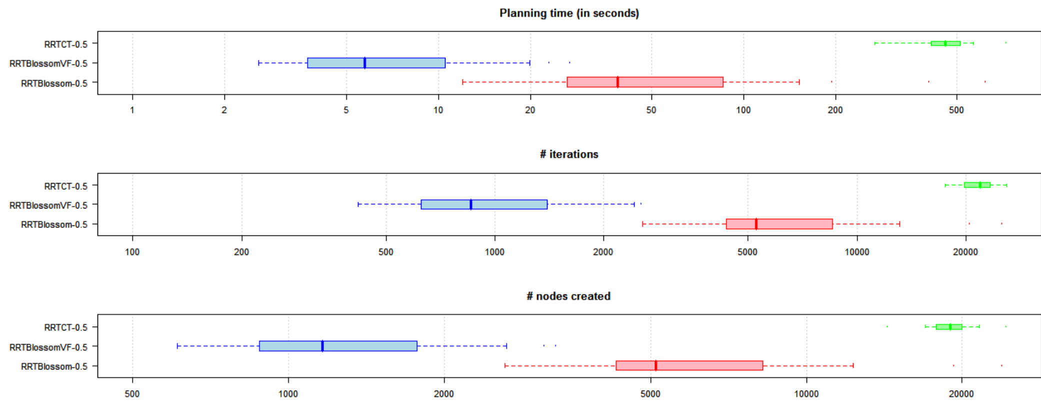
environ.	algorithm	runs	solns	time(s) median	time(s) average	iters	nodes	fail checks
complex (native)	RRTCT	10	10	473.8	469.7	22,306	19,711	71,969
	RRTBlossom	40	40	38.9	81.3	5,290	5,139	26,754
	RRTBlossomVF	40	40	6.3	7.9	868	1,188	5,287
Y	RRTCT	10	10	888.2	869.3	33,747	27,110	101,816
	RRTBlossom	40	40	230.3	234.6	12,421	12,356	62,939
	RRTBlossomVF	40	40	25.9	28.7	2,708	3,294	15,271
rooms-IKEA	RRTCT	10	10	706.0	745.7	34,026	23,760	96,806
	RRTBlossom	10	10	311.0	380.7	14,083	13,712	71,177
	RRTBlossomVF	40	40	57.2	83.8	3,727	4,304	20,192

Table 4.2: Performance comparison for **car**

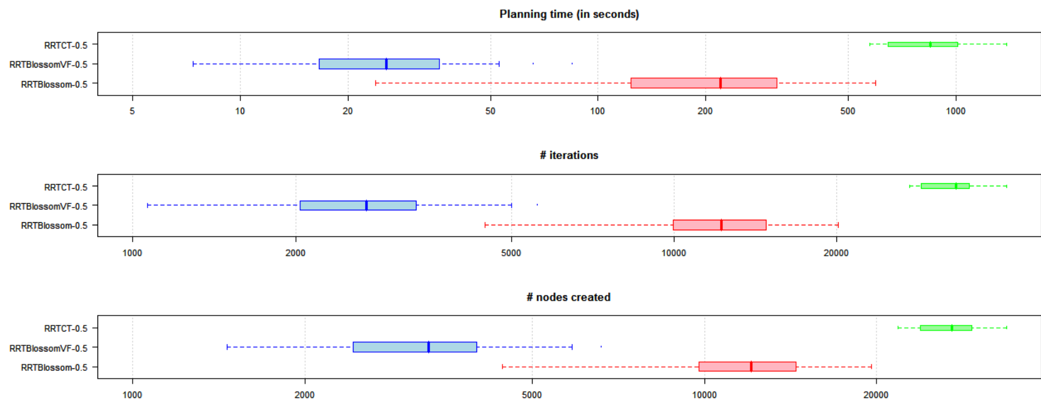
environ.	algorithm	runs	solns	time(s) median	time(s) average	iters	nodes	fail checks
complex (native)	RRTCT	40	40	74.5	75.2	8,414	8,696	25,862
	RRTBlossom	10	9	281.7	574.8	18,583	19,074	73,347
	RRTBlossomVF	40	40	5.3	6.1	669	967	3,223
Y	RRTCT	40	40	108.8	111.7	9,587	10,335	29,050
	RRTBlossom	10	10	64.5	188.7	8,155	8,669	32,946
	RRTBlossomVF	40	40	2.5	3.0	312	514	1,644
rooms-IKEA	RRTCT	10	10	350.5	315.4	21,025	19,418	60,130
	RRTBlossom	10	10	365.2	384.3	19,942	20,355	78,442
	RRTBlossomVF	40	37	121.3	269.3	5,567	7,756	25,803

Table 4.3: Performance comparison for **bike**

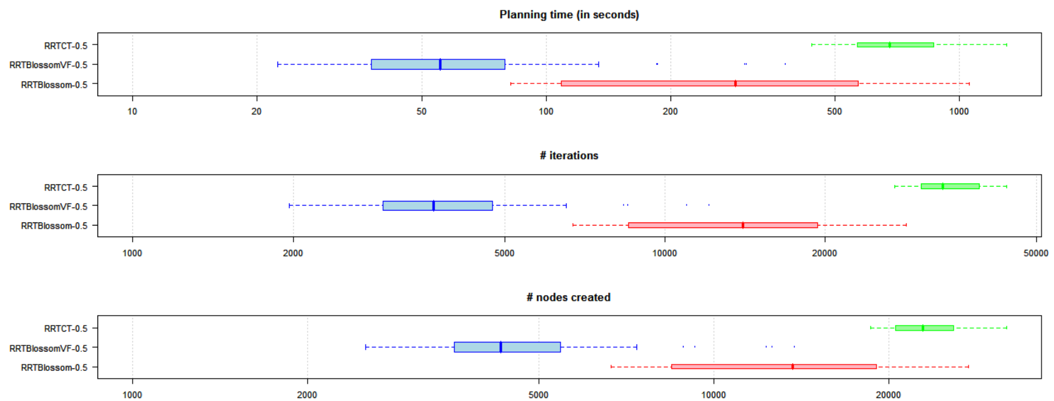
environ.	algorithm	runs	solns	time(s) median	time(s) average	iters	nodes	fail checks
complex (native)	RRTCT	10	10	318.5	371.5	28,362	16,123	87,380
	RRTBlossom	40	40	16.7	21.0	3,828	3,757	21,914
	RRTBlossomVF	40	40	5.1	5.6	626	770	4,184
Y	RRTCT	10	10	202.8	209.9	23,782	12,838	70,601
	RRTBlossom	40	40	10.5	13.5	2,945	2,850	16,624
	RRTBlossomVF	40	40	3.5	3.6	440	554	3,008
rooms-IKEA	RRTCT	10	9	2866.1	2148.6	78,108	46,474	247,012
	RRTBlossom	10	10	328.1	305.7	17,065	16,141	96,227
	RRTBlossomVF	40	40	29.5	34.3	1,956	2,175	12,202



(a)

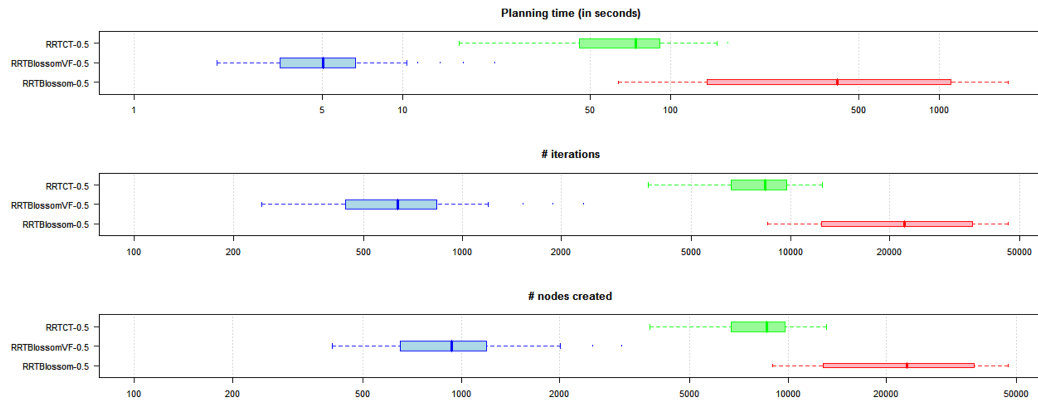


(b)

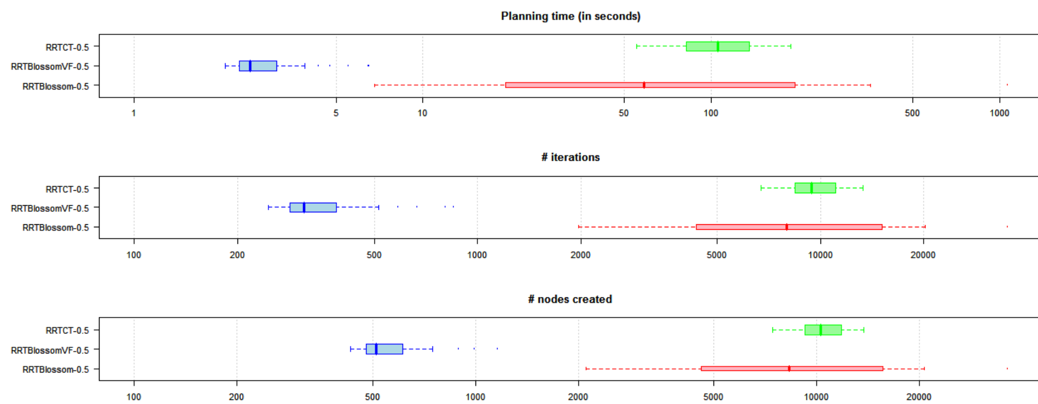


(c)

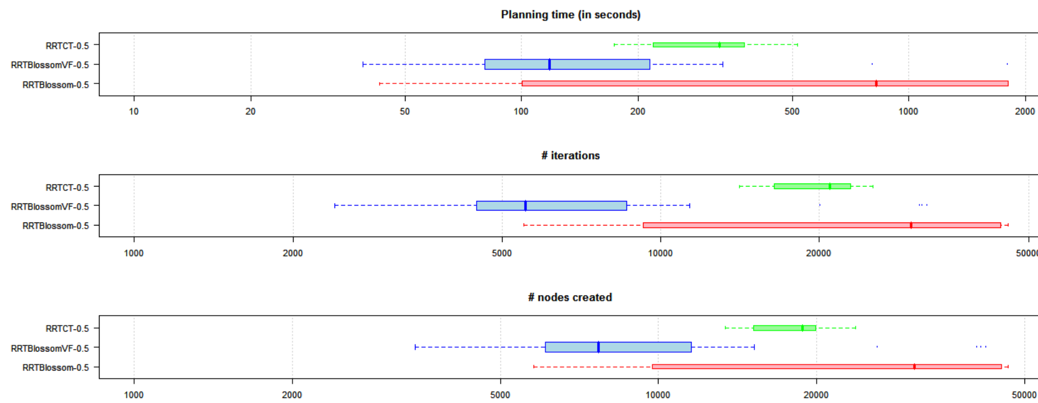
Figure 4.8: Box-and-whisker plots of trial results for **inertial point** agent in various environments: **(a)** complex; **(b)** Y; **(c)** rooms-IKEA. The x-axes are logarithmic to enlarge regions of detail; otherwise egregious outliers tend to marginalize the area of interest.



(a)

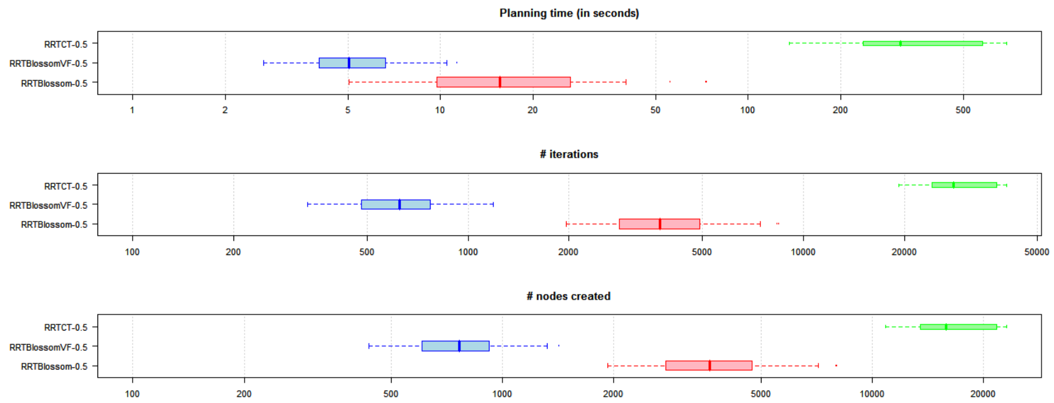


(b)

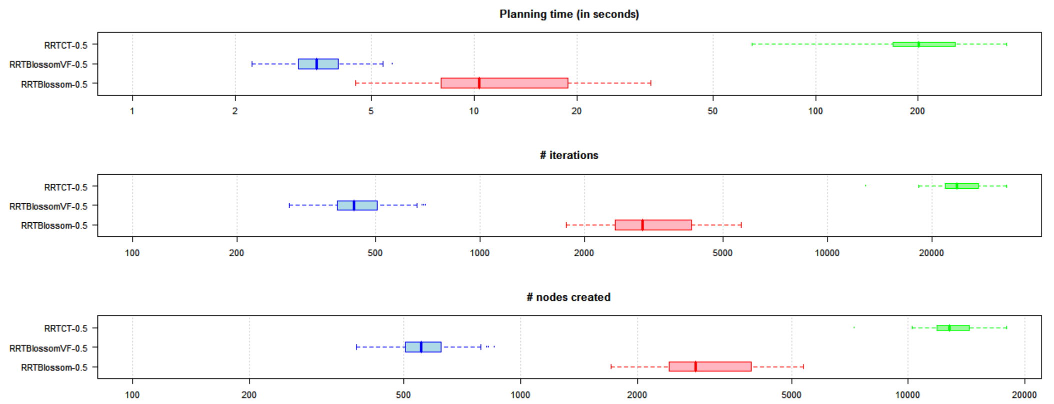


(c)

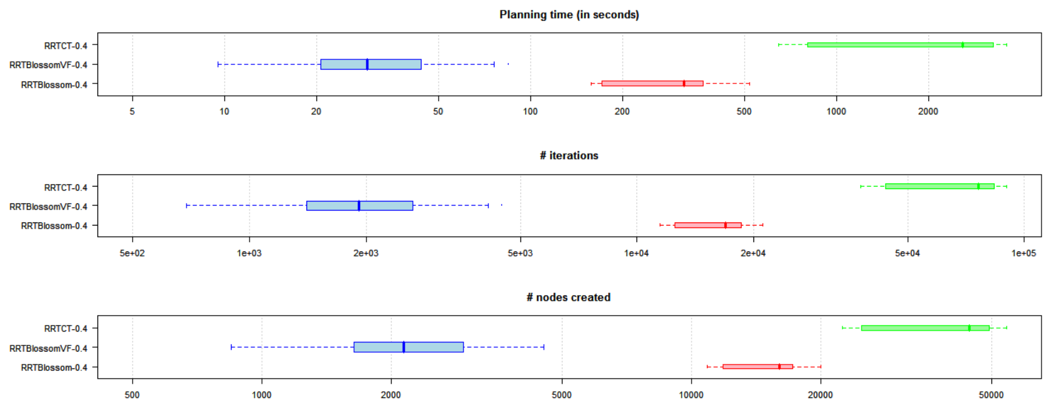
Figure 4.9: Box-and-whisker plots of trial results for **car** agent in various environments: **(a)** complex; **(b)** Y; **(c)** rooms-IKEA. The x-axes are logarithmic to enlarge regions of detail; otherwise egregious outliers tend to marginalize the area of interest.



(a)



(b)



(c)

Figure 4.10: Box-and-whisker plots of trial results for **bike** agent in various environments: **(a)** complex; **(b)** Y; **(c)** rooms-IKEA. The x-axes are logarithmic to enlarge regions of detail; otherwise egregious outliers tend to marginalize the area of interest.

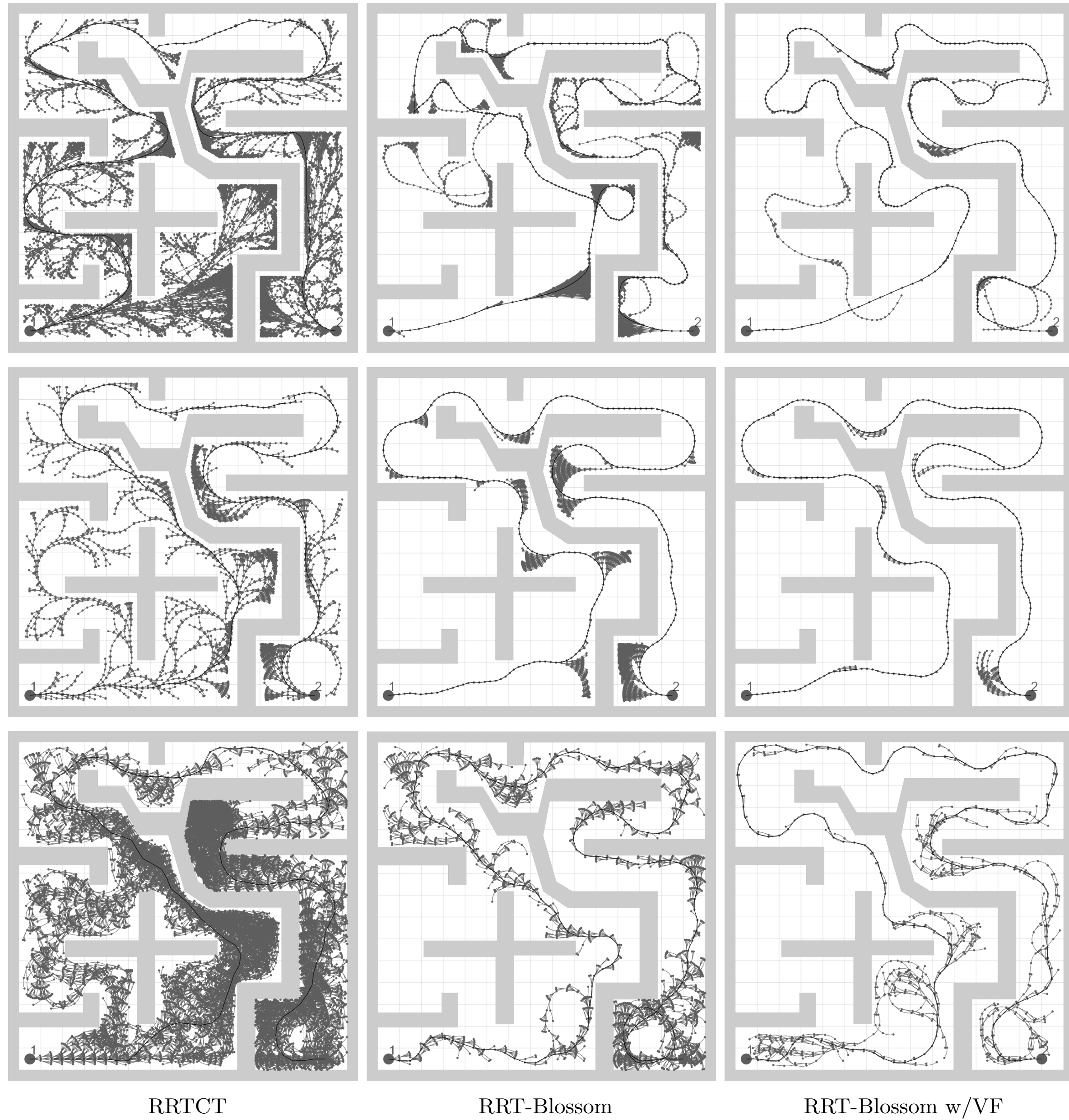


Figure 4.11: Visual comparison of tree density and structure; **top row**: inertial point agent; **middle row**: car agent; **bottom row**: bike agent.

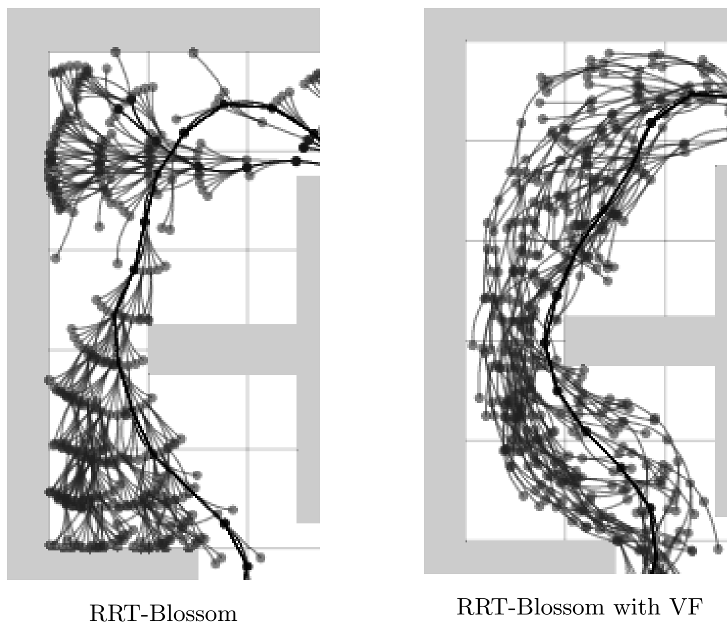


Figure 4.12: Magnified view of tree structures for the bike in a more difficult environment. The filtering planner (on right) avoids probing the corners and consists almost exclusively of viable trajectories.

denote, respectively, the x_{init} and x_{goal} trees. In a single-tree approach $\mathcal{T}_{init}$ must reach x_{goal}, which could be nonviable. In contrast, in a dual-tree approach $\mathcal{T}_{init}$ merely needs to reach one of the nodes in $\mathcal{T}_{goal}$, some of which are going to be viable, and hence reachable under viability filtering.[6] An analogous situation exists for $\mathcal{T}_{goal}$, in that $\mathcal{T}_{goal}$ need only reach any of the nodes in $\mathcal{T}_{init}$, rather than x_{init} specifically, which too could be (backward) nonviable.

The only foreseeable consequence of nonviable x_{init} or x_{goal} is that the two trees may only meet in space which is both, forward and backward viable. This usually does not have any noteworthy impact on the resulting solutions, but it may delay solutions in particularly constrained environments.

4.8.2 Choice of sensors

The speedup due to viability filtering is directly tied to the net balance of work saved (skipping exploration of futile branches) minus extra work incurred (computing sensor values and consulting the oracle). It is thus important to find sensors that are relatively cheap to compute, yet at the same time particularly adept at capturing viability-relevant attributes of the situation. Excessive concern for the computational cost can be misleading, however. For more difficult and failure-prone dynamical systems the potential gains of filtering can be so high as to justify the use of moderately expensive sensors. For example, it would be far cheaper to compute the sensory state of a car using three linear rangefinders in lieu of the whiskers, but this produces significantly poorer results,

[6] If no forward-viable states are (backward) reachable by $\mathcal{T}_{goal}$, then the motion planning problem has no solution. Likewise if no backward-viable states are (forward) reachable by $\mathcal{T}_{init}$.

despite the lower overhead.

A related but orthogonal issue to the above is the *number* of sensors used, or more specifically the number of dimensions in the sensory states, and ultimately the locally situated states. Clearly lengthier sensory states require more effort to compute, but more importantly they have a large negative impact on the accuracy of viability models. One may recall that the domain of the models is the locally situated state space Λ. The more dimensions there are to a locally situated state λ (which subsumes the sensory state), the larger the number of dimensions which the model must span. This is an issue because the number of training samples required to achieve a comparable level of model accuracy grows exponentially as the model dimensionality grows. Thus it is, again, preferable to use more expensive sensors if this results in a more compact sensory state vector.

4.8.3 Oracle error

The oracle has two sources of prediction error: modeling error and limitations in the sensors' ability to disambiguate states. The first is a result of either insufficient training samples or a poor choice in training procedure parameters, and is easily corrected. The second type of error is inherent in the data collected itself, and is much harder to fix. The problem stems from the agent's set of sensors being unable to disambiguate between some pairs of system states, x_1^+ and x_2^+, where one is viable and the other nonviable, and as a result mapping them both to the same locally situated state λ. Since the oracle will always return the same label for λ, whether it is computed from x_1^+ or x_2^+, then one of these system states will always be misclassified.

It is also worth looking at how model error affects filtering efficacy. Modeling error in the viability oracle can be either underinclusive (false negatives; viable states labeled as nonviable) or overinclusive (false positives; nonviable states labeled as viable). An underinclusive model restricts planner exploration more than it should. Although this results in a smaller search space, and consequently shorter planning times, it is detrimental in highly constrained environments since essential bottlenecks are easily made impassable when classified as nonviable. Overinclusive models, on the other hand, diminish the amount of filtering applied to the search space, causing the planner to gradually regress into the host planning algorithm (i.e., the algorithm to which viability filtering is being applied) as this error increases. In general, the amount of viability filtering is thus a function of model error, and spans a continuous spectrum as illustrated in Figure 4.13.

4.8.4 Oracle transferability

As the results show, a viability model can be effective in environments other than the one that was used in training. How well a model transfers is dependent on the degree to which the environments are similar in character and structure. For example, a model trained in wide open spaces will do poorly in highly constrained environments (and vice versa) since the model is being asked to predict

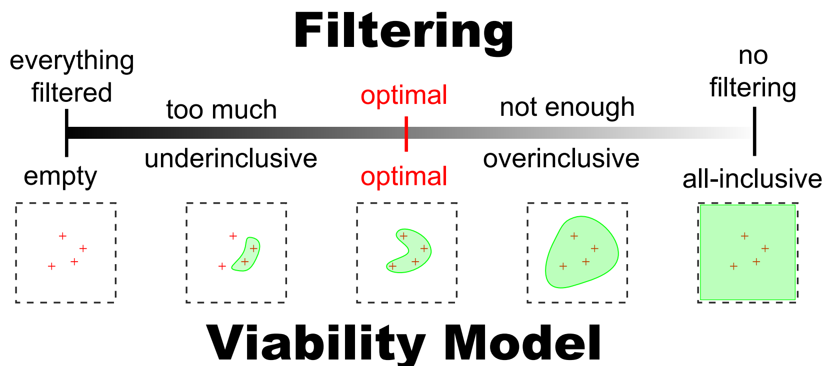


Figure 4.13: Amount of viability filtering as a function of model error

viability in an area it has little knowledge of; that is, the training samples used to derive the model will mostly span regions of Λ with large sensor values, whereas the model is being applied to an environment which generally constrains agent operation to the vicinity of Λ's origin (i.e., all sensors will tend to have low values). One obvious remedy to counter such oracle over-specialization is to train them on samples obtained from a variety of dissimilar environments. Initial experiments in such compositing of training data for the car have yielded good results.

Figure 4.14 shows when a model is applied in a excessively dissimilar environment. The left diagram shows a motion planning attempt using a model trained on the "complex" environment. Since that environment lacked large open areas (i.e., the model lacks knowledge of such open scenarios), the planner mistakenly considers the centre of the large chamber nonviable, and hence the agent prominently clings to the right wall, as this is at least partially within the realm of the model's learned experience. The right diagram, on the other hand, shows a motion planning attempt using a model trained in a 30 m by 30 m environment devoid of obstacles, aside from the bounding walls. Here the agent has no problem traveling through wide open spaces, but has a visible problem entering the upper corridor, despite the fact that a number of edges were already on the right track. This is a side effect of the model being unfamiliar with such tight passageways, which thus get labeled as nonviable.

4.8.5 Scarcity of samples and bootstrapping

There are two key issues with "bootstrapping" of the viability model, that is, the learning from scratch through self-observation by the planner. The primary issue is that viability filtering directly prevents the discovery of novel viable samples. Any such sample would be necessarily misclassified by the oracle, and thus barred from exploration. In short, a planner with viability filtering fully applied cannot extend its viability model using self-observation. Secondly, even if this were not so, the initial dearth of samples during bootstrapping would naturally result in heavily underinclusive models, and thus excessive filtering; this would in turn lead to frequent inability to find solutions to queries, thus jeopardizing the primary source of further training data.

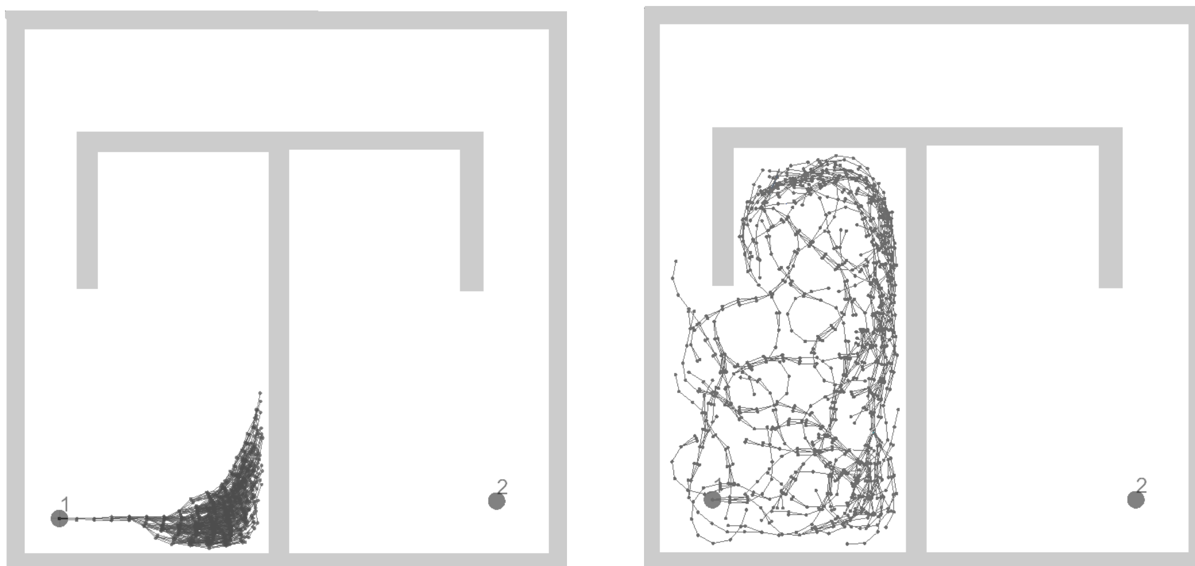


Figure 4.14: Effect of excessive mismatch between training and test environments; **left**: using a viability model trained on "complex" environment prevents traversing wide-open areas; **right**: using a viability model trained inside an equivalent square environment without any obstacles other than the walls, bars entry into narrow corridors.

Both problems can be likely overcome by stochastically phasing-in the viability filtering. That is, filtering could be applied only in iterations for which $r > \Phi$, where $r \in [0, 1]$ is a random variable, and $\Phi \in [0, 1]$ is a "phase-in" parameter that increases as the viability model fills out. The model's "maturity" could perhaps be gauged by the inverse of the rate at which novel samples are encountered. The general problem is similar to that of exploration-exploitation tradeoffs encountered in reinforcement learning.

It is hard to satisfactorily characterize the number of samples required to adequately capture the viable region. In general, the model grows in proportion to the rate at which novel samples are encountered, ones that lie outside current model bounds, and in proportion to the degree of their novelty. A practical characterization might be to claim that an adequate model has been reached when this rate of novel samples fall below some predetermined, sensible threshold.[7]

4.8.6 Expected reduction in planning effort

The viability filtering approach hinges on preventing the exploration of branches which are very unlikely to lead to a solution, hence the reduction in planning effort is proportional to the prevalence of such branches, which are usually the result of the interaction of system dynamics with imposed constraints (e.g., environment geometry). Such branches will typically be more numerous when the agent is generally unstable, has a very limited range of motion, or the environment is relatively constraining. The speedup will further depend on how conservative a viability model the oracle

[7] This is still not very general since it assumes that the training samples are drawn uniformly from Λ; this is usually not the case.

derives from training data (as discussed in §4.8.3). Finally, results will deteriorate if the target environment significantly differs in structure from those used in training, or when the sensors used do not sufficiently capture the local context of the agent.

4.8.7 Completeness

In general, the use of viability filtering precludes any completeness guarantees. This is because there is always the possibility that the planner, due to the imperfections of the viability models, will be barred from exploring bottlenecks in the environment that are critical to any solution. That is, error in the models, or the limitations of the sensor set, can always misclassify important states as nonviable, and thus render them impassable.

This could be easily "fixed" by foregoing the viability filtering mechanism every so many iterations, or in other probabilistic ways (e.g., see §4.8.5), thereby allowing the resultant search tree to match, in the limit, that of the underlying unfiltered algorithm. For example, in our implementation this would render the planner probabilistically complete, like the underlying RRT algorithm. Nonetheless, this is nothing more than a band-aid, petty and superficial, achieving little beyond attaining the apposite completeness label in a very inefficient manner. A far more preferable alternative is to run viability filtered planners in parallel with those that provide the desired completeness guarantee, as discussed in 2.1.6.

4.9 Conclusion and future work

This chapter proposes the use of locally situated state information as a means to give motion planners "sight", which then aids in the detecting and learning nonviable scenarios. This learned viability data can then be exploited to expedite planning by barring the planner from wasting its effort on exploring nonviable regions of the search space. Results are shown for three types of agents and demonstrate significant speedups, as well as generalization across environments.

4.9.1 Automation

The viability filtering approach described is nearly fully automatic. There are two aspects of it which still involve manual intervention, and it would be useful if these too could be automated. The first instance of this is the manual tweaking of the SVM learning parameters (i.e., γ, ν, and c) to ensure that a faithful model is derived. This is mostly a result of a simple implementation though; there are a number of cross-validation methods, such as "k-fold" and "leave-one-out", that should automatically achieve the desired goal. The other manually-designed aspect of the approach is the selection and design of virtual sensors. This is a more difficult problem with no obvious solution. The approach that currently appears to have the greatest chance of succeeding

is one where sensors are auto-generated such that they measure the distance to obstruction along agent trajectories corresponding to particular fixed control actions, similar to what was done with the car. This leaves the question of which control actions should be chosen for these sensors; it is reasonable to assume though that simple heuristics could provide adequate choices (e.g., controls with extremal and median values). This would likely produce excessive number of sensors, hence some form of automatic sensor evaluation and selection could be performed, whereby the utility in capturing local context is measured for each sensor, and a handful of the best are picked out to be used in the subsequent viability filtering.

4.9.2 Reinforcement Learning

An interesting and useful affinity exists between viability and Reinforcement Learning (RL). For example, a popular toy problem in RL literature is the learning to avoid falling over when riding a bike, which effectively amounts to finding the viable set of states. The common denominator between the two topics is dynamic programming: the most direct way to compute viability kernel in a discrete space is to use dynamic programming (e.g., [SP94]), while RL essentially consists of using dynamic programming to propagate reward information throughout the whole domain.

Computing the viable space of a system can be easily reformulated as an RL problem. One simply sets the reward (or, in this case, penalty) function to 1 for any state-action pair which immediately causes a collision, and 0 otherwise. The resulting value function then encodes the viability of the states: 0 indicates a viable state, while non-zero values indicate nonviability. In addition, the magnitude of the value indicates the minimal period of penetration, or more simply, how quickly the agent can recover from the collision. This latter value can be useful in many contexts, such as recovery from nonviable states, as discussed in the next chapter.

An intriguing possibility would be to learn the requisite local viability models by applying RL methods directly to the locally situated state space Λ, as opposed to doing so over $\mathcal{X}$, like it was done above. This would yield probabilistic models that, rather than classifying states as exclusively "viable" or "nonviable", would instead indicate the likelihood of viability. For example, using a slightly modified penalty function, where we assign 1 to any state-action pair (s_t, a) if s_t is collision-free and s_{t+1} is not[8], one would end up with a value function which reflects the proportion between viable and nonviable agent states x which map to a particular locally situated state λ. Clearly this greatly aid in compensating for suboptimal choice of sensors. How exactly to exploit such probabilistic viability information, or for that matter, how best to implement RL within a motion planner's framework, remain open problems.

[8] That is, the penalty function is nonzero only for transitions which represent the initial instant of collision.

4.9.3 λ histories

There are also some interesting extensions which would make viability filtering even more robust. For one, the learned models could predict viability based on $(\lambda_{k-j}, \ldots, \lambda_{k-2}, \lambda_{k-1}, \lambda_k)$, the *history* of locally situated states observed, rather than just on the most recent λ_k. This would likely yield more accurate estimates of viability, especially when suboptimal virtual sensors are used (e.g., automatically generate ones). In many ways this resembles work in information spaces, where histories also play an important role in planning.

4.9.4 Exploiting control action data

Another promising extension would be to use the training data to harvest not just the state trajectories, but also the corresponding control action sequences that have been applied to generate the former. This largely untapped source of information, especially when derived from observing the control of the agent by human subjects, could be combined with the use of locally situated state histories to provide the essential means to derive motion macro-primitives.

4.9.5 Extending applicability

Viability filtering could also be modified to extend its applicability. In particular, it would be interesting to enable it to handle dynamic environments. Even if the motion of the obstacles is fully known ahead of time, this presents a non-trivial problem. Partially observable environments could also be explored. The current formulation of viability filtering requires very little to adapt it to such problems, due to its highly local nature. In fact, the method should work without any changes: by placing an upper bound on the possible values returned by the virtual sensors, the planner is simply limited to operate within a more confined region of the viability model.

4.9.6 Other

Some loose ends also remain. It would be interesting to see whether viability filtering would produce greater or lesser improvements with other planners, such as RRT-CT. Bootstrapping is currently not possible since the method directly prevents the exploration of novel situations, but would be a worthwhile goal, yielding a planner that gains experience through its own operation. Solutions to this problem will likely involve some form of probabilistic relaxation of the regression constraint. The building and use of models based on composite training data from a number of dissimilar environments needs to be further tested. Although preliminary tests were very positive, it remains to be seen whether this is always the case. This would also require working out some guidelines, and related metrics, on how many different environments must be used, and how much do they need to differ, in order to yield a sufficiently robust model. Finally, it might be worthwhile to resolve the issue of a single-tree planner being unable to complete the last leg of a solution if x_{goal}

is nonviable. Although dual-tree planners are almost always used, there may be a few applications where this is not possible (e.g., agents for which reverse-time simulation is not possible).

Chapter 5

Safety enforcement with viability models

5.1 Introduction

For many user-controlled agents, such as cars, helicopters, and airplanes, collisions and system failures can have particularly dire consequences, such as the loss of human life or large monetary damages. It is thus often desirable to provide some form of automated protection against human error in such systems. Even when consequences are not so dire, or the agent is purely virtual as in computer animation, games, and prototyping, such automatic input correction can facilitate user-control of more difficult subjects. This chapter presents work which aims to address this goal by providing a simple general framework for arbitrary agents.

Automatic corrections on their own can be disorienting and confusing to the user when they come, hence an additional desirable property of such systems is some form of feedback that would allow the user to anticipate the adjustments. Haptic feedback (as opposed to visual, for example), applied to the mechanism through which the user controls the subject, is particularly well suited to this task because the haptic force can simultaneously fulfill both functions, hinting and correcting. A simple haptic implementation is briefly presented and discussed in this chapter.

Figures 5.1 and 5.2 show an example of such a safety system for a car. In the depicted scenario a "suicidal user" repeatedly attempts to steer the car off the track (see v_k in Figure 5.2); their actions are overridden—u_k is the control action actually applied—when this would lead to unrecoverable situations (i.e., nonviable states).

5.1.1 Collision checking vs. viability checking

Clearly any such safety enforcement system needs to rely on some form of look-ahead to determine the threat presented to the agent by the user's current control actions. A key idea of the proposed

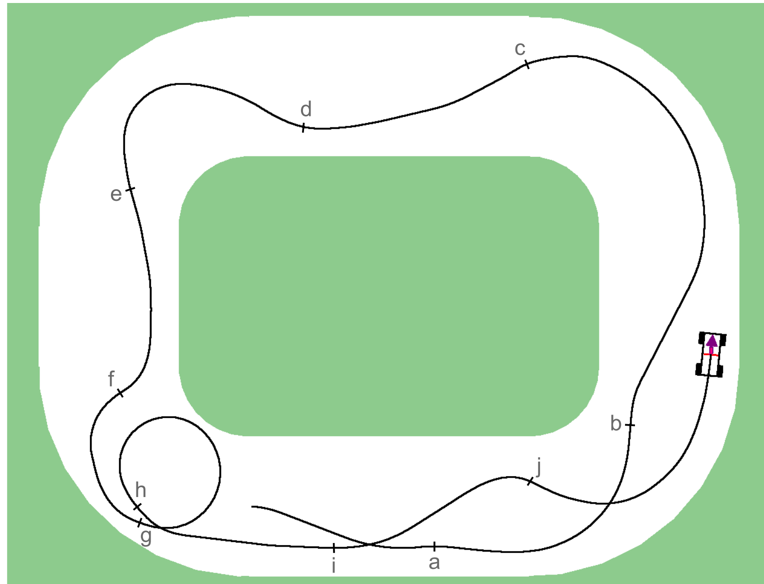


Figure 5.1: A car constrained to stay on the track; see Figure 5.2 for plot of corresponding control inputs.

system is that safety can be enforced more reliably and cheaper by keeping the agent from nonviable states, rather than through directly attempting to avoid collision. This is because in any physical implementation the look-ahead must be finite, yet most agents can encounter situations where the "time to unavoidable collision" exceeds the chosen look-ahead duration, and this could lead to system failure. For example, in the case of the lunar lander, however large we choose the look-ahead T_h, it is always possible to find a state for the agent where its downward velocity and insufficient altitude exceed the braking capacity of the bounded thruster, leading to a crash at time $t > T_h$. The danger of these states is thus undetectable by the chosen look-ahead. In contrast, a finite time horizon does not pose a problem for viability-based safety enforcement. In fact, theoretically even a single time-step look-ahead should be sufficient to ensure total system safety. If within that single time-step the user's control action causes the agent to become nonviable, the system merely applies a different control action, one that does not leave viable space. Such an action is guaranteed to exist by the definition of a viable state.

In safety enforcement the decision boundary between viable and nonviable space is of particular importance, while the remaining volume of the viability kernel $Viab(K)$ is only of secondary interest. We have thus found it convenient to coin and use the term *viability envelope* to denote this decision surface. The term carries the useful connotations of common phrases such as the "flight envelope" and "pushing the envelope", with the latter being particularly poignant and appropriate in the haptics context. It is sometimes also referred to as the "control envelope" since a "viable" state is also known as "controllable", in the terminology of the control field.

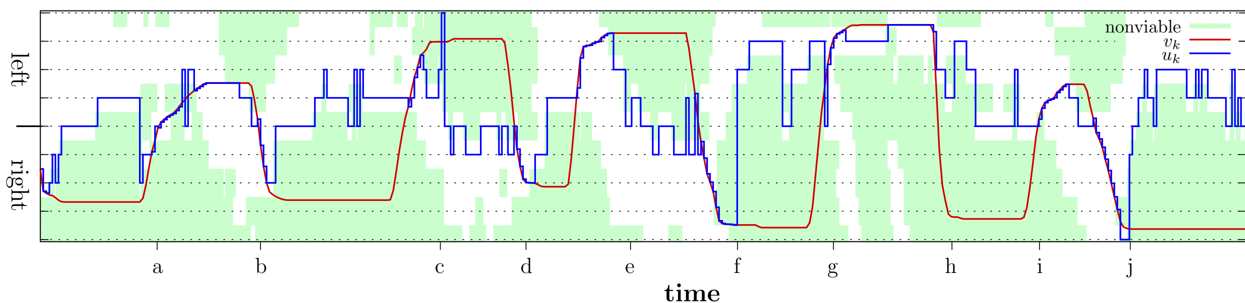


Figure 5.2: Plot of user's desired steering angle (v_k; red, smooth), actual steering applied (u_k; blue, discrete), and the viability of steering angles through time (control actions which approach the envelope are shaded pale green) for the simulation run shown in Figure 5.1.

5.1.2 Theoretical framework vs. sample implementation

In this chapter we present both, a theoretical framework for safety enforcement using viability (§5.2), as well as a rudimentary sample implementation (§5.3). Although the theoretical framework guarantees safety, the simplistic implementation violates a number of assumptions which void this guarantee. Nonetheless, even with its flaws the implementation generally provides a level of safety that is higher than that of typical approaches based on collision tests. Also, the main aim of the sample implementation has been to demonstrate the feasibility of the general approach, as well as to identify and explore any associated difficulties; robustness and application-readiness were of secondary importance.

The framework itself is theoretically simple, but its implementation requires addressing a number of practical issues, to which the bulk of this chapter is devoted to. The core of the system naturally focuses on detecting transitions into nonviable space, as well as strategies for averting this. Since this requires frequent checking of agent's viability—which may be expensive to compute or even unknown ahead of time—the agent's viability is therefore modeled with a fast machine learning classifier. The use of an inexact model in turn requires additional adjustments for dealing with classification error.

The guiding principle throughout the design of the system has been that of "least surprise"; the system has a mandate to be minimally intrusive. Without this additional guideline the general problem of nonviability avoidance is under-constrained as there are often many ways to respond to and avert imminent transitions to nonviable agent states (e.g., a car driving toward a brick wall, on the brink of getting too close, still has the choice of turning left or right). Since ultimately the goal is to facilitate manual control of an agent, it makes sense then to optimize toward a more effective and agreeable human-computer interface. In practical terms, this principle leads to such design decisions as always choosing the mildest correction that diverges the least from the user's desired actions, or using haptic feedback to alert user of upcoming needed corrections.

5.2 Framework

We first examine single-step containment, which is conceptually all that is needed to ensure safety. We then motivate and examine a multi-step look-ahead approach, and its various issues.

5.2.1 Single-step containment

The simplest strategy for ensuring agent viability can be stated as follows: apply the user's desired control action for a single time-step unless this causes the agent to become nonviable, in which case choose an alternate, safe control action. In theory, such a breach-free control action (i.e., one that does not breach the viability envelope) must exist by definition, since otherwise the previous state would not be viable to begin with. In practice, although this is usually the case, it is not guaranteed. The viability of some states may rely on continuous, arbitrary variation of the control action applied, whereas most physical implementations involve some form of control action discretization, whether of time or the control action itself. For example, in our implementation control actions assume a constant value over each time-step. Ultimately though, the effect of this issue is eclipsed by later trade-offs and approximations.

In accordance with the guiding principle of least surprise and intrusiveness, the single-step containment strategy consists of overriding the user's control action only if this presents a danger, and the correction applied should be the mildest possible. Formally this yields the following strategy:

$$u_k = \begin{cases} v_k & \text{if } F(x_k, v_k) \in \mathcal{X}_{viab}, \\ \underset{u \in \mathcal{U}_v}{\operatorname{argmin}} \|u - v_k\| & \text{otherwise.} \end{cases} \tag{5.1}$$

Here v_k is the control action requested by the user at time-step k, u_k is the control action actually applied, $x_k \in \mathcal{X}$ is the agent's state, $\mathcal{X}_{viab}$ is the viable region of state-space, F is a function that embodies the system dynamics in a discrete time setting (i.e., $x_{k+1} = F(x_k, u_k)$), and

$$\mathcal{U}_v = \{u \mid F(x_k, u) \in \mathcal{X}_{viab}\} \tag{5.2}$$

is the subset of control actions which maintain agent viability.

Discretization of U

Unfortunately it is often not feasible to construct the set $\mathcal{U}_v$, nor to inspect all its members. If the agent's control action space $\mathcal{U}$ is continuous, $\mathcal{U}_v$ will usually be infinite also. Clearly it is impossible to explicitly inspect all its elements, while analytical approaches often tend to be difficult and not general. Even with a finite $\mathcal{U}_v$ it may often be too expensive to perform explicit inspections. This presents a problem when attempting to find an alternate, safe control action.

We resolve this difficulty by (sparsely) discretizing $\mathcal{U}$, yielding the subset $\widehat{\mathcal{U}}$. That is, in case of an envelope breach by v_k, the search for an alternate safe control action is performed over the elements of $\widehat{\mathcal{U}}$, or rather its viable subset

$$\widehat{\mathcal{U}}_v = \{u \mid u \in \widehat{\mathcal{U}} \ \wedge \ F(x_k, u) \in \mathcal{X}_{viab}\}. \tag{5.3}$$

The size of $\widehat{\mathcal{U}}$ is chosen to be as small as possible to reduce computational load, but large enough so that it contains a viable control action for most situations. For simple systems (e.g., those amenable to bang-bang control) the discretization can be very sparse, since usually either the minimal or maximal input is viable. On the other hand, for complex systems even very dense discretizations can lack a viable control action in some situations, especially if this results in the viable subset of $\mathcal{U}$ being small and inconveniently distributed. This is a fragment of the larger issue touched on earlier, that viability in general assumes continuous time and control framework, whereas most physical implementations need to be discrete in at least one of these dimensions. We look at ways to mitigate the problem in the next section.

5.2.2 Multi-step containment

The single-step containment method, although very simple and easy to implement, has a number of undesirable properties. In particular, it often leads to severe control corrections, leaves little room for the errors which real-life implementations must always contend with, and does not allow much potential nor time to provide haptic feedback to the user. Thus in many ways the single-step approach fails to meet the earlier design principle of least surprise and intrusiveness.

Breach detection

A natural solution to the above short-comings is to extend the look-ahead period used in the single-step scheme, thereby enlarging the system's awareness of its surroundings. In particular, this allows for milder corrections, as suggested by Figures 5.3 and 5.4. The extension of the look-ahead applies to both, the checking of whether the user's current control action v_k is suitable, as well as to the searching for alternate safe control actions when it does not. The duration of the new look-ahead period, which we refer to as the *time horizon*, is denoted by T_h, and is measured as an integral number of time-steps[1] in the discrete time case.

When gauging the threat posed by a given course of action u, the system assumes that the user will maintain this control action fixed for the whole duration T_h. Exhaustively mapping out the full look-ahead tree of depth T_h, where each (non-leaf) tree node has all $|\widehat{\mathcal{U}}|$ children, is impractical in most cases. The above assumption thus constitutes a "best guess" at the user's future course

[1] This is more convenient in practice than measuring in seconds.

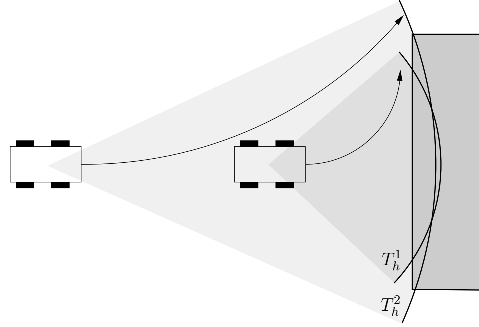


Figure 5.3: A larger time horizon usually allows milder corrections. Above, the left car has more time and space to manoeuvre and avoid the obstacle, and so is able to use a gentler turn. In general, using a larger time horizon does not preclude the use of the shorter control strategy, hence it is guaranteed, at the very least, to do no worse.

Figure 5.4: Trajectory comparison between single-step and multi-step containment. Multi-step containment tends to produce earlier and softer compliance with the viability envelope constraint.

of action, as discussed further in Section 5.5, and represents a compromise between predictive accuracy and computational load.

The look-ahead process may thus be envisioned as initially projecting a single trajectory in $\mathcal{X}$, starting at the agent's current state, and predicting its path throughout the next T_h time-steps. When this trajectory breaches the viability envelope, a look-ahead tree is then constructed, a set of $|\widehat{\mathcal{U}}|$ trajectories joined at their starting point (e.g., Figure 5.5), each predicting the agent's trajectory for a different control action $u \in \widehat{\mathcal{U}}$ through the next T_h time-steps. This approach degrades gracefully to the earlier single-step containment method when $T_h = 1$.

Time to envelope breach

How best to respond when an envelope breach is detected depends on the threat presented by the breach. A natural way to assess this threat is with the *time to envelope breach*,

$$T_{eb}(x_k, u_k) = \begin{cases} \min i \mid F^i(x_k, u_k) \notin \mathcal{X}_{viab} & \text{if } i \text{ exists and } i \leq T_h, \\ +\infty & \text{otherwise.} \end{cases} \tag{5.4}$$

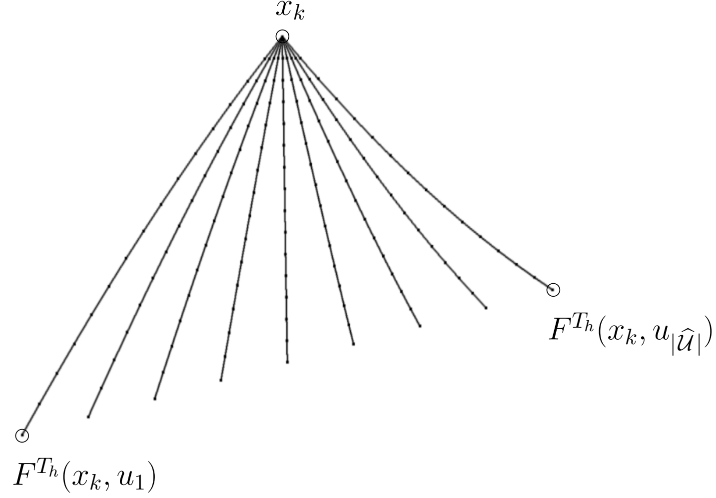


Figure 5.5: Structure of the look-ahead tree in the agent's state-space for the multi-step containment case. The look-ahead tree shown is that of the lunar lander agent. The tree has depth T_h.

$F^i(x, u)$ represents function iteration, the successive composition of F with itself i times:

$$\begin{aligned} F^i(x,u) &= F(F^{i-1}(x,u),u) \\ F^1(x,u) &= F(x,u) \\ F^0(x,u) &= x \end{aligned} \tag{5.5}$$

Simply put, T_{eb}, which is an integer value like T_h, is just the number of time-steps until a nonviable state is encountered, provided the breach occurs within the time horizon. That is, $0 \leq T_{eb} \leq T_h$. If no envelope breach is detected within the time horizon, it is convenient to let $T_{eb} = +\infty$; this is not strictly necessary, but it avoids leaving the value undefined in such cases, and it simplifies a number of formal descriptions later.

The array of T_{eb} values, one for each $u \in \widehat{\mathcal{U}}$, is how the agent perceives the environment. Figure 5.6 illustrates some examples and notes how they may be interpreted. This strongly parallels the use of virtual sensors in the previous chapter, but whereas earlier the sensor readings were used to predict the viability of the current state, here they merely serve to assess the most desirable course of action (based on projected departure from viable space).

Breach response

The time-to-envelope-breach metric is useful for assessing the severity of the threat to which the agent is subject to. There are four distinct threat levels that the agent may be in, each with its

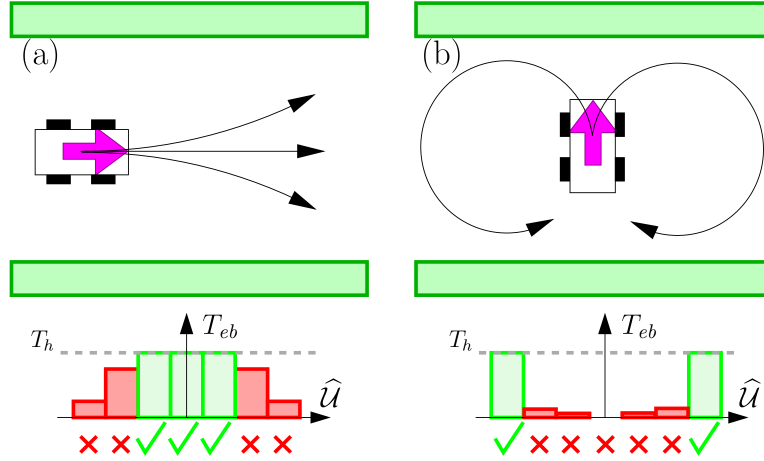


Figure 5.6: T_{eb} behaviour and viability of $\widehat{\mathcal{U}}$ for a fixed-velocity car; the trajectories drawn correspond to breach-free controls, which appear check-marked and green in the T_{eb} vs. $\widehat{\mathcal{U}}$ bar graphs underneath. Case (b) is particularly noteworthy: although the car's distance from the opposing roadside at first suggests some leeway, the car is in fact nearly upon the point of no return (i.e., envelope), as hinted by the many $T_{eb} \approx 0$, and must immediately choose one of the breach-free inputs.

own response strategy. The threat levels are:

$$\begin{aligned}
&\mathcal{L}_0: && T_{eb}(x_k, v_k) > T_h \\
&\mathcal{L}_1: && T_{eb}(x_k, v_k) \leq T_h && \wedge \quad \exists u \; T_{eb}(x_k, u) > T_h \\
&\mathcal{L}_2: && \forall \hat{u} \; T_{eb}(x_k, \hat{u}) \leq T_h && \wedge \quad x_k \in \mathcal{X}_{viab} \\
&\mathcal{L}_3: && x_k \notin \mathcal{X}_{viab}
\end{aligned} \tag{5.6}$$

while the corresponding response strategies adopted by the system are:

$$u_k = \begin{cases} v_k & \text{if } \mathcal{L}_0 \\ \operatorname*{argmin}_{u \in \mathcal{U}_v} \|u - v_k\| & \text{if } \mathcal{L}_1 \\ \operatorname*{argmax}_{\hat{u} \in \widehat{\mathcal{U}}} \; T_{eb}(x_k, \hat{u}) & \text{if } \mathcal{L}_2 \\ \text{—} & \text{if } \mathcal{L}_3 \end{cases} \tag{5.7}$$

where $\mathcal{U}_v$ is again the subset of $\mathcal{U}$ consisting of all *breach-free*[2] control actions at the given agent state; that is,

$$\mathcal{U}_v = \left\{ u \mid u \in \widehat{\mathcal{U}} \;\; \wedge \;\; \forall i \in \{0, 1, \ldots, T_h\} \;\; F^i(x_k, u) \in \mathcal{X}_{viab} \right\}. \tag{5.8}$$

In brief, the four modes represent progressively more hazardous threat levels. $\mathcal{L}_0$ and $\mathcal{L}_1$ constitute normal operation, and are analogous to the handling of the single-step containment case,

[2]Here, as in the rest of the chapter, the expression "breach-free" always implicitly assumes "...within the time horizon T_h."

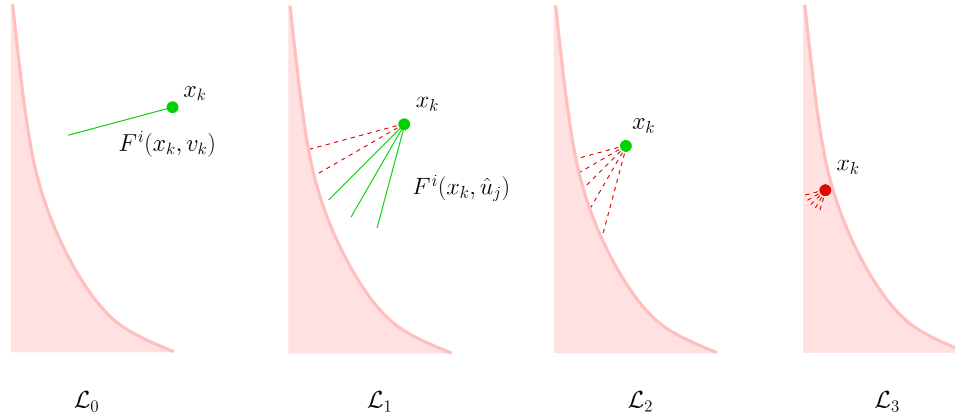


Figure 5.7: Example scenarios for the four threat levels, as seen in the agent's state-space $\mathcal{X}$. The shaded area under the curve indicates nonviable space. Dashed lines indicate envelope-breaching trajectories (i.e., where $T_{eb} < T_h$).

while $\mathcal{L}_2$ and $\mathcal{L}_3$ correspond to crisis handling modes. Figure 5.7 illustrates situations corresponding to the four threat levels.

Specifically, $\mathcal{L}_0$ represents "no threat" condition since the user's desired control action v_k does not incur an envelope breach within T_h; the user's control action is thus applied unaltered. In $\mathcal{L}_1$, v_k does lead to a breach, but other control actions in $\widehat{\mathcal{U}}$ do not; a suitable alternate action is thus chosen from among these.

In level $\mathcal{L}_2$ all control actions in $\widehat{\mathcal{U}}$ incur a breach within the time horizon (when held fixed for the duration T_h). Since $x_k \in \mathcal{X}_{viab}$, the definitions dictate that a viable control action exists, although it may lie outside $\widehat{\mathcal{U}}$, or require a finer discretization of time. Often though this condition simply indicates that the control action must be altered at least once within T_h. The system thus responds to this threat level by choosing the control action with the largest T_{eb}, with the expectation that doing so presents the best chance of leading to a successful control sequence, and if not, that it minimizes the egregiousness of any transgression.

Finally, in level $\mathcal{L}_3$, the agent has left viable space. Theoretically it should be impossible to reach this threat level, but due to the discretization of $\mathcal{U}$ and time, as well as some further approximations adopted later in this chapter, this not the case. It is not clear-cut or obvious way how best to resolve such situations; we discuss our implementation further on.

5.3 Sample implementation

This section presents a rudimentary sample implementation of the above framework. It details design decisions as well as implementation issues encountered.

5.3.1 Modeling viability

Analytic descriptions of viability are usually not available for more complex kinodynamic agents, and thus must be derived empirically or heuristically. Even if external viability models do exist, consulting them frequently may still be too expensive; the full look-ahead tree requires roughly $T_h \times |\widehat{\mathcal{U}}|$ checks per simulation time-step, and twice that during search for re-entry into viable space in $\mathcal{L}_3$, all to be performed at interactive rates (e.g., 30 Hz), and in addition to other computational costs of the system. Finally, even if a fast model does exist, it may be not amenable to some queries the system needs to perform, such as measuring the distance from an arbitrary state to the decision surface.

It is thus convenient to model the viability information using a fast classifier. In our implementation viability is assessed using empirical tests and heuristics (discussed in §5.4), and then captured using a Nearest Neighbour (NN) classifier

$$NN(x) = \begin{cases} \texttt{viable} & \text{if } \min_{z \in \widehat{\mathcal{X}}_{in}} \|x - z\| \leq \min_{z \in \widehat{\mathcal{X}}_{out}} \|x - z\|, \\ \texttt{nonviable} & \text{otherwise.} \end{cases} \tag{5.9}$$

$\widehat{\mathcal{X}}_{in}$ and $\widehat{\mathcal{X}}_{out}$ are sets of samples which have been classified by the oracle to be viable and nonviable, respectively. These sets are obtained in a trivial off-line pre-computation step, described in Algorithm 7. It is worth noting that a sampling approach such as this will produce viable state samples which are not necessarily reachable, thus giving a larger envelope than one might expect; this does not diminish the effectiveness of the resultant envelope though.

Algorithm 7 Computing $\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out}$ sample sets for *NN* classifier

$\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out} \leftarrow \{\varnothing\}$
for $i = 1$ to n **do**
  $\vec{x} \leftarrow$ `rand_uniform`$(\mathcal{X})$
  if `oracle`$(\vec{x}) =$ `viable` **then**
    $\widehat{\mathcal{X}}_{in} \leftarrow \widehat{\mathcal{X}}_{in} + \vec{x}$
  else
    $\widehat{\mathcal{X}}_{out} \leftarrow \widehat{\mathcal{X}}_{out} + \vec{x}$
$\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out} \leftarrow$ `scale_samples`$(\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out})$
$\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out} \leftarrow$ `dump_redundant`$(\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out})$

As with most learning methods, it is necessary to normalize or scale the training data prior to use, especially given that the NN classifier uses an L_2 norm distance metric in the agent's state-space. At present we select appropriate scaling factors manually, based on some understanding of the shape of the controllable region. The parameters are chosen so that the salient features of the envelope surface (i.e., important bumps, valleys, etc.) are not trivialized by classifier error and noise in other dimensions.

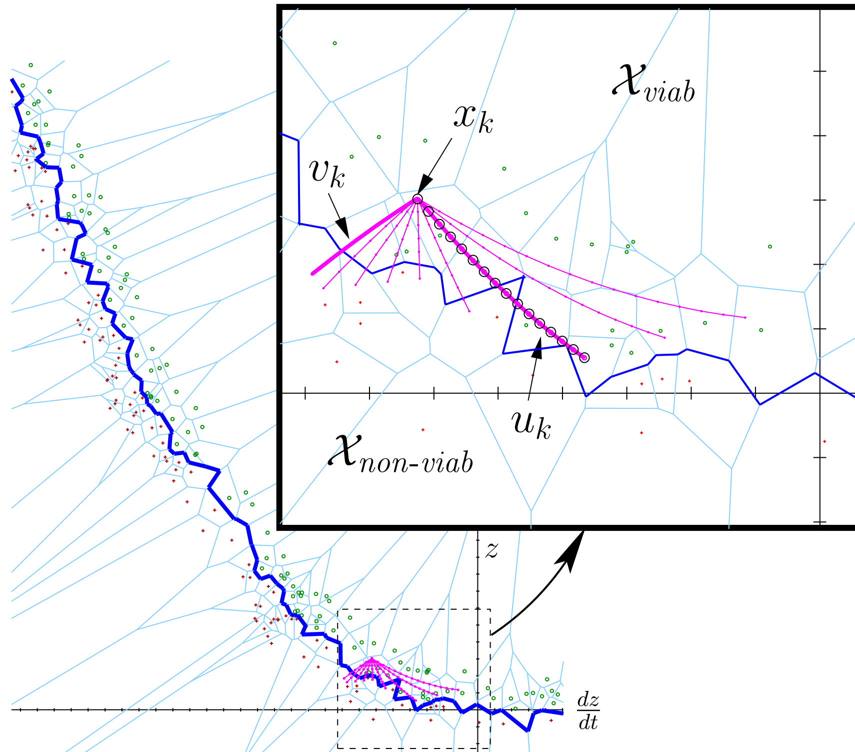


Figure 5.8: The lunar lander's 2D NN envelope; the set of trajectories emanating out of x_k shows the look-ahead tree, with $T_{gr} = 1$. The user's zero-thrust input (leftmost, labeled "v_k") is being overridden by the one labeled u_k. Also shown are the band of samples adjoining the envelope, and the resultant Voronoi tessellation.

A final measure taken to reduce unnecessary computational load is to discard redundant samples, ones which do not contribute to the NN decision surface, and thus whose removal leaves it unaltered. Although a number of methods exist to do this[P.E68, Cha74, Das91], we employ a simpler technique: since the samples are uniformly distributed, we compute the average inter-sample distance δ_s and then discard all samples which are further than $k\delta_s$ from the decision surface[3]. Progressively larger values for k are used (e.g.,$\{5, 10, 20\}$) until one is found that leads to a revised model that correctly classifies all the training samples. This yields a well-structured band of samples around the decision surface.

Figure 5.8 shows an example of a section of a viability envelope that was derived for the lunar lander agent.

5.3.2 Grace period

Finally, we employ a *grace period* when identifying envelope crossings, primarily to combat the noisy nature of NN envelopes. We define T_{gr}, the grace period, as the maximum number of time-steps that a trajectory may stray into the opposite side of the envelope without being officially labeled as a transition; conversely, a transition is only pronounced if the trajectory excursion into

[3]We approximate this by instead measuring the distance to the nearest NN sample of opposite class.

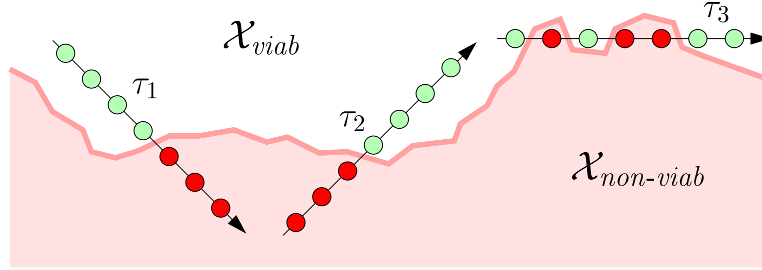


Figure 5.9: Using a grace period to combat envelope (approximation) noise; for $T_{gr} = 2$, τ_1 forms a definite breach, τ_2 a definite re-entry, and τ_3 merely a "brushing" of the envelope. That is, τ_1 and τ_2 are decreed as transitions, while τ_3 is not.

the opposing region lasts longer than T_{gr}. The rationale for this is that, as trajectories τ_1 and τ_2 of Figure 5.9 suggest, the longer a trajectory stays within the latter region of the model, the more certain one can be that the perceived transition reflects reality, and is not merely an artifact of error in the envelope model. A trajectory such as τ_3 in Figure 5.9 thus does not qualify as a transition according to this criterion.

5.3.3 Threat level response

$\mathcal{L}_3$ threat level

This most detrimental threat level occurs when the agent has left viable space, at least according to the viability model. Since avoidance is no longer an option, the next best strategy is to minimize impact of the transgression. We do this by choosing the "least detrimental" control action. The most direct method would be to either minimize depth of obstacle penetration or the duration of such a failure. In our implementation we adopt the second approach, by selecting trajectories which are the quickest in making the agent viable again. Alas, in many systems even a shallow transgression can result in arbitrarily long times prior to re-entry into viable space, whereas any physical implementation needs to have finite (and short) look-aheads. We thus increase the look-ahead period but limit it to up to twice the time horizon value T_h. If none of the look-ahead trajectories re-enters viable space within that period, we resort to approximation by selecting the control action that comes the closest.

The response to the $\mathcal{L}_3$ threat level can thus be formalized as

$$\mathcal{L}_3 : \quad u_k = \begin{cases} \underset{\hat{u}}{\operatorname{argmin}} \; \min i \mid F^i(x_k, \hat{u}) \in \mathcal{X}_{viab},\ i \leq T_{max} & \text{if it exists,} \\ \underset{\hat{u}}{\operatorname{argmin}} \; \rho_v\big(F^{T_{max}}(x_k, \hat{u})\big) & \text{otherwise.} \end{cases} \tag{5.10}$$

T_{max} is the upper bound on re-entry look-ahead ($T_{max} = 2\,T_h$ in our implementation), while $\rho_v(x)$ is a measure of how far the state x lies from viable space. We approximate this by averaging the distance from x to its k nearest neighbours in $\widehat{\mathcal{X}}_{in}$ ($k = 3$ in our implementation).

A complementary measure one may take is to use a conservative envelope, one which errs on the side of safety when placing the boundary. We have not yet explored any methods for doing this, but a straightforward one would be to shrink the original envelope by a small percentage. The benefit of this is that any shallow breach of this envelope, such as given by the least-detrimental criterion above, will likely not incur a breach of the true envelope, thereby maintaining system safety.

$\mathcal{L}_1$ threat level

The response strategy presented earlier for threat level $\mathcal{L}_1$ is the most conservative approach possible, but not necessarily the best. A more lenient approach would be to "ease-in" the corrections, based on immediacy of a breach. That is, the $\mathcal{L}_1$ response in Equation (5.7) could be instead set to

$$\alpha\, v_k \;+\; (1-\alpha) \operatorname*{argmin}_{u\,\in\,\mathcal{U}_v} \|u - v_k\|, \tag{5.11}$$

where α measures immediacy of a breach using

$$\alpha = \frac{T_{eb}(x_k, v_k) - 1}{T_h}. \tag{5.12}$$

This allows the user more freedom at longer lead times, but generally requires longer time horizons to be effective; this is not always feasible or desirable.

5.4 Experiments

Three agents were used in testing: the lunar lander, a bike, and a car. The lunar lander provides the simplest possible example, with a 2D viability envelope, and is useful for illustrating the basics of the approach, as well as the effect of various parameters. The bike provides a more complex, 3D envelope, and tests performance in the presence of more complex dynamics. Finally the car agent applies the method amid static obstacles.

Table 5.4 summarizes the relevant parameters of the tests: number[4] of NN samples (i.e., $||\widehat{\mathcal{X}}_{in} \cup \widehat{\mathcal{X}}_{out}||$), discretization of control space, time horizon, and grace period. The last two are expressed in terms of the number of simulation time-steps, which measured 1/30 s. It is worth pointing out that these parameters do not represent minimal or optimal values, but merely reflect some reasonable initial choices. In fact, many of these scenarios could run adequately with much smaller sample counts and control discretizations. Similarly, larger time horizons could be used without sacrificing

[4] The NN sample count listed is that after `dump_redundant()` has been called; initially the envelopes start with a million uniformly distributed samples. In general, prior to removal of redundant samples, the ratio of viable to nonviable samples reflects the ratio of viable to nonviable volume. After the samples are filtered, they are roughly of equal proportion, with the total number of samples being roughly proportional to the surface area of the decision surface.

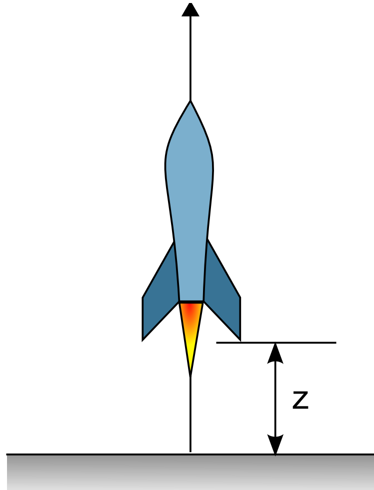


Figure 5.10: The lunar lander agent, modeled as a toy rocket constrained to the vertical axis. z is the rocket's altitude.

interactive rates; the values shown were chosen for reasons of obstacle approachability, as discussed later.

Table 5.1: Summary of relevant scenario parameters

scenario	# samples	$\lvert\widehat{\mathcal{U}}\rvert$	T_h	T_{gr}	Δt
lunar lander	1,870	9	30	1	$1/30$ s
bike	80,794	5×5	10	1	$1/30$ s
car @ track	126,467	9	10	1	$1/30$ s
car @ circles	224,772	9	10	1	$1/30$ s
car @ triangles	272,713	9	10	1	$1/30$ s

5.4.1 Testing platform

The implementation was tested on a 2.4 GHz Pentium IV machine, with simulation and safety checks being applied at a frequency of 30 Hz.

5.4.2 Lunar lander

The lunar lander, as introduced earlier, is a simple one dimensional motion system, in which a toy rocket is constrained to fly along a single vertical axis z. It has a bounded force thruster at the bottom, and it must avoid altitudes below $z < 0$, as this represents a collision with the ground. It is illustrated in Figure 5.10.

The rocket is subject to Earth's gravitational pull g, has a mass $m = 1$ kg, and has a maximum

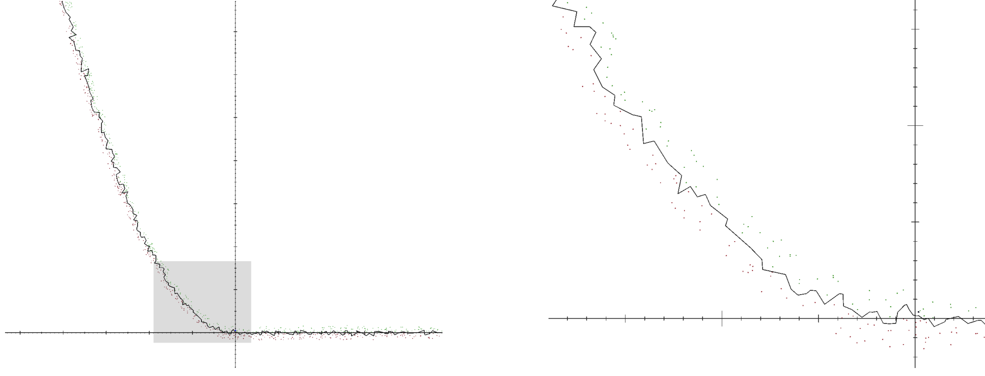


Figure 5.11: The derived lunar lander envelope. Plots show the agent's state-space, with altitude on y-axis, and vertical velocity on x-axis. The right subfigure shows a magnified view of the envelope near the origin (i.e., the part most relevant to a soft landing). See also Figure 5.8.

upward thrust $F_{max} = 20\,\mathrm{N}$. Motion is computed using rudimentary physics, namely

$$\ddot{z} = \frac{m}{F} \tag{5.13}$$

$$z = \dot{z}\Delta t + \frac{1}{2}\ddot{z}\Delta t^2 \tag{5.14}$$

with $\Delta t = 1/30\,\mathrm{s}$.

To derive the NN envelope, the agent's state-space was sampled in the range $-20\,\mathrm{m} < z < 200\,\mathrm{m}$, $-100\,\mathrm{m/s} < \dot{z} < 100\,\mathrm{m/s}$ using 100,000 uniformly distributed states. The viability of these states was derived analytically[5]. The samples were than normalized so that each dimension spanned the interval $[0, 1]$, and then filtered for redundancy, leaving 1870 samples, 935 of which were viable, and another 935 which were nonviable. Figure 5.11 shows the resulting envelope.

Figure 5.12 shows the resulting, safety constrained motion of the lunar lander using $T_h = 30$ (i.e., 1 second). In all trials $\widehat{\mathcal{U}}$ was obtained by uniformly sampling the allowable range of thrust (i.e., $[0\,\mathrm{N}, 20\,\mathrm{N}]$). The curve represents the rocket's altitude through time (age increases to the right). In this trial the user applied bang-bang control, initially requesting maximum thrust for a short duration, and then cutting it off completely. The deceleration and bringing to a hover of the agent is thus the result of the safety enforcement mechanism. The larger look-ahead duration (at least compared to values used later) results in a milder deceleration, evident by the milder slope of the left half of the curve.

The simplicity of the lunar lander envelope allows for clear illustrations of the effect of various parameters. For example, Figure 5.13 shows the effect of varying the time horizon T_h. It is worth noting that the case $T_h = 1$ corresponds to the single-step look-ahead discussed near the beginning of the chapter, and how poorly it handles the noisy envelope: in the corresponding plot the agent

[5] A number of additional tests bypassed NN modeling of the envelope and used the analytic test directly, yielding superior performance. But since the key purpose of this scenario here is to demonstrate the general approach, we follow through with all the steps.

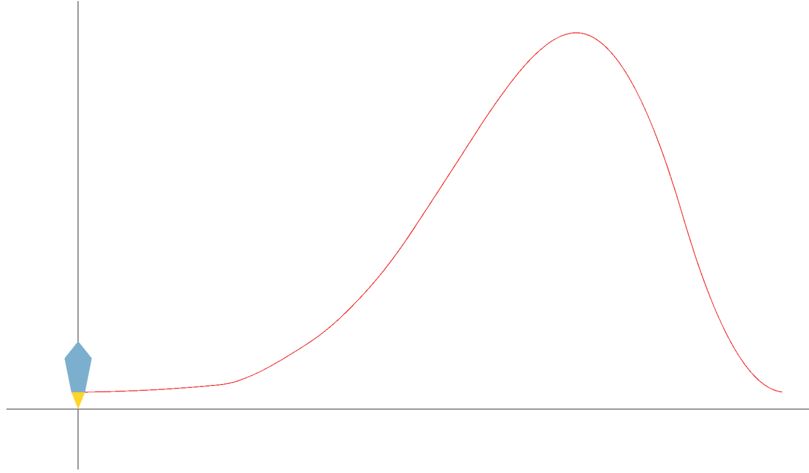


Figure 5.12: Lunar lander trajectory; the curve represents a plot of the agent's altitude over time (time advances to the left). In this trial the user applies maximum thrust from rest, and shortly cuts it (at the right inflection point); the safety-enforcement is soon triggered and turns on soon after the peak of the curve. Time-horizon T_h was set to 30 time-steps, which equates to 1 second.

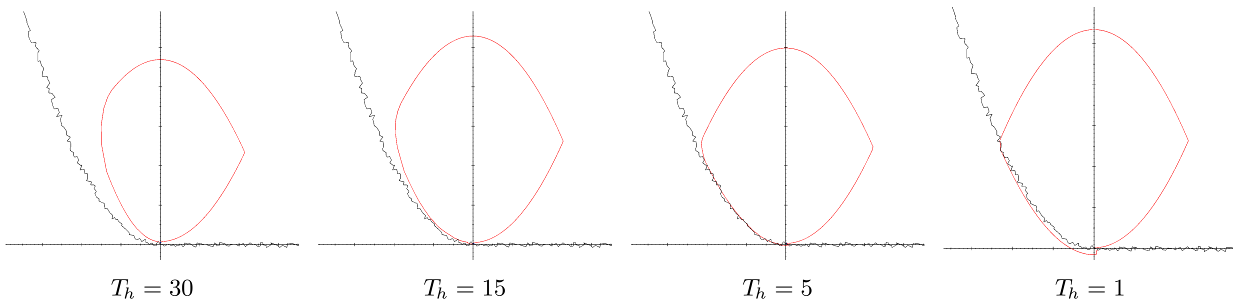


Figure 5.13: The effect of T_h on lunar lander trajectory. Plots show the agent's state-space, with altitude on y-axis, and vertical velocity on x-axis. In all cases $|\widehat{\mathcal{U}}| = 9$.

has left viable space and is unable to re-enter it, culminating in a collision ($z < 0$).

Figure 5.14 on the other hand illustrates the effect of the size of $|\widehat{\mathcal{U}}|$ on the agent trajectory. The main effect, as one would expect, is that it leads to trajectories of a more discrete nature. The case of $|\widehat{\mathcal{U}}| = 2$, where either full thrust or zero thrust is applied, is noticeably distinct, in that once the agent reaches higher elevations, it no longer can come arbitrarily close to the ground. This is a side-effect of the highly conservative response strategy to the $\mathcal{L}_1$ threat level, where the control action chosen is the one which does not incur an envelope breach within T_h: in the case of $|\widehat{\mathcal{U}}| = 2$, since "zero thrust" leads to a breach within T_h at sufficiently low altitudes, the "full thrust" control action must be applied. This hovering altitude is thus a direct function of T_h, with lower values yielding a lower hover height. Increasing the discretization of $|\widehat{\mathcal{U}}|$ makes the effect quickly disappear (it is barely noticeable in the $|\widehat{\mathcal{U}}| = 3$ case).

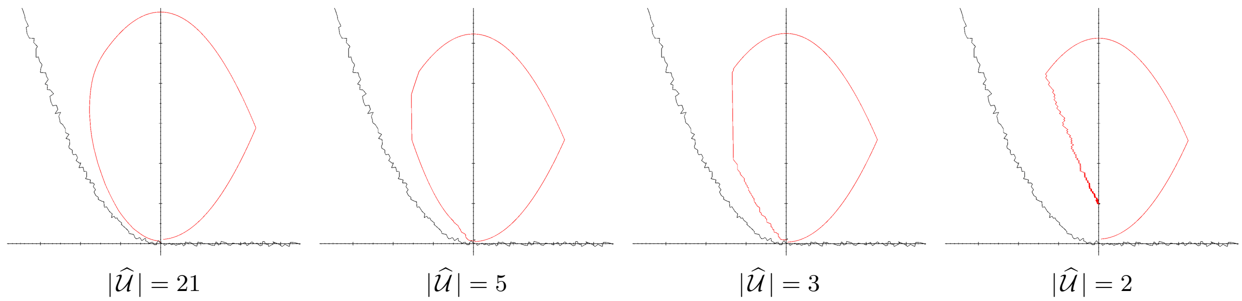


Figure 5.14: The effect of $|\widehat{\mathcal{U}}|$ on lunar lander trajectory. Plots show the agent's state-space, with altitude on y-axis, and vertical velocity on x-axis. In all cases $T_h = 30$.

5.4.3 Bike

This experiment was intended to gauge system responsiveness with a more complex kinodynamic system. Here the bike is in an obstacle-free environment, and the safety enforcement system must ensure that the bike's lean does not exceed a safe limit (to prevent falling). The maximum lean angle is set to $\phi_{max} = \pi/3\,\mathrm{rad}$, as measured from the vertical.

The bike agent used in these experiments is a slightly modified version of the agent used in the previous two chapters. The main difference is that the bike's state is extended to include forward speed, yielding

$$\vec{x} = (x, y, \theta, \phi, \dot{\phi}, V), \tag{5.15}$$

where V is the forward speed of the bike, and it is bounded such that $0\,\mathrm{m/s} \leq V \leq 10\,\mathrm{m/s}$. The user controls were correspondingly extended to include the bike's acceleration, giving

$$u = (\psi, \dot{V}). \tag{5.16}$$

The user can directly specify an acceleration in the interval $-0.5\,\mathrm{m/s^2} \leq \dot{V} \leq 0.5\,\mathrm{m/s^2}$, while the steering angle must lie in $-\pi/4\,\mathrm{rad} \leq \psi \leq \pi/4\,\mathrm{rad}$. The simulation and safety enforcement run at 30 Hz.

There are a number of ways to approach the discretization of a multi-dimensional $\mathcal{U}$. For the bike we simply discretize $\mathcal{U}$ into a uniform 5-by-5 grid, yielding $|\widehat{\mathcal{U}}| = 25$. Thus the system may enforce safety by correcting steering angle and/or linear acceleration of the bike.

The bike envelope was constructed by first obtaining one million[6] sample states, uniformly

[6]This large value was chosen to yield a detailed visual representation, and ensure that no minor features were hiding in the envelope noise. Equivalent performance can be obtained by starting with only 20,000 samples.

distributed over the region of state-space defined by the following intervals[7]:

$$\begin{aligned} -1.5\,\text{rad} &\leq \phi \leq 1.5\,\text{rad} \\ -15\,\text{rad/s} &\leq \dot{\phi} \leq 15\,\text{rad/s} \\ 0\,\text{m/s} &\leq V \leq 10\,\text{m/s} \end{aligned} \tag{5.17}$$

The viability of the resulting samples was estimated using a simple routine which checks whether either maximum turn, $u = (-\psi_{max}, 0)$ or $u = (\psi_{max}, 0)$, can halt the lateral fall of the bike before it falls over. That is, the routine simulates agent dynamics for both control actions until either the bike falls over ($|\phi| > \pi/3\,\text{rad}$), or recovers (reaches state with $\dot{\phi} = 0$). If both extremal control actions lead to failure, the state is considered nonviable; if either control actions recovers the bike, the state is deemed viable.

The sample values were then normalized in each dimension, and then scaled in $(\dot{\phi}, V)$ by scale factors $(10, 3)$. Finally, redundant samples were filtered out to yield 80,794 training samples (39,085 viable and 41,709 nonviable). Figure 5.15 shows the resultant envelope.

Figure 5.16 shows a trial run of the bike under the safety constraint system. The user controls the bike with a mouse, with left-right motion directly controlling the steering angle ϕ, and up-down motion directly controlling the bike's acceleration. The mouse is presumed to operate within a rectangular area whose boundaries are mapped to the extremal values for ψ and $\dot{V}$, with in-between values computed using linear interpolation.

The envelope successfully maintains bike balance through steering, and through accelerating when speed falls too low. Since the shape of the viability kernel is simple, nearly equivalent performance can be obtained with much smaller sample sets; for example, relatively equivalent performance was achieved with envelope of 2570 viable samples and 2922 nonviable samples.

Such drive-by-wire control of the bike turns out to be very difficult, primarily due to lack of proprioception, and the fact that in such a rider-less system there is no way to laterally shift the agent's centre of mass, which is an important additional means of bike control in the real world (i.e., slight lateral leaning with hips and torso). Safety enforcement thus helps to compensate for this shortage and maintain system viability, although it can result in counter-intuitive behaviour. With sufficient speed, the centrifugal force tends to throw the bike into the opposite lean, where it is "caught" by the viability containment mechanism, thus establishing the opposite turn from that which the user desired. It is extremely difficult to get just the right manual combination of steering angle and velocity such that the centrifugal force is neatly balanced by gravity, especially when one aims to first establish a particular turning radius or direction of travel. Interestingly, with some practice a simple and effective strategy emerges for driving the bike under such a system: one first

[7] Since there are no obstacles in the environment, x, y, and θ are irrelevant, and thus the envelope spans the 3D subspace defined by $(\phi, \dot{\phi}, V)$.

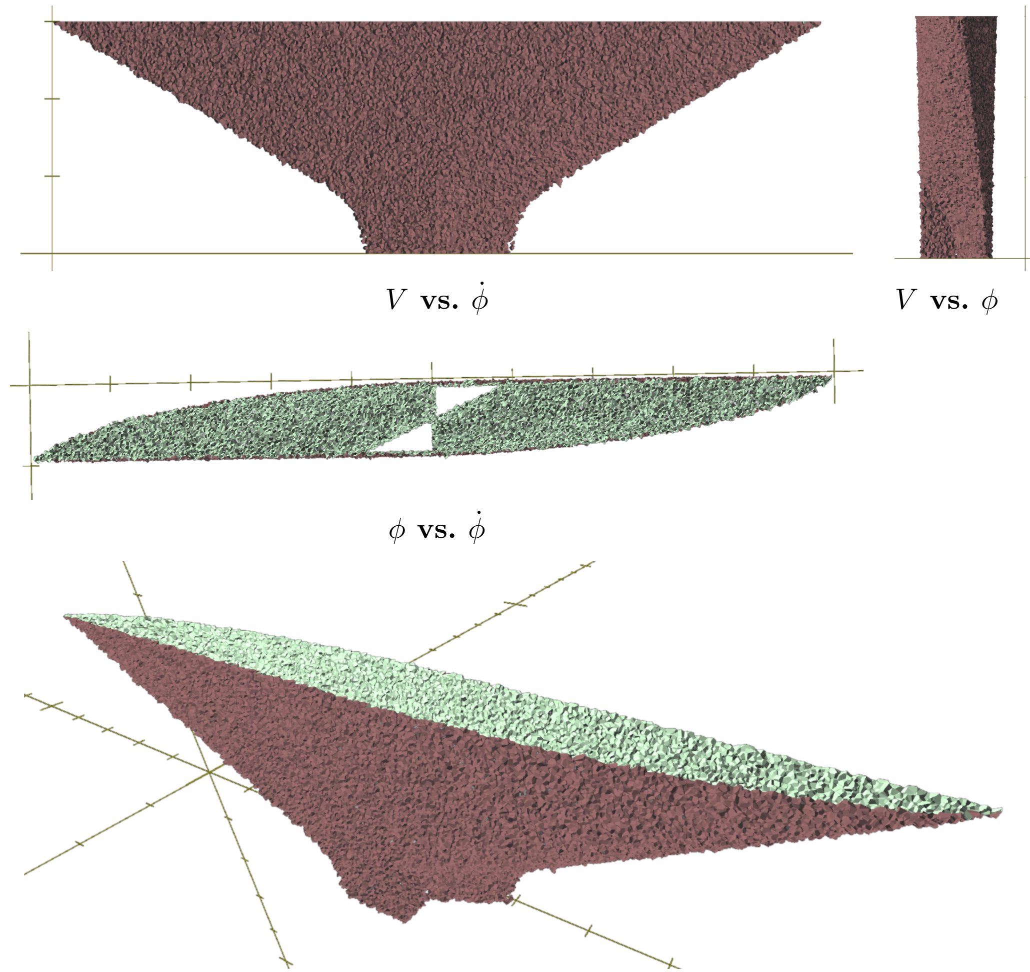


Figure 5.15: The derived bike envelope. The surface colour indicates direction of viable volume (green side faces viable space, red side faces nonviable space). The labels for the plots, "y vs. x", indicate what variable was plotted on each axis. The axes are shown only for orientation, and do not reflect true values (scaling and offsets were applied to the space).

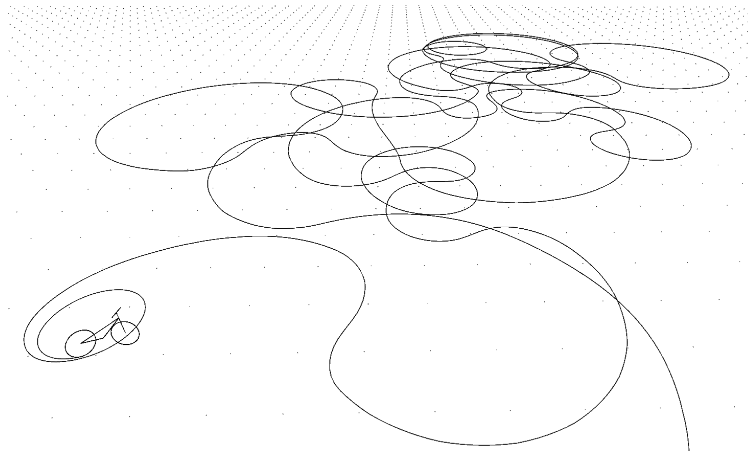


Figure 5.16: An example of a bike ride with safety enforcement engaged.

steers in the direction opposite to the desired one, applies acceleration (thus causing the bike to flip to the desired direction), and then modulates the turning radius through subsequent control of the agent's linear velocity. Driving exactly forward is extremely difficult, but can be emulated easily by snake-like motion through rapid weaving left and right.

5.4.4 Car

The purpose of this last set of experiments, using a simple car for an agent, was to assess performance of the system with arbitrary obstacles. The car used in these experiments is a slightly modified version of the agent used in the previous two chapters in that, in addition to steering angle, the user can also directly control the linear velocity of the car. That is,

$$u = (\psi, V), \tag{5.18}$$

where V is the forward speed of the car, which is constrained to lie in the range

$$-30\,{}^{\mathrm{m}}\!/\!_{\mathrm{s}} \leq V \leq 30\,{}^{\mathrm{m}}\!/\!_{\mathrm{s}}, \tag{5.19}$$

while the steering angle must lie in

$$-{}^{\pi}\!/\!_{4}\,\mathrm{rad} \leq \psi \leq {}^{\pi}\!/\!_{4}\,\mathrm{rad}. \tag{5.20}$$

The simulation and safety enforcement run at 30 Hz.

Similar to the bike, $\mathcal{U}$ for the car is also two-dimensional. Here though we employ a different strategy for discretizing the control space: rather than discretizing along both dimensions, we only discretize the steering angle, while agent velocity in all the look-ahead trajectories is set to the one currently selected by the user. That is, in the look-ahead tree for a state x_k, there are n trajectories, each of which is derived using a control action in the set

$$\widehat{\mathcal{U}} = \{\, (\psi_1, V_k),\ (\psi_2, V_k),\ \ldots,\ (\psi_n, V_k)\,\}, \tag{5.21}$$

where V_k is agent velocity at time-step k, and set to the second element of u_k. Thus $|\widehat{\mathcal{U}}| = n$, and $\{\psi_1,\ \psi_2,\ \ldots,\ \psi_n\}$ are the steering angles uniformly sampled from the allowable range. The benefit of this approach is that, for the same amount of computational effort, a much finer discretization of steering angles is possible, compared to discretization across both dimensions. This works well for the car agent because it is not a kinodynamic system; modulating the agent's velocity merely alters the rate at which the trajectory is traversed. In contrast, altering the bike's velocity in this fashion would have had a very large effect on its balance, and thus the final trajectory achieved.

In all the car scenarios a simple test procedure is used as the viability oracle: a state is pro-

nounced viable if the car can complete a collision-free circle using either the maximal left or maximal right steering angle. This is fast and works well with the tested environments, but will break down in more constrained environments, such as corridors where the car cannot turn around but can maintain viability by following the corridor.

In constructing the envelopes, the samples are first normalized in each of their dimensions, then scaled along (x, y, θ) by $(4, 4, 1)$, and finally filtered with `dump_redundant()`.

Environment: track

Figure 5.1 shows a trial run of safety enforcement for a car on a rectangular track with rounded corners. The dimensions of the environment are provided in Figure 5.17. The user is able to interactively steer the car at will but is prevented by the system from leaving the track. Figure 5.2 shows how the safety constraints project onto the control action space for this problem. The user input v_k consists of a sequence of left and right turns of the steering wheel, as represented by the continuous curve in the graph. The shaded regions correspond to control actions which make the agent approach the envelope (i.e., ones for which $T_{eb} \leq T_h$). The applied control input u_k is computed as given by Equation (5.7) and is represented by the blue, discrete curve (an artifact of the discretization of $\mathcal{U}$). The labels 'a' through 'j' indicate contemporary time-points in the two plots. The inter-label spacing varies between the plots since the car was moving at various speeds, which has a direct effect only on the world-space plot.

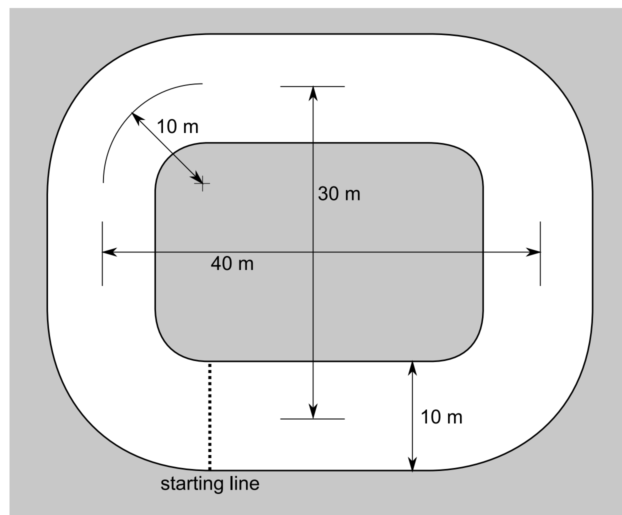


Figure 5.17: The "track" environment.

The envelope was derived by first uniformly sampling one million states in the volume[8]

$$\begin{array}{rcccl} -25\,\mathrm{m} & \leq & x & \leq & 45\,\mathrm{m} \\ -15\,\mathrm{m} & \leq & y & \leq & 45\,\mathrm{m} \\ -\pi\,\mathrm{rad} & \leq & \theta & \leq & \pi\,\mathrm{rad} \end{array} \tag{5.22}$$

The viability status was computed for this set, and then filtered for redundancy, leaving an envelope model with 126,467 samples (64,515 viable, 61,952 nonviable). Figure 5.18 illustrates the resultant model.

As one might expect, the envelope mirrors the shape of the track. The main interesting feature is a sharp ridge that winds around both, the inner and outer surfaces, and shifts along the θ axis whenever the direction of the roadway changes. The ridge originates in the fact that viability places limits on how close the car can safely approach a wall or boundary, as a function of their mutual orientation. This function is strongly correlated to the trajectory of an evasive manoeuvre being applied at the last possible instant. As Figure 5.19 illustrates, the cusp occurs when the agent is directly perpendicular to the roadside, since this requires the most anticipation and the longest evasive manoeuvre.

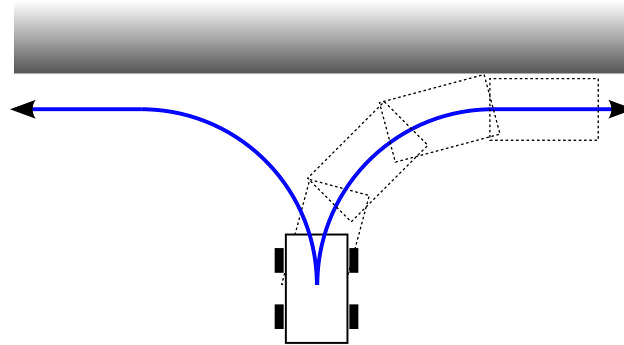


Figure 5.19: How close a car can safely approach a boundary depends on their mutual orientation, with the extremal value occurring when the agent is perpendicular to the wall. As the car's orientation is varied, the "minimum safe distance" curve traces out the observed notch pattern seen in the envelopes.

This ridge shifts up and down along the θ-axis when the track turns because the ridge's position in the envelope depends on the car's orientation relative to the roadway, whereas θ measures the car's absolute orientation angle (i.e., with respect to x-axis).

Careful examination of the envelope surface shows additional but minor variation along the θ-axis, over orientations in which the car faces away from the environment boundary. These variations are a consequence of the car's rectangular geometry; they would be absent if the car were circular or a point. They indicate that at some angles (relative to a wall) the car's rear corners protrude more, thus requiring the car to be offset correspondingly to avoid environment penetration.

[8]Note: the environment's origin lies in the centre of the staring line.

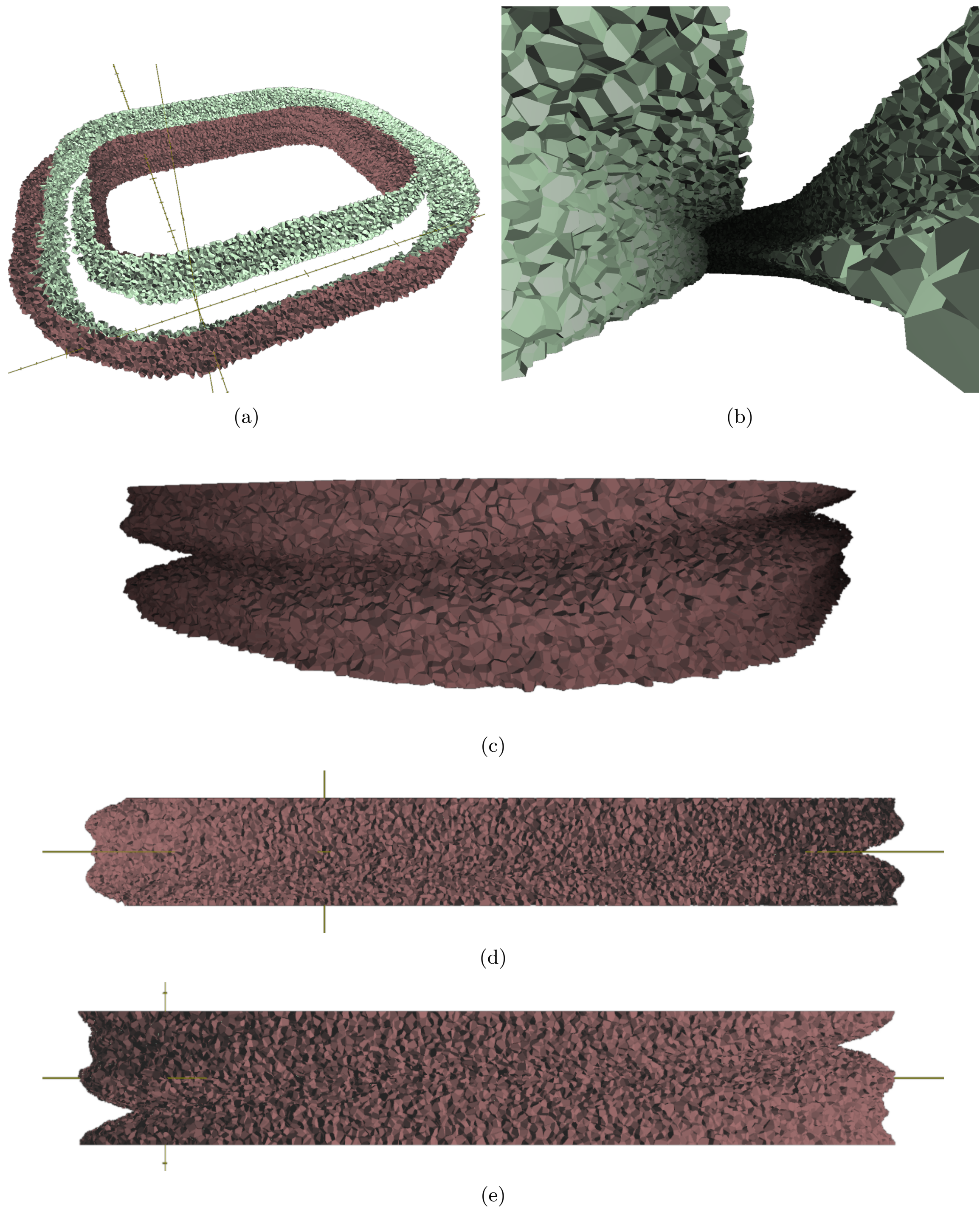


Figure 5.18: The derived envelope for car on track. **(a)** perspective view of the envelope (z axis corresponds to car's orientation); **(b)** view from the starting line; **(c)** view of one of the "corners" in the envelope from nonviable space; **(d)** orthogonal-projection view from nonviable space of one of the long stretches of the track; **(e)** orthogonal-projection view from nonviable space of one of the shorter stretches of the track.

Environment: circles

This environment, as shown in Figure 5.20, features four circles arranged in a square pattern, and is intended to check how the system responds to smaller, island-like obstacles.

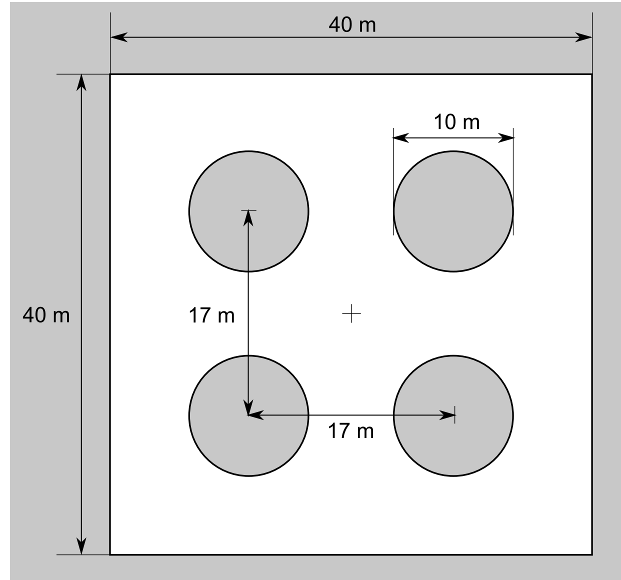


Figure 5.20: The "circles" environment.

The envelope was derived by first uniformly sampling one million states in the volume

$$\begin{array}{rcccl} -25\,\mathrm{m} & \leq & x & \leq & 25\,\mathrm{m} \\ -25\,\mathrm{m} & \leq & y & \leq & 25\,\mathrm{m} \\ -\pi\,\mathrm{rad} & \leq & \theta & \leq & \pi\,\mathrm{rad} \end{array} \tag{5.23}$$

The viability status was computed for this set, and then filtered for redundancy, leaving an envelope model with 224,772 samples (116,012 viable, 108,760 nonviable). Figure 5.21 illustrates the resultant model, as well as examples of safety enforcement with this model.

The most notable feature of this envelope is the "corkscrew" threading on the extruded circular obstacles in the state-space. This is simply a different manifestation of the ridge noted in the previous environment (see Figure 5.19). It may appear novel here because it winds continuously and at a constant rate; this is due to the circular obstacle boundary behaving similarly, changing orientation in a continuous and constant rate fashion.

Environment: triangles

This final environment, as shown in Figure 5.22, was meant as a (mild) stress test. It did not result in a particularly difficult challenge, suggesting that the approach could be applied to larger and more complex environments, especially considering how additional tests in these environment

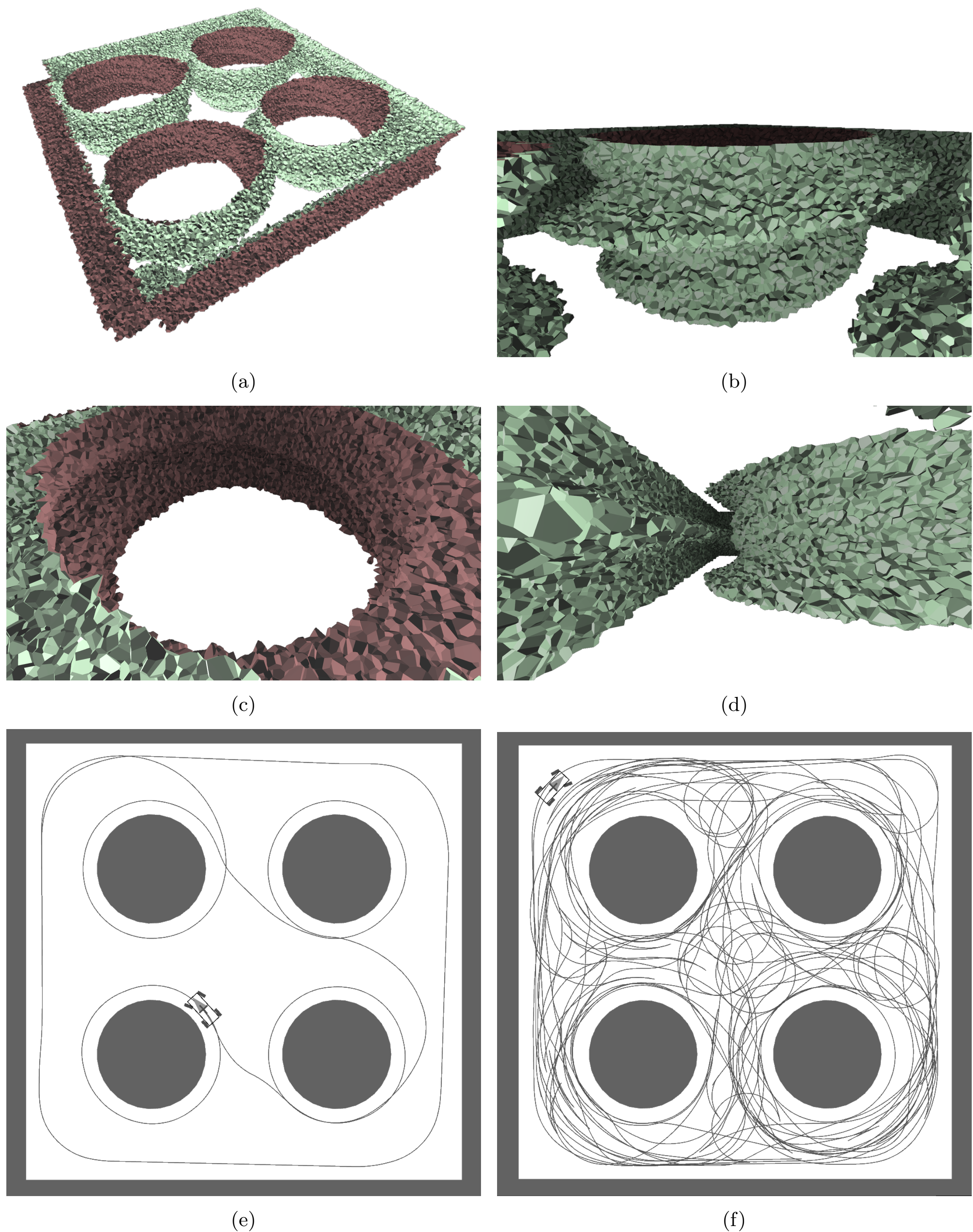


Figure 5.21: The "circles" environment. **(a)** perspective view of the envelope (z axis corresponds to car's orientation); **(b)** the "corkscrew" effect due to circular nature of obstacle; **(c)** the corkscrew from the nonviable side; **(d)** the intertwining of corkscrew threads; **(e)** car driving at moderate speed, hugging all obstacles as much as the envelope and T_h will permit; **(f)** car driving at various speeds, including backwards, with random to "suicidal" steering decisions.

resulted in comparable performance with much sparser envelope sample sets.

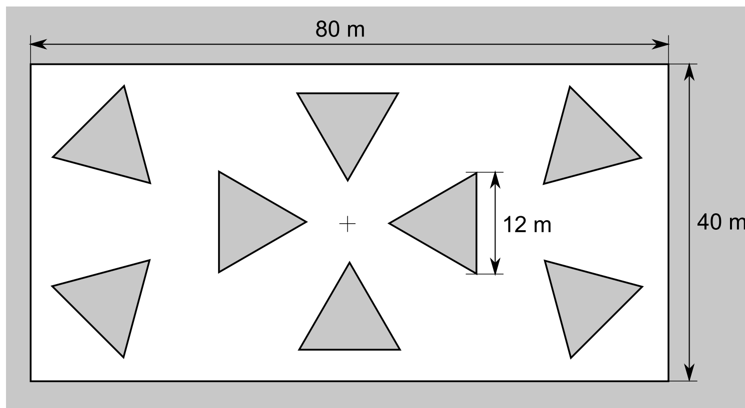


Figure 5.22: The "triangles" environment.

The envelope was derived by first uniformly sampling one million states in the volume

$$\begin{array}{rcccl} -45\,\mathrm{m} & \leq & x & \leq & 45\,\mathrm{m} \\ -25\,\mathrm{m} & \leq & y & \leq & 25\,\mathrm{m} \\ -\pi\,\mathrm{rad} & \leq & \theta & \leq & \pi\,\mathrm{rad} \end{array} \tag{5.24}$$

The viability status was computed for this set, and then filtered for redundancy, leaving an envelope model with 272,713 samples (136,661 viable, 136,052 nonviable). Figure 5.23 illustrates the resultant model, as well as examples of safety enforcement with this model.

The surfaces in this model display the same ridges as encountered earlier. The main new topological feature is the appearance of "holes" that connect nonviable areas together. These holes embody the fact that in narrow corridors some car orientations are impossible to recover from, because it is impossible to achieve a safe distance from both walls simultaneously (for that particular car orientation). For example, there is no safe way to safely place the car inside and perpendicular to the narrow corridors in this environment since the car's limited steering ability is insufficient to allow a turn that will avoid the wall being faced.

5.4.5 Haptic feedback experiments

Finally, some simple haptic experiments were performed using the car agent on the "track" environment. A "PHANToM" haptic device was used to both, control the car and provide haptic feedback. The user controlled steering using the horizontal axis of the PHANToM's end effector, and velocity with the vertical axis. Since the car's envelope containment mechanism implements corrections through the steering angle only, force feedback was thus present only on the horizontal

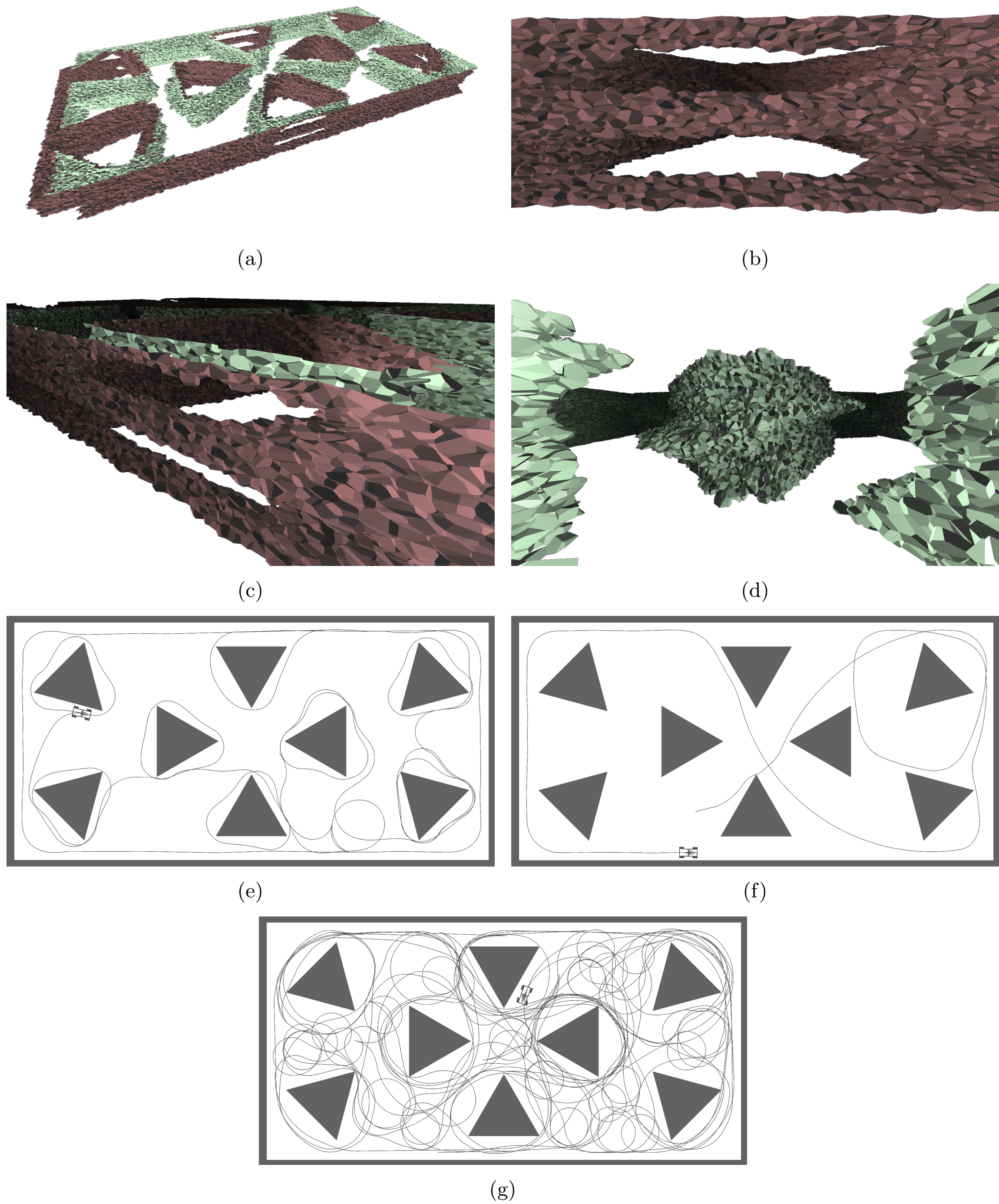


Figure 5.23: The "triangles" environment. **(a)** perspective view of the envelope (z axis corresponds to car's orientation); **(b,c)** "holes" form in the envelope where corridors are too narrow to allow the car to reach perpendicular orientations; **(d)** corkscrew-like threading seen on edges of triangles; **(e)** car driving at moderate speed, hugging all obstacles as much as the envelope and T_h will permit; **(f)** "stuck steering wheel" motion, where the user keeps v_k constant, set to a mild right turn at medium velocity; **(g)** car driving at various speeds, including backwards, with random to "suicidal" steering decisions.

axis. The horizontal force was computed using a spring-and-damper system:

$$\psi_{err} = \psi_{safe} - \psi_v \tag{5.25}$$

$$F_x = K_p \, \psi_{err} - K_d \, \dot{p}_x \tag{5.26}$$

ψ_v is the user-selected steering angle, ψ_{safe} is the one deemed safe by the system, and p_x is the position of the PHANToM's end effector along x-axis. These three variables are first normalized to the range $[-1, 1]$ prior to use. The constants K_p and K_d are the usual proportional and derivative coefficients; various values were tested, but the ideal settings are very subjective and, furthermore, dependent on the PHANToM device being used.

The linear nature of the above haptic strategy proved to be problematic. Considering the relatively short time horizon used for look-aheads with the car (to allow closer approach to roadsides), a fair amount of minimum "stiffness" was desirable, whereas the linear model above tended to result in either excessively weak corrections for small ψ_{err}, or excessively aggressive corrections for larger ψ_{err}, depending on the value of K_p. Further experimentation showed that

$$F_x = K \sqrt[3]{\psi_{err}} \tag{5.27}$$

provides a more natural feel. This model could likewise be extended with a damping term (i.e., $K_d \, \dot{p}_x$), but such velocity control did not prove necessary in the tests performed.

5.5 Discussion

5.5.1 Computational load and complexity

The amount of extra effort required to enforce viability depends on a number of factors: the frequency of safety checks (i.e., the assessment of threat level and the application of corresponding control law), the size of $\widehat{\mathcal{U}}$, the magnitude of the time horizon, and the cost of querying the viability model. Ideally one would perform safety checks as often as possible, but acceptable performance can be obtained by applying them at reasonable intervals (e.g., $1/30$ s). If the oracle's method of divining the viability of samples operates in a discrete fashion, it is generally advisable to at least match its time-step and set of control actions, when applying the derived model. Consider, for example, an oracle which determines viability of an agent state by demonstrating a viable trajectory out of the state. If a larger time-step or a more limited control set is used when applying the subsequent model, these viable trajectories might no longer be reproducible under the more restrictive parameters, causing the model to be misleading. It is also worth noting that the oracle's use of discrete methods results in a more conservative viability model, with larger granularities in time and control discretization leading to larger safety margins.

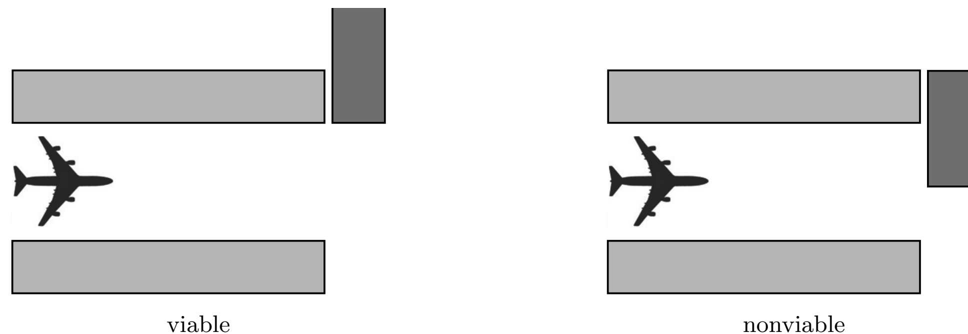


Figure 5.24: Minor obstacle motion can easily alter topology, with drastic impact to viability. In the left diagram the path through the tunnel is viable; in contrast, in the right diagram the minor obstacle translation renders the tunnel nonviable since a collision-free exit is no longer possible.

The cost of a single safety check consists mostly of the computation of the look-ahead. In threat level $\mathcal{L}_0$ the worst case cost is that of the full look-ahead trajectory for the user's selected control action v_k, namely $O(T_h\, c)$, where c is the cost of a single viability model check.[9] In $\mathcal{L}_1$ and $\mathcal{L}_2$ the worst case involves additionally computing the full look-ahead tree, at cost $O(|\widehat{\mathcal{U}}|\, T_h\, c)$, since there are $|\widehat{\mathcal{U}}|\, T_h$ nodes in the full tree of depth T_h, aside from the starting root node. Although in $\mathcal{L}_3$ the search for re-entries extends the look-ahead to $2\,T_h$, this merely doubles the cost, leading to the same time complexity as for $\mathcal{L}_1$ and $\mathcal{L}_2$. Thus the overall time-complexity of the method is linear in all the relevant factors.

5.5.2 Viability model

Static environment assumption

The current approach to modeling viability assumes that the environment is static. Even very minor variations in obstacle placement can alter the topology of traversable space, and thus have drastic impact to the viable landscape, as demonstrated in Figure 5.24. Such changes occur suddenly and discretely, regardless of the speed of obstacle motion or the number of obstacles involved.

In order to obtain a valid viability model for a dynamic environment one would have to compute it over the "space-time" $\mathcal{X} \times t$, the extension of the agent's state-space $\mathcal{X}$ with a time dimension, where obstacles extrude along the time axis, and obstacle motion modulates the path of extrusion. That is, the computation of the model would have to take into account space *and time*. However, doing so, and by extension working out an exact viability model for a dynamic environment, is not practical for a number of reasons. One key problem is that this requires full knowledge of future obstacle motion, from now to eternity. Furthermore, the above considers only the online computation of the viability model, for a particular future motion of the obstacles; if a precomputed model is desired, which is essential to interactive safety enforcement, it would have to be further

[9] Technically, c is a constant multiplier and can thus be omitted in big-O notation; we leave it in for the sake of clarity.

parameterized by obstacle motion. In the general case this is completely intractable due to the number of dimensions to such a model, as well as the amount of pre-computation effort required, since viability would essentially have to be computed for each possible “future history” of obstacle motion.

Since exact models of viability for dynamic environments are generally not practical, a more feasible approach would be to approximate them instead, using much cheaper methods. “Local viability”, the sensor-based locally parameterized approximate model of viability from the previous chapter, is a promising candidate for this. As pointed out earlier, model error would be mostly confined to the indeterminate cases, where all potentially viable trajectories travel beyond the sensing range (e.g., highly constrained sections of the environment). The impact of such error can be partially mitigated by increasing the T_h look-ahead time: adding states to the tail of a look-ahead in effect boosts its combined sensing range, much like adding to a string of street lights extends the lit area during the night, and allows one to search further for obstructions on the road.

Representation

One prominent issue with the current viability envelope models is that the Nearest Neighbour (NN) technique induces a very noisy and jagged envelope. This proves troublesome for the containment mechanism, thus necessitating extra measures to mitigate this artifact, namely the “grace period” provision. A more direct solution to this issue would be to switch to a “smoother” classifier. In particular, with only minor adjustments the envelope could be modeled with k-NN. Initial experiments in this direction did not show sufficient improvements—such as reduced incidence of higher threat levels, or reduced reliance on grace period—to outweigh the increased computational demands of the classifier though.

A more pressing problem is that NN methods are generally not very scalable, thus limiting the dimensionality of models that we can encode with it. To some degree this is the reason why the test environments for the car were relatively small. The car’s envelopes were three-dimensional, the highest attempted. It is feasible that NN could be yet applied to 4 and perhaps even 5 dimensional spaces, but beyond that it is almost certain that the number of viability samples needed to adequately capture the envelope would be excessive, leading to impractical space requirements and classifier query time.

The Support Vector Machine (SVM) classifiers are known to scale well to higher dimensions, although using them is not as trivial nor transparent as using NN classifiers. SVMs would largely address the scalability problem, but it remains to be seen whether the derived models would interact well with our framework, or whether they would introduce new artifacts, which would need to be mitigated in additional ways. We note that others have considered using SVMs to model viability kernels, [CD06] in particular, but the experiments there were also low-dimensional (2D), and did

not involve application in an interactive setting, or one that involves the human control of the subjects. Our initial tests with SVMs did not produce immediate improvements nor noteworthy results, but this may be the result of insufficient testing.

A better solution to the problem might be to employ localized models of viability, as used in the previous chapter. Such models have more modest scaling behaviour, each model is applicable to a whole class of terrains, and the models work with partially (locally) observable environments. In particular, although the number of dimensions to the model is likely to be the same or comparable, the number of training samples and the information content of the model are significantly smaller, unaffected by the extents of the environment. Switching to localized viability models has several drawbacks though. The main issue is the discovery of a robust set of virtual sensors with which to localize the agent's state, a problem shared with the work of the previous chapter. Also, even if a fairly good set of sensors were to be found, the viability model is still likely to be imperfect and can thus inject some error into the threat assessment mechanism; in a safety enforcement context this carries grave consequences, and is far more undesirable here than in the previous chapter.

5.5.3 "Constant u across T_h" assumption

When gauging the threat exposure resulting from a particular control action, the system assumes that the control is going to be held fixed throughout the time horizon T_h. This is primarily a concession made to maintain the computational feasibility of look-ahead. Ideally, one would determine the viability of a control action u_k from state x_k by considering the complete progeny tree rooted at the resultant x_{k+1}, the whole family of trajectories resulting from the application of all possible control action sequences within the time horizon T_h. This is clearly impractical for non-trivial time horizons, due to the exponential growth in the number of tree nodes. By assuming that the control action is going to be held fixed throughout the time horizon we thus get linear scaling in T_h, which is more acceptable.

In a way then, when the system gauges the viability of the control actions from a particular agent state x_k, it limits itself to a single exploratory trajectory for each possible $u_k \in \widehat{\mathcal{U}}$. This is not an unreasonable concession; the assumption of the same control action for all subsequent time-steps constitutes choosing the control sequence with the highest probability. That is, it is the best guess as to what the user is going to do next. In most applications the user is likely to be "coasting" most of the time (i.e., control input held in a fixed, neutral position). Also, course alterations may entail a fixed, non-neutral control action (e.g., executing a fixed-radius turn in a car involves maintaining a particular deflection of the steering wheel), and even when the control action varies it may often appear as nearly constant over the span of the time horizon, due to human aesthetics and tendency to prefer smooth and minimal control.

The fixed control action approach also makes sense when haptic feedback is considered. Since

the purpose of such feedback is to alert the user of future constraints on control actions, and since human reaction time must be factored in along with other delays required to understand and process the provided information, the feedback must relate to a larger time perspective. For example, it would be pointless to convey to the user that a particular control action is safe and desirable if this is contingent upon the control action being changed at the very next time-step, since the user would not be able to comprehend and respond at this speed.

Finally, the constant-control assumption also mimics the "generalized inertia principle" employed in viability theory.

The assumption does entail several drawbacks. Primarily it increases the chance of inappropriate breach detection and response. In particular, in many cases the agent might be truly viable but the evasive action needed may rely on applying different control actions in succession (e.g., when the agent finds itself in a serpentine corridor). The fixed-u look-ahead tree will thus not find the required evasive action, causing the system to be pushed into level $\mathcal{L}_2$. This undesirable since the response strategy for the latter level is not guaranteed to lead to the correct evasive action.

5.5.4 Envelope margin of safety

Considering the dire consequences of error, it may at first seem prudent and beneficial to use more conservative envelopes, ones in which the decision surface has been displaced toward viable space by some constant margin of safety (i.e., where the viable volume of the model has been uniformly shrunk). In the multi-step look-ahead scheme this could serve to reduce the rate of incidence of the more severe threat levels. In the single-step containment approach a more conservative envelope model would likely lead to greater robustness and resilience to envelope breach.

There are a number of open problems associated with this idea though, and its desirability must be weighed against user expectations. First, it is unclear how best to perform the envelope contraction: should it be uniform or should it vary across the decision surface? If variable, then how and where should it vary? In either case, how should the magnitude of the safety margin be determined? An important drawback of using a contracted envelope is that it may lead to conflict with user expectations and diminished aesthetic value: a contracted envelope will result in the user being unable to approach environment boundaries as closely as they might expect or desire to, sometimes by a significant amount. For some agents large safety margins may not be objectionable, such as for airplanes avoiding buildings and other airplanes, but for others they are undesirable. In our experiments even moderate safety margins for the lunar lander and the car were particularly noticeable and irritating.

5.5.5 Drawbacks and limitations

A key requirement for this system is the availability of a model of the agent's dynamics, so that its state may be predicted relatively accurately for some time into the future. Often one may only have a simplified model of system dynamics, and additional error may be injected by physical world limitations, such as inaccurate sensors and agent actuators. The system's resilience to such error has not been tested yet, and would likely necessitate incorporating a significant safety margin in the viability envelope model, as discussed, or some similar measures.

The time horizons used in the experiments were relatively small, ranging between $1/3$ s to $1/2$ s. Their choice was precipitated by the very cautious control strategy used for threat level $\mathcal{L}_1$, as laid out in Equation (5.6), in which the agent steers away at the first hint of trouble (i.e., as soon as $T_{eb} \leq T_h$). This directly limits how close one can approach obstacles, and leads to very constrained motion, especially under high speeds. In the car experiments this was particularly pronounced, and further exacerbated by the relatively small test environments. Reducing the time horizon provided greater freedom of motion, but sacrificed mildness of corrections and the haptic feedback lead time.

5.6 Future work

A key limitation of viability envelopes presented in this chapter is their tight binding to the environment in which they were constructed (i.e., lack of transferability to other environments). The most obvious next step would be to employ localized viability models, like those derived in Chapter 4. This would provide a far more versatile solution, and would automatically enable application to partially observable environments.

It would be worthwhile to establish some assurances of safety, especially if the method were to be applied to physical systems rather than simulation. Currently the models are not conservative and contain significant noise, which makes them undesirable for real-world applications where safety is a key concern. Although it seems unlikely that full guarantees of safety could be obtained, considering that the approach relies on empirically collected data and models learned using machine learning techniques, even partial guarantees can be useful sometimes, and once characterized it is probable that some progress could be made to strengthen them.

Even the extant, global envelopes could be improved by using a more resilient learning method that provides smoother boundary surfaces. As discussed in Section 5.5.2, Support Vector Machines techniques need to be retested as it would seem this is the most promising alternative at the moment.

Although the assumption of fixed-control inputs makes sense when the haptic component is viewed as an "early warning system", this sometimes presents challenges to maintaining the system within viable space. In particular, in the $\mathcal{L}_2$ threat level the agent is viable but all the possi-

ble evasive actions require application of non-constant control actions. The current, fixed-control lookahead approach may therefore mistakenly lead the system astray. It would be worthwhile to explore some more intelligent ways to handle such scenarios.

Finally, although some preliminary experiments with haptic feedback were performed, this aspect of the work is still largely unexplored. Many questions remain, such as how best to moderate the haptic feedback, or what is the optimal time horizon to use.

Chapter 6

Conclusion and future work

The work in this thesis has aimed at improving the efficiency of current motion planners, especially for a particularly difficult class of agents. The concept of viability played a key role in this, and was further exploited for safety enforcement. More specifically, the modifications to the RRT algorithm presented in Chapter 3 have yielded a more robust planner for agents with differential constraints, particularly in highly constrained environments. Chapter 4 introduced viability filtering as an important tactic for avoiding wasted effort, namely the searching through regions of space unlikely to lead to a solution, thus further improving planner runtimes. Finally, Chapter 5 introduced a control-assistance mechanism for user-control of agents, virtual ones in particular, that forestalls getting trapped in situations of unavoidable failure.

There are a number of potential directions for future work. In this chapter we look at a number of the most prominent, larger scope ones.

6.1 Motion planning with macro-primitives

As discussed near the beginning of this thesis, current motion planning methods start to stumble when the dimensionality or complexity of the problem exceeds some moderate limits. The research direction that has the potential to make the greatest strides in overcoming this, namely making the more complex problems tractable, is Motion Planning with Macro-Primitives (MPMP): the greatest hope in transcending the curse of dimensionality lies in approaching the problem at a more abstract level, by planning and deliberating using macro-primitives, building blocks of motion with semantic value. In fact, there is evidence in neuroscience (e.g., [MIB00, MIS04]) that human motion, and that of other biological organisms, relies on a similar mechanism, wherein motion is realized through the sequencing and superposition of such primitives or control programs. This, in effect, acts as a means to reduce the dimensionality of the input problem, allowing the organism to successfully execute much more complex tasks.

This idea has been successfully exploited in control systems and related fields, as in [Bro86, Ark98, Tan03, MIS04, Fal02]. Efforts to migrate this idea to motion planning have been minimal so far. The most relevant work is [KBM04], which presents a method for a trivial biped robot to solve for and execute appropriate stepping motions that climb and traverse stair-like structures in a 2D world. Although an interesting start, the greatest benefits of MPMP are to be reaped with agents on the other end of the complexity spectrum, such as a 40 DOF kinodynamic human figure, for example, which severely strain today's planners. MPMP is thus still largely unexplored and needs significant further research effort to resolve the many associated open problems. The work in this thesis represents a precursor and a useful stepping stone toward that larger goal, with Chapter 4 in particular providing some useful insights and hints on how key pieces of this approach could be implemented.

The use of motion macro-primitives for motion planning presents some novel challenges that are not present in the above control system contexts. In particular, motion planning problems typically grapple with the presence of obstacles, whereas most control problems above assume a barren environment. This ties into the larger, fundamental open problem, which is how best to derive the motion macro-primitives, and then how best to employ them in the motion planning process. A key component is bound to be context extraction, the identifying of which features of the environment are relevant to a particular macro-primitive, and under what circumstances. It would also be desirable for the planner to learn such things from self-observation, noting which new strategies work, and then constructing appropriate new macro-primitives to embody them. Presumably such a framework would initially be bootstrapped using trajectory data obtained from other, lower-level planning algorithms, such as RRT. Further investigation is needed to determine the best way to do this.

It is also not clear what is the best way to represent the motion macro-primitives themselves. For example, they could be merely individual agent trajectories observed previously or, at the opposite end of the spectrum, they could be highly abstract plans or motion recipes, describing general control strategies with respect to important features in the environment. Or they could be an intermediate form between these two extremes, for example families of agent trajectories, parameterized by some meaningful attribute of the captured manoeuvre. Each of these alternatives has associated open problems. At the least abstract end of the spectrum, the snippets of unparameterized motion could be used to construct solutions by mere sequencing or tiling, but this would likely require some fitting and alteration, and it remains to be seen how best to do this effectively and efficiently. When the primitives are parameterized, presumably the families of trajectories can be obtained through simple clustering of prior motion data, but how does one identify a meaningful parameter for the family, and further, how should the planner choose a particular parameter value during planning? Furthermore, ideally the motion planner's framework should be sufficiently general that all of the above macro-primitive formulations can be employed simultaneously, leading to greater flexibility

and power. These would presumably be layered, arranged in a genealogical hierarchy where more abstract primitives are derived from lesser ones. How to do this is a particularly interesting open problem.

Another interesting possibility for such a planner would be automated "missing skill discovery". That is, how could the planner automatically identify weak areas in its "motion space" (i.e., in the language expressible using the current set of motion macro-primitives)? Since the proposed planner would be built up incrementally through experience, it is clear that at any point in time there will be environment features or scenarios that it will not be equipped to handle. Analogous analysis methods have been explore for motion graphs in [RP04]. It would thus be helpful if the planner could itself identify such potential weak areas offline, and then automatically devise and exercise appropriate training scenarios, so that it could acquire the missing "skills" on its own.

6.2 Exploiting human expertise

Thanks to their wealth of knowledge and expertise, humans still outstrip computers and machines in terms of raw ability to perform certain tasks in some domains (e.g., understanding and translating human languages). In such cases it is typically more beneficial to tap this expertise rather than to assault the problem with algorithms alone. To some degree this idea also parallels the use of lookup tables in stead of direct computation (e.g., sine tables), or even data-driven programming.

Such harnessing of human expertise has many precedents. For example, in computer animation the difficult problem of realistically animating human figures is today typically handled by pre-recording motion of a live human actor and then repurposing the resultant data to the goal virtual character, a technique called "motion capture". In recent years it has become popular to further structure this data, by constructing the clips of motion trajectories into "motion graphs" [LCR+02, AF02, KGP02]. This allows for arbitrarily long, continuous animation of the subject figure by simply traversing the motion graph along appropriate branches.

The above is an example of offline application of human expertise; a related approach could also be effective in motion planning. The following subsections look at other ways human expertise could be employed, online and offline, and it identifies some of the corresponding open problems.

6.2.1 Interactive motion planning

An interesting premise suggested in the early chapters of this thesis is the inclusion of a human agent in the motion planning process. There are a number of roles that the user could fulfil "on-line". One obvious user role would be to help the planner with difficult segments of the problem, in particular those which cause the planner to stall or break down. For example, the user could point out feasible sequences of milestones for PRM-style planners whenever narrow gaps are encountered. Of course such user involvement does not make sense in a general motion planner—after all, one

of the key motivations for computer planners is the need for automation—but there are specialized applications where such leveraging of human intelligence and experience can be helpful and desirable. How best to incorporate such user-planner interaction remains an open problem that requires further research; it is not obvious how best to canvass the user's opinion, how to streamline the interaction and ensure its effectiveness, and how best to apply user suggestions.

A related role for online human intervention is the selection of desired stylistic or topological traits of solutions. This has many strong parallels with extant research in the computer animation field, where the user directs automated methods for generating animation (e.g., [LCR+02]). Ultimately even many computer games cover similar ground, at least those where the user directs the motion (and thus less directly the animation) of a virtual avatar or vehicle, especially when these are constructed using motion trees or other precomputed motion libraries. Although these latter areas suggest some useful ideas, ultimately the motion planning process is different enough that novel ways need to be derived to handle its unique features and characteristics (e.g., discovery and exploration of trajectories being disjointed and non-continuous in time).

6.2.2 Harnessing motion capture data

There are many ways in which motion capture data—whether the subject's own motion or that of an object being manipulated by the human—can be used. For example, as noted earlier macro-primitives could be extracted from the motion data by scouring the sample trajectories for recurring patterns, since repeated patterns of motion generally hint at the presence of some larger, semantic content or connotation. Such motion segmentation has already been explored in other works, such as [BH00] or the motion graph literature in general [LCR+02, AF02, KGP02].

Another tack would be to use sample motion libraries for biasing the planning process, encouraging it to seek out trajectories that mimic those in the library, thus giving the solutions a more "natural" look. This idea was recently explored in [YKH04], where realistic object manipulation motions are obtained by first planning a (kinematic) path for the object, and then applying inverse kinematics on the manipulating agent to compute a suitable corresponding motion. Since inverse kinematics problems are typically under-constrained, the single "best" motion is selected through minimization of an objective function; in this work the objective function rewards configurations and motions which best mimic those in the sample set, thus leading to the desired bias. Unfortunately this approach has limited applicability since the biasing is confined to the inverse kinematics sub-component, which is absent in most planners. A more general approach would be to perform the biasing directly in the motion planning loop. Furthermore, it would be useful if such planners preserved some global properties of the sample motions. For example, in human locomotion it would be useful to preserve the agent's dynamic balance and the systematic left-right alteration of foot-to-ground contacts.

The most aggressive way to employ the motion data would be to use it as a filter. That is, the motion planning process could be constrained (rather than biased) to only explore trajectories that are present in the sample dataset, or in its immediate neighbourhood. By employing a significantly more constrained motion space the resultant planning task should be correspondingly simpler, due to the smaller search space. As long as the planning problem supports "natural" solutions–that is, composed of elements in the sample dataset of observed motions–the additional constraints will not be detrimental (i.e., prevent the finding of a solution).

In many respects this idea bears much resemblance to the viability filtering approach outlined in Chapter 4, in that both methods aspire to limit the motion planner to only useful or relevant regions of the search-space. Of the two, the latter, restricting motion to that which mirrors the provided samples, is generally the more aggressive alternative since it typically will filter more: naturally occurring motions are automatically viable[1], while the converse is often not true. Additionally, such datasets generally capture only a fraction of the agent's allowable range of motion, thus yielding an even more selective filter. Because of this, much greater care needs to be exercised to avoid the pitfalls associated with excessive filtering, particularly the inadvertent blockage of critical search-space bottlenecks.

Motion generation through the traversal of motion graphs is essentially an even more constrained variant of the above idea. In fact, this is the simplest way to implement such filtering: the generated motion is automatically constrained to the sample dataset (i.e., the motion graph). This approach has been explored in [IAF05], where motion planning solutions are constructed through the judicious traversal of the character's motion graph. An approximate form of reinforcement learning is used to derive a model of which branches of the graph are preferable in any particular situation, given the location of the next waypoint, the local obstacles and their motion, and the presence of enemies whose gaze must be avoided. Goal-seeking motions of the agent are generated by repeated application of the learned model during motion graph traversal, while a simple kinematic path planner provides the desired waypoints. Although some interesting results were obtained with this approach, there is much to explore further, and there are a number of other potential ways in which motion graphs could be exploited for motion planning, all of which warrants further investigation.

One drawback to employing motion graphs for motion planning is that they generally do not encode or model the surrounding environment, meaning that collision avoidance must be enforced manually, typically by "unrolling" and embedding the motion graph[2] into the local environment at the agent's current state, and then culling away branches which intersect obstacles. The above work [IAF05] is no different, since branch selection is based on the agent's state after a branch has been followed for 1 second; this thus amounts to a simple collision look-ahead. Since compliance with environment geometry is a key factor in the difficulty of motion planning problems, this significantly

[1]It is assumed that the sample motion libraries include only collision-free trajectories.

[2]Motion graphs generally encode their motions relative to the agent's current position and orientation.

limits the usefulness of motion graphs in MP. An interesting direction of research would thus be to find useful ways to annotate motion trajectories with local environment information, on a global scale, and then making effective use of this data during planning.

The only motion graph work to incorporate environment information is that of [LCL06]. This approach consists of first deriving a set of building blocks, called "motion patches", each a small motion graph corresponding to a prototypical fragment of environment geometry. When presented with a new environment, a full motion graph is assembled by subdividing the environment into fragments and then linking together the matching motion patches. In this way the motions in the resultant graph, as in the motion patches themselves, automatically avoid collision thanks to the direct modeling of the environment, making this approach compelling for use in motion planning. Interestingly, this combination would fall under the umbrella of motion planning with macro-primitives, since each motion patch can be thought of as a motion primitive describing all the various ways of (safely) navigating a particular environment fragment or "feature constellation". This would be a particularly interesting tangent to research further.

6.3 Optimal planning

Popular motion planners, such as RRT or PRM, return solutions which are generally very non-optimal, yet in many applications some degree of optimality is desirable. Currently there are very few planning choices when optimal trajectories are needed. One option is to solve the problem using a control policy approach, which typically employs dynamic programming to effectively compute optimal solutions from every possible starting state. Naturally this is very slow and expensive. Alternatively, if the agent is relatively simple, it is possible that an agent-specific optimal planner exists. For example, when the agent is a simple car that can move forwards and backwards, one can use Reeds-Shepp curves [RS90] to find optimal trajectories between any two car configurations. Unfortunately such agent-specific optimal planners typically ignore environment constraints, and thus are not directly applicable when obstacles are introduced. The ideal solution would thus be to enhance the current crop of motion planners to yield more optimal solutions.

It is worth noting that in many applications strict optimality is not necessary, and that near-optimal solutions are often sufficient. After all, strict optimality frequently cannot be met exactly in real-world applications due to actuator error and noise, and often other requirements place competing demands on the resultant motion (e.g., safety considerations).

The most direct and obvious way to address the optimality problem is to post-process the solutions returned by such planners using trajectory deformation methods. These often rely on repeated attempts to replace segments of the solution trajectory with shorter, more optimal alternatives. For example, for a simple, freely-moving kinematic agent it is common to iteratively try to replace fragments of the solution path with straight lines, starting with the whole solution itself

and progressing to smaller and smaller fragments. These methods are all relatively slow though due to their brute force nature.

A more intelligent and effective approach would likely rely on "understanding" the environment, being able to ascertain which parts of the solution are constrained by which features of the environment, and how best these may be optimized. Interestingly, motion planning with macro-primitives could potentially fill this need. If the motion planner used parameterized macro-primitives, the pursuit of optimality would simply translate to tuning of the parameter values for the primitives involved in the solution. For example, it is likely that the parameterized primitives for a car agent would include those that implement turns, with their parameter controlling the turn radius. Since tweaking one parameter is bound to cause the need to adjust the parameters of subsequent primitives, this presents interesting optimization problems where multiple parameters must be simultaneously tuned while preserving the continuity of the trajectory, and its connection to the goal. This has striking similarities to the problem of inverse kinematics in computer graphics.

A related interesting problem is that of turning current planners, such as RRT and PRM, into "anytime solvers", in the sense that, after an initial solution is found, the planners would be required to spend any additional time on improving it. The simplest naive approach would be to simply run the planner repeatedly, each time from scratch, and to return the most optimal solution found so far when requested. It is likely though that a more organic approach, one that builds upon work so far rather than ignoring it, could lead to more efficient results.

6.4 Robust execution of offline plans

Much of the "theoretical" component of motion planning research assumes the general "AI model" for solving real world problems, namely the modeling of the environment and agent, the offline computation of a solution, and the subsequent execution of the derived plan. This approach has often proven to be not very successful or robust in reality. Such open-loop execution is often plagued by the unavoidable noise and error in actuators, sensors, inexactness of models, the non-static nature of the world, and the general unpredictability of the future. This problem has already received some attention, but none of the proposed solutions are adequately robust or satisfactory, and thus the problem remains open.

Perhaps the first work to consider the imperfect execution of motion plans is that of the earliest approaches to motion planning with kinodynamic agents. These approaches relied on first computing a purely kinematic path, and then handed it to a dedicated agent controller which would attempt to implement the path within the constraints and peculiarities of the agent's kinodynamic laws of motion. In effect, it was the controller's job to compensate for any discrepancy which arose between the planned and achieved paths. Alas, many of these controllers were fairly simple. They often resulted in significant overshoot on turns and were generally clumsy in tracking the desired

path, which could potentially lead to collisions, even though the kinematic path itself was collision-free. One way to compensate for this was to reduce the speed with which the agent executed the path, so that the tracking errors were less severe, but for many applications this is an overly heavy-handed and objectionable workaround.

The most robust and globally optimal approach to the problem is the computation of control policies, which encode the optimal control action for each potential agent state, one that propels the agent toward the goal in an optimal fashion. The control policy thus effectively encapsulates the full, optimal solution to x_{goal} from any arbitrary agent state. That is, by iteratively applying at each time-step the control action specified by the control policy, the agent will have traced out the optimal trajectory upon reaching the goal. The benefit of using the policy is that if noise or error causes the agent to drift from this optimal trajectory, the control actions suggested by the policy at the new states will still safely lead the agent to the goal, and in a near-optimal manner. With single-trajectory solutions, such as those returned by RRT and PRM, further execution is impossible when the agent leaves the proposed solution trajectory; with a control policy, in contrast, it is essentially impossible to "leave the solution trajectory", since it encodes a solution from every feasible agent state. Although this is clearly a superior solution to the planning problem, control policies are very expensive to compute, must be recomputed whenever x_{goal} changes, and are not transferable across environments. It is worth pointing out that there has been some recent work [Sto06] which attempts to create anytime solvers for control policies, where rough, suboptimal control policies are quickly computed based on environment features, and then subsequently improved upon as time permits.

The most naive workaround for agent drift in single-trajectory planners is to simply recompute a fresh new solution when it is detected during execution that the agent has left the proposed trajectory. A more intelligent approach though is to partially reuse some of the prior planning effort; there has recently been some interest (e.g., [FKS06]) in re-using parts of prior RRT search-trees when replanning is needed.

A more closed-loop approach is proposed by the work on Partial Motion Planning (PMP) in [PF05]. Here the system discretizes time into slots, and within each such time slot it computes the partial motion plan for the next time-slot, while executing that computed in the previous slot. Although the feedback loop between planning and execution ensures that the agent is always "on target", this approach cannot, in general, be made optimal since globally optimal plans usually require a global view of the problem.

6.5 Duality between motion planning and control

There are also some strong parallels between motion planning and control problems. Consider a canonical control problem of a cart surmounted by an inverted pendulum, where the pendulum must

be kept upright and balanced through only direct manipulation of lateral acceleration of the cart, which is further constrained by two bounding walls (i.e., its position is limited to a modest interval). Given a sufficiently accurate forward-model of the world and system dynamics, so that cart response to control actions can be correctly predicted, the problem can be easily solved using typical motion planning methods. Conversely, motion planning problems can often be solved through traditional control theory methods, such as the use of a controller which encapsulates a pre-computed control policy for the agent (i.e., the controller applies the control action dictated by the control policy for the current agent state).

There are however some differences between the two fields. Specifically, control theory tends to focus on the dynamics of the system being controlled, and generally assumes there are no additional constraints (e.g., no obstacles in the environment). Motion planning, at least the kinematic subset which constitutes the large majority of the literature, tends to focus purely on the geometric constraints of the environment while assuming that the agent's dynamics are not relevant. Kinodynamic motion planning is, in a sense, the interesting superset of traditional motion planning and control theory, in that both the environmental constraints and system dynamics play equally important roles in determining solutions.

The parallels between these two fields suggest that there may be significant potential and opportunity for cross-pollination of ideas and the mutual exploitation of methods developed in the dual field. In fact, to some extent the proposed notion of motion planning with macro-primitives is an example of this. It seems to parallel a similar idea that has been explored in control and computer graphics. In [Fal02] the animation of a human figure is achieved using multiple controllers which alternate in manipulating the figure, with each controller responsible for performing a particular task or motion. Even in a case like this, where a potential parallel idea exists in the dual field, studying its methods may be helpful in identifying potential future problems that one may encounter (i.e., with motion macro-primitives), and may even suggest possible solutions to them.

Bibliography

[AF02] Okan Arikan and D. A. Forsyth. Interactive motion generation from examples. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 483–490, 2002.

[Ark98] Ronald C. Arkin. *Behavior-based Robotics*. MIT Press, 1998.

[ASP04] Jean-Pierre Aubin and Patrick Saint-Pierre. An introduction to viability theory and management of renewable resources. online, 2004. Available at `http://lastre.asso.fr/aubin/main.pdf`.

[Aub90] Jean-Pierre Aubin. A survey of viability theory. *SIAM Journal on Control and Optimization*, 28(4):749–788, 1990.

[Aub91] Jean-Pierre Aubin. *Viability Theory*. Systems & Control: Foundations & Applications. Birkhäuser, 1991.

[Aub02a] Jean-Pierre Aubin. Viability kernels and capture basins. online Lecture Notes, May 2002. Available at `http://lastre.asso.fr/aubin/wa00.pdf`.

[Aub02b] Jean-Pierre Aubin. An introduction to viability theory and management of renewable resources, coupling climate and economic dynamics. online slides, November 21, 2002. Available at `http://lastre.asso.fr/aubin/Gen\'eve-02-11-21-SL.pdf`.

[BATM93] B. Bessière, J.-M. Ahuactzin, El.-G. Tabli, and E. Mazer. The “Ariadne’s clew” algorithm: Global planning with local methods. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Yokohama, Japan, 1993.

[BF95] Jérôme Barraquand and Pierre Ferbach. Motion planning with uncertainty: The information space approach. In *Proc. of the IEEE Int. Conf. on Robotics & Automation*, volume 2, pages 1341–1348, 1995.

[BH00] Matthew Brand and Aaron Hertzmann. Style machines. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 183–192, 2000.

[BK07] Kostas E. Bekris and Lydia E. Kavraki. Greedy but safe replanning under kinodynamic constraints. In *IEEE Int. Conf. on Robotics & Automation*, 2007.

[BL91] Jérôme Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, 1991.

[BMMZ04] Olivier Bokanowski, Sophie Martin, Remi Munos, and Hasnaa Zidani. An anti-diffusive scheme for viability problems. Technical Report 5431, INRIA Rocquencourt, December 2004.

[Bra84] Valentino Braitenberg. *Vehicles. Experiments in Synthetic Psychology.* MIT Press, 1984.

[Bro86] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics & Automation*, 2(1):14–23, Mar 1986.

[Can88] John F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[CD06] Laetitia Chapel and Guillaume Deffuant. SVM viability controller active learning. *Kernel machines for reinforcement learning workshop*, 2006.

[CFL03] Peng Cheng, Emilio Frazzoli, and Steven M. LaValle. Exploiting group symmetries to improve precision in kinodynamic and nonholonomic planning. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003.

[Cha74] Chin-Liang Chang. Finding prototypes for nearest neighbor classifiers. *IEEE Transactions on Computers*, C-23(11):313–318, November 1974. reproduced in [Das91].

[Che05] Peng Cheng. *Sampling-Based Motion Planning with Differential Constraints.* PhD thesis, University of Illinois, 2005.

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence*, pages 331–337, 1991.

[CL] Peng Cheng and Steven M. LaValle. Resolution completeness for sampling-based motion planning with differential constraints. Submitted to International Journal of Robotics Research, under revision.

[CL93] John F. Canny and Ming C. Lin. An opportunistic global path planner. *Algorithmica*, 10(2-4):102–120, 1993.

[CL01a] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[CL01b] Peng Cheng and Steven M. LaValle. Reducing metric sensitivity in randomized trajectory design. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2001.

[CL02] Peng Cheng and Steven M. LaValle. Resolution complete rapidly-exploring random trees. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2002.

[CLH+05] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[Cul99] Joe Culberson. Sokoban is PSPACE-complete. In E. Lodi, L. Pagli, and N. Santoro, editors, *Proceedings in Informatics 4, Fun With Algorithms*, pages 65–76, 1999.

[Das91] Belur V. Dasarathy. *Nearest Neighbor(NN) norms: NN pattern classification techniques*. IEEE Computer Society Press, 1991.

[Dub57] L.E. Dubins. On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.

[DW91] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kauffman, 1991.

[DX90] Bruce Randall Donald and Patrick G. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. In *Symposium on Computational Geometry*, pages 290–300, 1990.

[DXCR93] Bruce Randall Donald, Patrick G. Xavier, John F. Canny, and John H. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, 1993.

[DZ99] Dorit Dor and Uri Zwick. SOKOBAN and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[Eld01] Craig Eldershaw. *Heuristic algorithms for motion planning*. PhD thesis, The University of Oxford, 2001.

[FA04] Thierry Fraichard and Hajime Asama. Inevitable collision states: a step towards safer robots? *Advanced Robotics*, 18(10):1001–1024, 2004.

[Fal02] Petros Faloutsos. *Composable Controllers for Physics-Based Character Animation*. PhD thesis, University of Toronto, 2002.

[FDF99] E. Frazzoli, M. 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, 1999.

[FFD00] E. Feron, E. Frazzoli, and M. Dahleh. Real-time motion planning for agile autonomous vehicles. *In AIAA Conf. on Guidance, Navigation and Control*, 2000.

[FKS06] David Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with rrts. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 1243–1248, May 2006.

[FLS05] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A guide to heuristic-based path planning. In *Proc. of ICAPS Workshop on Planning under Uncertainty for Autonomous Systems*, 2005.

[Fra07] Thierry Fraichard. A short paper about motion safety. In *Proc. of the IEEE Int. Conf. on Robotics & Automation*, 2007.

[HA92a] Yong K. Hwang and Narendra Ahuja. Gross motion planning—a survey. *ACM Computing Surveys*, 24(3):219–291, 1992.

[HA92b] Yong K. Hwang and Narendra Ahuja. Gross motion planning—a survey. *ACM Computing Surveys (CSUR)*, 24(3):219–291, September 1992.

[HKLR00] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. In *Workshop on the Algorithmic Foundations of Robotics*, 2000.

[HKLR02] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3), 2002.

[Hog85] Neville Hogan. Impedance control: An approach to manipulation. Parts I-III. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control*, 107:1–24, 1985.

[IAF05] Leslie K. M. Ikemoto, Okan Arikan, and David A. Forsyth. Learning to move autonomously in a hostile world. Technical Report UCB/CSD-05-1395, EECS Department, University of California, Berkeley, June 2005.

[IMT03] Pekka Isto, Martti Mäntylä, and Juha Tuominen. On addressing the run-cost variance in randomized motion planners. *Proc. IEEE Int. Conf. on Robotics & Automation*, 2003.

[JC89] P. Jacobs and J. Canny. Planning smooth paths for mobile robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 1989.

[JJKKN+02] Jr. Jame J. Kuffner, Satoshi Kagami, Koichi Nishiwaki, Masayuki Inaba, and Hirochika Inoue. Dynamically-stable motion planning for humanoid robots. *Autonomous Robots*, 12(1):105–118, January 2002.

[KBM04] Marcelo Kallmann, Robert Bargmann, and Maja Matarić. Planning the sequencing of movement primitives. In *Proceeding of the International Conference on Simulation of Adaptive Behavior*, July 2004.

[KGP02] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 473–482, 2002.

[Kha86] O. Khatib. Real-time obstacle avoidance for manipulators and, mobile robots. *International Journal of Robotic Research*, 5(1):90–98, 1986.

[KHLR00] Robert Kindel, David Hsu, Jean-Claude Latombe, and Stephen Rock. Kinodynamic motion planning amidst moving obstacles. In *IEEE Int. Conf. on Robotics and Automation*, 2000.

[KL00] James J. Kuffner, Jr. and Steven M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *IEEE Int. Conf. on Robotics & Automation*, 2000.

[KNK+01] James Kuffner, Koichi Nishiwaki, Satoshi Kagami, Masayuki Inaba, and Hirochika Inoue. Motion planning for humanoid robots under obstacle and dynamic balance constraints. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, 2001.

[KvdP04] Maciej Kalisiak and Michiel van de Panne. Approximate safety enforcement using computed viability envelopes. In *IEEE Int. Conf. on Robotics & Automation*, volume 5, pages 4289–4294, 2004.

[KvdP06] Maciej Kalisiak and Michiel van de Panne. RRT-Blossom: RRT with a local flood-fill behavior. In *IEEE Int. Conf. on Robotics & Automation*, 2006.

[KvdP07] Maciej Kalisiak and Michiel van de Panne. Faster motion planning using learned local viability models. In *Submitted to IEEE Int. Conf. on Robotics & Automation*, 2007.

[Lat91] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Press, 1991.

[LaV98] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. technical report TR 98-11, Computer Science Dept., Iowa State University, 1998.

[LaV06] Steven M. LaValle. *Planning Algorithms.* Cambridge, 2006.

[LCL06] Kang Hoon Lee, Myung Geol Choi, and Jehee Lee. Motion patches: Building blocks for virtual environments annotated with motion data. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2006.

[LCR+02] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 491–500, 2002.

[LDW91] J.J. Leonard and H.F. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *Proc. of the IEEE Int. Workshop on Intelligent Robots and Systems*, pages 1442–1447, 1991.

[LFV04] Florent Lamiraux, Etienne Ferré, and Erwan Vallée. Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2004.

[Lin04] Frank Lingelbach. Path planning using probabilistic cell decomposition. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2004.

[LK99] Steven M. LaValle and James J. Kuffner, Jr. Randomized kinodynamic planning. In *IEEE Int. Conf. on Robotics & Automation*, 1999.

[LK00] Steven M. LaValle and James J. Kuffner, Jr. Rapidly-exploring random trees: Progress and prospects. In *Workshop on Algorithmic Foundations of Robotics*, 2000.

[LK05a] Andrew M. Ladd and Lydia E. Kavraki. Fast tree-based exploration of state space for robots with dynamics. *Algorithmic Foundations of Robotics VI*, pages 297–312, 2005.

[LK05b] Andrew M. Ladd and Lydia E. Kavraki. Motion planning in the presence of drift, underactuation and discrete sytem changes. *Robotics: Science and Systems I*, pages 233–241, 2005.

[LL03] Stephen R. Lindemann and Steven M. LaValle. Current issues in sampling-based motion planning. In *Proc. of the Int. Symposium on Robotics Research*, 2003.

[LL06] Stephen R. Lindemann and Steven M. LaValle. A multiresolution approach for motion planning under differential constraints. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 139–144, 2006.

[MIB00] F. A. Mussa-Ivaldi and E. Bizzi. Motor learning through the combination of primitives. *Philosophical Transactions: Biological Sciences*, 355(1404):1755–1769, 2000.

[MIS04] Ferdinando A. Mussa-Ivaldi and Sara A. Solla. Neural primitives for motion control. *IEEE Journal of Oceanic Engineering*, 29(3):640–650, July 2004.

[Mit02] Ian Mitchell. *Application of Level Set Methods to Control and Reachability Problems in Continuous and Hybrid Systems.* PhD thesis, Stanford, 2002.

[Nil69] N. J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *Proc. of the 1st Int. Joint Conf. on Artificial Intelligence*, pages 509–520, 1969.

[OS95] Mark H. Overmars and Petr Svestka. A probabilistic learning approach to motion planning. In *Workshop on the Algorithmic Foundations of Robotics*, 1995.

[Par06] Rishikesh Parthasarathi. Characterization of the inevitable collision states for a car-like vehicle. Master's thesis, Inst. Nat. Polytechnique de Grenoble, June 2006.

[P.E68] P.E.Hart. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, IT-14(3):515–516, May 1968.

[PF05] Stéphane Petti and Thierry Fraichard. Safe motion planning in dynamic environments. In *Proc. of the IEEE-RSJ Int. Conf. on Intelligent Robots and Systems*, August 2005.

[PF07] Rishikesh Parthasarathi and Thierry Fraichard. An inevitable collision state-checker for a car-like vehicle. In *Proc. of the IEEE Int. Conf. on Robotics & Automation*, 2007.

[PKV07] Erion Plaku, Lydia E. Kavraki, and Moshe Y. Vardi. Discrete search leading continuous exploration for kinodynamic motion planning. In *Robotics: Science & Systems*, 2007.

[Rei79] J. H. Reif. Complexity of the mover's problem and generalizations. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 421–427, 1979.

[Ros03] Eric J. Rossetter. *A potential field framework for active vehicle lanekeeping assistance.* PhD thesis, Stanford University, 2003.

[RP04] Paul S. A. Reitsma and Nancy S. Pollard. Evaluating motion graphs for character navigation. In *Proc. of ACM SIGGRAPH / Eurographics 2004 Symposium on Computer Animation*, 2004.

[RS90] J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific Journal of Mathematics*, 145(2):367–393, 1990.

[RW97] J. Reif and H. Wang. *Non-uniform discretization approximations for kinodynamic motion planning*, pages 72–112. A K Peters, 1997.

[SC86] Randall C. Smith and Peter Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5(4):56–68, 1986.

[SL01] Gildardo Sánchez and Jean-Claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Proc. of IEEE Int. Symposium on Robotics Research*, 2001.

[SML96] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1–2):17–29, 1996.

[SP94] Patrick Saint-Pierre. Approximation of the viability kernel. *Applied Mathematics & Optimization*, 1994.

[Sto06] Martin Stolle. Policies based on trajectory libraries. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2006.

[Str04] Morten Strandberg. Augmenting RRT-planners with local trees. In *IEEE Int. Conf. on Robotics & Automation*, pages 3258–3262, 2004.

[Tan03] Jun Tani. Learning to generate articulated behavior through the bottom-up and the top-down interaction processes. *Neural Networks*, 16(1):11–23, January 2003.

[vdP94] Michiel van de Panne. *Control Techniques for Physically-Based Animation.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994. Available at `http://www.cs.ubc.ca/~van/papers/phd.ps.gz`.

[VE03] Ardalan Vahidi and Azim Eskandarian. Research advances in intelligent collision avoidance and adaptive cruise control. *IEEE Transact. on Intelligent Transportations Systems*, 4(3):143–153, September 2003.

[VF04] Dizan Vasquez and Thierry Fraichard. Motion prediction for moving objects: a statistical approach. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 3931–3936, New Orleans, LA (US), April 2004.

[YKH04] Katsu Yamane, James Kuffner, and Jessica K. Hodgins. Synthesizing animations of human manipulation tasks. In *Proc. of the Int. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2004.