Kinodynamic Motion Planning with Viability Models

by

Maciej Kalisiak
<mac@dgp.toronto.edu>

A Research & Thesis Proposal
Graduate Department of Computer Science
University of Toronto

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 balance, collision avoidance, or prevention of other undesirable motions.

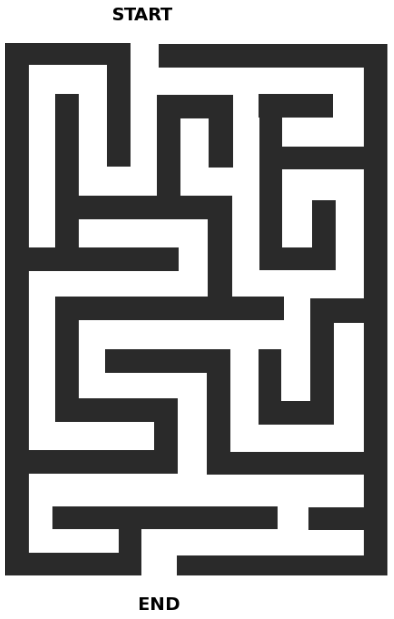


Figure 1.1: A maze: one of the simplest motion planning problems.

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 well-known toy example in motion planning which, in general, deals with the problem of how to maneuver a piano through narrow corridors, doorways and stairwells, all the while avoiding walls and furniture, so that the piano is delivered to its final resting place 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 pose additional constraints, such as the avoidance of manipulator self-intersection, and of inter-manipulator collisions (and this is all on top of the base problem of the collision-free motion of the object being manipulated). The general problem of planning, sequencing and coordinating movement of multiple agents also falls within the purview of this field.

With such wide scope it is not surprising that motion planning has many diverse and far-reaching applications in the real world, and thus garners considerable research attention. Aside from the most obvious applications to autonomous robot locomotion, 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 areas, such as protein folding and drug design.

1.1 Kinodynamic motion planning

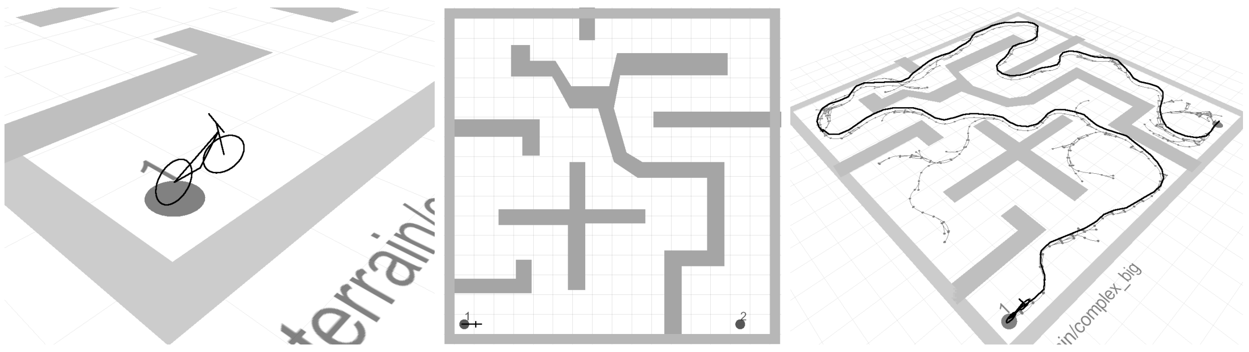
A difficult subclass of motion planning problems is that of *kinodynamic* systems. Whereas most typical motion planning problems are *kinematic*, in that they deal only with the agent's *configuration*—its position, orientation, and the internal arrangement of agent parts—kinodynamic motion planning, 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, $x = (q, q')$. What makes kinodynamic planning difficult, aside from the larger search space (twice the dimensions relative to kinematics-only planning with same agent), 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 towards its goal. These difficult regions are generally the result of the "cross product" of the environment's geometry and the agent's laws of motion.

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. In general, any system in which inertia or dynamic balance plays a significant role is kinodynamic, although the absence of these traits does not necessarily indicate to the contrary.

1.2 A sample kinodynamic problem

Throughout this proposal 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 then, 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., $\{-\pi/4, +\pi/4\}$). What therefore makes this system difficult to work with 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 proposal: a fixed-velocity bike. "1" and "2" indicate the locations of x_{init} and x_{goal}, respectively; at these points both states dictate that the bike be facing right, be upright, and have zero lateral 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 (i.e., the bike's steering angle). In the case of a continuous-time system, the function computes the resultant state vector derivative:

$$x' = f(x, u)$$

In the case of a discrete-time system, it computes the subsequent state of the system:

$$x_{k+1} = f(x_k, u_k)$$

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 mandated constraints. This trajectory is the planner's output, although for actuated systems the solution additionally 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 sequence can then be fed directly to the physical agent to achieve the discovered solution trajectory.

1.3 Research goal and themes

The main weakness of current kinodynamic motion planners is the effort they waste on exploring regions of the search space that are predestined to be fruitless, the dead-ends alluded to above. Thus the primary goal of the research presented here is the development of a smarter, more efficient kinodynamic planner, one that endeavours to extract maximum benefit from its work by avoiding such waste and other redundant behaviour.

More generally, the primary theme throughout this proposal is "smarter motion planning". This essentially involves pre-computing or learning aspects of potential solutions, on a global or local scale, leading to predictive models of where the feasible regions lie, or conversely, which regions are predestined to failure. These models are then used to constrain planner exploration. On an intuitive level, this approach resembles the use of cellular automata for solving mazes (see Figure 1.3), where on each iteration "dead-end" cells, namely ones with 3 surrounding walls, are "filled in". These same models, which capture the agent's viability[1], can also be used for "safety enforcement" during user control of the agent, which is a secondary theme in this proposal. This consists of

[1]Discussed later; briefly, a nonviable state is one in which the agent has passed the "point of no return", where failure or collision is no longer avoidable.

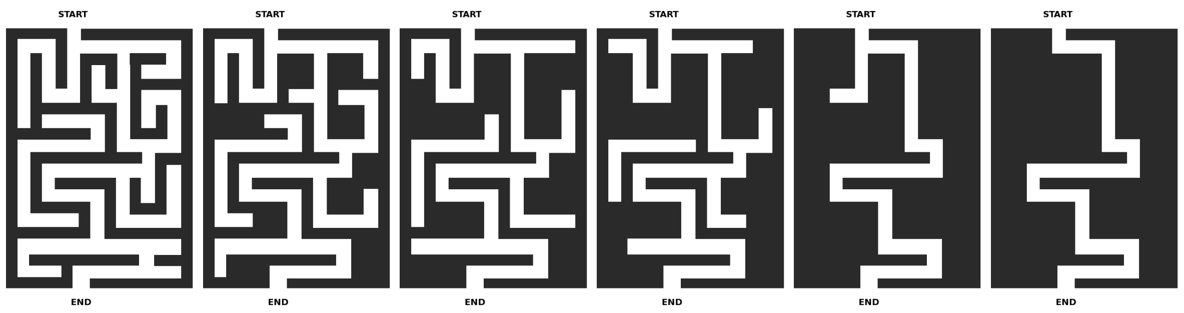


Figure 1.3: The solving of a maze using cellular automata. The proposed viability filtering of planner's search space strongly resembles the effect achieved by the cellular automata, in that only the (locally) feasible paths remain.

giving the user free reign over the agent, except when this threatens to leave the model-permitted space of operation, in which case the user-selected control action is automatically superseded by one that does not breach the model envelope.

The areas of motion planning and control share a significant relationship. In essence, both problems deal with searching, whether it be for a successful motion or a suitable control policy. But while motion planning most often addresses kinematic agents under geometric constraints (i.e., obstacles), control problems usually deal with dynamic systems in otherwise unfettered environments. Kinodynamic motion planning is thus, in a sense, a union of the two, in that it deals with dynamic agents under geometric constraints. Despite the prevalence of such systems, there is remarkably little literature dealing with kinodynamic motion planning, which, together with the lack of robustness of current methods, makes it an attractive area of research.

1.4 Proposal structure

The rest of this proposal is organized as follows. The next chapter presents previous work in the areas of motion planning and viability. Chapter 3 proposes a variant of a popular motion planner; this variant addresses the planner's poor performance in highly constrained environments. Chapter 4 builds on this by equipping the planner with learned viability models which serve to curtail the planner's wasteful exploration of areas that are predestined to fail. Chapter 5 finally looks at how these same learned viability models could be used for safety enforcement in user-controlled systems.

The key chapters are strongly patterned on corresponding papers. In particular, Chapters 3, 4, and 5 were derived from [KvdP06], [KvdP07], and [KvdP04], respectively.

1.5 Proposed thesis contributions

The primary contribution of the thesis would be the extension of the RRT algorithm, the currently best available general-purpose kinodynamic motion planner, yielding much improved operation with more difficult agents. In particular, the thesis would address RRT's poor performance in highly constrained environments (resulting in "RRT-Blossom"; Chapter 3), as well as with highly unstable or failure-prone agents (resulting in "RRT-Blossom with Viability Filtering"; Chapter 4). Figures 1.4 and 1.5 illustrate the achievable improvements with these extensions.

A related contribution would be the application of the learned viability models for safety enforcement in user-controlled systems, as outlined in Chapter 5. A key component of this is the proposed method for compensating for inaccuracies and error in the learned model.

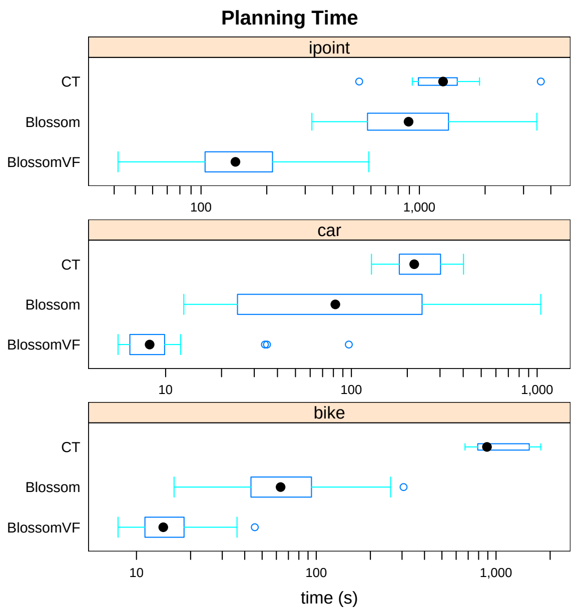


Figure 1.4: Representative improvements in planning time; boxplots show performance under various agents for RRT-CT (best current), RRT-Blossom (Chapter 3), and RRT-Blossom with Viability Filtering (Chapter 4).

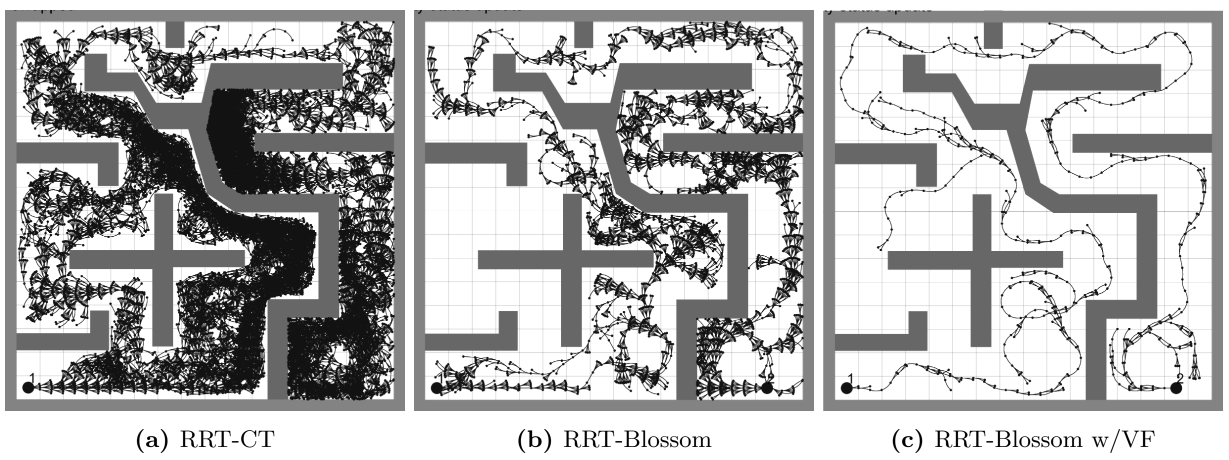


Figure 1.5: Visual comparison of "exploration effort" for the bike for best current RRT algorithm (RRT-CT), and our proposed variants. The viability-filtering planner on right, in particular, achieves a solution with noticeably less effort.

Chapter 2

Previous Work

2.1 Motion planning

As suggested earlier, motion planners can be categorized as either kinematic or kinodynamic. The former operate solely upon the agent's configuration q, whereas the latter operate upon the agent's configuration *and* its time derivative; this combined vector is called the agent's *state*, and denoted by x, where $x = (q, q')$. Thorough treatment of the general motion planning problem can be found in [LaV06] and [Lat91], while [DW91] adopts a unified viewpoint between planning and control.

Kinematic motion planning

Kinematic planners, by virtue of addressing the simpler problem, were first to be studied, yielding a vastly more extensive body of research today. One important class of such planners employs *potential fields* or *navigation functions*, scalar functions erected over the search space that "lead" the agent towards the goal. These potential fields are usually the sum of a number of simpler fields; typically, one field attracts the agent towards the goal, while others serve as repulsive forces that "push" the agent away from obstacles. Path planning then consists of performing gradient descent down the aggregate manifold. Figure 2.1 illustrates the key components of this approach. Unless stated otherwise, planning examples and figures in this proposal, including Figure 2.1, illustrate planner operation using purely geometric paths (i.e., the agent is a kinematic point capable of motion in arbitrary direction).

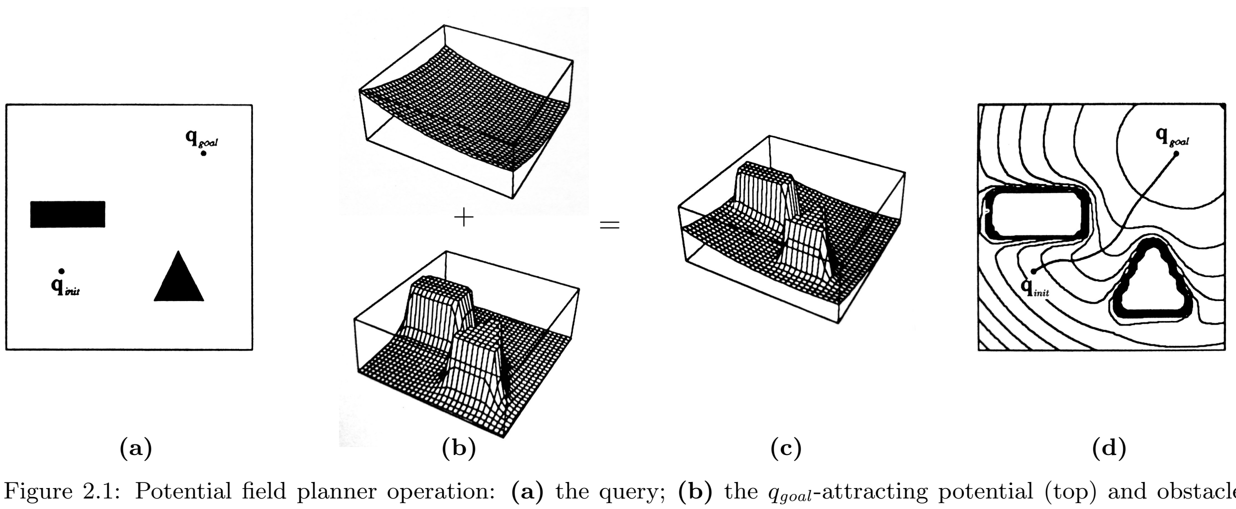


Figure 2.1: Potential field planner operation: **(a)** the query; **(b)** the q_{goal}-attracting potential (top) and obstacle-repulsing potential (bottom); **(c)** sum of potentials; **(d)** isoline plot of potential field, and gradient descent path. *Adapted from [Lat91], with permission.*

A key weakness of said planners, typical of gradient descent methods, is that they easily get stuck in local minima. One of the more successful of these algorithms, the Randomized Path Planner (or RPP)[BL91], 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 are free of local minima when employed with a kinematic point-mass agent without dynamics. Nonetheless, local minima can develop during gradient descent once agent geometry is factored in; typically, in such cases, the agent gets wedged in the surrounding obstacles, and in order to proceed in the direction suggested by the navigation function the agent must be backed up and re-oriented. This "nudging" is achieved using random-walks: when the planner detects that a local minimum has been reached, the agent undergoes a series of random perturbations. It should be noted that there is no guarantee that any particular random-walk will escape the local minimum, but the duration of the random-walk is chosen such that there is a good probability of this happening, based on the size and discretization of the workspace. If the random-walk fails 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 2.2.

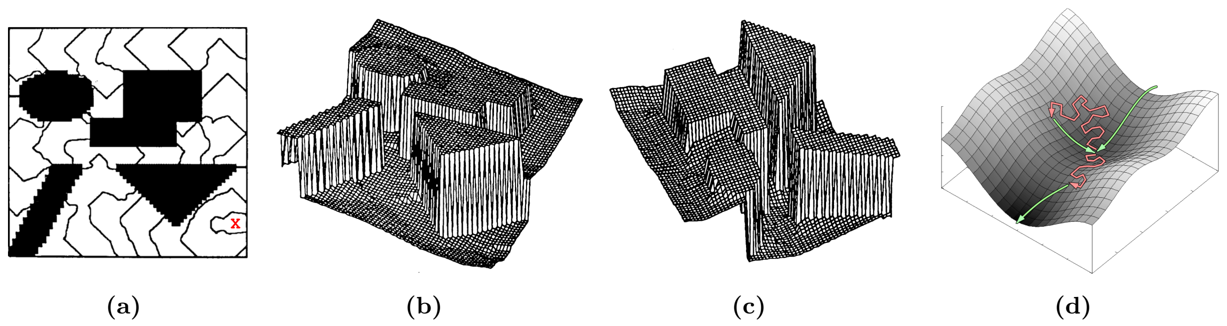


Figure 2.2: Randomized Path Planner (RPP) operation: the first step is computation of a "navigation function" using a combination of medial axis and level-set methods; (a) shows the function using isolines (red "X" in lower right marks the goal), while (b) and (c) render it as a height-field, seen from 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], with permission.

An important feature of RPP which allows it to produce good results for some problems is the use of sampling for low-cost approximation of the gradient of the potential field at a particular point. In highly dimensional spaces computing the gradient exactly is prohibitively expensive, rendering such potential field methods unfeasible otherwise. To some degree, this sampling trait allows the planners to break the curse of dimensionality of the search space, and it remains a key element of state-of-the-art planners of today.

One popular planner that exploits this trait is the Probabilistic Roadmap (PRM)[OS95]. This approach requires a relatively expensive pre-computation step, hence is geared more for answering multiple queries for the same static environment. As such, it will receive a much shorter treatment in this proposal. The pre-computation consists of first populating the workspace with *milestones*—a set of collision-free states of the agent, chosen uniformly from the free-space—and then constructing its "roadmap" by connecting each milestone with a number of its closest neighbours if said connection is collision-free. For simple, freely moving agents the milestones are simply connected with straight lines, but in the general case the connections are attempted using an externally provided "local planner". Given such a roadmap, motion planning consists of first connecting q_{init} and q_{goal} to the roadmap, and then finding the shortest path in the resultant graph. The solution trajectory is obtained by concatenating the trajectories of all the edges encountered on the graph walk from q_{init} to q_{goal}. Figure 2.3 illustrates the algorithm.

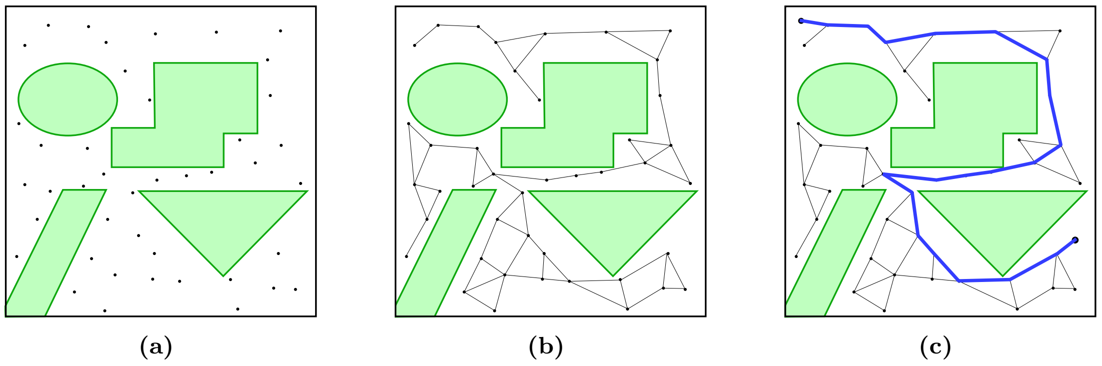


Figure 2.3: The Probabilistic Roadmap (PRM); (a) the environment is uniformly sampled with a set of random *milestones*; (b) each milestone is connected, where possible, with a number of nearest neighbours, yielding a *roadmap*; (c) solution is found by connecting q_{init} and q_{goal} to the roadmap, and then finding the shortest path through the graph.

The key issues with PRM are: 1) the lack of robust local motion planners for most dynamical systems; and 2) the "narrow passage problem". The latter refers to the difficulty of properly capturing the topology of narrow passages with the roadmap, which is a side-effect of the stochastic sampling of milestones. Even with a straight-line local planner one must put down a lot of (random) milestones before a subset of them line up in a way that the narrow passage can be traversed with a piecewise-linear curve, thus allowing the "discovery" of the passage.

The other predominant sampling-based approach to motion planning is that of the Rapidly-exploring Random Trees (or RRTs)[LaV98, LK00]. Rather than working globally like PRM, RRT explores the free-space in a more incremental fashion, building trees with a strong bias for diffusion. Since the planners in the rest of this proposal all build upon RRT, we provide a more extensive discussion of this algorithm and its variants.

The defining characteristic of RRTs, the reason for the "rapidly-exploring" moniker, is the tree bias for always growing towards unexplored space. RRT's operation is best illustrated using the simplest variant, which grows a single tree rooted at the initial configuration, q_{init}, until a branch reaches the goal configuration, q_{goal}. At each iteration the planner first picks a random, collision-free agent configuration, q_{tgt}, and identifies the tree node, q_{near}, that is nearest to it.[1] The planner then attempts to extend the tree from q_{near} towards q_{tgt}, using some small time-step. For simple agents with no constraints on q', the potential tree edge extends directly towards q_{tgt}. For actuated agents, the direction is determined indirectly: the agent's range of possible control actions are first discretized, and from the resulting set the planner then chooses and instantiates the control action that advances the closest to q_{tgt} in a single time-step (without incurring a collision). These steps are illustrated in Figure 2.4, while Algorithm 1 describes the planner with pseudocode. Figure 2.5 illustrates the structural difference between a tree where edges are grown in random directions and an RRT where growth is biased towards unexplored space.

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

RRT-Connect[KL00], another common extension, uses multiple time-steps when creating a new edge, rather than just a single one. That is, once q_{near} (and the best control action, in the actuated

[1]Note: we will often use the symbol q to refer to both, the agent configuration q as well as the tree node corresponding to that configuration. This applies to all related symbols, such as q_{init} and q_{near}. Later, when agent state replaces its configuration, an analogous situation will apply to the symbol x.

Algorithm 1 single-tree RRT for an actuated agent

```
1: function QUERY($q_{init}$,$q_{goal}$)
2:     $T \leftarrow$ tree($q_{init}$)
3:     while time_elapsed() < MAX_TIME do
4:         $q_{tgt} \leftarrow$ random_q()
5:         $q_{new} \leftarrow$ grow_tree($T, q_{tgt}$)
6:         if $q_{new} \wedge \rho(q_{new}, q_{goal}) < \epsilon$ then
7:             return extract_soln($q_{new}$)
8:     return *failed*
9: function GROW_TREE($T$,$q_{tgt}$)
10:    $q_{near} \leftarrow$ nearest_neighbor($T, q_{tgt}$)
11:    $u_{best} \leftarrow$ pick_ctrl($q_{near}$, $q_{tgt}$)
12:    if $u_{best}$ then
13:        $T \leftarrow T +$ new_edge($q_{near}$, $u_{best}$)
14:    return $q_{new}$
15: function PICK_CTRL($q$,$q_{tgt}$)
16:    $d_{min}, u_{best} \leftarrow \rho(q, q_{tgt}), \varnothing$
17:    for $u \in \mathcal{U}$ do
18:        $q_{new} \leftarrow$ sim($q, u$)
19:        if failure($q, u, q_{new}$) then
20:            next $u$
21:        $d \leftarrow \rho(q_{new}, q_{tgt})$
22:        if $d < d_{min}$ then
23:            $d_{min}, u_{best} \leftarrow d, u$
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)

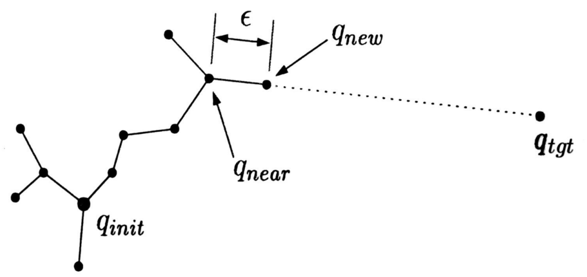


Figure 2.4: Steps of an RRT iteration: 1) randomly choose a target configuration, q_{tgt}; 2) find nearest tree node, q_{near}; 3) attempt to grow the tree from q_{near} towards q_{tgt}, for a single time-step ϵ, yielding new node at q_{new}.
Adapted from [KL00], with permission.

agent case) has been found, the new tree edge is extended a multiple of time-steps, until either it reaches the target, hits an obstacle, or starts receding from the target. This serves to span large open spaces quickly, and generally works for well for agents that move in a linear fashion.

Despite these improvements, the single-tree RRT still achieves relatively mediocre performance, and hence is rarely used. Much better results can be obtained by employing two trees[LK00]: the first at q_{init} as before, and a second one at q_{goal}, which is grown "backwards in time" (see Figure 2.6). As the two trees grow, it is easy to see how they provide a larger and larger target area for each other to connect to; the single tree, on the other hand, is always seeking a singular point (i.e., q_{goal}), a much more difficult task. Algorithm 2 outlines the operation of the dual-tree approach. In brief, on any given iteration the first tree is treated in the same manner as before: choose random q_{tgt}, find q_{near}, grow tree 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 with $q_{tgt} = q_{new_A}$; that is, the first tree is grown towards a randomly chosen q_{tgt}, yielding a new tree node q_{new_A}, while the second tree is grown towards q_{new_A}. After each iteration the trees switch roles.

The RRT-Connect technique can be applied equally well to the dual-tree RRT case, either to only one of the trees, or to both. This results in four possible variants, commonly named RRTConCon, RRTConExt, RRTExtCon, RRTExtExt. Here "Con", short for "connect", refers to the RRT-Connect method of growing edges, while "Ext", short for "extend", refers to the simpler, single time-step method; in each of the variant names, the first identifier specifies the growth method used in the q_{tgt}-seeking stage, while the second identifier specifies that of the tree-seeking, second stage (see Figure 3.3). RRTConCon is popular for freely-moving agents, but in some cases

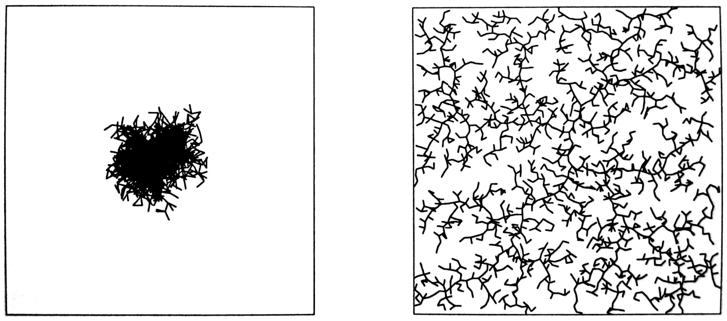


Figure 2.5: On left: a tree with edges grown in random directions; on right: RRT, where each edge is biased to grow toward unexplored space. Both trees have exactly 2000 nodes.
Reproduced from [LK99], with permission.

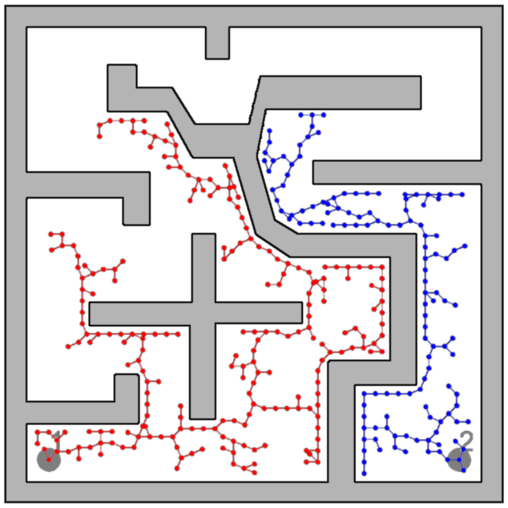


Figure 2.6: RRT commonly employs two trees, one at x_{init}, and the other at x_{goal}.

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

```
1: function QUERY($q_{init}$,$q_{goal}$)
2:     $T_a, T_b \leftarrow$ tree($q_{init}$), tree($q_{goal}$)
3:     while time_elapsed() $<$ MAX_TIME do
4:         $q_{tgt} \leftarrow$ random_q()
5:         $q_{new_A} \leftarrow$ grow_tree($T_a, q_{tgt}$)
6:         if $q_{new_A}$ then
7:             $q_{new_B} \leftarrow$ grow_tree($T_b, q_{new_A}$)
8:             if $q_{new_B}$ then
9:                 if $\rho(q_{new_A}, q_{new_B}) < \epsilon$ then
10:                    return extract_soln($q_{new_A}$, $q_{new_B}$)
11:        $T_b, T_a \leftarrow T_a, T_b$
12:    return *failed*
```

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

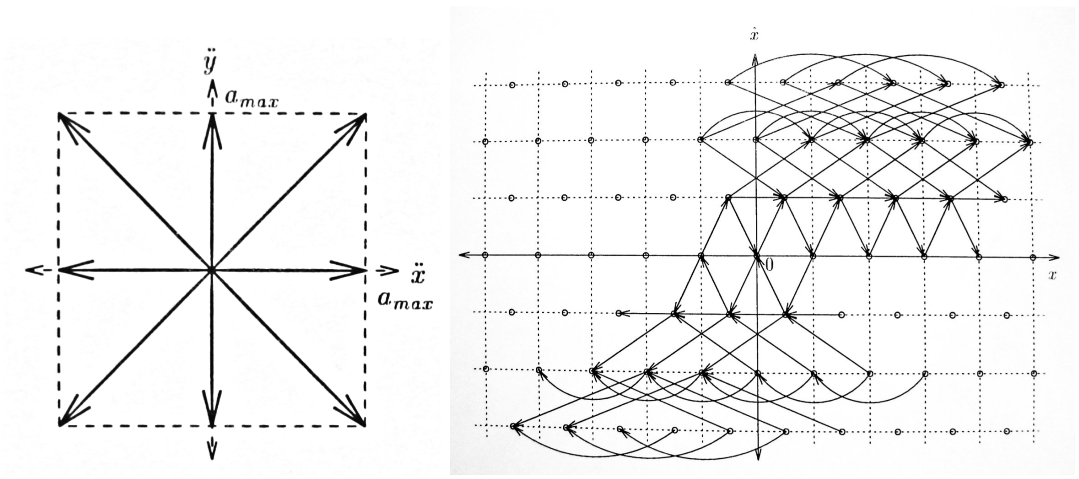


Figure 2.7: Seminal kinodynamic planner of [DXCR93]. The point agent is limited to a small set of allowed acceleration vectors (left) which are then applied for small durations τ. The use of such (a_{max}, τ)-*bang* motions results in a grid-like discretization of agent's state-space; right: cross-section of agent's state-space at $y = 0$.
Reproduced from [DXCR93], with permission.

RRTExtCon is preferred, as it emphasizes attempting to connect the trees over exploration.

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 target configuration towards which extant trees cannot be grown, the wayward 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. Whenever two branches meet, their trees are merged, while a connection between the primal, endpoint trees (i.e., 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 is advantageous because it populates with nodes the ingress points to this difficult area, thus making it easier to connect to that component of the free-space.

Kinodynamic motion planning

One of the earliest works to address the kinodynamic problem is [DXCR93], which concerns itself with finding near time-optimal trajectories for a kinodynamic point mass with velocity and acceleration bounds. The crux of the method is the discretization of the state-space using a regular grid (see Figure 2.7) and the subsequent use of the usual "shortest path in graph" methods to find the near optimal trajectory. Such discretization is made possible by limiting each component of acceleration of the agent to $\{-a_{max}, 0, +a_{max}\}$. 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 (i.e., 3 or less).

In the simplest general case, refitting a planner for kinodynamic operation consists simply of replacing references to agent's configuration with ones to the agent's state (i.e., replace q with x, where $x = (q, q')$). For example, PRM could be converted in this way, assuming an appropriate kinodynamic local planner is also made available. Unfortunately for kinodynamic systems such local planning is often just as difficult a problem as the global one, hence the original PRM algorithm is rarely used in this domain.

KDP[HKLR00] is a PRM-inspired planner specifically designed for kinodynamic systems. It bears a strong resemblance to RRT in that it is also a tree-oriented approach, but differs in its exploration biasing strategy: rather than using a randomly chosen target state to "attract" new growth, the next node to be expanded is chosen probabilistically, with likelihood being inversely proportional to the number of nearby neighbours that a node has (i.e., sparsely populated areas are favoured).

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

RRTs, on the other hand, can be easily refit for kinodynamic systems[LK99] without any loss of generality. One issue that requires careful attention is the growth of the second tree, from x_{goal}: since this tree is "grown backwards", its trajectories must be computed by integrating the agent's laws of motion using a negative time step, and this distinction needs to be observed throughout the planning process.

Although the kinodynamic RRT algorithm works well for simple systems, it tends to do poorly with more complex agents. One of the key problems is RRT's sensitivity to the distance metric used to bias the exploration process: if the metric does not accurately reflect the true cost-to-go, RRT's performance is severely affected. [CL01b] proposes to mitigate this problem by introducing two key changes to the algorithm: 1) that the algorithm keep track and subsequently disqualify from consideration any attempted tree edges which result in collisions; and 2) that the *collision tendency* of each node in the tree be tracked. We refer to this approach as RRT with Collision Tendency (RRT-CT). Algorithm 3 outlines the planner in more detail. The value tracked for collison tendency is only a lower-bound approximation: whenever an attempted edge incurs a collision, each ancestor node's collision tendency is increased in proportion to the edge's relative importance to that ancestor; thus an ancestor node's collision tendency asymptotically approaches the ultimate true value as the exploration of the subtree at that node completes.

2.2 Viability

Our proposed solutions to the above problems with kinodynamic planning rely on the concept of viability, hence it is worthwhile to review here the basics of viability theory. We present only a rudimentary introduction here; more comprehensive yet still concise introductions to this topic can be found in [ASP04, Aub02b, Aub02a], while a more thorough treatment is provided in [Aub91].

In plain terms, viability describes whether the dynamical system's operation is sustainable from a given starting point, i.e., whether the system can be kept from failure. More precisely, a *viable* state is guaranteed to have at least one sequence of control actions which, when applied from said state, will keep the agent from failure (i.e., "there is a way out"). It is important to note that viability merely indicates the existence of such sequences, but makes no claim in regard to their frequency; there could be *only one* such sequence. The negative characterization is perhaps the more familiar and intuitive: *nonviable* states of the system are those in which the agent has gone "past the point of no return", where collision or failure is no longer avoidable (but perhaps postponable for some finite time).

A related concept to viability is *reachability.* 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. A thorough treatment of reachability is provided, for example, by [Mit02]. In the more formal terminology of the field, these reachable states comprise the "forward reachable set" of x_{init}.

Algorithm 3 RRT w/“Collision Tendency” (RRT-CT)

```
    (inherits query() & grow_tree() from single- or dual-tree RRT)
 1: function NEAREST_NEIGHBOR($\tau, x$)
 2:     $d_{min}, n_{best} \leftarrow \infty, \varnothing$
 3:     for $n \in \tau$ do
 4:         if $\exists$ unexpanded input out of node $n$ then
 5:             $r \leftarrow$ random(),        $r \in [0, 1]$
 6:             if $r > \sigma(n)$ then
 7:                 $d \leftarrow \rho(n, x)$
 8:                 if $d < d_{min}$ then
 9:                     $d_{min}, n_{best} \leftarrow d, n$
10:     return $n_{best}$
11: function PICK_CTRL($x$,$x_{target}$)
12:     $d_{min} \leftarrow \infty$
13:     for $u \in \mathcal{U}$ do
14:         if $u$ has not been expanded for $x$ then
15:             $x_{new} \leftarrow$ sim($x$,$u$)
16:             if failure($x$,$x_{new}$) then
17:                 mark $u$ as expanded
18:                 update_tendencies($x$,$\tau$)
19:             else
20:                 $d \leftarrow \rho(x, x_{new})$
21:                 if $d < d_{min}$ then
22:                     $d_{min}, u_{best} \leftarrow d, u$
23:     mark $u_{best}$ as expanded
24:     return $u_{best}$
25: function UPDATE_TENDENCIES($x$,$\tau$)
26:     $p \leftarrow 1$
27:     while $x$ do
28:         $p \leftarrow p/||\mathcal{U}\,||$
29:         $\sigma(x) \leftarrow \sigma(x) + p$
30:         $x \leftarrow$ parent($x$)
```

where

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

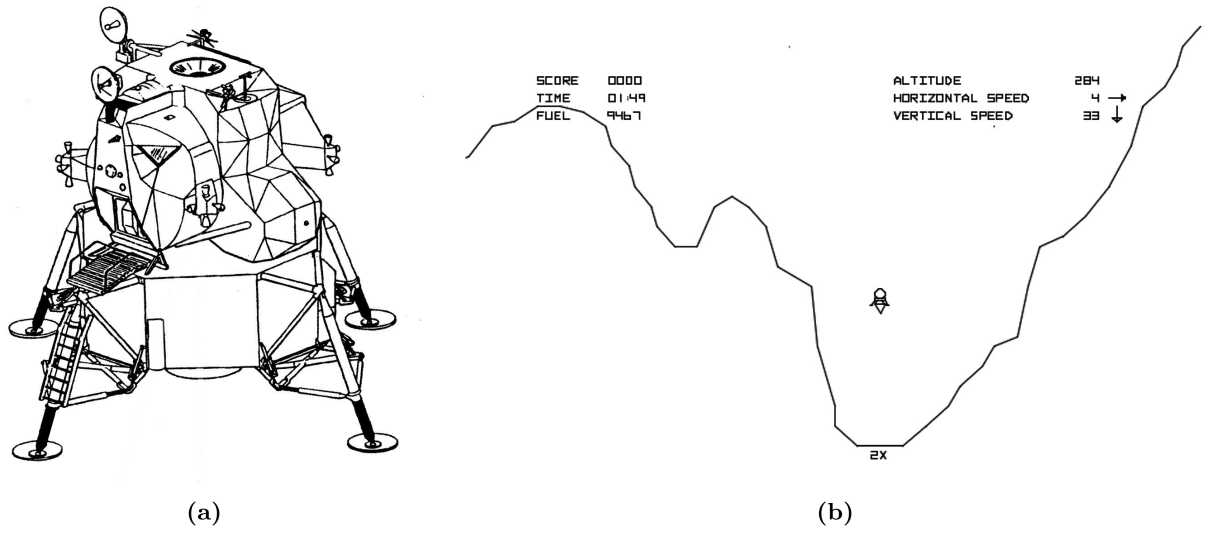


Figure 2.8: Toy problem for viability; **(a)** The Apollo Lunar Module; **(b)** the 1970s arcade game "Lunar Lander"

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 2.8), a simple, 2D simulation of landing an Apollo Lunar Module craft on the moon. In this example we further simplify things 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 $(z, z') = (0, 0)$, subject to $z \geq 0$, where z is the altitude.

A key property of this toy dynamical system is that the agent's thrusters, and thus acceleration, are bounded. This 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 large class of states, since the 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 phenomena correspond to viability and reachability. Figure 2.9 illustrates the lunar lander's state-space, and its division by these classifications into distinct regions.

Formal description

Viability for actuated systems can be captured formally as follows. We first assume that we are given a continuous-time dynamical system:

$$x'(t) = f\big(x(t), u(t)\big), \qquad u(t) \in U(t) \tag{2.1}$$

$$U(t) = g\big(x(t)\big) \tag{2.2}$$

where $x(t)$ is the system state at time t, while $u(t)$ is the control action applied. $U(t)$ is the set of allowable control actions at time t, and as illustrated here, can be dependent upon the system state, although in all our subsequent examples and test cases it is invariant of state.

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 agent is allowed to assume. In most applications of viability in this proposal, K is the set of all collision-free states of the agent.

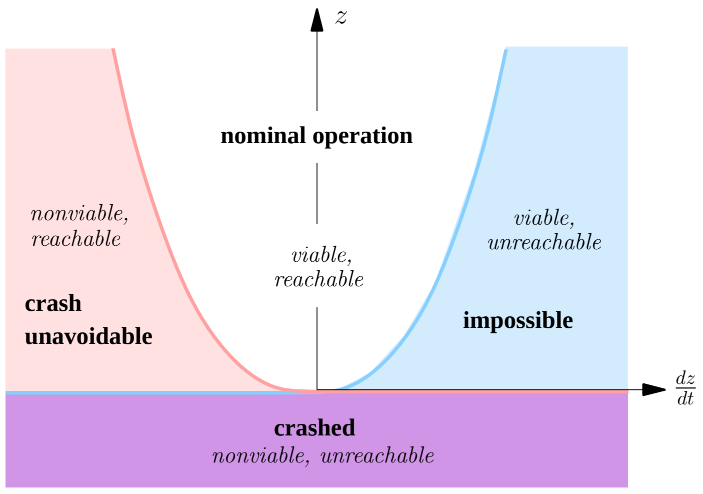


Figure 2.9: The viable and reachable regions of the lunar lander's state-space. The "crash unavoidable" region is nonviable since the downward velocity exceeds the braking power of the lander's (bounded) thrust. The "impossible" region is unreachable since the upward velocity exceeds what can be achieved, even if maximal thrust is applied from altitude $z = 0$.

In this framework, a state is viable if there exists a sequence of control actions that can keep the system within the admissible region K; that is, x_0 is a viable state if:

$$x(0) = x_0$$
$$\exists u(t),\ \forall t \geq 0, \quad x(t) \in K$$

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

$$Viab(K) = \left\{x_0 :\ x(0) = x_0 \bigwedge \exists u(t), \forall t \geq 0,\ x(t) \in K\right\}$$

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

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

$$\begin{aligned} x_{k+1} &= f(x_k, u_k), \quad u_{k+1} \in U_{k+1} && (2.3)\\ U_{k+1} &= g(x_{k+1}) && (2.4) \end{aligned}$$

where x_k, u_k, and U_k are the corresponding system state, control action applied, and the set of allowable actions at the k^{th} time-step, respectively. The viability kernel then becomes:

$$Viab(K) = \left\{x_0 :\ \exists\{u_0, u_1, u_2, \ldots\}, \forall k \in \mathbb{N},\ x_k \in K\right\}$$

In the above, $u(t)$ and u_k are the control inputs to a system which is governed by an external agency, be it a human user, if applied to an interactive system, or the motion planning algorithm, as in our current context. In the most general sense, viability theory attempts to make statements about systems with non-deterministic dynamics; in the systems in this proposal (and thesis) the non-determinism comes from the user's and motion planner's ability to choose the control action to apply next, which the system essentially sees as a random variable due to its relative unpredictability.

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 to us in our 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, refines the discrete approximation 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 is intended to mitigate at least the $\mathcal{U}$ control space curse of dimensionality. 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.

Chapter 3

RRT-Blossom: sustained, non-redundant exploration

The defining property of RRT is the "rapidly-exploring" aspect of its trees, yet this very feature falters for more complex kinodynamic systems. In order to ensure sustained progress, the planner must ensure a steady rate of tree growth, and that the new growth is not redundant. RRT falters because it fails to meet these criteria in many kinodynamic cases. In this section we propose modifications to RRT that directly address both of these two criteria.

3.1 Deficiencies of current RRT variants

The standard RRT algorithm, even though specifically extended to the kinodynamic setting in [LK99], often fails to make significant progress on queries for more complex dynamical systems, even when given large amounts of time. For example, as shown in Figure 3.1, RRT performs very poorly with the kinodynamic bike described in section 1.2, producing little progress even after 20 minutes of computation.

One major factor contributing to RRT's poor performance in this example is the algorithm's method for choosing the next tree node to explore (i.e., pick the node that is closest to a randomly sampled target state, x_{tgt}). This strategy grinds nearly to a halt when the tree develops a prominent but nonviable branch: since the branch is nonviable all its evolutions are doomed to failure, while

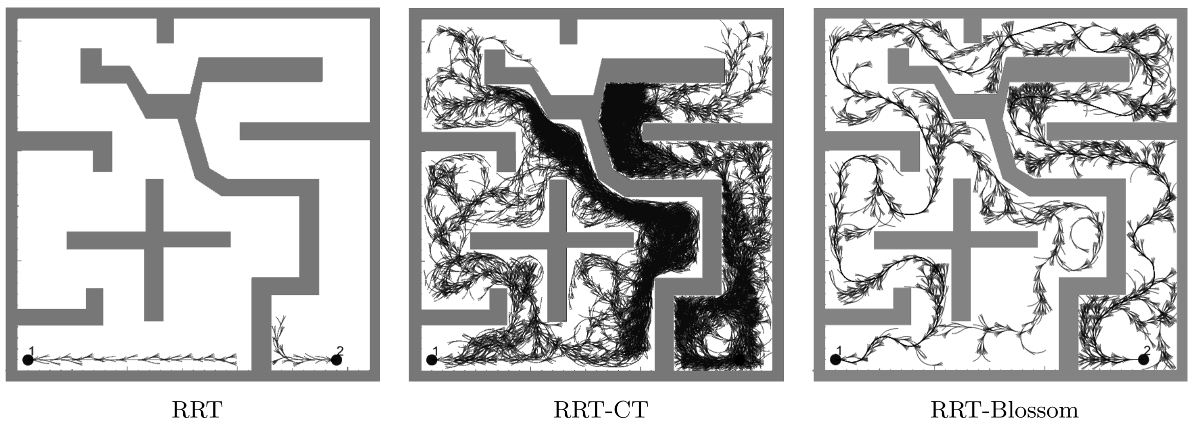


Figure 3.1: Comparison of evolved tree structure for the kinodynamic bike after a set amount of time. Note: all three algorithms use Extend operation for tree growth (i.e., all are variants of RRTExtExt).

its protrusion makes it a lightning rod for expansion trials, thus leading to many wasted iterations on futile expansion attempts. The problem is further exacerbated by RRT's lack of memory, especially of such previous failed attempts. A related issue that troubles RRT is the interplay of the above node selection strategy and RRT's requirement that, in order to be implemented, an edge must make progress towards x_{tgt}. For many kinodynamic systems the set of all possible outbound trajectories out of an arbitrary node is fairly "clumped", which leads to reduced performance if the node is prominent and this set is oriented towards explored space. In such cases, the node will again garner many expansion attempts, yet the orientation of the "clump" rules out any chance of tree growth. This leads to many wasted iterations, just like with the earlier, nonviable branches. Figure 3.2 illustrates this more clearly.

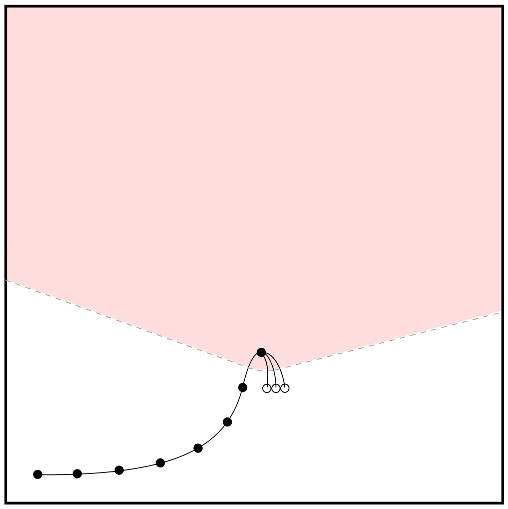


Figure 3.2: One problem with kinodynamic RRT: "clumped", wayward evolutions. If x_{tgt} is chosen anywhere within the shaded area, the prominent (top-most) tree node will be chosen as x_{near}; at the same time, since all its evolutions will recede from the chosen x_{tgt}, no tree growth will occur. The more prominent such a node, the larger the unfavourable area (i.e., the node's Voronoi polygon), the more planner iterations are wasted due to this node.

The RRT-CT algorithm resolves most of the above problems. Firstly, it allows edges that recede from x_{tgt}, yielding a much more sustained rate of exploration for difficult agents. Secondly, it remembers which node expansion attempts led to failure, and disqualifies these from future (redundant) consideration. Unfortunately its undiscerning admission of all x_{tgt}-receding edges leads to many that end up backtracking into already explored space, especially in pockets and cul-de-sacs in the environment, wasting memory and slowing down the planner.

3.2 RRT-Blossom

In essence, RRT-CT is trading one problem for another, accepting some loss in time and space efficiency to gain much better performance with difficult agents. It is important to note though that the net effect is positive, in that the resultant disadvantages are smaller. RRT-Blossom[KvdP06] constitutes a similar, successive trade-off, addressing RRT-CT's weakness at the cost of introducing a smaller one. It is expected that with further refinements, such as that in Chapter 4, the difficulties can be whittled down to an insignificant level, yielding a fully robust kinodynamic motion planner.

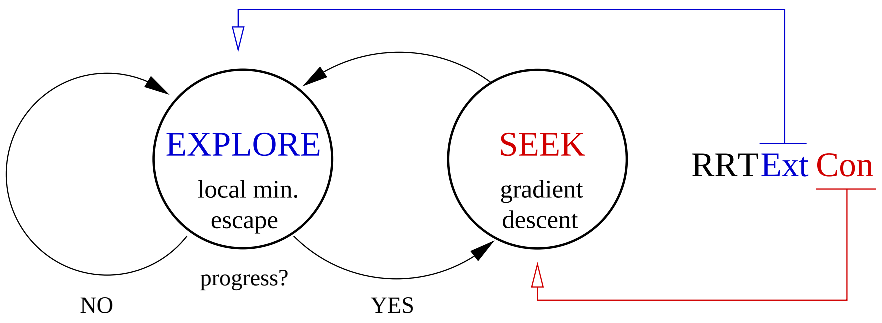


Figure 3.3: dual-tree RRT as a simple finite state machine (FSM)

"Potential field planner" viewpoint

The ideas behind RRT-Blossom are best exposed by first noting similarities between RRT and a potential field planner. As described in section 2.1, the central mechanism of such planners is the use of gradient descent over a "potential field", typically an approximation of the distance to goal that has been in some way modulated by the presence of obstacles. Unfortunately such methods are frequently susceptible to getting trapped in local minima, thus necessitating some form of an escape mechanism.

The single-tree, x_{goal}-biased RRT algorithm shares some basic ideas with potential field planners when its operation is viewed on a macro scale. The x_{goal}-biased iterations effect a descent down a potential field, viz. $\rho(x, x_{goal})$, while the tree growth towards random targets in state-space acts as a local minima escape mechanism.

The dual-tree variant of RRT can be likewise framed by viewing it in terms of a simple finite state machine, as shown in Figure 3.3. 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. The overall potential field planner parallel here is similar to the single-tree case, with the difference that the potential fields encourages the trees to grow towards each other rather than growing towards a fixed goal state.

Finally, we can think of RRT-CT in the potential-field planner context, where it introduces changes that result in an escape mechanism akin to flood-filling of local minima. A significant part of RRT-CT's power comes from a modification to the control action picking mechanism, viz. initializing[1] d_{min} to ∞, rather than to $\rho(x_{parent}, x_{goal})$. As pointed out earlier, the main consequence of this change is that it allows the creation of edges which *recede* from their appointed target. At first this may seem undesirable, as many such edges will regress into space already explored by the tree, but this is not always the case (see Figure 3.4). Receding edges which do not regress are highly beneficial since they provide tree growth in iterations which would otherwise be wasted, thus generating a more sustained rate of tree growth, an important characteristic for a flood-fill. Unfortunately RRT-CT has no mechanism to separate out the non-regressing edges and thus accepts all receding edges, leading to poor performance in regression-prone cases (e.g., kinematic systems). RRT-CT's other, more explicit changes, namely the tracking of failed edge expansions and use of collision tendency, do not address this problem, although they do further improve the planner's node generation rate (even though many of these nodes are redundant).

[1]cf. line 12 in Algorithm 3 vs. line 16 in Algorithm 1

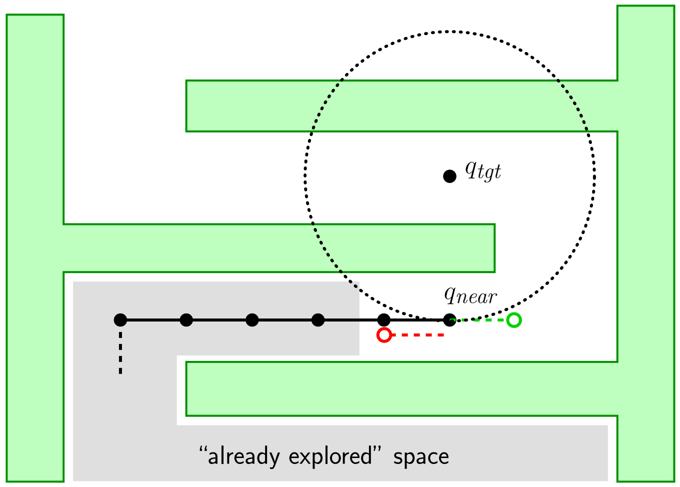


Figure 3.4: Receding edges and regression; in above RRT iteration, there are only two possible expansions (dashed) from q_{near} (leftward expansion is offset downward to avoid superposition). Both recede from q_{tgt} (i.e., they lie outside the dotted circle), hence are rejected by RRT. Receding edges can often be useful though (green, rightward), but care must be taken to avoid those that regresses into already-explored space (red, leftward).

Regression avoidance

The main contribution of RRT-Blossom then is to introduce a robust, local, flood-fill escape mechanism into RRT, similar to that in RRT-CT, but one that avoids regression into already explored space. This flood-fill behaviour proceeds in a stochastic fashion, unlike in level-set-like methods where the fill proceeds systematically outwards, in an exhaustive fashion; this allows for faster escapes from deep minima. A secondary improvement is the instantiation of *all* eligible edges when expanding a node[2] (subject to collision and regression constraints), not just the single best one. Although RRT-CT handles unsuccessful expansions efficiently, by ensuring they are never re-attempted, it is wasteful with regards to the others. Specifically, after it surveys the expansions out of a node and instantiates one of them, it then discards useful information for the remaining *collision-free* expansions, such as their end-states and collision-free status. Instantiating all eligible expansions mitigates this, and has little negative cost: in constrained regions these expansions would eventually be instantiated anyhow, while expansive spaces are traversed quickly with few node expansions, thus generating only negligible overhead. Also, such blossoming is more consistent with RRT's "rapidly-exploring" spirit.

Implementing a robust regression constraint is challenging, and is an another contribution of our work [KvdP06]. Computing an explicit model of the portion of state-space that is considered "already explored" is unwieldy because of issues of dimensionality, and it is furthermore difficult to even define the volume of state-space that is "occupied" by a branch of a search tree. These difficulties are sidestepped by using an implicit approximation that captures the desired spirit of the term, an approximation which has been found to be highly effective: a "leaf" edge (x_{parent}, x_{leaf}) is considered to be regressing if a node *other* than x_{parent} is closest to x_{leaf}:

$$\text{regression}: \quad \exists x \in T \ : \ \rho(x, x_{leaf}) < \rho(x_{parent}, x_{leaf})$$

where T is the search tree, and ρ is a distance metric. Figure 3.5 further illustrates the concept, while Algorithm 4, the pseudocode for RRT-Blossom, shows how this test is employed.

[2]Hence the "blossom" moniker.

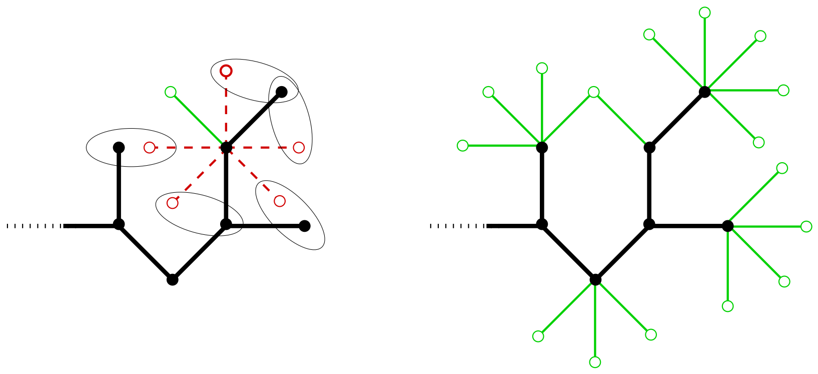


Figure 3.5: Implementing "regression"; Left: possible expansions for a particular node; all the red expansions (dashed) are *regressing* since a node other than the parent is closest to proposed leaf node (indicated with loops). Only the single edge in green is suitable for instantiation. Right: all the expansions in the tree that do *not* regress for the depicted tree state.

Interplay of viability & regression constraint

The above definition works well for simple kinematic systems, but it becomes problematic for nonholonomic or kinodynamic systems. Figure 3.6 illustrates one such case. Here, the "left turn" path of the car is expanded first, but all its subsequent evolutions lead to collisions. The "straight ahead" path would be feasible, on the other hand, but unfortunately it is disallowed because instantiating it would constitute a regression into space already explored by the "left turn" path.

The general problem is that it is possible for a nonviable edge, one that by definition cannot be part of any solution trajectory (because it inevitably leads to failure), to block a neighbouring, viable expansion. This is particularly detrimental when the blocked expansion lies on the critical path, as this makes solving the query outright impossible. The solution, then, is to simply disregard such nonviable edges when determining regression.

Unfortunately the nonviability of edges is not known ahead of time, and must therefore be incorporated retroactively when it is discovered. This is implemented by annotating nodes 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 for expansion are marked `untried`. Upon instantiation they become `live`. If the expansion is disallowed due to regression, it is marked `dormant`. Finally, `dead` edges are ones that have been found to have left viable space. Figure 3.7 shows the transitions in greater detail.

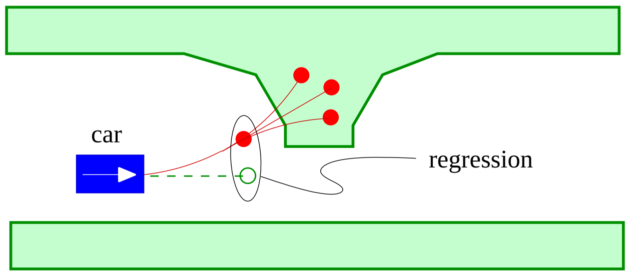


Figure 3.6: Interplay of viability and the regression constraint: the green (dashed) expansion is blocked by an extant nonviable edge because instantiating it would constitute a regression. Since the green edge is essential to any solution, it is now impossible for the planner to succeed.

Algorithm 4 RRT-blossom

```
(inherits query() from dual-tree RRT)
1: function GROW_TREE($\tau$,$x_{target}$)
2:     $x_{near} \leftarrow$ nearest_neighbor($\tau, x_{target}$)
3:     $x_{new} \leftarrow$ node_blossom($x_{near}, x_{target}, \tau$)
4:     return $x_{new}$
5: function NODE_BLOSSOM($x$,$x_{target}$,$\tau$)
6:     for $u \in \mathcal{U}$ do
7:         $x_{new} \leftarrow$ sim($x, u$)
8:         if failure($x, u, x_{new}$) then
9:             next $u$
10:        if regression($x, x_{new}, \tau$) then
11:            next $u$
12:        $\tau \leftarrow \tau +$ new_edge($x, u$)
13:    return the new node closest to $x_{target}$
14: function REGRESSION($x_{parent}$, $x_{new}$, $\tau$)
15:    for node $n \in \tau$ do
16:        if $\rho(n, x_{new}) < \rho(x_{parent}, x_{new})$ then
17:            return True
18:    return False
```

Since changing the status of an edge *may* precipitate a change in the parent, status changes must be propagated up the tree. This is done by traveling up the parent hierarchy towards 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.

Dormant deadlock

Despite these measures it is still possible for the planner to become unduly stuck. This occurs when all paths towards the goal, usually narrow choke-points, have been cut off by older, lengthier branches, ones which cannot explore the passages, even though they pass the closest to them. Figure 3.8 shows an example.

Fortunately these occasions are easily identified and mitigated. When the remaining accessible free-space has been exhausted, the `dormant` condition starts "backing up" the tree towards the root

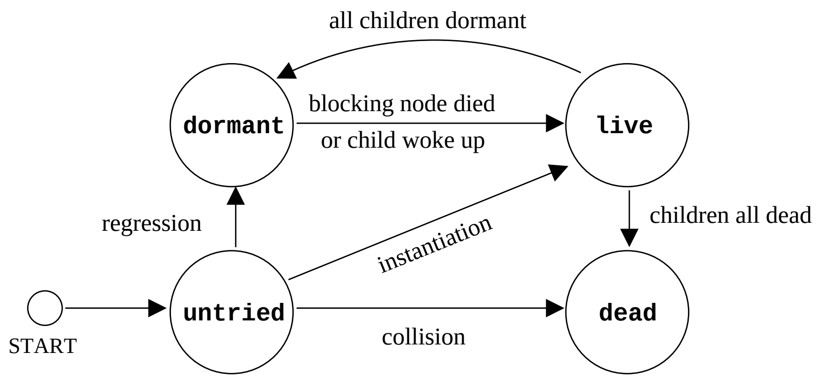


Figure 3.7: Progression of the viability status of an edge

node, eventually reaching it once all other lines of exploration have been exhausted. This signals the deadlock to the planner, which then allows the very next expansion attempt to disregard the regression constraint. In practice this works well, and more importantly, it has no impact on queries unaffected by this issue.

3.3 Results

This section presents a subset of our results; the thesis will present additional systems (also, see [KvdP06]). In particular, Figure 3.9 shows a box-and-whisker plot[3] (or "boxplot") of the runtimes for RRT-CT and the proposed RRT-Blossom algorithm when answering a query for the kinodynamic bike; standard RRT was not included in the plot as it failed to make any significant progress in the allotted time (about 20 minutes), let alone find a solution. Figure 3.1, presented at beginning of chapter, compares the structure of the evolved trees under the three algorithms (in the same time span). RRT's clear lack of progress is very prominent, as is RRT-CT's problem with re-exploration of the area bordering the central separating dividing wall and pockets therein. It should be noted that the RRT-Blossom tree structure presented is that at the moment of solution discovery, and hence represents work over a much shorter time span than that of the other two algorithms which ran the full allotted duration.

A more complete discussion of RRT-Blossom will be presented in the thesis, but it is worth making a few key observations here. As a byproduct of its operation, RRT-Blossom can often establish the reachability or viability of the states it encounters (e.g., any state encountered by a tree is obviously reachable from its root, while any state whose progeny is discovered to always lead to collision is clearly nonviable). This presents a valuable data source when later on we require training samples for viability and reachability models.

As stated earlier, the primary goal of RRT-Blossom is to maintain efficient and rapid rate of exploration, despite the difficulties presented by kinodynamic systems, but it achieves this goal only partially: 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 and unhindered by the multitude of `dead` branches there, it proceeds along expected lines, and is shortly found to be

[3]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 are considered "outliers".

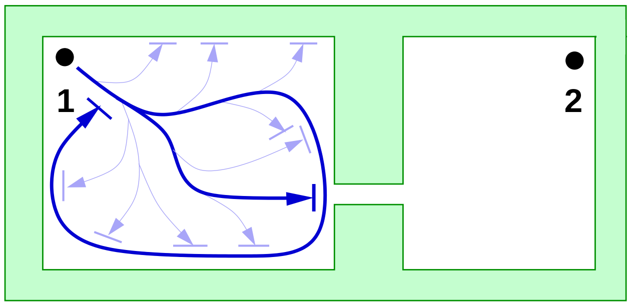


Figure 3.8: `dormant` deadlock: a viable branch may cutoff access to a critical passage without being able to explore it itself. This limits the planner's exploration to the "fenced off" area, and once this is exhausted, the remaining non-`dead` branches are locked in a cycle, mutually blocking each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

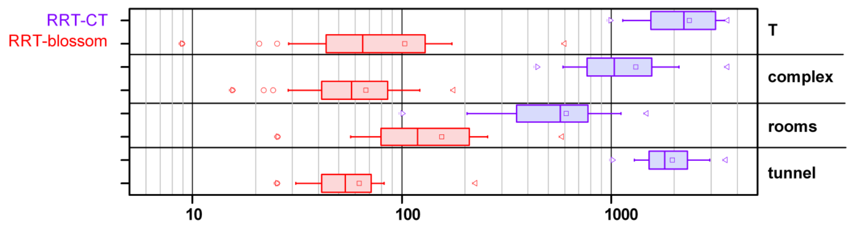


Figure 3.9: Algorithm runtimes for the kinodynamic bike, in seconds, for various test environments. Samples per boxplot: 40.

`dead` itself, thus adding 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 recognize ahead of time the nonviable regions of state-space. The motion planner could then simply ignore and avoid exploring those parts, thus completely side-stepping the above issue. Such viability precognition is the subject of the research presented in the following chapter.

Chapter 4

Motion planning with viability models

As indicated in the previous chapter, the key remaining weakness of RRT-Blossom is its inability to prevent re-exploration in nonviable areas, which is doubly wasteful since these areas normally can never even lead to a solution. The most effective way to correct this is to predict the nonviable areas *a priori* and simply avoid exploring them, thus cleanly sidestepping the problem. This optimization is not limited to RRT-Blossom; almost any kinodynamic planner could benefit greatly from such search-space culling, leading to "smarter planning".

Viability can be modeled in a number of ways. The next chapter, Chapter 5, looks at modeling viability globally, whereby the models are parametrized 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 parametrized by the local context of the agent. Such local models are usually preferable since they are applicable to a wider range of similar-yet-different environments, not just the original one used to train the model. They are also better suited for dynamic environments, since snapshots of the environment at different time points can be alternatively viewed as a set of static, similar-yet-different environments. Local models also often capture the agent's viability in a more efficient and compact form.

One way to implement such locally-parametrized viability models is to first think in terms of the *combined* state of the agent and the environment, and then to localize it by using a set of agent-mounted virtual sensors. A straight-forward example for a bike would be to outfit it with a spread of three sensors, oriented at $\{-\alpha, 0, +\alpha\}$ degrees relative to the bike's forward direction (see Figure 4.1). Such localized perception of the system state has been explored in control literature, yielding interesting and robust localized behaviour-based AI and control [Bra84, Bro86, Ark98].

In the subsequent sections we formalize the combined system state, the virtual sensors, and the resultant localized state of the system. Next, we describe the framework and implementation details of such localized viability models, as well as how to implement filtering in a planner using these models. This is followed by sample results, as well as a discussion of unresolved issues and open problems.

4.1 Sensing and situation encoding

Traditionally motion planners treat x, the state of the agent, and e, that of the environment[1], as separate entities, and only bring them together during collision checks. This is an unnecessarily limiting dichotomy; often it is more convenient to work with the combined, full *system state*, which we will denote with $x^+ = (x, e)$. The corresponding state spaces can be analogously combined: if

[1]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 thus time-varying.

$x \in \mathcal{X}$ and $e \in \mathcal{E}$ then $x^+ \in \mathcal{X}^+$, and $\mathcal{X}^+ = \mathcal{X} \times \mathcal{E}$.

A planner can be endowed with "sight" by fitting it with a number of *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:

$$\sigma(x^+) : \mathcal{X}^+ \rightarrow \mathbb{R}$$

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

$$s = \big(\sigma_1(x^+),\ \sigma_2(x^+),\ \ldots\big), \quad s \in \mathcal{S}$$

which we refer to as the *sensory state*. The set of all possible sensory states forms the *sensory space* $\mathcal{S}$.

It is generally unproductive to learn or reason about an agent's behaviour using the full system state x^+. Acquired knowledge that is parameterized using such global descriptors does not generalize well to other, similar-but-different environments. The sensory state s, on the other hand, when augmented with relevant variables describing the agent's internal state, can be seen as projecting x^+ into a local description that is usually more generalizable. Formally, we define the *locally situated state* of the agent

$$\lambda = (s, \hat{x}), \qquad \lambda \in \Lambda$$

where $\hat{x} \subset x$ are the 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, but often smaller.[2]

For example, in the dynamic bike system,

$$\begin{aligned} x &= (p, \theta, \phi, \phi') \\ s &= (\sigma_L, \sigma_F, \sigma_R) \\ \lambda &= (\sigma_L, \sigma_F, \sigma_R, \phi, \phi') \end{aligned}$$

where p is bike's position, θ is its orientation, and *phi* is its lean angle, while $\sigma_L, \sigma_F, \sigma_R$ are the three sensor readings.

[2]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.

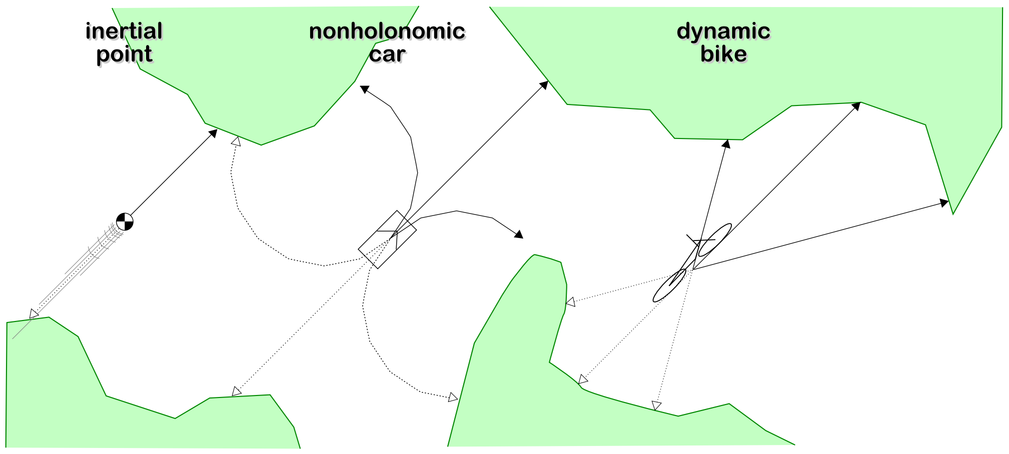


Figure 4.1: Virtual sensors for various agents. Lines indicate (virtual) agent-mounted range sensors used in planning (dotted lines indicate sensors used with x_{goal} tree). They are discussed in §4.3.

4.2 Modeling local viability

The locally situated state space Λ provides a more compact and general projection of the combined agent-environment state. By taking note of situated states that are encountered in successful solution trajectories, the planner can build a large pool of exemplars, its experience of "auspicious" (i.e., good) situations. This knowledge can then be leveraged during further planning by discouraging the planner from exploring regions where the density of such auspicious samples is low or non-existent, since any situation which significantly strays from past experience is likely to do so precisely because it does not lead to success, and hence cannot aid in finding a solution.

Viability

The notion of a situated state being 'good' or 'bad' correlates strongly with whether the corresponding system state x^+ is viable. In this context, our approach limits planner exploration to viable space. The underlying idea is that, since it is impossible for a nonviable state to lead to a viable one, then any encountered nonviable states cannot contribute to a solution, and hence represent wasted effort. The exceptional case where x_{goal} is itself nonviable is treated later on.

Our proposed *viability filtering* of planner exploration requires a model of the subject's viability, which we derive from the example viable states collected from previously planned motions. The planner trains (and then later consults) a *viability oracle*

$$\Omega_v(\lambda) : \Lambda \rightarrow \{\texttt{true}, \texttt{false}\}$$

where `true` and `false` correspond to "viable" and "nonviable" labels, respectively.

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 poor choice in training procedure parameters, and is easily corrected. The second type of error is inherent in the collected data, and is much harder to fix. The problem stems from the agent's set of sensors being unable to disambiguate between pairs of system states in certain situations, one viable and one nonviable, and as a result mapping them both to the same locally situated state λ. Since the oracle is deterministic, it must therefore always predict wrong for one of these problematic states. Section 4.4 discusses these errors further, and their consequences.

Implementation

The first step in implementing a planner with viability filtering is the collection of sample viable states. This can be done in a number of ways. An ideal approach would rely on "bootstrapping": the planner starts out with no viability knowledge but progressively builds a viability model by scanning its own solutions for novel viability states and incorporating them into the oracle. However, such online bootstrapping is problematic as we describe in section 4.4. Instead, we collect samples of viable states from very long random-walk trajectories created by applying a random control action at each time step and backtracking whenever the agent encounters failure.

Only positive samples of viability are collected. Nonviable states are not collected since nonviability is far more difficult to determine: to prove a state is viable it is sufficient to demonstrate a single viable trajectory out of it; to prove nonviability one must show that all trajectories out of the state necessarily end in collision. But proving viability requires a concession since a "viable trajectory" is one that can self-perpetuate *indefinitely*, and thus is not something easy to show. We thus adopt a time horizon in our viability deliberations (e.g., 10s in our implementation), such that any state which supports a collision-free trajectory longer than this duration is deemed viable, with the expectation that most failure modes of the subject are confined to durations shorter than the time horizon. In our experience, any error (i.e., contamination of experience pool with nonviable states) introduced with this approximation is insignificant in relation to other sources of error in our approach.

Once a sufficient number of samples have been collected, the oracle can be trained. Since the collected samples all belong to the same class (i.e., the "viable" set), this is naturally a one-class classification problem. We use the one-class classifier in the excellent libSVM[CL01a] library.

Given a trained viability oracle, it is trivial to instrument a planner with 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()`:

Algorithm 5 Function `is_nonviable(`x, e`)`

```
if is_collision($x, e$) then
    return True
$s \leftarrow \sigma_1(x,e),\ \sigma_2(x,e),\ \ldots$
$\hat{x} \leftarrow$extract_internal_state($x$)
$\lambda \leftarrow (s, \hat{x})$
return $\neg\ \Omega_v\ (\lambda)$
```

The function `is_nonviable()` still calls `is_collision()` since usually the viability model will not be perfect, thus if used by itself, could occasionally admit an in-collision state.

Single-tree considerations

Given our finite time-horizon definition of viability, any state in the sample datasets (i.e., prior successful solution trajectories) that is followed by one time-horizon's worth of motion (e.g., 10s) is deemed to be viable. Thus the subset of viable states is readily extracted by discarding one time-horizon's worth of data from the end of every training trajectory in the datasets.

A related issue is that a single-tree planner (i.e., one using only forward simulation) will be unable to find a solution if x_{goal} is itself nonviable. This is because the planner will be actively deterred by the viability filter from completing the last leg of any solution. Since this is not an issue in the more common, dual-tree planner approach, we forgo exploring solutions to this problem.

Dual-tree considerations

In dual-tree planners, the x_{goal} tree is usually built using reverse-time simulation of system dynamics. Such trees require some obvious modifications to our filtering approach. Since the subject generally moves "backwards" in reverse-time, it is necessary to flip the sensor orientation front-to-back (see dotted lines in Figure 4.1). This further implies that reverse-time trees need to build and use their own oracle, $\Omega_{v_{rev}}$, based on data from this "flipped" sensor set. The resulting oracle models viability in reverse-time, which is equivalent to modeling reachability for a forward-simulation setting (see Figure 2.9), thus requiring that training trajectories be trimmed from the front end rather than the tail.

The dual-tree approach sidesteps the single-tree problem of a nonviable x_{goal}. This ceases to be an issue with two trees since a solution requires only a tree-tree connection, not a tree-node one. That is, the x_{init} tree, T_{init}, no longer needs to reach x_{goal}; it only needs to reach any node in the x_{goal} tree, T_{goal}, a much larger target. Analogously, x_{init} does not have to lie in reachable space since T_{goal} need only reach any node in the tree T_{init}. The only consequence of this is that the two trees may only meet in space which is both, viable and reachable, but this does not seem to have any noteworthy impact on the resulting solutions.

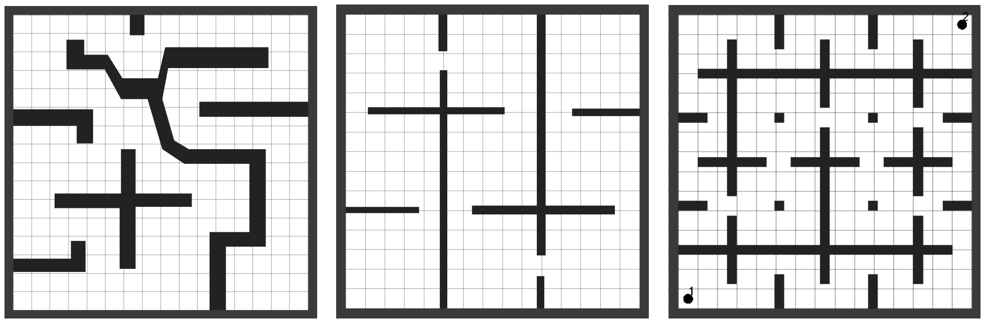


Figure 4.2: Example problem environments tested.

4.3 Results

Agents & environments

We apply the viability filtering method to three agents. Only the forward-time sensors are described here; for the reverse-time trees the logical analogues are used. Both sensor sets are illustrated in Figure 4.1, while Figure 4.2 depicts the environments used in the experiments. This section also uses the following notation: p refers to the subject's 2D position in the environment; $\mathcal{U}$ refers to the subject's control input space, the set of discrete control actions allowed.

The "inertial point" robot is a point-mass with inertia and has four constant-force thrusters mounted in the four cardinal directions. During operation the agent is required to have at all times one, and only one of these thrusters on. The agent has upper and lower bounds on its velocity.[3] The subject's state, control actions, sensory state, and locally situated state are:

$$\begin{aligned} x &= (p, p') & s &= \sigma_{p'} \\ \mathcal{U} &= \{\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}\} & \lambda &= (\sigma_{p'}, \|p'\|) \end{aligned}$$

where $\sigma_{p'}$ measures the distance from agent to the nearest obstacle along the velocity vector p'.

The non-holonomic car used is very similar to that in [KvdP06], but less maneuverable, with a smaller maximum steering angle ψ_{max}. The relevant parameters are:

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

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 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.

The last and most complex subject is the dynamic bike model from [KvdP06], with the following parametrization:

$$\begin{aligned} &x = (p, \theta, \phi, \phi') \qquad s = (\sigma_L, \sigma_F, \sigma_R) \qquad \lambda = (\sigma_L, \sigma_F, \sigma_R, \phi, \phi') \\ &\mathcal{U} = \{-\psi_{max}, -\tfrac{\psi_{max}}{2},\ 0,\ \tfrac{\psi_{max}}{2},\ \psi_{max}\} \end{aligned}$$

[3] A small lower bound was found useful in expediting the planning process with all planners. Without it planners spent much 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.

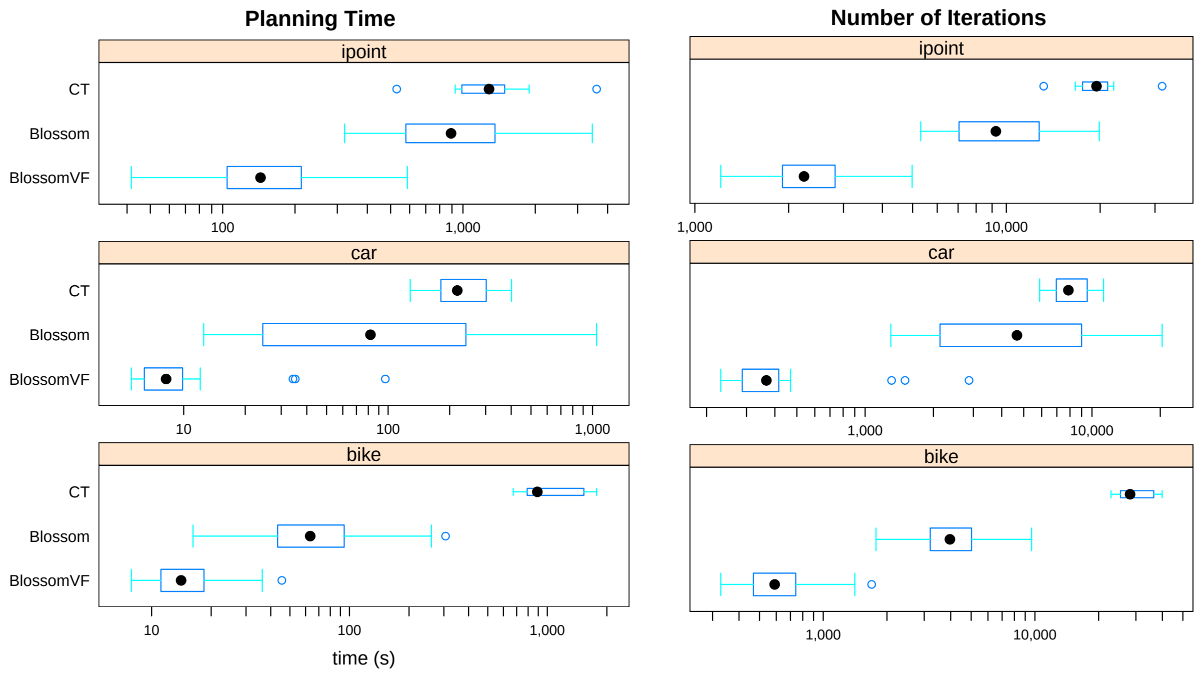


Figure 4.3: Box-and-whisker plots of planning time and required number of iterations for "complex" environment. The x-axes are logarithmic to enlarge regions of detail; otherwise egregious outliers tend to marginalize the area of interest.

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.

Implementation details

The implementation was done using a test platform written in Python 2.4, on a Pentium IV 2.4GHz Linux machine (kernel 2.6). The base planning algorithm used was the dual-tree RRT-Blossom presented in [KvdP06]. 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.

The SVM learning process is straight-forward. Prior to learning, the training samples $\lambda \in \Lambda$ are first column-standardized[4], and then further scaled in some of the dimensions to give those features more "resolution". Radial Basis Function (RBF) kernel is used, usually with $\gamma \in \{0.5, 1, 2\}$, while $\nu \in \{0.005, 0.01\}$.

Performance evaluation

Table 4.1 summarizes the numerical findings. The labels "CT", "Blossom", and "BlossomVF" refer respectively to RRT-CT[CL01b], RRT-Blossom[KvdP06] and our present algorithm (i.e., RRT-Blossom with Viability Filtering). The values are averages over 20 runs for RRT-Blossom and RRT-Blossom/VF, and over 10 runs for RRT-CT. Figure 4.3 further details the results for the

[4] $\lambda_i \leftarrow (\lambda_i - \mu_i)/\sigma_i$, where λ_i is the i'th coordinate of vector λ, and μ_i and σ_i are the corresponding mean and standard deviation, respectively.

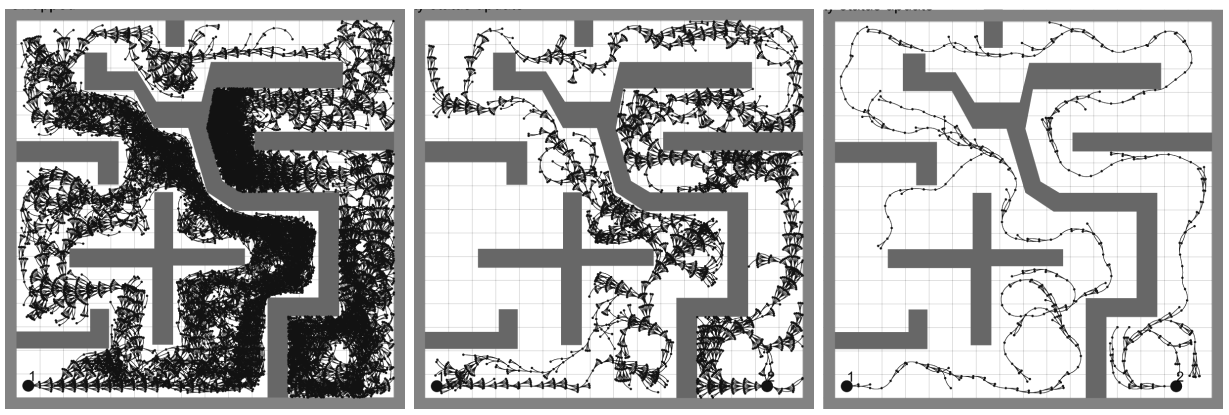


Figure 4.4: Visual comparison of tree density and structure for the bike with various RRT variants. The proposed viability-filtering planner (on right) achieves a solution with noticeably less effort.

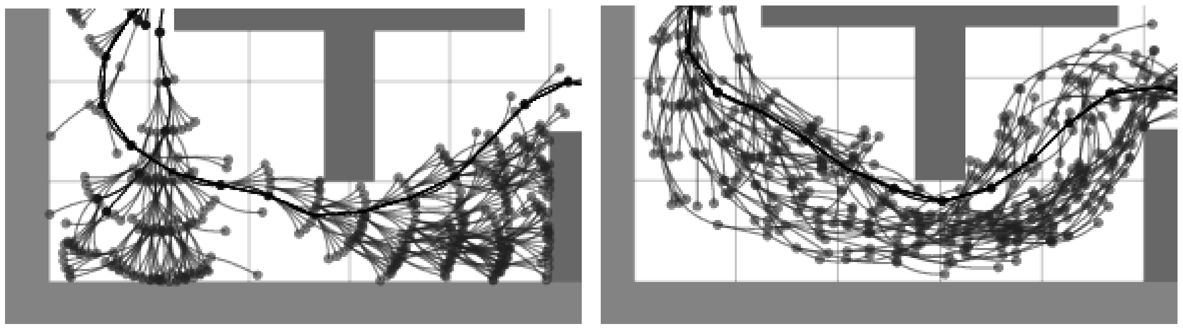


Figure 4.5: Magnified view of tree structures for the bike in a more difficult environment. The filtering planner (on right) conspicuously avoids probing the corners and consists almost exclusively of viable trajectories.

"complex" environment using box-and-whisker[5] plots for "time taken" and "number of iterations expended"; the plot for "number of nodes created" closely mirrors that for iterations, hence was not included. Finally Figures 4.4 and 4.5 illustrate the effect that the viability filtering has on tree structure.

It is important to note that each agent uses the same viability and reachability models for *all three environments*. These models were all trained on data from the "complex" environment, which largely explains why the gains in that environment are the most dramatic (shaded row in Table 4.1). At the same time, it is encouraging to see that the models are still effective when applied to other environments. The diminished gains in "non-native" environments are a consequence of the sensors' perceptual limits introducing environment-specific artifacts into the learned model, and such artifacts are not generally transferrable.

The planning algorithms can be compared on runtime alone, but it is more instructive to include the other columns in the comparison since they present a clearer picture of the reduction in

[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".

Table 4.1: Performance comparison

		inertial point			**car**			**bike**		
environment	**algorithm**	**time (s)**	**# iter.**	**# nodes**	**time (s)**	**# iter.**	**# nodes**	**time (s)**	**# iter.**	**# nodes**
complex	CT	1448.4	20,064.0	21,468.5	301.8	9408.2	9969.4	1088.7	29,775.6	17,070.5
	Blossom	1033.6	10,033.7	12,007.4	186.8	6187.0	6439.3	72.4	4137.1	4019.1
	BlossomVF	**171.1**	2401.0	4208.9	**12.2**	478.7	720.9	**15.9**	651.3	805.5
rooms-525	CT	488.5	14,157.2	12,088.3	204.3	6404.2	7892.0	943.4	23,521.1	14,245.4
	Blossom	1287.2	10,027.3	13,711.2	1428.5	19,308.5	19,895.9	193.3	6904.0	6695.1
	BlossomVF	**326.0**	3232.5	5682.9	**63.4**	1954.5	2503.2	**89.6**	2669.8	2773.4
rooms-IKEA	CT	1403.7	26,018.4	20,114.4	1142.6	22,044.1	20,086.9	409.8	19,146.4	9765.1
	Blossom	1062.4	9839.0	12,241.0	1425.5	19,040.0	19,605.0	105.7	4910.3	4750.3
	BlossomVF	**481.2**	4578.1	7113.6	**685.0**	7144.6	9617.9	**40.8**	1066.1	1291.8

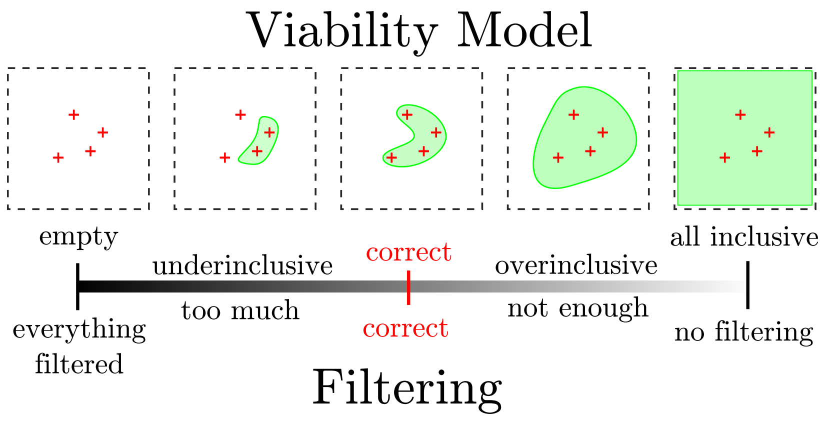


Figure 4.6: Amount of viability filtering as a function of model error.

planning effort. The time column presents these gains in a diluted form, since it also incorporates time consumed in computing sensor values and querying the oracle, both of which could likely be significantly improved in an optimized implementation.

4.4 Discussion

Choice of sensors

The benefit of 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, as stated earlier, 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 emphasis of 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 is far cheaper to compute the sensory state of a car outfitted with three linear rangefinders, in place of the whiskers, but this produces poorer results, despite the much reduced sensor computation time.

Consequences of oracle error

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 ought to. 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 of the search space, causing the planner to gradually regress into the host planning algorithm (i.e., the algorithm without filtering) as this error increases. In general, the amount of viability filtering is a function of model error, and spans a continuous spectrum as illustrated in Figure 4.6.

Bootstrapping & scarcity of samples

There are two key issues with bootstrapping. First, since viability is modeled using a one-class classifier, which generally attempts to wrap as tight a boundary as possible around the training samples, and since there would be very few samples in the initial stages of bootstrapping, it is clear that the model would be heavily underinclusive, leading to severely excessive filtering. Even if the planner manages to answer a query within such exacting constraints, the solution will necessarily consist only of states which comply with the current model; the filtering operation thus prevents the discovery of novel samples.

Both problems can likely be overcome by stochastically phasing-in the viability filtering. That is, filtering could be applied only to iterations in 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 characterize the number of samples required to adequately capture the viable region. In general, the model grows only when novel samples are encountered, ones that lie outside current model bounds; the rate of growth is directly tied to the rate at which such samples appear, as well as the degree of their novelty. The best one can do then in characterizing the adequate sample size is by doing so in terms of a threshold value on the rate of appearance of novel samples (i.e., one has an adequate model when the novel-sample rate falls below threshold).[6]

Oracle transferability

As the results show, a viability model can be effective in environments other than the one that was used in training, but there are limits. 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 will mostly span regions of Λ with large sensor values, whereas the latter environment will generally constrain agent operation to the vicinity of Λ's origin (i.e., all sensors will tend to have low values). The obvious remedy to counter such oracle over-specialization is to train them on sample pools obtained from a variety of dissimilar environments. Initial experiments in such compositing of training data for the car have yielded good results.

[6]This is still not very general, as it assumes that the training samples are drawn from a uniform distribution over Λ, which is usually not the case.

Chapter 5

Safety enforcement with viability models

Many user-controlled vehicles and systems (e.g., cars, helicopters, airplanes, etc.) are susceptible to particularly dangerous failures (e.g., collision), and hence any form of automatic safety assurance would be highly desirable in these systems. Ideally such a mechanism should be minimally intrusive, overriding the user's actions only if they lead to unavoidable failure. The earlier discussed viability models are particularly suitable for this purpose. Specifically, the safety enforcement approach proposed in this chapter gives the user free reign over the agent, except in instances where this would cause the agent to leave viable space; in those cases the mechanism overrides the user's input with an alternate control action that does not breach the "viability envelope". At least one such action must, by definition, always exist for a viable state.

A sample implementation for a car can be seen in Figures 5.1 and 5.2. In the scenario pictured the user is in fact steering with the explicit purpose of leaving the track (see v_k in Figure 5.2), but is overruled (u_k) when necessary.

Since such safety enforcement only activates near the boundaries of the viability kernel $Viab(K)$, this is where most of our attention will be focussed. In this chapter we will refer to this boundary as the "viability envelope", a term inspired by common usage (e.g., the "flight envelope", "pushing the envelope", etc). We sometimes also refer to this as the "control envelope" since a "viable" state is also called "controllable" (in the field of control).

Related safety boundary problems have been investigated in [Mit02], where backward reachable

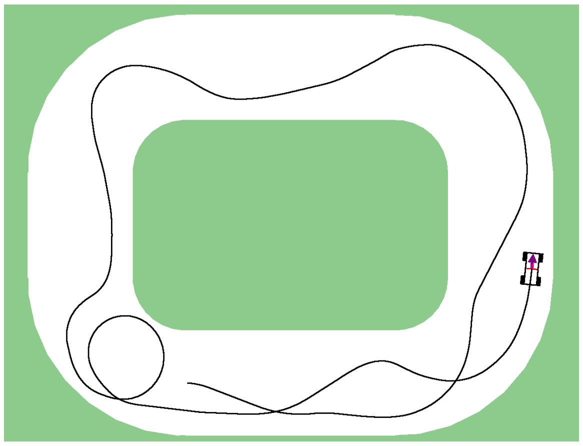


Figure 5.1: A car constrained to stay on the track; see Fig.5.2 for plot of corresponding control inputs.

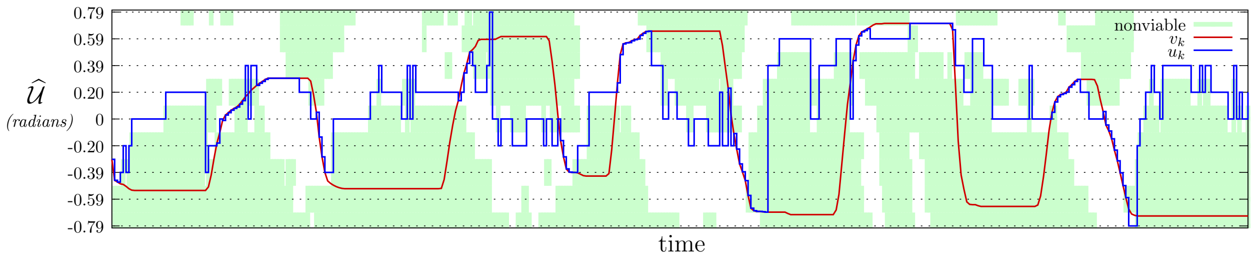


Figure 5.2: Plot of user's desired steering angle (v_k; red, smooth), actual steering applied (u_k; blue, discrete), and the viability of steering angles through time (control actions ($u \in \widehat{\mathcal{U}}$) leading to nonviable states are shaded pale green) for the simulation run shown in Fig.5.1.

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" for two airplanes in the classical "game of two identical vehicles", where one is the evader and the other the pursuer. This capture region has safety implications outside the toy example: if a real plane finds itself within the capture region of another, it means there exists a sequence of control actions for the first airplane, which could be potentially executed due to pilot error, that can lead to collision, regardless of what the second pilot does; for maximum safety the airplanes should thus stay out of each other's capture regions.

In this chapter we model viability globally, in terms of the agent's state x. Such viability models can be obtained from, for example, exhaustive runs of RRT-Blossom (`dead` nodes correspond to nonviable states), or RRT-CT (nodes with a collision tendency of 1 correspond to nonviable states), through appropriate application of one-class or similar classifiers. Safety enforcement could also be achieved using localized, perception-based viability models, like those in the previous chapter, but we consider this to be outside the scope of this proposal and thesis. Although global viability models are only relevant to the static environments on which they were trained, this still admits a large class of applications. For example, such global models might be used in manufacturing and assembly plants to moderate the motion of robots (e.g., "Puma" arms) in what are essentially static, known environments. This could allow greater freedom and more dynamic motions, compared to those derived in a purely kinematic safety framework.

The rest of the chapter proceeds as follows. First, we address the problem of containing the agent state to within the envelope, which may contain some error. Next, the modeling of viability using a Nearest Neighbour classifier is proposed, for reasons of speed and simplicity, and a number of useful adjustments due to this are discussed. This is followed by additional implementation details, and the chapter ends with presentation of some results.

5.1 Containment

Single-step containment

The simplest strategy for ensuring that the state stays within the viable region, $\mathcal{X}_{in}$, is to prevent an exit the instant it is about to occur. This can be achieved by giving the user full reign until such time, and then simply overriding the unsafe control input with a safe one. Within a discrete time framework, the control strategy can be formalized as

$$u_k = \begin{cases} v_k & \text{if } x_{k+1} = F(x_k, v_k) \in \mathcal{X}_{in} \\ N(x_k, v_k) & \text{otherwise} \end{cases} \tag{5.1}$$

where u_k is the control input applied at time step k, v_k is the control input the user requested, $x_k \in \mathcal{X}$ is the system's state, F is a function that embodies the system dynamics in a discrete time

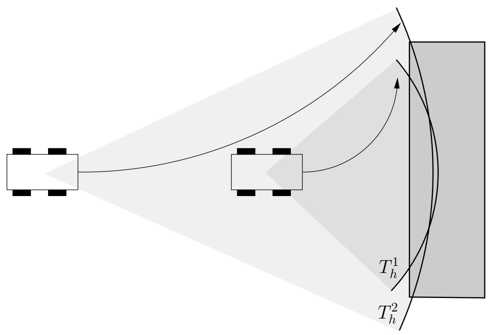


Figure 5.3: A larger time horizon allows milder corrections. Above, the driver on the left has more time and space to maneuver to avoid the obstacle, and so is able to use a gentler turn. In general, using a larger horizon does not preclude the use of the shorter control policy, hence it is guaranteed,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

setting, so that $x_{k+1} = F(x_k, u_k)$, and N is a function that returns an appropriate, safe control input. It should be noted that since $x_k \in \mathcal{X}_{in}$, at least one such safe control is guaranteed to exist. Also, since it is desirable to limit the system's intrusiveness, N should return the safe control input *nearest* to v_k.

Multi-step containment

The above method has one undesirable property: it produces severe corrections. For a user controlling the system, the experience of having control abruptly torn from the their hands is likely to be disorienting and frustrating. It also can have adverse effects if the model of $\mathcal{X}_{in}$ is inexact. This can be mitigated by responding earlier to upcoming envelope breaches, by testing *multiple* subsequent time steps, instead of just one. As Fig. 5.3 illustrates, using a larger *time horizon* (T_h) generally leads to milder corrections. The desired magnitude of the horizon is clearly task- and system-dependent, although there are some inherent and natural upper bounds. For example, it makes little sense for a car to be concerned with a turn in the road that is over an hour away. It is worth noting that, for the most part, the time horizon's magnitude is driven by human response times and mental acuity, rather than by the complexity or particulars of the physical system being controlled.

In the multi-step approach, the user is given free reign until their control input leads to a breach within the time horizon, at which time the input is automatically supplanted by a safe alternative. The latter is chosen by extrapolating state trajectories for all constant-valued[1] control inputs available, up to T_h into the future, and selecting the best candidate. Although control is still usurped from the user, the effect is significantly milder. An additional benefit of the multi-step approach is that the information on temporal breach proximity can be fortuitously used elsewhere, for example in haptic guidance, in mapping control inputs to meaningful feedback levels.

The concept of temporal proximity of a breach figures prominently in our work, so it is helpful to introduce a concise notation for it. Thus, let $T_{eb}(x, u)$ denote the *time to envelope breach* for a control input u, if the system is initially in state x. If u causes a breach within the time horizon, then $0 \leq T_{eb}(x, u) \leq T_h$; if not, it is convenient to let $T_{eb}(x, u) = +\infty$.

Using T_{eb} we can now fully characterize our online algorithm as follows. The method runs in

[1]This assumption embodies the *generalized inertia principle* from viability theory. In our method it serves as the best guess of user's future input.

one of four modes of operation, based on the assessed meta-state of the system:

$$\begin{array}{ll} \text{L1:} & T_{eb}(x_k, v_k) > T_h \\ \text{L2:} & T_{eb}(x_k, v_k) <= T_h, \\ & \text{and } \exists u \in \mathcal{U} : T_{eb}(x_k, u) > T_h \\ \text{L3:} & \forall u \in \mathcal{U} : T_{eb}(x_k, u) <= T_h, \\ & \text{and } x_k \in \mathcal{X}_{in} \\ \text{L4:} & x_k \in \mathcal{X}_{out} \end{array}$$

The corresponding applied (corrected) control input is then given by

$$u_k = \begin{cases} v_k & \text{if L1} \\ B(x_k, v_k) & \text{if L2} \\ \underset{u \in \mathcal{U}_{bf}}{\arg\max}\ T_{eb}(x_k, u) & \text{if L3} \\ \text{N/A} & \text{if L4} \end{cases} \tag{5.2}$$

where $\mathcal{U}_{bf}$ is a set-valued function which describes the set of constant-valued control inputs which are *breach-free* for a given state (i.e., $\mathcal{U}_{bf}(x_k) \subseteq \mathcal{U}$), and B is a function that picks an appropriate, safe control (related to the earlier $N(x_k, v_k)$; discussed later), biased in some way by the user's control input, v_k. It should be noted that whenever we refer to any control or trajectory as "breach-free", we implicitly mean "... within T_h".

In brief, the four modes represent progressive levels of severity of the system meta-state. L1 and L2 constitute normal operation, while L3 and L4 correspond to crisis handling modes. In particular, L1 corresponds to the most benign case, where the user's control input v_k does not breach the envelope within T_h, and hence is applied as is. In L2 the user's input does lead to a breach, but other values exist that do not; an appropriate input is chosen from among these. In L3 all the control inputs lead to a breach. Since $x_k \in \mathcal{X}_{in}$, an achievable breach-free control policy is guaranteed to exist, but in this case it will involve time-varying control inputs. Choosing the control with the largest T_{eb} is likely to maximize the chances that the system will track, at least initially, one of these desired non-constant inputs, although this is not guaranteed. Finally, in L4, the system state is already outside the envelope. No control law is provided for this case as it does not occur with analytic, exact envelopes; it is listed here for completeness, and plays a greater role in further sections.

5.2 Practical approximations

The framework, as outlined above, is difficult to implement, especially in an interactive setting. This section presents the approximations which make this goal achievable.

Discretization of $\mathcal{U}$

A recurring problem throughout the framework is the need to search all of $\mathcal{U}$ for some desired control input, or performing a computation on each of its members. A simple remedy is to discretize $\mathcal{U}$. We thus define $\widehat{\mathcal{U}}$ as the set of controls uniformly sampled from $\mathcal{U}$, and use this subset wherever $\mathcal{U}$ is called for. This then, for example, allows direct computation of the "nearest safe input" function N, which can now be formally defined as

$$N(x_k, v_k) = \underset{u \in \widehat{\mathcal{U}}_{bf}}{\arg\min} |u - v_k| \tag{5.3}$$

where $\widehat{\mathcal{U}}_{bf}(x_k) \subseteq \widehat{\mathcal{U}}$ is the discretized equivalent of $\mathcal{U}_{bf}(x_k)$. One can now also easily establish the system's mode (i.e., $\in \{L1, L2, L3, L4\}$), since T_{eb} has to be computed only for a finite set of control inputs.

The size of $\widehat{\mathcal{U}}$ is chosen to be as small as possible to reduce computational load, but large enough so that under most circumstances it captures at least one breach-free input. For simple systems (e.g., those amenable to bang-bang control) the discretization can be very sparse, since either the minimal or maximal input is frequently breach-free. For complex systems, on the other hand, even very dense discretizations can sometimes fail to produce a suitable candidate, especially if the controllable subset of $\mathcal{U}$ is small and inconveniently distributed. This carries important repercussions, most principally that the L3 policy of setting u_k to the control with the largest T_{eb}, as shown in (5.2), likely becomes ineffective then, and admits breaches. The best one can do in this case is to treat the situation more severely, applying the L4 control law instead, which we discuss in the next section.

Approximate envelopes

Analytic descriptions of the viability envelopes can usually only be obtained for the simplest of systems. For more complex systems the envelope may be approximated through some form of empirical sampling of the state space, and the use of classification methods from machine learning to infer the controllability of the system for arbitrary query states. We have explored the use of both, Support Vector Machines (SVM) and Nearest Neighbour (NN) techniques, and found the latter to be preferable for reasons of speed and simplicity, which make it more amenable to application-specific customization and extension.

In general then, the envelope approximation is captured using a NN classifier

$$NN(x) = \begin{cases} 1 & \text{if } \min\limits_{z \in \widehat{\mathcal{X}}_{in}} |x - z| \leq \min\limits_{z \in \widehat{\mathcal{X}}_{out}} |x - z| \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where $\widehat{\mathcal{X}}_{in}$ and $\widehat{\mathcal{X}}_{out}$ are sets of samples which are known to be inside and outside the envelope, respectively (i.e., viable and nonviable samples, respectively). These sets are obtained in an off-line pre-computation step, as detailed in the following section. Fig. 5.4 shows an example of a part of a viability envelope that was computed for a 2D dynamical system.

By virtue of being approximations, these envelopes will under- and over-approximate at various points bordering the true envelope, leading to false negatives ($NN(x) = 0$ when $x \in \mathcal{X}_{in}$) and false positives ($NN(x) = 1$ when $x \in \mathcal{X}_{out}$). The former is less of a problem than the latter since marking extra regions of state-space as uncontrollable merely results in a more conservative envelope. The key problem is the presence of the false positives, which can deceive the system into unknowingly entering $\mathcal{X}_{out}$. That is, they make it possible to enter the L4 mode.

If confronted with the unavoidability of failure (L4 mode), we choose to minimize a metric of the failure's severity. This is done by selecting the "least detrimental" control input, one which causes the system to spend the least amount of time in $\mathcal{X}_{out}$. Because the time spent in $\mathcal{X}_{out}$ for any input may be arbitrarily large, the search for envelope re-entries needs to be bounded using a suitable criterion.

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 straight-forward 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, thus maintaining system safety.

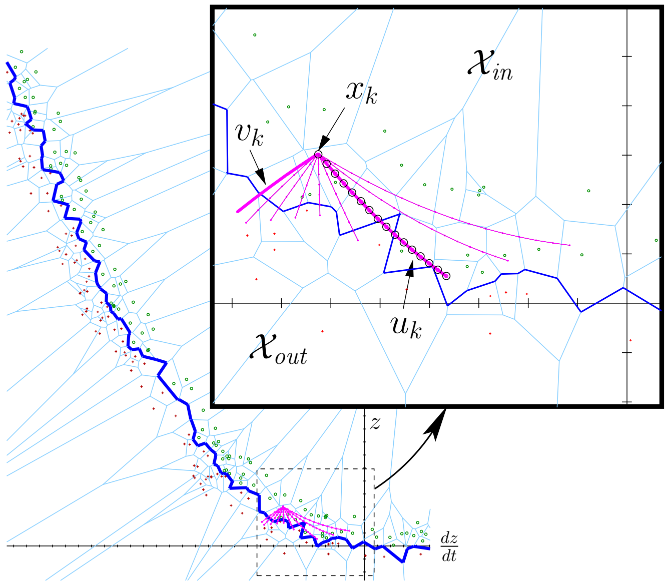


Figure 5.4: A lunar lander's 2D NN envelope; the set of trajectories originating at x_k show the computation of $T_{eb}(x_k, u)\ \forall u \in \widehat{\mathcal{U}}$, with $T_{gr} = 1$. The user's zero-thrust input (leftmost, labeled "v_k") is being overridden by the one shown in bold. Also shown are the band of samples adjoining the envelope, and the resultant Voronoi tessellation.

Finally, we employ a *grace period* when identifying envelope crossings, primarily to combat the error-induced noisy nature of NN envelopes. We define T_{gr}, the grace period, as the maximum amount of time that a trajectory may enter the alternate region without being labeled as a transition. Conversely, a transition is only pronounced if the trajectory excursion into the opposing region lasts longer than T_{gr}. The rationale for this is that, as trajectories τ_1 and τ_2 of Fig. 5.5 suggest, the longer a trajectory stays within the latter region, the more likely it is that the perceived transition did in fact occur, and was not an artifact of the envelope representation. A trajectory such as τ_3 in Fig. 5.5 thus does not qualify as a transition according to this criterion.

5.3 Implementation

Computation of $T_{eb}(x, u)$

We define $T_{eb}(x, u)$ in the discrete case as the time period to the first state that is classified as being in $\mathcal{X}_{out}$. Although measuring T_{eb} in seconds may seem natural, it is more practical to express it as an integer number of fixed time steps Δt, with T_h and T_{gr} measured likewise. In computing T_{eb} there is usually no need to search for a breach past T_h; it is always sufficient to know that $T_{eb} > T_h$ instead of the actual value (and thus our earlier use of $T_{eb} = +\infty$ in this case).

When searching for re-entries, we search up to an additional T_h into the future. If a re-entry

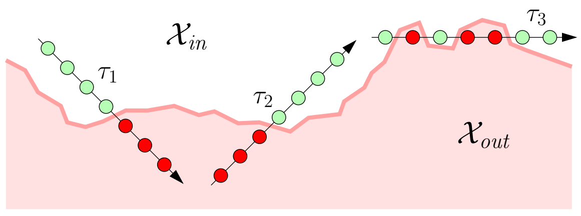


Figure 5.5: Using a grace period to combat envelope (approximation) noise; for $T_{gr} = 2$, τ_1 forms a definite breach, τ_2 a definite re-entry, and τ_3 merely a "brushing" of the envelope (i.e., *not* a transition).

is not found within that span, the next best thing is to select the control input whose state-space trajectory comes the closest. The relative proximity of the various trajectory endpoints can be effectively approximated as the average distance to the k nearest NN samples from $\mathcal{X}_{in}$, with $k = 3$ usually being sufficient; $k = 1$ tends to be unreliable.

Blending function

There are a number of ways to implement $B(x_k, v_k)$ that appears in (5.2). We have used a conservative approach, namely $B(x_k, v_k) = N(x_k, v_k)$. A more flexible and general approach is to implement it as a blending function

$$B_f(x_k, v_k) = \alpha v_k + (1 - \alpha)N(x_k, v_k) \tag{5.5}$$

where

$$\alpha = min\left[\frac{T_{eb}(x_k, v_k) - 1}{T_h},\ 1\right] \tag{5.6}$$

This modulates the strength of the correction based on the immediacy of a breach, and thus allows the user more freedom at longer lead times. The approach gives corrections whose magnitudes vary between those of single-step containment and the $B = N$ case above.

Envelope construction

As mentioned earlier, we employ NN methods to classify query points based upon samples that are known to be viable or unviable. Algorithm 6 describes how the classified sample points are obtained, while (5.4) describes their application in the classifier.

Algorithm 6 Computation of $\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out}$ for the 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($\vec{x}$) $= 1$ 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($\widehat{\mathcal{X}}_{in}, \widehat{\mathcal{X}}_{out}$)
```

The `oracle(`$\vec{x}$`)` is a function that authoritatively answers the question of whether the given state is controllable. It may obtain its knowledge by analytic, empirical, or heuristic means. The reason we do not consult the oracle directly during online simulation is that frequently these are extremely slow. The NN classifier essentially serves to embody the oracle's knowledge in a form that is optimized for query speed, and serves as a universal encoding to which all other representations may be easily translated. If explicit viable and nonviable samples can be obtained from an external source, such as those that could be extracted from exhaustive RRT-Blossom runs, these may be used directly, obviating the above sample generation.

As with most learning methods, it is necessary to scale or normalize the training data prior to use, given that the NN classifier uses an L_2 norm distance metric in state-space. At present we select appropriate scaling factors manually, based on some understanding of the shape of the controllable region. The parameters should be chosen so that the significant features of the envelope surface (i.e., bumps, valleys, ridges, etc.) are of similar magnitude in all the state-space dimensions in

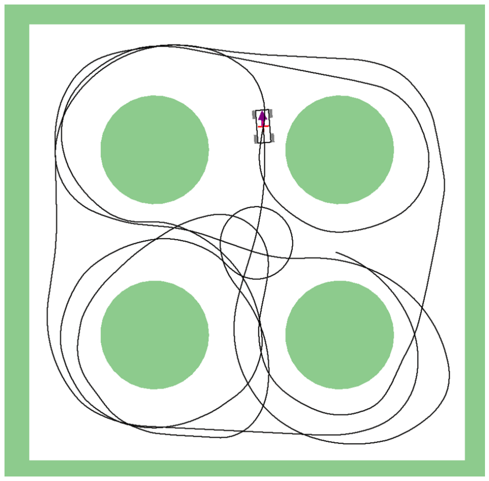


Figure 5.6: The “4 obstacles” terrain for the car.

which they lie, to avoid being trivialized and hidden in the noise or error inherent in the classifier's representation of the surface.

A final measure taken to reduce unnecessary load is to discard redundant samples, ones which do not contribute to the NN decision surface, and consequently ones whose removal does not change it. Although a number of methods exist to do this[P.E68, Cha74, Das91], we employ a simpler technique. We make use of the fact that the samples are uniformly distributed, and compute the average inter-sample distance δ_s. We then discard all samples which are further than $k\,\delta_s$ from the decision surface[2]. The value of k is chosen by trying a number of possibilities, typically $k \in \{5, 10, 20, \dots\}$, and seek the first one that results in a consistent subset, namely one that properly classifies every sample from the original sets. This yields a well-structured band of samples around the decision surface.

5.4 Results

We have successfully applied the viability envelope method to four systems:

1. the lunar lander, as discussed in section 2.2
2. a dynamical model of bicycle balance having a 2D state space $(\theta, d\theta/dt)$, where θ represents the tilt of the bicycle
3. a steerable car restricted to an infinitely long straight road of limited width, having a 2D state space (y, θ) where y represents the distance of the car from one of the curbs, and θ represents the car's orientation with respect to the road
4. a steerable car restricted to a terrain of arbitrary, bounded geometry

The last example is the most complex in terms of having a 3D state-space (x, y, θ) and no easily-modeled analytic solution. For brevity, we restrict our results and discussions to this last example.

Fig. 5.1 shows a trial run for a car on a track using a 3D viability envelope. The user is able to interactively steer the car at will but is prevented by the system from leaving the track. Fig. 5.2 shows how the safety constraints project onto the control input space for this problem. The user

[2]This can be approximated by measuring instead the distance to the nearest NN sample of opposite class.

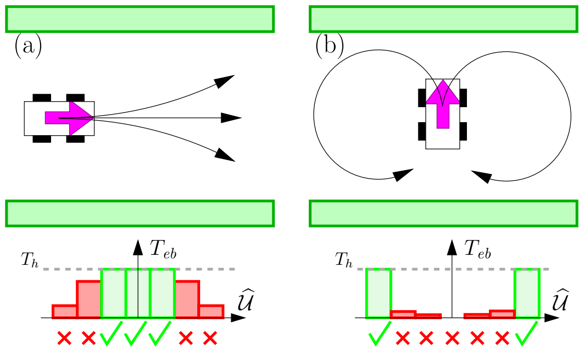


Figure 5.7: T_{eb} behaviour and viability of $\widehat{\mathcal{U}}$ for a fixed-velocity car; the trajectories correspond to breach-free controls, which appear check-marked and green in the T_{eb} vs. $\widehat{\mathcal{U}}$ bar graphs underneath. Case (b) is instructive: although the car's distance from the opposing curb 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.

input v_k consists of a sequence of right and left turns of the steering wheel, as represented by the smooth line on the graph. The nonviable control inputs are given by the shaded areas. The applied control input, u_k is computed as given by Equation 5.2 and is represented by the blue, discrete line (an artifact of the discretization of $\mathcal{U}$). Lastly, Fig. 5.7 shows the set of viable and nonviable steering directions for various states of the car during a simulation.

Our method runs interactively on a 2.4GHz Pentium IV. We target a 30Hz interactive simulation rate to give the user reasonable responsiveness. Table 5.1 lists the various parameters used for the scenarios: number of NN samples[3], discretization of control space, time horizon, and grace period. The latter two are expressed in terms of number of simulation steps (i.e., $\frac{1}{30}s$). It should be also noted that the original number of NN samples for each case is much larger; the reported number is that of the remaining set when redundant samples have been removed by `dump_redundant(`$\widehat{\mathcal{X}}_{in}$`,`$\widehat{\mathcal{X}}_{out}$`)`.

Table 5.1: System parameters per scenario

scenario	# samples	$\|\|\widehat{\mathcal{U}}\|\|$	T_h	T_{gr}
lunar lander	1,218	9	15	1
bike	3,513	9	10	1
car (straight road)	N/A	9	10	0
car (track)	42,117	31	10	1
car (4 obstacles)	176,545	15	10	1

[3]In the case of a car on a straight road, which was the first we experimented with, a hand-generated polygonal envelope was used instead of a NN classifier.

Bibliography

[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`.

[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%E8ve-02-11-21-SL.pdf`.

[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 and Automation*, 2(1):14–23, Mar 1986.

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

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

[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.

[CR00] Stefano Caselli and Monica Reggiani. ERPP: An experience-based randomized path planner. In *IEEE Int. Conf. on Robotics & Automation*, pages 1002–1008, 2000.

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

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

[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.

[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.

[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.

[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.

[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. 2007. To appear in *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.

[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.

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

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

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

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

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