Chapter 14

Sampling-Based Planning Under Differential Constraints

After Chapter 13, it seems that differential constraints arise nearly everywhere. For example, they may arise when wheels roll, aircraft fly, and when the dynamics of virtually any mechanical system is considered. This makes the basic model used for motion planning in Part II invalid for many applications because differential constraints were neglected. Formulation 4.1, for example, was concerned only with obstacles in the C-space.

This chapter incorporates the differential models of Chapter 13 into sampling-based motion planning. The detailed modeling (e.g., Lagrangian mechanics) of Chapter 13 is not important here. This chapter works directly with a given system, expressed as $\dot{x} = f(x, u)$. The focus is limited to *sampling-based* approaches because very little can be done with combinatorial methods if differential constraints exist. However, if there are no obstacles, then powerful analytical techniques may apply. This subject is complementary to motion planning with obstacles and is the focus of Chapter 15.

Section 14.1 provides basic definitions and concepts for motion planning under differential constraints. It is particularly important to explain the distinctions made in literature between nonholonomic planning, kinodynamic planning, and trajectory planning, all of which are cases of planning under differential constraints. Another important point is that obstacles may be somewhat more complicated in phase spaces, which were introduced in Section 13.2. Section 14.2 introduces sampling over the space of action trajectories, which is an essential part of later planning algorithms.

Section 14.3 revisits the incremental sampling and searching framework of Section 5.4 and extends it to handle differential constraints. This leads to several sampling-based planning approaches, which are covered in Section 14.4. Familiar choices such as dynamic programming or the RDTs of Section 5.5 appear once again. The resulting planning methods can be used for a wide variety of problems that involve differential constraints on C-spaces or phase spaces.

Section 14.5 briefly covers feedback motion planning under differential con-

straints. Approximate, optimal plans can be obtained by a simple adaptation of value iteration from Section 8.5.2. Section 14.6 describes decoupled methods, which start with a collision-free path that ignores differential constraints, and then perform refinements to obtain the desired trajectory. Such approaches often lose completeness and optimality, but they offer substantial computational savings in many settings. Section 14.7 briefly surveys numerical techniques for optimizing a trajectory subjected to differential constraints; the techniques can be used to improve solutions computed by planning algorithms.

14.1 Introduction

14.1.1 Problem Formulation

Motion planning under differential constraints can be considered as a variant of classical *two-point boundary value problems* (BVPs) [440]. In that setting, initial and goal states are given, and the task is to compute a path through a state space that connects initial and goal states while satisfying differential constraints. Motion planning involves the additional complication of avoiding obstacles in the state space. Techniques for solving BVPs are unfortunately not well-suited for motion planning because they are not designed for handling obstacle regions. For some methods, adaptation may be possible; however, the obstacle constraints usually cause these classical methods to become inefficient or incomplete. Throughout this chapter, the BVP will refer to motion planning with differential constraints and no obstacles. BVPs that involve more than two points also exist; however, they are not considered in this book.

It is assumed that the differential constraints are expressed in a state transition equation, $\dot{x} = f(x, u)$, on a smooth manifold X, called the *state space*, which may be a C-space $\mathcal{C}$ or a phase space of a C-space. A solution path will not be directly expressed as in Part II but is instead derived from an action trajectory via integration of the state transition equation.

Let the action space U be a bounded subset of $\mathbb{R}^m$. A planning algorithm computes an *action trajectory* $\tilde{u}$, which is a function of the form $\tilde{u} : [0, \infty) \rightarrow U$. The action at a particular time t is expressed as $u(t)$. To be consistent with standard notation for functions, it seems that this should instead be denoted as $\tilde{u}(t)$. This abuse of notation was intentional, to make the connection to the discrete-stage case clearer and to distinguish an action, $u \in U$, from an action trajectory $\tilde{u}$. If the action space is state-dependent, then $u(t)$ must additionally satisfy $u(t) \in U(x(t)) \subseteq U$. For state-dependent models, this will be assumed by default. It will also be assumed that a termination action u_T is used, which makes it possible to specify all action trajectories over $[0, \infty)$ with the understanding that at some time t_F, the termination action is applied.

The connection between the action and state trajectories needs to be formulated. Starting from some initial state $x(0)$ at time $t = 0$, a *state trajectory* is

derived from an action trajectory $\tilde{u}$ as

$$x(t) = x(0) + \int_0^t f(x(t'), u(t'))dt', \tag{14.1}$$

which integrates the state transition equation $\dot{x} = f(x, u)$ from the initial condition $x(0)$. Let $\tilde{x}(x(0), \tilde{u})$ denote the state trajectory over all time, obtained by integrating (14.1). Differentiation of (14.1) leads back to the state transition equation. Recall from Section 13.1.1 that if u is fixed, then the state transition equation defines a vector field. The state transition equation is an alternative expression of (8.14) from Section 8.3, which is the expression for an integral curve of a vector field. The state trajectory is the integral curve in the present context.

The problem of motion planning under differential constraints can be formulated as an extension of the Piano Mover's Problem in Formulation 4.1. The main differences in this extension are 1) the introduction of time, 2) the state or phase space, and 3) the state transition equation. The resulting formulation follows.

Formulation 14.1 (Motion Planning Under Differential Constraints)

1. A *world* $\mathcal{W}$, a *robot* $\mathcal{A}$ (or $\mathcal{A}_1$, ..., $\mathcal{A}_m$ for a linkage), an *obstacle region* $\mathcal{O}$, and a *configuration space* $\mathcal{C}$, which are defined the same as in Formulation 4.1.

2. An unbounded *time interval* $T = [0, \infty)$.

3. A smooth manifold X, called the *state space*, which may be $X = \mathcal{C}$ or it may be a phase space derived from $\mathcal{C}$ if dynamics is considered; see Section 13.2. Let $\kappa : X \rightarrow \mathcal{C}$ denote a function that returns the configuration $q \in \mathcal{C}$ associated with $x \in X$. Hence, $q = \kappa(x)$.

4. An obstacle region X_{obs} is defined for the state space. If $X = \mathcal{C}$, then $X_{obs} = \mathcal{C}_{obs}$. For general phase spaces, X_{obs} is described in detail in Section 14.1.3. The notation $X_{free} = X \setminus X_{obs}$ indicates the states that avoid collision and satisfy any additional global constraints.

5. For each state $x \in X$, a bounded *action space* $U(x) \subseteq \mathbb{R}^m \cup \{u_T\}$, which includes a termination action u_T and m is some fixed integer called the *number of action variables*. Let U denote the union of $U(x)$ over all $x \in X$.

6. A system is specified using a state transition equation $\dot{x} = f(x, u)$, defined for every $x \in X$ and $u \in U(x)$. This could arise from any of the differential models of Chapter 13. If the termination action is applied, it is assumed that $f(x, u_T) = 0$ (and no cost accumulates, if a cost functional is used).

7. A state $x_I \in X_{free}$ is designated as the *initial state*.

8. A set $X_G \subset X_{free}$ is designated as the *goal region*.

9. A complete algorithm must compute an *action trajectory* $\tilde{u} : T \rightarrow U$, for which the state trajectory $\tilde{x}$, resulting from (14.1), satisfies: 1) $x(0) = x_I$, and 2) there exists some $t > 0$ for which $u(t) = u_T$ and $x(t) \in X_G$.

Additional constraints may be placed on $\tilde{u}$, such as continuity or smoothness over time. At the very least, $\tilde{u}$ must be chosen so that the integrand of (14.1) is integrable over time. Let $\mathcal{U}$ denote the set of all *permissible action trajectories* over $T = [0, \infty)$. By default, $\mathcal{U}$ is assumed to include any integrable action trajectory. If desired, continuity and smoothness conditions can be enforced by introducing new phase variables. The method of placing integrators in front of action variables, which was covered in Section 13.2.4, can usually achieve the desired constraints. If optimizing a criterion is additionally important, then the cost functional given by (8.39) can be used. The existence of optimal solutions requires that U is a closed set, in addition to being bounded.

A final time does not need to be stated because of the termination action u_T. As usual, once u_T is applied, cost does not accumulate any further and the state remains fixed. This might seem strange for problems that involve dynamics because momentum should keep the state in motion. Keep in mind that the termination action is a trick to make the formulation work correctly. In many cases, the goal corresponds to a subset of X in which the velocity components are zero. In this case, there is no momentum and hence no problem. If the goal region includes states that have nonzero velocity, then it is true that a physical system may keep moving after u_T has been applied; however, the cost functional will not measure any additional cost. The task is considered to be completed after u_T is applied, and the simulation is essentially halted. If the mechanical system eventually collides due to momentum, then this is the problem of the user who specified a goal state that involves momentum.

The overwhelming majority of solution techniques are sampling-based. This is motivated primarily by the extreme difficultly of planning under differential constraints. The standard Piano Mover's Problem from Formulation 4.1 is a special case of Formulation 14.1 and is already PSPACE-hard [817]. Optimal planning is also NP-hard, even for a point in a 3D polyhedral environment without differential constraints [172]. The only known methods for exact planning under differential constraints in the presence of obstacles are for the double integrator system $\ddot{q} = u$, for $\mathcal{C} = \mathbb{R}$ [747] and $\mathcal{C} = \mathbb{R}^2$ [171].

Section 14.1.2 provides some perspective on motion planning problems under differential constraints that fall under Formulation 14.1, which assumes that the initial state is given and future states are predictable. Section 14.5 briefly addresses the broader problem of feedback motion planning under differential constraints.

14.1.2 Different Kinds of Planning Problems

There are many ways to classify motion planning problems under differential constraints. Some planning approaches rely on particular properties of the system; therefore, it is helpful to characterize these general differences. The different kinds

of problems described here are specializations of Formulation 14.1. In spite of differences based on the kinds of models described below, all of them can be unified under the topic of planning under differential constraints.

One factor that affects the differential model is the way in which the task is decomposed. For example, the task of moving a robot usually requires the consideration of mechanics. Under the classical robotics approach that was shown in Figure 1.19, the motion planning problem is abstracted away from the mechanics of the robot. This enables the motion planning ideas of Part II to be applied. This decomposition is arbitrary. The mechanics of the robot can be considered directly in the planning process. Another possibility is that only part of the constraints may be considered. For example, perhaps only the rolling constraints of a vehicle are considered in the planning process, but dynamics are handled by another planning module. Thus, it is important to remember that the kinds of differential constraints that appear in the planning problem depend not only on the particular mechanical system, but also on how the task is decomposed.

14.1.2.1 Terms from planning literature

Nonholonomic planning The term *nonholonomic planning* was introduced by Laumond [593] to describe the problem of motion planning for wheeled mobile robots (see [595, 633] for overviews). It was informally explained in Section 13.1 that *nonholonomic* refers to differential constraints that cannot be completely integrated. This means they cannot be converted into constraints that involve no derivatives. A more formal definition of *nonholonomic* will be given in Section 15.4. Most planning research has focused on velocity constraints on $\mathcal{C}$, as opposed to a phase space X. This includes most of the models given in Section 13.1, which are specified as nonintegrable velocity constraints on the C-space $\mathcal{C}$. These are often called *kinematic constraints*, to distinguish them from constraints that arise due to dynamics.

In mechanics and control, the term nonholonomic also applies to nonintegrable velocity constraints on a phase space [112, 113]. Therefore, it is perfectly reasonable for the term nonholonomic planning to refer to problems that also involve dynamics. However, in most applications to date, the term nonholonomic planning is applied to problems that have kinematic constraints only. This is motivated primarily by the early consideration of planning for wheeled mobile robots. In this book, it will be assumed that nonholonomic planning refers to planning under nonintegrable velocity constraints on $\mathcal{C}$ or any phase space X.

For the purposes of sampling-based planning, complete integrability is actually not important. In many cases, even if it can be theoretically established that constraints are integrable, it does not mean that performing the integration is practical. Furthermore, even if integration can be performed, each constraint may be implicit and therefore not easily parameterizable. Suppose, for example, that constraints arise from closed kinematic chains. Usually, a parameterization is not available. By differentiating the closure constraint, a velocity constraint is

obtained on $\mathcal{C}$. This can be treated in a sampling-based planner as if it were a nonholonomic constraint, even though it can easily be integrated.

Kinodynamic planning The term *kinodynamic planning* was introduced by Canny, Donald, Reif, and Xavier [290] to refer to motion planning problems for which velocity and acceleration bounds must be satisfied. This means that there are second-order constraints on $\mathcal{C}$. The original work used the double integrator model $\ddot{q} = u$ for $\mathcal{C} = \mathbb{R}^2$ and $\mathcal{C} = \mathbb{R}^3$. A scalar version of this model appeared Example 13.3. More recently, the term has been applied by some authors to virtually any motion planning problem that involves dynamics. Thus, any problem that involves second-order (or higher) differential constraints can be considered as a form of kinodynamic planning. Thus, if x includes velocity variables, then kinodynamic planning includes any system, $\dot{x} = f(x, u)$.

Note that kinodynamic planning is not necessarily a form of nonholonomic planning; in most cases considered so far, it is not. A problem may even involve both nonholonomic and kinodynamic planning. This requires the differential constraints to be both nonintegrable and at least second-order. This situation often results from constrained Lagrangian analysis, covered in Section 13.4.3. The car with dynamics which was given Section 13.3.3 is both kinodynamic and nonholonomic.

Trajectory planning The term *trajectory planning* has been used for decades in robotics to refer mainly to the problem of determining both a path and velocity function for a robot arm (e.g., PUMA 560). This corresponds to finding a path in the phase space X in which $x \in X$ is defined as $x = (q, \dot{q})$. Most often the problem is solved using the refinement approach mentioned in Section 1.4 by first computing a path through $\mathcal{C}_{free}$. For each configuration q along the path, a velocity $\dot{q}$ must be computed that satisfies the differential constraints. An inverse control problem may also exist, which involves computing for each t, the action $u(t)$ that results in the desired $\dot{q}(t)$. The refinement approach is often referred to as *time scaling* of a path through $\mathcal{C}$ [456]. In recent times, trajectory planning seems synonymous with kinodynamic planning, assuming that the constraints are second-order (x includes only configuration and velocity variables). One distinction is that trajectory planning still perhaps bears the historical connotations of an approach that first plans a path through $\mathcal{C}_{free}$.

14.1.2.2 Terms from control theory

A significant amount of terminology that is appropriate for planning has been developed in the control theory community. In some cases, there are even conflicts with planning terminology. For example, the term *motion planning* has been used to refer to nonholonomic planning in the absence of obstacles [156, 727]. This can be considered as a kind of BVP. In some cases, this form of planning is referred to as the *steering problem* (see [596, 725]) and will be covered in Section 15.5. The

term *motion planning* is reserved in this book for problems that involve obstacle avoidance and possibly other constraints.

Open-loop control laws Differential models, such as any of those from Chapter 13, are usually referred to as *control systems* or just *systems*, a term that we have used already. These are divided into *linear* and *nonlinear* systems, as described in Sections 13.2.2 and 13.2.3, respectively. Formulation 14.1 can be considered in control terminology as the design of an *open-loop control law* for the system (subjected to nonconvex constraints on the state space). The *open-loop* part indicates that no feedback is used. Only the action trajectory needs to be specified over time (the feedback case is called *closed-loop*; recall Section 8.1). Once the initial state is given, the state trajectory can be inferred from the action trajectory. It may also be qualified as a *feasible* open-loop control law, to indicate that it satisfies all constraints but is not necessarily optimal. It is then interesting to consider designing an *optimal* open-loop control law. This is extremely challenging, even for problems that appear to be very simple. Elegant solutions exist for some restricted cases, including linear systems and some wheeled vehicle models, but in the absence of obstacles. These are covered in Chapter 15.

Drift The term *drift* arose in Section 13.2.1 and implies that from some states it is impossible to instantaneously stop. This difficulty arises in mechanical systems due to momentum. Infinite deceleration, and therefore infinite energy, would be required to remove all kinetic energy from a mechanical system in an instant of time. Kinodynamic and trajectory planning generally involve drift. Nonholonomic planning problems may be *driftless* if only velocity constraints exist on the C-space; the models of Section 13.1.2 are driftless. From a planning perspective, systems with drift are usually more challenging than driftless systems.

Underactuation Action variables, the components of u, are often referred to as *actuators*, and a system is called *underactuated* if the number of actuators is strictly less than the dimension of $\mathcal{C}$. In other words, there are less independent action variables than the degrees of freedom of the mechanical system. Underactuated nonlinear systems are typically nonholonomic. Therefore, a substantial amount of nonholonomic system theory and planning for nonholonomic systems involves applications to underactuated systems. As an example of an underactuated system, consider a free-floating spacecraft in $\mathbb{R}^3$ that has three thrusters. The amount of force applied by each thruster can be declared as an action variable; however, the system is underactuated because there are only three actuators, and the dimension of $\mathcal{C}$ is six. Other examples appeared Section 13.1.2. If the system is not underactuated, it is called *fully actuated*, which means that the number of actuators is equal to the dimension of $\mathcal{C}$. Kinodynamic planning has mostly addressed fully actuated systems.

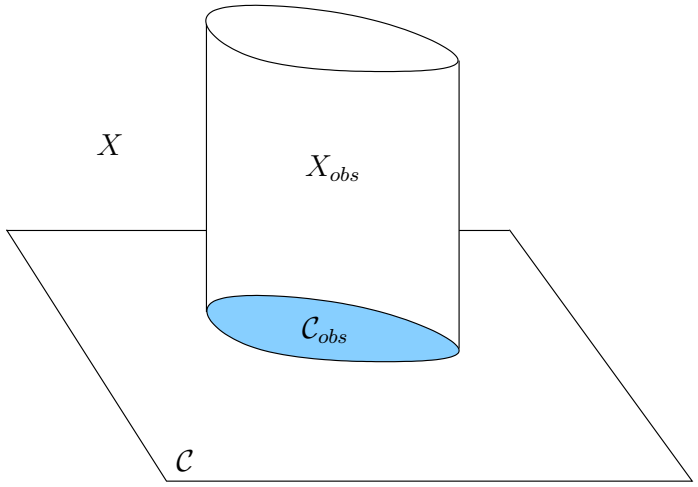


Figure 14.1: An obstacle region $\mathcal{C}_{obs} \subset \mathcal{C}$ generates a cylindrical obstacle region $X_{obs} \subset X$ with respect to the phase variables.

Symmetric systems Finally, one property of systems that is important in some planning algorithms is *symmetry*.[1] A system $\dot{x} = f(x, u)$ is symmetric if the following condition holds. If there exists an action trajectory that brings the system from some x_I to some x_G, then there exists another action trajectory that brings the system from x_G to x_I by visiting the same points in X, but in reverse time. At each point along the path, this means that the velocity can be negated by a different choice of action. Thus, it is possible for a symmetric system to reverse any motions. This is usually not possible for systems with drift. An example of a symmetric system is the differential drive of Section 13.1.2. For the simple car, the Reeds-Shepp version is symmetric, but the Dubins version is not because the car cannot travel in reverse.

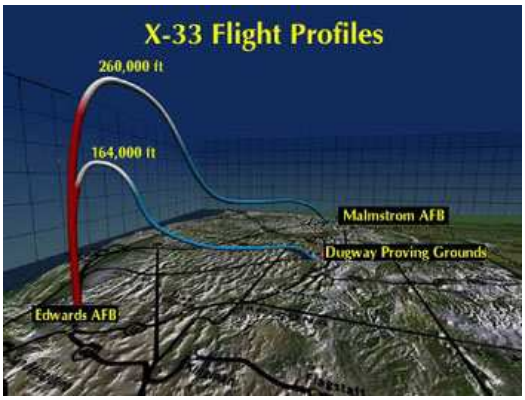
14.1.3 Obstacles in the Phase Space

In Formulation 14.1, the specification of the obstacle region in Item 4 was intentionally left ambiguous. Now it will be specified in more detail. If $X = \mathcal{C}$, then $X_{obs} = \mathcal{C}_{obs}$, which was defined in (4.34) for a rigid robot and in (4.36) for a robot with multiple links. The more interesting case occurs if X is a phase space that includes velocity variables in addition to configuration information.

Any state for which its associated configuration lies in $\mathcal{C}_{obs}$ must also be a member of X_{obs}. The velocity is irrelevant if a collision occurs in the world $\mathcal{W}$. In most cases that involve a phase space, the obstacle region X_{obs} is therefore defined

[1]Sometimes in control theory, the term symmetry applies to Lie groups. This is a different concept and means that the system is invariant with respect to transformations in a group such as $SE(3)$. For example, the dynamics of a car should not depend on the direction in which the car is pointing.

NASA/Lockheed Martin X-33



Re-entry trajectory

Figure 14.2: In the NASA/Lockheed Martin X-33 re-entry problem, there are complicated constraints on the phase variables, which avoid states that cause the craft to overheat or vibrate uncontrollably. (Courtesy of NASA)

as

$$X_{obs} = \{x \in X \mid \kappa(x) \in \mathcal{C}_{obs}\}, \tag{14.2}$$

in which $\kappa(x)$ is the configuration associated with the state $x \in X$. If the first n variables of X are configuration parameters, then X_{obs} has the cylindrical structure shown in Figure 14.1 with respect to the other variables. If κ is a complicated mapping, as opposed to simply selecting the configuration coordinates, then the structure might not appear cylindrical. In these cases, (14.2) still indicates the correct obstacle region in X.

14.1.3.1 Additional constraints on phase variables

In many applications, additional constraints may exist on the phase variables. These are called *phase constraints* and are generally of the form $h_i(x) \leq 0$. For example, a car or hovercraft may have a maximum speed for safety reasons. Therefore, simple bounds on the velocity variables will exist. For example, it might be specified that $\|\dot{q}\| \leq \dot{q}_{max}$ for some constant $\dot{q}_{max} \in (0, \infty)$. Such simple bounds are often incorporated directly into the definition of X by placing limits on the velocity variables.

In other cases, however, constraints on velocity may be quite complicated. For example, the problem of computing the re-entry trajectory of the NASA/Lockheed Martin X-33 reusable spacecraft[2] (see Figure 14.2) requires remaining within a complicated, narrow region in the phase space. Even though there are no hard obstacles in the traditional sense, many bad things can happen by entering the wrong part of the phase space. For example, the craft may overheat or vibrate uncontrollably [160, 201, 662]. For a simpler example, imagine constraints on X

[2]This project was canceled in 2001, but similar crafts have been under development.

to ensure that an SUV or a double-decker tour bus (as often seen in London, for example) will not tumble sideways while turning.

The additional constraints can be expressed implicitly as $h_i(x) \leq 0$. As part of determining whether some state x lies in X_{free} or X_{obs}, it must be substituted into each constraint to determine whether it is satisfied. If a state lies in X_{free}, it will generally be called *violation-free*, which implies that it is both collision-free and does not violate any additional phase constraints.

14.1.3.2 The region of inevitable collision

One of the most challenging aspects of planning can be visualized in terms of the *region of inevitable collision*, denoted by X_{ric}. This is the set of states from which entry into X_{obs} will eventually occur, regardless of any actions that are applied. As a simple example, imagine that a robotic vehicle is traveling 100 km/hr toward a large wall and is only 2 meters away. Clearly the robot is doomed. Due to momentum, collision will occur regardless of any efforts to stop or turn the vehicle. At low enough speeds, X_{ric} and X_{obs} are approximately the same; however, X_{ric} grows dramatically as the speed increases.

Let $\mathcal{U}_\infty$ denote the set of all trajectories $\tilde{u} : [0, \infty) \rightarrow U$ for which the termination action u_T is *never* applied (we do not want inevitable collision to be avoided by simply applying u_T). The *region of inevitable collision* is defined as

$$X_{ric} = \{x(0) \in X \mid \text{ for any } \tilde{u} \in \mathcal{U}_\infty \, , \, \exists t > 0 \text{ such that } x(t) \in X_{obs}\}, \qquad (14.3)$$

in which $x(t)$ is the state at time t obtained by applying (14.1) from $x(0)$. This does not include cases in which motions are eventually blocked, but it is possible to bring the system to a state with zero velocity. Suppose that the Dubins car from Section 13.1.2 is used and the car is unable to back its way out of a dead-end alley. In this case, it can avoid collision by stopping and remaining motionless. If it continues to move, it will eventually have no choice but to collide. This case appears more like being trapped and technically does not fit under the definition of X_{ric}. For driftless systems, $X_{ric} = X_{obs}$.

Example 14.1 (Region of Inevitable Collision) Figure 14.3 shows a simple illustration of X_{ric}. Suppose that $\mathcal{W} = \mathbb{R}$, and the robot is a particle (or point mass) that moves according to the double integrator model $\ddot{q} = u$ (for mass, assume $m = 1$). For simplicity, suppose that u represents a force that must be chosen from $U = [-1, 1]$. The C-space is $\mathcal{C} = \mathbb{R}$, the phase space is $X = \mathbb{R}^2$, and a phase (or state) is expressed as $x = (q, \dot{q})$. Suppose that there are two obstacles in $\mathcal{C}$: a point and an interval. These are shown in Figure 14.3 along the q-axis. In the cylinder above them, X_{obs} appears. In the slice at $\dot{q} = 0$, $X_{ric} = X_{obs} = \mathcal{C}_{obs}$. As $\dot{q}$ increases, X_{ric} becomes larger, even though X_{obs} remains fixed. Note that X_{ric} only grows toward the left because $\dot{q} > 0$ indicates a positive velocity, which causes momentum in the positive q direction. As this momentum increases, the distance required to stop increases quadratically. From a speed of $\dot{q} = v$, the minimum

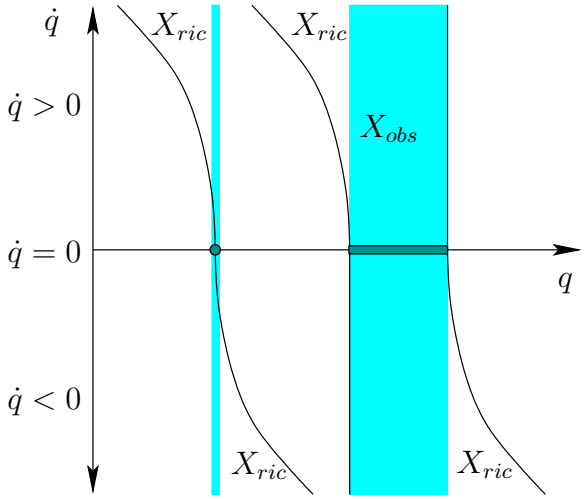


Figure 14.3: The region of inevitable collision grows quadratically with the speed.

distance required to stop is $v^2/2$, which can be calculated by applying the action $u = -1$ and integrating $\ddot{q} = u$ twice. If $\dot{q} > 0$ and q is to the right of an obstacle, then it will safely avoid the obstacle, regardless of its speed. If $\dot{q} < 0$, then X_{ric} extends to the right instead of the left. Again, this is due to the required stopping distance. ■

In higher dimensions and for more general systems, the problem becomes substantially more complicated. For example, in $\mathbb{R}^2$ the robot can swerve to avoid small obstacles. In general, the particular direction of motion becomes important. Also, the topology of X_{ric} may be quite different from that of X_{obs}. Imagine that a small airplane flies into a cave that consists of a complicated network of corridors. Once the plane enters the cave, there may be no possible actions that can avoid collision. The entire part of the state space that corresponds to the plane in the cave would be included in X_{ric}. Furthermore, even parts of the state space from which the plane cannot avoid entering the cave must be included.

In sampling-based planning under differential constraints, X_{ric} is not computed because it is too complicated.[3] It is not even known how to make a "collision detector" for X_{ric}. By working instead with X_{obs}, challenges arise due to momentum. There may be large parts of the state space that are never worth exploring because they lie in X_{ric}. Unfortunately, there is no practical way at present to accurately determine whether states lie in X_{ric}. As the momentum and amount of clutter increase, this becomes increasingly problematic.

[3]It may, however, be possible to compute crude approximations of X_{ric} and use them in planning.

14.2 Reachability and Completeness

This section provides preliminary concepts for sampling-based planning algorithms. In Chapter 5, sampling over $\mathcal{C}$ was of fundamental importance. The most important consideration was that a sequence of samples should be *dense* so that samples get arbitrarily close to any point in $\mathcal{C}_{free}$. Planning under differential constraints is complicated by the specification of solutions by an action trajectory instead of a path through X_{free}. For sampling-based algorithms to be resolution complete, sampling and searching performed on the space of action trajectories must somehow lead to a dense set in X_{free}.

14.2.1 Reachable Sets

For the algorithms in Chapter 5, resolution completeness and probabilistic completeness rely on having a sampling sequence that is dense on $\mathcal{C}$. In the present setting, this would require dense sampling on X. Differential constraints, however, substantially complicate the sampling process. It is generally not reasonable to prescribe precise samples in X that must be reached because reaching them may be impossible or require solving a BVP. Since paths in X are obtained indirectly via action trajectories, completeness analysis begins with considering which points can be reached by integrating action trajectories.

14.2.1.1 Reachable set

Assume temporarily that there are no obstacles: $X_{free} = X$. Let $\mathcal{U}$ be the set of all permissible action trajectories on the time interval $[0, \infty)$. From each $\tilde{u} \in \mathcal{U}$, a state trajectory $\tilde{x}(x_0, \tilde{u})$ is defined using (14.1). Which states in X are visited by these trajectories? It may be possible that all of X is visited, but in general some states may not be reachable due to differential constraints.

Let $R(x_0, \mathcal{U}) \subseteq X$ denote the *reachable set* from x_0, which is the set of all states that are visited by any trajectories that start at x_0 and are obtained from some $\tilde{u} \in \mathcal{U}$ by integration. This can be expressed formally as

$$R(x_0, \mathcal{U}) = \{x_1 \in X \mid \exists \tilde{u} \in \mathcal{U} \text{ and } \exists t \in [0, \infty) \text{ such that } x(t) = x_1\}, \qquad (14.4)$$

in which $x(t)$ is given by (14.1) and requires that $x(0) = x_0$.

The following example illustrates some simple cases.

Example 14.2 (Reachable Sets for Simple Inequality Constraints) Suppose that $X = \mathcal{C} = \mathbb{R}^2$, and recall some of the simple constraints from Section 13.1.1. Let a point in $\mathbb{R}^2$ be denoted as $q = (x, y)$. Let the state transition equation be $\dot{x} = u_1$ and $\dot{y} = u_2$, in which $(u_1, u_2) \in U = \mathbb{R}^2$.

Several constraints will now be imposed on U, to define different possible action spaces. Suppose it is required that $u_1 > 0$ (this was $\dot{x} > 0$ in Section 13.1.1). The reachable set $R(q_0, \mathcal{U})$ from any $q_0 = (x_0, y_0) \in \mathbb{R}^2$ is an open half-plane that is

defined by the set of all points to the right of the vertical line $x = x_0$. In the case of $u_1 \leq 0$, then $R(q_0, \mathcal{U})$ is a closed half-plane to the left of the same vertical line. If U is defined as the set of all $(u_1, u_2) \in \mathbb{R}^2$ such that $u_1 > 0$ and $u_2 > 0$, then the reachable set from any point is a quadrant.

For the constraint $au_1 + bu_2 = 0$, the reachable set from any point is a line in $\mathbb{R}^2$ with normal vector (a, b). The location of the line depends on the particular q_0. Thus, a family of parallel lines is obtained by considering reachable states from different initial states. This is an example of a *foliation* in differential geometry, and the lines are called *leaves* [872].

In the case of $u_1^2 + u_2^2 \leq 1$, the reachable set from any (x_0, y_0) is $\mathbb{R}^2$. Thus, any state can reach any other state. ■

So far the obstacle region has not been considered. Let $\mathcal{U}_{free} \subseteq \mathcal{U}$ denote the set of all action trajectories that produce state trajectories that map into X_{free}. In other words, $\mathcal{U}_{free}$ is obtained by removing from $\mathcal{U}$ all action trajectories that cause entry into X_{obs} for some $t > 0$. The reachable set that takes the obstacle region into account is denoted $R(x_0, \mathcal{U}_{free})$, which replaces $\mathcal{U}$ by $\mathcal{U}_{free}$ in (14.4). This assumes that for the trajectories in $\mathcal{U}_{free}$, the termination action can be applied to avoid inevitable collisions due to momentum. A smaller reachable set could have been defined that eliminates trajectories for which collision inevitably occurs without applying u_T.

The completeness of an algorithm can be expressed in terms of reachable sets. For any given pair $x_I, x_G \in X_{free}$, a complete algorithm must report a solution action trajectory if $x_G \in R(x_I, \mathcal{U}_{free})$, or report failure otherwise. Completeness is too difficult to achieve, except for very limited cases [171, 747]; therefore, sampling-based notions of completeness are more valuable.

14.2.1.2 Time-limited reachable set

Consider the set of all states that can be reached up to some fixed time limit. Let the *time-limited reachable set* $R(x_0, \mathcal{U}, t)$ be the subset of $R(x_0, \mathcal{U})$ that is reached up to and including time t. Formally, this is

$$R(x_0, \mathcal{U}, t) = \{x_1 \in X \mid \exists \tilde{u} \in \mathcal{U} \text{ and } \exists t' \in [0, t] \text{ such that } x(t') = x_1\}. \quad (14.5)$$

For the last case in Example 14.2, the time-limited reachable sets are closed discs of radius t centered at (x_0, y_0). A version of (14.5) that takes the obstacle region into account can be defined as $R(x_0, \mathcal{U}_{free}, t)$.

Imagine an animation of $R(x_0, \mathcal{U}, t)$ that starts at $t = 0$ and gradually increases t. The boundary of $R(x_0, \mathcal{U}, t)$ can be imagined as a propagating wavefront that begins at x_0. It eventually reaches the boundary of $R(x_0, \mathcal{U})$ (assuming it has a boundary; it does not if $R(x_0, \mathcal{U}) = X$). The boundary of $R(x_0, \mathcal{U}, t)$ can actually be interpreted as a level set of the optimal cost-to-come from x_0 for a cost functional that measures the elapsed time. The boundary is also a kind of forward projection, as considered for discrete spaces in Section 10.1.2. In that context, possible future

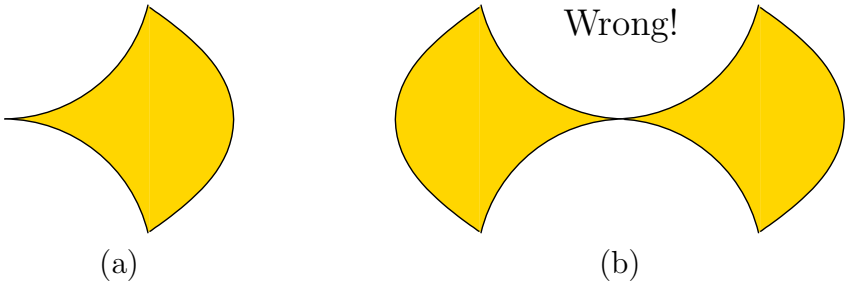


Figure 14.4: (a) The time-limited reachable set for the Dubins car facing to the right; (b) this is *not* the time-limited reachable set for the Reeds-Shepp car!

states due to nature were specified in the forward projection. In the current setting, possible future states are determined by the unspecified actions of the robot. Rather than looking k stages ahead, the time-limited reachable set looks for duration t into the future. In the present context there is essentially a continuum of stages.

Example 14.3 (Reachable Sets for Simple Cars) Nice illustrations of reachable sets can be obtained from the simple car models from Section 13.1.2. Suppose that $X = \mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$ and $X_{obs} = \emptyset$.

Recall that the Dubins car can only drive forward. From an arbitrary configuration, the time-limited reachable set appears as shown in Figure 14.4a. The time limit t is small enough so that the car cannot rotate by more than $\pi/2$. Note that Figure 14.4a shows a 2D projection of the reachable set that gives translation only. The true reachable set is a 3D region in $\mathcal{C}$. If $t > 2\pi$, then the car will be able to drive in a circle. For any q, consider the limiting case as t approaches infinity, which results in $R(q, \mathcal{U})$. Imagine a car driving without reverse on an infinitely large, flat surface. It is possible to reach any desired configuration by driving along a circle, driving straight for a while, and then driving along a circle again. Therefore, $R(q, \mathcal{U}) = \mathcal{C}$ for any $q \in \mathcal{C}$. The lack of a reverse gear means that some extra maneuvering space may be needed to reach some configurations.

Now consider the Reeds-Shepp car, which is allowed to travel in reverse. Any time-limited reachable set for this car must include all points from the corresponding reachable set for the Dubins car because new actions have been added to U but none have been removed. It is tempting to assert that the time-limited reachable set appears as in Figure 14.4b; however, this is wrong. In an arbitrarily small amount of time (or space) a car with reverse can be wiggled sideways. This is achieved in practice by familiar parallel-parking maneuvers. It turns out in this case that $R(q, \mathcal{U}, t)$ always contains an open set around q, which means that it grows in all directions (see Section 15.3.2). The property is formally referred to as small-time controllability and is covered in Section 15.4. ■

14.2.1.3 Backward reachable sets

The reachability definitions have a nice symmetry with respect to time. Rather than describing all points reachable from some $x \in X$, it is just as easy to describe all points from which some $x \in X$ can be reached. This is similar to the alternative between forward and backward projections in Section 10.1.2.

Let the *backward reachable set* be defined as

$$B(x_f, \mathcal{U}) = \{x_0 \in X \mid \exists \tilde{u} \in \mathcal{U} \text{ and } \exists t \in [0, \infty) \text{ such that } x(t) = x_f\}, \quad (14.6)$$

in which $x(t)$ is given by (14.1) and requires that $x(0) = x_0$. Note the intentional similarity to (14.4). The *time-limited backward reachable set* is defined as

$$B(x_f, \mathcal{U}, t) = \{x_0 \in X \mid \exists \tilde{u} \in \mathcal{U} \text{ and } \exists t' \in [0, t] \text{ such that } x(t') = x_f\}, \quad (14.7)$$

which once again requires that $x(0) = x_0$ in (14.1). Completeness can even be defined in terms of backward reachable sets by defining a backward-time counterpart to $\mathcal{U}$.

At this point, there appear to be close parallels between forward, backward, and bidirectional searches from Chapter 2. The same possibilities exist in sampling-based planning under differential constraints. The forward and backward reachable sets indicate the possible states that can be reached under such schemes. The algorithms explore subsets of these reachable sets.

14.2.2 The Discrete-Time Model

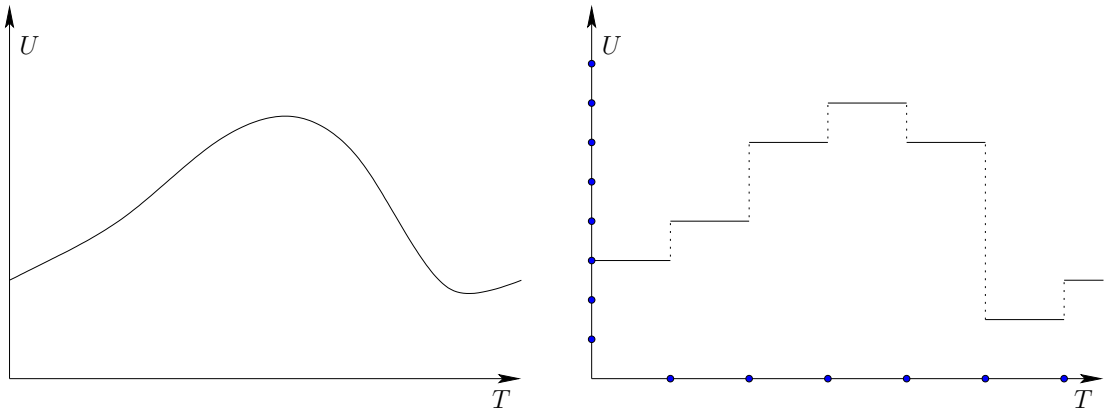
This section introduces a simple and effective way to sample the space of action trajectories. Section 14.2.3 covers the more general case. Under differential constraints, sampling-based motion planning algorithms all work by sampling the space of action trajectories. This results in a reduced set of possible action trajectories. To ensure some form of completeness, a motion planning algorithm should carefully construct and refine the sample set. As in Chapter 5, the qualities of a sample set can be expressed in terms of dispersion and denseness. The main difference in the current setting is that the algorithms here work with a sample sequence over $\mathcal{U}$, as opposed to over $\mathcal{C}$ as in Chapter 5. This is required because solution paths can no longer be expressed directly on $\mathcal{C}$ (or X).

The *discrete-time model* is depicted in Figure 14.5 and is characterized by three aspects:

1. Time T is partitioned into intervals of length Δt. This enables stages to be assigned, in which stage k indicates that $(k-1)\Delta t$ units of time have elapsed.

2. A finite subset U_d of the action space U is chosen. If U is already finite, then this selection may be $U_d = U$.

3. The action $u(t) \in U_d$ must remain constant over each time interval.



A trajectory in $\mathcal{U}$ A trajectory in $\mathcal{U}_d$

Figure 14.5: The discrete-time model results in $\mathcal{U}_d \subset \mathcal{U}$, which is obtained by partitioning time into regular intervals and applying a constant action over each interval. The action is chosen from a finite subset U_d of U.

The first two discretize time and the action spaces. The third condition is needed to relate the time discretization to the space of action trajectories. Let $\mathcal{U}_d$ denote the set of all action trajectories allowed under a given time discretization. Note that $\mathcal{U}_d$ completely specifies the discrete-time model.

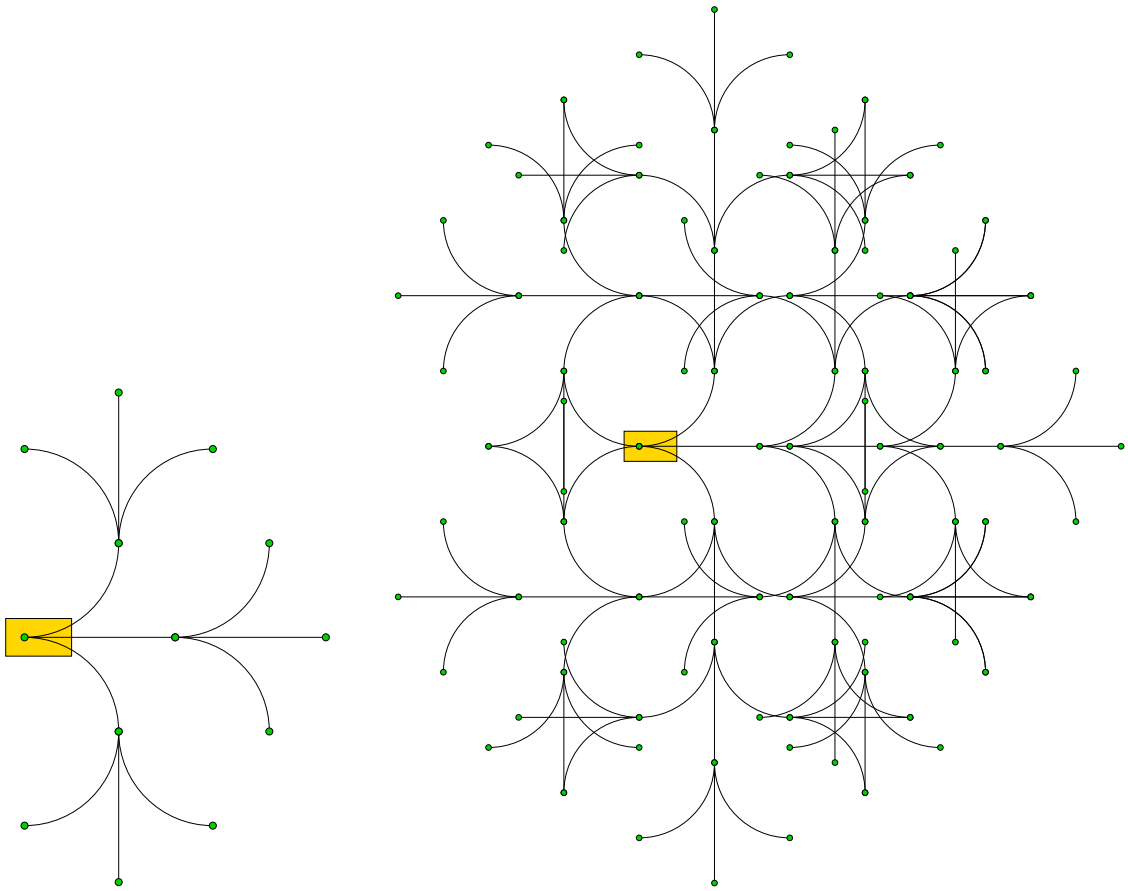
For some problems, U may already be finite. Imagine, for example, a model of firing one of several thrusters (turn them *on* or *off*) on a free-floating spacecraft. In this case no discretization of U is necessary. In the more general case, U may be a continuous set. The sampling methods of Section 5.2 can be applied to determine a finite subset $U_d \subseteq U$.

Any action trajectory in $\mathcal{U}_d$ can be conveniently expressed as an *action sequence* $(u_1, u_2, \ldots, u_k)$, in which each $u_i \in U_d$ gives the action to apply from time $(i-1)\Delta t$ to time $i\Delta t$. After stage k, it is assumed that the termination action is applied.

14.2.2.1 Reachability graph

After time discretization has been performed, the reachable set can be adapted to $\mathcal{U}_d$ to obtain $R(x_0, \mathcal{U}_d)$. An interesting question is: What is the effect of sampling on the reachable set? In other words, how do $R(x_0, \mathcal{U})$ and $R(x_0, \mathcal{U}_d)$ differ? This can be addressed by defining a reachability graph, which will be revealed incrementally by a planning algorithm.

Let $T_r(x_0, \mathcal{U}_d)$ denote a *reachability tree*, which encodes the set of all trajectories from x_0 that can be obtained by applying trajectories in $\mathcal{U}_d$. Each vertex of $T_r(x_0, \mathcal{U}_d)$ is a reachable state, $x \in R(x_0, \mathcal{U}_d)$. Each edge of $T_r(x_0, \mathcal{U}_d)$ is directed; its source represents a starting state, and its destination represents the state obtained by applying a constant action $u \in U_d$ over time Δt. Each edge e represents an action trajectory segment, $e : [0, \Delta t] \rightarrow U$. This can be transformed into a state trajectory, $\tilde{x}_e$, via integration using (14.1), from 0 to Δt of $f(x, u)$ from the source



Two stages Four stages

Figure 14.6: A reachability tree for the Dubins car with three actions. The kth stage produces 3^k new vertices.

state of e.

Thus, in terms of $\tilde{x}_e$, T_r can be considered as a topological graph in X (T_r will be used as an abbreviation of $T_r(x_0, \mathcal{U}_d)$). The *swath* $S(T_r)$ of T_r is

$$S(T_r) = \bigcup_{e \in E} \bigcup_{t \in [0, \Delta t]} x_e(t), \tag{14.8}$$

in which $x_e(t)$ denotes the state obtained at time t from edge e. (Recall topological graphs from Example 4.6 and the swath from Section 5.5.1.)

Example 14.4 (Reachability Tree for the Dubins Car) Several stages of the reachability tree for the Dubins car are shown in Figure 14.6. Suppose that there are three actions (straight, right-turn, left-turn), and Δt is chosen so that if the right-turn or left-turn action is applied, the car travels enough to rotate by $\pi/2$. After the second stage, there are nine leaves in the tree, as shown in Figure 14.6a.

Each stage produces 3^k new leaves. In Figure 14.6b, 81 new leaves are added in stage $k = 4$, which yields a total of $81 + 27 + 9 + 3 + 1$ vertices. In many cases, the same state is reachable by different action sequences. The swath after the first four stages is the set of all points visited so far. This is a subset of $\mathcal{C}$ that is the union of all vertices and all points traced out by $\tilde{x}_e$ for each $e \in E$. ■

From Example 14.4 it can be seen that it is sometimes possible to arrive at the same state using two or more alternative action trajectories. Since each action trajectory can be expressed as an action sequence, the familiar issue arises from classical AI search of detecting whether the same state has been reached from different action sequences. For some systems, the reachability problem can be dramatically simplified by exploiting this information. If the same state is reached from multiple action sequences, then only one vertex needs to be represented.

This yields a directed *reachability graph* $\mathcal{G}_r(x_0, \mathcal{U}_d)$, which is obtained from $T_r(x_0, \mathcal{U}_d)$ by merging its duplicate states. If every action sequence arrives at a unique state, then the reachability graph reduces to the reachability tree. However, if multiple action sequences arrive at the same state, this is represented as a single vertex $\mathcal{G}_r$. From this point onward, the reachability graph will be primarily used. As for a reachability tree, a reachability graph can be interpreted as a topological graph in X, and its swath $S(\mathcal{G}_r)$ is defined by adapting (14.8).

The simplest case of arriving at the same state was observed in Example 2.1. The discrete grid in the plane can be modeled using the terminology of Chapter 13 as a system of the form $\dot{x} = u_1$ and $\dot{y} = u_2$ for a state space $X = \mathbb{R}^2$. The discretized set U_d of actions is $\{(1, 0), (0, 1), (-1, 0), (0, -1)\}$. Let $\Delta t = 1$. In this case, the reachability graph becomes the familiar 2D grid. If $(0, 0)$ is the initial state, then the grid vertices consist of all states in which both coordinates are integers.

Through careless discretization of an arbitrary system, such a nice grid usually does not arise. However, in many cases a discretization can be carefully chosen so that the states become trapped on a grid or lattice. This has some advantages in sampling-based planning. Section 14.4.1 covers a method that exploits such structure for the system $\ddot{q} = u$. It can even be extended to more general systems, provided that the system can be expressed as $\ddot{q} = g(q, \dot{q}, u)$ and it is not underactuated. It was shown recently that by a clever choice of discretization, a very large class of nonholonomic systems[4] can also be forced onto a lattice [762]. This is usually difficult to achieve, and under most discretizations the vertices of the reachability graph are dense in the reachable set.

It is also possible to define backward versions of the reachability tree and reachability graph, in the same way that backward reachable sets were obtained. These indicate initial states and action sequences that will reach a given goal state and are no more difficult to define or compute than their forward counterparts.

[4]The class is all driftless, nilpotent systems. The term nilpotent will be defined in Section 15.5.

They might appear more difficult, but keep in mind that the initial states are not fixed; thus, no BVP appears. The initial states can be obtained by reverse-time integration of the state transition equation; see Section 14.3.2.

14.2.2.2 Resolution completeness for $\dot{x} = u$

Sampling-based notions of completeness can be expressed in terms of reachable sets and the reachability graph. The requirement is to sample $\mathcal{U}$ in a way that causes the vertices of the reachability graph to eventually become dense in the reachable set, while also making sure that the reachability graph is systematically searched. All of the completeness concepts can be expressed in terms of forward or backward reachability graphs. Only the forward case will be described because the backward case is very similar.

To help bridge the gap with respect to motion planning as covered in Part II, first suppose: 1) $X = \mathcal{C} = \mathbb{R}^2$, 2) a state is denoted as $q = (x, y)$, 3) $U = [-1, 1]^2$, and 4) the state transition equation is $\dot{x} = u_1$ and $\dot{y} = u_2$. Suppose that the discrete-time model is applied to $\mathcal{U}$. Let $\Delta t = 1$ and

$$U_d = \{(-1, 0), (0, -1), (1, 0), (0, 1)\}, \tag{14.9}$$

which yields the Manhattan motion model from Example 7.4. Staircase paths are produced as was shown in Figure 7.40. In the present setting, these paths are obtained by integrating the action trajectory. From some state x_I, the reachability graph represents the set of all possible staircase paths with unit step size that can be obtained via (14.1).

Suppose that under this model, X_{free} is a bounded, open subset of $\mathbb{R}^2$. The connection to resolution completeness from Chapter 5 can be expressed clearly in this case. For any fixed Δt, a grid of a certain resolution is implicitly defined via the reachability graph. The task is to find an action sequence that leads to the goal (or a vertex close to it in the reachability graph) while remaining in X_{free}. Such a sequence can be found by a systematic search, as considered in Section 2.2. If the search is systematic, then it will correctly determine whether the reachability graph encodes a solution. If no solution exists, then the planning algorithm can decrease Δt by a constant factor (e.g., 2), and perform the systematic search again. This process repeats indefinitely until a solution is found. The algorithm runs forever if no solution exists (in practice, of course, one terminates early and gives up). The approach just described is resolution complete in the sense used in Chapter 5, even though all paths are expressed using action sequences.

The connection to ordinary motion planning is clear for this simple model because the action trajectories integrate to produce motions that follow a grid. As the time discretization is improved, the staircase paths can come arbitrarily close to some solution path. Looking at Figure 14.5, it can be seen that as the sampling resolution is improved with respect to U and T, the trajectories obtained via discrete-time approximations converge to any trajectory that can be obtained by integrating some $\tilde{u}$. In general, convergence occurs as Δt and the dispersion

of the sampling in U are driven to zero. This also holds in the same way for the more general case in which $\dot{x} = u$ and X is any smooth manifold. Imagine placing a grid down on X and refining it arbitrarily by reducing Δt.

14.2.2.3 Resolution completeness for $\dot{x} = f(x, u)$

Beyond the trivial case of $\dot{x} = u$, the reachability graph is usually not a simple grid. Even if X is bounded, the reachability graph may have an infinite number of vertices, even though Δt is fixed and U_d is finite. For a simple example, consider the Dubins car under the discretization $\Delta t = 1$. Fix $u_\phi = -\phi_{max}$ (turn left) for all $t \in T$. This branch alone generates a countably infinite number of vertices in the reachability graph. The circumference of the circle is $2\pi\rho_{min}$, in which ρ_{min} is the minimum turning radius. Let $\rho_{min} = 1$. Since the circumference is an irrational number, it is impossible to revisit the initial point by traveling k seconds for some integer k. It is even impossible to revisit any point on the circle. The set of vertices in the reachability graph is actually dense in the circle. This did not happen in Figure 14.6 because Δt and the circumference were rationally related (i.e., one can be obtained from the other via multiplication by a rational number). Consider what happens in the current example when $\rho_{min} = 1/\pi$ and $\Delta t = 1$.

Suppose that $\dot{x} = f(x, u)$ and the discrete-time model is used. To ensure convergence of the discrete-time approximation, f must be well-behaved. This can be established by requiring that all of the derivatives of f with respect to u and x are bounded above and below by a constant. More generally, f is assumed to be Lipschitz, which is an equivalent condition for cases in which the derivatives exist, but it also applies at points that are not differentiable. If U is finite, then the Lipschitz condition is that there exists some $c \in (0, \infty)$ such that

$$\|f(x, u) - f(x', u)\| \leq c\|x - x'\| \tag{14.10}$$

for all $x, x' \in X$, for all $u \in U$, and $\| \cdot \|$ denotes a norm on X. If U is infinite, then the condition is that there must exist some $c \in (0, \infty)$ such that

$$\|f(x, u) - f(x', u')\| \leq c(\|x - x'\| + \|u - u'\|), \tag{14.11}$$

for all $x, x' \in X$, and for all $u, u' \in U$. Intuitively, the Lipschitz condition indicates that if x and u are approximated by x' and u', then the error when substituted into f will be manageable. If convergence to optimal trajectories with respect to a cost functional is important, then Lipschitz conditions are also needed for $l(x, u)$. Under such mild assumptions, if Δt and the dispersion of samples of U_d is driven down to zero, then the trajectories obtained from integrating discrete action sequences come arbitrarily close to solution trajectories. In other words, action sequences provide arbitrarily close approximations to any $\tilde{u} \in U$. If f is Lipschitz, then the integration of (14.14) yields approximately the same result for $\tilde{u}$ as the approximating action sequence.

In the limit as Δt and the dispersion of U_d approach zero, the reachability graph becomes dense in the reachable set $R(x_I, \mathcal{U})$. Ensuring a systematic search

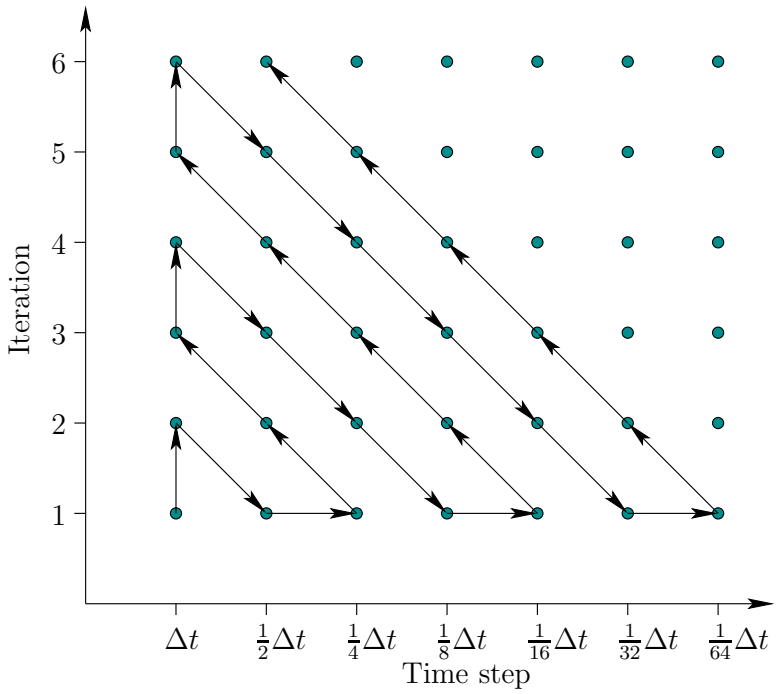


Figure 14.7: By systematically alternating between exploring different reachability graphs, resolution completeness can be achieved, even if each reachability graph has a countably infinite number of vertices.

for the case of a grid was not difficult because there is only a finite number of vertices at each resolution. Unfortunately, the reachability graph may generally have a countably infinite number of vertices for some fixed discrete-time model, even if X is bounded.

To see that resolution-complete algorithms nevertheless exist if the reachability graph is countably infinite, consider *triangular enumeration*, which proves that $\mathbb{N} \times \mathbb{N}$ is countable, in which $\mathbb{N}$ is the set of natural numbers. The proof proceeds by giving a sequence that starts at $(0, 0)$ and proceeds by sweeping diagonally back and forth across the first quadrant. In the limit, all points are covered. The same idea can be applied to obtain resolution-complete algorithms. A sequence of discrete-time models can be made for which the time step Δt and the dispersion of the sampling of U approach zero. Each discretization produces a reachability graph that has a countable number of vertices.

A resolution-complete algorithm can be made by performing the same kind of zig-zagging that was used to show that $\mathbb{N} \times \mathbb{N}$ is countable. See Figure 14.7; suppose that U is finite and $U_d = U$. Along the horizontal axis is a sequence of improving discrete-time models. Each model generates its own reachability graph, for which a systematic search eventually explores all of its vertices. Imagine this exploration occurs one step at a time, in which one new vertex is reached in each step. The vertical axis in Figure 14.7 indicates the number of vertices reached so far by the search algorithm. A countably infinite set of computers could explore all

of reachability graphs in parallel. With a single computer, it can still be assured that everything is eventually explored by zig-zagging as shown. Thus a resolution-complete algorithm always exists if U is finite. If U is not finite, then U_d must also be refined as the time step is decreased. Of course, there are numerous other ways to systematically explore all of the reachability graphs. The challenging task is to find a way that leads to good performance in practice.

The discussion so far has assumed that a sampling-based algorithm can uncover a subgraph of the reachability graph. This neglects numerical issues such as arithmetic precision and numerical integration error. Such issues can additionally be incorporated into a resolution completeness analysis [196].

14.2.3 Motion Primitives

The discrete-time model of Section 14.2.2 is just one of many possible ways to discretize the space of action trajectories. It will now be considered as a special case of specifying *motion primitives*. The restriction to constant actions over fixed time intervals may be too restrictive in many applications. Suppose we want to automate the motions of a digital actor for use in a video game or film. Imagine having a database of interesting motion primitives. Such primitives could be extracted, for example, from motion capture data [35, 553]. For example, if the actor is designed for kung-fu fighting, then each motion sequence may correspond to a basic move, such a kick or punch. It is unlikely that such motion primitives correspond to constant actions over a fixed time interval. The durations of the motion primitives will usually vary.

Such models can generally be handled by defining a more general kind of discretization. The discrete-time model can be used to formulate a discrete-time state transition equation of the form

$$x_{k+1} = f_d(x_k, u_k), \tag{14.12}$$

in which $x_k = x((k-1)\Delta t)$, $x_{k+1} = x(k\Delta t)$, and u_k is the action in U_d that is applied from time $(k-1)\Delta t$ to time $k\Delta t$. Thus, f_d is a function $f_d : X \times U_d \rightarrow X$ that represents an approximation to f, the original state transition function. Every constant action $u \in U_d$ applied over Δt can be considered as a motion primitive.

Now generalize the preceding construction to allow more general motion primitives. Let $\tilde{u}^p$ denote a motion primitive, which is a function from an interval of time into U. Let the interval of time start at 0 and stop at $t_F(\tilde{u}^p)$, which is a final time that depends on the particular primitive. From any state $x \in X_{free}$, suppose that a set $\mathcal{U}^p(x)$ of motion primitives is available. The set may even be infinite, in which case some additional sampling must eventually be performed over the space of motion primitives by a local planning method. A state transition equation that operates over discrete stages can be defined as

$$x_{k+1} = f_p(x_k, \tilde{u}_k^p), \tag{14.13}$$

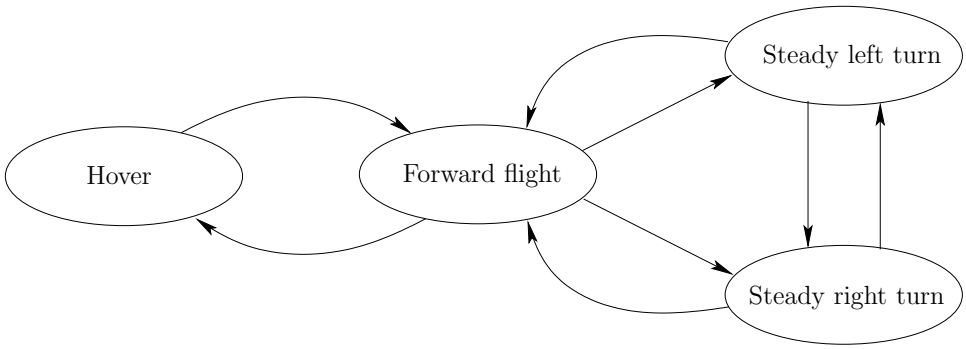


Figure 14.8: A maneuver automaton, proposed by Frazzoli [360], captures the constraints on allowable sequences of motion primitives.

in which $\tilde{u}_k^p$ is a motion primitive that must be chosen from $\mathcal{U}^p(x_k)$. The time discretization model and (14.12) can be considered as a special case in which the motion primitives are all constant over a fixed time interval $[0, \Delta t)$. Note that in (14.13) the stage index k does not necessarily correspond to time $(k-1)\Delta t$. The index k merely represents the fact that $k-1$ motion primitives have been applied so far, and it is time to decide on the kth motion primitive. The current time is determined by summing the durations of all $k-1$ primitives applied so far. If a set $\mathcal{U}^p(x)$ of primitives is given for all $x \in X$, then a reachability graph and its swath can be defined by simple extensions of the discrete-time case. The discrete-time model $\mathcal{U}_d$ can now be interpreted as a special set of motion primitives.

For some motion primitives, it may not be possible to immediately sequence them without applying transitional motions. For example, in [362], two different kinds of motion primitives, called *trim trajectories* and *maneuvers*, are defined for autonomous helicopter flight. The trim trajectories correspond to steady motions, and maneuvers correspond to unsteady motions that are needed to make transitions between steady motions. Transitions from one trim trajectory to another are only permitted through the execution of a maneuver. The problem can be nicely modeled as a hybrid system in which each motion primitive represents a mode [360] (recall hybrid system concepts from Sections 7.3, 8.3.1, and 10.6). The augmented state space is $X \times M$, in which M is a set of modes. The transition equation (14.13) can be extended over the augmented state space so that motion primitives can change modes in addition to changing the original state. The possible trajectories for the helicopter follow paths in a graph called the *maneuver automaton*. An example from [360] is shown in Figure 14.8. Every edge and every vertex corresponds to a mode in the maneuver automaton. Each edge or vertex actually corresponds to a parameterized family of primitives, from which a particular one is chosen based on the state. A similar state machine is proposed in [452] for animating humans, and the motion primitives are called *behaviors*.

Discretizations based on general motion primitives offer great flexibility, and in many cases dramatic performance improvements can be obtained in a sampling-based planning algorithm. The main drawback is that the burden of establishing

resolution completeness is increased.

14.3 Sampling-Based Motion Planning Revisited

Now that the preliminary concepts have been defined for motion planning under differential constraints, the focus shifts to extending the sampling-based planning methods of Chapter 5. This primarily involves extending the incremental sampling and searching framework from Section 5.4 to incorporate differential constraints. Following the general framework, several popular methods are covered in Section 14.4 as special cases of the framework. If an efficient BVP solver is available, then it may also be possible to extend sampling-based roadmaps of Section 5.6 to handle differential constraints.

14.3.1 Basic Components

This section describes how Sections 5.1 to 5.3 are adapted to handle phase spaces and differential constraints.

14.3.1.1 Distance and volume in X

Recall from Chapter 5 that many sampling-based planning algorithms rely on measuring distances or volumes in $\mathcal{C}$. If $X = \mathcal{C}$, as in the wheeled systems from Section 13.1.2, then the concepts of Section 5.1 apply directly. The equivalent is needed for a general state space X, which may include phase variables in addition to the configuration variables. In most cases, the topology of the phase variables is trivial. For example, if $x = (q, \dot{q})$, then each $\dot{q}_i$ component is constrained to an interval of $\mathbb{R}$. In this case the velocity components are just an axis-aligned rectangular region in $\mathbb{R}^{n/2}$, if n is the dimension of X. It is straightforward in this case to extend a measure and metric defined on $\mathcal{C}$ up to X by forming the Cartesian product.

A metric can be defined using the Cartesian product method given by (5.4). The usual difficulty arises of arbitrarily weighting different components and combining them into a single scalar function. In the case of $\mathcal{C}$, this has involved combining translations and rotation. For X, this additionally includes velocity components, which makes it more difficult to choose meaningful weights.

Riemannian metrics A rigorous way to define a metric on a smooth manifold is to define a *metric tensor* (or *Riemannian tensor*), which is a quadratic function of two tangent vectors. This can be considered as an inner product on X, which can be used to measure angles. This leads to the definition of the *Riemannian metric*, which is based on the shortest paths (called *geodesics*) in X [133]. An example of this appeared in the context of Lagrangian mechanics in Section 13.4.1. The kinetic energy, (13.70), serves as the required metric tensor, and the geodesics are the motions taken by the dynamical system to conserve energy. The metric

can be defined as the length of the geodesic that connects a pair of points. If the chosen Riemannian metric has some physical significance, as in the case of Lagrangian mechanics, then the resulting metric provides meaningful information. Unfortunately, it may be difficult or expensive to compute its value.

The ideal distance function The ideal way to define distance on X is to use a cost functional and then define the distance from $x \in X_{free}$ to $x' \in X_{free}$ as the optimal cost-to-go from x to x' while remaining in X_{free}. In some cases, it has been also referred to as the *nonholonomic metric*, *Carnot-Caratheodory metric*, or *sub-Riemannian metric* [596]. Note that this not a true metric, as mentioned in Section 5.1.2, because the cost may not be symmetric. For example, traveling a small distance forward with Dubins car is much shorter than traveling a small distance backward. If there are obstacles, it may not even be possible to reach configurations behind the car.

This concept of distance should be somewhat disturbing because it requires optimally solving the motion planning problem of Formulation 14.1. Thus, it cannot be practical for efficient use in a motion planning algorithm. Nevertheless, understanding this ideal notion of distance can be very helpful in designing practical distance functions on X. For example, rather than using a weighted Euclidean metric (often called *Mahalanobis metric*) for the Dubins car, a distance function can be defined based on the length of the shortest path between two configurations. These lengths are straightforward to compute, and are based on the optimal curve families that will be covered in Section 15.3. This distance function neglects obstacles, but it should still provide better distance information than the weighted Euclidean metric. It may also be useful for car models that involve dynamics.

The general idea is to get as close as possible to the optimal cost-to-go without having to perform expensive computations. It is often possible to compute a useful underestimate of the optimal cost-to-go by neglecting some of the constraints, such as obstacles or dynamics. This may help in applying A^* search heuristics.

Defining measure As mentioned already, it is straightforward to extend a measure on $\mathcal{C}$ to X if the topology associated with the phase variables is trivial. It may not be possible, however, to obtain an invariant measure. In most cases, $\mathcal{C}$ is a transformation group, in which the Haar measure exists, thereby yielding the "true" volume in a sense that is not sensitive to parameterizations of $\mathcal{C}$. This was observed for $SO(3)$ in Section 5.1.4. For a general state space X, a Haar measure may not exist. If a Riemannian metric is defined, then intrinsic notions of surface integration and volume exist [133]; however, these may be difficult to exploit in a sampling-based planning algorithm.

14.3.1.2 Sampling theory

Section 14.2.2 already covered some of the sampling issues. There are at least two continuous spaces: X, and the time interval T. In most cases, the action space

U is also continuous. Each continuous space must be sampled in some way. In the limit, it is important that any sample sequence is dense in the space on which sampling occurs. This was required for the resolution completeness concepts of Section 14.2.2.

Sampling of T and U can be performed by directly using the random or deterministic methods of Section 5.2. Time is just an interval of $\mathbb{R}$, and U is typically expressed as a convex m-dimensional subset of $\mathbb{R}^m$. For example, U is often an axis-aligned rectangular subset of $\mathbb{R}^m$.

Some planning methods may require sampling on X. The definitions of discrepancy and dispersion from Section 5.2 can be easily adapted to any measure space and metric space, respectively. Even though it may be straightforward to define a good criterion, generating samples that optimize the criterion may be difficult or impossible.

A convenient way to avoid this problem is to work in a coordinate neighborhood of X. This makes the manifold appear as an n-dimensional region in $\mathbb{R}^n$, which in many cases is rectangular. This enables the sampling concepts of Section 5.2 to be applied in a straightforward manner. While this is the most straightforward approach, the sampling quality depends on the particular parameterization used to define the coordinate neighborhood. Note that when working with a coordinate neighborhood (for example, by imagining that X is a cube), appropriate identifications must be taken into account.

14.3.1.3 Collision detection

As in Chapter 5, efficient collision detection algorithms are a key enabler of sampling-based planning. If $X = \mathcal{C}$, then the methods of Section 5.3 directly apply. If X includes phase constraints, then additional tests must be performed. These constraints are usually given and are therefore straightforward to evaluate. Recall from Section 4.3 that this is not efficient for the obstacle constraints on $\mathcal{C}$ due to the complicated mapping between obstacles in $\mathcal{W}$ and obstacles in $\mathcal{C}$.

If only pointwise tests are performed, the trajectory segment between the points is not guaranteed to stay in X_{free}. This problem was addressed in Section 5.3.4 by using distance information from collision checking algorithms. The same problem exists for the phase constraints of the form $h_i(x) \leq 0$. In this general form there is no additional information that can be used to ensure that some neighborhood of x is contained in X_{free}. Fortunately, the phase constraints are not complicated in most applications, and it is possible to ensure that x is at least some distance away from the constraint boundary. In general, careful analysis of each phase constraint is required to ensure that the state trajectory segments are violation-free.

In summary, determining whether $x \in X_{free}$ involves

1. Using a collision detection algorithm as in Section 5.3 to ensure that $\kappa(x) \in \mathcal{C}_{free}$.

2. Checking x to ensure that other constraints of the form $h_i(x) \leq 0$ have been satisfied.

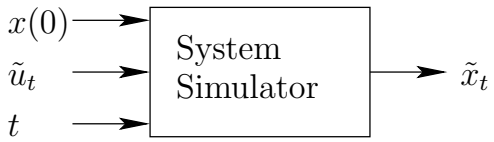


Figure 14.9: Using a system simulator, the system $\dot{x} = f(x, u)$ is integrated from $x(0)$ using $\tilde{u}_t : [0, t] \rightarrow U$ to produce a state trajectory $\tilde{x}_t : [0, t] \rightarrow X$. Sometimes $\tilde{x}$ is specified as a parameterized path, but most often it is approximated as a sequence of samples in X.

Entire trajectory segments should theoretically be checked. Often times, in practice, only individual points are checked, which is more efficient but technically incorrect.

14.3.2 System Simulator

A new component is needed for sampling-based planning under differential constraints because of (14.1). Motions are now expressed in terms of an action trajectory, but collision detection and constraint satisfaction tests must be performed in X. Therefore, the system, $\dot{x} = f(x, u)$ needs to be integrated frequently during the planning process. Similar to the modeling of collision detection as a "black box," the integration process is modeled as a module called the *system simulator*. See Figure 14.9. Since the systems considered in this chapter are time-invariant, the starting time for any required integration can always be shifted to start at $t = 0$. Integration can be considered as a module that implements (14.1) by computing the state trajectory resulting from a given initial state $x(0)$, an action trajectory $\tilde{u}_t$, and time t. The incremental simulator encapsulates the details of integrating the state transition equation so that they do not need to be addressed in the design of planners. However, that information from the particular state transition equation may still be important in the design of the planning algorithm.

Closed-form solutions According to (14.1), the action trajectory must be integrated to produce a state trajectory. In some cases, this integration leads to a closed-form expression. For example, if the system is a chain of integrators, then a polynomial expression can easily be obtained for $x(t)$. For example, suppose q is a scalar and $\ddot{q} = u$. If $q(0) = \dot{q}(0) = 0$ and a constant action $u = 1$ is applied, then $x(t) = t^2/2$. If $\dot{x} = f(x, u)$ is a linear system (which includes chains of integrators; recall the definition from Section 13.2.2), then a closed-form expression for the state trajectory can always be obtained. This is based on matrix exponentials and is given in many control theory texts (e.g, [192]).

Euler method For most systems, the integration must be performed numerically. A system simulator based on numerical integration can be constructed by breaking t into smaller intervals and iterating classical methods for computing

numerical solutions to differential equations. The Euler method is the simplest of these methods. Let Δt denote a small time interval over which the approximation will be made. This can be considered as an internal parameter of the system simulator. In practice, this Δt is usually much smaller than the Δt used in the discrete-time model of Section 14.2.2. Suppose that $x(0)$ and $u(0)$ are given and the task is to estimate $x(\Delta t)$.

By performing integration over time, the state transition equation can be used to determine the state after some fixed amount of time Δt has passed. For example, if $x(0)$ is given and $u(t')$ is known over the interval $t' \in [0, \Delta t]$, then the state at time Δt can be determined as

$$x(\Delta t) = x(0) + \int_0^{\Delta t} f(x(t), u(t))dt. \tag{14.14}$$

The integral cannot be evaluated directly because $x(t)$ appears in the integrand and is unknown for time $t > 0$.

Using the fact that

$$f(x, u) = \dot{x} = \frac{dx}{dt} \approx \frac{x(\Delta t) - x(0)}{\Delta t}, \tag{14.15}$$

solving for $x(\Delta t)$ yields the classic *Euler integration method*

$$x(\Delta t) \approx x(0) + \Delta t \; f(x(0), u(0)). \tag{14.16}$$

The approximation error depends on how quickly $x(t)$ changes over time and on the length of the interval Δt. If the planning algorithm applies a motion primitive $\tilde{u}^p$, it gives $t_F(\tilde{u}^p)$ as the time input, and the system simulator may subdivide the time interval to maintain higher accuracy. This allows the developer of the planning algorithm to ignore numerical accuracy issues.

Runge-Kutta methods Although Euler integration is efficient and easy to understand, it generally yields poor approximations. Taking a Taylor series expansion of $\tilde{x}$ at $t = 0$ yields

$$x(\Delta t) = x(0) + \Delta t \; \dot{x}(0) + \frac{(\Delta t)^2}{2!}\ddot{x}(0) + \frac{(\Delta t)^3}{3!}x^{(3)}(0) + \cdots . \tag{14.17}$$

Comparing to (14.16), it can be seen that the Euler method just uses the first term of the Taylor series, which is an exact representation (if $\tilde{x}$ is analytic). Thus, the neglected terms reflect the approximation error. If $x(t)$ is roughly linear, then the error may be small; however, if $\dot{x}(t)$ or higher order derivatives change quickly, then poor approximations are obtained.

Runge-Kutta methods are based on using higher order terms of the Taylor series expansion. One of the most widely used and efficient numerical integration methods is the fourth-order Runge-Kutta method. It is simple to implement and

yields good numerical behavior in most applications. Also, it is generally recommended over Euler integration. The technique can be derived by performing a Taylor series expansion at $x(\frac{1}{2}\Delta t)$. This state itself is estimated in the approximation process.

The fourth-order *Runge-Kutta integration method* is

$$x(\Delta t) \approx x(0) + \frac{\Delta t}{6}(w_1 + 2w_2 + 2w_3 + w_4), \tag{14.18}$$

in which

$$\begin{aligned} w_1 &= f(x(0), u(0)) \\ w_2 &= f(x(0) + \tfrac{1}{2}\Delta t\ w_1,\ u(\tfrac{1}{2}\Delta t)) \\ w_3 &= f(x(0) + \tfrac{1}{2}\Delta t\ w_2,\ u(\tfrac{1}{2}\Delta t)) \\ w_4 &= f(x(0) + \Delta t\ w_3,\ u(\Delta t)). \end{aligned} \tag{14.19}$$

Although this is more expensive than Euler integration, the improved accuracy is usually worthwhile in practice. Note that the action is needed at three different times: 0, $\frac{1}{2}\Delta t$, and Δt. If the action is constant over $[0, \Delta t)$, then the same value is used at all three times.

The approximation error depends on how quickly higher order derivatives of $\tilde{x}$ vary over time. This can be expressed using the remaining terms of the Taylor series. In practice, it may be advantageous to adapt Δt over successive iterations of Runge-Kutta integration. In [247], for example, it is suggested that Δt is scaled by $(\Delta t/\Delta x)^{1/5}$, in which $\Delta x = \|x(\Delta t) - x(0)\|$, the Euclidean distance in $\mathbb{R}^n$.

Multistep methods Runge-Kutta methods represent a popular trade-off between simplicity and efficiency. However, by focusing on the integration problem more carefully, it is often possible to improve efficiency further. The Euler and Runge-Kutta methods are often referred to as *single-step methods*. There exist *multi-step methods*, which rely on the fact that a sequence of integrations will be performed, in a manner analogous to incremental collision detection in Section 5.3.3. The key issues are ensuring that the methods properly initialize, ensuring numerical stability over time, and estimating error to adaptively adjust the step size. Many books on numerical analysis cover multi-step methods [51, 440, 863]. One of the most popular families is the *Adams methods*.

Multistep methods require more investment to understand and implement. For a particular application, the decision to pursue this route should be based on the relative costs of planning, collision detection, and numerical integration. If integration tends to dominate and efficiency is critical, then multi-step methods could improve running times dramatically over Runge-Kutta methods.

Black-box simulators For some problems, a state transition equation might not be available; however, it is still possible to compute future states given a current state and an action trajectory. This might occur, for example, in a complex

software system that simulates the dynamics of a automobile or a collection of parts that bounce around on a table. In computer graphics applications, simulations may arise from motion capture data. Some simulators may even work internally with implicit differential constraints of the form $g_i(x, \dot{x}, u) = 0$, instead of $\dot{x} = f(x, u)$. In such situations, many sampling-based planners can be applied because they rely only on the existence of the system simulator. The planning algorithm is thus shielded from the particular details of how the system is represented and integrated.

Reverse-time system simulation Some planning algorithms require integration in the reverse-time direction. For some given $x(0)$ and action trajectory that runs from $-\Delta t$ to 0, the *backward system simulator* computes a state trajectory, $\tilde{x} : [-t, 0] \rightarrow X$, which when integrated from $-\Delta t$ to 0 under the application of $\tilde{u}_t$ yields $x(0)$. This may seem like an *inverse control problem* [856] or a BVP as shown in Figure 14.10; however, it is much simpler. Determining the action trajectory for given initial and goal states is more complicated; however, in reverse-time integration, the action trajectory and final state are given, and the initial state does not need to be fixed.

The reverse-time version of (14.14) is

$$x(-\Delta t) = x(0) + \int_0^{-\Delta t} f(x(t), u(t))dt = x(0) + \int_0^{\Delta t} -f(x(t), u(t))dt, \quad (14.20)$$

which relies on the fact that $\dot{x} = f(x, u)$ is time-invariant. Thus, reverse-time integration is obtained by simply negating the state transition equation. The Euler and Runge-Kutta methods can then be applied in the usual way to $-f(x(t), u(t))$.

14.3.3 Local Planning

The methods of Chapter 5 were based on the existence of a local planning method (LPM) that is simple and efficient. This represented an important part of both the incremental sampling and searching framework of Section 5.4 and the sampling-based roadmap framework of Section 5.6. In the absence of obstacles and differential constraints, it is trivial to define an LPM that connects two configurations. They can, for example, be connected using the shortest path (geodesic) in $\mathcal{C}$. The sampling-based roadmap approach from Section 5.6 relies on this simple LPM.

In the presence of differential constraints, the problem of constructing an LPM that connects two configurations or states is considerably more challenging. Recall from Section 14.1 that this is the classical BVP, which is difficult to solve for most systems. There are two main alternatives to handle this difficulty in a sampling-based planning algorithm:

1. Design the sampling scheme, which may include careful selection of motion primitives, so that the BVP can be trivially solved.

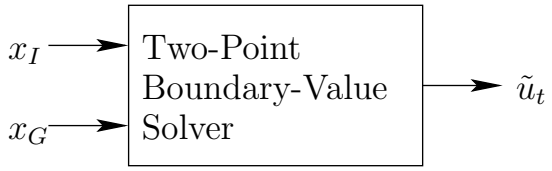


Figure 14.10: Some methods in Chapter 15 can solve two-point boundary value problems in the absence of X_{obs}. This is difficult to obtain for most systems, but it is more powerful than the system simulator. It is very valuable, for example, in making a sampling-based roadmap that satisfies differential constraints.

2. Design the planning algorithm so that as few as possible BVPs need to be solved. The LPM in this case does not specify precise goal states that must be reached.

Under the first alternative, the BVP solver can be considered as a black box, as shown in Figure 14.10, that efficiently connects x_I to x_G in the absence of obstacles. In the case of the Piano Mover's Problem, this was obtained by moving along the shortest path in $\mathcal{C}$. For many of the wheeled vehicle systems from Section 13.1.2, *steering methods* exist that could serve as an efficient BVP solver; see Section 15.5. Efficient techniques also exist for linear systems and are covered in Section 15.2.2.

If the BVP is efficiently solved, then virtually any sampling-based planning algorithm from Chapter 5 can be adapted to the case of differential constraints. This is achieved by using the module in Figure 14.10 as the LPM. For example, a sampling-based roadmap can use the computed solution in the place of the shortest path through $\mathcal{C}$. If the BVP solver is not efficient enough, then this approach becomes impractical because it must typically be used thousands of times to build a roadmap. The existence of an efficient module as shown in Figure 14.10 magically eliminates most of the complications associated with planning under differential constraints. The only remaining concern is that the solutions provided by the BVP solver could be quite long in comparison to the shortest path in the absence of differential constraints (for example, how far must the Dubins car travel to move slightly backward?).

Under the second alternative, it is assumed that solving the BVP is very costly. The planning method in this case should avoid solving BVPs whenever possible. Some planning algorithms may only require an LPM that *approximately* reaches intermediate goal states, which is simpler for some systems. Other planning algorithms may not require the LPM to make any kind of connection. The LPM may return a motion primitive that appears to make some progress in the search but is not designed to connect to a prescribed state. This usually involves incremental planning methods, which are covered in Section 14.4 and extends the methods of Sections 5.4 and 5.5 to handle differential constraints.

14.3.4 General Framework Under Differential Constraints

The framework presented here is a direct extension of the sampling and searching framework from Section 5.4.1 and includes the extension of Section 5.5 to allow the selection of any point in the swath of the search graph. This replaces the vertex selection method (VSM) by a swath-point selection method (SSM). The framework also naturally extends the discrete search framework of Section 2.2.4. The components are are follows:

1. **Initialization:** Let $\mathcal{G}(V, E)$ represent an undirected *search graph*, for which the vertex set V contains a vertex for x_I and possibly other states in X_{free}, and the edge set E is empty. The graph can be interpreted as a topological graph with a swath $S(\mathcal{G})$.

2. **Swath-point Selection Method (SSM):** Choose a vertex $x_{cur} \in S(\mathcal{G})$ for expansion.

3. **Local Planning Method (LPM):** Generate a motion primitive $\tilde{u}^p : [0, t_F] \rightarrow X_{free}$ such that $u(0) = x_{cur}$ and $u(t_F) = x_r$ for some $x_r \in X_{free}$, which may or may not be a vertex in $\mathcal{G}$. Using the system simulator, a collision detection algorithm, and by testing the phase constraints, $\tilde{u}^p$ must be verified to be violation-free. If this step fails, then go to Step 2.

4. **Insert an Edge in the Graph:** Insert $\tilde{u}^p$ into E. Upon integration, $\tilde{u}^p$ yields a state trajectory from x_{cur} to x_r. If x_r is not already in V, it is added. If x_{cur} lies in the interior of an edge trajectory for some $e \in E$, then e is split by the introduction of a new vertex at x_{cur}.

5. **Check for a Solution:** Determine whether $\mathcal{G}$ encodes a solution path. In some applications, a small gap in the state trajectory may be tolerated.

6. **Return to Step 2:** Iterate unless a solution has been found or some termination condition is satisfied. In the latter case, the algorithm reports failure.

The general framework may be applied in the same ways as in Section 5.4.1 to obtain unidirectional, bidirectional, and multidirectional searches. The issues from the Piano Mover's Problem extend to motion planning under differential constraints. For example, bug traps cause the same difficulties, and as the number of trees increases, it becomes difficult to coordinate the search.

The main new complication is due to BVPs. See Figure 14.11. Recall from Section 14.1.1 that for most systems it is important to reduce the number of BVPs that must be solved during planning as much as possible. Assume that connecting precisely to a prescribed state is difficult. Figure 14.11a shows the best situation, in which forward, unidirectional search is used to enter a large goal region. In this case, no BVPs need to be solved. As the goal region is reduced, the problem

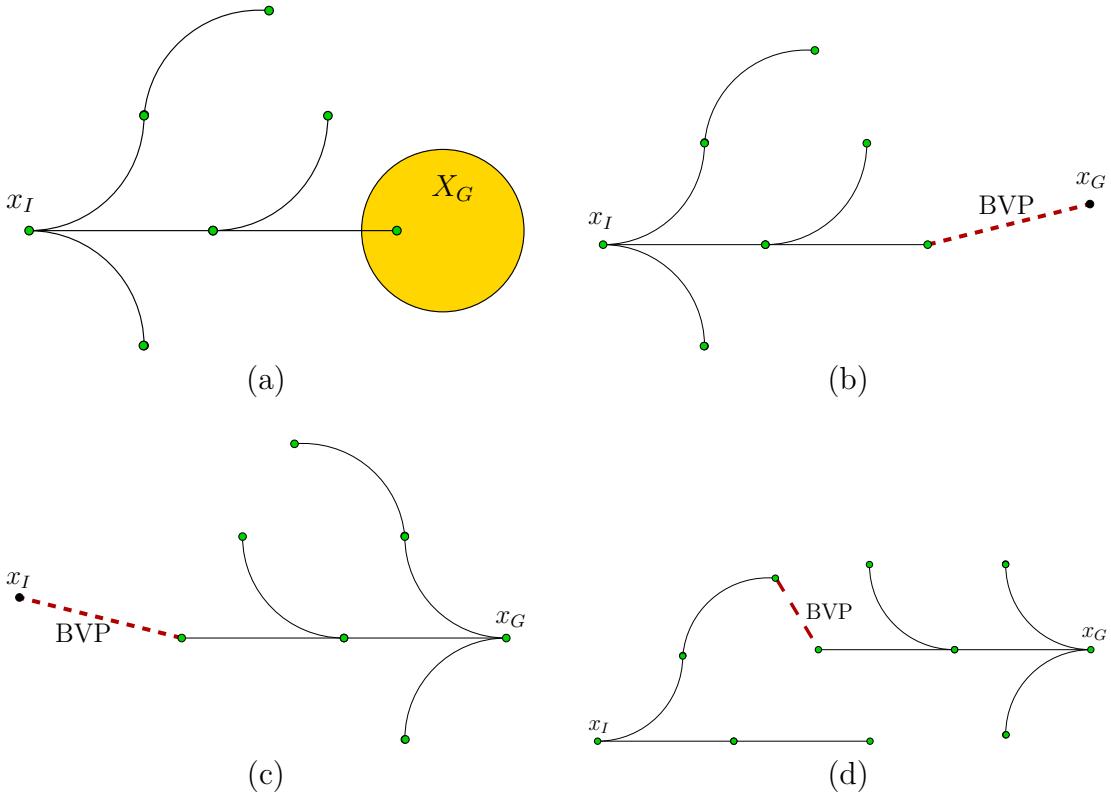


Figure 14.11: (a) Forward, unidirectional search for which the BVP is avoided. (b) Reaching the goal precisely causes a BVP. (c) Backward, unidirectional search also causes a BVP. (d) For bidirectional search, the BVP arises when connecting the trees.

becomes more challenging. Figure 14.11b shows the limiting case in which X_G is a point $\{x_G\}$. This requires the planning algorithm to solve at least one BVP.

Figure 14.11c shows the case of backward, unidirectional search. This has the effect of moving the BVP to x_I. Since x_I is precisely given (there is no "initial region"), the BVP cannot be avoided as in the forward case. If an algorithm produces a solution $\tilde{u}$ for which $x(0)$ is very close to x_I, and if X_G is large, then it may be possible to salvage the solution. The system simulator can be applied to $\tilde{u}$ from x_I instead of $x(0)$. It is known that $\tilde{x}(x(0), \tilde{u})$ is violation-free, and $\tilde{x}(x_I, \tilde{u})$ may travel close to $\tilde{x}(x(0), \tilde{u})$ at all times. This requires f to vary only a small amount with respect to changes in x (this would be implied by a small Lipschitz constant) and also for $\|x_I - x(0)\|$ to be small. One problem is that the difference between points on the two trajectories usually increases as time increases. If it is verified by the system simulator that $\tilde{x}(x_I, \tilde{u})$ is violation-free and the final state still lies in X_G, then a solution can be declared.

For bidirectional search, a BVP must be solved somewhere in the middle of a trajectory, as shown in Figure 14.11d. This complicates the problem of determining

whether the two trees can be connected. Once again, if the goal region is large, it may be possible remove the gap in the middle of the trajectory by moving the starting state of the trajectory produced by the backward tree. Let $\tilde{u}_1$ and $\tilde{u}_2$ denote the action trajectories produced by the forward and backward trees, respectively. Suppose that their termination times are t_1 and t_2, respectively. The action trajectories can be concatenated to yield a function $\tilde{u} : [0, t_1 + t_2] \rightarrow U$ by shifting the domain of $\tilde{u}_2$ from $[0, t_2]$ to $[t_1, t_1 + t_2]$. If $t \leq t_1$, then $u(t) = u_1(t)$; otherwise, $u(t) = u_2(t - t_1)$. If there is a gap, the new state trajectory $\tilde{x}(x_I, \tilde{u})$ must be checked using the system simulator to determine whether it is violation-free and terminates in X_G. Multi-directional search becomes even more difficult because more BVPs are created. It is possible in principle to extend the ideas above to concatenate a sequence of action trajectories, which tries to remove all of the gaps.

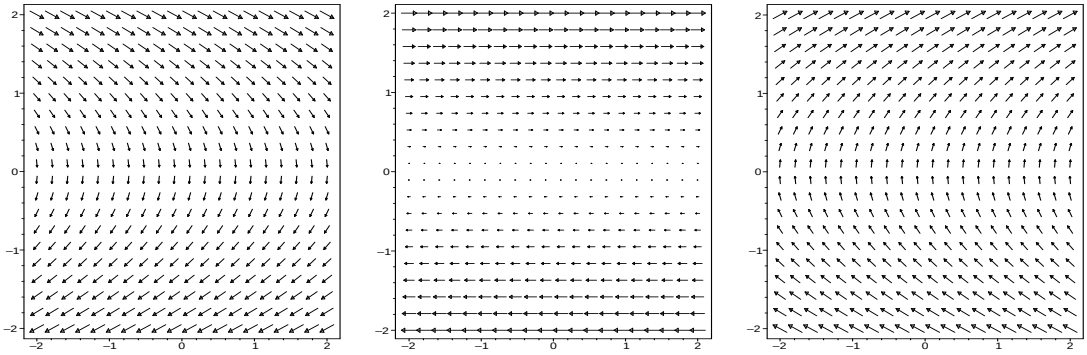
Consider the relationship between the search graph and reachability graphs. In the case of unidirectional search, the search graph is always a subset of a reachability graph (assuming perfect precision and no numerical integration error). In the forward case, the reachability graph starts at x_I, and in the backward case it starts at x_G. In the case of bidirectional search, there are two reachability graphs. It might be the case that vertices from the two coincide, which is another way that the BVP can be avoided. Such cases are unfortunately rare, unless x_I and x_G are intentionally chosen to cause this. For example, the precise location of x_G may be chosen because it is known to be a vertex of the reachability graph from x_I. For most systems, it is difficult to force this behavior. Thus, in general, BVPs arise because the reachability graphs do not have common vertices. In the case of multi-directional search, numerous reachability graphs are being explored, none of which may have vertices that coincide with vertices of others.

14.4 Incremental Sampling and Searching Methods

The general framework of Section 14.3.4 will now be specialized to obtain three important methods for planning under differential constraints.

14.4.1 Searching on a Lattice

This section follows in the same spirit as Section 5.4.2, which adapted grid search techniques to motion planning. The difficulty in the current setting is to choose a discretization that leads to a lattice that can be searched using any of the search techniques of Section 2.2. The section is inspired mainly by kinodynamic planning work [288, 290, 441].



(a) $u = -1$ (b) $u = 0$ (c) $u = 1$

Figure 14.12: The reachability graph will be obtained by switching between these vector fields at every Δt. The middle one produces horizontal phase trajectories, and the others produce parabolic curves.

14.4.1.1 A double-integrator lattice

First consider the double integrator from Example 13.3. Let $\mathcal{C} = \mathcal{C}_{free} = \mathbb{R}$ and $\ddot{q} = u$. This models the motion of a free-floating particle in $\mathbb{R}$, as described in Section 13.3.2. The phase space is $X = \mathbb{R}^2$, and $x = (q, \dot{q})$. Let $U = [-1, 1]$. The coming ideas can be easily generalized to allow any acceleration bound $a_{max} > 0$ by letting $U = [-a_{max}, a_{max}]$; however, $a_{max} = 1$ will be chosen to simplify the presentation.

The differential equation $\ddot{q} = u$ can be integrated once to yield

$$\dot{q}(t) = \dot{q}(0) + ut, \tag{14.21}$$

in which $\dot{q}(0)$ is an initial speed. Upon integration of (14.21), the position is obtained as

$$q(t) = q(0) + \dot{q}(0)\, t + \tfrac{1}{2} u t^2, \tag{14.22}$$

which uses two initial conditions, $q(0)$ and $\dot{q}(0)$.

A discrete-time model exists for which the reachability graph is trapped on a lattice. This is obtained by letting $U_d = \{-1, 0, 1\}$ and Δt be any positive real number. The vector fields over X that correspond to the cases of $u = -1$, $u = 0$, and $u = 1$ are shown in Figure 14.12. Switching between these fields at every Δt and integrating yields the reachability graph shown in Figure 14.13.

This leads to a discrete-time transition equation of the form $x_{k+1} = f_d(x_k, u_k)$, in which $u_k \in U_d$, and k represents time $t = (k-1)\Delta t$. Any action trajectory can be specified as an action sequence; for example a six-stage action sequence may be given by $(-1, 1, 0, 0, -1, 1)$. Start from $x_1 = x(0) = (q_1, \dot{q}_1)$. At any stage k and for any action sequence, the resulting state $x_k = (q_k, \dot{q}_k)$ can be expressed as

$$\begin{aligned} q_k &= q_1 + i\tfrac{1}{2}(\Delta t)^2 \\ \dot{q}_k &= \dot{q}_1 + j\Delta t, \end{aligned} \tag{14.23}$$

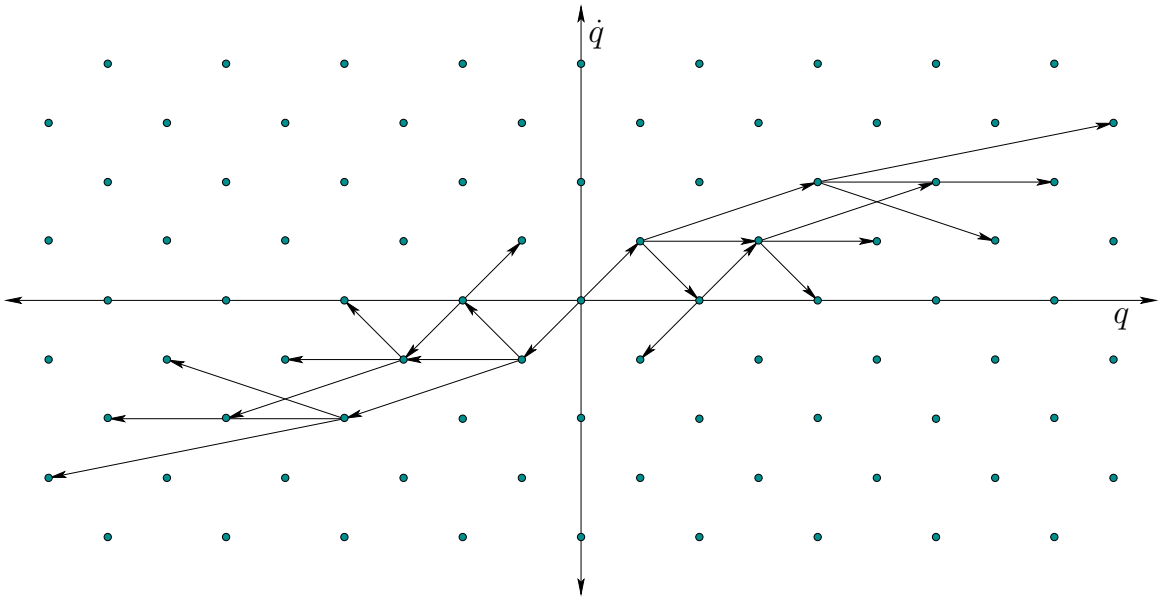
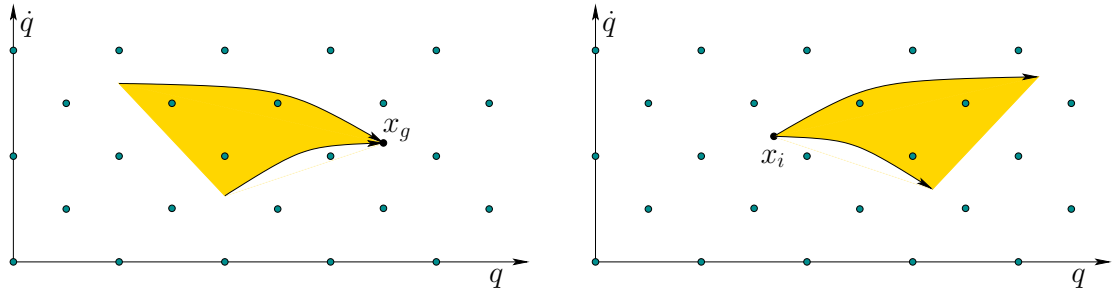


Figure 14.13: The reachability graph from the origin is shown after three stages (the true state trajectories are actually parabolic arcs when acceleration or deceleration occurs). Note that a lattice is obtained, but the distance traveled in one stage increases as $|\dot{q}|$ increases.

in which i, j are integers that can be computed from the action sequence. Thus, any action sequence leads to a state that can be expressed using integer coordinates (i, j) in the plane. Starting at $x_1 = (0, 0)$, this forms the lattice of points shown in Figure 14.13. The lattice is slanted (with slope 1) because changing speed requires some motion. If infinite acceleration were allowed, then $\dot{q}$ could be changed instantaneously, which corresponds to moving vertically in X. As seen in (14.21), $\dot{q}$ changes linearly over time. If $q \neq 0$, then the configuration changes quadratically. If $u = 0$, then it changes linearly, except when $\dot{q} = 0$; in this case, no motion occurs.

The neighborhood structure is not the same as those in Section 5.4.2 because of drift. For $u = 0$, imagine having a stack of horizontal conveyor belts that carry points to the right if they are above the q-axis, and to the left if they are below it (see Figure 14.12b). The speed of the conveyor belt is given by $\dot{q}$. If $u = 0$, the distance traveled along q is $\dot{q}\Delta t$. This causes horizontal motion to the right in the phase plane if $\dot{q} > 0$ and horizontal motion to the left if $\dot{q} < 0$. Observe in Figure 14.13 that larger motions result as $|\dot{q}|$ increases. If $\dot{q} = 0$, then no horizontal motion can occur. If $q \neq 0$, then the $\dot{q}$ coordinate changes by $\pm\frac{1}{2}u(\Delta t)^2$. This slowing down or speeding up also affects the position along q.

For most realistic problems, there is an upper bound on speed. Let $v_{max} > 0$ be a positive constant and assume that $|\dot{q}| \leq v_{max}$. Furthermore, assume that $\mathcal{C}$ is bounded (all values of $q \in \mathcal{C}$ are contained in an interval of $\mathbb{R}$). Since the reachability graph is a lattice and the states are now confined to a bounded subset



(a) Backward reachable set from x_G (b) Forward reachable set from x_I

Figure 14.14: The initial and goal states can be connected to lattice points that call within cones in X that represent time-limited reachable sets.

of $\mathbb{R}^2$, the number of vertices in the reachability graph is finite. For any fixed Δt, the lattice can be searched using any of the algorithms of Section 2.2. The search starts on a reachability graph for which the initial vertex is x_I. Trajectories that are approximately time-optimal can be obtained by using breadth-first search (Dijkstra's algorithm could alternatively be used, but it is more expensive). Resolution completeness can be obtained by reducing Δt by a constant factor each time the search fails to find a solution. As mentioned in Section 5.4.2, it is not required to construct an entire grid resolution at once. Samples can be gradually added, and the connectivity can be updated efficiently using the union-find algorithm [243, 823]. A rigorous approximation algorithm framework will be presented shortly, which indicates how close the solution is to optimal, expressed in terms of input parameters to the algorithm.

Recall the problem of connecting to grid points, which was illustrated in Figure 5.14b. If the goal region X_G contains lattice points, then exact arrival at the goal occurs. If it does not contain lattice points, as in the common case of X_G being a single point, then some additional work is needed to connect a goal state to a nearby lattice point. This actually corresponds to a BVP, but it is easy to solve for the double integrator. The set of states that can be reached from some state x_G within time Δt lie within a cone, as shown in Figure 14.14a. Lattice points that fall into the cone can be easily connected to x_G by applying a constant action in U. Likewise, x_I does not even have to coincide with a lattice point. Thus, it is straightforward to connect x_I to a lattice point, obtain a trajectory that arrives at a lattice point near x_G, and then connect it exactly to x_G.

14.4.1.2 Extensions and other considerations

Alternative lattices for the double integrator Many alternative lattices can be constructed over X. Different discretizations of U and time can be used. Great flexibility is allowed if feasibility is the only concern, as opposed to optimality. Since $\mathcal{C} = \mathbb{R}$, it is difficult to define an obstacle avoidance problem; however, the concepts will be soon generalized to higher dimensions. In this case, finding a

feasible trajectory that connects from some initial state to a goal state may be the main concern. Note, however, that if x_I and x_G are states with zero velocity, then the state could hover around close to the q-axis, and the speeds will be so slow that momentum is insignificant. This provides some incentive for at least reducing the travel time as much as possible, even if the final result is not optimal. Alternatively, the initial and goal states may not have zero velocity, in which case, any feasible solution may be desired. For example, suppose the goal is to topple a sports utility vehicle (SUV) as part of safety analysis.

To get a feeling for how to construct lattices, recall again the analogy to conveyor belts. A lattice can be designed by placing horizontal rows of sample points at various values of $\dot{q}$. These could, for example, be evenly spaced in the $\dot{q}$ direction as in Figure 14.13. Imagine the state lies on a conveyor belt. If desired, a move can be made to any other conveyor belt, say at $\dot{q}'$, by applying a nonzero action for some specific amount of time. If $\dot{q}' > \dot{q}$, then $u > 0$; otherwise, $u < 0$. If the action is constant, then after time $|\dot{q} - \dot{q}'|/u$ has passed, the state will arrive at $\dot{q}'$. Upon arrival, the position q on the conveyor belt might not coincide with a sample point. This is no problem because the action $u = 0$ can be applied until the state drifts to the next sample point. An alternative is to choose an action from U that drives directly to a lattice point within its forward, time-limited reachable set. Recall Figure 14.14; the cone can be placed on a lattice point to locate other lattice points that can be reached by application of a constant action in U over some time interval.

Recall from Figure 14.13 that longer distances are traveled over time Δt as $|\dot{q}|$ increases. This may be undesirable behavior in practice because the resolution is essentially much poorer at higher speeds. This can be compensated for by placing the conveyor belts closer together as $|\dot{q}|$ increases. As the speed increases, a shorter time interval is needed to change belts, and the distance traveled can be held roughly the same for all levels. This corresponds to the intuition that faster response times are needed at higher speeds.

A multi-resolution version can also be made [816]. The simple problem considered so far can actually be solved combinatorially, without any approximation error [747]; however, the lattice-based approach was covered because it can be extended to much harder problems, as will be explained next.

Multiple, independent double integrators Now consider generalizing to a vector of n double integrators. In this case, $\mathcal{C} = \mathbb{R}^n$ and each $q \in \mathcal{C}$ is an n-dimensional vector. There are n action variables and n double integrators of the form $\ddot{q}_i = u_i$. The action space for each variable is $U_i = [-1, 1]$ (once again, any acceleration bound can be used). The phase space X is $\mathbb{R}^{2n}$, and each point is $x = (q_1, \ldots, q_n, \dot{q}_1, \ldots, \dot{q}_n)$. The ith double integrator produces two scalar equations of the phase transition equation: $\dot{x}_i = x_{n+i}$ and $\dot{x}_{n+i} = u_i$.

Even though there are n double integrators, they are decoupled in the state transition equation. The phase of one integrator does not depend on the phase of another. Therefore, the ideas expressed so far can be extended in a straightforward

way to obtain a lattice over $\mathbb{R}^{2n}$. Each action is an n-dimensional vector u. Each U_i is discretized to yield values -1, 0, and 1. There are 3^n edges emanating from any lattice point for which $\dot{q}_i \neq 0$ for all i. For any double integrator for which $\dot{q}_i = 0$, there are only two choices because $u_i = 0$ produces no motion. The projection of the reachability graph down to (x_i, x_{n+i}) for any i from 1 to n looks exactly like Figure 14.13 and characterizes the behavior of the ith integrator.

The standard search algorithms can be applied to the lattice over $\mathbb{R}^{2n}$. Breadth-first search once again yields solutions that are approximately time-optimal. Resolution completeness can be obtained again by bounding X and allowing Δt to converge to zero. Now that there are more dimensions, a complicated obstacle region X_{obs} can be removed from X. The traversal of each edge then requires collision detection along each edge of the graph. Note that the state trajectories are linear or parabolic arcs. Numerical integration is not needed because (14.22) already gives the closed-form expression for the state trajectory.

Unconstrained mechanical systems A lattice can even be obtained for the general case of a fully actuated mechanical system, which for example includes most robot arms. Recall from (13.4) that any system in the form $\dot{q} = f(q, u)$ can alternatively be expressed as $\dot{q} = u$, if $U(q)$ is defined as the image of f for a fixed q. The main purpose of using f is to make it easy to specify a fixed action space U that maps differently into the tangent space for each $q \in \mathcal{C}$.

A similar observation can be made regarding equations of the form $\ddot{q} = h(q, \dot{q}, u)$, in which $u \in U$ and U is an open subset of $\mathbb{R}^n$. Recall that this form was obtained for general unconstrained mechanical systems in Sections 13.3 and 13.4. For example, (13.148) expresses the dynamics of open-chain robot arms. Such equations can be expressed as $\ddot{q} = u'$ by directly specifying the set of allowable accelerations. Each u will map to a new action u' in an action space given by

$$U'(q, \dot{q}) = \{\ddot{q} \in \mathbb{R}^n \mid \exists u \in U \text{ such that } \ddot{q} = h(q, \dot{q}, u)\} \tag{14.24}$$

for each $q \in \mathcal{C}$ and $\dot{q} \in \mathbb{R}^n$.

Each $u' \in U'(q, \dot{q})$ directly expresses an acceleration vector in $\mathbb{R}^n$. Therefore, using $u' \in U(q, \dot{q})$, the original equation expressed using h can be now written as $\ddot{q} = u'$. In its new form, this appears just like the multiple, independent double integrators. The main differences are

1. The set $U'(q, \dot{q})$ may describe a complicated region in $\mathbb{R}^n$, whereas U in the case of the true double integrators was a cube centered at the origin.

2. The set $U'(q, \dot{q})$ varies with respect to q and $\dot{q}$. Special concern must be given for this variation over the time sampling interval Δt. In the case of the true double integrators, U was fixed.

The first difference is handled by performing grid sampling over $\mathbb{R}^n$ and making an edge in the reachability graph for every grid point that falls into $U'(q, \dot{q})$; see Figure 14.15a. The grid resolution can be improved along with Δt to obtain

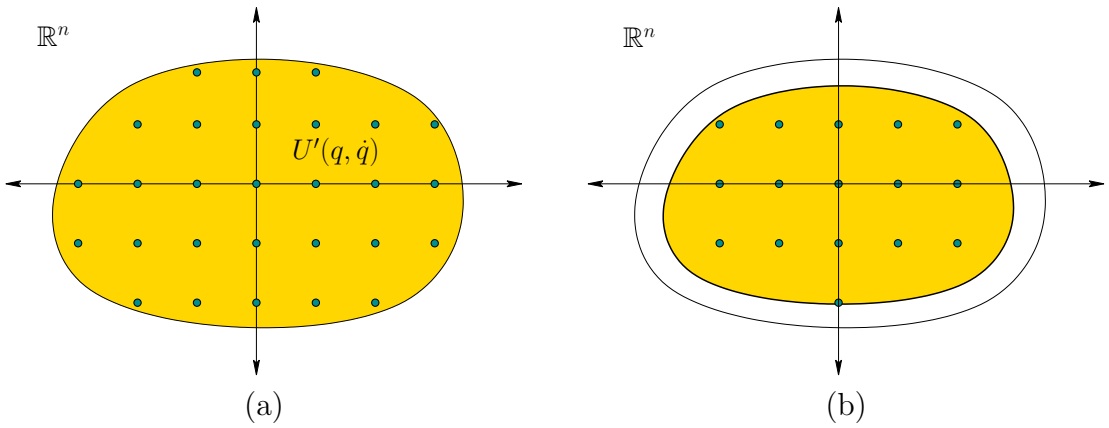


Figure 14.15: (a) The set, $U'(q, \dot{q})$, of new actions and grid-based sampling. (b) Reducing the set by some safety margin to account for its variation over time.

resolution completeness. To address the second problem, think of $U'(q(t), \dot{q}(t))$ as a shape in $\mathbb{R}^n$ that moves over time. Choosing u' close to the boundary of $U'(q(t), \dot{q}(t))$ is dangerous because as t increases, u' may fall outside of the new action set. It is often possible to obtain bounds on how quickly the boundary of $U'(q, \dot{q})$ can vary over time (this can be determined, for example, by differentiating h with respect to q and $\dot{q}$). Based on the bound, a thin layer near the boundary of $U'(q, \dot{q})$ can be removed from consideration to ensure that all attempted actions remain in $U'(q(t), \dot{q}(t))$ during the whole interval Δt. See Figure 14.15b.

These ideas were applied to extend the approximation algorithm framework to the case of open-chain robot arms, for which h is given by (13.148). Suppose that U is an axis-aligned rectangle, which is often the case for manipulators because the bounds for each u_i correspond to torque limits for each motor. If q and $\dot{q}$ are fixed, then (13.140) applies a linear transformation to obtain $\ddot{q}$ from u. The rectangle is generally sheared into a parallelepiped (a n-dimensional extension of a parallelogram). Recall such transformations from Section 3.5 or linear algebra.

Approximation algorithm framework The lattices developed in this section were introduced in [290] for analyzing the kinodynamic planning problem in the rigorous *approximation algorithm* framework for NP-hard problems [765]. Suppose that there are two or three independent double integrators. The analysis shows that the computed solutions are approximately optimal in the following sense. Let c_0 and c_1 be two positive constants that define a function

$$\delta(c_0, c_1)(\dot{q}) = c_0 + c_1 \|\dot{q}\|. \tag{14.25}$$

Let t_F denote the time at which the termination action is applied. A state trajectory is called $\delta(c_0, c_1)$-*safe* if for all $t \in [0, t_F]$, the ball of radius $\delta(c_0, c_1)(\dot{q})$ that is centered at $q(t)$ does not cause collisions with obstacles in $\mathcal{W}$. Note that the ball radius grows linearly as the speed increases. The robot can be imagined

as a disk with a radius determined by speed. Let x_I, x_G, c_0, and c_1 be given (only a point goal region is allowed). Suppose that for a given problem, there exists a $\delta(c_0, c_1)$-safe state trajectory (resulting from integrating any $\tilde{u} \in \mathcal{U}$) that terminates in x_G after time t_{opt}. It was shown that by choosing the appropriate Δt (given by a formula in [290]), applying breadth-first search to the reachability lattice will find a $(1-\epsilon)\delta(c_0, c_1)$-safe trajectory that takes time at most $(1+\epsilon)t_{opt}$, and approximately connects x_I to x_G (which means that the closeness in X depends on ϵ). Furthermore, the running time of the algorithm is polynomial in $1/\epsilon$ and the number of primitives used to define polygonal obstacles.[5] One of the key steps in the analysis is to show that any state trajectory can be closely tracked using only actions from U_d and keeping them constant over Δt. One important aspect is that it does not necessarily imply that the computed solution is close to the true optimum, as it travels through X (only the execution times are close). Thus, the algorithm may give a solution from a different homotopy class from the one that contains the true optimal trajectory. The analysis was extended to the general case of open-chain robot arms in [288, 441].

Backward and bidirectional versions There is a perfect symmetry to the concepts presented so far in this section. A reachability lattice similar to the one in Figure 14.13 can be obtained by integrating backward in time. This indicates action sequences and associated initial states from which a fixed state can be reached. Note that applying the ideas in the reverse direction does not require the system to be symmetric. Given that the graphs exist in both directions, bidirectional search can be performed. By using the forward and backward time-limited reachability cones, the initial and goal states can be connected to a common lattice, which is started, for example, at the origin.

Underactuated and nonholonomic systems Many interesting systems cannot be expressed in the form $\ddot{q} = h(q, \dot{q}, u)$ with n independent action variables because of underactuation or other constraints. For example, the models in Section 13.1.2 are underactuated and nonholonomic. In this case, it is not straightforward to convert the equations into a vector of double integrators because the dimension of $U(q, \dot{q})$ is less than n, the dimension of $\mathcal{C}$. This makes it impossible to use grid-based sampling of $U(q, \dot{q})$. Nevertheless, it is still possible in many cases to discretize the system in a clever way to obtain a lattice. If this can be obtained, then a straightforward resolution-complete approach based on classical search algorithms can be developed. If X is bounded (or a bounded region is obtained after applying the phase constraints), then the search is performed on a finite graph. If failure occurs, then the resolution can be improved in the usual way to eventually obtain resolution completeness. As stated in Section 14.2.2, obtaining such a lat-

[5] One technical point: It is actually only pseudopolynomial [765] in a_{max}, v_{max}, c_0, c_1, and the width of the bounding cube in $\mathcal{W}$. This means that the running time is polynomial if the representations of these parameters are treated as having constant size; however, it is not polynomial in the actual number of bits needed to truly represent them.

tice is possible for a large family of nonholonomic systems [762]. Next, a method is presented for handling reachability graphs that are not lattices.

14.4.2 Incorporating State Space Discretization

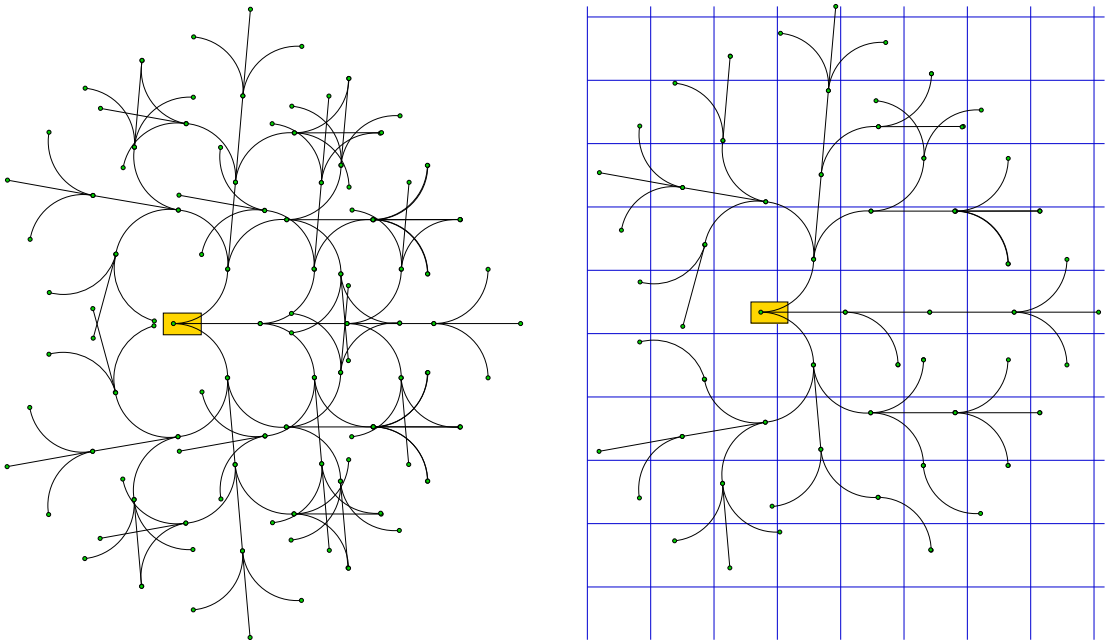
If the reachability graph is not a lattice, which is typically the case with underactuated and nonholonomic systems, then state space discretization can be used to force it to behave like a lattice. If there are no differential constraints, then paths can be easily forced to travel along a lattice, as in the methods of Section 7.7.1. Under differential constraints, the state cannot be forced, for example, to follow a staircase path. Instead of sampling X and forcing trajectories to visit specific points, X can be partitioned into small cells, within which no more than one vertex is allowed in the search graph. This prevents a systematic search algorithm from running forever if the search graph has an infinite number of vertices in some bounded region. For example, with the Dubins car, if u is fixed to an integer, an infinite number of vertices on a circle is obtained, as mentioned in Section 14.2.2. The ideas in this section are inspired mainly by the Barraquand-Latombe dynamic programming method [73], which has been mainly applied to the models in Section 13.1.2. In the current presentation, however, the approach is substantially generalized. Here, optimality is not even necessarily required (but can be imposed, if desired).

Decomposing X into cells At the outset, X is decomposed into a collection of cells without considering collision detection. Suppose that X is an n-dimensional rectangular subset of $\mathbb{R}^n$. If X is more generally a smooth manifold, then the rectangular subset can be defined in a coordinate neighborhood. If desired, identifications can be used to respect the topology of X; however, coordinate changes are technically needed at the boundaries to properly express velocities (recall Section 8.3).

The most common cell decomposition is obtained by splitting X into n-dimensional cubes of equal size by quantizing each coordinate. This will be called a *cubical partition*. Assume in general that X is partitioned into a collection $\mathcal{D}$ of n-dimensional cells. Let $D \in \mathcal{D}$ denote a *cell*, which is a subset of X. It is assumed here that all cells have dimension n. In the case of cubes, this means that points on common boundaries between cubes are declared to belong to only one neighboring cube (thus, the cells may be open, closed, or neither).

Note that X is partitioned into cells, and not X_{free}, as might be expected from the methods in Chapter 6. This means that collision detection and other constraints on X are ignored when defining $\mathcal{D}$. The cells are defined in advance, just as grids were declared in Section 5.4.2. In the case of a cubical partition, the cells are immediately known upon quantization of each coordinate axis.

Searching The algorithm fits directly into the framework of Section 14.3.4. A search graph is constructed incrementally from x_I by applying any systematic



(a) (b)

Figure 14.16: (a) The first four stages of a dense reachability graph for the Dubins car. (b) One possible search graph, obtained by allowing at most one vertex per cell. Many branches are pruned away. In this simple example, there are no cell divisions along the θ-axis.

search algorithm. It is assumed that the system has been discretized in some way. Most often, the discrete-time model of Section 14.2.2 is used, which results in a fixed Δt and a finite set U_d of actions.

In the basic search algorithms of Section 2.2.1, it is important to keep track of which vertices have been explored. Instead of applying this idea to vertices, it is applied here to cells. A cell D is called *visited* if the search graph that has been constructed so far contains a vertex in D; otherwise, D is called *unvisited*. Initially, only the cell that contains x_I is marked as *visited*. All others are initialized to *unvisited*. These labels are used to prune the reachability graph during the search, as shown in Figure 14.16.

The basic algorithm outline is shown in Figure 14.17. Let Q represent a priority queue in which the elements are vertices of the search graph. If optimization of a cost functional is required, then Q may be sorted by the cost accumulated along the path constructed so far from x_I to x. This cost can be assigned in many different ways. It could simply represent the time (number of Δt steps), or it could count the number of times the action has changed. As the algorithm explores, new candidate vertices are encountered. They are only saved in the search graph and placed into Q if they lie in a cell marked *unvisited* and are violation-free. Upon encountering such a cell, it becomes marked as *visited*. The REACHED function generates a set

```
CELL-BASED_SEARCH(x_I, x_G)
 1   Q.insert(x_I);
 2   G.init(x_I);
 3   while Q ≠ ∅ and x_G is unvisited
 4       x_cur → Q.pop();
 5       for each (ũ_t, x) ∈ REACHED(x_cur)
 6           if x is unvisited
 7               Q.insert(x);
 8               G.add_vertex(x);
 9               G.add_edge(ũ_t);
10               Mark cell that contains x as visited;
11   Return G;
```

Figure 14.17: Searching by using a cell decomposition of X.

of violation-free trajectory segments. Under the discrete-time model, this means applying each $u \in U_d$ over time Δt and reporting only those states reached without violating the constraints (including collision avoidance).

As usual, the BVP issue may arise if X_G is small relative to the cell size. If X_G is large enough to include entire cells, then this issue is avoided. If x_G is a single point, then it may only be possible to approximately reach x_G. Therefore, the algorithm must accept reaching x_G to within a specified tolerance. This can be modeled by defining X_G to be larger; therefore, tolerance is not explicitly mentioned.

Maintaining the cells There are several alternatives for maintaining the cells. The main operation that needs to be performed efficiently is *point location* [264]: determine which cell contains a given state. The original method in [73] preallocates an n-dimensional array. The collision-checking is even performed in advance. Any cell that contains at least one point in X_{obs} can be labeled as *occupied*. This allows cells that contain collision configurations to be avoided without having to call the collision detection module. For a fixed dimension, this scheme finds the correct cell and updates the labels in constant time. Unfortunately, the space requirement is exponential in dimension.

An alternative is to use a hash table to maintain the collection of cells that are labeled as *visited*. This may be particularly valuable if optimality is not important and if it is expected that solutions will be found before most of the cells are reached. The point location problem can be solved efficiently without explicitly storing a multi-dimensional array.

Suppose that the cubical decomposition is not necessarily used. One general approach is to define $\mathcal{D}$ as the Voronoi regions of a collection P of m samples $\{p_1, \ldots, p_m\}$ in X. The "name" of each cell corresponds uniquely to a sample in P. The cell that contains some $x \in X$ is defined as the nearest sample in P, using some

predetermined metric on X. As a special case, the cubical decomposition defines the cells based on a Sukharev grid (recall Figure 5.5a). If the dimension of X is not too high, then efficient nearest-neighbor schemes can be used to determine the appropriate cell in near-logarithmic time in the number of points in P (in practice, Kd-trees, mentioned in Section 5.5.2, should perform well). For maintaining a cubical decomposition, this approach would be cumbersome; however, it works for *any* sample set P. If no solution is found for a given P, then the partition could be improved by adding more samples. This allows any dense sequence to be used to guide the exploration of X while ensuring resolution completeness, which is discussed next.

Resolution issues One of the main complications in using state discretization is that there are three spaces over which sampling occurs: time, the action space, and the state space. Assume the discrete-time model is used. If obtaining optimal solutions is important, then very small cells should be used (e.g., 50 to 100 per axis). This limits its application to state spaces of a few dimensions. The time interval Δt should also be made small, but if it is too small relative to the cell size, then it may be impossible to leave a cell. If only feasibility is the only requirement, then larger cells may be used, and Δt must be appropriately increased. A course quantization of U may cause solutions to be missed, particularly if Δt is large. As Δt decreases, the number of samples in U_d becomes less important.

To obtain resolution completeness, the sampling should be improved each time the search fails. Each time that the search is started, the sampling dispersion for at least one of the three spaces should be decreased. The possibilities are 1) the time interval Δt may be reduced, 2) more actions may be added to U_d, or 3) more points may be added to P to reduce the cell size. If the dispersion approaches zero for all three spaces, and if X_G contains an open subset of X_{free}, then resolution completeness is obtained. If X_G is only a point, then solutions that come within some $\epsilon > 0$ must be tolerated.

Recall that resolution completeness assumes that f has bounded derivatives or at least satisfies a Lipschitz condition (14.11). The actual rate of convergence is mainly affected by three factors: 1) the rate at which f varies with respect to changes in u and x (characterized by Lipschitz constants), 2) the required traversal of narrow regions in X_{free}, and 3) the controllability of the system. The last condition will be studied further for nonholonomic systems in Section 15.4. For a concrete example, consider making a U-turn with a Dubins car that has a very large turning radius, as shown in Figure 14.18. A precise turn may be required to turn around, and this may depend on an action that was chosen many stages earlier. The Dubins car model does not allow zig-zagging (e.g., parallel parking) maneuvers to make local corrections to the configuration.

Backward and bidirectional versions As usual, both backward and bidirectional versions of this approach can be made. If the X_G is large (or the goal tolerance is large) and the BVP is costly to solve, then the backward version seems

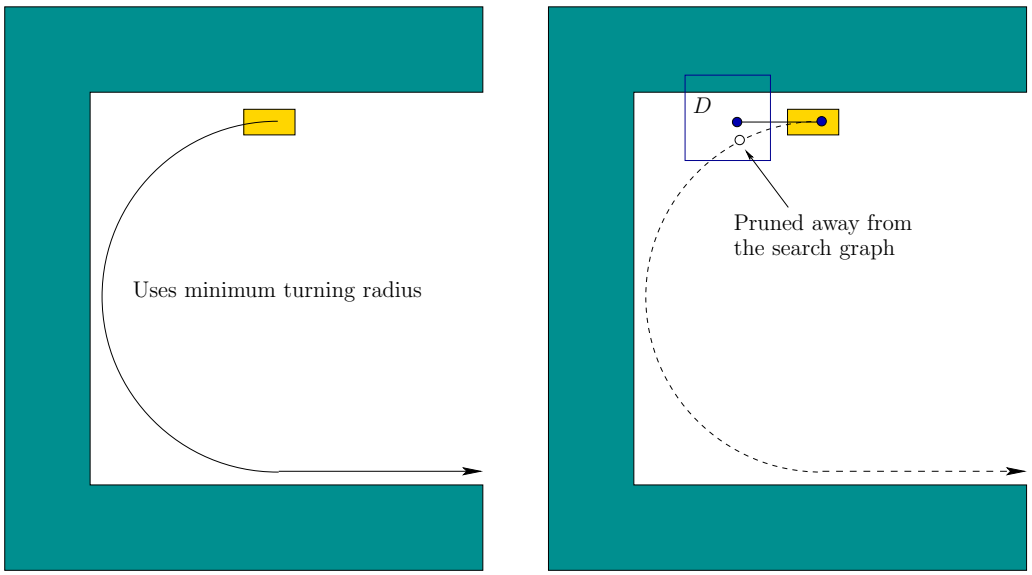


Figure 14.18: (a) The Dubins car is able to turn around if it turns left as sharply as possible. (b) Unfortunately, the required vertex is pruned because one cell along the required trajectory already contains a vertex. This illustrates how missing a possible action can cause serious problems many stages later.

less desirable if the BVP is hard. The forward direction is preferred because the BVP can be avoided altogether.

For a bidirectional algorithm, the same collection $\mathcal{D}$ of cells can be used for both trees. The problem could be considered solved if the same cell is reached by both trees; however, one must be careful to still ensure that the remaining BVP can be solved. It must be possible to find an action trajectory segment that connects a vertex from the initial-based tree to a vertex of the goal-based tree. Alternatively, connections made to within a tolerance may be acceptable.

14.4.3 RDT-Based Methods

The rapidly exploring dense tree (RDT) family of methods, which includes the RRT, avoids maintaining a lattice altogether. RDTs were originally developed for handling differential constraints, even though most of their practical application has been to the Piano Mover's Problem. This section extends the ideas of Section 5.5 from $\mathcal{C}$ to X and incorporates differential constraints. The methods covered so far in Section 14.4 produce approximately optimal solutions if the graph is searched using dynamic programming and the resolution is high enough. By contrast, RDTs are aimed at returning only feasible trajectories, even as the resolution improves. They are often successful at producing a solution trajectory with relatively less sampling. This performance gain is enabled in part by the lack of concern for

SIMPLE_RDT_WITH_DIFFERENTIAL_CONSTRAINTS(x_0)

```
1  G.init(x_0);
2  for i = 1 to k do
3      x_n ← NEAREST(S(G), α(i));
4      (ũ^p, x_r) ← LOCAL_PLANNER(x_n, α(i));
5      G.add_vertex(x_r);
6      G.add_edge(ũ^p);
```

Figure 14.19: Extending the basic RDT algorithm to handle differential constraints. In comparison to Figure 5.16, an LPM computes x_r, which becomes the new vertex, instead of $\alpha(i)$. In some applications, line 4 may fail, in which case lines 5 and 6 are skipped.

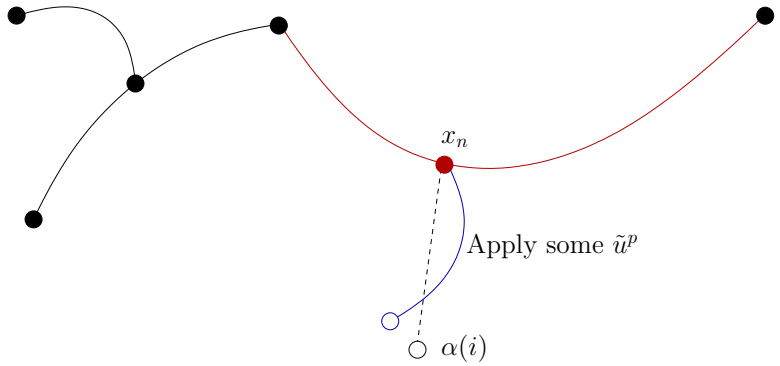


Figure 14.20: If the nearest point S lies in the state trajectory segment associated to an edge, then the edge is split into two, and a new vertex is inserted into $\mathcal{G}$.

optimality.

Let α denote an infinite, dense sequence of samples in X. Let $\rho : X \times X \rightarrow [0, \infty]$ denote a distance function on X, which may or may not be a proper metric. The distance function may not be symmetric, in which case $\rho(x_1, x_2)$ represents the directed distance from x_1 to x_2.

The RDT is a search graph as considered so far in this section and can hence be interpreted as a subgraph of the reachability graph under some discretization model. For simplicity, first assume that the discrete-time model of Section 14.2.2 is used, which leads to a finite action set U_d and a fixed time interval Δt. The set $\mathcal{U}^p$ of motion primitives is all action trajectories for which some $u \in U_d$ is held constant from time 0 to Δt. The more general case will be handled at the end of this section.

Paralleling Section 5.5.1, the RDT will first be defined in the absence of obstacles. Hence, let $X_{free} = X$. The construction algorithm is defined in Figure 14.19; it may be helpful to compare it to Figure 5.16, which was introduced on $\mathcal{C}$ for the Piano Mover's Problem. The RDT, denoted by $\mathcal{G}$, is initialized with a single vertex at some $x_0 \in X$. In each iteration, a new edge and vertex are added

to $\mathcal{G}$. Line 3 uses ρ to choose x_n, which is the nearest point to $\alpha(i)$ in the swath of $\mathcal{G}$. In the RDT algorithm of Section 5.5, each sample of α becomes a vertex. Due to the BVP and the particular motion primitives in $\mathcal{U}^p$, it may be difficult or impossible to precisely reach $\alpha(i)$. Therefore, line 4 calls an LPM to determine a primitive $\tilde{u}^p \in \mathcal{U}^p$ that produces a new state x_r upon integration from x_n. The result is depicted in Figure 14.20. For the default case in which $\mathcal{U}^p$ represents the discrete-time model, the action is chosen by applying all $u \in U$ over time Δt and selecting the one that minimizes $\rho(x_r, \alpha(i))$. One additional constraint is that if x_n has been chosen in a previous iteration, then $\tilde{u}^p$ must be a motion primitive that has not been previously tried from x_n; otherwise, duplicate edges would result in $\mathcal{G}$ or time would be wasted performing collision checking for reachability graph edges that are already known to be in collision. The remaining steps add the new vertex and edge from x_n. If x_n is contained in the trajectory produced by an edge e, then e is split as described in Section 5.5.1.

Efficiently finding nearest points The issues of Section 5.5.2 arise again for RDTs under differential constraints. In fact, the problem is further complicated because the edges in $\mathcal{G}$ are generally curved. This prevents the use of simple point-segment distance tests. Furthermore, an exact representation of the state trajectory is usually not known. Instead, it is approximated numerically by the system simulator. For these reasons, it is best to use the approximate method of determining the nearest point in the swath, which is a straightforward extension of the discussion in Section 5.5.2; recall Figure 5.22. Intermediate vertices may be inserted if the applied motion primitive yields a state trajectory that travels far in X_{free}. If the dimension is low enough (e.g., less than 20), then efficient nearest-neighbor algorithms (Section 5.5.2) can be used to offset the cost of maintaining intermediate vertices.

Handling obstacles Now suppose that $X_{obs} \neq \emptyset$. In Section 5.5.1, the RDT was extended until a stopping configuration q_s was reached, just in front of an obstacle. There are two new complications under differential constraints. The first is that motion primitives are used. If Δt is small, then in many cases the time will expire before the boundary is reached. This can be alleviated by using a large Δt and then taking only the violation-free portion of the trajectory. In this case, the trajectory may even be clipped early to avoid overshooting $\alpha(i)$. The second complication is due to X_{ric}. If momentum is substantial, then pulling the tree as close as possible to obstacles will increase the likelihood that the RDT becomes trapped. Vertices close to obstacles will be selected often because they have large Voronoi regions, but expansion is not possible. In the case of the Piano Mover's Problem, this was much less significant because the tree could easily follow along the boundary. In most experimental work, it therefore seems best to travel only part of the way (perhaps half) to the boundary.

Tree-based planners Planning algorithms can be constructed from RDTs in the same way as in Section 5.5. Forward, backward, and bidirectional versions can be made. The main new complication is the familiar BVP that the other sampling-based methods of this section have also suffered from. If it is expensive or even impossible to connect nearby states, then the usual complications arise. If X_G contains a sizable open set, then a forward, single-tree planner with a gentle bias toward the goal could perform well while avoiding the BVP. However, if X_G is a point, then a tolerance must be set on how close the RDT must get to the goal before it can declare that it has a solution. For systems with drift, the search time often increases dramatically as this tolerance decreases.

Bidirectional search offers great performance advantages in many cases, but the BVP exists when attempting connections between the two trees. One possibility is to set the tolerance very small and then concatenate the two action trajectories, as described in Section 14.3.4. If it succeeds, then the planning algorithm successfully terminates. Unfortunately, the performance once again depends greatly on the tolerance, particularly if the drift is substantial. Recent studies have shown that using a bidirectional RDT with a large connection tolerance and then closing the gap by using efficient variational techniques provides dramatic improvement in performance [198, 576]. Unfortunately, variational techniques are not efficient for all systems because they must essentially solve the BVP by performing a gradient descent in the trajectory space; see Section 14.7.

Distance function issues The RDT construction algorithm is heavily influenced by the distance function ρ. This was also true for RDTs applied to the Piano Mover's Problem; however, it becomes more critical and challenging to design a good metric in the presence of differential constraints. For example, the metric given by Example 5.3 is inappropriate for measuring the distance between configurations for the Dubins car. A more appropriate metric is to use length of the shortest path from q to q' (this length is easy to compute; see Section 15.5). Such a metric would be more appropriate than the one in Example 5.3 for comparing the configurations, even for car models that involve dynamics and obstacles.

Although many challenging problems can be solved using weighted Euclidean metrics [611], dramatic improvements can be obtained by exploiting particular properties of the system. This problem might seem similar to the choice of a potential function for the randomized potential field planer of Section 5.4.3; however, since RDTs approach many different samples in $\alpha(i)$, instead of focusing only on the goal, the performance degradation is generally not as severe as the local minimum problem for a potential field planner. There are many more opportunities to escape in an RDT. Metrics that would fail miserably as a potential function often yield good performance in an RDT-based planner.

The ideal distance function, as mentioned in Section 14.3, is to use the optimal cost-to-go, denoted here as ρ^*. Of course, computing ρ^* is at least as hard as solving the motion planning problem. Therefore, this idea does not seem practical. However, it is generally useful to consider ρ^* because the performance of RDT-

based planners generally degrades as ρ, the actual metric used in the RDT, and ρ^* diverge. An effort to make a crude approximation to ρ^*, even if obstacles are neglected, often leads to great improvements in performance. An excellent example of this appears in [363], in which value iteration was used to compute the optimal cost-to-go in the absence of obstacles for an autonomous helicopter using the maneuver automaton model of Figure 14.8.

Ensuring resolution completeness Suppose that the discrete-time model is used. If α is dense in X, then each RDT vertex is visited a countably infinite number of times after it is constructed. By ensuring that the same motion primitive is never applied twice from the same vertex, all available motion primitives will eventually be tried. This ensures that the full reachability graph is explored for a fixed Δt. Since the reachability graph is not necessarily finite, obtaining resolution completeness is more challenging. The scheme described in Figure 14.7 can be applied by periodically varying Δt during execution, and using smaller and smaller of values of Δt in later iterations. If U is finite, refinements can also be made to U_d. This leads to a resolution-complete RDT.

Designing good motion primitives Up to this point, only the discrete-time model has been considered. Although it is the most straightforward and general, there are often many better motion primitives that can be used. For a particular system, it may be possible to design a nice family of trajectories off-line in the absence of obstacles and then use them as motion primitives in the RDT construction. If possible, it is important to carefully analyze the system under consideration to try to exploit any special structure it may have or techniques that might have been developed for it. For motion planning of a vehicle, symmetries can be exploited to apply the primitives from different states. For example, in flying a helicopter, the yaw angle and the particular position (unless it is close to the ground) may not be important. A family of trajectories designed for one yaw angle and position should work well for others.

Using more complicated motion primitives may increase the burden on the LPM. In some cases, a simple control law (e.g., PID [50]) may perform well. Ideally, the LPM should behave like a good steering method, which could be obtained using methods in Chapter 15. It is important to note, though, that the RDT's ability to solve problems does not hinge on this. It will greatly improve performance if there are excellent motion primitives and a good steering method in the LPM. The main reason for this is that the difficulties of the differential constraints have essentially been overcome once this happens (except for the adverse effects of drift). Although having good motion primitives can often improve performance in practice, it can also increase the difficulty of ensuring resolution completeness.

14.4.4 Other Methods

Extensions of virtually any other method in Chapter 5 can be made to handle differential constraints. Several possibilities are briefly mentioned in this section.

Randomized potential fields The randomized potential field method of Section 5.4.3 can be easily adapted to handle differential constraints. Instead of moving in any direction to reduce the potential value, motion primitives are applied and integrated to attempt to reduce the value. For example, under the discrete-time model, each $u \in U_d$ can be applied over Δt, and the one for which the next state has the lowest potential value should be selected as part of the descent. Random walks can be tried whenever no such action exists, but once again, motion in any direction is not possible. Random actions can be chosen instead. The main problems with the method under differential constraints are 1) it is extremely challenging to design a good potential function, and 2) random actions do not necessarily provide motions that are similar to those of a random walk. Section 15.1.2 discusses Lyapunov functions, which serve as good potential functions in the presence of differential constraints (but usually neglect obstacles). In the place of random walks, other planning methods, such as an RDT, could be used to try to escape local minima.

Other tree-based planners Many other tree-based planners can be extended to handle differential constraints. For example, an extension of the expansive space planner from Section 5.4.4 to kinodynamic planning for spacecrafts appears in [466]. Recently, a new tree-based method, called the *path-directed subdivision tree*, has been proposed for planning under differential constraints [572]. The method works by choosing points at random in the swath, applying random actions, and also using a space-partition data structure to control the exploration.

Sampling-based roadmap planners As stated already, it is generally difficult to construct sampling-based roadmaps unless the BVP can be efficiently solved. The steering methods of Section 15.5 can serve this purpose [934, 859]. In principle, any of the single-query methods of Section 14.4 could be used; however, it may be too costly to use them numerous times, which is required in the roadmap construction algorithm.

14.5 Feedback Planning Under Differential Constraints

14.5.1 Problem Definition

Formulation 14.1 assumed that feedback is not necessary. If the initial state is given, then the solution takes the form of an action trajectory, which upon inte-

gration yields a time-parametrized path through X_{free}. This extended the Piano Mover's Problem of Section 4.3.1 to include phase spaces and differential constraints. Now suppose that feedback is required. The reasons may be that the initial state is not given or the plan execution might not be predictable due to disturbances or errors in the system model. Recall the motivation from Section 8.1.

With little effort, the feedback motion planning framework from Chapter 8 can be extended to handle differential constraints. Compare Formulations 8.2 and 14.1. Feedback motion planning under differential constraints is obtained by making the following adjustments to Formulation 8.2:

1. In Formulation 8.2, $X = \mathcal{C}_{free}$, which automatically removed $\mathcal{C}_{obs}$ from $\mathcal{C}$ by definition. Now let X be any C-space or phase space, and let X_{obs} be defined as in Formulation 8.2. This leads to X_{free}, as defined in Formulation 14.1.

2. In Formulation 8.2, the state transition equation was $\dot{x} = u$, which directly specified velocities in the tangent space $T_x(X)$. Now let any system, $\dot{x} = f(x, u)$, be used instead. In this case, $U(x)$ is no longer a subset of $T_x(X)$. It still includes the special termination action u_T.

3. Formulation 14.1 includes x_I, which is now removed for the feedback case to be consistent with Formulation 8.2.

4. A feedback plan is now defined as a function $\pi : X_{free} \rightarrow U$. For a given state $x \in X_{free}$, an action $\pi(x)$ is produced. Composing π with f yields a velocity in $T_x(X)$ given by $\dot{x} = f(x, \pi(x))$. Therefore, π defines a vector field on X_{free}.

Let t_F denote the time at which u_T is applied. Both feasible and optimal planning can be defined using a cost functional,

$$L(\tilde{x}_{t_F}, \tilde{u}_{t_F}) = \int_0^{t_F} l(x(t), u(t))dt + l_F(x(t_F)), \tag{14.26}$$

which is identical to that given in Section 8.4.1. This now specifies the problem of feedback motion planning under differential constraints.

The most important difference with respect to Chapter 8 is that $\dot{x} = u$ is replaced with $\dot{x} = f(x, u)$, which allows complicated differential models of Chapter 13 to be used. The vector field that results from π must satisfy the differential constraints imposed by $\dot{x} = f(x, u)$. In Section 8.4.4, simple constraints on the allowable vector fields were imposed, such as velocity bounds or smoothness; however, these constraints were not as severe as the models in Chapter 13. For example, the Dubins car does not allow motions in the reverse direction, whereas the constraints in Section 8.4.4 permit motions in any direction.

14.5.2 Dynamic Programming with Interpolation

As observed in Section 14.4, motion planning under differential constraints is extremely challenging. Additionally requiring feedback complicates the problem even further. If $X_{obs} = \emptyset$, then a feedback plan can be designed using numerous techniques from control theory. See Section 15.2.2 and [192, 523, 846]. In many cases, designing feedback plans is no more difficult than computing an open-loop trajectory. However, if $X_{obs} \neq \emptyset$, feedback usually makes the problem much harder.

Fortunately, dynamic programming once again comes to the rescue. In Section 2.3, value iteration yielded feedback plans for discrete state spaces and state transition equations. It is remarkable that this idea can be generalized to the case in which U and X are continuous and there is a continuum of stages (called time). Most of the tools required to apply dynamic programming in the current setting were already introduced in Section 8.5.2. The main ideas in that section were to represent the optimal cost-to-go G^* by interpolation and to use a discrete-time approximation to the motion planning problem.

The discrete-time model of Section 14.2.2 can be used in the current setting to obtain a discrete-stage state transition equation of the form $x_{k+1} = f_d(x_k, u_k)$. The cost functional is approximated as in Section 8.5.2 by using (8.65). This integral can be evaluated numerically by using the result of the system simulator and yields the cost-per-stage as $l_d(x_k, u_k)$. Using backward value iteration, the dynamic programming recurrence is

$$G_k^*(x_k) = \min_{u_k \in U_d} \Big\{ l_d(x_k, u_k) + G_{k+1}^*(x_{k+1}) \Big\}, \tag{14.27}$$

which is similar to (2.11) and (8.56). The finite set U_d of action samples is used if U is not already finite. The system simulator is applied to determine whether some points along the trajectory lie in X_{obs}. In this case, $l_d(x_k, u_k) = \infty$, which prevents actions from being chosen that violate constraints.

As in Section 8.5.2, a set $P \subset X$ of samples is used to approximate G^* over X. The required values at points in $X \setminus P$ are obtained by interpolation. For example, the barycentric subdivision scheme of Figure 8.20 may be applied here to interpolate over simplexes in $O(n \lg n)$ time, in which n is the dimension of X.

As usual, backward value iteration starts at some final stage F and proceeds backward through the stage indices. Termination occurs when all of the cost-to-go values stabilize. The initialization at stage F yields $G_F^*(x) = 0$ for $x \in X_G \cap P$; otherwise, $G_F^*(x) = \infty$. Each subsequent iteration is performed by evaluating (14.27) on each $x \in P$ and using interpolation to obtain $G_{k+1}^*(x_{k+1})$.

The resulting stationary cost-to-go function G^* can serve as a navigation function over X_{free}, as described in Section 8.5.2. Recall from Chapter 8 that a navigation function is converted into a feedback plan by applying a local operator. The local operator in the present setting is

$$\pi(x) = \operatorname*{argmin}_{u \in U_d} \Big\{ l_d(x, u) + G^*(f_d(x, u)) \Big\}, \tag{14.28}$$

which yields an action for any state in X_{free} that falls into an interpolation neighborhood of some samples in P.

Unfortunately, the method presented here is only useful in spaces of a few dimensions. If $X = \mathcal{C}$, then it may be applied, for example, to the systems in Section 13.1.2. If dynamics are considered, then in many circumstances the dimension is too high because the dimension of X is usually twice that of $\mathcal{C}$. For example, if $\mathcal{A}$ is a rigid body in the plane, then the dimension of X is six, which is already at the usual limit of practical use.

It is interesting to compare the use of dynamic programming here with that of Sections 14.4.1 and 14.4.2, in which a search graph was constructed. If Dijkstra's algorithm is used (or even breadth-first search in the case of time optimality), then by the dynamic programming principle, the resulting solutions are approximately optimal. To ensure convergence, resolution completeness arguments were given based on Lipschitz conditions on f. It was important to allow the resolution to improve as the search failed to find a solution. Instead of computing a search graph, value iteration is based on computing cost-to-go functions. In the same way that both forward and backward versions of the tree-based approaches were possible, both forward and backward value iteration can be used here. Providing resolution completeness is more difficult, however, because x_I is not fixed. It is therefore not known whether some resolution is good enough for the intended application. If x_I is known, then G^* can be used to generate a trajectory from x_I using the system simulator. If the trajectory fails to reach X_G, then the resolution can be improved by adding more samples to P and U_d or by reducing Δt. Under Lipschitz conditions on f, the approach converges to the true optimal cost-to-go [92, 168, 565]. Therefore, value iteration can be considered resolution complete with respect to a given x_I. The convergence even extends to computing optimal feedback plans with additional actions that are taken by nature, which is modeled nondeterministically or probabilistically. This extends the value iteration method of Section 10.6.

The relationship between the methods based on a search graph and on value iteration can be brought even closer by constructing Dijkstra-like versions of value iteration, as described at the end of Section 8.5.2. These extend Dijkstra's algorithm, which was viewed for the finite case in Section 2.3.3 as an improvement to value iteration. The improvement to value iteration is made by recognizing that in most evaluations of (14.27), the cost-to-go value does not change. This is caused by two factors: 1) From some states, no trajectory has yet been found that leads to X_G; therefore, the cost-to-go remains at infinity. 2) The optimal cost-to-go from some state might already be computed; no future iterations would improve the cost.

A forward or backward version of a Dijkstra-like algorithm can be made. Consider the backward case. The notion of a backprojection was used in Section 8.5.2 to characterize the set of states that can reach another set of states in one stage. This was used in (8.68) to define the *frontier* during the execution of the Dijkstra-like algorithm. There is essentially no difference in the current setting to handle

the system $\dot{x} = f(x, u)$. Once the discrete-time approximation has been made, the definition of the backprojection is essentially the same as in (8.66) of Section 8.5.2. Using the discrete-time model of Section 14.2.2, the backprojection of a state $x \in X_{free}$ is

$$\mathrm{B}(x) = \{x' \in X_{free} \mid \exists u \in U_d \text{ such that } x = f_d(x', u)\}. \tag{14.29}$$

The backprojection is closely related to the backward time-limited reachable set from Section 14.2.1. The backprojection can be considered as a discrete, one-stage version, which indicates the states that can reach x through the application of a constant action $u \in U_d$ over time Δt. As mentioned in Section 8.5.2, computing an overapproximation to the frontier set may be preferable in practice. This can be obtained by approximating the backprojections, which are generally more complicated under differential constraints than for the case considered in Section 8.5.2. One useful simplification is to ignore collisions with obstacles in defining $\mathrm{B}(x)$. Also, a simple bounding volume of the true backprojection may be used. The trade-offs are similar to those in collision detection, as mentioned in Section 5.3.2. Sometimes the structure of the particular system greatly helps in determining the backprojections. A nice wavefront propagation algorithm can be obtained, for example, for a double integrator; this is exploited in Section 14.6.3. For more on value iteration and Dijkstra-like versions, see [607].

14.6 Decoupled Planning Approaches

14.6.1 Different Ways to Decouple the Big Problem

As sampling-based algorithms continue to improve along with computation power, it becomes increasingly feasible in practice to directly solve challenging planning problems under differential constraints. There are many situations, however, in which computing such solutions is still too costly due to expensive numerical integration, collision detection, and complicated obstacles in a high-dimensional state space. Decoupled approaches become appealing because they divide the big problem into modules that are each easier to solve. For versions of the Piano Mover's Problem, such methods were already seen in Chapter 7. Section 7.1.3 introduced the velocity-tuning method to handle time-varying obstacles, and Section 7.2.2 presented decoupled approaches to coordinating multiple robots.

Ideally, we would like to obtain feedback plans on any state space in the presence of obstacles and differential constraints. This assumes that the state can be reliably measured during execution. Section 14.5 provided the best generic techniques for solving the problem, but they are unfortunately limited to a few dimensions. If there is substantial sensing uncertainty, then the feedback plan must be defined on the I-space, which was covered in Chapter 11. Back in Section 1.4, Figure 1.19 showed a popular model of decoupling the big planning problem into a sequence of refinements. A typical decoupled approach involves four modules:

1. Use a motion planning algorithm to find a collision-free path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$.

2. Transform τ into a new path τ' so that velocity constraints on $\mathcal{C}$ (if there are any) are satisfied. This might, for example, ensure that the Dubins car can actually follow the path. At the very least, some path-smoothing is needed in most circumstances.

3. Compute a timing function $\sigma : [0, t_F] \rightarrow [0,1]$ for τ' so that $\tau' \circ \sigma$ is a time-parameterized path through $\mathcal{C}_{free}$ with the following requirement. The state trajectory $\tilde{x}$ must satisfy $\dot{x} = f(x(t), u(t))$ and $u(t) \in U(x(t))$ for all time, until u_T is applied at time t_F.

4. Design a feedback plan (or feedback control law) $\pi : X \rightarrow U$ that tracks $\tilde{x}$. The plan should attempt to minimize the error between the desired state and the measured state during execution.

Given recent techniques and computation power, the significance of this approach may diminish somewhat; however, it remains an important way to decompose and solve problems. Be aware, however, that this decomposition is arbitrary. If every module can be solved, then it is sufficient for producing a solution; however, such a decomposition is not necessary. At any step along the way, completeness may be lost because of poor choices in earlier modules. It is often difficult for modules to take into account problems that may arise later.

Various ways to merge the modules have been considered. The methods of Section 14.4 solve either: 1) the first two modules simultaneously, if paths that satisfy $\dot{q} = f(q,u)$ are computed through $\mathcal{C}_{free}$, or 2) the first three modules simultaneously, if paths that satisfy $\dot{x} = f(x,u)$ are computed through X_{free}. Section 14.5 solved all four modules simultaneously but was limited to low-dimensional state spaces.

Now consider keeping the modules separate. Planning methods from Part II can be applied to solve the first module. Section 14.6.2 will cover methods that implement the second module. Section 14.6.3 will cover methods that solve the third module, possibly while also solving the second module. The fourth module is a well-studied control problem that is covered in numerous texts [523, 846, 856].

14.6.2 Plan and Transform

For the decoupled approach in this section, assume that $X = \mathcal{C}$, which means there are only velocity constraints, as covered in Section 13.1. The system may be specified as $\dot{q} = f(q,u)$ or implicitly as a set of constraints of the form $g_i(q, \dot{q}) = 0$. The ideas in this section can easily be extended to phase spaces. The method given here was developed primarily by Laumond (see [596]) and was also applied to the simple car of Section 13.1.2 in [587]; other applications of the method are covered in [596].

PLAN-AND-TRANSFORM APPROACH

1. Compute a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ using a motion planning algorithm, such as one from Part II.

2. Choose some $s_1, s_2 \in [0,1]$ such that $s_1 < s_2$ and use an LPM to attempt to replace the portion of τ from $\tau(s_1)$ to $\tau(s_2)$ with a path γ that satisfies the differential constraints.

3. If τ now satisfies the differential constraints over all $[0,1]$, then the algorithm terminates. Otherwise, go to Step 2.

Figure 14.21: A general outline of the plan-and-transform approach.

An outline of the *plan-and-transform* approach is shown in Figure 14.21. In the first step, a collision-free path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ is computed by ignoring differential constraints. The path is then iteratively modified until it satisfies the constraints. In each iteration, a subinterval $[s_1, s_2] \subseteq [0,1]$ is selected by specifying some $s_1, s_2 \in [0,1]$ so that $s_1 < s_2$. These points may be chosen using random sequences or may be chosen deterministically. The approach may use binary subdivision to refine intervals and gradually improve the resolution on $[0,1]$ over the iterations.

For each chosen interval $[s_1, s_2]$, an LPM is used to compute a path segment $\gamma : [0,1] \rightarrow \mathcal{C}_{free}$ that satisfies the conditions $\gamma(0) = \tau(s_1)$ and $\gamma(1) = \tau(s_2)$. It might be the case that the LPM fails because it cannot connect the two configurations or a collision may occur. In this case, another subinterval is chosen, and the process repeats. Each time the LPM succeeds, τ is updated to τ' as

$$\tau'(s) = \begin{cases} \tau(s) & \text{if } s < s_1 \\ \gamma((s - s_1)/(s_2 - s_1)) & \text{if } s \in [s_1, s_2] \\ \tau(s) & \text{if } s > s_2. \end{cases} \tag{14.30}$$

The argument to γ reparameterizes it to run from s_1 to s_2, instead of 0 to 1.

Example 14.5 (Plan-and-Transform for the Dubins Car) For a concrete example, suppose that the task is to plan a path for the Dubins car. Figure 14.22 shows a path τ that might be computed by a motion planning algorithm that ignores differential constraints. Two sharp corners cannot be traversed by the car. Suppose that s_1 and s_2 are chosen at random, and appear at the locations shown in Figure 14.22. The portion of τ between $\tau(s_1)$ and $\tau(s_2)$ needs to be replaced by a path that can be executed by the Dubins car. Note that matching the orientations at $\tau(s_1)$ and $\tau(s_2)$ is important because they are part of the configuration.

A replacement path γ is shown in Figure 14.23. This is obtained by implementing the following LPM. For the Dubins car, a path between any configurations can be found by drawing circles at the starting and stopping configurations as shown

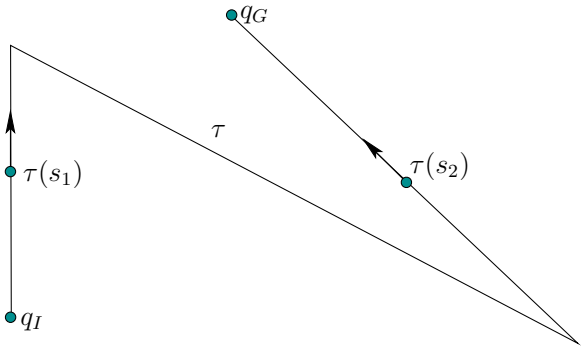


Figure 14.22: An initial path that ignores differential constraints.

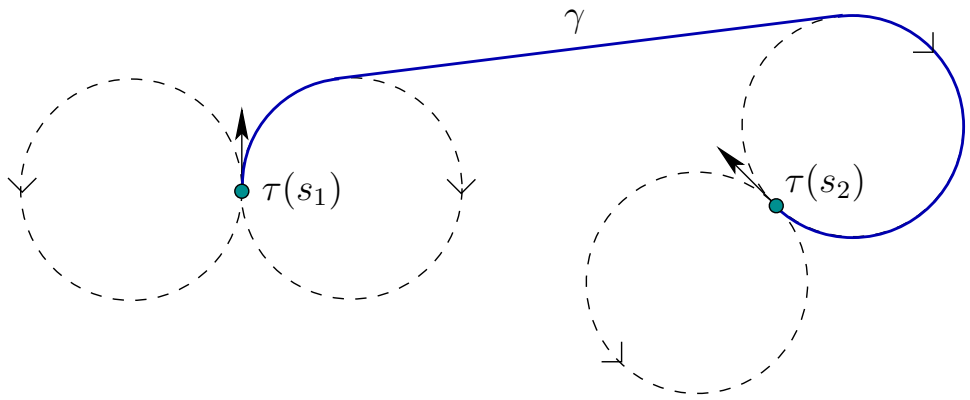


Figure 14.23: A path for the Dubins car can always be found by connecting a bitangent to two circles generated by the minimum turning radius. The path is not necessarily optimal; see Section 15.3.1 for optimal paths.

in the figure. Each circle corresponds to the sharpest possible left turn or right turn. It is straightforward to find a line that is tangent to one circle from each configuration and also matches the direction of flow for the car (the circles are like one-way streets). Using γ, the path τ is updated to obtain τ', which is shown in Figure 14.24, and satisfies the differential constraints for the Dubins car. This problem was very simple, and in practice dozens of iterations may be necessary to replace path segments. Also, if randomization is used, then intervals of the form $[0, s]$ and $[s, 1]$ must not be neglected.

■

Example 14.5 seemed easy because of the existence of a simple local planner. Also, there were no obstacles. Imagine that τ instead traveled through a narrow, zig-zagging corridor. In this case, a solution might not even exist because of sharp corners that cannot be turned by the Dubins car. If there had been an single obstacle that happened to intersect the loop in Figure 14.24, then the replacement would have failed. In general, there is no guarantee that the replacement segment

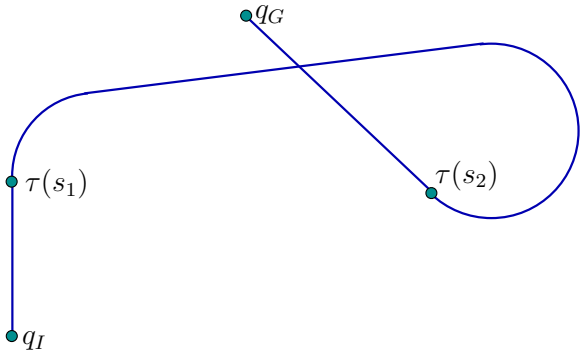


Figure 14.24: Upon replacement, the resulting path τ' can be followed by the Dubins car.

is collision-free. It is important for the LPM to construct path segments that are as close as possible to the original path. For the Dubins car, this is not possible in many cases. For example, moving the Dubins car a small distance backward requires moving along the circles shown in Figure 14.23. Even as the distance between two configurations is reduced, the distance that the car needs to travel does not approach zero. This is true even if the shortest possible paths are used for the Dubins car.

What property should an LPM have to ensure resolution completeness of the plan-and-transform approach? A sufficient condition is given in [596]. Let ρ denote a metric on X. An LPM is said to satisfy the *topological property* if and only if the following statement holds: For any $\epsilon > 0$, there exists some $\delta > 0$ such that for any pair $q, q' \in \mathcal{C}_{free}$ having $\rho(q, q') < \delta$ implies that $\rho(\tau(s), q) < \epsilon$ for all $s \in [0, 1]$. If an LPM satisfies the topological property, then any collision-free path through $\mathcal{C}_{free}$ can be transformed into one that satisfies the differential constraints. Suppose that a path τ has some clearance of at least ϵ in $\mathcal{C}_{free}$. By dividing the domain of τ into intervals so that the change in q is no more than δ over each interval, then the LPM will produce collision-free path segments for replacement.

It turns out that for the Reeds-Shepp car (which has reverse) such an LPM can be designed because it is *small-time locally controllable*, a property that will be covered in Sections 15.1.3 and 15.4. In general, many techniques from Chapter 15 may be useful for analyzing and designing effective LPMs.

An interesting adaptation of the plan-and-transform approach has been developed for problems that involve k implicit constraints of the form $g_i(q, \dot{q}) = 0$. An outline of the *multi-level* approach, which was introduced in [859], is shown in Figure 14.25 (a similar approach was also introduced in [333]). The idea is to sort the k constraints into a sequence and introduce them one at a time. Initially, a path is planned that ignores the constraints. This path is first transformed to satisfy $g_1(q, \dot{q}) = 0$ and avoid collisions by using the plan-and-transform method of Figure 14.21. If successful, then the resulting path is transformed into one that is collision-free and satisfies both $g_1(q, \dot{q}) = 0$ and $g_2(q, \dot{q}) = 0$. This process repeats by adding one constraint each time, until either the method fails or all k

MULTI-LEVEL APPROACH

1. Compute a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ using a standard motion planning algorithm (as in Part II), and let $i = 1$.

2. Transform τ into a collision free path that satisfies $g_j(q, \dot{q}) = 0$ for all j from 1 to i.

3. If the transformation failed in Step 2, then terminate and report failure.

4. If $i < k$, the number of implicit velocity constraints, then increment i and go to Step 2. Otherwise, terminate and successfully report τ as a path that satisfies all constraints.

Figure 14.25: The multi-level approach considers implicit constraints one at a time.

constraints have been taken into account.

14.6.3 Path-Constrained Trajectory Planning

This section assumes that a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ has been given. It may be computed by a motion planning algorithm from Part II or given by hand. The remaining task is to determine the speed along the path in a way that satisfies differential constraints on the phase space X. Assume that each state $x \in X$ represents both a configuration and its time derivative, to obtain $x = (q, \dot{q})$. Let n denote the dimension of $\mathcal{C}$; hence, the dimension of X is $2n$. Once a path is given, there are only two remaining degrees of freedom in X: 1) the position $s \in [0,1]$ along the domain of τ, and 2) the speed $\dot{s} = ds/dt$ at each s. The full state, x, can be recovered from these two parameters. As the state changes, it must satisfy a given system, $\dot{x} = f(x, u)$. It will be seen that a 2D planning problem arises, which can be solved efficiently using many alternative techniques. Similar concepts appeared for decoupled versions of time-varying motion planning in Section 7.1. The presentation in the current section is inspired by work in time-scaling paths for robot manipulators [456, 876, 879], which was developed a couple of decades ago. At that time, computers were much slower, which motivated the development of strongly decoupled approaches.

14.6.3.1 Expressing systems in terms of s, $\dot{s}$, and $\ddot{s}$

Suppose that a system is given in the form

$$\ddot{q} = h(q, \dot{q}, u), \tag{14.31}$$

in which there are n action variables $u = (u_1, \ldots, u_n)$. It may be helpful to glance ahead to Example 14.6, which will illustrate the coming concepts for the simple

case of double integrators $\ddot{q} = u$. The acceleration in $\mathcal{C}$ is determined from the state $x = (q, \dot{q})$ and action u. Assume $u \in U$, in which U is an n-dimensional subset of $\mathbb{R}^n$. If h is nonsingular at x, then an n-dimensional set of possible accelerations arises from choices of $u \in U$. This means it is fully actuated. If there were fewer than n action variables, then there would generally not be enough freedom to follow a specified path. Therefore, U must be n-dimensional. Which choices of u, however, constrain the motion to follow the given path τ? To determine this, the q, $\dot{q}$, and $\ddot{q}$ variables need to be related to the path domain s and its first and second time derivatives $\dot{s}$ and $\ddot{s}$, respectively. This leads to a subset of U that corresponds to actions that follow the path.

Suppose that s, $\dot{s}$, $\ddot{s}$, and a path τ are given. The configuration $q \in \mathcal{C}_{free}$ is

$$q = \tau(s). \tag{14.32}$$

Assume that all first and second derivatives of τ exist. The velocity $\dot{q}$ can be determined by the chain rule as

$$\dot{q} = \frac{d\tau}{ds}\frac{ds}{dt} = \frac{d\tau}{ds}\,\dot{s}, \tag{14.33}$$

in which the derivative $d\tau/ds$ is evaluated at s. The acceleration is obtained by taking another derivative, which yields

$$\begin{aligned} \ddot{q} &= \frac{d}{dt}\left(\frac{d\tau}{ds}\,\dot{s}\right) \\ &= \frac{d^2\tau}{ds^2}\frac{ds}{dt}\,\dot{s} + \frac{d\tau}{ds}\,\ddot{s} \\ &= \frac{d^2\tau}{ds^2}\,\dot{s}^2 + \frac{d\tau}{ds}\,\ddot{s}, \end{aligned} \tag{14.34}$$

by application of the product rule. The full state $x = (q, \dot{q})$ can be recovered from $(s, \dot{s})$ using (14.32) and (14.33).

The next step is to obtain an equation that looks similar to (14.31), but is expressed in terms of s, $\dot{s}$, and $\ddot{s}$. A function $h'(s, \dot{s}, u)$ can be obtained from $h(q, \dot{q}, u)$ by substituting $\tau(s)$ for q and the right side of (14.33) for $\dot{q}$:

$$h'(s, \dot{s}, u) = h(\tau(s), \frac{d\tau}{ds}\,\dot{s}, u). \tag{14.35}$$

This yields

$$\ddot{q} = h'(s, \dot{s}, u). \tag{14.36}$$

For a given state x (which can be obtained from s and $\dot{s}$), the set of accelerations that can be obtained by a choice of u in (14.36) is the same as that for the original system in (14.31). The only difference is that x is now constrained to a 2D subset of X, which are the states that can be reached by selecting values for s and $\dot{s}$.

Applying (14.34) to the left side of (14.36) constrains the accelerations to cause motions that follow τ. This yields

$$\frac{d^2\tau}{ds^2}\ \dot{s}^2 + \frac{d\tau}{ds}\ \ddot{s} = h'(s, \dot{s}, u), \tag{14.37}$$

which can also be expressed as

$$\frac{d\tau}{ds}\ \ddot{s} = h'(s, \dot{s}, u) - \frac{d^2\tau}{ds^2}\ \dot{s}^2, \tag{14.38}$$

by moving the first term of (14.34) to the right. Note that n equations are actually represented in (14.38). For each i in which $d\tau_i/ds \neq 0$, a constraint of the form

$$\ddot{s} = \frac{1}{d\tau_i/ds} h_i'(s, \dot{s}, u_i) - \frac{d^2\tau_i}{ds^2}\ \dot{s}^2 \tag{14.39}$$

is obtained by solving for $\ddot{s}$.

14.6.3.2 Determining the allowable accelerations

The actions in U that cause τ to be followed can now be characterized. An action $u \in U$ follows τ if and only if every equation of the form (14.39) is satisfied. If $d\tau_i/ds \neq 0$ for all i from 1 to n, then n such equations exist. Suppose that u_1 is chosen, and the first equation is solved for $\ddot{s}$. The required values of the remaining action variables $u_2, \ldots, u_n$ can be obtained by substituting the determined $\ddot{s}$ value into the remaining $n-1$ equations. This means that the actions that follow τ are at most a one-dimensional subset of U.

If $d\tau_i/ds = 0$ for some i, then following the path requires that $\dot{q}_i = 0$. Instead of (14.39), the constraint is that $h_i(q, \dot{q}, u) = 0$. Example 14.6 will provide a simple illustration of this. If $d\tau_i/ds = 0$ for all i, then the configuration is not allowed to change. This occurs in the degenerate (and useless) case in which τ is a constant function.

In many cases, a value of u does not exist that satisfies all of the constraint equations. This means that the path cannot be followed at that particular state. Such states should be removed, if possible, by defining phase constraints on X. By a poor choice of path τ violating such a phase constraint may be unavoidable. There may exist some s for which no $u \in U$ can follow τ, regardless of $\dot{s}$.

Even if a state trajectory may be optimal in some sense, its quality ultimately depends on the given path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$. Consider the path shown in Figure 14.26. At $\tau(1/3)$, a "corner" is reached. This violates the differentiability assumption and would require infinite acceleration to traverse while remaining on τ. For some models, it may be possible to stop at $\tau(1/3)$ and then start again. For example, imagine a floating particle in the plane. It can be decelerated to rest exactly at $\tau(1/3)$ and then started in a new direction to exactly follow the curve. This assumes that the particle is fully actuated. If there are nonholonomic constraints on $\mathcal{C}$, as in the case of the Dubins car, then the given path must at least satisfy

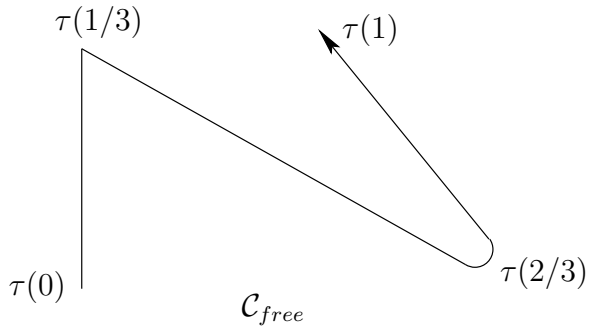


Figure 14.26: A bad path for path-constrained trajectory planning.

them before accelerations can be considered. The solution in this case depends on the existence of *decoupling vector fields* [157, 224].

It is generally preferable to round off any corners that might have been produced by a motion planning algorithm in constructing τ. This helps, but it still does not completely resolve the issue. The portion of the path around $\tau(2/3)$ is not desirable because of high curvature. At a fixed speed, larger accelerations are generally needed to follow sharp turns. The speed may have to be decreased simply because τ carelessly requires sharp turns in $\mathcal{C}$. Imagine developing an autonomous double-decker tour bus. It is clear that following the curve around $\tau(2/3)$ may cause the bus to topple at high speeds. The bus will have to slow down because it is a slave to the particular choice of τ.

14.6.3.3 The path-constrained phase space

Recall the approach in Section 14.4.1 that enabled systems of the form $\ddot{q} = h(q, \dot{q}, u)$ to be expressed as $\ddot{q} = u'$ for some suitable $U'(q, \dot{q}) \subseteq U$ (this was illustrated in Figure 14.15). This enabled many systems to be imagined as multiple, independent double integrators with phase-dependent constraints on the action space. The same idea can be applied here to obtain a single integrator.

Let S denote a 2D *path-constrained phase space*, in which each element is of the form $(s, \dot{s})$ and represents the position and velocity along τ. This parameterizes a 2D subset of the original phase space X. Each original state vector is $x = (q, \dot{q}) = (\tau(s), d\tau/ds\ \dot{s})$. Which accelerations are possible at points in S? At each $(s, \dot{s})$, a subset of U can be determined that satisfies the equations of the form (14.39). Each valid action yields an acceleration $\ddot{s}$. Let $U'(s, \dot{s}) \subseteq \mathbb{R}$ denote the set of all values of $\ddot{s}$ that can be obtained from an action $u \in U$ that satisfies (14.39) for each i (except the ones for which $d\tau_i/ds = 0$). Now the system can be expressed as $\ddot{s} = u'$, in which $u' \in U'(s, \dot{s})$. After all of this work, we have arrived at the double integrator. The main complication is that $U'(s, \dot{s})$ can be challenging to determine for some systems. It could consist of a single interval, disjoint intervals, or may even be empty. Assuming that $U'(s, \dot{s})$ has been characterized, it is straightforward to solve the remaining planning problem using techniques already presented in this chapter. One double integrator is not very challenging; hence, efficient sampling-

based algorithms exist.

An obstacle region $S_{obs} \subset S$ will now be considered. This includes any states that belong to X_{free}. Given s and $\dot{s}$, the state x can be computed to determine whether any constraints on X are violated. Usually, τ is constructed to avoid obstacle collision; however, some phase constraints may also exist. The obstacle region S_{obs} also includes any points $(s, \dot{s})$ for which $U'(s, \dot{s})$ is empty. Let S_{free} denote $S \setminus S_{obs}$.

Before considering computation methods, we give some examples.

Example 14.6 (Path-Constrained Double Integrators) Consider the case of two double integrators. This could correspond physically to a particle moving in $\mathbb{R}^2$. Hence, $\mathcal{C} = \mathcal{W} = \mathbb{R}^2$. Let $U = [-1, 1]^2$ and $\ddot{q} = u$ for $u \in U$. The path τ will be chosen to force the particle to move along a line. For linear paths, $d\tau/ds$ is constant and $d^2\tau/ds^2 = 0$. Using these observations and the fact that $h'(s, \dot{s}, u) = u$, (14.39) simplifies to

$$\ddot{s} = \frac{u_i}{d\tau_i/ds}, \tag{14.40}$$

for $i = 1, 2$.

Suppose that $\tau(s) = (s, s)$, which means that the particle must move along a diagonal line through the origin of $\mathcal{C}$. This further simplifies (14.40) to $\ddot{s} = u_1$ and $\ddot{s} = u_2$. Hence any $u_1 \in [-1, 1]$ may be chosen, but u_2 must then be chosen as $u_2 = u_1$. The constrained system can be written as one double integrator $\ddot{s} = u'$, in which $u' \in [-1, 1]$. Both u_1 and u_2 are derived from u' as $u_1 = u_2 = u'$. Note that U' does not vary over S; this occurs because a linear path is degenerate.

Now consider constraining the motion to a general line:

$$\tau(s) = (a_1 s + b_1,\ a_2 s + b_2), \tag{14.41}$$

in which a_1 and a_2 are nonzero. In this case, (14.40) yields $\ddot{s} = u_1/a_1$ and $\ddot{s} = u_2/a_2$. Since each $u_i \in [-1, 1]$, each equation indicates that $\ddot{s} \in [-1/a_i, 1/a_i]$. The acceleration must lie in the intersection of these two intervals. If $|a_1| \geq |a_2|$, then $\ddot{s} \in [-1/a_1, 1/a_1]$. We can designate $u' = u_1$ and let $u_2 = u' a_2/a_1$. If $|a_1| > |a_2|$, then $\ddot{s} \in [-1/a_2, 1/a_2]$, $u' = u_2$, and $u_1 = u' a_1/a_2$.

Suppose that $a_1 = 0$ and $a_2 \neq 0$. The path is

$$\tau(s) = (q_1,\ a_2 s + b_2), \tag{14.42}$$

in which q_1 is fixed and the particle is constrained to move along a vertical line in $\mathcal{C} = \mathbb{R}^2$. In this case, only one constraint, $\ddot{s} = u_2$, is obtained from (14.40). However, u_1 is independently constrained to $u_1 = 0$ because horizontal motions are prohibited.

If n independent, double integrators are constrained to a line, a similar result is obtained. There are n equations of the form (14.40). The $i \in \{1, \ldots, n\}$ for which $|a_i|$ is largest determines the acceleration range as $\ddot{s} \in [-1/a_i, 1/a_i]$. The action u'

is defined as $u' = u_i$, and the u_j for $j \neq i$ are obtained from the remaining $n-1$ equations.

Now assume τ is nonlinear, in which case (14.39) becomes

$$\ddot{s} = \frac{u_i}{d\tau_i/ds} - \frac{d^2\tau_i}{ds^2}\,\dot{s}^2, \tag{14.43}$$

for each i for which $d\tau_i/ds \neq 0$. Now the set $U'(s, \dot{s})$ varies over S. As the speed $\dot{s}$ increases, it becomes less likely that $U'(s, \dot{s})$ is nonempty. In other words, it is less likely that a solution exists to all equations of the form (14.43). In a physical system, that means that staying on the path requires turning too sharply. At a high speed, this may require an acceleration $\ddot{q}$ that lies outside of $[-1, 1]^n$. ■

The same ideas can be applied to systems that are much more complicated. This should not be surprising because in Section 14.4.1 systems of the form $\ddot{q} = h(q, \dot{q})$ were interpreted as multiple, independent double integrators of the form $\ddot{q} = u'$, in which $u' \in U'(q, \dot{q})$ provided the possible accelerations. Under this interpretation, and in light of Example 14.6, constraining the motions of a general system to a path τ just further restricts $U'(q, \dot{q})$. The resulting set of allowable accelerations may be at most one-dimensional.

The following example indicates the specialization of (14.39) for a robot arm.

Example 14.7 (Path-Constrained Manipulators) Suppose that the system is described as (13.142) from Section 13.4.2. This is a common form that has been used for controlling robot arms for decades. Constraints of the form (14.39) can be derived by expressing q, $\dot{q}$, and $\ddot{q}$ in terms of s, $\dot{s}$, and $\ddot{s}$. This requires using (14.32), (14.33), and (14.34). Direct substitution into (13.142) yields

$$M(\tau(s))\left(\frac{d^2\tau}{ds^2}\dot{s}^2 + \frac{d\tau}{ds}\ddot{s}\right) + C\left(\tau(s), \frac{d\tau}{ds}\dot{s}\right)\frac{d\tau}{ds}\dot{s} + g(\tau(s)) = u. \tag{14.44}$$

This can be simplified to n equations of the form

$$\alpha_i(s)\ddot{s} + \beta_i(s)\dot{s}^2 + \gamma_i(s)\dot{s} = u_i. \tag{14.45}$$

Solving each one for $\ddot{s}$ yields a special case of (14.39). As in Example 14.6, each equation determines a bounding interval for $\ddot{s}$. The intersection of the intervals for all n equations yields the allowed interval for $\ddot{s}$. The action u' once again indicates the acceleration in the interval, and the original action variables u_i can be obtained from (14.45). If $d\tau_i/ds = 0$, then $\alpha_i(s) = 0$, which corresponds to the case in which the constraint does not apply. Instead, the constraint is that the vector u must be chosen so that $\dot{q}_i = 0$. ■

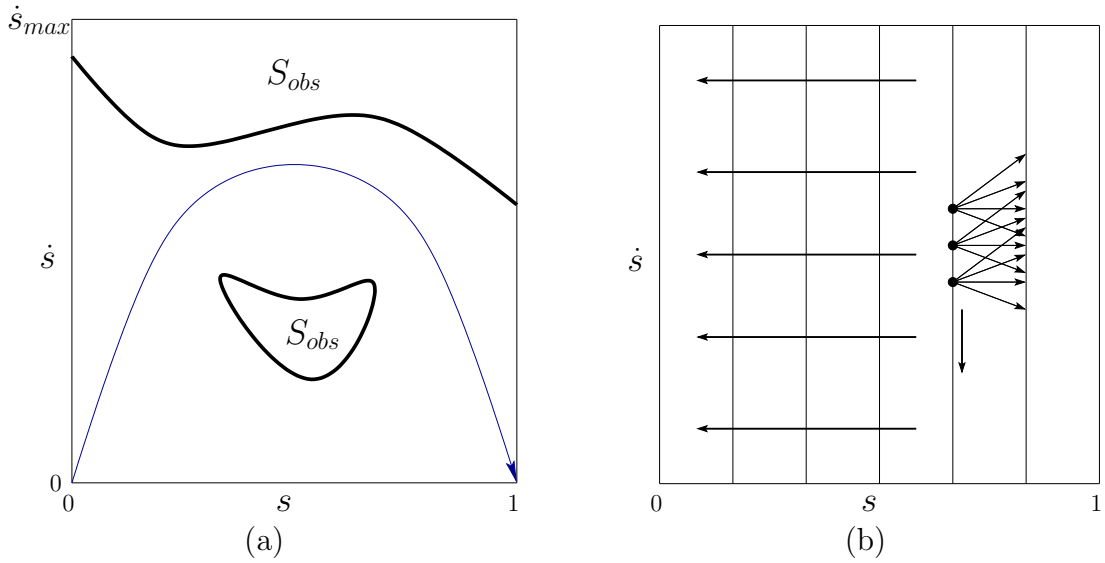


Figure 14.27: (a) Planning occurs in the path-constrained phase space. (b) Due to the forward-progress assumption, value iteration can be reduced to a quick wavefront propagation across regularly spaced vertical lines in S.

14.6.3.4 Computing optimal solutions via dynamic programming

Dynamic programming with interpolation, as covered in Section 14.5, can be applied to solve the problem once it is formulated in terms of the path-constrained phase space $S \subset \mathbb{R}^2$. The domain of τ provides the constraint $0 \leq s \leq 1$. Assume that only forward progress along the path is needed; moving in the reverse direction should not be necessary. This implies that $\dot{s} > 0$. To make S bounded, an upper bound, $\dot{s}_{max}$, is usually assumed, beyond which it is known that the speed is too high to follow the path.

This results in the planning problem shown in Figure 14.27a. The system is expressed as $\ddot{s} = u'$, in which $u' \in U'(s, \dot{s})$. The initial phase in S is $(0, \dot{s}_i)$ and the goal phase is $(1, \dot{s}_g)$. Typically, $\dot{s}_i = \dot{s}_g = 0$. The region shown in Figure 14.27 is contained in the first quadrant of the phase space because only positive values of s and $\dot{s}$ are allowed (in Figure 14.13, q and $\dot{q}$ could be positive or negative). This implies that all motions are to the right. The actions determine whether accelerations or decelerations will occur.

Backward value iteration with interpolation can be easily applied by discretizing S and $U'(s, \dot{s})$. Due to the constraint $\dot{s} > 0$, making a Dijkstra-like version of the algorithm is straightforward. A simple wavefront propagation can even be performed, starting at $s = 1$ and progressing backward in vertical waves until $s = 0$ is reached. See Figure 14.27b. The backprojection (14.29) can be greatly simplified. Suppose that the s-axis is discretized into $m + 1$ regularly spaced values s_0, ..., s_m at every Δs, for some fixed $\Delta s > 0$. Thus, $s_k = (k\Delta s)/m$. The index k can be interpreted as the stage. Starting at $k = m$, the final cost-to-go $G^*_m(s_m, \dot{s}_m)$

is defined as 0 if the corresponding phase represents the goal, and ∞ otherwise. At each s_k, the $\dot{s}$ values are sampled, and the cost-to-go function is represented using one-dimensional linear interpolation along the vertical axis. At each stage, the dynamic programming computation

$$G_k^*(s_k, \dot{s}_k) = \min_{u' \in U'(s_k, \dot{s}_k)} \Big\{ l'_d(s_k, \dot{s}_k, u') + G_{k+1}^*(s_{k+1}, \dot{s}_{k+1}) \Big\} \tag{14.46}$$

is performed at each $\dot{s}$ sample. This represents a special form of (14.27). Linear interpolation over discretized $\dot{s}$ values is used to evaluate $G_{k+1}^*(s_{k+1}, \dot{s}_{k+1})$. The cost term l'_d is obtained from l_d by computing the original state $x \in X$ from s and $\dot{s}$; however, if the trajectory segment enters S_{obs}, it receives infinite cost. The computations proceed until stage $k = 1$, at which time the optimal cost-to-go $G_1^*(s_1, \dot{s}_1)$ is computed. The optimal trajectory is obtained by using the cost-to-go function at each stage as a navigation function.

The dynamic programming approach is so general that it can even be extended to path-constrained trajectory planning in the presence of higher order constraints [880]. For example, if a system is specified as $q^{(3)} = h(q, \dot{q}, \ddot{q}, u)$, then a 3D path-constrained phase space results, in which each element is expressed as $(s, \dot{s}, \ddot{s})$. The actions in this space are jerks, yielding $s^{(3)} = u'$ for $u' \in U'(s, \dot{s}, \ddot{s})$.

14.6.3.5 A bang-bang approach for time optimality

The dynamic programming approach is already very efficient because the search is confined to two dimensions. Nevertheless, trajectories that are time optimal can be computed even more efficiently if S_{obs} has some special structure. The idea is to find an alternating sequence between two motion primitives: one of maximum acceleration and one of maximum deceleration. This kind of switching between extreme opposites is often called *bang-bang control* and arises often in the development of time-optimal control laws (look ahead to Example 15.4). The method explained here was introduced in [121, 879]. One drawback of obtaining time-optimal trajectories is that they cannot be tracked (the fourth module from Section 14.6.1) if errors occur because the solutions travel on the boundary of the reachable set.

The approach was developed for robot arms, as considered in Example 14.7. Suppose that S_{obs} is a single connected component that is bounded above by $\dot{s}_{max}$, and on the sides it is bounded by $s = 0$ and $s = 1$. It is assumed that S arises only due to the vanishing of the interval of allowable values for $\ddot{s}$ (in this case, $U'(s, \dot{s})$ becomes empty). It is also assumed that the lower boundary of S_{obs} can be expressed as a differentiable function $\phi : [0, 1] \rightarrow S$, called the *limit curve*, which yields the maximum speed $\dot{s} = \phi(s)$ for every $s \in [0, 1]$. The method is extended to handle multiple obstacles in [879], but this case is not considered here. Assume also that $d\tau_i/ds \neq 0$ for every i; the case of $d\tau_i/ds = 0$ can also be handled in the method [878].

Let $u'_{min}(s, \dot{s})$ and $u'_{max}(s, \dot{s})$ denote the smallest and largest possible accelerations, respectively, from $(s, \dot{s}) \in S$. If $(s, \dot{s}) \notin S_{obs}$, then $u'_{min}(s, \dot{s}) < u'_{max}(s, \dot{s})$.

BANG-BANG APPROACH

1. From the final state $(1, 0)$, apply reverse-time integration to $\ddot{s} = u'_{min}(s, \dot{s})$. Continue constructing the curve numerically until either the interior of S_{obs} is entered or $\dot{s} = 0$. In the latter case, the algorithm terminates with failure.

2. Let $(s_{cur}, \dot{s}_{cur}) = (0, 0)$.

3. Apply forward integration $\ddot{s} = u'_{max}(s, \dot{s})$ from $(s_{cur}, \dot{s}_{cur})$ until either the interior of S_{obs} is entered or the curve generated in Step 1 is crossed. In the latter case, the problem is solved.

4. Starting at the point where the trajectory from Step 3 crossed the limit curve, find next tangent point $(s_{tan}, \dot{s}_{tan})$ to the right along the limit curve. From $(s_{tan}, \dot{s}_{tan})$, perform reverse integration on $\ddot{s} = u'_{min}(s, \dot{s})$ until the curve from Step 3 is hit. Let $(s_{cur}, \dot{s}_{cur}) = (s_{tan}, \dot{s}_{tan})$ and go to Step 3.

Figure 14.28: The bang-bang approach finds a time-optimal, path-constrained trajectory with less searching than the dynamic programming approach.

At the limit curve, $u'_{min}(s, \phi(s)) = u'_{max}(s, \phi(s))$. Applying the only feasible action in this case generates a velocity that is tangent to the limit curve. This is called a *tangent point*, $(s_{tan}, \dot{s}_{tan})$, to ϕ. Inside of S_{obs}, no accelerations are possible.

The *bang-bang approach* is described in Figure 14.28, and a graphical illustration appears in Figure 14.29. Assume that the initial and goal phases are $(0, 0)$ and $(1, 0)$, respectively. Step 1 essentially enlarges the goal by constructing a maximum-deceleration curve that terminates at $(1, 0)$. A trajectory that contacts this curve can optimally reach $(1, 0)$ by switching to maximum deceleration. Steps 3 and 4 construct a maximum-acceleration curve followed by a maximum-deceleration curve. The acceleration curve runs until it pierces the limit curve. This constraint violation must be avoided. Therefore, a deceleration must be de-

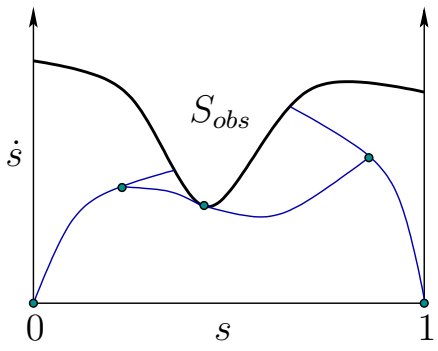


Figure 14.29: An illustration of the bang-bang approach to computing a time-optimal trajectory. The solution trajectory is obtained by connecting the dots.

termined that departs earlier from the acceleration curve and just barely misses entering the interior of S_{obs}. This curve must become tangent to the limit curve; therefore, a search is made along the limit curve for the next possible tangent point. From there, reverse-time integration is used in Step 4 to generate a deceleration curve that contacts the acceleration curve. A portion of the solution has now been obtained in which an acceleration is followed by a deceleration that arrives at a tangent point of ϕ. It is possible that Step 4 is not reached because the curve that connects to the goal is contacted. Starting from the tangent point, Steps 3 and 4 are repeated until the goal curve is contacted.

14.7 Gradient-Based Trajectory Optimization

This section provides a brief overview of a complementary problem to motion planning. Suppose that an algorithm in this chapter returns a feasible action trajectory. How can the solution be improved? *Trajectory optimization* refers to the problem of perturbing the trajectory while satisfying all constraints so that its quality can be improved. For example, it may be desirable to shorten a trajectory computed by an RRT, to remove some of the arbitrary changes in actions due to randomization. Trajectory optimization is considered complementary to motion planning because it usually requires an initial guess, which could be provided by a planning algorithm. Trajectory optimization can be considered as a kind of BVP, but one that improves an initial guess, as opposed to determining trajectories from scratch.

The optimization issue also exists for paths computed by sampling-based algorithms for the Piano Mover's Problem; however, without differential constraints, it is much simpler to shorten paths. The plan and transform method of Section 14.6.2 can be applied, and the LPM just connects pairs of configurations along the shortest path in $\mathcal{C}$. In the presence of differential constraints, the BVP must be faced.

In the most general setting, it is very difficult to improve trajectories. There are numerous methods from optimization literature; see [98, 151, 664] for overviews. The purpose of this section is to encourage further study by briefly mentioning the various kinds of methods that have been developed, instead of explaining them in detail. The methods fall under the area of *nonlinear programming* (NLP) (or *nonlinear optimization*), as opposed to *linear programming*, which was used to find randomized security strategies in Section 9.3. The optimization actually occurs in a space of possible trajectories, each of which is a function of time. Therefore, the calculus of variations, which was used in Section 13.4.1, becomes relevant to characterize extrema. The functional Φ from that setting becomes the cost functional L in the current setting. The system $\dot{x} = f(x, u)$ forms an additional set of constraints that must be satisfied, but u can be selected in the optimization.

To enable numerical computation methods, a family of trajectories is specified in terms of a parameter space. The optimization can then be viewed as an incremental search in the parameter space while satisfying all constraints. The direction

of motion in each step is determined by computing the gradient of a cost functional with respect to the parameters while constrained to move in a direction tangent to the constraints. Hence, much of nonlinear programming can be considered as an application of Newton's method or gradient descent. As in standard optimization, second-order derivatives of the cost functional can be used to indicate when the search should terminate. The numerical issues associated with these methods are quite involved; several NLP software packages, such as the NAG Fortran Library or packages within Matlab, are available.

Nonlinear optimal control theory can be considered as a variant of NLP. The dynamic programming recurrence becomes a differential equation in the continuous-time setting, and Hamilton's equations (13.198) generalize to Pontryagin's minimum principle. These are covered in Section 15.2. The extra variables that arise in the minimum principle can be considered as Lagrange multipliers of a constrained optimization, in which $\dot{x} = f(x, u)$ is the constraint. The differential equations arising from dynamic programming or the minimum principle are difficult to solve analytically; therefore, in most cases, numerical techniques are used. The case of numerical dynamic programming was covered in Section 14.5.

Shooting methods constitute the simplest family of trajectory optimization methods. As a simple example, suppose that an action trajectory $\tilde{u} : [0, t_F] \rightarrow \mathbb{R}$ has been computed of the form

$$u(t) = w_1 + w_2 t, \tag{14.47}$$

in which w_1 and w_2 are some fixed parameters. Consider perturbing w_1 and w_2 by some small amount and applying the integration in (14.1). If f satisfies Lipschitz conditions, then a small perturbation should produce a small change in $\tilde{x}$. The resulting new trajectory can be evaluated by a cost functional to determine whether it is an improvement. It might, for example, have lower maximum curvature. Rather than picking a perturbation at random, the gradient of the cost functional with respect to the parameters can be computed. A small step in the parameter space along the negative gradient direction should reduce the cost. It is very likely, however, that perturbing w_1 and w_2 will move the final state $x(t_F)$. Usually, a termination condition, such as $x(t_F) = x_G$, must be enforced as a constraint in the optimization. This removes degrees of freedom from the optimization; therefore, more trajectory parameters are often needed.

Suppose more generally that a motion planning algorithm computes an action sequence based on the discrete-time model. Each action in the sequence remains constant for duration Δt. The time duration of each action can instead be defined as a parameter to be perturbed. Each action variable u_i over each interval could also be perturbed using by (14.47) with the initial condition that $w_1 = u_i$ and $w_2 = 0$. The dimension of the search has increased, but there are more degrees of freedom. In some formulations, the parameters may appear as implicit constraints; in this case, a BVP must be solved in each iteration. The minimum principle is often applied in this case [98]. More details on formulating and solving the trajectory optimization problem via shooting appear in [151].

Several difficulties are encountered when applying the shooting technique to trajectory optimization among obstacles. Each perturbation requires integration and collision-checking. For problems involving vehicles, the integrations can sometimes be avoided by exploiting symmetries [197]. For example, a path for the Dubins car can be perturbed by changing a steering angle over a short amount of time, and the rest of the trajectory can simply be transformed using a matrix of $SE(2)$. A critical problem is that following the negative gradient may suggest shortening the path in a way that causes collision. The problem can be alleviated by breaking the trajectory into segments, as in the plan-and-transform approach; however, this yields more optimizations. Another possible solution is to invent a penalty function for the obstacles; however, this is difficult due to local minima problems and the lack of representing the precise boundary of X_{obs}.

Another difficulty with shooting is that a small change in the action near the starting time may lead to great changes in the states at later times. One way to alleviate this problem is by *multiple shooting* (as opposed to *single shooting*, which has been described so far). In this case, the trajectory is initially broken into segments. These could correspond to the time boundaries imposed by a sequence of motion primitives. In this case, imagine perturbing each motion primitive separately. Extra constraints are needed in this case to indicate that all of the trajectory pieces must remain connected. The multiple shooting method can be generalized to a family of methods called *transcription* or *collocation* (see [98] for references). These methods again split the trajectory into segments, but each connection constraint relates more points along the trajectory than just the segment endpoints. One version of transcription uses implicit constraints, which require using another BVP solver, and another version uses parametric constraints, which dramatically increases the dimension of the search. The latter case is still useful in practice by employing fast, sparse-matrix computation methods.

One of the main difficulties with trajectory optimization methods is that they can become stuck in a local minimum in the space of trajectories. This means that their behavior depends strongly on the initial guess. It is generally impossible for them to find a trajectory that is not homotopic to the initial trajectory. They cannot recover from an initial guess in a bad homotopy class. If X_{obs} is complicated, then this issue becomes increasingly important. In many cases, variational techniques might not even find an optimal solution within a single homotopy class. Multiple local minima may exist if the closure of X_{free} contains positive curvature. If it does not, the space is called *nonpositively curved* (NPC) or CAT(0), which is a property that can be derived directly from the metric on X [139]. For these spaces, the locally optimal trajectory with respect to the metric is always the best within its homotopy class.

Further Reading

The characterization and computation of reachable sets has been growing in interest [100, 102, 706, 707, 916, 955]. One motivation for studying reachability is *verification*, which ensures that a control system behaves as desired under all possible disturbances.

This can actually be modeled as a game against nature, in which nature attempts to bring the system into an undesirable state (e.g., crashing an airplane). For recent progress on characterizing X_{ric}, see [355]. The triangularization argument for completeness appears in a similar context in [292]. The precise rate of convergence, expressed in terms of dispersion and Lipschitz conditions, for resolution-complete sampling-based motion planning methods under differential constraints is covered in [196]. For the computational complexity of control problems, see [114, 766]. For further reading on motion primitives in the context of planning, see [360, 362, 363, 393, 787, 794, 848]. For further reading on dynamical simulation and numerical integration, see [331, 440, 863].

Section 14.4.1 was based on [288, 290, 441]. For more works on kinodynamic planning, see [203, 237, 289, 356, 360, 611, 780, 999]. Section 14.4.2 was inspired by [73]. Section 14.4.3 was drawn from [611]. For more work on RRTs under differential constraints, see [138, 199, 224, 324, 360, 393, 509, 949]. For other works on nonholonomic planning, see the survey [596] and [67, 277, 334, 335, 354, 357, 482, 579, 633, 672]. Combinatorial approaches to nonholonomic planning have appeared in [13, 128, 347].

Section 14.5 was developed by adapting value iteration to motion planning problems. For general convergence theorems for value iteration with interpolation, see [168, 292, 400, 565, 567]. In [168], global constraints on the phase space are actually considered. The use of these techniques and the development of Dijkstra-like variants are covered in [607]. Related work exists in artificial intelligence [722] and control theory [946].

Decoupled approaches to planning, as covered in Section 14.6, are very common in robotics literature. For material related to the plan-and-transform method, see [333, 596, 859]. For more on decoupled trajectory planning and time scaling, see [353, 456, 457, 843, 876, 877, 880, 881], and see [104, 120, 121, 785, 879, 894, 878] for particular emphasis on time-optimal trajectories.

For more on gradient-based techniques in general, see [98] and references therein. Classical texts on the subject are [151, 664]. Gradient-based approaches to path deformation in the context of nonholonomic planning appear in [197, 343, 575].

The techniques presented in this chapter are useful in other fields beyond robotics. For aerospace applications of motion planning, see [86, 202, 436, 437, 786]. Motion planning problems and techniques have been gaining interest in computer graphics, particularly for generating animations of virtual humans (or digital actors); works in this area include [35, 86, 393, 498, 544, 554, 557, 591, 617, 649, 712, 802, 980]. In many of these works, *motion capture* is a popular way to generate a database of recorded motions that serves as a set of motion primitives in the planning approach.

Exercises

1. Characterize X_{ric} for the case of a point mass in $\mathcal{W} = \mathbb{R}^2$, with each coordinate modeled as a double integrator. Assume that $u_1 = 1$ and u_2 may take any value in $[-1, 1]$. Determine X_{ric} for:

   (a) A point obstacle at $(0, 0)$ in $\mathcal{W}$.

   (b) A segment from $(0, -1)$ to $(0, 1)$ in $\mathcal{W}$.

   Characterize the solutions in terms of the phase variables $q_1(0)$, $q_2(0)$, $\dot{q}_1(0)$, and $\dot{q}_2(0)$.

2. Extending the double integrator:

   (a) Develop a lattice for the triple integrator $q^{(3)} = u$ that extends naturally from the double-integrator lattice.

   (b) Describe how to develop a lattice for higher order integrators $q^{(n)}$ for $n > 3$.

3. Make a figure similar to Figure 14.6b, but for three stages of the Reeds-Shepp car.

4. Determine expressions for the upper and lower boundaries of the time-limited reachable sets shown in Figure 14.14. Express them as parabolas, with $\dot{q}$ as a function of q.

5. A reachability graph can be made by "rolling" a polyhedron in the plane. For example, suppose a solid, regular tetrahedron is placed on a planar surface. Assuming high friction, the tetrahedron can be flipped in one of four directions by pushing on the top. Construct the three-stage reachability graph for this problem.

6. Construct a four-stage reachability graph similar to the one shown in Figure 14.6b, but for the case of a differential drive robot modeled by (13.17). Use the three actions $(1, 0)$, $(0, 1)$, and $(1, 1)$. Draw the graph in the plane and indicate the configuration coordinates of each vertex.

7. Section 14.2.2 explained how resolution-complete algorithms exist for planning under differential constraints. Suppose that in addition to continuous state variables, there are discrete modes, as introduced in Section 7.3, to form a hybrid system. Explain how resolution-complete planning algorithms can be developed for this case. Extend the argument shown in Figure 14.7.

Implementations

8. Compare the performance and accuracy of Euler integration to fourth-order Runge-Kutta on trajectories generated for a single, double, and triple integrator. For accuracy, compare the results to solutions obtained analytically. Provide recommendations of which one to use under various conditions.

9. Improve Figure 14.13 by making a plot of the actual trajectories, which are parabolic in most cases.

10. In Figure 14.13, the state trajectory segments are longer as $|\dot{x}|$ increases. Develop a lattice that tries to keep all segments as close to the same length as possible by reducing Δt as $|\dot{x}|$ increases. Implement and experiment with different schemes and report on the results.

11. Develop an implementation for computing approximately time-optimal state trajectories for a point mass in a 2D polygonal world. The robot dynamics can be modeled as two independent double integrators. Search the double-integrator lattice in $X = \mathbb{R}^4$ to solve the problem. Animate the computed solutions.

12. Experiment with RDT methods applied to a spacecraft that is modeled as a 3D rigid body with thrusters. Develop software that computes collision-free trajectories for the robot. Carefully study the issues associated with choosing the metric on X.

13. Solve the problem of optimally bringing the Dubins car to a goal region in a polygonal world by using value iteration with interpolation.

14. Select and implement a planning algorithm that computes pushing trajectories for a differential drive robot that pushes a box in a polygonal environment. This was given as an example of a nonholonomic system in Section 13.1.3. To use the appropriate constraints on U, see [671].

15. Select and implement a planning algorithm that computes trajectories for parking a car while pulling a single trailer, using (13.19). Make an obstacle region in $\mathcal{W}$ that corresponds to a tight parking space and vary the amount of clearance. Also, experiment with driving the vehicle through an obstacle course.

16. Generate a 3D rendering of reachability graphs for the airplane model in (13.20). Assume that in each stage there are nine possible actions, based on combinations of flying to the right, left, or straight and decreasing, increasing, or maintaining altitude.

17. Implement the dynamic programming algorithm shown in Figure 14.27 for the two-link manipulator model given in Example 13.13.

18. Implement the bang-bang algorithm shown in Figure 14.28 for the two-link manipulator model given in Example 13.13.

19. For the Dubins car (or another system), experiment with generating a search graph based on Figure 14.7 by alternating between various step sizes. Plot in the plane, the vertices and state trajectories associated with the edges of the graph. Experiment with different schemes for generating a resolution-complete search graph in a rectangular region and compare the results.

20. Use value iteration with interpolation to compute the optimal cost-to-go for the Reeds-Shepp car. Plot level sets of the cost-to-go, which indicate the time-limited reachable sets. Compare the result to Figure 14.4.