Chapter 8

Feedback Motion Planning

So far in Part II it has been assumed that a continuous path sufficiently solves a motion planning problem. In many applications, such as computer-generated animation and virtual prototyping, there is no need to challenge this assumption because models in a virtual environment usually behave as designed. In applications that involve interaction with the physical world, future configurations may not be predictable. A traditional way to account for this in robotics is to use the refinement scheme that was shown in Figure 1.19 to design a feedback control law that attempts to follow the computed path as closely as possible. Sometimes this is satisfactory, but it is important to recognize that this approach is highly decoupled. Feedback and dynamics are neglected in the construction of the original path; the computed path may therefore not even be usable.

Section 8.1 motivates the consideration of feedback in the context of motion planning. Section 8.2 presents the main concepts of this chapter, but only for the case of a discrete state space. This requires less mathematical concepts than the continuous case, making it easier to present feedback concepts. Section 8.3 then provides the mathematical background needed to extend the feedback concepts to continuous state spaces (which includes C-spaces). Feedback motion planning methods are divided into complete methods, covered in Section 8.4, and sampling-based methods, covered in Section 8.5.

8.1 Motivation

For most problems involving the physical world, some form of feedback is needed. This means the actions of a plan should depend in some way on information gathered during execution. The need for feedback arises from the unpredictability of future states. In this chapter, every state space will be either discrete, or $X = \mathcal{C}$, which is a configuration space as considered in Chapter 4.

Two general ways to model uncertainty in the predictability of future states are

1. **Explicitly:** Develop models that explicitly account for the possible ways

	Open Loop	Feedback
Free motions	Traditional motion planning	Chapter 8
Dynamics	Chapters 14 and 15	Traditional control theory

Figure 8.1: By separating the issue of dynamics from feedback, two less-investigated topics emerge.

that the actual future state can drift away from the planned future state. A planning algorithm must take this uncertainty directly into account. Such explicit models of uncertainty are introduced and incorporated into the planning model in Part III.

2. **Implicitly:** The model of state transitions indicates that no uncertainty is possible; however, a feedback plan is constructed to ensure that it knows which action to apply, just in case it happens to be in some unexpected state during execution. This approach is taken in this chapter.

The implicit way to handle this uncertainty may seem strange at first; therefore, some explanation is required. It will be seen in Part III that explicitly modeling uncertainty is extremely challenging and complicated. The requirements for expressing reliable models are much stronger; the complexity of the problem increases, making algorithm design more difficult and leading to greater opportunities to make modeling errors. The implicit way of handling uncertainty in predictability arose in control theory [108, 122, 686]. It is well known that a feedback control law is needed to obtain reliable performance, yet it is peculiar that the formulation of dynamics used in most contexts does not explicitly account for this. Classical control theory has always assumed that feedback is crucial; however, only in modern branches of the field, such as *stochastic control* and *robust control*, does this uncertainty get explicitly modeled. Thus, there is a widely accepted and successful practice of designing feedback control laws that use state feedback to implicitly account for the fact that future states may be unpredictable. Given the widespread success of this control approach across numerous applications over the past century, it seems valuable to utilize this philosophy in the context of motion planning as well (if you still do not like it, then jump to Chapter 10).

Due to historical reasons in the development of feedback control, it often seems that feedback and dynamics are inseparable. This is mainly because control theory was developed to reliably alter the behavior of dynamical systems. In traditional motion planning, neither feedback nor dynamics is considered. A solution path is considered *open loop*, which means there is no feedback of information during execution to *close* the loop. Dynamics are also not handled because the additional complications of differential constraints and higher dimensional phase spaces arise (see Part IV).

By casting history aside and separating feedback from dynamics, four separate topics can be made, as shown in Figure 8.1. The topic of open-loop planning that involves dynamics has received increasing attention in recent years. This is the focus throughout most of Part IV. Those fond of classical control theory may criticize it for failing to account for feedback; however, such open-loop trajectories (paths in a phase space) are quite useful in applications that involve simulations. Furthermore, a trajectory that accounts for dynamics is more worthwhile in a decoupled approach than using a path that ignores dynamics, which has been an acceptable practice for decades. These issues will be elaborated upon further in Part IV.

The other interesting topic that emerges in Figure 8.1 is to develop feedback plans for problems in which there are no explicit models of dynamics or other differential constraints. If it was reasonable to solve problems in classical motion planning by ignoring differential constraints, one should certainly feel no less guilty designing feedback motion plans that still neglect differential constraints.[1] This uses the implicit model of uncertainty in predictability without altering any of the other assumptions previously applied in traditional motion planning.

Even if there are no unpredictability issues, another important use of feedback plans is for problems in which the initial state is not known. A feedback plan indicates what action to take from every state. Therefore, the specification of an initial condition is not important. The analog of this in graph algorithms is the single-destination shortest-path problem, which indicates how to arrive at a particular vertex optimally from any other vertex. Due to this connection, the next section presents feedback concepts for discrete state spaces, before extending the ideas to continuous spaces, which are needed for motion planning.

For these reasons, feedback motion planning is considered in this chapter. As a module in a decoupled approach used in robotics, feedback motion plans are at least as useful as a path computed by the previous techniques. We expect feedback solutions to be more reliable in general, when used in the place of open-loop paths computed by traditional motion planning algorithms.

8.2 Discrete State Spaces

This section is provided mainly to help to explain similar concepts that are coming in later sections. The presentation is limited to discrete spaces, which are much simpler to formulate and understand. Following this, an extension to configuration spaces and other continuous state spaces can be made. The discussion here is also relevant background for the feedback planning concepts that will be introduced in Section 8.4.1. In that case, uncertainty will be explicitly modeled. The resulting formulation and concepts can be considered as an extension of this section.

[1]Section 8.4.4 will actually consider some simple differential constraints, such as acceleration bounds; the full treatment of differential constraints is deferred until Part IV.

8.2.1 Defining a Feedback Plan

Consider a discrete planning problem similar to the ones defined in Formulations 2.1 and 2.3, except that the initial state is not given. Due to this, the cost functional cannot be expressed only as a function of a plan. It is instead defined in terms of the *state history* and *action history.* At stage k, these are defined as

$$\tilde{x}_k = (x_1, x_2, \ldots, x_k) \tag{8.1}$$

and

$$\tilde{u}_k = (u_1, u_2, \ldots, u_k), \tag{8.2}$$

respectively. Sometimes, it will be convenient to alternatively refer to $\tilde{x}_k$ as the *state trajectory.*

The resulting formulation is

Formulation 8.1 (Discrete Optimal Feedback Planning)

1. A finite, nonempty *state space* X.
2. For each state, $x \in X$, a finite *action space* $U(x)$.
3. A *state transition function* f that produces a state, $f(x,u) \in X$, for every $x \in X$ and $u \in U(x)$. Let U denote the union of $U(x)$ for all $x \in X$.
4. A set of *stages*, each denoted by k, that begins at $k = 1$ and continues indefinitely.
5. A *goal set*, $X_G \subset X$.
6. Let L denote a stage-additive *cost functional*,

   $$L(\tilde{x}_F, \tilde{u}_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F), \tag{8.3}$$

   in which $F = K + 1$.

There is one other difference in comparison to the formulations of Chapter 2. The state space is assumed here to be finite. This facilitates the construction of a feedback plan, but it is not necessary in general.

Consider defining a plan that solves Formulation 8.1. If the initial condition is given, then a sequence of actions could be specified, as in Chapter 2. Without having the initial condition, one possible approach is to determine a sequence of actions for each possible initial state, $x_1 \in X$. Once the initial state is given, the appropriate action sequence is known. This approach, however, wastes memory. Suppose some x is given as the initial state and the first action is applied, leading to the next state x'. What action should be applied from x'? The second action in the sequence at x can be used; however, we can also imagine that x' is now the initial state and use its first action. This implies that keeping an action sequence for every state is highly redundant. It is sufficient at each state to keep only the first action in the sequence. The application of that action produces the next state, at which the next appropriate action is stored. An execution sequence can be imagined from an initial state as follows. Start at some state, apply the action stored there, arrive at another state, apply its action, arrive at the next state, and so on, until the goal is reached.

It therefore seems appropriate to represent a feedback plan as a function that maps every state to an action. Therefore, a *feedback plan* π is defined as a function $\pi : X \rightarrow U$. From every state, $x \in X$, the plan indicates which action to apply. If the goal is reached, then the termination action should be applied. This is specified as part of the plan: $\pi(x) = u_T$, if $x \in X_G$. A feedback plan is called a *solution* to the problem if it causes the goal to be reached from every state that is reachable from the goal.

If an initial state x_1 and a feedback plan π are given, then the state and action histories can be determined. This implies that the execution cost, (8.3), also can be determined. It can therefore be alternatively expressed as $L(\pi, x_1)$, instead of $L(\tilde{x}_F, \tilde{u}_K)$. This relies on future states always being predictable. In Chapter 10, it will not be possible to make this direct correspondence due to uncertainties (see Section 10.1.3).

Feasibility and optimality The notions of feasible and optimal plans need to be reconsidered in the context of feedback planning because the initial condition is not given. A plan π is called a solution to the *feasible* planning problem if from *every* $x \in X$ from which X_G is reachable the goal set is indeed reached by executing π from x. This means that the cost functional is ignored (an alternative to Formulation 8.1 can be defined in which the cost functional is removed). For convenience, π will be called a *feasible* feedback plan.

Now consider optimality. From a given state x, it is clear that an optimal plan exists using the concepts of Section 2.3. Is it possible that a different optimal plan needs to be associated with every $x \in X$ that can reach X_G? It turns out that only one plan is needed to encode optimal paths from every initial state to X_G. Why is this true? Suppose that the optimal cost-to-go is computed over X using Dijkstra's algorithm or value iteration, as covered in Section 2.3. Every cost-to-go value at some $x \in X$ indicates the cost received under the implementation of the optimal open-loop plan from x. The first step in this optimal plan can be determined by (2.19), which yields a new state $x' = f(x, u)$. From x', (2.19) can be applied once again to determine the next optimal action. The cost at x' represents both the optimal cost-to-go if x' is the initial condition and also the optimal cost-to-go when continuing on the optimal path from x. The two must be equivalent because of the dynamic programming principle. Since all such costs must coincide, a single feedback plan can be used to obtain the optimal cost-to-go from every initial condition.

A feedback plan π is therefore defined as *optimal* if from *every* $x \in X$, the total

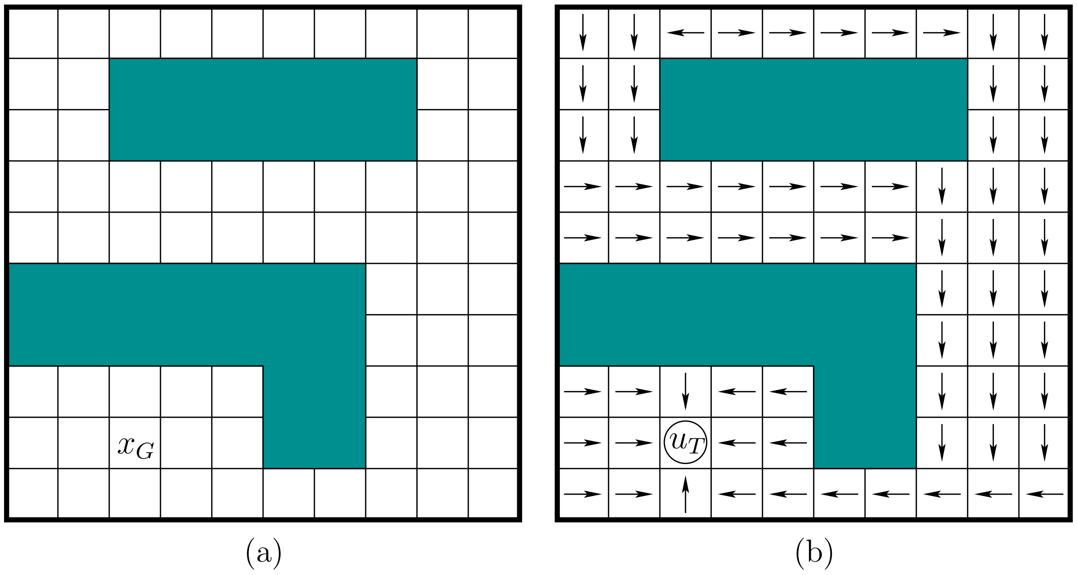


Figure 8.2: a) A 2D grid-planning problem. b) A solution feedback plan.

cost, $L(\pi, x)$, obtained by executing π is the lowest among all possible plans. The requirement that this holds for every initial condition is important for feedback planning.

Example 8.1 (Feedback Plan on a 2D Grid) This example uses the 2D grid model explained in Example 2.1. A robot moves on a grid, and the possible actions are up ($\uparrow$), down ($\downarrow$), left ($\leftarrow$), right ($\rightarrow$), and terminate (u_T); some directions are not available from some states. A solution feedback plan is depicted in Figure 8.2. Many other possible solutions plans exist. The one shown here happens to be optimal in terms of the number of steps to the goal. Some alternative feedback plans are also optimal (figure out which arrows can be changed). To apply the plan from any initial state, simply follow the arrows to the goal. In each stage, the application of the action represented by the arrow leads to the next state. The process terminates when u_T is applied at the goal. ■

8.2.2 Feedback Plans as Navigation Functions

It conveniently turns out that tools for computing a feedback plan were already given in Chapter 2. Methods such as Dijkstra's algorithm and value iteration produce information as a side effect that can be used to represent a feedback plan. This section explains how this information is converted into a feedback plan. To achieve this, a feedback plan will be alternatively expressed as a potential function over the state space (recall potential functions from Section 5.4.3). The potential values are computed by planning algorithms and can be used to recover the appropriate actions during execution. In some cases, an *optimal* feedback plan can even be represented using potential functions.

Navigation functions Consider a (discrete) *potential function*, defined as $\phi : X \to [0, \infty]$. The potential function can be used to define a feedback plan through the use of a *local operator*, which is a function that selects the action that reduces the potential as much as possible. First, consider the case of a feasible planning problem. The potential function, ϕ, defines a feedback plan by selecting u through the *local operator*,

$$u^* = \operatorname*{argmin}_{u \in U(x)} \Big\{ \phi(f(x,u)) \Big\}, \tag{8.4}$$

which means that $u^* \in U(x)$ is chosen to reduce ϕ as much as possible. The local operator yields a kind of *greedy* descent of the potential. Note that the action u^* may not be unique. In the continuous-space analog to this, the corresponding local operator performs a descent along the negative gradient (often referred to as *gradient descent*).

In the case of optimal planning, the local operator is defined as

$$u^* = \operatorname*{argmin}_{u \in U(x)} \Big\{ l(x,u) + \phi(f(x,u)) \Big\}, \tag{8.5}$$

which looks similar to the dynamic programming condition, (2.19). It becomes identical to (2.19) if ϕ is interpreted as the optimal cost-to-go. A simplification of (8.5) can be made if the planning problem is *isotropic*, which means that the cost is the same in every direction: $l(x,u) = l(x,u')$ for all $u, u' \in U(x) \backslash \{u_T\}$. In this case, the cost term $l(x,u)$ does not affect the minimization in (8.5). A common example in which this assumption applies is if the cost functional counts the number of stages required to reach the goal. The costs of particular actions chosen along the way are not important. Using the isotropic property, (8.5) simplifies back to (8.4).

When is a potential function useful? Many useless potential functions can be defined that fail to reach the goal, or cause states to cycle indefinitely, and so on. The most desirable potential function is one that for any initial state causes arrival in X_G, if it is reachable. This requires only a few simple properties. A potential function that satisfies these will be called a *navigation function*.[2]

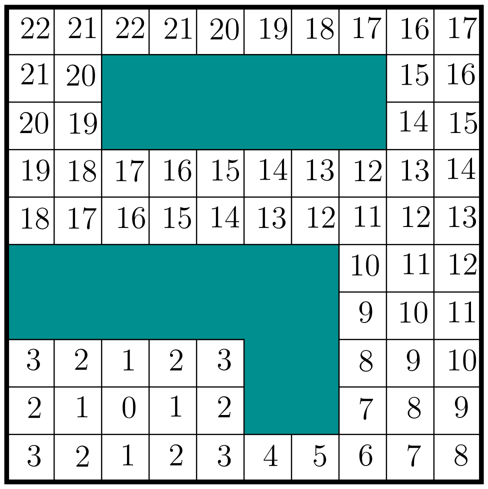
Suppose that the cost functional is isotropic. Let $x' = f(x, u^*)$, which is the state reached after applying the action $u^* \in U(x)$ that was selected by (8.4). A potential function, ϕ, is called a *(feasible) navigation function* if

1. $\phi(x) = 0$ for all $x \in X_G$.

2. $\phi(x) = \infty$ if and only if no point in X_G is reachable from x.

3. For every reachable state, $x \in X \setminus X_G$, the local operator produces a state x' for which $\phi(x') < \phi(x)$.

[2]This term was developed for continuous configuration spaces in [541, 829]; it will be used more broadly in this book but still retains the basic idea.



22	21	22	21	20	19	18	17	16	17
21	20	🟩	🟩	🟩	🟩	🟩	🟩	15	16
20	19	🟩	🟩	🟩	🟩	🟩	🟩	14	15
19	18	17	16	15	14	13	12	13	14
18	17	16	15	14	13	12	11	12	13
🟩	🟩	🟩	🟩	🟩	🟩	🟩	10	11	12
🟩	🟩	🟩	🟩	🟩	🟩	🟩	9	10	11
3	2	1	2	3	🟩	🟩	8	9	10
2	1	0	1	2	🟩	🟩	7	8	9
3	2	1	2	3	4	5	6	7	8

Figure 8.3: The cost-to-go values serve as a navigation function.

The first condition requires the goal to have zero potential (this condition is actually not necessary but is included for convenience). The second condition requires that ∞ serves as a special indicator that the goal is not reachable from some state. The third condition means that the potential function has no local minima except at the goal. This means that the execution of the resulting feedback plan will progress without cycling and the goal region will eventually be reached.

An *optimal navigation function* is defined as the optimal cost-to-go, G^*. This means that in addition to the three properties above, the navigation function must also satisfy the principle of optimality:

$$\phi(x) = \min_{u \in U(x)} \Big\{ l(x,u) + \phi(f(x,u)) \Big\}, \tag{8.6}$$

which is just (2.18) with G^* replaced by ϕ. See Section 15.2.1 for more on this connection.

Example 8.2 (Navigation Function on a 2D Grid) Return to the planning problem in Example 8.1. Assume that an isotropic cost model is used: $l(x,u) = 1$ if $u \neq u_T$. Figure 8.3 shows a navigation function. The numbers shown in the tiles represent ϕ. Verify that ϕ satisfies the three requirements for a navigation function.

At any state, an action is applied that reduces the potential value. This corresponds to selecting the action using (8.4). The process may be repeated from any state until X_G is reached. This example clearly illustrates how a navigation function can be used as an alternative definition of a feedback plan. ■

Example 8.3 (Airport Terminal) You may have found yourself using a navigation function to find the exit after arriving in an unfamiliar airport terminal. Many terminals are tree-structured, with increasing gate numbers as the distance to the terminal exit increases. If you wish to leave the terminal, you should normally walk toward the lower numbered gates. ■

Computing navigation functions There are many ways to compute navigation functions. The cost-to-go function determined by Dijkstra's algorithm working backward from X_G yields an *optimal navigation function*. The third condition of a navigation function under the anisotropic case is exactly the stationary dynamic programming equation, (2.18), if the navigation function ϕ is defined as the optimal cost-to-go G^*. It was mentioned previously that the optimal actions can be recovered using only the cost-to-go. This was actually an example of using a navigation function, and the resulting procedure could have been considered as a feedback plan.

If optimality is not important, then virtually any backward search algorithm from Section 2.2 can be used, provided that it records the distance to the goal from every reached state. The distance does not have to be optimal. It merely corresponds to the cost obtained if the current vertex in the search tree is traced back to the root vertex (or back to any vertex in X_G, if there are multiple goal states).

If the planning problem does not even include a cost functional, as in Formulation 2.1, then a cost functional can be invented for the purposes of constructing a navigation function. At each $x \in X$ from which X_G is reachable, the number of edges in the search graph that would be traversed from x to X_G can be stored as the cost. If Dijkstra's algorithm is used to construct the navigation function, then the resulting feedback plan yields executions that are shortest in terms of the number of stages required to reach the goal.

The navigation function itself serves as the representation of the feedback plan, by recovering the actions from the local operator. Thus, a function, $\pi : X \rightarrow U$, can be recovered from a navigation function, $\phi : X \rightarrow [0, \infty]$. Likewise, a navigation function, ϕ, can be constructed from π. Therefore, the π and ϕ can be considered as interchangeable representations of feedback plans.

8.2.3 Grid-Based Navigation Functions for Motion Planning

To consider feedback plans for continuous spaces, vector fields and other basic definitions from differential geometry will be needed. These will be covered in Section 8.3; however, before handling such complications, we first will describe how to use the ideas presented so far in Section 8.2 as a discrete approximation to feedback motion planning.

Examples 8.1 and 8.2 have already defined feedback plans and navigation functions for 2D grids that contain obstacles. Imagine that this model is used to approximate a motion planning problem for which $\mathcal{C} \subset \mathbb{R}^2$. Section 5.4.2 showed how to make a topological graph that approximates the motion planning problem

```
WAVEFRONT PROPAGATION ALGORITHM
1. Initialize $W_0 = X_G$; $i = 0$.
2. Initialize $W_{i+1} = \emptyset$.
3. For every $x \in W_i$, assign $\phi(x) = i$ and insert all unexplored neighbors of $x$ into $W_{i+1}$.
4. If $W_{i+1} = \emptyset$, then terminate; otherwise, let $i := i + 1$ and go to Step 2.
```

Figure 8.4: The wavefront propagation algorithm is a specialized version of Dijkstra's algorithm that optimizes the number of stages to reach the goal.

with a grid of samples. The motions used in Example 8.1 correspond to the 1-neighborhood definition, (5.37). This idea was further refined in Section 7.7.1 to model approximate optimal motion planning by moving on a grid; see Formulation 7.4. By choosing the Manhattan motion model, as defined in Example 7.4, a grid with the same motions considered in Example 8.1 is produced.

To construct a navigation function that may be useful in mobile robotics, a high-resolution (e.g., 50 to 100 points per axis) grid is usually required. In Section 5.4.2, only a few points per axis were needed because feedback was not assumed. It was possible in some instances to find a collision-free path by investigating only a few points per axis. During the execution of a feedback plan, it is assumed that the future states of the robot are not necessarily predictable. Wherever the robot may end up, the navigation function in combination with the local operator must produce the appropriate action. If the current state (or configuration) is approximated by a grid, then it is important to reduce the approximation error as much as possible. This is accomplished by setting the grid resolution high. In the feedback case, the grid can be viewed as "covering" the whole configuration space, whereas in Section 5.4.2 the grid only represented a topological graph of paths that cut across the space.[3]

Wavefront propagation algorithms Once the approximation has been made, any of the methods discussed in Section 8.2.2 can be used to compute a navigation function. An optimal navigation function can be easily computed using Dijkstra's algorithm from the goal. If each motion has unit cost, then a useful simplification can be made. Figure 8.4 describes a wavefront propagation algorithm that computes an optimal navigation function. It can be considered as a special case of Dijkstra's algorithm that avoids explicit construction of the priority queue. In Dijkstra's algorithm, the cost of the smallest element in the queue is monotonically

[3]Difficulty in distinguishing between these two caused researchers for many years to believe that grids yield terrible performance for the open-loop path planning problems of Chapter 5. This was mainly because it was assumed that a high-resolution grid was necessary. For many problems, however, they could terminate early after only considering a few points per axis.

nondecreasing during the execution of the algorithm. In the case of each motion having unit cost, there will be many states in the queue that have the same cost. Dijkstra's algorithm could remove in parallel all elements that have the same, smallest cost. Suppose the common, smallest cost value is i. These states are organized into a *wavefront*, W_i. The initial wavefront is W_0, which represents the states in X_G. The algorithm can immediately assign an optimal cost-to-go value of 1 to every state that can be reached in one stage from any state in W_0. These must be optimal because no other cost value is optimal. The states that receive cost 1 can be organized into the wavefront W_1. The unexplored neighbors of W_1 are assigned cost 2, which also must be optimal. This process repeats inductively from i to $i+1$ until all reachable states have been reached. In the end, the optimal cost-to-go is computed in $O(n)$ time, in which n is the number of reachable grid states. For any states that were not reached, the value $\phi(x) = \infty$ can be assigned. The navigation function shown in Figure 8.3 can actually be computed using the wavefront propagation algorithm.

Maximum clearance One problem that typically arises in mobile robotics is that optimal motion plans bring robots too close to obstacles. Recall from Section 6.2.4 that the shortest Euclidean paths for motion planning in a polygonal environment must be allowed to touch obstacle vertices. This motivated the maximum clearance roadmap, which was covered in Section 6.2.3. A grid-based approximate version of the maximum clearance roadmap can be made. Furthermore, a navigation function can be defined that guides the robot onto the roadmap, then travels along the roadmap, and finally deposits the robot at a specified goal. In [588], the resulting navigation function is called *NF2*.

Assume that there is a single goal state, $x_G \in X$. The computation of a maximum clearance navigation function proceeds as follows:

1. Instead of X_G, assign W_0 to be the set of all states from which motion in at least one direction is blocked. These are the states on the boundary of the discretized collision-free space.

2. Perform wavefront iterations that propagate costs in waves outward from the obstacle boundaries.

3. As the wavefronts propagate, they will meet approximately at the location of the maximum clearance roadmap for the original, continuous problem. Mark any state at which two wavefront points arrive from opposing directions as a *skeleton state*. It may be the case that the wavefronts simply touch each other, rather than arriving at a common state; in this case, one of the two touching states is chosen as the skeleton state. Let S denote the set of all skeleton states.

4. After the wavefront propagation ends, connect x_G to the skeleton by inserting x_G and all states along the path to the skeleton into S. This path can be found using any search algorithm.

5. Compute a navigation function ϕ_1 over S by treating all other states as if they were obstacles and using the wavefront propagation algorithm. This navigation function guides any point in S to the goal.

6. Treat S as a goal region and compute a navigation function ϕ_2 using the wavefront propagation algorithm. This navigation function guides the state to the nearest point on the skeleton.

7. Combine ϕ_1 and ϕ_2 as follows to obtain ϕ. For every $x \in S$, let $\phi(x) = \phi_1(x)$. For every remaining state, the value $\phi(x) = \phi_1(x') + \phi_2(x)$ is assigned, in which x' is the nearest state to x such that $x' \in S$. The state x' can easily be recorded while ϕ_2 is computed.

If $\mathcal{C}_{free}$ is multiply connected, then there may be multiple ways to each x_G by traveling around different obstacles (the paths are not homotopic). The method described above does not take into account the problem that one route may have a tighter clearance than another. The given approach only optimizes the distance traveled along the skeleton; it does not, however, maximize the nearest approach to an obstacle, if there are multiple routes.

Dial's algorithm Now consider generalizing the wavefront propagation idea. Wavefront propagation can be applied to any discrete planning problem if $l(x, u) = 1$ for any $x \in X$ and $u \in U(x)$ (except $u = u_T$). It is most useful when the transition graph is sparse (imagine representing the transition graph using an adjacency matrix). The grid problem is a perfect example where this becomes important. More generally, if the cost terms assume integer values, then *Dial's algorithm* [272] results, which is a generalization of wavefront propagation, and a specialization of Dijkstra's algorithm. The idea is that the priority queue can be avoided by assigning the alive vertices to buckets that correspond to different possible cost-to-go values. In the wavefront propagation case, there are never more than two buckets needed at a time. Dial's algorithm allows all states in the smallest cost bucket to be processed in parallel. The scheme was enhanced in [939] to yield a linear-time algorithm.

Other extensions Several ideas from this section can be generalized to produce other navigation functions. One disadvantage of the methods discussed so far is that undesirable staircase motions (as shown in Figure 7.40) are produced. If the 2-neighborhood, as defined in (5.38), is used to define the action spaces, then the motions will generally be shorter. Dial's algorithm can be applied to efficiently compute an optimal navigation function in this case.

A grid approximation can be made to higher dimensional configuration spaces. Since a high resolution is needed, however, it is practical only for a few dimensions (e.g., 3 or 4). If the 1-neighborhood is used, then wavefront propagation can be easily applied to compute navigation functions. Dial's algorithm can be adapted for general k-neighborhoods.

Constructing navigation functions over grids may provide a practical solution in many applications. In other cases it may be unacceptable that staircase motions occur. In many cases, it may not even be possible to compute the navigation function quickly enough. Factors that influence this problem are 1) very high accuracy, and a hence high-resolution grid may be necessary; 2) the dimension of the configuration space may be high; and 3) the environment may be frequently changing, and a real-time response is required. To address these issues, it is appealing to abandon grid approximations. This will require defining potential functions and velocities directly on the configuration space. Section 8.3 presents the background mathematical concepts to make this transition.

8.3 Vector Fields and Integral Curves

To consider feedback motion plans over continuous state spaces, including configuration spaces, we will need to define a vector field and the trajectory that is obtained by integrating the vector field from an initial point. A vector field is ideal for characterizing a feedback plan over a continuous state space. It can be viewed as providing the continuous-space analog to the feedback plans on grids, as shown in Figure 8.2b.

This section presents two alternative presentations of the background mathematical concepts. Section 8.3.1 assumes that $X = \mathbb{R}^n$, which leads to definitions that appear very similar to those you may have learned in basic calculus and differential equations courses. Section 8.3.2 covers the more general case of vector fields on manifolds. This requires significantly more technical concepts and builds on the manifold definitions of Section 4.1.2.

Some readers may have already had some background in differentiable manifolds. If, however, you are seeing it for the first time, then it may be difficult to comprehend on the first reading. In addition to rereading, here are two other suggestions. First, try studying background material on this subject, which is suggested at the end of the chapter. Second, disregard the manifold technicalities in the subsequent sections and pretend that $X = \mathcal{C} = \mathbb{R}^n$. Nearly everything will make sense without the additional technicalities. Imagine that a manifold is defined as a cube, $[0,1]^n$, with some sides identified, as in Section 4.1.2. The concepts that were presented for $\mathbb{R}^n$ can be applied everywhere except at the boundary of the cube. For example, if $\mathbb{S}^1$ is defined as $[0,1]/\sim$, and a function f is defined on $\mathbb{S}^1$, how can we define the derivative at $f(0)$? The technical definitions of Section 8.3.2 fix this problem. Sometimes, the technicalities can be avoided in practice by cleverly handling the identification points.

8.3.1 Vector Fields on $\mathbb{R}^n$

This section revisits some basic concepts from introductory courses such as calculus, linear algebra, and differential equations. You may have learned most of these for $\mathbb{R}^2$ and $\mathbb{R}^3$. We eventually want to describe velocities in $\mathbb{R}^n$ and on manifolds, and then use the notion of a vector field to express a feedback plan in Section 8.4.1.

Vector spaces Before defining a vector field, it is helpful to be precise about what is meant by a *vector*. A *vector space* (or *linear space*) is defined as a set, V, that is closed under two algebraic operations called *vector addition* and *scalar multiplication* and satisfies several axioms, which will be given shortly. The vector space used in this section is $\mathbb{R}^n$, in which the scalars are real numbers, and a vector is represented as a sequence of n real numbers. Scalar multiplication multiplies each component of the vector by the scalar value. Vector addition forms a new vector by adding each component of two vectors.

A *vector space* V can be defined over any field $\mathbb{F}$ (recall the definition from Section 4.4.1). The field $\mathbb{F}$ represents the *scalars*, and V represents the *vectors*. The concepts presented below generalize the familiar case of the vector space $\mathbb{R}^n$. In this case, $V = \mathbb{R}^n$ and $\mathbb{F} = \mathbb{R}$. In the definitions that follow, you may make these substitutions, if desired. We will not develop vector spaces that are more general than this; the definitions are nevertheless given in terms of V and $\mathbb{F}$ to clearly separate scalars from vectors. The *vector addition* is denoted by $+$, and the *scalar multiplication* is denoted by $\cdot$. These operations must satisfy the following axioms (a good exercise is to verify these for the case of $\mathbb{R}^n$ treated as a vector space over the field $\mathbb{R}$):

1. (**Commutative Group Under Vector Addition**) The set V is a commutative group with respect to vector addition, $+$.

2. (**Associativity of Scalar Multiplication**) For any $v \in V$ and any $\alpha, \beta \in \mathbb{F}$, $\alpha(\beta v) = (\alpha\beta)v$.

3. (**Distributivity of Scalar Sums**) For any $v \in V$ and any $\alpha, \beta \in \mathbb{F}$, $(\alpha + \beta)v = \alpha v + \beta v$.

4. (**Distributivity of Vector Sums**) For any $v, w \in V$ and any $\alpha \in \mathbb{F}$, $\alpha(v + w) = \alpha v + \alpha w$.

5. (**Scalar Multiplication Identity**) For any $v \in V$, $1v = v$ for the multiplicative identity $1 \in \mathbb{F}$.

The first axiom allows vectors to be added in any order. The rest of the axioms require that the scalar multiplication interacts with vectors in the way that we would expect from the familiar vector space $\mathbb{R}^n$ over $\mathbb{R}$.

A *basis* of a vector space V is defined as a set, $v_1,\ldots,v_n$, of vectors for which every $v \in V$ can be uniquely written as a *linear combination*:

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n, \tag{8.7}$$

for some $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$. This means that every vector has a unique representation as a linear combination of basis elements. In the case of $\mathbb{R}^3$, a familiar basis is

[0 0 1], [0 1 0], and [1 0 0]. All vectors can be expressed as a linear combination of these three. Remember that a basis is not necessarily unique. From linear algebra, recall that any three linearly independent vectors can be used as a basis for $\mathbb{R}^3$. In general, the basis must only include linearly independent vectors. Even though a basis is not necessarily unique, the number of vectors in a basis is the same for any possible basis over the same vector space. This number, n, is called the *dimension* of the vector space. Thus, we can call $\mathbb{R}^n$ an n-dimensional vector space over $\mathbb{R}$.

Example 8.4 (The Vector Space $\mathbb{R}^n$ Over $\mathbb{R}$) As indicated already, $\mathbb{R}^n$ can be considered as a vector space. A natural basis is the set of n vectors in which, for each $i \in \{1, \ldots, n\}$, a unit vector is constructed as follows. Let $x_i = 1$ and $x_j = 0$ for all $j \neq i$. Since there are n basis vectors, $\mathbb{R}^n$ is an n-dimensional vector space. The basis is not unique. Any set of n linearly independent vectors may be used, which is familiar from linear algebra, in which nonsingular $n \times n$ matrices are used to transform between them. ■

To illustrate the power of these general vector space definitions, consider the following example.

Example 8.5 (A Vector Space of Functions) The set of all continuous, real-valued functions $f : [0, 1] \rightarrow \mathbb{R}$, for which

$$\int_0^1 f(x)dx \tag{8.8}$$

is finite, forms a vector space over $\mathbb{R}$. It is straightforward to verify that the vector space axioms are satisfied. For example, if two functions f_1 and f_2 are added, the integral remains finite. Furthermore, $f_1 + f_2 = f_2 + f_1$, and all of the group axioms are satisfied with respect to addition. Any function f that satisfies (8.8) can be multiplied by a scalar in $\mathbb{R}$, and the integral remains finite. The axioms that involve scalar multiplication can also be verified.

It turns out that this vector space is infinite-dimensional. One way to see this is to restrict the functions to the set of all those for which the Taylor series exists and converges to the function (these are called *analytic functions*). Each function can be expressed via a Taylor series as a polynomial that may have an infinite number of terms. The set of all monomials, x, x^2, x^3, and so on, represents a basis. Every continuous function can be considered as an infinite vector of coefficients; each coefficient is multiplied by one of the monomials to produce the function. This provides a simple example of a *function space*; with some additional definitions, this leads to a *Hilbert space*, which is crucial in functional analysis, a subject that characterizes spaces of functions [836, 838]. ■

The remainder of this chapter considers only finite-dimensional vector spaces over $\mathbb{R}$. It is important, however, to keep in mind the basic properties of vector spaces that have been provided.

Vector fields A vector field looks like a "needle diagram" over $\mathbb{R}^n$, as depicted in Figure 8.5. The idea is to specify a direction at each point $p \in \mathbb{R}^n$. When used to represent a feedback plan, it indicates the direction that the robot needs to move if it finds itself at p.

For every $p \in \mathbb{R}^n$, associate an n-dimensional vector space called the *tangent space* at p, which is denoted as $T_p(\mathbb{R}^n)$. Why not just call it a vector space at p? The use of the word "tangent" here might seem odd; it is motivated by the generalization to manifolds, for which the tangent spaces will be "tangent" to points on the manifold.

A *vector field*[4] $\vec{V}$ on $\mathbb{R}^n$ is a function that assigns a vector $v \in T_p(\mathbb{R}^n)$ to every $p \in \mathbb{R}^n$. What is the range of this function? The vector $\vec{V}(p)$ at each $p \in \mathbb{R}^n$ actually belongs to a different tangent space. The range of the function is therefore the union

$$T(\mathbb{R}^n) = \bigcup_{p \in \mathbb{R}^n} T_p(\mathbb{R}^n), \tag{8.9}$$

which is called the *tangent bundle* on $\mathbb{R}^n$. Even though the way we describe vectors from $T_p(\mathbb{R}^n)$ may appear the same for any $p \in \mathbb{R}^n$, each tangent space is assumed to produce distinct vectors. To maintain distinctness, a point in the tangent bundle can be expressed with $2n$ coordinates, by specifying p and v together. This will become important for defining phase space concepts in Part IV. In the present setting, it is sufficient to think of the range of $\vec{V}$ as $\mathbb{R}^n$ because $T_p(\mathbb{R}^n) = \mathbb{R}^n$ for every $p \in \mathbb{R}^n$.

A vector field can therefore be expressed using n real-valued functions on $\mathbb{R}^n$. Let $f_i(x_1, \ldots, x_n)$ for i from 1 to n denote such functions. Using these, a vector field is specified as

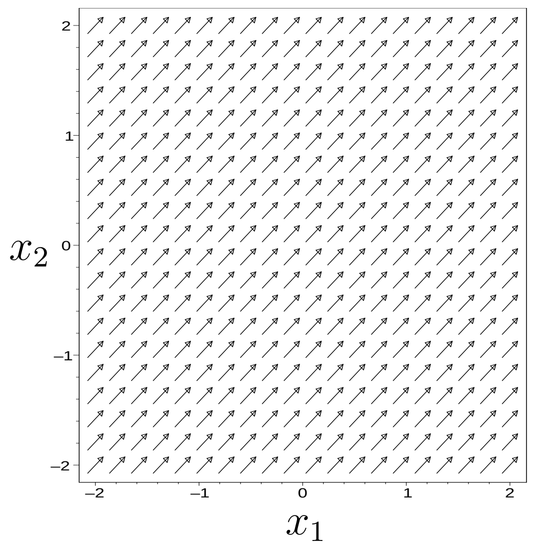
$$f(x) = [f_1(x_1, \ldots, x_n) \;\; f_2(x_1, \ldots, x_n) \;\; \cdots \;\; f_n(x_1, \ldots, x_n)]. \tag{8.10}$$

In this case, it appears that a vector field is a function f from $\mathbb{R}^n$ into $\mathbb{R}^n$. Therefore, standard function notation will be used from this point onward to denote a vector field.

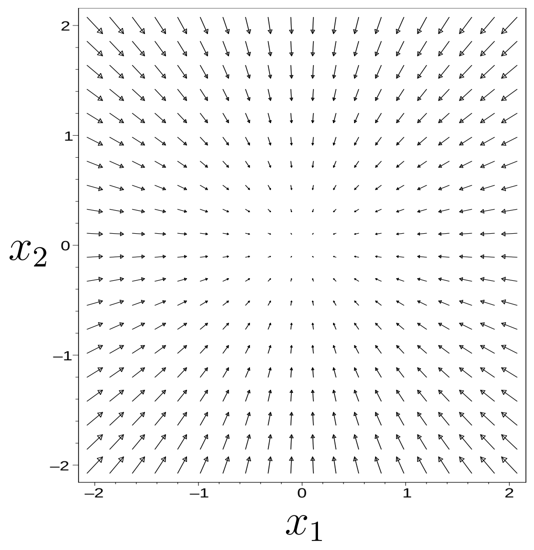
Now consider some examples of vector fields over $\mathbb{R}^2$. Let a point in $\mathbb{R}^2$ be represented as $p = (x, y)$. In standard vector calculus, a vector field is often specified as $[f_1(x,y) \;\; f_2(x,y)]$, in which f_1 and f_2 are functions on $\mathbb{R}^2$

Example 8.6 (Constant Vector Field) Figure 8.5a shows a *constant vector field*, which assigns the vector $[1 \;\; 2]$ to every $(x,y) \in \mathbb{R}^2$. $\blacksquare$

[4]Unfortunately, the term *field* appears in two unrelated places: in the definition of a vector space and in the term *vector field*. Keep in mind that this is an accidental collision of terms.



(a)



(b)

Figure 8.5: (a) A constant vector field, $f(x, y) = [1 \quad 1]$. (b) A vector field, $f(x, y) = [-x \quad -y]$ in which all vectors point to the origin.

Example 8.7 (Inward Flow) Figure 8.5b depicts a vector field that assigns $[-x \quad -y]$ to every $(x, y) \in \mathbb{R}^2$. This causes all vectors to point to the origin. ■

Example 8.8 (Swirl) The vector field in Figure 8.6 assigns $[(y - x) \quad (-x - y)]$ to every $(x, y) \in \mathbb{R}^2$. ■

Due to obstacles that arise in planning problems, it will be convenient to sometimes restrict the domain of a vector field to an open subset of $\mathbb{R}^n$. Thus, for any open subset $O \subset \mathbb{R}^n$, a vector field $f : O \to \mathbb{R}^n$ can be defined.

Smoothness A function f_i from a subset of $\mathbb{R}^n$ into $\mathbb{R}$ is called a *smooth function* if derivatives of any order can be taken with respect to any variables, at any point in the domain of f_i. A vector field is said to be *smooth* if every one of its n defining functions, $f_1, \ldots, f_n$, is smooth. An alternative name for a smooth function is a *C^∞ function.* The superscript represents the order of differentiation that can be taken. For a *C^k function*, its derivatives can be taken at least up to order k. A *C^0 function* is an alternative name for a continuous function. The notion of a homeomorphism can be extended to a *diffeomorphism*, which is a homeomorphism that is a smooth function. Two topological spaces are called *diffeomorphic* if there exists a diffeomorphism between them.

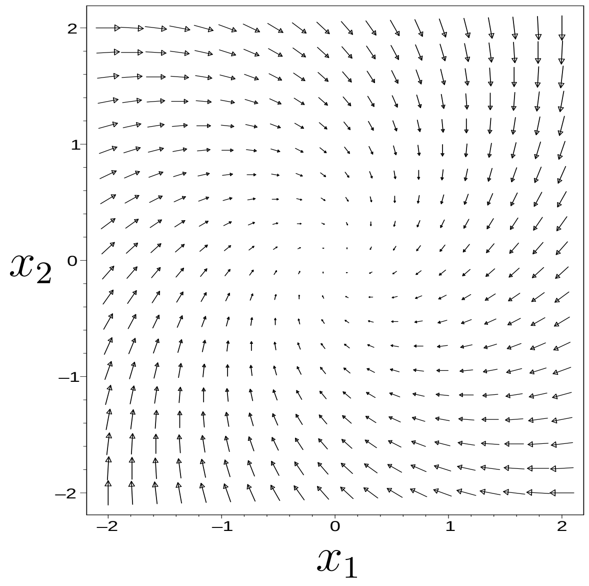


Figure 8.6: A swirling vector field, $f(x, y) = [(y - x) \;\; (-x - y)]$.

Vector fields as velocity fields We now give a particular interpretation to vector fields. A vector field expressed using (8.10) can be used to define a set of first-order differential equations as

$$\begin{aligned} \frac{dx_1}{dt} &= f_1(x_1, \ldots, x_n) \\ \frac{dx_2}{dt} &= f_2(x_1, \ldots, x_n) \\ &\vdots \\ \frac{dx_n}{dt} &= f_n(x_1, \ldots, x_n). \end{aligned} \tag{8.11}$$

Each equation represents the derivative of one coordinate with respect to time. For any point $x \in \mathbb{R}^n$, a *velocity vector* is defined as

$$\frac{dx}{dt} = \left[\frac{dx_1}{dt} \;\; \frac{dx_2}{dt} \;\; \cdots \;\; \frac{dx_n}{dt} \right]. \tag{8.12}$$

This enables f to be interpreted as a *velocity field*.

It is customary to use the short notation $\dot{x} = dx/dt$. Each velocity component can be shortened to $\dot{x}_i = dx_i/dt$. Using f to denote the vector of functions f_1, ..., f_n, (8.11) can be shorted to

$$\dot{x} = f(x). \tag{8.13}$$

The use of f here is an intentional coincidence with the use of f for the state transition equation. In Part IV, we will allow vector fields to be parameterized by actions. This leads to a continuous-time state transition equation that looks like

$\dot{x} = f(x, u)$ and is very similar to the transition equations defined over discrete stages in Chapter 2.

The differential equations expressed in (8.11) are often referred to as *autonomous* or *stationary* because f does not depend on time. A time-varying vector field could alternatively be defined, which yields $\dot{x} = f(x(t), t)$. This will not be covered, however, in this chapter.

An integral curve If a vector field f is given, then a velocity vector is defined at each point using (8.10). Imagine a point that starts at some $x_0 \in \mathbb{R}^n$ at time $t = 0$ and then moves according to the velocities expressed in f. Where should it travel? Its *trajectory* starting from x_0 can be expressed as a function $\tau : [0, \infty) \rightarrow \mathbb{R}^n$, in which the domain is a time interval, $[0, \infty)$. A trajectory represents an *integral curve* (or *solution trajectory*) of the differential equations with initial condition $\tau(0) = x_0$ if

$$\frac{d\tau}{dt}(t) = f(\tau(t)) \tag{8.14}$$

for every time $t \in [0, \infty)$. This is sometimes expressed in integral form as

$$\tau(t) = x_0 + \int_0^t f(\tau(s))ds \tag{8.15}$$

and is called a solution to the differential equations in the *sense of Caratheodory.* Intuitively, the integral curve starts at x_0 and flows along the directions indicated by the velocity vectors. This can be considered as the continuous-space analog of following the arrows in the discrete case, as depicted in Figure 8.2b.

Example 8.9 (Integral Curve for a Constant Velocity Field) The simplest case is a constant vector field. Suppose that a constant field $x_1 = 1$ and $x_2 = 2$ is defined on $\mathbb{R}^2$. The integral curve from $(0, 0)$ is $\tau(t) = (t, 2t)$. It can be easily seen that (8.14) holds for all $t \geq 0$. ■

Example 8.10 (Integral Curve for a Linear Velocity Field) Consider a velocity field on $\mathbb{R}^2$. Let $\dot{x}_1 = -2x_1$ and $\dot{x}_2 = -x_2$. The function $\tau(t) = (e^{-2t}, e^{-t})$ represents the integral curve from $(1, 1)$. At $t = 0$, $\tau(0) = (1, 1)$, which is the initial state. If can be verified that for all $t > 0$, (8.14) holds. This is a simple example of a linear velocity field. In general, if each f_i is a linear function of the coordinate variables x_1, …, x_n, then a linear velocity field is obtained. The integral curve is generally found by determining the eigenvalues of the matrix A when the velocity field is expressed as $\dot{x} = Ax$. See [192] for numerous examples. ■

A basic result from differential equations is that a unique integral curve exists to $\dot{x} = f(x)$ if f is smooth. An alternative condition is that a unique solution exists if f satisfies a *Lipschitz condition*. This means that there exists some constant $c \in (0, \infty)$ such that

$$\|f(x) - f(x')\| \leq c\|x - x'\| \tag{8.16}$$

for all $x, x' \in X$, and $\|\cdot\|$ denotes the Euclidean norm (vector magnitude). The constant c is often called a *Lipschitz constant*. Note that if f satisfies the Lipschitz condition, then it is continuous. Also, if all partial derivatives of f over all of X can be bounded by a constant, then f is Lipschitz. The expression in (8.16) is preferred, however, because it is more general (it does not even imply that f is differentiable everywhere).

Piecewise-smooth vector fields It will be important to allow vector fields that are smooth only over a finite number of patches. At a *switching boundary* between two patches, a discontinuous jump may occur. For example, suppose that an $(n-1)$-dimensional switching boundary, $S \subset \mathbb{R}^n$, is defined as

$$S = \{x \in \mathbb{R}^n |\; s(x) = 0\}, \tag{8.17}$$

in which s is a function $s : \mathbb{R}^n \rightarrow \mathbb{R}$. If $\mathbb{R}^n$ has dimension n and s is not singular, then S has dimension $n-1$. Define

$$S_+ = \{x \in \mathbb{R}^n |\; s(x) > 0\} \tag{8.18}$$

and

$$S_- = \{x \in \mathbb{R}^n |\; s(x) < 0\}. \tag{8.19}$$

The definitions are similar to the construction of implicit models using geometric primitives in Section 3.1.2. Suppose that $f(x)$ is smooth over S_+ and S_- but experiences a discontinuous jump at S. Such differential equations model *hybrid systems* in control theory [137, 409, 634]. The task there is to design a *hybrid control system*. Can we still determine a solution trajectory in this case? Under special conditions, we can obtain what is called a solution to the differential equations in the *sense of Filipov* [338, 846].

Let $B(x, \delta)$ denote an open ball of radius δ centered at x. Let $f(B(x, \delta))$ denote the set

$$f(B(x, \delta)) = \{x' \in X \mid \exists x'' \in B(x, \delta) \text{ for which } x' = f(x'')\}. \tag{8.20}$$

Let X_0 denote any subset of $\mathbb{R}^n$ that has measure zero (i.e., $\mu(X_0) = 0$). Let hull(A) denote the convex hull of a set, A, of points in $\mathbb{R}^n$. A path $\tau : [0, t_f] \rightarrow \mathbb{R}^n$ is called a *solution in the sense of Filipov* if for almost all $t \in [0, t_f]$,

$$\frac{d\tau}{dt}(t) \in \bigcap_{\delta > 0} \left\{ \bigcap_{X_0} \text{hull}(f(B(\tau(t), \delta) \setminus X_0)) \right\}, \tag{8.21}$$

in which the intersections are taken over all possible $\delta > 0$ and sets, X_0, of measure zero. The expression (8.21) is actually called a *differential inclusion* [53] because

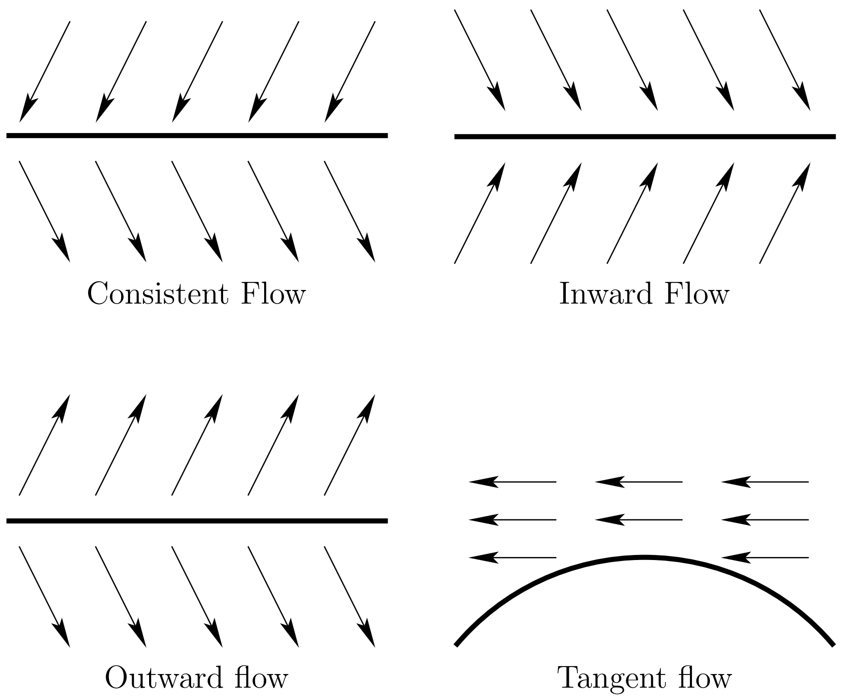


Figure 8.7: Different kinds of flows around a switching boundary.

a set of choices is possible for $\dot{x}$. The "for almost all" requirement means that the condition can even fail to hold on a set of measure zero in $[0, t_f]$. Intuitively, it says that almost all of the velocity vectors produced by τ must point "between" the velocity vectors given by f in the vicinity of $\tau(x(t))$. The "between" part comes from using the convex hull. Filipov's sense of solution is an incredible generalization of the solution concept in the sense of Caratheodory. In that case, every velocity vector produced by τ must agree with $f(x(t))$, as given in (8.14). The condition in (8.21) allows all sorts of sloppiness to appear in the solution, even permitting f to be discontinuous.

Many bizarre vector fields can yield solutions in the sense of Filipov. The switching boundary model is relatively simple among those permitted by Filipov's condition. Figure 8.7 shows various cases that can occur at the switching boundary S. For the case of consistent flow, solutions occur as you may intuitively expect. Filipov's condition, (8.21), requires that at S the velocity vector of τ points between vectors before and after crossing S (for example, it can point down, which is the average of the two directions). The magnitude must also be between the two magnitudes. For the inward flow case, the integral curve moves along S, assuming the vectors inside of S point in the same direction (within the convex hull) as the vectors on either side of the boundary. In applications that involve physical systems, this may lead to oscillations around S. This can be alleviated by regularization, which thickens the boundary [846] (the subject of *sliding-mode control* addresses this issue [303]). The outward flow case can lead to nonuniqueness if the initial state lies in S. However, trajectories that start outside of S will not cross S, and there will be no such troubles. If the flow is tangent on both sides of a boundary, then other forms of nonuniqueness may occur. The tangent-flow case will be avoided in this chapter.

8.3.2 Smooth Manifolds

The manifold definition given in Section 4.1.2 is often called a *topological manifold*. A manifold defined in this way does not necessarily have enough axioms to ensure that calculus operations, such as differentiation and integration, can be performed. We would like to talk about velocities on the configuration space $\mathcal{C}$ or in general for a continuous state space X. As seen in Chapter 4, the configuration space could be a manifold such as $\mathbb{RP}^3$. Therefore, we need to define some more qualities that a manifold should possess to enable calculus. This leads to the notion of a *smooth manifold*.

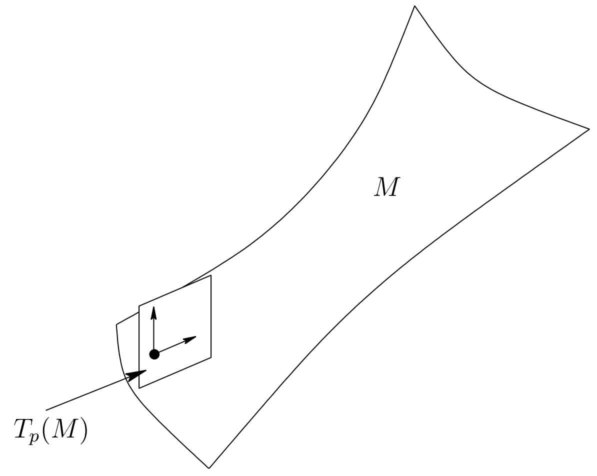


Figure 8.8: Intuitively, the tangent space is a linear approximation to the manifold in a neighborhood around p.

Assume that M is a topological manifold, as defined in Section 4.1.2. For example, M could represent $SO(3)$, the set of all rotation matrices for $\mathbb{R}^3$. A simpler example that will be helpful to keep in mind is $M = \mathbb{S}^2$, which is a sphere in $\mathbb{R}^3$. We want to extend the concepts of Section 8.3.1 from $\mathbb{R}^n$ to manifolds. One of the first definitions will be the tangent space $\mathbb{T}_p(M)$ at a point $p \in M$. As you might imagine intuitively, the tangent vectors are tangent to a surface, as shown in Figure 8.8. These will indicate possible velocities with which we can move along the manifold from p. This is more difficult to define for a manifold than for $\mathbb{R}^n$ because it is easy to express any point in $\mathbb{R}^n$ using n coordinates, and all local coordinate frames for the tangent spaces at every $p \in \mathbb{R}^n$ are perfectly aligned with each other. For a manifold such as $\mathbb{S}^2$, we must define tangent spaces in a way that is not sensitive to coordinates and handles the fact that the tangent plane rotates as we move around on $\mathbb{S}^2$.

First think carefully about what it means to assign coordinates to a manifold. Suppose M has dimension n and is embedded in $\mathbb{R}^m$. For $M = SO(3)$, $n = 3$ and $m = 9$. For $M = \mathbb{S}^2$, $n = 2$ and $m = 3$. The number of coordinates should be n, the dimension of M; however, manifolds embedded in $\mathbb{R}^m$ are often expressed as a subset of $\mathbb{R}^m$ for which some equality constraints must be obeyed. We would like to express some part of M in terms of coordinates in $\mathbb{R}^n$.

Coordinates and parameterizations For any open set $U \subseteq M$ and function $\phi : U \to \mathbb{R}^n$ such that ϕ is a homeomorphism onto a subset of $\mathbb{R}^n$, the pair (U, ϕ) is called a *coordinate neighborhood* (or *chart* in some literature). The values $\phi(p)$ for some $p \in U$ are called the *coordinates* of p.

Example 8.11 (Coordinate Neighborhoods on $\mathbb{S}^1$) A simple example can be obtained for the circle $M = \mathbb{S}^1$. Suppose M is expressed as the unit circle embedded in $\mathbb{R}^2$ (the set of solutions to $x^2 + y^2 = 1$). Let (x, y) denote a point in $\mathbb{R}^2$. Let U be the subset of $\mathbb{S}^1$ for which $x > 0$. A coordinate function $\phi : U \to (-\pi/2, \pi/2)$, can be defined as $\phi(x, y) = \tan^{-1}(y/x)$.

Let $W = \phi(U)$ (the range of ϕ) for some coordinate neighborhood (U, ϕ). Since U and W are homeomorphic via ϕ, the inverse function ϕ^{-1} can also be defined. It turns out that the inverse is the familiar idea of a *parameterization*. Continuing Example 8.11, ϕ^{-1} yields the mapping $\theta \mapsto (\cos\theta, \sin\theta)$, which is the familiar parameterization of the circle but restricted to $\theta \in (-\pi/2, \pi/2)$. ■

To make differentiation work at a point $p \in M$, it will be important to have a coordinate neighborhood defined over an open subset of M that contains p. This is mainly because defining derivatives of a function at a point requires that an open set exists around the point. If the coordinates appear to have no boundary, then this will be possible. It is unfortunately not possible to cover all of M with a single coordinate neighborhood, unless $M = \mathbb{R}^n$ (or M is at least homeomorphic to $\mathbb{R}^n$). We must therefore define multiple neighborhoods for which the domains cover all of M. Since every domain is an open set, some of these domains must overlap. What happens in this case? We may have two or more alternative coordinates for the same point. Moving from one set of coordinates to another is the familiar operation used in calculus called a *change of coordinates*. This will now be formalized.

Suppose that (U, ϕ) and (V, ψ) are coordinate neighborhoods on some manifold M, and $U \cap V \neq \emptyset$. Figure 8.9 indicates how to change coordinates from ϕ to ψ. This change of coordinates is expressed using function composition as $\psi \circ \phi^{-1} : \mathbb{R}^n \to \mathbb{R}^n$ (ϕ^{-1} maps from $\mathbb{R}^n$ into M, and ψ maps from a subset of M to $\mathbb{R}^n$).

Example 8.12 (Change of Coordinates) Consider changing from Euler angles to quaternions for $M = SO(3)$. Since $SO(3)$ is a 3D manifold, $n = 3$. This means that any coordinate neighborhood must map a point in $SO(3)$ to a point in $\mathbb{R}^3$. We can construct a coordinate function $\phi : SO(3) \to \mathbb{R}^3$ by computing Euler angles from a given rotation matrix. The functions are actually defined in (3.47),

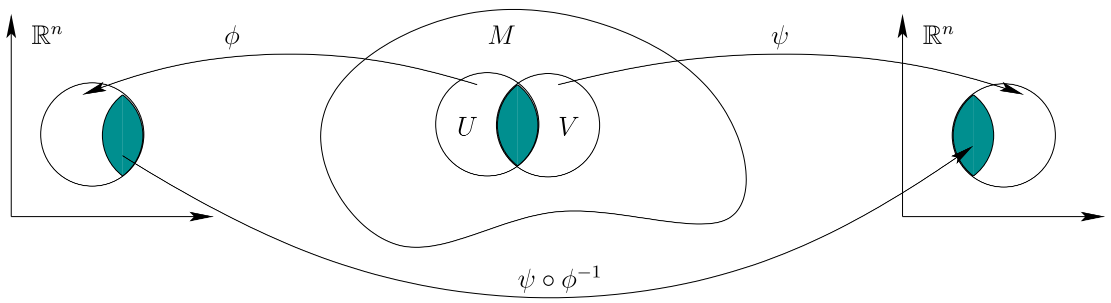


Figure 8.9: An illustration of a change of coordinates.

(3.48), and (3.49). To make this a coordinate neighborhood, an open subset U of M must be specified.

We can construct another coordinate function $\psi : SO(3) \to \mathbb{R}^3$ by using quaternions. This may appear to be a problem because quaternions have four components; however, the fourth component can be determined from the other three. Using (4.24) to (4.26), the a, b, and c coordinates can be determined.

Now suppose that we would like to change from Euler angles to quaternions in the overlap region $U \cap V$, in which V is an open set on which the coordinate neighborhood for quaternions is defined. The task is to construct a change of coordinates, $\psi \circ \phi^{-1}$. We first have to invert ϕ over U. This means that we instead need a parameterization of M in terms of Euler angles. This is given by (3.42), which yields a rotation matrix, $\phi^{-1}(\alpha, \beta, \gamma) \in SO(3)$ for α, β, and γ. Once this matrix is determined, then ψ can be applied to it to determine the quaternion parameters, a, b, and c. This means that we have constructed three real-valued functions, f_1, f_2, and f_3, which yield $a = f_1(\alpha, \beta, \gamma)$, $b = f_2(\alpha, \beta, \gamma)$, and $c = f_3(\alpha, \beta, \gamma)$. Together, these define $\psi \circ \phi^{-1}$. ■

There are several reasons for performing coordinate changes in various contexts. Example 8.12 is motivated by a change that frequently occurs in motion planning. Imagine, for example, that a graphics package displays objects using quaternions, but a collision-detection algorithm uses Euler angles. It may be necessary in such cases to frequently change coordinates. From studies of calculus, you may recall changing coordinates to simplify an integral. In the definition of a smooth manifold, another motivation arises. Since coordinate neighborhoods are based on homeomorphisms of open sets, several may be required just to cover all of M. For example, even if we decide to use quaternions for $SO(3)$, several coordinate neighborhoods that map to quaternions may be needed. On the intersections of their domains, a change of coordinates is necessary.

Now we are ready to define a smooth manifold. Changes of coordinates will appear in the manifold definition, and they must satisfy a smoothness condition.

A *smooth structure*[5] on a (topological) manifold M is a family[6] $\mathcal{U} = \{U_\alpha, \phi_\alpha\}$ of coordinate neighborhoods such that:

1. The union of all U_α contains M. Thus, it is possible to obtain coordinates in $\mathbb{R}^n$ for any point in M.

2. For any (U, ϕ) and (V, ψ) in $\mathcal{U}$, if $U \cap V \neq \emptyset$, then the changes of coordinates, $\psi \circ \phi^{-1}$ and $\phi \circ \psi^{-1}$, are smooth functions on $U \cap V$. The changes of coordinates must produce diffeomorphisms on the intersections. In this case, the coordinate neighborhoods are called *compatible*.

3. The family $\mathcal{U}$ is maximal in the sense that if some (U, ϕ) is compatible with every coordinate neighborhood in $\mathcal{U}$, then (U, ϕ) must be included in $\mathcal{U}$.

A well-known theorem (see [133], p. 54) states that if a set of compatible neighborhoods covers all of M, then a unique smooth structure exists that contains them.[7] This means that a differential structure can often be specified by a small number of neighborhoods, and the remaining ones are implied.

A manifold, as defined in Section 4.1.2, together with a smooth structure is called a *smooth manifold*.[8]

Example 8.13 ($\mathbb{R}^n$ as a Smooth Manifold) We should expect that the concepts presented so far apply to $\mathbb{R}^n$, which is the most straightforward family of manifolds. A single coordinate neighborhood $\mathbb{R}^n \rightarrow \mathbb{R}^n$ can be used, which is the identity map. For all integers $n \in \{1, 2, 3\}$ and $n > 4$, this is the only possible smooth structure on $\mathbb{R}^n$. It is truly amazing that for $\mathbb{R}^4$, there are uncountably many incompatible smooth structures, called *exotic* $\mathbb{R}^4$ [291]. There is no need to worry, however; just use the one given by the identity map for $\mathbb{R}^4$. ■

Example 8.14 ($\mathbb{S}^n$ as a Smooth Manifold) One way to define $\mathbb{S}^n$ as a smooth manifold uses $2(n+1)$ coordinate neighborhoods and results in simple expressions. Let $\mathbb{S}^n$ be defined as

$$\mathbb{S}^n = \{(x_1, \ldots, x_{n+1}) \in \mathbb{R}^{n+1} |\; x_1^2 + \cdots + x_{n+1}^2 = 1\}. \tag{8.22}$$

The domain of each coordinate neighborhood is defined as follows. For each i from 1 to $n + 1$, there are two neighborhoods:

$$U_i^+ = \{(x_1, \ldots, x_{n+1}) \in \mathbb{R}^{n+1} |\; x_i > 0\} \tag{8.23}$$

[5]Alternative names are *differentiable structure* and C^∞ *structure*.

[6]In literature in which the coordinate neighborhoods are called *charts*, this family is called an *atlas*.

[7]This is under the assumption that M is Hausdorff and has a countable basis of open sets, which applies to the manifolds considered here.

[8]Alternative names are *differentiable manifold* and C^∞ *manifold*.

and

$$U_i^- = \{(x_1, \ldots, x_{n+1}) \in \mathbb{R}^{n+1} | \; x_i < 0\}. \tag{8.24}$$

Each neighborhood is an open set that covers half of $\mathbb{S}^n$ but misses the great circle at $x_i = 0$. The coordinate functions can be defined by projection down to the $(n-1)$-dimensional hyperplane that contains the great circle. For each i,

$$\phi_i^+(x_1, \ldots, x_{n+1}) = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \tag{8.25}$$

over U_i^+. Each ϕ_i^- is defined the same way, but over U_i^-. Each coordinate function is a homeomorphism from an open subset of $\mathbb{S}^n$ to an open subset of $\mathbb{R}^n$, as required. On the subsets in which the neighborhoods overlap, the changes of coordinate functions are smooth. For example, consider changing from ϕ_i^+ to ϕ_j^- for some $i \neq j$. The change of coordinates is a function $\phi_j^- \circ (\phi_i^+)^{-1}$. The inverse of ϕ_i^+ is expressed as

$$\begin{aligned} &(\phi_i^+)^{-1}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = \\ &(x_1, \ldots, x_{i-1}, 1 - \sqrt{1 - x_1^2 - \cdots - x_{i-1}^2 - x_{i+1}^2 - \cdots - x_n^2}, x_{i+1}, \ldots, x_{n+1}). \end{aligned} \tag{8.26}$$

When composed with ϕ_j^-, the jth coordinate is dropped. This yields

$$\begin{aligned} \phi_k^- \circ (\phi_i^+)^{-1}&(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = \\ &(x_1, \ldots, x_{i-1}, 1 - \sqrt{1 - x_1^2 - \cdots - x_{i-1}^2 - x_{i+1}^2 - \cdots - x_n^2}, \\ &x_{i+1}, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n), \end{aligned} \tag{8.27}$$

which is a smooth function over the domain U_i^+. Try visualizing the changes of coordinates for the circle $\mathbb{S}^1$ and sphere $\mathbb{S}^2$.

The smooth structure can alternatively be defined using only two coordinate neighborhoods by using *stereographic projection*. For $\mathbb{S}^2$, one coordinate function maps almost every point $x \in \mathbb{S}^2$ to $\mathbb{R}^2$ by drawing a ray from the north pole to x and mapping to the point in the $x_3 = 0$ plane that is crossed by the ray. The only excluded point is the north pole itself. A similar mapping can be constructed from the south pole. ■

Example 8.15 ($\mathbb{RP}^n$ as a Smooth Manifold) This example is particularly important because $\mathbb{RP}^3$ is the same manifold as $SO(3)$, as established in Section 4.2.2. Recall from Section 4.1.2 that $\mathbb{RP}^n$ is defined as the set of all lines in $\mathbb{R}^{n+1}$ that pass through the origin. This means that for any $\alpha \in \mathbb{R}$ such that $\alpha \neq 0$, and any $x \in \mathbb{R}^{n+1}$, both x and αx are identified. In projective space, scale does not matter.

A smooth structure can be specified by only $n + 1$ coordinate neighborhoods. For each i from 1 to $n + 1$, let

$$\phi_i(x_1, \ldots, x_{n+1}) = (x_1/x_i, \ldots, x_{i-1}/x_i, x_{i+1}/x_i, \ldots, x_n/x_i), \tag{8.28}$$

over the open set of all points in $\mathbb{R}^{n+1}$ for which $x_i \neq 0$. The inverse coordinate function is given by

$$\phi_i^{-1}(z_1, \ldots, z_n) = (z_1, \ldots, z_{i-1}, 1, z_i, \ldots, z_{n+1}). \tag{8.29}$$

It is not hard to verify that these simple transformations are smooth on overlapping neighborhoods.

A smooth structure over $SO(3)$ can be derived as a special case because $SO(3)$ is topologically equivalent to $\mathbb{RP}^3$. Suppose elements of $SO(3)$ are expressed using unit quaternions. Each (a, b, c, d) is considered as a point on $\mathbb{S}^3$. There are four coordinate neighborhoods. For example, one of them is

$$\phi_b(a, b, c, d) = (a/b,\ c/b,\ d/b), \tag{8.30}$$

which is defined over the subset of $\mathbb{R}^4$ for which $b \neq 0$. The inverse of $\phi_b(a, b, c, d)$ needs to be defined so that a point on $SO(3)$ maps to a point in $\mathbb{R}^4$ that has unit magnitude. ■

Tangent spaces on manifolds Now consider defining tangent spaces on manifolds. Intuitively, the tangent space $T_p(M)$ at a point p on an n-dimensional manifold M is an n-dimensional hyperplane in $\mathbb{R}^m$ that best approximates M around p, when the hyperplane origin is translated to p. This is depicted in Figure 8.8. The notion of a tangent was actually used in Section 7.4.1 to describe local motions for motion planning of closed kinematic chains (see Figure 7.22).

To define a tangent space on a manifold, we first consider a more complicated definition of the tangent space at a point in $\mathbb{R}^n$, in comparison to what was given in Section 8.3.1. Suppose that $M = \mathbb{R}^2$, and consider taking directional derivatives of a smooth function $f : \mathbb{R}^2 \to \mathbb{R}$ at a point $p \in \mathbb{R}^2$. For some (unnormalized) direction vector, $v \in \mathbb{R}^2$, the directional derivative of f at p can be defined as

$$\nabla_v(f)\Big|_p = v_1 \frac{\partial f}{\partial x_1}\Big|_p + v_2 \frac{\partial f}{\partial x_2}\Big|_p. \tag{8.31}$$

The directional derivative used here does not normalize the direction vector (contrary to basic calculus). Hence, $\nabla_v(f) = \nabla(f) \cdot v$, in which "$\cdot$" denotes the inner product or dot product, and $\nabla(f)$ denotes the gradient of f. The set of all possible direction vectors that can be used in this construction forms a two-dimensional vector space that happens to be the tangent space $T_p(\mathbb{R}^2)$, as defined previously. This can be generalized to n dimensions to obtain

$$\nabla_v(f)\Big|_p = \sum_{i=1}^{n} v_i \frac{\partial f}{\partial x_i}\Big|_p, \tag{8.32}$$

for which all possible direction vectors represent the tangent space $T_p(\mathbb{R}^n)$. The set of all directions can be interpreted for our purposes as the set of possible velocity vectors.

Now consider taking (unnormalized) directional derivatives of a smooth function, $f : M \rightarrow \mathbb{R}$, on a manifold. For an n-dimensional manifold, the tangent space $T_p(M)$ at a point $p \in M$ can be considered once again as the set of all unnormalized directions. These directions must intuitively be tangent to the manifold, as depicted in Figure 8.8. There exists a clever way to define them without even referring to specific coordinate neighborhoods. This leads to a definition of $T_p(M)$ that is intrinsic to the manifold.

At this point, you may accept that $T_p(M)$ is an n-dimensional vector space that is affixed to M at p and oriented as shown in Figure 8.8. For the sake of completeness, however, a technical definition of $T_p(M)$ from differential geometry will be given; more details appear in [133, 872]. The construction is based on characterizing the set of all possible directional derivative operators. Let $C^\infty(p)$ denote the set of all smooth functions that have domains that include p. Now make the following identification. Any two functions $f, g \in C^\infty(p)$ are defined to be *equivalent* if there exists an open set $U \subset M$ such that for any $p \in U$, $f(p) = g(p)$. There is no need to distinguish equivalent functions because their derivatives must be the same at p. Let $\tilde{C}^\infty(p)$ denote C^∞ under this identification. A directional derivative operator at p can be considered as a function that maps from $\tilde{C}^\infty(p)$ to $\mathbb{R}$ for some direction. In the case of $\mathbb{R}^n$, the operator appears as ∇_v for each direction v. Think about the set of all directional derivative operators that can be made. Each one must assign a real value to every function in $\tilde{C}^\infty(p)$, and it must obey two axioms from calculus regarding directional derivatives. Let ∇_v denote a directional derivative operator at some $p \in M$ (be careful, however, because here v is not explicitly represented since there are no coordinates). The directional derivative operator must satisfy two axioms:

1. **Linearity:** For any $\alpha, \beta \in \mathbb{R}$ and $f, g \in \tilde{C}^\infty(p)$,
$$\nabla_v(\alpha f + \beta g) = \alpha \nabla_v f + \beta \nabla_v g. \tag{8.33}$$

2. **Leibniz Rule (or Derivation):** For any $f, g \in \tilde{C}^\infty(p)$,
$$\nabla_v(fg) = \nabla_v f \; g(p) + f(p) \nabla_v g. \tag{8.34}$$

You may recall these axioms from standard vector calculus as properties of the directional derivative. It can be shown that the set of all possible operators that satisfy these axioms forms an n-dimensional vector space [133]. This vector space is called the *tangent space*, $T_p(M)$, at p. This completes the definition of the tangent space without referring to coordinates.

It is helpful, however, to have an explicit way to express vectors in $T_p(M)$. A basis for the tangent space can be obtained by using coordinate neighborhoods. An important theorem from differential geometry states that if $F : M \rightarrow N$ is a diffeomorphism onto an open set $U \subset N$, then the tangent space, $T_p(M)$, is isomorphic to $T_{F(p)}(N)$. This means that by using a parameterization (the inverse of a coordinate neighborhood), there is a bijection between velocity vectors in

$T_p(M)$ and velocity vectors in $T_{F(p)}(N)$. Small perturbations in the parameters cause motions in the tangent directions on the manifold N. Imagine, for example, making a small perturbation to three quaternion parameters that are used to represent $SO(3)$. If the perturbation is small enough, motions that are tangent to $SO(3)$ occur. In other words, the perturbed matrices will lie very close to $SO(3)$ (they will not lie in $SO(3)$ because $SO(3)$ is defined by nonlinear constraints on $\mathbb{R}^9$, as discussed in Section 4.1.2).

Example 8.16 (The Tangent Space for $\mathbb{S}^2$) The discussion can be made more concrete by developing the tangent space for $\mathbb{S}^2$, which is embedded in $\mathbb{R}^3$ as the set of all points $(x, y, z) \in \mathbb{R}^3$ for which $x^2 + y^2 + z^2 = 1$. A coordinate neighborhood can be defined that covers most of $\mathbb{S}^2$ by using standard spherical coordinates. Let f denote the coordinate function, which maps from (x, y, z) to angles (θ, ϕ). The domain of f is the open set defined by $\theta \in (0, 2\pi)$ and $\phi \in (0, \pi)$ (this excludes the poles). The standard formulas are $\theta = \operatorname{atan2}(y, x)$ and $\phi = \cos^{-1} z$. The inverse, f^{-1}, yields a parameterization, which is $x = \cos\theta \sin\phi$, $y = \sin\theta \sin\phi$, and $z = \cos\phi$.

Now consider different ways to express the tangent space at some point $p \in \mathbb{S}^2$, other than the poles (a change of coordinates is needed to cover these). Using the coordinates (θ, ϕ), velocities can be defined as vectors in $\mathbb{R}^2$. We can imagine moving in the plane defined by θ and ϕ, provided that the limits $\theta \in (0, 2\pi)$ and $\phi \in (0, \pi)$ are respected.

We can also use the parameterization to derive basis vectors for the tangent space as vectors in $\mathbb{R}^3$. Since the tangent space has only two dimensions, we must obtain a plane that is "tangent" to the sphere at p. These can be found by taking derivatives. Let f^{-1} be denoted as $x(\theta, \phi)$, $y(\theta, \phi)$, and $z(\theta, \phi)$. Two basis vectors for the tangent plane at p are

$$\begin{bmatrix} \dfrac{dx(\theta,\phi)}{d\theta} & \dfrac{dy(\theta,\phi)}{d\theta} & \dfrac{dz(\theta,\phi)}{d\theta} \end{bmatrix} \tag{8.35}$$

and

$$\begin{bmatrix} \dfrac{dx(\theta,\phi)}{d\phi} & \dfrac{dy(\theta,\phi)}{d\phi} & \dfrac{dz(\theta,\phi)}{d\phi} \end{bmatrix}. \tag{8.36}$$

Computing these derivatives and normalizing yields the vectors $[-\sin\theta \;\; \cos\theta \;\; 0]$ and $[\cos\theta\cos\phi \;\; \sin\theta\cos\phi \;\; -\sin\phi]$. These can be imagined as the result of making small perturbations of θ and ϕ at p. The vector space obtained by taking all linear combinations of these vectors is the tangent space at $\mathbb{R}^2$. Note that the direction of the basis vectors depends on $p \in \mathbb{S}^2$, as expected.

The tangent vectors can now be imagined as lying in a plane that is tangent to the surface, as shown in Figure 8.8. The normal vector to a surface specified as $g(x, y, z) = 0$ is ∇g, which yields $[x \;\; y \;\; z]$ after normalizing. This could alternatively be obtained by taking the cross product of the two vectors above and using the parameterization f^{-1} to express it in terms of x, y, and z. For a point

$p = (x_0, y_0, z_0)$, the plane equation is

$$x_0(x - x_0) + y_0(y - y_0) + z_0(z - z_0) = 0. \tag{8.37}$$

■

Vector fields and velocity fields on manifolds The notation for a tangent space on a manifold looks the same as for $\mathbb{R}^n$. This enables the vector field definition and notation to extend naturally from $\mathbb{R}^n$ to smooth manifolds. A *vector field* on a manifold M assigns a vector in $T_p(M)$ for every $p \in M$. It can once again be imagined as a needle diagram, but now the needle diagram is spread over the manifold, rather than lying in $\mathbb{R}^n$.

The velocity field interpretation of a vector field can also be extended to smooth manifolds. This means that $\dot{x} = f(x)$ now defines a set of n differential equations over M and is usually expressed using a coordinate neighborhood of the smooth structure. If f is a smooth vector field, then a *solution trajectory*, $\tau : [0, \infty) \rightarrow M$, can be defined from any $x_0 \in M$. Solution trajectories in the sense of Filipov can also be defined, for the case of piecewise-smooth vector fields.

8.4 Complete Methods for Continuous Spaces

A complete feedback planning algorithm must compute a feedback solution if one exists; otherwise, it must report failure. Section 8.4.1 parallels Section 8.2 by defining feedback plans and navigation functions for the case of a continuous state space. Section 8.4.2 indicates how to define a feasible feedback plan from a cell complex that was computed using cell decomposition techniques. Section 8.4.3 presents a combinatorial approach to computing an optimal navigation function and corresponding feedback plan in $\mathbb{R}^2$. Sections 8.4.2 and 8.4.3 allow the feedback plan to be a discontinuous vector field. In many applications, especially those in which dynamics dominate, some conditions need to be enforced on the navigation functions and their resulting vector fields. Section 8.4.4 therefore considers constraints on the allowable vector fields and navigation functions. This coverage includes navigation functions in the sense of Rimon-Koditschek [829], from which the term navigation function was introduced.

8.4.1 Feedback Motion Planning Definitions

Using the concepts from Section 8.3, we are now ready to define feedback motion planning over configuration spaces or other continuous state spaces. Recall Formulation 4.1, which defined the basic motion planning problem in terms of configuration space. The differences in the current setting are that there is no initial condition, and the requirement of a solution path is replaced by a solution vector field. The formulation here can be considered as a continuous-time adaptation to Formulation 8.1.

Formulation 8.2 (Feedback Motion Planning)

1. A *state space*, X, which is a smooth manifold. The state space will most often be $\mathcal{C}_{free}$, as defined in Section 4.3.1.[9]

2. For each state, $x \in X$, an *action space*, $U(x) = T_x(X)$. The zero velocity, $0 \in T_x(X)$, is designated as the termination action, u_T. Using this model, the robot is capable of selecting its velocity at any state.[10]

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

4. A *state transition (differential) equation*,
$$\dot{x} = u, \tag{8.38}$$
which is expressed using a coordinate neighborhood and yields the velocity, $\dot{x}$, directly assigned by the action u. The velocity produced by u_T is $0 \in T_x(X)$ (which means "stop").

5. A *goal set*, $X_G \subset X$.

A *feedback plan*, π, for Formulation 8.2 is defined as a function π, which produces an action $u \in U(x)$ for each $x \in X$. A feedback plan can equivalently be considered as a vector field on X because each $u \in U(x)$ specifies a velocity vector (u_T specifies zero velocity). Since the initial state is not fixed, it becomes slightly more complicated to define what it means for a plan to be a solution to the problem. Let $X_r \subset X$ denote the set of all states from which X_G is *reachable.* More precisely, a state x_I belongs to X_r if and only if a continuous path $\tau : [0,1] \to X$ exists for which $\tau(0) = x_I$ and $\tau(1) = x_G$ for some $x_G \in X_G$. This means that a solution path exists from x_I for the "open-loop" motion planning problem, which was considered in Chapter 4.

8.4.1.1 Solution concepts

A feedback plan, π, is called a *solution* to the problem in Formulation 8.2 if from all $x_I \in X_r$, the integral curves of π (considered as a vector field) arrive in X_G, at which point the termination action is applied. Some words of caution must be given about what it means to "arrive" in X_G. Notions of stability from control theory [523, 846] are useful for distinguishing different cases; see Section 15.1. If

[9]Note that X already excludes the obstacle region. For some problems in Part IV, the state space will be $X = \mathcal{C}$, which includes the obstacle region.

[10]This allows discontinuous changes in velocity, which is unrealistic in many applications. Additional constraints, such as imposing acceleration bounds, will also be discussed. For a complete treatment of differential constraints, see Part IV.

X_G is a small ball centered on x_G, then the ball will be reached after finite time using the inward vector field shown in Figure 8.5b. Now suppose that X_G is a single point, x_G. The inward vector field produces velocities that bring the state closer and closer to the origin, but when is it actually reached? It turns out that convergence to the origin in this case is only *asymptotic*; the origin is reached in the limit as the time approaches infinity. Such stability often arises in control theory from smooth vector fields. We may allow such asymptotic convergence to the goal (if the vector field is smooth and the goal is a point, then this is unavoidable). If any integral curves result in only asymptotic convergence to the goal, then a solution plan is called an *asymptotic solution plan*. Note that in general it may be impossible to require that π is a smooth (or even continuous) nonzero vector field. For example, due to the *hairy ball theorem* [834], it is known that no such vector field exists for $\mathbb{S}^n$ for any even integer n. Therefore, the strongest possible requirement is that π is smooth except on a set of measure zero; see Section 8.4.4. We may also allow solutions π for which *almost all* integral curves arrive in X_G.

However, it will be assumed by default in this chapter that a solution plan converges to x_G in finite time. For example, if the inward field is normalized to produce unit speed everywhere except the origin, then the origin will be reached in finite time. A constraint can be placed on the set of allowable vector fields without affecting the existence of a solution plan. As in the basic motion planning problem, the speed along the path is not important. Let a *normalized vector field* be any vector field for which either $\|f(x)\| = 1$ or $f(x) = 0$, for all $x \in X$. This means that all velocity vectors are either unit vectors or the zero vector, and the speed is no longer a factor. A normalized vector field provides either a direction of motion or no motion. Note that any vector field f can be converted into a normalized vector field by dividing the velocity vector $f(x)$ by its magnitude (unless the magnitude is zero), for each $x \in X$.

In many cases, unit speed does not necessarily imply a constant speed in some true physical sense. For example, if the robot is a floating rigid body, there are many ways to parameterize position and orientation. The speed of the body is sensitive to this parameterization. Therefore, other constraints may be preferable instead of $\|f(x)\| = 1$; however, it is important to keep in mind that the constraint is imposed so that $f(x)$ provides a *direction* at x. The particular magnitude is assumed unimportant.

So far, consideration has been given only to a *feasible feedback motion planning problem*. An *optimal feedback motion planning problem* can be defined by introducing a cost functional. Let $\tilde{x}_t$ denote the function $\tilde{x}_t : [0, t] \rightarrow X$, which is called the *state trajectory* (or *state history*). This is a continuous-time version of the state history, which was defined previously for problems that have discrete stages. Similarly, let $\tilde{u}_t$ denote the *action trajectory* (or *action history*), $\tilde{u}_t : [0, t] \rightarrow U$. Let L denote a cost functional, which may be applied from any x_I to yield

$$L(\tilde{x}_{t_F}, \tilde{u}_{t_F}) = \int_0^{t_F} l(x(t), u(t))dt + l_F(x(t_F)), \tag{8.39}$$

in which t_F is the time at which the termination action is applied. The term

$l(x(t), u(t))$ can alternatively be expressed as $l(x(t), \dot{x}(t))$ by using the state transition equation (8.38). A normalized vector field that optimizes (8.39) from all initial states that can reach the goal is considered as an *optimal feedback motion plan*.

Note that the state trajectory can be determined from an action history and initial state. In fact, we could have used action trajectories to define a solution path to the motion planning problem of Chapter 4. Instead, a solution was defined there as a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ to avoid having to introduce velocity fields on smooth manifolds. That was the only place in the book in which the action space seemed to disappear, and now you can see that it was only hiding to avoid inessential notation.

8.4.1.2 Navigation functions

As in Section 8.2.2, potential functions can be used to represent feedback plans, assuming that a local operator is developed that works for continuous state spaces. In the discrete case, the local operator selects an action that reduces the potential value. In the continuous case, the local operator must convert the potential function into a vector field. In other words, a velocity vector must be defined at each state. By default, it will be assumed here that the vector fields derived from the navigation function are not necessarily normalized.

Assume that $\pi(x) = u_T$ is defined for all $x \in X_G$, regardless of the potential function. Suppose that a potential function $\phi : X \rightarrow \mathbb{R}$ has been defined for which the gradient

$$\nabla \phi = \begin{bmatrix} \frac{\partial \phi}{\partial x_1} & \frac{\partial \phi}{\partial x_2} & \cdots & \frac{\partial \phi}{\partial x_n} \end{bmatrix} \tag{8.40}$$

exists over all of $X \setminus X_G$. The corresponding feedback plan can then be defined as $\pi(x) = -\nabla \phi|_x$. This defines the local operator, which means that the velocity is taken in the direction of the steepest descent of ϕ. The idea of using potential functions in this way was proposed for robotics by Khatib [525, 526] and can be considered as a form of *gradient descent*, which is a general optimization technique.

It is also possible to work with potential functions for which the gradient does not exist everywhere. In these cases, a continuous-space version of (8.4) can be defined for a small, fixed Δt as

$$u^* = \underset{u \in U(x)}{\operatorname{argmin}} \Big\{ \phi(x') \Big\}, \tag{8.41}$$

in which x' is the state obtained by integrating velocity u from x for time Δt. One problem is that Δt should be chosen to use the smallest possible neighborhood around ϕ. It is best to allow only potential functions for which Δt can be made arbitrarily small at every x without affecting the decision in (8.41). To be precise, this means that an infinite sequence of u^* values can be determined from a sequence of Δt values that converges to 0. A potential function should then be chosen to ensure after some point in the sequence, u^*, exists and the same u^* can be chosen to satisfy (8.41) as Δt approaches 0. A special case of this is if the gradient of ϕ exists; the infinite sequence in this case converges to the negative gradient.

A potential function, ϕ, is called a *navigation function* if the vector field that is derived from it is a solution plan. The optimal cost-to-go serves as an *optimal navigation function*. If multiple vector fields can be derived from the same ϕ, then every possible derived vector field must yield a solution feedback plan. If designed appropriately, the potential function can be viewed as a kind of "ski slope" that guides the state to X_G. If there are extra local minima that cause the state to become trapped, then X_G will not be reached. To be a navigation function, such local minima outside of X_G are not allowed. Furthermore, there may be additional requirements to ensure that the derived vector field satisfies additional constraints, such as bounded acceleration.

Example 8.17 (Quadratic Potential Function) As a simple example, suppose $X = \mathbb{R}^2$, there are no obstacles, and $q_{goal} = (0, 0)$. A quadratic function $\phi(x, y) = \frac{1}{2}x_1^2 + \frac{1}{2}x_2^2$ serves as a good potential function to guide the state to the goal. The feedback motion strategy is defined as $f = -\nabla\phi = [-x_1 \quad -x_2]$, which is the inward vector field shown in Figure 8.5b.

If the goal is instead at some $(x_1', x_2') \in \mathbb{R}^2$, then a potential function that guides the state to the goal is $\phi(x_1, x_2) = (x_1 - x_1')^2 + (x_2 - x_2')^2$. ■

Suppose the state space represents a configuration space that contains point obstacles. The previous function ϕ can be considered as an attractive potential because the configuration is attracted to the goal. One can also construct a repulsive potential that repels the configuration from the obstacles to avoid collision. Let ϕ_a denote the attractive component and ϕ_r denote a repulsive potential that is summed over all obstacle points. A potential function of the form $\phi = \phi_a + \phi_r$ can be defined to combine both effects. The robot should be guided to the goal while avoiding obstacles. The problem is that it is difficult in general to ensure that the potential function will not contain multiple local minima. The configuration could become trapped at a local minimum that is not in the goal region. This was an issue with the planner from Section 5.4.3.

8.4.2 Vector Fields Over Cell Complexes

This section describes how to construct a piecewise-smooth vector field over a cell complex. Only normalized vector fields will be considered. It is assumed that each cell in the complex has a simple shape over which it is easy to define a patch of the vector field. In many cases, the cell decomposition techniques that were introduced in Chapter 6 for motion planning can be applied to construct a feedback plan.

Suppose that an n-dimensional state space X has been decomposed into a cell complex, such as a simplicial complex or singular complex, as defined in Section 6.3.1. Assume that the goal set is a single point, x_G. Defining a feedback plan π over X requires placing a vector field on X for which all integral curves lead to

x_G (if x_G is reachable). This is accomplished by defining a smooth vector field for each n-cell. Each $(n-1)$-cell is a switching boundary, as considered in Section 8.3.1. This leads directly to solution trajectories in the sense of Filipov. If π is allowed to be discontinuous, then it is actually not important to specify values on any of the cells of dimension $n-1$ or less.

A hierarchical approach is taken to the construction of π:

1. Define a discrete planning problem over the n-cells. The cell that contains x_G is designated as the goal, and a discrete navigation function is defined over the cells.

2. Define a vector field over each n-cell. The field should cause all states in the cell to flow into the next cell as prescribed by the discrete navigation function.

One additional consideration that is important in applications is to try to reduce the effect of the discontinuity across the boundary as much as possible. It may be possible to eliminate the discontinuity, or even construct a smooth transition between n-cells. This issue will not be considered here, but it is nevertheless quite important [235, 643].

The approach will now be formalized. Suppose that a cell complex has been defined over a continuous state space, X. Let $\check{X}$ denote the set of n-cells, which can be interpreted as a finite state space. A discrete planning problem will be defined over $\check{X}$. To avoid confusion with the original continuous problem, the prefix *super* will be applied to the discrete planning components. Each superstate $\check{x} \in \check{X}$ corresponds to an n-cell. From each $\check{x}$, a superaction, $\check{u} \in \check{U}(\check{x})$ exists for each neighboring n-cell (to be neighboring, the two cells must share an $(n-1)$-dimensional boundary). Let the goal superstate $\check{x}_g$ be the n-cell that contains x_G. Assume that the cost functional is defined for the discrete problem so that every action (other than u_T) produces a unit cost. Now the concepts from Section 8.2 can be applied to the discrete problem. A discrete navigation function, $\check{\phi} : \check{X} \rightarrow \mathbb{R}$, can be computed using Dijkstra's algorithm (or another algorithm, particularly if optimality is not important). Using the discrete local operator from Section 8.2.2, this results in a discrete feedback plan, $\check{\pi} : \check{X} \rightarrow \check{U}$.

Based on the discrete feedback plan, there are two kinds of n-cells. The first is the goal cell, $\check{x}_g$, for which a vector field needs to be defined so that all integral curves lead to X_g in finite time.[11] A termination action can be applied when x_G is actually reached. The remaining n-cells are of the second kind. For each cell $\check{x}$, the boundary that is shared with the cell reached by applying $\check{u} = \check{\pi}(\check{x})$ is called the *exit face*. The vector field over the n-cell $\check{x}$ must be defined so that all integral curves lead to the exit face. When the exit face is reached, a transition will occur into the next n-cell. If the n-cells are convex, then defining this transition is straightforward (unless there are additional requirements on the field, such as

[11]This is possible in finite time, even if X_g is a single point, because the vector field is not continuous. Otherwise, only asymptotic convergence may be possible.

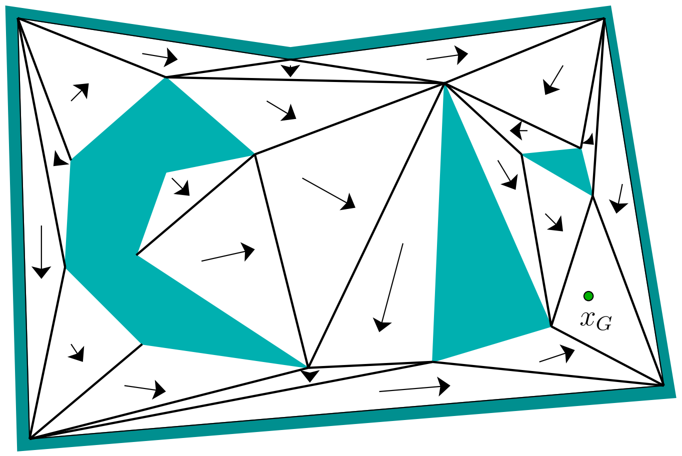


Figure 8.10: A triangulation is used to define a vector field over X. All solution trajectories lead to the goal.

smoothness at the boundary). For more complicated cells, one possibility is to define a vector field that retracts all points onto a single curve in the cell.

A simple example of the approach is illustrated for the case of $X = \mathcal{C}_{free} \subset \mathbb{R}^2$, in which the boundary of $\mathcal{C}_{free}$ is polygonal. This motion planning problem was considered in Section 6.2, but without feedback. Suppose that a triangulation of X has been computed, as described in Section 6.3.2. An example was shown in Figure 6.16. A discrete feedback plan is shown for a particular goal state in Figure 8.10. Each 2-cell (triangle) is labeled with an arrow that points to the next cell.

For the cell that contains x_G, a normalized version of the inward vector field shown in Figure 8.5b can be formed by dividing each nonzero vector by its magnitude. It can then be translated to move its origin to x_G. For each remaining 2-cell, a vector field must be constructed that flows into the appropriate neighboring cell. Figure 8.11 illustrates a simple way to achieve this. An outward vector field can be made by negating the field shown in Figure 8.5b to obtain $f = [x \;\; y]$. This field can be normalized and translated to move the origin to the triangle vertex that is not incident to the exit edge. This is called the *repulsive vertex* in Figure 8.11. This generates a vector field that pushes all points in the triangle to the ext edge. If the fields are constructed in this way for each triangle, then the global vector field represents a solution feedback plan for the problem. Integral curves (in the sense of Filipov) lead to x_G in finite time.

8.4.3 Optimal Navigation Functions

The vector fields developed in the last section yield feasible trajectories, but not necessarily optimal trajectories unless the initial and goal states are in the same convex n-cell. If $X = \mathbb{R}^2$, then it is possible to make a continuous version of Dijkstra's algorithm [708]. This results in an exact cost-to-go function over X based on the Euclidean shortest path to a goal, x_G. The cost-to-go function serves

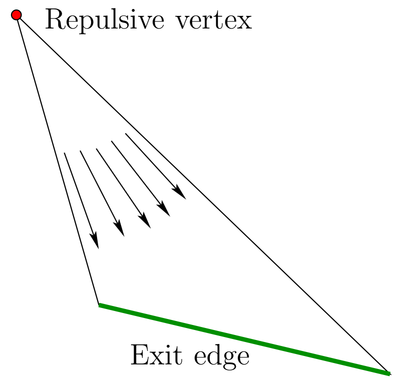


Figure 8.11: A vector field can be defined for each triangle by repelling from a vertex that opposes the exit edge.

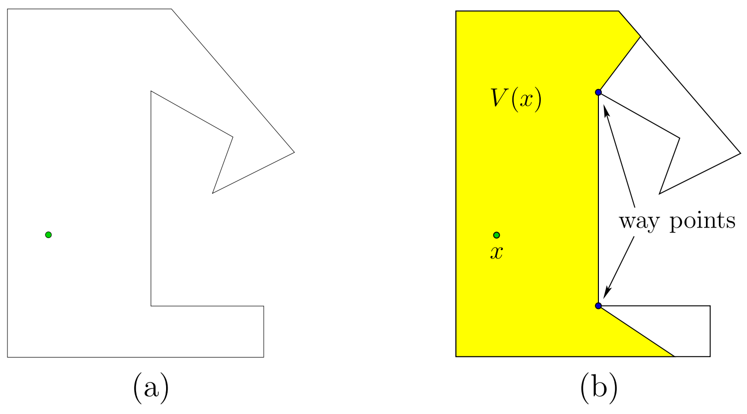


Figure 8.12: (a) A point, x, in a simple polygon. (b) The visibility polygon, $V(x)$.

as the navigation function, from which the feedback plan is defined by using a local steepest descent.

Suppose that X is bounded by a simple polygon (no holes). Assume that only normalized vector fields are allowed. The cost functional is assumed to be the Euclidean distance traveled along a state trajectory. Recall from Section 6.2.4 that for optimal path planning, $X = \mathrm{cl}(\mathcal{C}_{free})$ must be used. Assume that $\mathcal{C}_{free}$ and $\mathrm{cl}(\mathcal{C}_{free})$ have the same connectivity.[12] This technically interferes with the definition of tangent spaces from Section 8.3 because each point of X must be contained in an open neighborhood. Nevertheless, we allow vectors along the boundary, provided that they "point" in a direction tangent to the boundary. This can be formally defined by considering boundary regions as separate manifolds.

Consider computing the optimal cost-to-go to a point x_G for a problem such as that shown in Figure 8.12a. For any $x \in X$, let the *visibility polygon* $V(x)$ refer

[12]This precludes a choice of $\mathcal{C}_{free}$ for which adding the boundary point enables a homotopically distinct path to be made through the boundary point. An example of this is when two square obstacles in $\mathbb{R}^2$ contact each other only at a pair of corners.

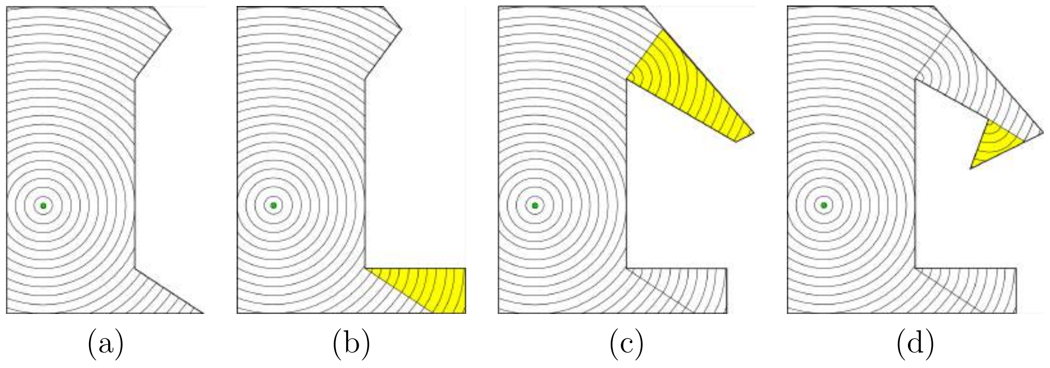


Figure 8.13: The optimal navigation function is computed in four iterations. In each iteration, the navigation function is extended from a new way point.

to the set of all points visible from x, which is illustrated in Figure 8.12b. A point x' lies in $V(x)$ if and only if the line segment from x' to x is contained in X. This implies that the cost-to-go from x' to x is just the Euclidean distance from x' to x. The optimal navigation function can therefore be immediately defined over $V(x_G)$ as

$$\phi(x) = \|x - x_G\|. \tag{8.42}$$

Level sets at regularly spaced values of this navigation function are shown in Figure 8.13a.

How do we compute the optimal cost-to-go values for the points in $X \setminus V(x_G)$? For the segments on the boundary of $V(x)$ for any $x \in X$, some edges are contained in the boundary of X, and others cross the interior of X. For the example in Figure 8.12b, there are two edges that cross the interior. For each segment that crosses the interior, let the closer of the two vertices to x be referred to as a *way point*. Two way points are indicated in Figure 8.12b. The way points of $V(x_G)$ are places through which some optimal paths must cross. Let $W(x)$ for any $x \in X$ denote the set of way points of $V(x)$.

A straightforward algorithm proceeds as follows. Let Z_i denote the set of points over which ϕ has been defined, in the ith iteration of the algorithm. In the first iteration, $Z_1 = V(x_G)$, which is the case shown in Figure 8.13a. The way points of $V(x_G)$ are placed in a queue, Q. In each following iteration, a way point x is removed from Q. Let Z_i denote the domain over which ϕ is defined so far. The domain of ϕ is extended to include all new points visible from x. These new points are $V(x) \setminus Z_i$. This yields $Z_{i+1} = Z_i \cup V(x)$, the extended domain of ϕ. The values of $\phi(x')$ for $x' \in Z_{i+1} \setminus Z_i$ are defined by

$$\phi(x') = \phi(x) + \|x' - x\|, \tag{8.43}$$

in which x is the way point that was removed from Q (the optimal cost-to-go value of x was already computed). The way points of $V(x)$ that do not lie in Z_{i+1} are added to Q. Each of these will yield new portions of X that have not yet been seen. The algorithm terminates when Q is empty, which implies that $Z_k = X$ for some k. The execution of the algorithm is illustrated in Figure 8.13.

The visibility polygon can be computed in time $O(n \lg n)$ if X is described by n edges. This is accomplished by performing a *radial sweep*, which is an adaptation of the method applied in Section 6.2.2 for vertical cell decomposition. The difference for computing $V(x)$ is that a ray anchored at x is swept radially (like a radar sweep). The segments that intersect the ray are sorted by their distance from x. For the algorithm that constructs the navigation function, no more than $O(n)$ visibility polygons are computed because each one is computed from a unique way point. This implies $O(n^2 \lg n)$ running time for the whole algorithm. Unfortunately, there is no extension to higher dimensions; recall from Section 7.7.1 that computing shortest paths in a 3D environment is NP-hard [172].

The algorithm given here is easy to describe, but it is not the most general, nor the most efficient. If X has holes, then the level set curves can collide by arriving from different directions when traveling around an obstacle. The queue, Q, described above can be sorted as in Dijkstra's algorithm, and special data structures are needed to identify when critical events occur as the cost-to-go is propagated outward. It was shown in [443] that this can be done in time $O(n \lg n)$ and space $O(n \lg n)$.

8.4.4 A Step Toward Considering Dynamics

If dynamics is an important factor, then the discontinuous vector fields considered so far are undesirable. Due to momentum, a mechanical system cannot instantaneously change its velocity (see Section 13.3). In this context, vector fields should be required to satisfy additional constraints, such as smoothness or bounded acceleration. This represents only a step toward considering dynamics. Full consideration is given in Part IV, in which precise equations of motions of dynamical systems are expressed as part of the model. The approach in this section is to make vector fields that are "dynamics-ready" rather than carefully considering particular equations of motion.

A framework has been developed by defining a navigation function that satisfies some desired constraints over a simple region, such as a disc [829]. A set of transformations is then designed that are proved to preserve the constraints while adapting the navigation function to more complicated environments. For a given problem, a complete algorithm for constructing navigation functions is obtained by applying the appropriate series of transformations from some starting shape.

This section mostly focuses on constraints that are maintained under this transformation-based framework. Sections 8.4.2 and 8.4.3 worked with normalized vector fields. Under this constraint, virtually any vector field could be defined, provided that the resulting algorithm constructs fields for which integral curves exist in the sense of Filipov. In this section, we remove the constraint that vector fields must be normalized, and then consider other constraints. The velocity given by the vector field is now assumed to represent the true speed that must be executed when the vector field is applied as a feedback plan.

One implication of adding constraints to the vector field is that optimal solutions may not satisfy them. For example, the optimal navigation functions of Section 8.4.3 lead to discontinuous vector fields, which violate the constraints to be considered in this section. The required constraints restrict the set of allowable vector fields. Optimality must therefore be defined over the restricted set of vector fields. In some cases, an optimal solution may not even exist (see the discussion of open sets and optimality in Section 9.1.1). Therefore, this section focuses only on feasible solutions.

8.4.4.1 An acceleration-based control model

To motivate the introduction of constraints, consider a control model proposed in [235, 830]. The action space, defined as $U(x) = T_x(X)$ in Formulation 8.2, produces a velocity for each action $u \in U(x)$. Therefore, $\dot{x} = u$. Suppose instead that each action produces an acceleration. This can be expressed as $\ddot{x} = u$, in which $\ddot{x}$ is an *acceleration vector*,

$$\ddot{x} = \frac{d\dot{x}}{dt} = \left[\frac{d^2 x_1}{dt^2} \;\; \frac{d^2 x_2}{dt^2} \;\; \cdots \;\; \frac{d^2 x_n}{dt^2} \right]. \tag{8.44}$$

The velocity $\dot{x}$ is obtained by integration over time. The state trajectory, $\tilde{x} : T \rightarrow X$, is obtained by integrating (8.44) twice.

Suppose that a vector field is given in the form $\dot{x} = f(x)$. How can a feedback plan be derived? Consider how the velocity vectors specified by $f(x)$ change as x varies. Assume that $f(x)$ is smooth (or at least C^1), and let

$$\nabla_{\dot{x}} f(x) = [\nabla_{\dot{x}} f_1(x) \;\; \nabla_{\dot{x}} f_2(x) \;\; \cdots \;\; \nabla_{\dot{x}} f_n(x)], \tag{8.45}$$

in which $\nabla_{\dot{x}}$ denotes the unnormalized directional derivative in the direction of $\dot{x}$: $\nabla f_i \cdot \dot{x}$. Suppose that an initial state x_I is given, and that the initial velocity is $\dot{x} = f(x_I)$. The feedback plan can now be defined as

$$u = \nabla_{\dot{x}} f(x). \tag{8.46}$$

This is equivalent to the previous definition of a feedback plan from Section 8.4.1; the only difference is that now two integrations are needed (which requires both x and $\dot{x} = f(x_I)$ as initial conditions) and a differentiability condition must be satisfied for the vector field.

Now the relationship between $\dot{x}$ and $f(x)$ will be redefined. Suppose that $\dot{x}$ is the true measured velocity during execution and that $f(x)$ is the prescribed velocity, obtained from the vector field f. During execution, it is assumed that $\dot{x}$ and $f(x)$ are not necessarily the same, but the task is to keep them as close to each other as possible. A discrepancy between them may occur due to dynamics that have not been modeled. For example, if the field $f(x)$ requests that the velocity must suddenly change, a mobile robot may not be able to make a sharp turn due to its momentum.

Using the new interpretation, the difference, $f(x) - \dot{x}$, can be considered as a discrepancy or error. Suppose that a vector field f has been computed. A feedback plan becomes the *acceleration-based control* model

$$u = K(f(x) - \dot{x}) + \nabla_{\dot{x}} f(x), \tag{8.47}$$

in which K is a scalar *gain constant*. A larger value of K will make the control system more aggressively attempt to reduce the error. If K is too large, then acceleration or energy constraints may be violated. Note that if $\dot{x} = f(x)$, then $u = \nabla_{\dot{x}} f(x)$, which becomes equivalent to the earlier formulation.

8.4.4.2 Velocity and acceleration constraints

Considering the acceleration-based control model, some constraints can be placed on the set of allowable vector fields. A *bounded-velocity model* means that $\|\dot{x}\| < v_{max}$, for some positive real value v_{max} called the *maximum speed*. This could indicate, for example, that the robot has a maximum speed for safety reasons. It is also possible to bound individual components of the velocity vector. For example, there may be separate bounds for the maximum angular and linear velocities of an aircraft. Intuitively, velocity bounds imply that the functions f_i, which define the vector field, cannot take on large values.

A *bounded-acceleration model* means that $\|\ddot{x}\| \leq a_{max}$, in which a_{max} is a positive real value called the *maximum acceleration*. Intuitively, acceleration bounds imply that the velocity cannot change too quickly while traveling along an integral curve. Using the control model $\ddot{x} = u$, this implies that $\|u\| \leq a_{max}$. It also imposes the constraint that vector fields must satisfy $\|\nabla_{\dot{x}} f(x)\| \leq a_{max}$ for all $\dot{x}$ and $x \in X$. The condition $\|u\| \leq a_{max}$ is very important in practice because higher accelerations are generally more expensive (bigger motors are required, more fuel is consumed, etc.). The action u may correspond directly to the torques that are applied to motors. In this case, each motor usually has an upper limit.

As has already been seen, setting an upper bound on velocity generally does not affect the existence of a solution. Imagine that a robot can always decide to travel more slowly. If there is also an upper bound on acceleration, then the robot can attempt to travel more slowly to satisfy the bound. Imagine slowing down in a car to make a sharp turn. If you would like to go faster, then it may be more difficult to satisfy acceleration constraints. Nevertheless, in most situations, it is preferable to go faster.

A discontinuous vector field fails to satisfy any acceleration bound because it essentially requires infinite acceleration at the discontinuity to cause a discontinuous jump in the velocity vector. If the vector field satisfies the Lipschitz condition (8.16) for some constant C, then it satisfies the acceleration bound if $C < a_{max}$.

In Chapter 13, we will precisely specify $U(x)$ at every $x \in X$, which is more general than imposing simple velocity and acceleration bounds. This enables virtually any physical system to be modeled.

8.4.4.3 Navigation function in the sense of Rimon-Koditschek

Now consider constructing a navigation function from which a vector field can be derived that satisfies constraints motivated by the acceleration-based control model, (8.47). As usual, the definition of a navigation function begins with the consideration of a potential function, $\phi : X \rightarrow \mathbb{R}$. What properties does a potential function need to have so that it may be considered as a navigation function as defined in Section 8.4.1 and also yield a vector field that satisfies an acceleration bound? Sufficient conditions will be given that imply that a potential function will be a navigation function that satisfies the bound.

To give the conditions, it will first be important to characterize extrema of multivariate functions. Recall from basic calculus that a function $f : \mathbb{R} \rightarrow \mathbb{R}$ has a critical point when the first derivative is zero. At such points, the sign of the second derivative indicates whether the critical point is a minimum or maximum. These ideas can be generalized to higher dimensions. A *critical point* of ϕ is one for which $\nabla\phi = 0$. The *Hessian* of ϕ is defined as the matrix

$$H(\phi) = \begin{pmatrix} \dfrac{\partial^2\phi}{\partial^2 x_1^2} & \dfrac{\partial^2\phi}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2\phi}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2\phi}{\partial x_2 \partial x_1} & \dfrac{\partial^2\phi}{\partial x_2^2} & \cdots & \dfrac{\partial^2\phi}{\partial x_2 \partial x_n} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial^2\phi}{\partial x_n \partial x_1} & \dfrac{\partial^2\phi}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2\phi}{\partial x_n^2} \end{pmatrix}. \qquad (8.48)$$

At each critical point, the Hessian gives some information about the extremum. If the rank of $H(\phi)$ at x is n, then the Hessian indicates the kind of extremum. If (8.48) is positive definite,[13] then the ϕ achieves a *local minimum* at x. If (8.48) is negative definite,[14] then the ϕ achieves a *local maximum* at x. In all other cases, x is a *saddle point*. If the rank of $H(\phi)$ at x is less than n, then the Hessian is *degenerate*. In this case the Hessian cannot classify the type of extremum. An example of this occurs when x lies in a plateau (there is no direction in which ϕ increases or decreases. Such behavior is obviously bad for a potential function because the local operator would not be able to select a direction.

Suppose that the navigation function is required to be smooth, to ensure the existence of a gradient at every point. This enables gradient descent to be performed. If X is not contractible, then it turns out there must exist some critical points other than x_G at which $\nabla\phi(x) = 0$. The critical points can even be used

[13]Positive definite for an $n \times n$ matrix A means that for all $x \in \mathbb{R}^n$, $x^T A x > 0$. If A is symmetric (which applies to $H(\phi)$), then this is equivalent to A having all positive eigenvalues.

[14]Negative definite means that for all $x \in \mathbb{R}^n$, $x^T A x < 0$. If A is symmetric, then this is equivalent to A having all negative eigenvalues.

to infer the topology of X, which is the basic idea in the subject of *Morse theory* [701, 234]. Unfortunately, this implies that there does not exist a solution navigation function for such spaces because the definition in Section 8.4.1 required that the integral curve from any state that can reach x_G must reach it using the vector field derived from the navigation function. If the initial state is a critical point, the integral curve is constant (the state remains at the critical point). Therefore, under the smoothness constraint, the definition of a navigation function should be modified to allow critical points at a small number of places (only on a set that has measure zero). It is furthermore required that the set of states from which the integral curves arrive at each critical point (i.e., the domain of attraction of each critical point) has measure zero. From all possible initial states, except from a set of measure zero, the integral curves must reach x_G, if it is reachable. This is ensured in the following definition.

A function $\phi : X \rightarrow \mathbb{R}$ is called a *navigation function in the sense of Rimon-Koditschek* if [829]:

1. It is smooth (or at least C^2).
2. Among all values on the connected component of $\mathcal{C}_{free}$ that contains x_G, there is only one local minimum, which is at x_G.[15]
3. It is maximal and constant on $\partial\mathcal{C}_{free}$, the boundary of $\mathcal{C}_{free}$.
4. It is a Morse function [701], which means that at each critical point x (i.e., $\nabla\phi|_x = 0$), the Hessian of ϕ is not degenerate.[16] Such functions are known to exist on any smooth manifold.

If ϕ is smooth in the C^∞ sense, then by Sard's Theorem [234] the set of critical points has measure zero.

Methods for constructing navigation functions are outlined in [829] for a general family of problems in which $\mathcal{C}_{free}$ has a semi-algebraic description. The basic idea is to start with simple shapes over which a navigation function can be easily defined. One example of this is a spherical subset of $\mathbb{R}^n$, which contains spherical obstacles. A set of distorting transformations is then developed to adapt the navigation functions to other shapes while ensuring that the four properties above are maintained. One such transformation extends a ball into any visibility region (in the sense defined in Section 8.4.3). This is achieved by smoothly stretching out the ball into the shape of the visibility region. (Such regions are sometimes called *star-shaped*.) The transformations given in [829] can be combined to define navigation functions for a large family of configuration spaces. The main problem is that the configuration space obstacles and the connectivity of $\mathcal{C}_{free}$ are represented only implicitly, which makes it difficult to correctly apply the method to

[15]Some authors do not include the global minimum as a local minimum. In this case, one would say that there are no local minima.

[16]Technically, to be Morse, the values of the function must also be distinct at each critical point.

complicated high-dimensional problems. One of the advantages of the approach is that proving convergence to the goal is simplified. In many cases, Lyapunov stability analysis can be performed (see Section 15.1.1).

8.4.4.4 Harmonic potential functions

Another important family of navigation functions is constructed from harmonic functions [236, 239, 240, 481, 529]. A function ϕ is called a *harmonic function* if it satisfies the differential equation

$$\nabla^2 \phi = \sum_{i=1}^{n} \frac{\partial^2 \phi}{\partial x_i^2} = 0. \tag{8.49}$$

There are many possible solutions to the equation, depending on the conditions along the boundary of the domain over which ϕ is defined. A simple disc-based example is given in [235] for which an analytical solution exists. Complicated navigation functions are generally defined by imposing constraints on ϕ along the boundary of $\mathcal{C}_{free}$. A *Dirichlet boundary condition* means that the boundary must be held to a constant value. Using this condition, a harmonic navigation function can be developed that guides the state into a goal region from anywhere in a simply connected state space. If there are interior obstacles, then a *Neumann boundary condition* forces the velocity vectors to be tangent to the obstacle boundary. By solving (8.49) under a combination of both boundary conditions, a harmonic navigation function can be constructed that avoids obstacles by moving parallel to their boundaries and eventually landing in the goal. It has been shown under general conditions that navigation functions can be produced [240, 239]; however, the main problems are that the boundary of $\mathcal{C}_{free}$ is usually not constructed explicitly (recall why this was avoided in Chapter 5) and that a numerical solution to (8.49) is expensive to compute. This can be achieved, for example, by using Gauss-Seidel iterations (as indicated in [240]), which are related to value iteration (see [96] for the distinction). A sampling-based approach to constructing navigation functions via harmonic functions is presented in [124]. Value iteration will be used to produce approximate, optimal navigation functions in Section 8.5.2.

8.5 Sampling-Based Methods for Continuous Spaces

The methods in Section 8.4 can be considered as the feedback-case analogs to the combinatorial methods of Chapter 6. Although such methods provide elegant solutions to the problem, the issue arises once again that they are either limited to lower dimensional problems or problems that exhibit some special structure. This motivates the introduction of sampling-based methods. This section presents the feedback-case analog to Chapter 5.

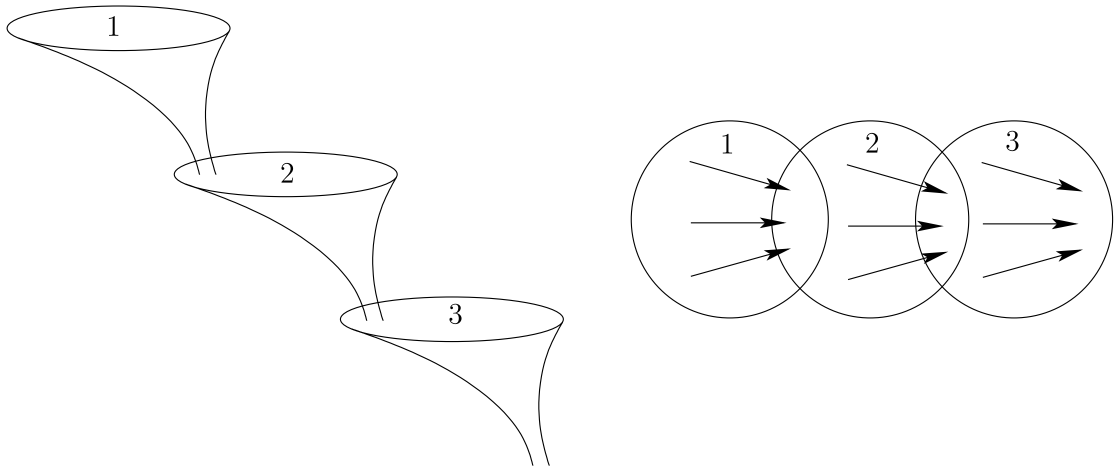


Figure 8.14: A navigation function and corresponding vector field can be designed as a composition of funnels.

8.5.1 Computing a Composition of Funnels

Mason introduced the concept of a *funnel* as a metaphor for motions that converge to the same small region of the state space, regardless of the initial position [679]. As grains of sand in a funnel, they follow the slope of the funnel until they reach the opening at the bottom. A navigation function can be imagined as a funnel that guides the state into the goal. For example, the cost-to-go function depicted in Figure 8.13d can be considered as a complicated funnel that sends each piece of sand along an optimal path to the goal.

Rather than designing a single funnel, consider decomposing the state space into a collection of simple, overlapping regions. Over each region, a funnel can be designed that leads the state into another funnel; see Figure 8.14. As an example, the approach in [162] places a *Lyapunov function* (such functions are covered in Section 15.1.2) over each funnel to ensure convergence to the next funnel. A feedback plan can be constructed by composing several funnels. Starting from some initial state in X, a sequence of funnels is visited until the goal is reached. Each funnel essentially solves the subgoal of reaching the next funnel. Eventually, a funnel is reached that contains the goal, and a navigation function on this funnel causes the goal to be reached. In the context of sensing uncertainty, for which the funnel metaphor was developed, the composition of funnels becomes the preimage planning framework [659], which is covered in Section 12.5.1. In this section, however, it is assumed that the current state is always known.

8.5.1.1 An approximate cover

Figure 8.15 illustrates the notion of an approximate cover, which will be used to represent the funnel domains. Let $\tilde{X}$ denote a subset of a state space X. A *cover* of $\tilde{X}$ is a collection $\mathcal{O}$ of sets for which

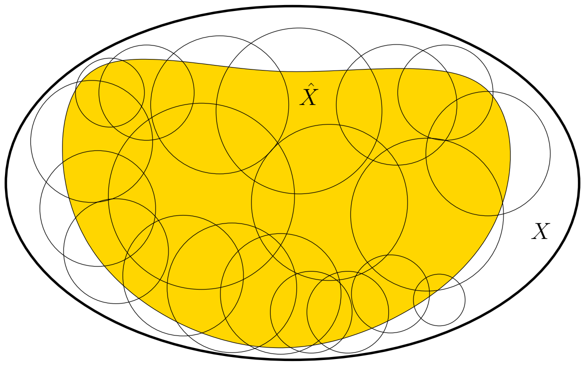


Figure 8.15: An approximate cover is shown. Every point of $\tilde{X}$ is contained in at least one neighborhood, and $\tilde{X}$ is a subset of X.

1. $O \subseteq X$ for each $O \in \mathcal{O}$.
2. $\tilde{X}$ is a subset of the union of all sets in the cover:

$$\tilde{X} \subseteq \bigcup_{O \in \mathcal{O}} O. \tag{8.50}$$

Let each $O \in \mathcal{O}$ be called a *neighborhood*. The notion of a cover was actually used in Section 8.3.2 to define a smooth manifold using a cover of coordinate neighborhoods.

In general, a cover allows the following:

1. Any number of neighborhoods may overlap (have nonempty intersection).
2. Any neighborhood may contain points that lie outside of $\tilde{X}$.

A cell decomposition, which was introduced in Section 6.3.1, is a special kind of cover for which the neighborhoods form a partition of $\tilde{X}$, and they must fit together nicely (recall Figure 6.15).

So far, no constraints have been placed on the neighborhoods. They should be chosen in practice to greatly simplify the design of a navigation function over each one. For the original motion planning problem, cell decompositions were designed to make the determination of a collision-free path trivial in each cell. The same idea applies here, except that we now want to construct a feedback plan. Therefore, it is usually assumed that the cells have a simple shape.

A cover is called *approximate* if $\tilde{X}$ is a strict subset of X. Ideally, we would like to develop an *exact cover*, which implies that $\tilde{X} = X$ and each neighborhood has some nice property, such as being convex. Developing such covers is possible in practice for state spaces that are either low-dimensional or exhibit some special structure. This was observed for the cell decomposition methods of Chapter 6.

Consider constructing an approximate cover for X. The goal should be to cover as much of X as possible. This means that $\mu(X \setminus \tilde{X})$ should be made as small as

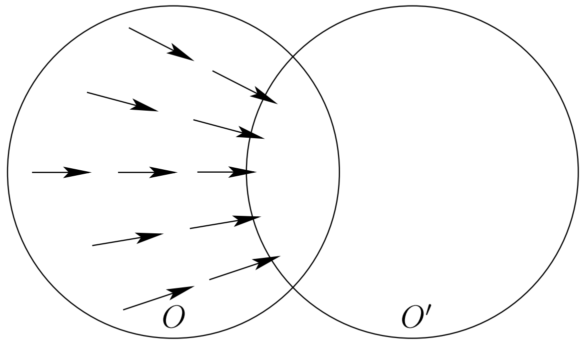


Figure 8.16: A transition from O to O' is caused by a vector field on O for which all integral curves lead into $O \cap O'$.

possible, in which μ denotes Lebesgue measure, as defined in Section 5.1.3. It is also desirable to ensure that $\tilde{X}$ preserves the connectivity of X. In other words, if a path between two points exists in X, then it should also exist in $\tilde{X}$.

8.5.1.2 Defining a feedback plan over a cover

The ideas from Section 8.4.2 can be adapted to define a feedback plan over $\tilde{X}$ using a cover. Let $\check{X}$ denote a discrete state space in which each superstate is a neighborhood. Most of the components of the associated discrete planning problems are the same as in Section 8.4.2. The only difference is in the definition of superactions because neighborhoods can overlap in a cover. For each neighborhood $O \in \mathcal{O}$, a superaction exists for each other neighborhood, $O' \in \mathcal{O}$ such that $O \cap O' \neq \emptyset$ (usually, their interiors overlap to yield $\mathrm{int}(O) \cap \mathrm{int}(O') \neq \emptyset$).

Note that in the case of a cell decomposition, this produces no superactions because it is a partition. To follow the metaphor of composing funnels, the domains of some funnels should overlap, as shown in Figure 8.14. A transition from one neighborhood, O, to another, O', is obtained by defining a vector field on O that sends all states from $O \setminus O'$ into $O \cap O'$; see Figure 8.16. Once O' is reached, the vector field of O is no longer followed; instead, the vector field of O' is used. Using the vector field of O', a transition may be applied to reach another neighborhood. Note that the jump from the vector field of O to that of O' may cause the feedback plan to be a discontinuous vector field on $\tilde{X}$. If the cover is designed so that $O \cap O'$ is large (if they intersect), then gradual transitions may be possible by blending the vector fields from O and O'.

Once the discrete problem has been defined, a discrete feedback plan can be computed over $\check{X}$, as defined in Section 8.2. This is converted into a feedback plan over X by defining a vector field on each neighborhood that causes the appropriate transitions. Each $\check{x} \in \check{X}$ can be interpreted both as a superstate and a neighborhood. For each $\check{x}$, the discrete feedback plan produces a superaction $\check{u} = \pi(\check{x})$, which yields a new neighborhood $\check{x}'$. The vector field over $\check{x} = O$ is then designed to send all states into $\check{x}' = O'$.

If desired, a navigation function ϕ over X can even be derived from a navigation function, $\check{\phi}$, over $\check{X}$. Suppose that $\check{\phi}$ is constructed so that every $\check{\phi}(\check{x})$ is distinct for every $\check{x} \in \check{X}$. Any navigation function can be easily transformed to satisfy this constraint (because $\check{X}$ is finite). Let ϕ_O denote a navigation function over some $O \in \mathcal{O}$. Assume that X_G is a point, x_G (extensions can be made to more general cases). For every neighborhood $O \in \mathcal{O}$ such that $x_G \notin O$, ϕ_O is defined so that performing gradient descent leads into the overlapping neighborhood for which $\check{\phi}(\check{x})$ is smallest. If O contains x_G, the navigation function ϕ_O simply guides the state to x_G.

The navigation functions over each $O \in \mathcal{O}$ can be easily pieced together to yield a navigation function over all of X. In places where multiple neighborhoods overlap, ϕ is defined to be the navigation function associated with the neighborhood for which $\check{\phi}(\check{x})$ is smallest. This can be achieved by adding a large constant to each ϕ_O. Let c denote a constant for which $\phi_O(x) < c$ over all $O \in \mathcal{O}$ and $x \in O$ (it is assumed that each ϕ_O is bounded). Suppose that $\check{\phi}$ assumes only integer values. Let $\mathcal{O}(x)$ denote the set of all $O \in \mathcal{O}$ such that $x \in O$. The navigation function over X is defined as

$$\phi(x) = \min_{O \in \mathcal{O}(x)} \Big\{ \phi_O(x) + c\,\check{\phi}(O) \Big\}. \tag{8.51}$$

8.5.1.3 A sampling-based approach

There are numerous alternative ways to construct a cover. To illustrate the ideas, an approach called the *sampling-based neighborhood graph* is presented here [983]. Suppose that $X = \mathcal{C}_{free}$, which is a subset of some configuration space. As introduced in Section 5.4, let α be a dense, infinite sequence of samples in X. Assume that a collision detection algorithm is available that returns the distance, (5.28), between the robot and obstacles in the world. Such algorithms were described in Section 5.3.

An incremental algorithm is given in Figure 8.17. Initially, $\mathcal{O}$ is empty. In each iteration, if $\alpha(i) \in \mathcal{C}_{free}$ and it is not already contained in some neighborhood, then a new neighborhood is added to $\mathcal{O}$. The two main concerns are 1) how to define a new neighborhood, O, such that $O \subset \mathcal{C}_{free}$, and 2) when to terminate. At any given time, the cover is approximate. The union of all neighborhoods is $\tilde{X}$, which is a strict subset of X. In comparison to Figure 8.15, the cover is a special case in which the neighborhoods do not extend beyond $\tilde{X}$.

Defining new neighborhoods For defining new neighborhoods, it is important to keep them simple because during execution, the neighborhoods that contain the state x must be determined quickly. Suppose that all neighborhoods are open balls:

$$B(x, r) = \{x' \in X \mid \rho(x, x') < r\}, \tag{8.52}$$

in which ρ is the metric on $\mathcal{C}$. There are efficient algorithms for determining whether $x \in O$ for some $O \in \mathcal{O}$, assuming all of the neighborhoods are balls [700].

INCREMENTAL COVER CONSTRUCTION

1. Initialize $\mathcal{O} = \emptyset$ and $i = 1$.
2. Let $x = \alpha(i)$, and let d be the distance returned by the collision detection algorithm applied at x.
3. If $d > 0$ (which implies that $x \in \mathcal{C}_{free}$) and $x \notin O$ for all $O \in \mathcal{O}$, then insert a new neighborhood, O_n, into $\mathcal{O}$. The neighborhood size and shape are determined from x and d.
4. If the termination condition is not satisfied, then let $i := i + 1$, and go to Step 1.
5. Remove any neighborhoods from $\mathcal{O}$ that are contained entirely inside of another neighborhood.

Figure 8.17: The cover is incrementally extended by adding new neighborhoods that are guaranteed to be collision-free.

In practice, methods based on Kd-trees yield good performance [47, 52] (recall Section 5.5.2). A new ball, $B(x, r)$, can be constructed in Step 3 for $x = \alpha(i)$, but what radius can be assigned? For a point robot that translates in $\mathbb{R}^2$ or $\mathbb{R}^3$, the Hausdorff distance d between the robot and obstacles in $\mathcal{W}$ is precisely the distance to $\mathcal{C}_{obs}$ from $\alpha(i)$. This implies that we can set $r = d$, and $B(x, r)$ is guaranteed to be collision-free.

In a general configuration space, it is possible to find a value of r such that $B(x, r) \subseteq \mathcal{C}_{free}$, but in general $r < d$. This issue arose in Section 5.3.4 for checking path segments. The transformations of Sections 3.2 and 3.3 become important in the determination of r. For illustrative purposes, suppose that $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$, which corresponds to a rigid robot, $\mathcal{A}$, that can translate and rotate in $\mathcal{W} = \mathbb{R}^2$. Each point $a \in \mathcal{A}$ is transformed using (3.35). Now imagine starting with some configuration $q = (x, y, \theta)$ and perturbing each coordinate by some Δx, Δy, and $\Delta\theta$. What is the maximum distance that a point on $\mathcal{A}$ could travel? Translation affects all points on $\mathcal{A}$ the same way, but rotation affects points differently. Recall Figure 5.12 from Section 5.3.4. Let $a_r \in \mathcal{A}$ denote the point that is furthest from the origin $(0, 0)$. Let r denote the distance from a_r to the origin. If the rotation is perturbed by some small amount, $\Delta\theta$, then the displacement of any $a \in \mathcal{A}$ is no more than $r\Delta\theta$. If all three configuration parameters are perturbed, then

$$(\Delta x)^2 + (\Delta y)^2 + (r\Delta\theta)^2 < d^2 \tag{8.53}$$

is the constraint that must be satisfied to ensure that the resulting ball is contained in $\mathcal{C}_{free}$. This is actually the equation of a solid ellipsoid, which becomes a ball if $r = 1$. This can be made into a ball by reparameterizing $SE(2)$ so that $\Delta\theta$ has the same affect as Δx and Δy. A transformation $h : \theta \mapsto r\theta$ maps θ into a new domain $Z = [0, 2\pi r)$. In this new space, the equation of the ball is

$$(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2 < d^2, \tag{8.54}$$

in which Δz represents the change in $z \in Z$. The reparameterized version of (3.35) is

$$T = \begin{pmatrix} \cos(\theta/r) & -\sin(\theta/r) & x_t \\ \sin(\theta/r) & \cos(\theta/r) & y_t \\ 0 & 0 & 1 \end{pmatrix}. \tag{8.55}$$

For a 3D rigid body, similar reparameterizations can be made to Euler angles or quaternions to generate six-dimensional balls. Extensions can be made to chains of bodies [983]. One of the main difficulties, however, is that the balls are not the largest possible. In higher dimensions the problem becomes worse because numerous balls are needed, and the radii constructed as described above tend to be much smaller than what is possible. The number of balls can be reduced by also allowing axis-aligned cylinders, but it still remains difficult to construct a cover over a large fraction of $\mathcal{C}_{free}$ in more than six dimensions.

Termination The sampling-based planning algorithms in Chapter 5 were designed to terminate upon finding a solution path. In the current setting, termination is complicated by the fact that we are interested in solutions from all initial configurations. Since α is dense, the volume of uncovered points in $\mathcal{C}_{free}$ tends to zero. After some finite number of iterations, it would be nice to measure the quality of the approximation and then terminate when the desired quality is achieved. This was also possible with the visibility sampling-based roadmap in Section 5.6.2. Using random samples, an estimate of the fraction of $\mathcal{C}_{free}$ can be obtained by recording the percentage of failures in obtaining a sample in $\mathcal{C}_{free}$ that is outside of the cover. For example, if a new neighborhood is created only once in 1000 iterations, then it can be estimated that 99.9 percent of $\mathcal{C}_{free}$ is covered. High-probability bounds can also be determined. Termination conditions are given in [983] that ensure with probability greater than P_c that at least a fraction $\alpha \in (0, 1)$ of $\mathcal{C}_{free}$ has been covered. The constants P_c and α are given as parameters to the algorithm, and it will terminate when the condition has been satisfied using rigorous statistical tests. If deterministic sampling is used, then termination can be made to occur based on the dispersion, which indicates the largest ball in $\mathcal{C}_{free}$ that does not contain the center of another neighborhood. One problem with volume-based criteria, such as those suggested here, is that there is no way to ensure that the cover preserves the connectivity of $\mathcal{C}_{free}$. If two portions of $\mathcal{C}_{free}$ are connected by a narrow passage, the cover may miss a neighborhood that has very small volume yet is needed to connect the two portions.

Example 8.18 (2D Example of Computed Funnels) Figure 8.18 shows a 2D example that was computed using random samples and the algorithm in Figure 8.17. Note that once a cover is computed, it can be used to rapidly compute different navigation functions and vector fields for various goals. This example is mainly

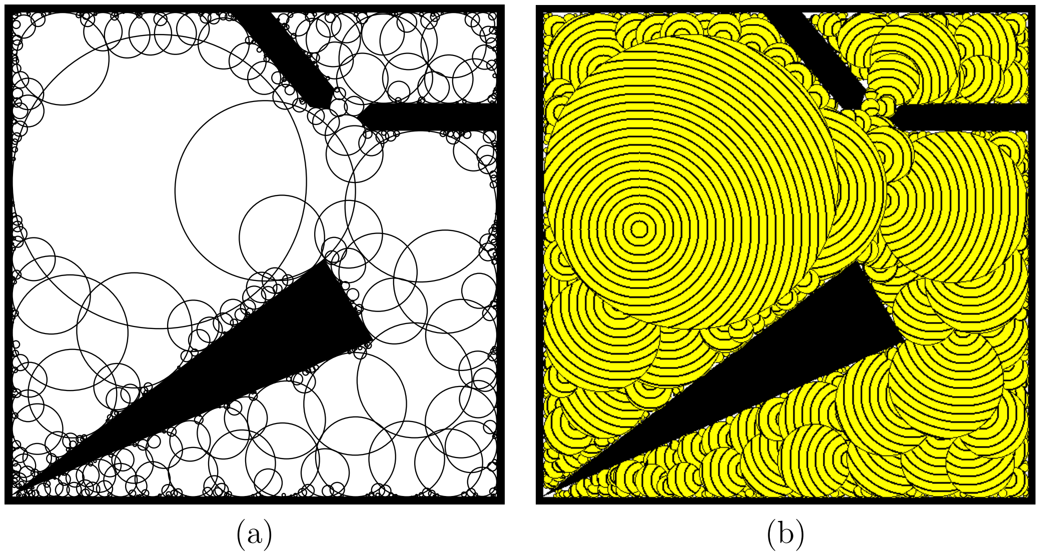


Figure 8.18: (a) A approximate cover for a 2D configuration space. (b) Level sets of a navigation function
.

for illustrative purposes. For the case of a polygonal environment, constructing covers based on convex polygons would be more efficient. ■

8.5.2 Dynamic Programming with Interpolation

This section concludes Part II by solving the motion planning problem with value iteration, which was introduced in Section 2.3. It has already been applied to obtain discrete feedback plans in Section 8.2. It will now be adapted to continuous spaces by allowing interpolation first over a continuous state space and then by additionally allowing interpolation over a continuous action space. This yields a numerical approach to computing optimal navigation functions and feedback plans for motion planning. The focus will remain on backward value iteration; however, the interpolation concepts may also be applied in the forward direction. The approach here views optimal feedback motion planning as a discrete-time optimal control problem [28, 84, 151, 583].

8.5.2.1 Using interpolation for continuous state spaces

Consider a problem formulation that is identical to Formulation 8.1 except that X is allowed to be continuous. Assume that X is bounded, and assume for now that the action space, $U(x)$, it finite for all $x \in X$. Backward value iteration can be applied. The dynamic programming arguments and derivation are identical to

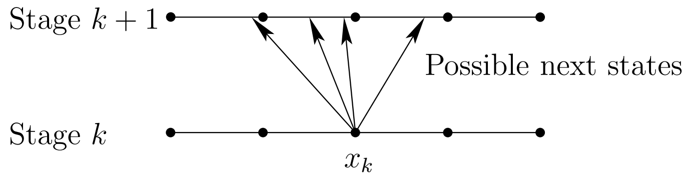


Figure 8.19: Even though x_k is a sample point, the next state, x_{k+1}, may land between sample points. For each $u_k \in U(x_k)$, interpolation may be needed for the resulting next state, $x_{k+1} = f(x_k, u_k)$.

those in Section 2.3. The resulting recurrence is identical to (2.11) and is repeated here for convenience:

$$G^*_k(x_k) = \min_{u_k \in U(x_k)} \Big\{ l(x_k, u_k) + G^*_{k+1}(x_{k+1}) \Big\}. \tag{8.56}$$

The only difficulty is that $G^*_k(x_k)$ cannot be stored for every $x_k \in X$ because X is continuous. There are two general approaches. One is to approximate G^*_k using a parametric family of surfaces, such as polynomials or nonlinear basis functions derived from neural networks [97]. The other is to store G^*_k only over a finite set of sample points and use interpolation to obtain its value at all other points [582, 583].

Suppose that a finite set $S \subset X$ of samples is used to represent cost-to-go functions over X. The evaluation of (8.56) using interpolation is depicted in Figure 8.19. In general, the samples should be chosen to reduce the dispersion (defined in Section 5.2.3) as much as possible. This prevents attempts to approximate the cost-to-go function on large areas that contain no sample points. The rate of convergence ultimately depends on the dispersion [92] (in combination with Lipschitz conditions on the state transition equation and the cost functional). To simplify notation and some other issues, assume that S is a grid of regularly spaced points in $\mathbb{R}^n$.

First, consider the case in which $X = [0,1] \subset \mathbb{R}$. Let $S = \{s_0, s_1, \ldots, s_r\}$, in which $s_i = i/r$. For example, if $r = 3$, then $S = \{0, 1/3, 2/3, 1\}$. Note that this always yields points on the boundary of X, which ensures that for any point in $(0,1)$ there are samples both above and below it. Let i be the largest integer such that $s_i < x$. This implies that $s_{i+1} > x$. The samples s_i and s_{i+1} are called *interpolation neighbors* of x.

The value of G^*_{k+1} in (8.56) at any $x \in [0,1]$ can be obtained via *linear interpolation* as

$$G^*_{k+1}(x) \approx \alpha G^*_{k+1}(s_i) + (1-\alpha) G^*_{k+1}(s_{i+1}), \tag{8.57}$$

in which the coefficient $\alpha \in [0,1]$ is computed as

$$\alpha = 1 - \frac{x - s_i}{r}. \tag{8.58}$$

If $x = s_i$, then $\alpha = 1$, and (8.57) reduces to $G^*_{k+1}(s_i)$, as expected. If $x = s_{i+1}$, then $\alpha = 0$, and (8.57) reduces to $G^*_{k+1}(s_{i+1})$. At all points in between, (8.57) blends the cost-to-go values at s_i and s_{i+1} using α to provide the appropriate weights.

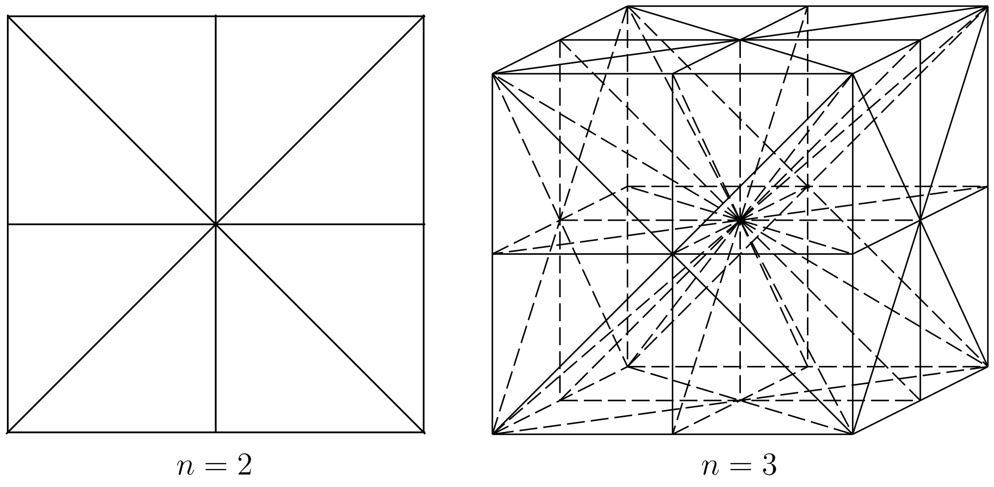


Figure 8.20: Barycentric subdivision can be used to partition each cube into simplexes, which allows interpolation to be performed in $O(n \lg n)$ time, instead of $O(2^n)$.

The interpolation idea can be naturally extended to multiple dimensions. Let X be a bounded subset of $\mathbb{R}^n$. Let S represent an n-dimensional grid of points in $\mathbb{R}^n$. Each sample in S is denoted by $s(i_1, i_2, \ldots, i_n)$. For some $x \in X$, there are 2^n interpolation neighbors that "surround" it. These are the corners of an n-dimensional cube that contains x. Let $x = (x_1, \ldots, x_n)$. Let i_j denote the largest integer for which the jth coordinate of $s(i_1, i_2, \ldots, i_n)$ is less than x_j. The 2^n samples are all those for which either i_j or $i_j + 1$ appears in the expression $s(\cdot, \cdot, \ldots, \cdot)$, for each $j \in \{1, \ldots, n\}$. This requires that samples exist in S for all of these cases. Note that X may be a complicated subset of $\mathbb{R}^n$, provided that for any $x \in X$, all of the required 2^n interpolation neighbors are in S. Using the 2^n interpolation neighbors, the value of G^*_{k+1} in (8.56) on any $x \in X$ can be obtained via *multi-linear interpolation.* In the case of $n = 2$, this is expressed as

$$\begin{aligned} G^*_{k+1}(x) \approx \;& \alpha_1\alpha_2 \, G^*_{k+1}(s(i_1, i_2)) + \\ & \alpha_1(1-\alpha_2) \, G^*_{k+1}(s(i_1, i_2+1)) + \\ & (1-\alpha_1)\alpha_2 \, G^*_{k+1}(s(i_1+1, i_2)) + \\ & (1-\alpha_1)(1-\alpha_2) \, G^*_{k+1}(s(i_1+1, i_2+1)), \end{aligned} \tag{8.59}$$

in which α_1 and α_2 are defined similarly to α in (8.58) but are based on distances along the x_1 and x_2 directions, respectively. The expressions for multi-linear interpolation in higher dimensions are similar but are more cumbersome to express. Higher order interpolation, such a quadratic interpolation may alternatively be used [583].

Unfortunately, the number of interpolation neighbors grows exponentially with the dimension, n. Instead of using all 2^n interpolation neighbors, one improvement is to decompose the cube defined by the 2^n samples into simplexes. Each simplex

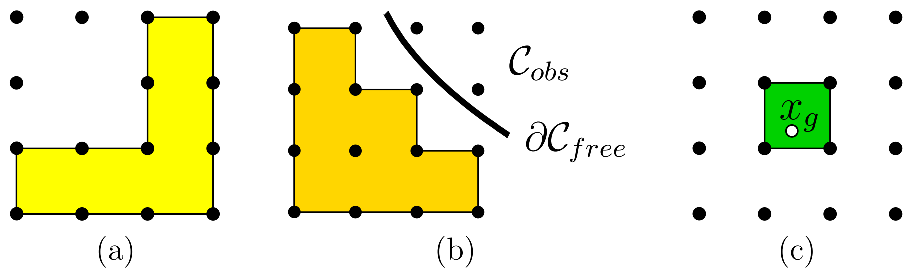


Figure 8.21: (a) An interpolation region, $R(S)$, is shown for a set of sample points, S. (b) The interpolation region that arises due to obstacles. (c) The interpolation region for goal points must not be empty.

has only $n+1$ samples as its vertices. Only the vertices of the simplex that contains x are declared to be the interpolation neighbors of x; this reduces the cost of evaluating $G^*_{k+1}(x)$ to $O(n)$ time. The problem, however, is that determining the simplex that contains x may be a challenging *point-location problem* (a common problem in computational geometry [264]). If barycentric subdivision is used to decompose the cube using the midpoints of all faces, then the point-location problem can be solved in $O(n \lg n)$ time [263, 607, 721], which is an improvement over the $O(2^n)$ scheme described above. Examples of this decomposition are shown for two and three dimensions in Figure 8.20. This is sometimes called the *Coxeter-Freudenthal-Kuhn triangulation*. Even though n is not too large due to practical performance considerations (typically, $n \leq 6$), substantial savings occur in implementations, even for $n = 3$.

It will be convenient to refer directly to the set of all points in X for which all required interpolation neighbors exist. For any finite set $S \subseteq X$ of sample points, let the *interpolation region* $R(S)$ be the set of all $x \in X \setminus S$ for which $G^*(x)$ can be computed by interpolation. This means that $x \in R(S)$ if and only if all interpolation neighbors of x lie in S. Figure 8.21a shows an example. Note that some sample points may not contribute any points to R. If a grid of samples is used to approximate G^*, then the volume of $X \setminus R(S)$ approaches zero as the sampling resolution increases.

Continuous action spaces Now suppose that $U(x)$ is continuous, in addition to X. Assume that $U(x)$ is both a closed and bounded subset of $\mathbb{R}^n$. Once again, the dynamic programming recurrence, (8.56), remains the same. The trouble now is that the *min* represents an optimization problem over an uncountably infinite number of choices. One possibility is to employ nonlinear optimization techniques to select the optimal $u \in U(x)$. The effectiveness of this depends heavily on $U(x)$, X, and the cost functional.

Another approach is to evaluate (8.56) over a finite set of samples drawn from $U(x)$. Again, it is best to choose samples that reduce the dispersion as much as possible. In some contexts, it may be possible to eliminate some actions from consideration by carefully utilizing the properties of the cost-to-go function and its representation via interpolation.

8.5.2.2 The connection to feedback motion planning

The tools have now been provided to solve motion planning problems using value iteration. The configuration space is a continuous state space; let $X = \mathcal{C}_{free}$. The action space is also continuous, $U(x) = T_x(X)$. For motion planning problems, $0 \in T_x(X)$ is only obtained only when u_T is applied. Therefore, it does not need to be represented separately. To compute optimal cost-to-go functions for motion planning, the main concerns are as follows:

1. The action space must be bounded.
2. A discrete-time approximation must be made to derive a state transition equation that works over stages.
3. The cost functional must be discretized.
4. The obstacle region, $\mathcal{C}_{obs}$, must be taken into account.
5. At least some interpolation region must yield $G^*(x) = 0$, which represents the goal region.

We now discuss each of these.

Bounding the action space Recall that using normalized vector fields does not alter the existence of solutions. This is convenient because $U(x)$ needs to be bounded to approximate it with a finite set of samples. It is useful to restrict the action set to obtain

$$U(x) = \{u \in \mathbb{R}^n \mid \; \|u\| \leq 1\}. \tag{8.60}$$

To improve performance, it is sometimes possible to use only those u for which $\|u\| = 1$ or $u = 0$; however, numerical instability problems may arise. A finite sample set for $U(x)$ should have low dispersion and always include $u = 0$.

Obtaining a state transition equation Value iterations occur over discrete stages; however, the integral curves of feedback plans occur over continuous time. Therefore, the time interval T needs to be sampled. Let Δt denote a small positive constant that represents a fixed interval of time. Let the stage index k refer to time $(k-1)\Delta t$. Now consider representing a velocity field $\dot{x}$ over $\mathbb{R}^n$. By definition,

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t}. \tag{8.61}$$

In Section 8.3.1, a velocity field was defined by assigning some $u \in U(x)$ to each $x \in X$. If the velocity vector u is integrated from $x(t)$ over a small Δt, then a new state, $x(t + \Delta t)$, results. If u remains constant, then

$$x(t + \Delta t) = x(t) + \Delta t \; u, \tag{8.62}$$

which is called an *Euler approximation.* If a feedback plan is executed, then u is determined from x via $u = \pi(x(t))$. In general, this means that u could vary as the state is integrated forward. In this case, (8.62) is only approximate,

$$x(t + \Delta t) \approx x(t) + \Delta t \; \pi(x(t)). \tag{8.63}$$

The expression in (8.62) can be considered as a state transition equation that works over stages. Let $x_{k+1} = x(t + \Delta t)$ and $x_k = x(t)$. The transitions can now be expressed as

$$x_{k+1} = f(x_k, u) = x_k + \Delta t \; u. \tag{8.64}$$

The quality of the approximation improves as Δt decreases. Better approximations can be made by using more sample points along time. The most widely known approximations are the Runge-Kutta family. For optimal motion planning, it turns out that the direction vector almost always remains constant along the integral curve. For example, in Figure 8.13d, observe that piecewise-linear paths are obtained by performing gradient descent of the optimal navigation function. The direction vector is constant over most of the resulting integral curve (it changes only as obstacles are contacted). Therefore, approximation problems tend not to arise in motion planning problems. When approximating dynamical systems, such as those presented in Chapter 13, then better approximations are needed; see Section 14.3.2. One important concern is that Δt is chosen in a way that is compatible with the grid resolution. If Δt is so small that the actions do not change the state enough to yield new interpolation neighbors, then the interpolated cost-to-go values will remain constant. This implies that Δt must be chosen to ensure that $x(t + \Delta t)$ has a different set of interpolation neighbors than $x(t)$.

An interesting connection can be made to the approximate motion planning problem that was developed in Section 7.7. Formulation 7.4 corresponds precisely to the approximation defined here, except that ϵ was used instead of Δt because velocities were not yet considered (also, the initial condition was specified because there was no feedback). Recall the different possible action spaces shown in Figure 7.41. As stated in Section 7.7, if the Manhattan or independent-joint models are used, then the configurations remain on a grid of points. This enables discrete value iterations to be performed. A discrete feedback plan and navigation function, as considered in Section 8.2.3, can even be computed. If the Euclidean motion model is used, which is more natural, then the transitions allow a continuum of possible configurations. This case can finally be handled by using interpolation over the configuration space, as described in this section.

Approximating the cost functional A discrete cost functional must be derived from the continuous cost functional, (8.39). The final term is just assigned as $l_F(x_F) = l_F(x(t_f))$. The cost at each stage is

$$l_d(x_k, u_k) = \int_0^{\Delta t} l(x(t), u(t)) dt, \tag{8.65}$$

and $l_d(x_k, u_k)$ is used in the place of $l(x_k, u_k)$ in (8.56). For many problems, the integral does not need to be computed repeatedly. To obtain Euclidean shortest paths, $l_d(x_k, u_k) = \|u_k\|$ can be safely assigned for all $x_k \in X$ and $u_k \in U(x_k)$. A reasonable approximation to (8.65) if Δt is small is $l(x(t), u(t))\Delta t$.

Handling obstacles A simple way to handle obstacles is to determine for each $x \in S$ whether $x \in \mathcal{C}_{obs}$. This can be computed and stored in an array before the value iterations are performed. For rigid robots, this can be efficiently computed using *fast Fourier transforms* [513]. For each $x \in \mathcal{C}_{obs}$, $G^*(x) = \infty$. No value iterations are performed on these states; their values must remain at infinity. During the evaluation of (8.59) (or a higher dimensional version), different actions are attempted. For each action, it is required that all of the interpolation neighbors of x_{k+1} lie in $\mathcal{C}_{free}$. If one of them lies in $\mathcal{C}_{obs}$, then that action produces infinite cost. This has the effect of automatically reducing the interpolation region, $R(S)$, to all cubes whose vertices all lie in $\mathcal{C}_{free}$, as shown in Figure 8.21b. All samples in $\mathcal{C}_{obs}$ are assumed to be deleted from S in the remainder of this section; however, the full grid is still used for interpolation so that infinite values represent the obstacle region.

Note that as expressed so far, it is possible that points in $\mathcal{C}_{obs}$ may lie in $R(S)$ because collision detection is performed only on the samples. In practice, either the grid resolution must be made fine enough to minimize the chance of this error occurring or distance information from a collision detection algorithm must be used to infer that a sufficiently large ball around each sample is collision free. If an interpolation region cannot be assured to lie in $\mathcal{C}_{free}$, then the resolution may have to be increased, at least locally.

Handling the goal region Recall that backward value iterations start with the final cost-to-go function and iterate backward. Initially, the final cost-to-go is assigned as infinity at all states except those in the goal. To properly initialize the final cost-to-go function, there must exist some subset of X over which the zero value can be obtained by interpolation. Let $G = S \cap X_G$. The requirement is that the interpolation region $R(G)$ must be nonempty. If this is not satisfied, then the grid resolution needs to be increased or the goal set needs to be enlarged. If X_g is a single point, then it needs to be enlarged, regardless of the resolution (unless an alternative way to interpolate near a goal point is developed). In the interpolation region shown in Figure 8.21c, all states in the vicinity of x_G yield an interpolated cost-to-go value of zero. If such a region did not exist, then all costs would remain at infinity during the evaluation of (8.59) from any state. Note that Δt must be chosen large enough to ensure that new samples can reach G.

Using G^* as a navigation function After the cost-to-go values stabilize, the resulting cost-to-go function, G^* can be used as a navigation function. Even though G^* is defined only over $S \subset X$, the value of the navigation function can be obtained using interpolation over any point in $R(S)$. The optimal action is selected as the one that satisfies the min in (8.6). This means that the state trajectory does not have to visit the grid points as in the Manhattan model. A trajectory can visit any point in $R(S)$, which enables trajectories to converge to the true optimal solution as Δt and the grid spacing tend to zero.

Topological considerations So far there has been no explicit consideration of the topology of $\mathcal{C}$. Assuming that $\mathcal{C}$ is a manifold, the concepts discussed so far can be applied to any open set on which coordinates are defined. In practice, it is often convenient to use the manifold representations of Section 4.1.2. The manifold can be expressed as a cube, $[0,1]^n$, with some faces identified to obtain $[0,1]^n/\sim$. Over the interior of the cube, all of the concepts explained in this section work without modification. At the boundary, the samples used for interpolation must take the identification into account. Furthermore, actions, u_k, and next states, x_{k+1}, must function correctly on the boundary. One must be careful, however, in declaring that some solution is optimal, because Euclidean shortest paths depend on the manifold parameterization. This ambiguity is usually resolved by formulating the cost in terms of some physical quantity, such as time or energy. This often requires modeling dynamics, which will be covered in Part IV.

Value iteration with interpolation is extremely general. It is a generic algorithm for approximating the solution to optimal control problems. It can be applied to solve many of the problems in Part IV by restricting $U(x)$ to take into account complicated differential constraints. The method can also be extended to problems that involve explicit uncertainty in predictability. This version of value iteration is covered in Section 10.6.

8.5.2.3 Obtaining Dijkstra-like algorithms

For motion planning problems, it is expected that $x(t + \Delta t)$, as computed from (8.62), is always close to $x(t)$ relative to the size of X. This suggests the use of a Dijkstra-like algorithm to compute optimal feedback plans more efficiently. As discussed for the finite case in Section 2.3.3, many values remain unchanged during the value iterations, as indicated in Example 2.5. Dijkstra's algorithm maintains a data structure that focuses the computation on the part of the state space where values are changing. The same can be done for the continuous case by carefully considering the sample points [607].

During the value iterations, there are three kinds of sample points, just as in the discrete case (recall from Section 2.3.3):

1. **Dead:** The cost-to-go has stabilized to its optimal value.
2. **Alive:** The current cost-to-go is finite, but it is not yet known whether the value is optimal.
3. **Unvisited:** The cost-to-go value remains at infinity because the sample has not been reached.

The sets are somewhat harder to maintain for the case of continuous state spaces because of the interaction between the sample set S and the interpolated region $R(S)$.

Imagine the first value iteration. Initially, all points in G are set to zero values. Among the collection of samples S, how many can reach $R(G)$ in a single stage? We expect that samples very far from G will not be able to reach $R(G)$; this keeps their values are infinity. The samples that are close to G should reach it. It would be convenient to prune away from consideration all samples that are too far from G to lower their value. In every iteration, we eliminate iterating over samples that are too far away from those already reached. It is also unnecessary to iterate over the dead samples because their values no longer change.

To keep track of reachable samples, it will be convenient to introduce the notion of a backprojection, which will be studied further in Section 10.1. For a single state, $x \in X$, its *backprojection* is defined as

$$\mathrm{B}(x) = \{x' \in X \mid \exists u' \in U(x') \text{ such that } x = f(x', u')\}. \tag{8.66}$$

The backprojection of a set, $X' \subseteq X$, of points is just the union of backprojections for each point:

$$\mathrm{B}(X') = \bigcup_{x \in X'} \mathrm{B}(x). \tag{8.67}$$

Now consider a version of value iteration that uses backprojections to eliminate some states from consideration because it is known that their values cannot change. Let i refer to the number of stages considered by the current value iteration. During the first iteration, $i = 1$, which means that all one-stage trajectories are considered. Let S be the set of samples (assuming already that none lie in $\mathcal{C}_{obs}$). Let D_i and A_i refer to the *dead* and *alive* samples, respectively. Initially, $D_1 = G$, the set of samples in the goal set. The first set, A_1, of alive samples is assigned by using the concept of a frontier. The *frontier* of a set $S' \subseteq S$ of sample points is

$$\mathrm{Front}(S') = (\mathrm{B}(R(S')) \setminus S') \cap S. \tag{8.68}$$

This is the set of sample points that can reach $R(S')$ in one stage, excluding those already in S'. Figure 8.22 illustrates the frontier. Using (8.68), A_1 is defined as $A_1 = \mathrm{Front}(D_1)$.

Now the approach is described for iteration i. The cost-to-go update (8.56) is computed at all points in A_i. If $G^*_{k+1}(s) = G^*_k(s)$ for some $s \in A_i$, then s is declared dead and moved to D_{i+1}. Samples are never removed from the dead set; therefore, all points in D_i are also added to D_{i+1}. The next active set, A_{i+1}, includes all samples in A_i, excluding those that were moved to the dead set. Furthermore, all samples in $\mathrm{Front}(A_i)$ are added to A_{i+1} because these will produce a finite cost-to-go value in the next iteration. The iterations continue as usual until some stage, m, is reached for which A_m is empty, and D_m includes all samples from which the goal can be reached (under the approximation assumptions made in this section).

For efficiency purposes, an approximation to Front may be used, provided that the true frontier is a proper subset of the approximate frontier. For example, the

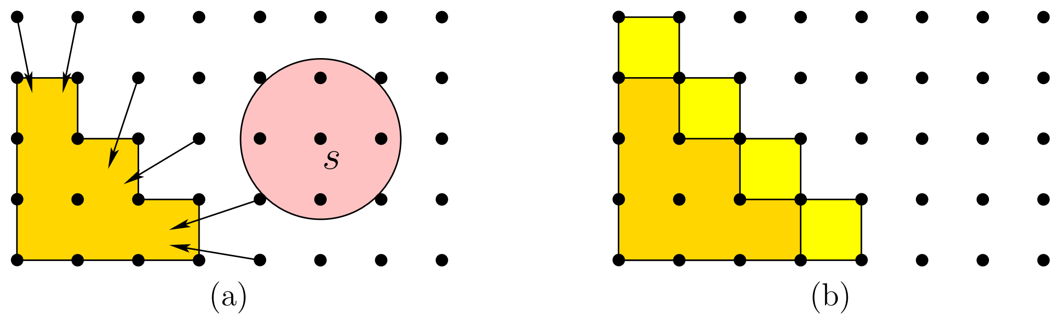


Figure 8.22: An illustration of the frontier concept: (a) the shaded disc indicates the set of all reachable points in one stage, from the sample on the left. The sample cannot reach in one stage the shaded region on the right, which represents $R(S')$. (b) The frontier is the set of samples that can reach $R(S')$. The inclusion of the frontier increases the interpolation region beyond $R(S')$.

frontier might add all new samples within a specified radius of points in S'. In this case, the updated cost-to-go value for some $s \in A_i$ may remain infinite. If this occurs, it is of course not added to D_{i+1}. Furthermore, it is deleted from A_i in the computation of the next frontier (the frontier should only be computed for samples that have finite cost-to-go values).

The approach considered so far can be expected to reduce the amount of computations in each value iteration by eliminating the evaluation of (8.56) on unnecessary samples. The same cost-to-go values are obtained in each iteration because only samples for which the value cannot change are eliminated in each iteration. The resulting algorithm still does not, however, resemble Dijkstra's algorithm because value iterations are performed over all of A_i in each stage.

To make a version that behaves like Dijkstra's algorithm, a queue Q will be introduced. The algorithm removes the smallest element of Q in each iteration. The interpolation version first assigns $G^*(s) = 0$ for each $s \in G$. It also maintains a set D of dead samples, which is initialized to $D = G$. For each $s \in \text{Front}(G)$, the cost-to-go update (8.56) is computed. The priority Q is initialized to $\text{Front}(G)$, and elements are sorted by their current cost-to-go values (which may not be optimal). The algorithm iteratively removes the smallest element from Q (because its optimal cost-to-go is known to be the current value) and terminates when Q is empty. Each time the smallest element, $s_s \in Q$, is removed, it is inserted into D. Two procedures are then performed: 1) Some elements in the queue need to have their cost-to-go values recomputed using (8.56) because the value $G^*(s_s)$ is known to be optimal, and their values may depend on it. These are the samples in Q that lie in $\text{Front}(D)$ (in which D just got extended to include s_s). 2) Any samples in $\text{B}(R(D))$ that are not in Q are inserted into Q after computing their cost-to-go values using (8.56). This enables the active set of samples to grow as the set of dead samples grows. Dijkstra's algorithm with interpolation does not compute values that are identical to those produced by value iterations because

G^*_{k+1} is not explicitly stored when G^*_k is computed. Each computed value is *some* cost-to-go, but it is only known to be the optimal when the sample is placed into D. It can be shown, however, that the method converges because computed values are no higher than what would have been computed in a value iteration. This is also the basis of dynamic programming using Gauss-Seidel iterations [96].

A specialized, wavefront-propagation version of the algorithm can be made for the special case of finding solutions that reach the goal in the smallest number of stages. The algorithm is similar to the one shown in Figure 8.4. It starts with an initial wavefront $W_0 = G$ in which $G^*(s) = 0$ for each $s \in G$. In each iteration, the optimal cost-to-go value i is increased by one, and the wavefront, W_{i+1}, is computed from W_i as $W_{i+1} = \text{Front}(W_i)$. The algorithm terminates at the first iteration in which the wavefront is empty.

Further Reading

There is much less related literature for this chapter in comparison to previous chapters. As explained in Section 8.1, there are historical reasons why feedback is usually separated from motion planning. Navigation functions [541, 829] were one of the most influential ideas in bringing feedback into motion planning; therefore, navigation functions were a common theme throughout the chapter. For other works that use or develop navigation functions, see [206, 274, 750]. The ideas of *progress measures* [317], Lyapunov functions (covered in Section 15.1.1), and cost-to-go functions are all closely related. For Lyapunov-based design of feedback control laws, see [278]. In the context of motion planning, the Error Detection and Recovery (EDR) framework also contains feedback ideas [284].

In [325], the *topological complexity* of C-spaces is studied by characterizing the minimum number of regions needed to cover $\mathcal{C} \times \mathcal{C}$ by defining a continuous path function over each region. This indicates limits on navigation functions that can be constructed, assuming that both q_I and q_G are variables (throughout this chapter, q_G was instead fixed). Further work in this direction includes [326, 327].

To gain better intuitions about properties of vector fields, [44] is a helpful reference, filled with numerous insightful illustrations. A good introduction to smooth manifolds that is particularly suited for control-theory concepts is [133]. Basic intuitions for 2D and 3D curves and surfaces can be obtained from [753]. Other sources for smooth manifolds and differential geometry include [4, 107, 234, 279, 872, 906, 960]. For discussions of piecewise-smooth vector fields, see [27, 634, 846, 998].

Sections 8.4.2 and 8.4.3 were inspired by [235, 643] and [708], respectively. Many difficulties were avoided because discontinuous vector fields were allowed in these approaches. By requiring continuity or smoothness, the subject of Section 8.4.4 was obtained. The material is based mainly on [829, 830]. Other work on navigation functions includes [249, 651, 652].

Section 8.5.1 was inspired mainly by [162, 679], and the approach based on neighborhood graphs is drawn from [983].

Value iteration with interpolation, the subject of Section 8.5.2, is sometimes forgotten in motion planning because computers were not powerful enough at the time it was developed [84, 85, 582, 583]. Presently, however, solutions can be computed for challenging problems with several dimensions (e.g., 3 or 4). Convergence of discretized value iteration to solving the optimal continuous problem was first established in [92], based on Lipschitz conditions on the state transition equation and cost functional. Analyses that take interpolation into account, and general discretization issues, appear in [168, 292, 400, 565, 567]. A multi-resolution variant of value iteration was proposed in [722]. The discussion of Dijkstra-like versions of value iteration was based on [607, 946]. The level-set method is also closely related [532, 534, 533, 862].

Exercises

1. Suppose that a very fast path planning algorithm runs on board of a mobile robot (for example, it may find an answer in a few milliseconds, which is reasonable using trapezoidal decomposition in $\mathbb{R}^2$). Explain how this method can be used to simulate having a feedback plan on the robot. Explain the issues and trade-offs between having a fast on-line algorithm that computes open-loop plans vs. a better off-line algorithm that computes a feedback plan.

2. Use Dijkstra's algorithm to construct navigation functions on a 2D grid with obstacles. Experiment with adding a penalty to the cost functional for getting too close to obstacles.

3. If there are alternative routes, the NF2 algorithm does not necessarily send the state along the route that has the largest maximum clearance. Fix the NF2 algorithm so that it addresses this problem.

4. Tangent space problems:

   (a) For the manifold of unit quaternions, find basis vectors for the tangent space in $\mathbb{R}^4$ at any point.

   (b) Find basis vectors for the tangent space in $\mathbb{R}^9$, assuming that matrices in $SO(3)$ are parameterized with quaternions, as shown in (4.20).

5. Extend the algorithm described in Section 8.4.3 to make it work for polygons that have holes. See Example 8.16 for a similar problem.

6. Give a complete algorithm that uses the vertical cell decomposition for a polygonal obstacle region in $\mathbb{R}^2$ to construct a vector field that serves as a feedback plan. The vector field may be discontinuous.

7. Figure 8.23 depicts a 2D example for which X_{free} is an open annulus. Consider designing a vector field for which all integral curves flow into X_G and the vector field is continuous outside of X_G. Either give a vector field that achieves this or explain why it is not possible.

8. Use the maximum-clearance roadmap idea from Section 6.2.3 to define a cell decomposition and feedback motion plan (vector field) that maximizes clearance. The vector field may be discontinuous.

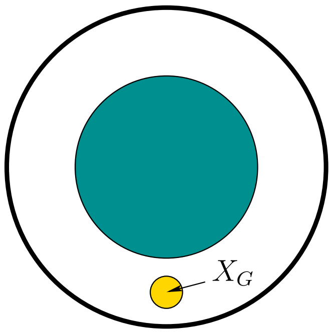


Figure 8.23: Consider designing a continuous vector field that flows into X_G.

9. Develop an algorithm that computes an exact cover for a polygonal configuration space and ensures that if two neighborhoods intersect, then their intersection always contains an open set (i.e., the overlap region is two-dimensional). The neighborhoods in the cover should be polygonal.

10. Using a distance measurement and Euler angles, determine the expression for a collision-free ball that can be inferred (make the ball as large as possible). This should generalize (8.54).

11. Using a distance measurement and quaternions, determine the expression for a collision-free ball (once again, make it as large as possible).

12. Generalize the multi-linear interpolation scheme in (8.59) from 2 to n dimensions.

13. Explain the convergence problems for value iteration that can result if $\|u\| = 1$ is used to constraint the set of allowable actions, instead of $\|u\| \leq 1$.

Implementations

14. Experiment with numerical methods for solving the function (8.49) in two dimensions under various boundary conditions. Report on the efficiency and accuracy of the methods. How well can they be applied in higher dimensions?

15. Implement value iteration with interpolation (it is not necessary to use the method in Figure 8.20) for a polygonal robot that translates and rotates among polygonal obstacles in $\mathcal{W} = \mathbb{R}^2$. Define the cost functional so that the distance traveled is obtained with respect to a weighted Euclidean metric (the weights that compare rotation to translation can be set arbitrarily).

16. Evaluate the efficiency of the interpolation method shown in Figure 8.20 applied to multi-linear interpolation given by generalizing (8.59) as in Exercise 12. You do not need to implement the full value iteration approach (alternatively, this could be done, which provides a better comparison of the overall performance).

17. Implement the method of Section 8.4.2 of computing vector fields on a triangulation. For given input polygons, have your program draw a needle diagram of the

computed vector field. Determine how fast the vector field can be recomputed as the goal changes.

18. Optimal navigation function problems:

    (a) Implement the algorithm illustrated in Figure 8.13. Show the level sets of the optimal cost-to-go function.

    (b) Extend the algorithm and implementation to the case in which there are polygonal holes in X_{free}.

19. Adapt value iteration with interpolation so that a point robot moving in the plane can keep track of a predictable moving point called a *target*. The cost functional should cause a small penalty to be added if the target is not visible. Optimizing this should minimize the amount of time that the target is not visible. Assume that the initial configuration of the robot is given. Compute optimal feedback plans for the robot.

20. Try to experimentally construct navigation functions by adding potential functions that repel the state away from obstacles and attract the state toward x_G. For simplicity, you may assume that $X = \mathbb{R}^2$ and the obstacles are discs. Start with a single disc and then gradually construct more complicated obstacle regions. How difficult is it to ensure that the resulting potential function has no local minima outside of x_G?