Chapter 5

Sampling-Based Motion Planning

There are two main philosophies for addressing the motion planning problem, in Formulation 4.1 from Section 4.3.1. This chapter presents one of the philosophies, *sampling-based motion planning*, which is outlined in Figure 5.1. The main idea is to avoid the explicit construction of $\mathcal{C}_{obs}$, as described in Section 4.3, and instead conduct a search that probes the C-space with a sampling scheme. This probing is enabled by a collision detection module, which the motion planning algorithm considers as a "black box." This enables the development of planning algorithms that are independent of the particular geometric models. The collision detection module handles concerns such as whether the models are semi-algebraic sets, 3D triangles, nonconvex polyhedra, and so on. This general philosophy has been very successful in recent years for solving problems from robotics, manufacturing, and biological applications that involve thousands and even millions of geometric primitives. Such problems would be practically impossible to solve using techniques that explicitly represent $\mathcal{C}_{obs}$.

Notions of completeness It is useful to define several notions of completeness for sampling-based algorithms. These algorithms have the drawback that they result in weaker guarantees that the problem will be solved. An algorithm is considered *complete* if for any input it correctly reports whether there is a so-

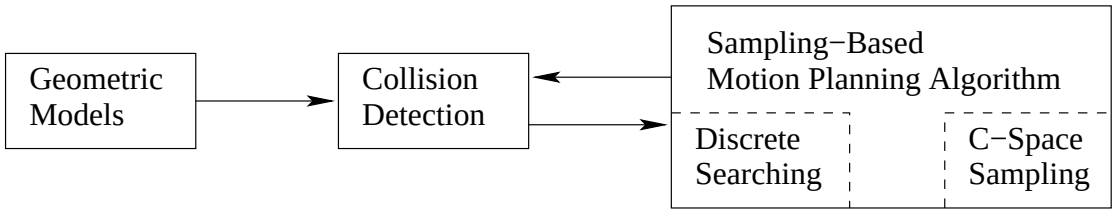


Figure 5.1: The sampling-based planning philosophy uses collision detection as a "black box" that separates the motion planning from the particular geometric and kinematic models. C-space sampling and discrete planning (i.e., searching) are performed.

lution in a finite amount of time. If a solution exists, it must return one in finite time. The combinatorial motion planning methods of Chapter 6 will achieve this. Unfortunately, such completeness is not achieved with sampling-based planning. Instead, weaker notions of completeness are tolerated. The notion of denseness becomes important, which means that the samples come arbitrarily close to any configuration as the number of iterations tends to infinity. A deterministic approach that samples densely will be called *resolution complete*. This means that if a solution exists, the algorithm will find it in finite time; however, if a solution does not exist, the algorithm may run forever. Many sampling-based approaches are based on random sampling, which is dense with probability one. This leads to algorithms that are *probabilistically complete*, which means that with enough points, the probability that it finds an existing solution converges to one. The most relevant information, however, is the rate of convergence, which is usually very difficult to establish.

Section 5.1 presents metric and measure space concepts, which are fundamental to nearly all sampling-based planning algorithms. Section 5.2 presents general sampling concepts and quality criteria that are effective for analyzing the performance of sampling-based algorithms. Section 5.3 gives a brief overview of collision detection algorithms, to gain an understanding of the information available to a planning algorithm and the computation price that must be paid to obtain it. Section 5.4 presents a framework that defines algorithms which solve motion planning problems by integrating sampling and discrete planning (i.e., searching) techniques. These approaches can be considered *single query* in the sense that a single pair, (q_I, q_G), is given, and the algorithm must search until it finds a solution (or it may report early failure). Section 5.5 focuses on *rapidly exploring random trees* (RRTs) and *rapidly exploring dense trees* (RDTs), which are used to develop efficient single-query planning algorithms. Section 5.6 covers *multiple-query* algorithms, which invest substantial preprocessing effort to build a data structure that is later used to obtain efficient solutions for many initial-goal pairs. In this case, it is assumed that the obstacle region $\mathcal{O}$ remains the same for every query.

5.1 Distance and Volume in C-Space

Virtually all sampling-based planning algorithms require a function that measures the distance between two points in $\mathcal{C}$. In most cases, this results in a *metric space*, which is introduced in Section 5.1.1. Useful examples for motion planning are given in Section 5.1.2. It will also be important in many of these algorithms to define the volume of a subset of $\mathcal{C}$. This requires a *measure space*, which is introduced in Section 5.1.3. Section 5.1.4 introduces invariant measures, which should be used whenever possible.

5.1.1 Metric Spaces

It is straightforward to define Euclidean distance in $\mathbb{R}^n$. To define a distance function over any $\mathcal{C}$, however, certain axioms will have to be satisfied so that it coincides with our expectations based on Euclidean distance.

The following definition and axioms are used to create a function that converts a topological space into a metric space.[1] A *metric space* (X, ρ) is a topological space X equipped with a function $\rho : X \times X \to \mathbb{R}$ such that for any $a, b, c \in X$:

1. **Nonnegativity:** $\rho(a, b) \geq 0$.
2. **Reflexivity:** $\rho(a, b) = 0$ if and only if $a = b$.
3. **Symmetry:** $\rho(a, b) = \rho(b, a)$.
4. **Triangle inequality:** $\rho(a, b) + \rho(b, c) \geq \rho(a, c)$.

The function ρ defines distances between points in the metric space, and each of the four conditions on ρ agrees with our intuitions about distance. The final condition implies that ρ is optimal in the sense that the distance from a to c will always be less than or equal to the total distance obtained by traveling through an intermediate point b on the way from a to c.

L_p metrics The most important family of metrics over $\mathbb{R}^n$ is given for any $p \geq 1$ as

$$\rho(x, x') = \left(\sum_{i=1}^{n} |x_i - x'_i|^p \right)^{1/p}. \tag{5.1}$$

For each value of p, (5.1) is called an L_p *metric* (pronounced "el pee"). The three most common cases are

1. L_2: The *Euclidean metric*, which is the familiar Euclidean distance in $\mathbb{R}^n$.
2. L_1: The *Manhattan metric*, which is often nicknamed this way because in $\mathbb{R}^2$ it corresponds to the length of a path that is obtained by moving along an axis-aligned grid. For example, the distance from $(0, 0)$ to $(2, 5)$ is 7 by traveling "east two blocks" and then "north five blocks".
3. L_∞: The L_∞ *metric* must actually be defined by taking the limit of (5.1) as p tends to infinity. The result is

$$L_\infty(x, x') = \max_{1 \leq i \leq n} \{|x_i - x'_i|\}, \tag{5.2}$$

which seems correct because the larger the value of p, the more the largest term of the sum in (5.1) dominates.

[1] Some topological spaces are not *metrizable*, which means that no function exists that satisfies the axioms. Many metrization theorems give sufficient conditions for a topological space to be metrizable [451], and virtually any space that has arisen in motion planning will be metrizable.

An L_p metric can be derived from a norm on a vector space. An L_p *norm* over $\mathbb{R}^n$ is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{5.3}$$

The case of $p = 2$ is the familiar definition of the magnitude of a vector, which is called the *Euclidean norm*. For example, assume the vector space is $\mathbb{R}^n$, and let $\| \cdot \|$ be the standard Euclidean norm. The L_2 metric is $\rho(x, y) = \|x - y\|$. Any L_p metric can be written in terms of a vector subtraction, which is notationally convenient.

Metric subspaces By verifying the axioms, it can be shown that any subspace $Y \subset X$ of a metric space (X, ρ) itself becomes a metric space by restricting the domain of ρ to $Y \times Y$. This conveniently provides metrics on any of the manifolds and varieties from Chapter 4 by simply using any L_p metric on $\mathbb{R}^m$, the space in which the manifold or variety is embedded.

Cartesian products of metric spaces Metrics extend nicely across Cartesian products, which is very convenient because C-spaces are often constructed from Cartesian products, especially in the case of multiple bodies. Let (X, ρ_x) and (Y, ρ_y) be two metric spaces. A metric space (Z, ρ_z) can be constructed for the Cartesian product $Z = X \times Y$ by defining the metric ρ_z as

$$\rho_z(z, z') = \rho_z(x, y, x', y') = c_1 \rho_x(x, x') + c_2 \rho_y(y, y'), \tag{5.4}$$

in which $c_1 > 0$ and $c_2 > 0$ are any positive real constants, and $x, x' \in X$ and $y, y' \in Y$. Each $z \in Z$ is represented as $z = (x, y)$.

Other combinations lead to a metric for Z; for example,

$$\rho_z(z, z') = \Big(c_1 \big[\rho_x(x, x')\big]^p + c_2 \big[\rho_y(y, y')\big]^p \Big)^{1/p}, \tag{5.5}$$

is a metric for any positive integer p. Once again, two positive constants must be chosen. It is important to understand that many choices are possible, and there may not necessarily be a "correct" one.

5.1.2 Important Metric Spaces for Motion Planning

This section presents some metric spaces that arise frequently in motion planning.

Example 5.1 ($SO(2)$ Metric Using Complex Numbers) If $SO(2)$ is represented by unit complex numbers, recall that the C-space is the subset of $\mathbb{R}^2$ given by $\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}$. Any L_p metric from $\mathbb{R}^2$ may be applied. Using the Euclidean metric,

$$\rho(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}, \tag{5.6}$$

for any pair of points (a_1, b_1) and (a_2, b_2). ■

Example 5.2 ($SO(2)$ Metric by Comparing Angles) You might have noticed that the previous metric for $SO(2)$ does not give the distance traveling along the circle. It instead takes a shortcut by computing the length of the line segment in $\mathbb{R}^2$ that connects the two points. This distortion may be undesirable. An alternative metric is obtained by directly comparing angles, θ_1 and θ_2. However, in this case special care has to be given to the identification, because there are two ways to reach θ_2 from θ_1 by traveling along the circle. This causes a min to appear in the metric definition:

$$\rho(\theta_1, \theta_2) = \min\big\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\big\}, \tag{5.7}$$

for which $\theta_1, \theta_2 \in [0, 2\pi]/\sim$. This may alternatively be expressed using the complex number representation $a + bi$ as an angle between two vectors:

$$\rho(a_1, b_1, a_2, b_2) = \cos^{-1}(a_1 a_2 + b_1 b_2), \tag{5.8}$$

for two points (a_1, b_1) and (a_2, b_2). ■

Example 5.3 (An $SE(2)$ Metric) Again by using the subspace principle, a metric can easily be obtained for $SE(2)$. Using the complex number representation of $SO(2)$, each element of $SE(2)$ is a point $(x_t, y_t, a, b) \in \mathbb{R}^4$. The Euclidean metric, or any other L_p metric on $\mathbb{R}^4$, can be immediately applied to obtain a metric. ■

Example 5.4 ($SO(3)$ Metrics Using Quaternions) As usual, the situation becomes more complicated for $SO(3)$. The unit quaternions form a subset $\mathbb{S}^3$ of $\mathbb{R}^4$. Therefore, any L_p metric may be used to define a metric on $\mathbb{S}^3$, but this will not be a metric for $SO(3)$ because antipodal points need to be identified. Let $h_1, h_2 \in \mathbb{R}^4$ represent two unit quaternions (which are being interpreted here as elements of $\mathbb{R}^4$ by ignoring the quaternion algebra). Taking the identifications into account, the metric is

$$\rho(h_1, h_2) = \min\big\{\|h_1 - h_2\|, \|h_1 + h_2\|\big\}, \tag{5.9}$$

in which the two arguments of the min correspond to the distances from h_1 to h_2 and $-h_2$, respectively. The $h_1 + h_2$ appears because h_2 was negated to yield its antipodal point, $-h_2$.

As in the case of $SO(2)$, the metric in (5.9) may seem distorted because it measures the length of line segments that cut through the interior of $\mathbb{S}^3$, as opposed to traveling along the surface. This problem can be fixed to give a very natural metric for $SO(3)$, which is based on *spherical linear interpolation*. This takes the line segment that connects the points and pushes it outward onto $\mathbb{S}^3$. It is easier to visualize this by dropping a dimension. Imagine computing the distance between two points on $\mathbb{S}^2$. If these points lie on the equator, then spherical linear interpolation yields a distance proportional to that obtained by traveling along

the equator, as opposed to cutting through the interior of $\mathbb{S}^2$ (for points not on the equator, use the *great circle* through the points).

It turns out that this metric can easily be defined in terms of the inner product between the two quaternions. Recall that for unit vectors v_1 and v_2 in $\mathbb{R}^n$, $v_1 \cdot v_2 = \cos\theta$, in which θ is the angle between the vectors. This angle is precisely what is needed to give the proper distance along $\mathbb{S}^3$. The resulting metric is a surprisingly simple extension of (5.8). The distance along $\mathbb{S}^3$ between two quaternions is

$$\rho_s(h_1, h_2) = \cos^{-1}(a_1a_2 + b_1b_2 + c_1c_2 + d_1d_2), \tag{5.10}$$

in which each $h_i = (a_i, b_i, c_i, d_i)$. Taking identification into account yields the metric

$$\rho(h_1, h_2) = \min\big\{\rho_s(h_1, h_2), \rho_s(h_1, -h_2)\big\}. \tag{5.11}$$

■

Example 5.5 (Another $SE(2)$ Metric) For many C-spaces, the problem of relating different kinds of quantities arises. For example, any metric defined on $SE(2)$ must compare both distance in the plane and an angular quantity. For example, even if $c_1 = c_2 = 1$, the range for $\mathbb{S}^1$ is $[0, 2\pi)$ using radians but $[0, 360)$ using degrees. If the same constant c_2 is used in either case, two very different metrics are obtained. The units applied to $\mathbb{R}^2$ and $\mathbb{S}^1$ are completely incompatible. ■

Example 5.6 (Robot Displacement Metric) Sometimes this incompatibility problem can be fixed by considering the robot displacement. For any two configurations $q_1, q_2 \in \mathcal{C}$, a robot displacement metric can be defined as

$$\rho(q_1, q_2) = \max_{a \in \mathcal{A}} \big\{\|a(q_1) - a(q_2)\|\big\}, \tag{5.12}$$

in which $a(q_i)$ is the position of the point a in the world when the robot $\mathcal{A}$ is at configuration q_i. Intuitively, the robot displacement metric yields the maximum amount in $\mathcal{W}$ that any part of the robot is displaced when moving from configuration q_1 to q_2. The difficulty and efficiency with which this metric can be computed depend strongly on the particular robot geometric model and kinematics. For a convex polyhedral robot that can translate and rotate, it is sufficient to check only vertices. The metric may appear to be ideal, but efficient algorithms are not known for most situations. ■

Example 5.7 ($\mathbb{T}^n$ Metrics) Next consider making a metric over a torus $\mathbb{T}^n$. The Cartesian product rules such as (5.4) and (5.5) can be extended over every copy of $\mathbb{S}^1$ (one for each parameter θ_i). This leads to n arbitrary coefficients c_1, c_2, ..., c_n.

Robot displacement could be used to determine the coefficients. For example, if the robot is a chain of links, it might make sense to weight changes in the first link more heavily because the entire chain moves in this case. When the last parameter is changed, only the last link moves; in this case, it might make sense to give it less weight. ■

Example 5.8 ($SE(3)$ Metrics) Metrics for $SE(3)$ can be formed by applying the Cartesian product rules to a metric for $\mathbb{R}^3$ and a metric for $SO(3)$, such as that given in (5.11). Again, this unfortunately leaves coefficients to be specified. These issues will arise again in Section 5.3.4, where more details appear on robot displacement. ■

Pseudometrics Many planning algorithms use functions that behave somewhat like a distance function but may fail to satisfy all of the metric axioms. If such distance functions are used, they will be referred to as *pseudometrics*. One general principle that can be used to derive pseudometrics is to define the distance to be the optimal cost-to-go for some criterion (recall discrete cost-to-go functions from Section 2.3). This will become more important when differential constraints are considered in Chapter 14.

In the continuous setting, the cost could correspond to the distance traveled by a robot or even the amount of energy consumed. Sometimes, the resulting pseudometric is not symmetric. For example, it requires less energy for a car to travel downhill as opposed to uphill. Alternatively, suppose that a car is only capable of driving forward. It might travel a short distance to go forward from q_1 to some q_2, but it might have to travel a longer distance to reach q_1 from q_2 because it cannot drive in reverse. These issues arise for the Dubins car, which is covered in Sections 13.1.2 and 15.3.1.

An important example of a pseudometric from robotics is a *potential function*, which is an important part of the randomized potential field method, which is discussed in Section 5.4.3. The idea is to make a scalar function that estimates the distance to the goal; however, there may be additional terms that attempt to repel the robot away from obstacles. This generally causes local minima to appear in the distance function, which may cause potential functions to violate the triangle inequality.

5.1.3 Basic Measure Theory Definitions

This section briefly indicates how to define volume in a metric space. This provides a basis for defining concepts such as integrals or probability densities. Measure theory is an advanced mathematical topic that is well beyond the scope of this

book; however, it is worthwhile to briefly introduce some of the basic definitions because they sometimes arise in sampling-based planning.

Measure can be considered as a function that produces real values for subsets of a metric space, (X, ρ). Ideally, we would like to produce a nonnegative value, $\mu(A) \in [0, \infty]$, for any subset $A \subseteq X$. Unfortunately, due to the Banach-Tarski paradox, if $X = \mathbb{R}^n$, there are some subsets for which trying to assign volume leads to a contradiction. If X is finite, this cannot happen. Therefore, it is hard to visualize the problem; see [836] for a construction of the bizarre nonmeasurable sets. Due to this problem, a workaround was developed by defining a collection of subsets that avoids the paradoxical sets. A collection $\mathcal{B}$ of subsets of X is called a *sigma algebra* if the following axioms are satisfied:

1. The empty set is in $\mathcal{B}$.
2. If $B \in \mathcal{B}$, then $X \setminus B \in \mathcal{B}$.
3. For any collection of a countable number of sets in $\mathcal{B}$, their union must also be in $\mathcal{B}$.

Note that the last two conditions together imply that the intersection of a countable number of sets in $\mathcal{B}$ is also in $\mathcal{B}$. The sets in $\mathcal{B}$ are called the *measurable sets*.

A nice sigma algebra, called the *Borel sets*, can be formed from any metric space (X, ρ) as follows. Start with the set of all open balls in X. These are the sets of the form

$$B(x, r) = \{x' \in X \mid \rho(x, x') < r\} \tag{5.13}$$

for any $x \in X$ and any $r \in (0, \infty)$. From the open balls, the *Borel sets* $\mathcal{B}$ are the sets that can be constructed from these open balls by using the sigma algebra axioms. For example, an open square in $\mathbb{R}^2$ is in $\mathcal{B}$ because it can be constructed as the union of a countable number of balls (infinitely many are needed because the curved balls must converge to covering the straight square edges). By using Borel sets, the nastiness of nonmeasurable sets is safely avoided.

Example 5.9 (Borel Sets) A simple example of $\mathcal{B}$ can be constructed for $\mathbb{R}$. The open balls are just the set of all open intervals, $(x_1, x_2) \subset \mathbb{R}$, for any $x_1, x_2 \in \mathbb{R}$ such that $x_1 < x_2$. ■

Using $\mathcal{B}$, a *measure* μ is now defined as a function $\mu : \mathcal{B} \rightarrow [0, \infty]$ such that the *measure axioms* are satisfied:

1. For the empty set, $\mu(\emptyset) = 0$.
2. For any collection, $E_1, E_2, E_3, \ldots$, of a countable (possibly finite) number of pairwise disjoint, measurable sets, let E denote their union. The measure μ must satisfy

$$\mu(E) = \sum_i \mu(E_i), \tag{5.14}$$

in which i counts over the whole collection.

Example 5.10 (Lebesgue Measure) The most common and important measure is the *Lebesgue measure*, which becomes the standard notions of length in $\mathbb{R}$, area in $\mathbb{R}^2$, and volume in $\mathbb{R}^n$ for $n \geq 3$. One important concept with Lebesgue measure is the existence of sets of *measure zero*. For any countable set A, the Lebesgue measure yields $\mu(A) = 0$. For example, what is the total length of the point $\{1\} \subset \mathbb{R}$? The length of any single point must be zero. To satisfy the measure axioms, sets such as $\{1, 3, 4, 5\}$ must also have measure zero. Even infinite subsets such as $\mathbb{Z}$ and $\mathbb{Q}$ have measure zero in $\mathbb{R}$. If the dimension of a set $A \subseteq \mathbb{R}^n$ is m for some integer $m < n$, then $\mu(A) = 0$, according to the Lebesgue measure on $\mathbb{R}^n$. For example, the set $\mathbb{S}^2 \subset \mathbb{R}^3$ has measure zero because the sphere has no volume. However, if the measure space is restricted to $\mathbb{S}^2$ and then the surface area is defined, then nonzero measure is obtained. ■

Example 5.11 (The Counting Measure) If (X, ρ) is finite, then the *counting measure* can be defined. In this case, the measure can be defined over the entire power set of X. For any $A \subset X$, the counting measure yields $\mu(A) = |A|$, the number of elements in A. Verify that this satisfies the measure axioms. ■

Example 5.12 (Probability Measure) Measure theory even unifies discrete and continuous probability theory. The measure μ can be defined to yield probability mass. The probability axioms (see Section 9.1.2) are consistent with the measure axioms, which therefore yield a measure space. The integrals and sums needed to define expectations of random variables for continuous and discrete cases, respectively, unify into a single measure-theoretic integral. ■

Measure theory can be used to define very general notions of integration that are much more powerful than the Riemann integral that is learned in classical calculus. One of the most important concepts is the *Lebesgue integral*. Instead of being limited to partitioning the domain of integration into intervals, virtually any partition into measurable sets can be used. Its definition requires the notion of a *measurable function* to ensure that the function domain is partitioned into measurable sets. For further study, see [346, 546, 836].

5.1.4 Using the Correct Measure

Since many metrics and measures are possible, it may sometimes seem that there is no "correct" choice. This can be frustrating because the performance of sampling-based planning algorithms can depend strongly on these. Conveniently, there is a

natural measure, called the Haar measure, for some transformation groups, including $SO(N)$. Good metrics also follow from the Haar measure, but unfortunately, there are still arbitrary alternatives.

The basic requirement is that the measure does not vary when the sets are transformed using the group elements. More formally, let G represent a matrix group with real-valued entries, and let μ denote a measure on G. If for any measurable subset $A \subseteq G$, and any element $g \in G$, $\mu(A) = \mu(gA) = \mu(Ag)$, then μ is called the *Haar measure*[2] for G. The notation gA represents the set of all matrices obtained by the product ga, for any $a \in A$. Similarly, Ag represents all products of the form ag.

Example 5.13 (Haar Measure for $SO(2)$) The Haar measure for $SO(2)$ can be obtained by parameterizing the rotations as $[0,1]/\sim$ with 0 and 1 identified, and letting μ be the Lebesgue measure on the unit interval. To see the invariance property, consider the interval $[1/4, 1/2]$, which produces a set $A \subset SO(2)$ of rotation matrices. This corresponds to the set of all rotations from $\theta = \pi/2$ to $\theta = \pi$. The measure yields $\mu(A) = 1/4$. Now consider multiplying every matrix $a \in A$ by a rotation matrix, $g \in SO(2)$, to yield Ag. Suppose g is the rotation matrix for $\theta = \pi$. The set Ag is the set of all rotation matrices from $\theta = 3\pi/2$ up to $\theta = 2\pi = 0$. The measure $\mu(Ag) = 1/4$ remains unchanged. Invariance for gA may be checked similarly. The transformation g translates the intervals in $[0,1]/\sim$. Since the measure is based on interval lengths, it is invariant with respect to translation. Note that μ can be multiplied by a fixed constant (such as 2π) without affecting the invariance property.

An invariant metric can be defined from the Haar measure on $SO(2)$. For any points $x_1, x_2 \in [0,1]$, let $\rho = \mu([x_1, x_2])$, in which $[x_1, x_2]$ is the shortest length (smallest measure) interval that contains x_1 and x_2 as endpoints. This metric was already given in Example 5.2.

To obtain examples that are not the Haar measure, let μ represent probability mass over $[0,1]$ and define any nonuniform probability density function (the uniform density yields the Haar measure). Any shifting of intervals will change the probability mass, resulting in a different measure.

Failing to use the Haar measure weights some parts of $SO(2)$ more heavily than others. Sometimes imposing a bias may be desirable, but it is at least as important to know how to eliminate bias. These ideas may appear obvious, but in the case of $SO(3)$ and many other groups it is more challenging to eliminate this bias and obtain the Haar measure. ■

Example 5.14 (Haar Measure for $SO(3)$) For $SO(3)$ it turns out once again that quaternions come to the rescue. If unit quaternions are used, recall that $SO(3)$ becomes parameterized in terms of $\mathbb{S}^3$, but opposite points are identified.

[2]Such a measure is unique up to scale and exists for any locally compact topological group [346, 836].

It can be shown that the surface area on $\mathbb{S}^3$ is the Haar measure. (Since $\mathbb{S}^3$ is a 3D manifold, it may more appropriately be considered as a surface "volume.") It will be seen in Section 5.2.2 that uniform random sampling over $SO(3)$ must be done with a uniform probability density over $\mathbb{S}^3$. This corresponds exactly to the Haar measure. If instead $SO(3)$ is parameterized with Euler angles, the Haar measure will not be obtained. An unintentional bias will be introduced; some rotations in $SO(3)$ will have more weight than others for no particularly good reason. ■

5.2 Sampling Theory

5.2.1 Motivation and Basic Concepts

The state space for motion planning, $\mathcal{C}$, is uncountably infinite, yet a sampling-based planning algorithm can consider at most a countable number of samples. If the algorithm runs forever, this may be countably infinite, but in practice we expect it to terminate early after only considering a finite number of samples. This mismatch between the cardinality of $\mathcal{C}$ and the set that can be probed by an algorithm motivates careful consideration of sampling techniques. Once the sampling component has been defined, discrete planning methods from Chapter 2 may be adapted to the current setting. Their performance, however, hinges on the way the C-space is sampled.

Since sampling-based planning algorithms are often terminated early, the particular order in which samples are chosen becomes critical. Therefore, a distinction is made between a sample *set* and a sample *sequence*. A unique sample set can always be constructed from a sample sequence, but many alternative sequences can be constructed from one sample set.

Denseness Consider constructing an infinite sample sequence over $\mathcal{C}$. What would be some desirable properties for this sequence? It would be nice if the sequence eventually reached every point in $\mathcal{C}$, but this is impossible because $\mathcal{C}$ is uncountably infinite. Strangely, it is still possible for a sequence to get arbitrarily close to every element of $\mathcal{C}$ (assuming $\mathcal{C} \subseteq \mathbb{R}^m$). In topology, this is the notion of denseness. Let U and V be any subsets of a topological space. The set U is said to be *dense* in V if $\text{cl}(U) = V$ (recall the closure of a set from Section 4.1.1). This means adding the boundary points to U produces V. A simple example is that $(0, 1) \subset \mathbb{R}$ is dense in $[0, 1] \subset \mathbb{R}$. A more interesting example is that the set $\mathbb{Q}$ of rational numbers is both countable and dense in $\mathbb{R}$. Think about why. For any real number, such as $\pi \in \mathbb{R}$, there exists a sequence of fractions that converges to it. This sequence of fractions must be a subset of $\mathbb{Q}$. A sequence (as opposed to a set) is called *dense* if its underlying set is dense. The bare minimum for sampling methods is that they produce a dense sequence. Stronger requirements, such as uniformity and regularity, will be explained shortly.

A random sequence is probably dense Suppose that $\mathcal{C} = [0,1]$. One of the simplest ways conceptually to obtain a dense sequence is to pick points at random. Suppose $I \subset [0,1]$ is an interval of length e. If k samples are chosen independently at random,[3] the probability that none of them falls into I is $(1-e)^k$. As k approaches infinity, this probability converges to zero. This means that the probability that any nonzero-length interval in $[0,1]$ contains no points converges to zero. One small technicality exists. The infinite sequence of independently, randomly chosen points is only dense *with probability one*, which is not the same as being guaranteed. This is one of the strange outcomes of dealing with uncountably infinite sets in probability theory. For example, if a number between $[0,1]$ is chosen at random, the probably that $\pi/4$ is chosen is zero; however, it is still possible. (The probability is just the Lebesgue measure, which is zero for a set of measure zero.) For motion planning purposes, this technicality has no practical implications; however, if k is not very large, then it might be frustrating to obtain only probabilistic assurances, as opposed to absolute guarantees of coverage. The next sequence is guaranteed to be dense because it is deterministic.

The van der Corput sequence A beautiful yet underutilized sequence was published in 1935 by van der Corput, a Dutch mathematician [952]. It exhibits many ideal qualities for applications. At the same time, it is based on a simple idea. Unfortunately, it is only defined for the unit interval. The quest to extend many of its qualities to higher dimensional spaces motivates the formal quality measures and sampling techniques in the remainder of this section.

To explain the van der Corput sequence, let $\mathcal{C} = [0,1]/\sim$, in which $0 \sim 1$, which can be interpreted as $SO(2)$. Suppose that we want to place 16 samples in $\mathcal{C}$. An ideal choice is the set $S = \{i/16 \mid 0 \leq i < 16\}$, which evenly spaces the points at intervals of length $1/16$. This means that no point in $\mathcal{C}$ is further than $1/32$ from the nearest sample. What if we want to make S into a sequence? What is the best ordering? What if we are not even sure that 16 points are sufficient? Maybe 16 is too few or even too many.

The first two columns of Figure 5.2 show a naive attempt at making S into a sequence by sorting the points by increasing value. The problem is that after $i = 8$, half of $\mathcal{C}$ has been neglected. It would be preferable to have a nice covering of $\mathcal{C}$ for any i. Van der Corput's clever idea was to reverse the order of the bits, when the sequence is represented with binary decimals. In the original sequence, the most significant bit toggles only once, whereas the least significant bit toggles in every step. By reversing the bits, the most significant bit toggles in every step, which means that the sequence alternates between the lower and upper halves of $\mathcal{C}$. The third and fourth columns of Figure 5.2 show the original and reversed-order binary representations. The resulting sequence dances around $[0,1]/\sim$ in a nice way, as shown in the last two columns of Figure 5.2. Let $\nu(i)$ denote the ith point of the van der Corput sequence.

[3] See Section 9.1.2 for a review of probability theory.

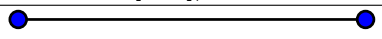
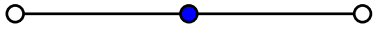
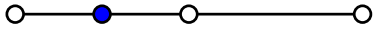
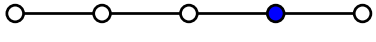
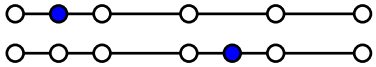
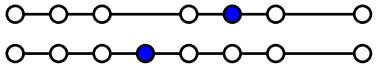
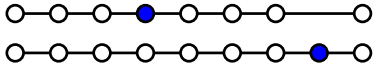
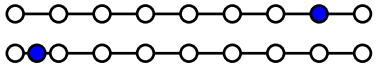
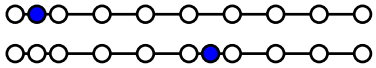
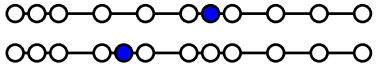
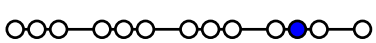
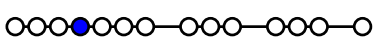
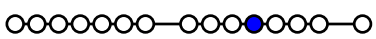
i	Naive Sequence	Binary	Reverse Binary	Van der Corput	Points in $[0,1]/\sim$
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

Figure 5.2: The van der Corput sequence is obtained by reversing the bits in the binary decimal representation of the naive sequence.

In contrast to the naive sequence, each $\nu(i)$ lies far away from $\nu(i+1)$. Furthermore, the first i points of the sequence, for any i, provide reasonably uniform coverage of $\mathcal{C}$. When i is a power of 2, the points are perfectly spaced. For other i, the coverage is still good in the sense that the number of points that appear in any interval of length l is roughly il. For example, when $i = 10$, every interval of length $1/2$ contains roughly 5 points.

The length, 16, of the naive sequence is actually not important. If instead 8 is used, the same $\nu(1)$, ..., $\nu(8)$ are obtained. Observe in the reverse binary column of Figure 5.2 that this amounts to removing the last zero from each binary decimal representation, which does not alter their values. If 32 is used for the naive sequence, then the same $\nu(1)$, ..., $\nu(16)$ are obtained, and the sequence continues nicely from $\nu(17)$ to $\nu(32)$. To obtain the van der Corput sequence from $\nu(33)$ to $\nu(64)$, six-bit sequences are reversed (corresponding to the case in which the naive sequence has 64 points). The process repeats to produce an infinite sequence that does not require a fixed number of points to be specified a priori. In addition to the nice uniformity properties for every i, the infinite van der Corput sequence is also dense in $[0,1]/\sim$. This implies that every open subset must contain at least one sample.

You have now seen ways to generate nice samples in a unit interval both randomly and deterministically. Sections 5.2.2–5.2.4 explain how to generate dense samples with nice properties in the complicated spaces that arise in motion plan-

ning.

5.2.2 Random Sampling

Now imagine moving beyond $[0, 1]$ and generating a dense sample sequence for any bounded C-space, $\mathcal{C} \subseteq \mathbb{R}^m$. In this section the goal is to generate *uniform random* samples. This means that the probability density function $p(q)$ over $\mathcal{C}$ is uniform. Wherever relevant, it also will mean that the probability density is also consistent with the Haar measure. We will not allow any artificial bias to be introduced by selecting a poor parameterization. For example, picking uniform random Euler angles does *not* lead to uniform random samples over $SO(3)$. However, picking uniform random unit quaternions works perfectly because quaternions use the same parameterization as the Haar measure; both choose points on $\mathbb{S}^3$.

Random sampling is the easiest of all sampling methods to apply to C-spaces. One of the main reasons is that C-spaces are formed from Cartesian products, and independent random samples extend easily across these products. If $X = X_1 \times X_2$, and uniform random samples x_1 and x_2 are taken from X_1 and X_2, respectively, then (x_1, x_2) is a uniform random sample for X. This is very convenient in implementations. For example, suppose the motion planning problem involves 15 robots that each translate for any $(x_t, y_t) \in [0, 1]^2$; this yields $\mathcal{C} = [0, 1]^{30}$. In this case, 30 points can be chosen uniformly at random from $[0, 1]$ and combined into a 30-dimensional vector. Samples generated this way are uniformly randomly distributed over $\mathcal{C}$. Combining samples over Cartesian products is much more difficult for nonrandom (deterministic) methods, which are presented in Sections 5.2.3 and 5.2.4.

Generating a random element of $SO(3)$ One has to be very careful about sampling uniformly over the space of rotations. The probability density must correspond to the Haar measure, which means that a random rotation should be obtained by picking a point at random on $\mathbb{S}^3$ and forming the unit quaternion. An extremely clever way to sample $SO(3)$ uniformly at random is given in [883] and is reproduced here. Choose three points $u_1, u_2, u_3 \in [0, 1]$ uniformly at random. A uniform, random quaternion is given by the simple expression

$$h = (\sqrt{1 - u_1} \sin 2\pi u_2,\ \sqrt{1 - u_1} \cos 2\pi u_2,\ \sqrt{u_1} \sin 2\pi u_3,\ \sqrt{u_1} \cos 2\pi u_3). \quad (5.15)$$

A full explanation of the method is given in [883], and a brief intuition is given here. First drop down a dimension and pick $u_1, u_2 \in [0, 1]$ to generate points on $\mathbb{S}^2$. Let u_1 represent the value for the third coordinate, $(0, 0, u_1) \in \mathbb{R}^3$. The slice of points on $\mathbb{S}^2$ for which u_1 is fixed for $0 < u_1 < 1$ yields a circle on $\mathbb{S}^2$ that corresponds to some line of latitude on $\mathbb{S}^2$. The second parameter selects the longitude, $2\pi u_2$. Fortunately, the points are uniformly distributed over $\mathbb{S}^2$. Why? Imagine $\mathbb{S}^2$ as the crust on a spherical loaf of bread that is run through a bread slicer. The slices are cut in a direction parallel to the equator and are of equal thickness. The crusts of each slice have equal area; therefore, the points are uniformly distributed. The

method proceeds by using that fact that $\mathbb{S}^3$ can be partitioned into a spherical arrangement of circles (known as the Hopf fibration); there is an $\mathbb{S}^1$ copy for each point in $\mathbb{S}^2$. The method above is used to provide a random point on $\mathbb{S}^2$ using u_2 and u_3, and u_1 produces a random point on $\mathbb{S}^1$; they are carefully combined in (5.15) to yield a random rotation. To respect the antipodal identification for rotations, any quaternion h found in the lower hemisphere (i.e., $a < 0$) can be negated to yield $-h$. This does not distort the uniform random distribution of the samples.

Generating random directions Some sampling-based algorithms require choosing motion directions at random.[4] From a configuration q, the possible directions of motion can be imagined as being distributed around a sphere. In an $(n+1)$-dimensional C-space, this corresponds to sampling on $\mathbb{S}^n$. For example, choosing a direction in $\mathbb{R}^2$ amounts to picking an element of $\mathbb{S}^1$; this can be parameterized as $\theta \in [0, 2\pi]/\sim$. If $n = 4$, then the previously mentioned trick for $SO(3)$ should be used. If $n = 3$ or $n > 4$, then samples can be generated using a slightly more expensive method that exploits spherical symmetries of the multidimensional Gaussian density function [341]. The method is explained for $\mathbb{R}^{n+1}$; boundaries and identifications must be taken into account for other spaces. For each of the $n+1$ coordinates, generate a sample u_i from a zero-mean Gaussian distribution with the same variance for each coordinate. Following from the Central Limit Theorem, u_i can be approximately obtained by generating k samples at random over $[-1, 1]$ and adding them ($k \geq 12$ is usually sufficient in practice). The vector $(u_1, u_2, \ldots, u_{n+1})$ gives a random direction in $\mathbb{R}^{n+1}$ because each u_i was obtained independently, and the level sets of the resulting probability density function are spheres. We did not use uniform random samples for each u_i because this would bias the directions toward the corners of a cube; instead, the Gaussian yields spherical symmetry. The final step is to normalize the vector by taking $u_i/\|u\|$ for each coordinate.

Pseudorandom number generation Although there are advantages to uniform random sampling, there are also several disadvantages. This motivates the consideration of deterministic alternatives. Since there are trade-offs, it is important to understand how to use both kinds of sampling in motion planning. One of the first issues is that computer-generated numbers are not random.[5] A *pseudorandom number generator* is usually employed, which is a deterministic method that simulates the behavior of randomness. Since the samples are not truly random, the advantage of extending the samples over Cartesian products does not necessarily hold. Sometimes problems are caused by unforeseen deterministic dependencies. One of the best pseudorandom number generators for avoiding such

[4]The directions will be formalized in Section 8.3.2 when smooth manifolds are introduced. In that case, the directions correspond to the set of possible velocities that have unit magnitude. Presently, the notion of a direction is only given informally.

[5]There are exceptions, which use physical phenomena as a random source [808].

troubles is the Mersenne twister [684], for which implementations can be found on the Internet.

To help see the general difficulties, the classical *linear congruential* pseudo-random number generator is briefly explained [619, 738]. The method uses three integer parameters, M, a, and c, which are chosen by the user. The first two, M and a, must be relatively prime, meaning that $\gcd(M, a) = 1$. The third parameter, c, must be chosen to satisfy $0 \leq c < M$. Using modular arithmetic, a sequence can be generated as

$$y_{i+1} = ay_i + c \mod M, \tag{5.16}$$

by starting with some arbitrary seed $1 \leq y_0 \leq M$. Pseudorandom numbers in $[0, 1]$ are generated by the sequence

$$x_i = y_i/M. \tag{5.17}$$

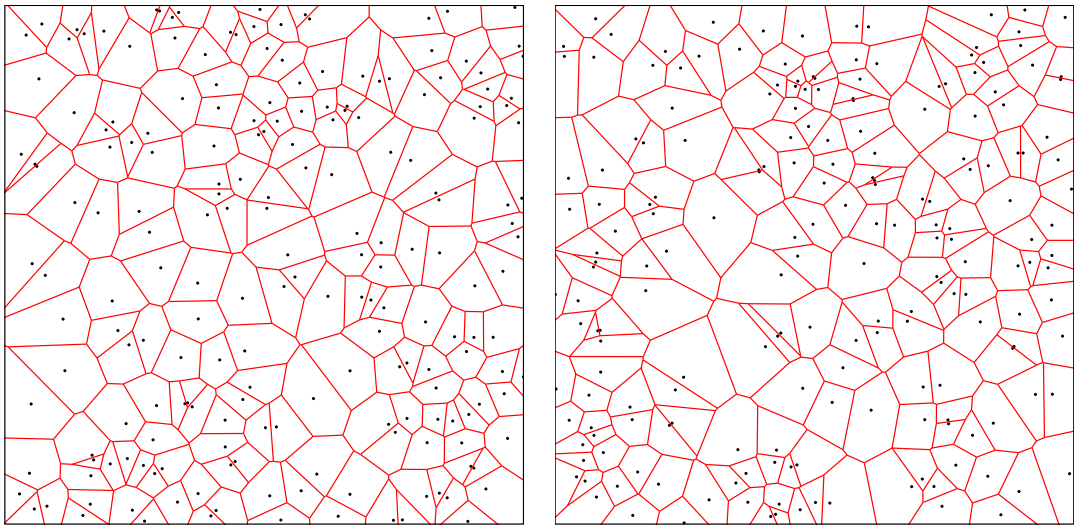
The sequence is periodic; therefore, M is typically very large (e.g., $M = 2^{31} - 1$). Due to periodicity, there are potential problems of regularity appearing in the samples, especially when applied across a Cartesian product to generate points in $\mathbb{R}^n$. Particular values must be chosen for the parameters, and statistical tests are used to evaluate the samples either experimentally or theoretically [738].

Testing for randomness Thus, it is important to realize that even the "random" samples are deterministic. They are designed to optimize performance on statistical tests. Many sophisticated statistical tests of uniform randomness are used. One of the simplest, the *chi-square test*, is described here. This test measures how far computed statistics are from their expected value. As a simple example, suppose $\mathcal{C} = [0, 1]^2$ and is partitioned into a 10 by 10 array of 100 square boxes. If a set P of k samples is chosen at random, then intuitively each box should receive roughly $k/100$ of the samples. An error function can be defined to measure how far from true this intuition is:

$$e(P) = \sum_{i=1}^{100} (b_i - k/100)^2, \tag{5.18}$$

in which b_i is the number of samples that fall into box i. It is shown [521] that $e(P)$ follows a *chi-squared* distribution. A surprising fact is that the goal is not to minimize $e(P)$. If the error is too small, we would declare that the samples are too uniform to be random! Imagine $k = 1,000,000$ and exactly $10,000$ samples appear in each of the 100 boxes. This yields $e(P) = 0$, but how likely is this to ever occur? The error must generally be larger (it appears in many statistical tables) to account for the irregularity due to randomness.

This irregularity can be observed in terms of *Voronoi diagrams*, as shown in Figure 5.3. The Voronoi diagram partitions $\mathbb{R}^2$ into regions based on the samples. Each sample x has an associated *Voronoi region* $\mathrm{Vor}(x)$. For any point $y \in \mathrm{Vor}(x)$, x is the closest sample to y using Euclidean distance. The different sizes and shapes



(a) 196 pseudorandom samples (b) 196 pseudorandom samples

Figure 5.3: Irregularity in a collection of (pseudo)random samples can be nicely observed with Voronoi diagrams.

of these regions give some indication of the required irregularity of random sampling. This irregularity may be undesirable for sampling-based motion planning and is somewhat repaired by the deterministic sampling methods of Sections 5.2.3 and 5.2.4 (however, these methods also have drawbacks).

5.2.3 Low-Dispersion Sampling

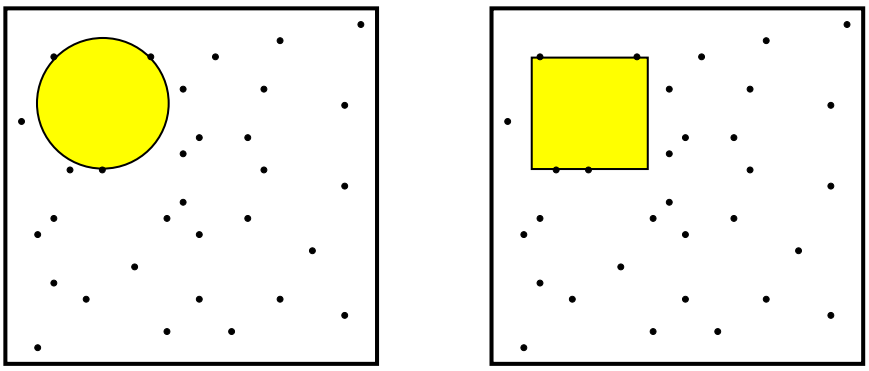
This section describes an alternative to random sampling. Here, the goal is to optimize a criterion called *dispersion* [738]. Intuitively, the idea is to place samples in a way that makes the largest uncovered area be as small as possible. This generalizes of the idea of *grid resolution*. For a grid, the *resolution* may be selected by defining the step size for each axis. As the step size is decreased, the resolution increases. If a grid-based motion planning algorithm can increase the resolution arbitrarily, it becomes resolution complete. Using the concepts in this section, it may instead reduce its dispersion arbitrarily to obtain a resolution complete algorithm. Thus, dispersion can be considered as a powerful generalization of the notion of "resolution."

Dispersion definition The *dispersion*[6] of a finite set P of samples in a metric space (X, ρ) is[7]

$$\delta(P) = \sup_{x \in X} \Big\{ \min_{p \in P} \big\{ \rho(x, p) \big\} \Big\}. \tag{5.19}$$

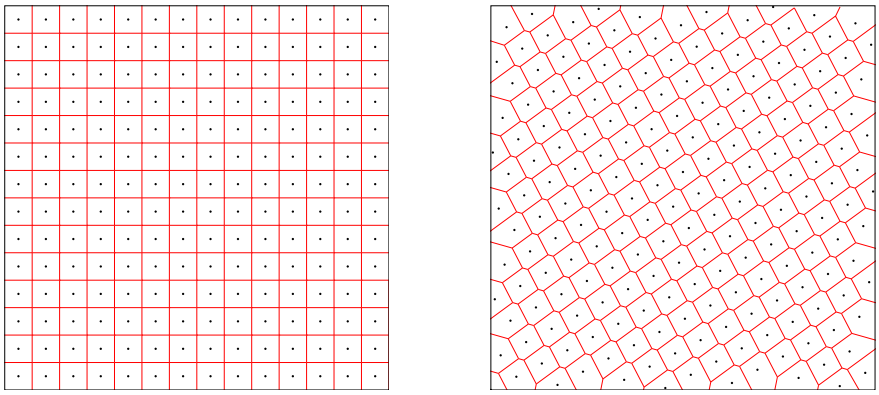
[6]The definition is unfortunately backward from intuition. Lower dispersion means that the points are nicely dispersed. Thus, more dispersion is bad, which is counterintuitive.

[7]The sup represents the *supremum*, which is the least upper bound. If X is closed, then the sup becomes a max. See Section 9.1.1 for more details.



(a) L_2 dispersion (b) L_∞ dispersion

Figure 5.4: Reducing the dispersion means reducing the radius of the largest empty ball.



(a) 196-point Sukharev grid (b) 196 lattice points

Figure 5.5: The Sukharev grid and a nongrid lattice.

Figure 5.4 gives an interpretation of the definition for two different metrics. An alternative way to consider dispersion is as the radius of the largest empty ball (for the L_∞ metric, the balls are actually cubes). Note that at the boundary of X (if it exists), the empty ball becomes truncated because it cannot exceed the boundary. There is also a nice interpretation in terms of Voronoi diagrams. Figure 5.3 can be used to help explain L_2 dispersion in $\mathbb{R}^2$. The *Voronoi vertices* are the points at which three or more Voronoi regions meet. These are points in $\mathcal{C}$ for which the nearest sample is far. An open, empty disc can be placed at any Voronoi vertex, with a radius equal to the distance to the three (or more) closest samples. The radius of the largest disc among those placed at all Voronoi vertices is the dispersion. This interpretation also extends nicely to higher dimensions.

Making good grids Optimizing dispersion forces the points to be distributed more uniformly over $\mathcal{C}$. This causes them to fail statistical tests, but the point distribution is often better for motion planning purposes. Consider the best way to reduce dispersion if ρ is the L_∞ metric and $X = [0,1]^n$. Suppose that the number of samples, k, is given. Optimal dispersion is obtained by partitioning $[0,1]$ into a grid of cubes and placing a point at the center of each cube, as shown for $n = 2$ and $k = 196$ in Figure 5.5a. The number of cubes per axis must be $\lfloor k^{\frac{1}{n}} \rfloor$, in which $\lfloor \cdot \rfloor$ denotes the *floor*. If $k^{\frac{1}{n}}$ is not an integer, then there are leftover points that may be placed anywhere without affecting the dispersion. Notice that $k^{\frac{1}{n}}$ just gives the number of points per axis for a grid of k points in n dimensions. The resulting grid will be referred to as a *Sukharev grid* [922].

The dispersion obtained by the Sukharev grid is the best possible. Therefore, a useful lower bound can be given for *any* set P of k samples [922]:

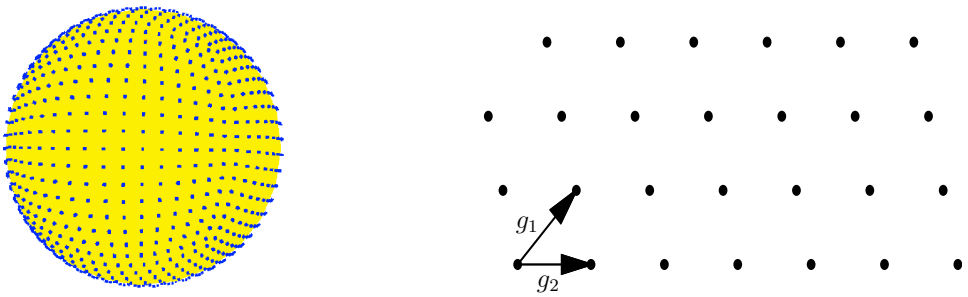
$$\delta(P) \geq \frac{1}{2\lfloor k^{\frac{1}{d}} \rfloor}. \tag{5.20}$$

This implies that keeping the dispersion fixed *requires* exponentially many points in the dimension, d.

At this point you might wonder why L_∞ was used instead of L_2, which seems more natural. This is because the L_2 case is extremely difficult to optimize (except in $\mathbb{R}^2$, where a tiling of equilateral triangles can be made, with a point in the center of each one). Even the simple problem of determining the best way to distribute a fixed number of points in $[0,1]^3$ is unsolved for most values of k. See [241] for extensive treatment of this problem.

Suppose now that other topologies are considered instead of $[0,1]^n$. Let $X = [0,1]/\sim$, in which the identification produces a torus. The situation is quite different because X no longer has a boundary. The Sukharev grid still produces optimal dispersion, but it can also be shifted without increasing the dispersion. In this case, a *standard grid* may also be used, which has the same number of points as the Sukharev grid but is translated to the origin. Thus, the first grid point is $(0,0)$, which is actually the same as $2^n - 1$ other points by identification. If X represents a cylinder and the number of points, k, is given, then it is best to just use the Sukharev grid. It is possible, however, to shift each coordinate that behaves like $\mathbb{S}^1$. If X is rectangular but not a square, a good grid can still be made by tiling the space with cubes. In some cases this will produce optimal dispersion. For complicated spaces such as $SO(3)$, no grid exists in the sense defined so far. It is possible, however, to generate grids on the faces of an inscribed Platonic solid [251] and lift the samples to $\mathbb{S}^n$ with relatively little distortion [987]. For example, to sample $\mathbb{S}^2$, Sukharev grids can be placed on each face of a cube. These are lifted to obtain the warped grid shown in Figure 5.6a.

Example 5.15 (Sukharev Grid) Suppose that $n = 2$ and $k = 9$. If $X = [0,1]^2$, then the Sukharev grid yields points for the nine cases in which either coordinate may be $1/6$, $1/2$, or $5/6$. The L_∞ dispersion is $1/6$. The spacing between the points



(a) (b)

Figure 5.6: (a) A distorted grid can even be placed over spheres and $SO(3)$ by putting grids on the faces of an inscribed cube and lifting them to the surface [987]. (b) A lattice can be considered as a grid in which the generators are not necessarily orthogonal.

along each axis is $1/3$, which is twice the dispersion. If instead $X = [0,1]^2/\sim$, which represents a torus, then the nine points may be shifted to yield the standard grid. In this case each coordinate may be 0, $1/3$, or $2/3$. The dispersion and spacing between the points remain unchanged. ■

One nice property of grids is that they have a lattice structure. This means that neighboring points can be obtained very easily be adding or subtracting vectors. Let g_j be an n-dimensional vector called a *generator*. A point on a lattice can be expressed as

$$x = \sum_{j=1}^{n} k_j g_j \tag{5.21}$$

for n independent generators, as depicted in Figure 5.6b. In a 2D grid, the generators represent "up" and "right." If $X = [0,100]^2$ and a standard grid with integer spacing is used, then the neighbors of the point $(50,50)$ are obtained by adding $(0,1)$, $(0,-1)$, $(-1,0)$, or $(1,0)$. In a general lattice, the generators need not be orthogonal. An example is shown in Figure 5.5b. In Section 5.4.2, lattice structure will become important and convenient for defining the search graph.

Infinite grid sequences Now suppose that the number, k, of samples is not given. The task is to define an infinite sequence that has the nice properties of the van der Corput sequence but works for any dimension. This will become the notion of a *multi-resolution grid*. The resolution can be iteratively doubled. For a multi-resolution standard grid in $\mathbb{R}^n$, the sequence will first place one point at the origin. After 2^n points have been placed, there will be a grid with two points per axis. After 4^n points, there will be four points per axis. Thus, after 2^{ni} points for any positive integer i, a grid with 2^i points per axis will be represented. If only complete grids are allowed, then it becomes clear why they appear inappropriate

for high-dimensional problems. For example, if $n = 10$, then full grids appear after 1, 2^{10}, 2^{20}, 2^{30}, and so on, samples. Each doubling in resolution multiplies the number of points by 2^n. Thus, to use grids in high dimensions, one must be willing to accept *partial grids* and define an infinite sequence that places points in a nice way.

The van der Corput sequence can be extended in a straightforward way as follows. Suppose $X = \mathbb{T}^2 = [0,1]^2/\sim$. The original van der Corput sequence started by counting in binary. The least significant bit was used to select which half of $[0,1]$ was sampled. In the current setting, the two least significant bits can be used to select the quadrant of $[0,1]^2$. The next two bits can be used to select the quadrant within the quadrant. This procedure continues recursively to obtain a complete grid after $k = 2^{2i}$ points, for any positive integer i. For any k, however, there is only a partial grid. The points are distributed with optimal L_∞ dispersion. This same idea can be applied in dimension n by using n bits at a time from the binary sequence to select the orthant (n-dimensional quadrant). Many other orderings produce L_∞-optimal dispersion. Selecting orderings that additionally optimize other criteria, such as discrepancy or L_2 dispersion, are covered in [639, 644]. Unfortunately, it is more difficult to make a multi-resolution Sukharev grid. The base becomes 3 instead of 2; after every 3^{ni} points a complete grid is obtained. For example, in one dimension, the first point appears at $1/2$. The next two points appear at $1/6$ and $5/6$. The next complete one-dimensional grid appears after there are 9 points.

Dispersion bounds Since the sample sequence is infinite, it is interesting to consider asymptotic bounds on dispersion. It is known that for $X = [0,1]^n$ and any L_p metric, the best possible asymptotic dispersion is $O(k^{-1/n})$ for k points and n dimensions [738]. In this expression, k is the variable in the limit and n is treated as a constant. Therefore, any function of n may appear as a constant (i.e., $O(f(n)k^{-1/n}) = O(k^{-1/n})$ for any positive $f(n)$). An important practical consideration is the size of $f(n)$ in the asymptotic analysis. For example, for the van der Corput sequence from Section 5.2.1, the dispersion is bounded by $1/k$, which means that $f(n) = 1$. This does not seem good because for values of k that are powers of two, the dispersion is $1/2k$. Using a multi-resolution Sukharev grid, the constant becomes $3/2$ because it takes a longer time before a full grid is obtained. Nongrid, low-dispersion infinite sequences exist that have $f(n) = 1/\ln 4$ [738]; these are not even uniformly distributed, which is rather surprising.

5.2.4 Low-Discrepancy Sampling

In some applications, selecting points that align with the coordinate axis may be undesirable. Therefore, extensive sampling theory has been developed to determine methods that avoid alignments while distributing the points uniformly. In sampling-based motion planning, grids sometimes yield unexpected behavior because a row of points may align nicely with a corridor in $\mathcal{C}_{free}$. In some cases, a

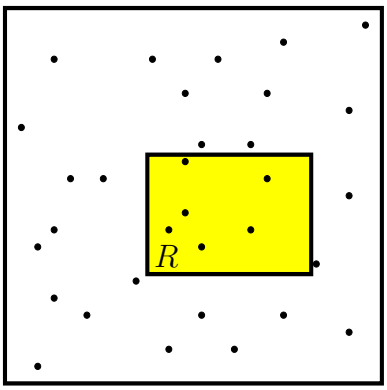


Figure 5.7: Discrepancy measures whether the right number of points fall into boxes. It is related to the chi-square test but optimizes over all possible boxes.

solution is obtained with surprisingly few samples, and in others, too many samples are necessary. These alignment problems, when they exist, generally drive the variance higher in computation times because it is difficult to predict when they will help or hurt. This provides motivation for developing sampling techniques that try to reduce this sensitivity.

Discrepancy theory and its corresponding sampling methods were developed to avoid these problems for numerical integration [738]. Let X be a measure space, such as $[0,1]^n$. Let $\mathcal{R}$ be a collection of subsets of X that is called a *range space*. In most cases, $\mathcal{R}$ is chosen as the set of all axis-aligned rectangular subsets; hence, this will be assumed from this point onward. With respect to a particular point set, P, and range space, $\mathcal{R}$, the *discrepancy* [965] for k samples is defined as (see Figure 5.7)

$$D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \left\{ \left| \frac{|P \cap R|}{k} - \frac{\mu(R)}{\mu(X)} \right| \right\}, \tag{5.22}$$

in which $|P \cap R|$ denotes the number of points in $P \cap R$. Each term in the supremum considers how well P can be used to estimate the volume of R. For example, if $\mu(R)$ is $1/5$, then we would hope that about $1/5$ of the points in P fall into R. The discrepancy measures the largest volume estimation error that can be obtained over all sets in $\mathcal{R}$.

Asymptotic bounds There are many different asymptotic bounds for discrepancy, depending on the particular range space and measure space [682]. The most widely referenced bounds are based on the standard range space of axis-aligned rectangular boxes in $[0,1]^n$. There are two different bounds, depending on whether the number of points, k, is given. The best possible asymptotic discrepancy for a single sequence is $O(k^{-1} \log^n k)$. This implies that k is not specified. If, however, for every k a new set of points can be chosen, then the best possible discrepancy is $O(k^{-1} \log^{n-1} k)$. This bound is lower because it considers the best that can be achieved by a sequence of points sets, in which every point set may be completely

different. In a single sequence, the next set must be extended from the current set by adding a single sample.

Relating dispersion and discrepancy Since balls have positive volume, there is a close relationship between discrepancy, which is measure-based, and dispersion, which is metric-based. For example, for any $P \subset [0,1]^n$,

$$\delta(P, L_\infty) \leq D(P, \mathcal{R})^{1/d}, \tag{5.23}$$

which means low-discrepancy implies low-dispersion. Note that the converse is not true. An axis-aligned grid yields high discrepancy because of alignments with the boundaries of sets in $\mathcal{R}$, but the dispersion is very low. Even though low-discrepancy implies low-dispersion, lower dispersion can usually be obtained by ignoring discrepancy (this is one less constraint to worry about). Thus, a trade-off must be carefully considered in applications.

Low-discrepancy sampling methods Due to the fundamental importance of numerical integration and the intricate link between discrepancy and integration error, most sampling literature has led to low-discrepancy sequences and point sets [738, 893, 937]. Although motion planning is quite different from integration, it is worth evaluating these carefully constructed and well-analyzed samples. Their potential use in motion planning is no less reasonable than using pseudorandom sequences, which were also designed with a different intention in mind (satisfying statistical tests of randomness).

Low-discrepancy sampling methods can be divided into three categories: 1) Halton/Hammersley sampling; 2) (t,s)-sequences and (t,m,s)-nets; and 3) lattices. The first category represents one of the earliest methods, and is based on extending the van der Corput sequence. The *Halton sequence* is an n-dimensional generalization of the van der Corput sequence, but instead of using binary representations, a different basis is used for each coordinate [430]. The result is a reasonable deterministic replacement for random samples in many applications. The resulting discrepancy (and dispersion) is lower than that for random samples (with high probability). Figure 5.8a shows the first 196 Halton points in $\mathbb{R}^2$.

Choose n relatively prime integers $p_1, p_2, \ldots, p_n$ (usually the first n primes, $p_1 = 2$, $p_2 = 3$, ..., are chosen). To construct the ith sample, consider the base-p representation for i, which takes the form $i = a_0 + pa_1 + p^2a_2 + p^3a_3 + \ldots$. The following point in $[0,1]$ is obtained by reversing the order of the bits and moving the decimal point (as was done in Figure 5.2):

$$r(i,p) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \frac{a_3}{p^4} + \cdots . \tag{5.24}$$

For $p = 2$, this yields the ith point in the van der Corput sequence. Starting from $i = 0$, the ith sample in the Halton sequence is

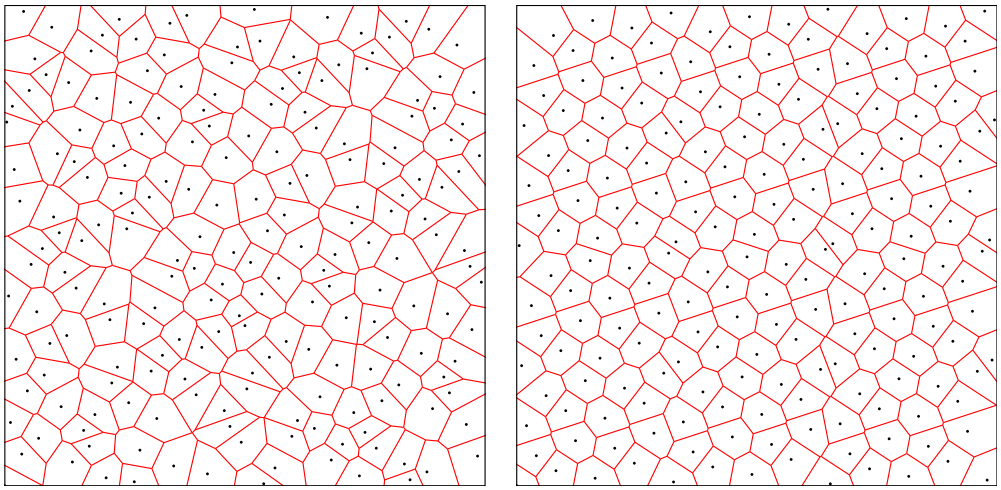
$$\big(r(i,p_1), r(i,p_2), \ldots, r(i,p_n)\big). \tag{5.25}$$

Suppose instead that k, the required number of points, is known. In this case, a better distribution of samples can be obtained. The *Hammersley point set* [431] is an adaptation of the Halton sequence. Using only $d-1$ distinct primes and starting at $i=0$, the ith sample in a Hammersley point set with k elements is

$$\big(i/k, r(i,p_1), \ldots, r(i,p_{n-1})\big). \tag{5.26}$$

Figure 5.8b shows the Hammersley set for $n=2$ and $k=196$.

The construction of Halton/Hammersley samples is simple and efficient, which has led to widespread application. They both achieve asymptotically optimal discrepancy; however, the constant in their asymptotic analysis increases more than exponentially with dimension [738]. The constant for the dispersion also increases exponentially, which is much worse than for the methods of Section 5.2.3.



(a) 196 Halton points (b) 196 Hammersley points

Figure 5.8: The Halton and Hammersley points are easy to construct and provide a nice alternative to random sampling that achieves more regularity. Compare the Voronoi regions to those in Figure 5.3. Beware that although these sequences produce asymptotically optimal discrepancy, their performance degrades substantially in higher dimensions (e.g., beyond 10).

Improved constants are obtained for sequences and finite points by using (t,s)-sequences, and (t,m,s)-nets, respectively [738]. The key idea is to enforce zero discrepancy over particular subsets of $\mathcal{R}$ known as *canonical rectangles*, and all remaining ranges in $\mathcal{R}$ will contribute small amounts to discrepancy. The most famous and widely used (t,s)-sequences are Sobol' and Faure (see [738]). The Niederreiter-Xing (t,s)-sequence has the best-known asymptotic constant, $(a/n)^n$, in which a is a small positive constant [739].

The third category is *lattices*, which can be considered as a generalization of grids that allows nonorthogonal axes [682, 893, 959]. As an example, consider

Figure 5.5b, which shows 196 lattice points generated by the following technique. Let α be a positive irrational number. For a fixed k, generate the ith point according to $(i/k, \{i\alpha\})$, in which $\{\cdot\}$ denotes the fractional part of the real value (modulo-one arithmetic). In Figure 5.5b, $\alpha = (\sqrt{5}+1)/2$, the *golden ratio*. This procedure can be generalized to n dimensions by picking $n-1$ distinct irrational numbers. A technique for choosing the α parameters by using the roots of irreducible polynomials is discussed in [682]. The ith sample in the lattice is

$$\left(\frac{i}{k}, \{i\alpha_1\}, \ldots, \{i\alpha_{n-1}\} \right). \tag{5.27}$$

Recent analysis shows that some lattice sets achieve asymptotic discrepancy that is very close to that of the best-known nonlattice sample sets [445, 938]. Thus, restricting the points to lie on a lattice seems to entail little or no loss in performance, but has the added benefit of a regular neighborhood structure that is useful for path planning. Historically, lattices have required the specification of k in advance; however, there has been increasing interest in extensible lattices, which are infinite sequences [446, 938].

5.3 Collision Detection

Once it has been decided where the samples will be placed, the next problem is to determine whether the configuration is in collision. Thus, collision detection is a critical component of sampling-based planning. Even though it is often treated as a black box, it is important to study its inner workings to understand the information it provides and its associated computational cost. In most motion planning applications, the majority of computation time is spent on collision checking.

A variety of collision detection algorithms exist, ranging from theoretical algorithms that have excellent computational complexity to heuristic, practical algorithms whose performance is tailored to a particular application. The techniques from Section 4.3 can be used to develop a collision detection algorithm by defining a logical predicate using the geometric model of $\mathcal{C}_{obs}$. For the case of a 2D world with a convex robot and obstacle, this leads to an linear-time collision detection algorithm. In general, however, it can be determined whether a configuration is in collision more efficiently by avoiding the full construction of $\mathcal{C}_{obs}$.

5.3.1 Basic Concepts

As in Section 3.1.1, collision detection may be viewed as a logical predicate. In the current setting it appears as $\phi : \mathcal{C} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, in which the domain is $\mathcal{C}$ instead of $\mathcal{W}$. If $q \in \mathcal{C}_{obs}$, then $\phi(q) = \text{TRUE}$; otherwise, $\phi(q) = \text{FALSE}$.

Distance between two sets For the Boolean-valued function ϕ, there is no information about how far the robot is from hitting the obstacles. Such information is very important in planning algorithms. A *distance function* provides this

information and is defined as $d : \mathcal{C} \rightarrow [0, \infty)$, in which the real value in the range of f indicates the distance in the world, $\mathcal{W}$, between the closest pair of points over all pairs from $\mathcal{A}(q)$ and $\mathcal{O}$. In general, for two closed, bounded subsets, E and F, of $\mathbb{R}^n$, the *distance* is defined as

$$\rho(E, F) = \min_{e \in E} \Big\{ \min_{f \in F} \Big\{ \| e - f \| \Big\} \Big\}, \tag{5.28}$$

in which $\| \cdot \|$ is the Euclidean norm. Clearly, if $E \cap F \neq \emptyset$, then $\rho(E, F) = 0$. The methods described in this section may be used to either compute distance or only determine whether $q \in \mathcal{C}_{obs}$. In the latter case, the computation is often much faster because less information is required.

Two-phase collision detection Suppose that the robot is a collection of m attached links, $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_m$, and that $\mathcal{O}$ has k connected components. For this complicated situation, collision detection can be viewed as a two-phase process.

1. **Broad Phase:** In the *broad phase*, the task is to avoid performing expensive computations for bodies that are far away from each other. Simple bounding boxes can be placed around each of the bodies, and simple tests can be performed to avoid costly collision checking unless the boxes overlap. Hashing schemes can be employed in some cases to greatly reduce the number of pairs of boxes that have to be tested for overlap [703]. For a robot that consists of multiple bodies, the pairs of bodies that should be considered for collision must be specified in advance, as described in Section 4.3.1.

2. **Narrow Phase:** In the *narrow phase*, individual pairs of bodies are each checked carefully for collision. Approaches to this phase are described in Sections 5.3.2 and 5.3.3.

5.3.2 Hierarchical Methods

In this section, suppose that two complicated, nonconvex bodies, E and F, are to be checked for collision. Each body could be part of either the robot or the obstacle region. They are subsets of $\mathbb{R}^2$ or $\mathbb{R}^3$, defined using any kind of geometric primitives, such as triangles in $\mathbb{R}^3$. *Hierarchical methods* generally decompose each body into a tree. Each vertex in the tree represents a *bounding region* that contains some subset of the body. The bounding region of the root vertex contains the whole body.

There are generally two opposing criteria that guide the selection of the type of bounding region:

1. The region should fit the intended body points as tightly as possible.

2. The intersection test for two regions should be as efficient as possible.

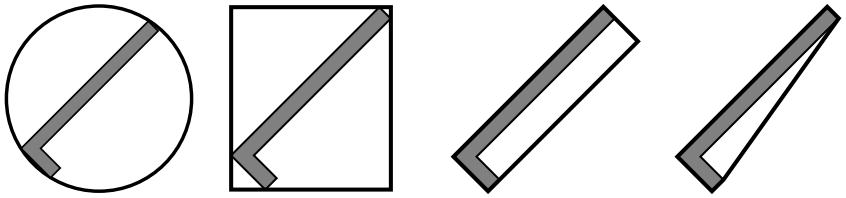


Figure 5.9: Four different kinds of bounding regions: (a) sphere, (b) axis-aligned bounding box (AABB), (c) oriented bounding box (OBB), and (d) convex hull. Each usually provides a tighter approximation than the previous one but is more expensive to test for overlapping pairs.

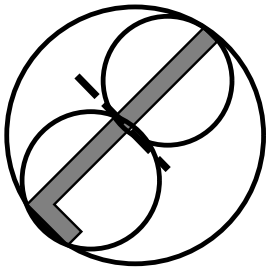


Figure 5.10: The large circle shows the bounding region for a vertex that covers an L-shaped body. After performing a split along the dashed line, two smaller circles are used to cover the two halves of the body. Each circle corresponds to a child vertex.

Several popular choices are shown in Figure 5.9 for an L-shaped body.

The tree is constructed for a body, E (or alternatively, F) recursively as follows. For each vertex, consider the set X of all points in E that are contained in the bounding region. Two child vertices are constructed by defining two smaller bounding regions whose union covers X. The split is made so that the portion covered by each child is of similar size. If the geometric model consists of primitives such as triangles, then a split could be made to separate the triangles into two sets of roughly the same number of triangles. A bounding region is then computed for each of the children. Figure 5.10 shows an example of a split for the case of an L-shaped body. Children are generated recursively by making splits until very simple sets are obtained. For example, in the case of triangles in space, a split is made unless the vertex represents a single triangle. In this case, it is easy to test for the intersection of two triangles.

Consider the problem of determining whether bodies E and F are in collision. Suppose that the trees T_e and T_f have been constructed for E and F, respectively. If the bounding regions of the root vertices of T_e and T_f do not intersect, then it is known that T_e and T_f are not in collision without performing any additional computation. If the bounding regions do intersect, then the bounding regions of

the children of T_e are compared to the bounding region of T_f. If either of these intersect, then the bounding region of T_f is replaced with the bounding regions of its children, and the process continues recursively. As long as the bounding regions overlap, lower levels of the trees are traversed, until eventually the leaves are reached. If triangle primitives are used for the geometric models, then at the leaves the algorithm tests the individual triangles for collision, instead of bounding regions. Note that as the trees are traversed, if a bounding region from the vertex v_1 of T_e does not intersect the bounding region from a vertex, v_2, of T_f, then no children of v_1 have to be compared to children of v_1. Usually, this dramatically reduces the number of comparisons, relative in a naive approach that tests all pairs of triangles for intersection.

It is possible to extend the hierarchical collision detection scheme to also compute distance. The closest pair of points found so far serves as an upper bound that prunes aways some future pairs from consideration. If a pair of bounding regions has a distance greater than the smallest distance computed so far, then their children do not have to be considered [638]. In this case, an additional requirement usually must be imposed. Every bounding region must be a proper subset of its parent bounding region [807]. If distance information is not needed, then this requirement can be dropped.

5.3.3 Incremental Methods

This section focuses on a particular approach called *incremental distance computation*, which assumes that between successive calls to the collision detection algorithm, the bodies move only a small amount. Under this assumption the algorithm achieves "almost constant time" performance for the case of convex polyhedral bodies [636, 702]. Nonconvex bodies can be decomposed into convex components.

These collision detection algorithms seem to offer wonderful performance, but this comes at a price. The models must be *coherent*, which means that all of the primitives must fit together nicely. For example, if a 2D model uses line segments, all of the line segments must fit together perfectly to form polygons. There can be no isolated segments or chains of segments. In three dimensions, polyhedral models are required to have all faces come together perfectly to form the boundaries of 3D shapes. The model cannot be an arbitrary collection of 3D triangles.

The method will be explained for the case of 2D convex polygons, which are interpreted as convex subsets of $\mathbb{R}^2$. Voronoi regions for a convex polygon will be defined in terms of features. The *feature set* is the set of all vertices and edges of a convex polygon. Thus, a polygon with n edges has $2n$ features. Any point outside of the polygon has a closest feature in terms of Euclidean distance. For a given feature, F, the set of all points in $\mathbb{R}^2$ from which F is the closest feature is called the Voronoi region of F and is denoted $\mathrm{Vor}(F)$. Figure 5.11 shows all ten Voronoi regions for a pentagon. Each feature is considered as a point set in the discussion below.

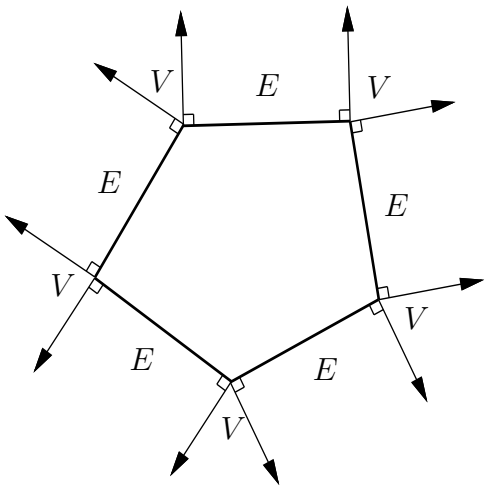


Figure 5.11: The Voronoi regions alternate between being edge-based and vertex-based. The Voronoi regions of vertices are labeled with a "V" and the Voronoi regions of edges are labeled with an "E."

For any two convex polygons that do not intersect, the closest point is determined by a pair of points, one on each polygon (the points are unique, except in the case of parallel edges). Consider the feature for each point in the closest pair. There are only three possible combinations:

- **Vertex-Vertex** Each point of the closest pair is a vertex of a polygon.
- **Edge-Vertex** One point of the closest pair lies on an edge, and the other lies on a vertex.
- **Edge-Edge** Each point of the closest pair lies on an edge. In this case, the edges must be parallel.

Let P_1 and P_2 be two convex polygons, and let F_1 and F_2 represent any feature pair, one from each polygon. Let $(x_1, y_1) \in F_1$ and $(x_2, y_2) \in F_2$ denote the closest pair of points, among all pairs of points in F_1 and F_2, respectively. The following condition implies that the distance between (x_1, y_1) and (x_2, y_2) is the distance between P_1 and P_2:

$$(x_1, y_1) \in \text{Vor}(F_2) \text{ and } (x_2, y_2) \in \text{Vor}(F_1). \tag{5.29}$$

If (5.29) is satisfied for a given feature pair, then the distance between P_1 and P_2 equals the distance between F_1 and F_2. This implies that the distance between P_1 and P_2 can be determined in constant time. The assumption that P_1 moves only a small amount relative to P_2 is made to increase the likelihood that the closest feature pair remains the same. This is why the phrase "almost constant time" is used to describe the performance of the algorithm. Of course, it is possible that the closest feature pair will change. In this case, neighboring features are tested

using the condition above until the new closest pair of features is found. In this worst case, this search could be costly, but this violates the assumption that the bodies do not move far between successive collision detection calls.

The 2D ideas extend to 3D convex polyhedra [247, 636, 702]. The primary difference is that three kinds of features are considered: faces, edges, and vertices. The cases become more complicated, but the idea is the same. Once again, the condition regarding mutual Voronoi regions holds, and the resulting incremental collision detection algorithm has "almost constant time" performance.

5.3.4 Checking a Path Segment

Collision detection algorithms determine whether a configuration lies in $\mathcal{C}_{free}$, but motion planning algorithms require that an entire path maps into $\mathcal{C}_{free}$. The interface between the planner and collision detection usually involves validation of a path segment (i.e., a path, but often a short one). This cannot be checked point-by-point because it would require an uncountably infinite number of calls to the collision detection algorithm.

Suppose that a path, $\tau : [0, 1] \rightarrow \mathcal{C}$, needs to be checked to determine whether $\tau([0, 1]) \subset \mathcal{C}_{free}$. A common approach is to sample the interval $[0, 1]$ and call the collision checker only on the samples. What resolution of sampling is required? How can one ever guarantee that the places where the path is not sampled are collision-free? There are both practical and theoretical answers to these questions. In practice, a fixed $\Delta q > 0$ is often chosen as the C-space step size. Points $t_1, t_2 \in [0, 1]$ are then chosen close enough together to ensure that $\rho(\tau(t_1), \tau(t_2)) \leq \Delta q$, in which ρ is the metric on $\mathcal{C}$. The value of Δq is often determined experimentally. If Δq is too small, then considerable time is wasted on collision checking. If Δq is too large, then there is a chance that the robot could jump through a thin obstacle.

Setting Δq empirically might not seem satisfying. Fortunately, there are sound algorithmic ways to verify that a path is collision-free. In some applications the methods are still not used because they are trickier to implement and they often yield worse performance. Therefore, both methods are presented here, and you can decide which is appropriate, depending on the context and your personal tastes.

Ensuring that $\tau([0, 1]) \subset \mathcal{C}_{free}$ requires the use of both distance information and bounds on the distance that points on $\mathcal{A}$ can travel in $\mathbb{R}$. Such bounds can be obtained by using the robot displacement metric from Example 5.6. Before expressing the general case, first we will explain the concept in terms of a rigid robot that translates and rotates in $\mathcal{W} = \mathbb{R}^2$. Let $x_t, y_t \in \mathbb{R}^2$ and $\theta \in [0, 2\pi]/\sim$. Suppose that a collision detection algorithm indicates that $\mathcal{A}(q)$ is at least d units away from collision with obstacles in $\mathcal{W}$. This information can be used to determine a region in $\mathcal{C}_{free}$ that contains q. Suppose that the next candidate configuration to be checked along τ is q'. If no point on $\mathcal{A}$ travels more than distance d when moving from q to q' along τ, then q' and all configurations between q and q' must be collision-free. This assumes that on the path from q to q', every visited configuration must lie between q_i and q'_i for the ith coordinate and any i from 1 to n.

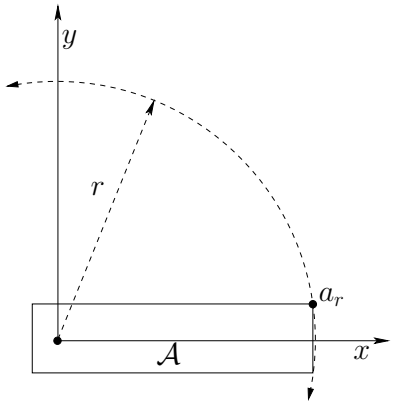


Figure 5.12: The furthest point on $\mathcal{A}$ from the origin travels the fastest when $\mathcal{A}$ is rotated. At most it can be displaced by $2\pi r$, if x_t and y_t are fixed.

If the robot can instead take any path between q and q', then no such guarantee can be made).

When $\mathcal{A}$ undergoes a translation, all points move the same distance. For rotation, however, the distance traveled depends on how far the point on $\mathcal{A}$ is from the rotation center, $(0, 0)$. Let $a_r = (x_r, y_r)$ denote the point on $\mathcal{A}$ that has the largest magnitude, $r = \sqrt{x_r^2 + y_r^2}$. Figure 5.12 shows an example. A transformed point $a \in \mathcal{A}$ may be denoted by $a(x_t, y_t, \theta)$. The following bound is obtained for any $a \in \mathcal{A}$, if the robot is rotated from orientation θ to θ':

$$\|a(x_t, y_t, \theta) - a(x_t, y_t, \theta')\| \leq \|a_r(x_t, y_t, \theta) - a_r(x_t, y_t, \theta')\| < r|\theta - \theta'|, \qquad (5.30)$$

assuming that a path in $\mathcal{C}$ is followed that interpolates between θ and θ' (using the shortest path in $\mathbb{S}^1$ between θ and θ'). Thus, if $\mathcal{A}(q)$ is at least d away from the obstacles, then the orientation may be changed without causing collision as long as $r|\theta - \theta'| < d$. Note that this is a loose upper bound because a_r travels along a circular arc and can be displaced by no more than $2\pi r$.

Similarly, x_t and y_t may individually vary up to d, yielding $|x_t - x_t'| < d$ and $|y_t - y_t'| < d$. If all three parameters vary simultaneously, then a region in $\mathcal{C}_{free}$ can be defined as

$$\{(x_t', y_t', \theta') \in \mathcal{C} \mid |x_t - x_t'| + |y_t - y_t'| + r|\theta - \theta'| < d\}. \qquad (5.31)$$

Such bounds can generally be used to set a step size, Δq, for collision checking that guarantees the intermediate points lie in $\mathcal{C}_{free}$. The particular value used may vary depending on d and the direction[8] of the path.

For the case of $SO(3)$, once again the displacement of the point on $\mathcal{A}$ that has the largest magnitude can be bounded. It is best in this case to express the bounds in terms of quaternion differences, $\|h - h'\|$. Euler angles may also be used

[8]To formally talk about directions, it would be better to define a differentiable structure on $\mathcal{C}$. This will be deferred to Section 8.3, where it seems unavoidable.

to obtain a straightforward generalization of (5.31) that has six terms, three for translation and three for rotation. For each of the three rotation parts, a point with the largest magnitude in the plane perpendicular to the rotation axis must be chosen.

If there are multiple links, it becomes much more complicated to determine the step size. Each point $a \in \mathcal{A}_i$ is transformed by some nonlinear function based on the kinematic expressions from Sections 3.3 and 3.4. Let $a : \mathcal{C} \rightarrow \mathcal{W}$ denote this transformation. In some cases, it might be possible to derive a *Lipschitz condition* of the form

$$\|a(q) - a(q')\| < c\|q - q'\|, \tag{5.32}$$

in which $c \in (0, \infty)$ is a fixed constant, a is any point on $\mathcal{A}_i$, and the expression holds for any $q, q' \in \mathcal{C}$. The goal is to make the *Lipschitz constant*, c, as small as possible; this enables larger variations in q.

A better method is to individually bound the link displacement with respect to each parameter,

$$\|a(q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n) - a(q_1, \ldots, q_{i-1}, q'_i, q_{i+1}, \ldots, q_n)\| < c_i|q_i - q'_i|, \tag{5.33}$$

to obtain the Lipschitz constants c_1, ..., c_n. The bound on robot displacement becomes

$$\|a(q) - a(q')\| < \sum_{i=1}^{n} c_i|q_i - q'_i|. \tag{5.34}$$

The benefit of using individual parameter bounds can be seen by considering a long chain. Consider a 50-link chain of line segments in $\mathbb{R}^2$, and each link has length 10. The C-space is $\mathbb{T}^{50}$, which can be parameterized as $[0, 2\pi]^{50}/\sim$. Suppose that the chain is in a straight-line configuration ($\theta_i = 0$ for all $1 \leq i \leq 50$), which means that the last point is at $(500, 0) \in \mathcal{W}$. Changes in θ_1, the orientation of the first link, dramatically move $\mathcal{A}_{50}$. However, changes in θ_{50} move $\mathcal{A}_{50}$ a smaller amount. Therefore, it is advantageous to pick a different Δq_i for each $1 \leq i \leq 50$. In this example, a smaller value should be used for $\Delta\theta_1$ in comparison to $\Delta\theta_{50}$.

Unfortunately, there are more complications. Suppose the 50-link chain is in a configuration that folds all of the links on top of each other ($\theta_i = \pi$ for each $1 \leq i \leq n$). In this case, $\mathcal{A}_{50}$ does not move as fast when θ_1 is perturbed, in comparison to the straight-line configuration. A larger step size for θ_1 could be used for this configuration, relative to other parts of $\mathcal{C}$. The implication is that, although Lipschitz constants can be made to hold over all of $\mathcal{C}$, it might be preferable to determine a better bound in a local region around $q \in \mathcal{C}$. A linear method could be obtained by analyzing the Jacobian of the transformations, such as (3.53) and (3.57).

Another important concern when checking a path is the order in which the samples are checked. For simplicity, suppose that Δq is constant and that the path is a constant-speed parameterization. Should the collision checker step along from 0 up to 1? Experimental evidence indicates that it is best to use a recursive binary strategy [379]. This makes no difference if the path is collision-free, but

it often saves time if the path is in collision. This is a kind of sampling problem over $[0, 1]$, which is addressed nicely by the van der Corput sequence, ν. The last column in Figure 5.2 indicates precisely where to check along the path in each step. Initially, $\tau(1)$ is checked. Following this, points from the van der Corput sequence are checked in order: $\tau(0)$, $\tau(1/2)$, $\tau(1/4)$, $\tau(3/4)$, $\tau(1/8)$, ... The process terminates if a collision is found or when the dispersion falls below Δq. If Δq is not constant, then it is possible to skip over some points of ν in regions where the allowable variation in q is larger.

5.4 Incremental Sampling and Searching

5.4.1 The General Framework

The algorithms of Sections 5.4 and 5.5 follow the *single-query model*, which means (q_I, q_G) is given only once per robot and obstacle set. This means that there are no advantages to precomputation, and the sampling-based motion planning problem can be considered as a kind of search. The *multiple-query model*, which favors precomputation, is covered in Section 5.6.

The sampling-based planning algorithms presented in the present section are strikingly similar to the family of search algorithms summarized in Section 2.2.4. The main difference lies in step 3 below, in which applying an action, u, is replaced by generating a path segment, τ_s. Another difference is that the search graph, $\mathcal{G}$, is undirected, with edges that represent paths, as opposed to a directed graph in which edges represent actions. It is possible to make these look similar by defining an action space for motion planning that consists of a collection of paths, but this is avoided here. In the case of motion planning with differential constraints, this will actually be required; see Chapter 14.

Most single-query, sampling-based planning algorithms follow this template:

1. **Initialization:** Let $\mathcal{G}(V, E)$ represent an undirected *search graph*, for which V contains at least one vertex and E contains no edges. Typically, V contains q_I, q_G, or both. In general, other points in $\mathcal{C}_{free}$ may be included.

2. **Vertex Selection Method (VSM):** Choose a vertex $q_{cur} \in V$ for expansion.

3. **Local Planning Method (LPM):** For some $q_{new} \in \mathcal{C}_{free}$ that may or may not be represented by a vertex in V, attempt to construct a path $\tau_s : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_{cur}$ and $\tau(1) = q_{new}$. Using the methods of Section 5.3.4, τ_s must be checked to ensure that it does not cause a collision. If this step fails to produce a collision-free path segment, then go to step 2.

4. **Insert an Edge in the Graph:** Insert τ_s into E, as an edge from q_{cur} to q_{new}. If q_{new} is not already in V, then it is inserted.

5. **Check for a Solution:** Determine whether $\mathcal{G}$ encodes a solution path. As in the discrete case, if there is a single search tree, then this is trivial; otherwise, it can become complicated and expensive.

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

In the present context, $\mathcal{G}$ is a topological graph, as defined in Example 4.6. Each vertex is a configuration and each edge is a path that connects two configurations. In this chapter, it will be simply referred to as a graph when there is no chance of confusion. Some authors refer to such a graph as a *roadmap*; however, we reserve the term roadmap for a graph that contains enough paths to make any motion planning query easily solvable. This case is covered in Section 5.6 and throughout Chapter 6.

A large family of sampling-based algorithms can be described by varying the implementations of steps 2 and 3. Implementations of the other steps may also vary, but this is less important and will be described where appropriate. For convenience, step 2 will be called the vertex selection method (VSM) and step 3 will be called the *local planning method* (LPM). The role of the VSM is similar to that of the priority queue, Q, in Section 2.2.1. The role of the LPM is to compute a collision-free path segment that can be added to the graph. It is called *local* because the path segment is usually simple (e.g., the shortest path) and travels a short distance. It is not *global* in the sense that the LPM does not try to solve the entire planning problem; it is expected that the LPM may often fail to construct path segments.

It will be formalized shortly, but imagine for the time being that any of the search algorithms from Section 2.2 may be applied to motion planning by approximating $\mathcal{C}$ with a high-resolution grid. The resulting problem looks like a multi-dimensional extension of Example 2.1 (the "labyrinth" walls are formed by $\mathcal{C}_{obs}$). For a high-resolution grid in a high-dimensional space, most classical discrete searching algorithms have trouble getting trapped in a local minimum. There could be an astronomical number of configurations that fall within a concavity in $\mathcal{C}_{obs}$ that must be escaped to solve the problem, as shown in Figure 5.13a. Imagine a problem in which the C-space obstacle is a giant "bowl" that can trap the configuration. This figure is drawn in two dimensions, but imagine that the $\mathcal{C}$ has many dimensions, such as six for $SE(3)$ or perhaps dozens for a linkage. If the discrete planning algorithms from Section 2.2 are applied to a high-resolution grid approximation of $\mathcal{C}$, then they will all waste their time filling up the bowl before being able to escape to q_G. The number of grid points in this bowl would typically be on the order of 100^n for an n-dimensional C-space. Therefore, sampling-based motion planning algorithms combine sampling and searching in a way that attempts to overcome this difficulty.

As in the case of discrete search algorithms, there are several classes of algorithms based on the number of search trees.

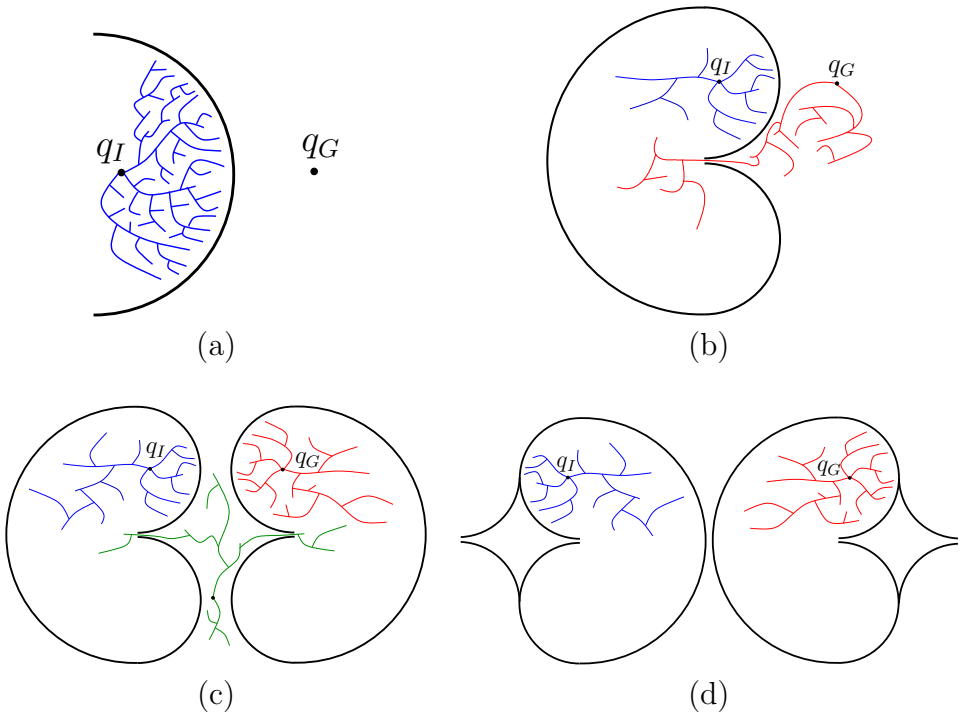


Figure 5.13: All of these depict high-dimensional obstacle regions in C-space. (a) The search must involve some sort of multi-resolution aspect, otherwise, that algorithm may explore too many points within a cavity. (b) Sometimes the problem is like a bug trap, in which case bidirectional search can help. (c) For a double bug trap, multi-directional search may be needed. (d) This example is hard to solve even for multi-directional search.

Unidirectional (single-tree) methods: In this case, the planning appears very similar to discrete forward search, which was given in Figure 2.4. The main difference between algorithms in this category is how they implement the VSM and LPM. Figure 5.13b shows a *bug trap*[9] example for which forward-search algorithms would have great trouble; however, the problem might not be difficult for backward search, if the planner incorporates some kind of greedy, best-first behavior. This example, again in high dimensions, can be considered as a kind of "bug trap." To leave the trap, a path must be found from q_I into the narrow opening. Imagine a fly buzzing around through the high-dimensional trap. The escape opening might not look too difficult in two dimensions, but if it has a small range with respect to each configuration parameter, it is nearly impossible to find the opening. The tip of the "volcano" would be astronomically small compared to the rest of the bug trap. Examples such as this provide some motivation for bidirectional

[9]This principle is actually used in real life to trap flying bugs. This analogy was suggested by James O'Brien in a discussion with James Kuffner.

algorithms. It might be easier for a search tree that starts in q_G to arrive in the bug trap.

Bidirectional (two-tree) methods: Since it is not known whether q_I or q_G might lie in a bug trap (or another challenging region), a bidirectional approach is often preferable. This follows from an intuition that two propagating wavefronts, one centered on q_I and the other on q_G, will meet after covering less area in comparison to a single wavefront centered at q_I that must arrive at q_G. A bidirectional search is achieved by defining the VSM to alternate between trees when selecting vertices. The LPM sometimes generates paths that explore new parts of $\mathcal{C}_{free}$, and at other times it tries to generate a path that connects the two trees.

Multi-directional (more than two trees) methods: If the problem is so bad that a double bug trap exists, as shown in Figure 5.13c, then it might make sense to grow trees from other places in the hopes that there are better chances to enter the traps in the other direction. This complicates the problem of connecting trees, however. Which pairs of trees should be selected in each iteration for possible connection? How often should the same pair be selected? Which vertex pairs should be selected? Many heuristic parameters may arise in practice to answer these questions.

Of course, one can play the devil's advocate and construct the example in Figure 5.13d, for which virtually all sampling-based planning algorithms are doomed. Even harder versions can be made in which a sequence of several narrow corridors must be located and traversed. We must accept the fact that some problems are hopeless to solve using sampling-based planning methods, unless there is some problem-specific structure that can be additionally exploited.

5.4.2 Adapting Discrete Search Algorithms

One of the most convenient and straightforward ways to make sampling-based planning algorithms is to define a grid over $\mathcal{C}$ and conduct a discrete search using the algorithms of Section 2.2. The resulting planning problem actually looks very similar to Example 2.1. Each edge now corresponds to a path in $\mathcal{C}_{free}$. Some edges may not exist because of collisions, but this will have to be revealed incrementally during the search because an explicit representation of $\mathcal{C}_{obs}$ is too expensive to construct (recall Section 4.3).

Assume that an n-dimensional C-space is represented as a unit cube, $\mathcal{C} = [0,1]^n/\sim$, in which $\sim$ indicates that identifications of the sides of the cube are made to reflect the C-space topology. Representing $\mathcal{C}$ as a unit cube usually requires a reparameterization. For example, an angle $\theta \in [0, 2\pi)$ would be replaced with $\theta/2\pi$ to make the range lie within $[0,1]$. If quaternions are used for $SO(3)$, then the upper half of $\mathbb{S}^3$ must be deformed into $[0,1]^3/\sim$.

Discretization Assume that $\mathcal{C}$ is *discretized* by using the *resolutions* k_1, k_2,..., and k_n, in which each k_i is a positive integer. This allows the resolution to be different for each C-space coordinate. Either a standard grid or a Sukharev grid can be used. Let

$$\Delta q_i = [0 \;\; \cdots \;\; 0 \;\; 1/k_i \;\; 0 \;\; \cdots \;\; 0], \tag{5.35}$$

in which the first $i-1$ components and the last $n-i$ components are 0. A *grid point* is a configuration $q \in \mathcal{C}$ that can be expressed in the form[10]

$$\sum_{i=1}^{n} j_i \Delta q_i, \tag{5.36}$$

in which each $j_i \in \{0, 1, \ldots, k_i\}$. The integers j_1, ..., j_n can be imagined as array indices for the grid. Let the term *boundary grid point* refer to a grid point for which $j_i = 0$ or $j_i = k_i$ for some i. Due to identifications, boundary grid points might have more than one representation using (5.36).

Neighborhoods For each grid point q we need to define the set of nearby grid points for which an edge may be constructed. Special care must be given to defining the neighborhood of a boundary grid point to ensure that identifications and the C-space boundary (if it exists) are respected. If q is not a boundary grid point, then the *1-neighborhood* is defined as

$$N_1(q) = \{q + \Delta q_1, \ldots, q + \Delta q_n, q - \Delta q_1, \ldots, q - \Delta q_n\}. \tag{5.37}$$

For an n-dimensional C-space there at most $2n$ 1-neighbors. In two dimensions, this yields at most four 1-neighbors, which can be thought of as "up," "down," "left," and "right." There are *at most* four because some directions may be blocked by the obstacle region.

A *2-neighborhood* is defined as

$$N_2(q) = \{q \pm \Delta q_i \pm \Delta q_j \;\; | \;\; 1 \leq i, j \leq n, \; i \neq j\} \cup N_1(q). \tag{5.38}$$

Similarly, a *k-neighborhood* can be defined for any positive integer $k \leq n$. For an n-neighborhood, there are at most $3^n - 1$ neighbors; there may be fewer due to boundaries or collisions. The definitions can be easily extended to handle the boundary points.

Obtaining a discrete planning problem Once the grid and neighborhoods have been defined, a discrete planning problem is obtained. Figure 5.14 depicts the process for a problem in which there are nine Sukharev grid points in $[0, 1]^2$. Using 1-neighborhoods, the potential edges in the search graph, $\mathcal{G}(V, E)$, appear in Figure 5.14a. Note that $\mathcal{G}$ is a topological graph, as defined in Example 4.6, because each vertex is a configuration and each edge is a path. If q_I and q_G do not

[10] Alternatively, the general lattice definition in (5.21) could be used.

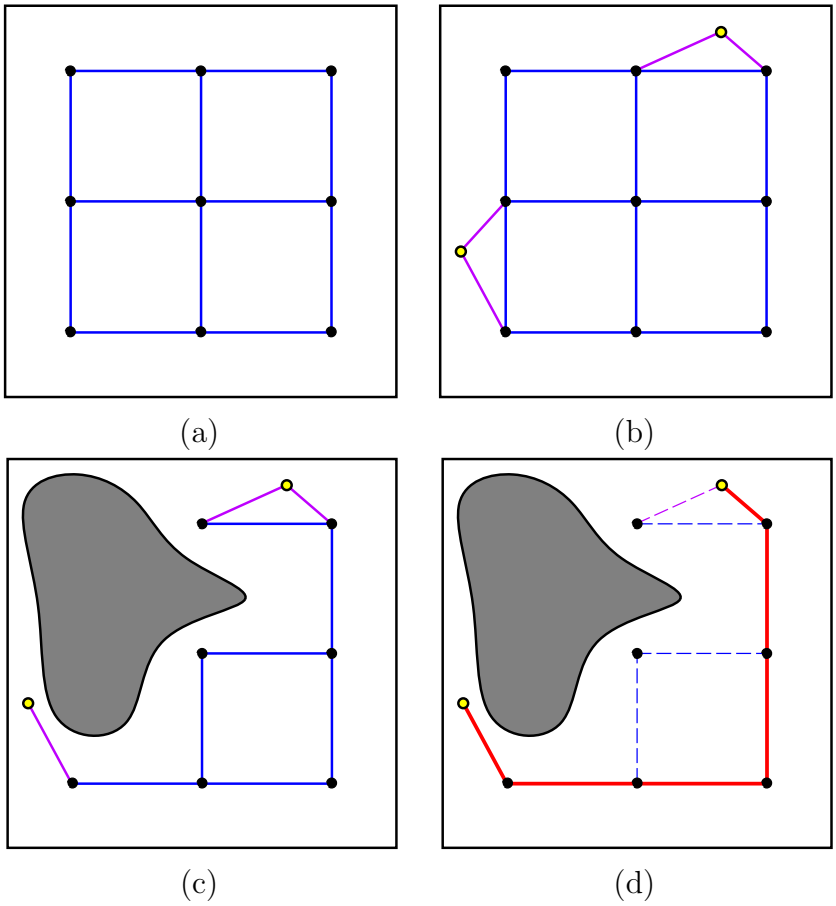


Figure 5.14: A topological graph can be constructed during the search and can successfully solve a motion planning problem using very few samples.

coincide with grid points, they need to be connected to some nearby grid points, as shown in Figure 5.14b. What grid points should q_I and q_G be connected to? As a general rule, if k-neighbors are used, then one should try connecting q_I and q_G to any grid points that are at least as close as the furthest k-neighbor from a typical grid point.

Usually, all of the vertices and edges shown in Figure 5.14b do not appear in $\mathcal{G}$ because some intersect with $\mathcal{C}_{obs}$. Figure 5.14c shows a more typical situation, in which some of the potential vertices and edges are removed because of collisions. This representation could be computed in advance by checking all potential vertices and edges for collision. This would lead to a roadmap, which is suited for multiple queries and is covered in Section 5.6. In this section, it is assumed that $\mathcal{G}$ is revealed "on the fly" during the search. This is the same situation that occurs for the discrete planning methods from Section 2.2. In the current setting, the potential edges of $\mathcal{G}$ are validated during the search. The candidate edges to evaluate are given by the definition of the k-neighborhoods. During the search,

any edge or vertex that has been checked for collision explicitly appears in a data structure so that it does not need to be checked again. At the end of the search, a path is found, as depicted in Figure 5.14d.

Grid resolution issues The method explained so far will nicely find the solution to many problems when provided with the correct resolution. If the number of points per axis is too high, then the search may be too slow. This motivates selecting fewer points per axis, but then solutions might be missed. This trade-off is fundamental to sampling-based motion planning. In a more general setting, if other forms of sampling and neighborhoods are used, then enough samples have to be generated to yield a sufficiently small dispersion.

There are two general ways to avoid having to select this resolution (or more generally, dispersion):

1. Iteratively refine the resolution until a solution is found. In this case, sampling and searching become interleaved. One important variable is how frequently to alternate between the two processes. This will be presented shortly.

2. An alternative is to abandon the adaptation of discrete search algorithms and develop algorithms directly for the continuous problem. This forms the basis of the methods in Sections 5.4.3, 5.4.4, and 5.5.

The most straightforward approach is to iteratively improve the grid resolution. Suppose that initially a standard grid with 2^n points total and 2 points per axis is searched using one of the discrete search algorithms, such as best-first or A^*. If the search fails, what should be done? One possibility is to double the resolution, which yields a grid with 4^n points. Many of the edges can be reused from the first grid; however, the savings diminish rapidly in higher dimensions. Once the resolution is doubled, the search can be applied again. If it fails again, then the resolution can be doubled again to yield 8^n points. In general, there would be a full grid for 2^{ni} points, for each i. The problem is that if n is large, then the rate of growth is too large. For example, if $n = 10$, then there would initially be 1024 points; however, when this fails, the search is not performed again until there are over one million points! If this also fails, then it might take a very long time to reach the next level of resolution, which has 2^{30} points.

A method similar to iterative deepening from Section 2.2.2 would be preferable. Simply discard the efforts of the previous resolution and make grids that have i^n points per axis for each iteration i. This yields grids of sizes 2^n, 3^n, 4^n, and so on, which is much better. The amount of effort involved in searching a larger grid is insignificant compared to the time wasted on lower resolution grids. Therefore, it seems harmless to discard previous work.

A better solution is not to require that a complete grid exists before it can be searched. For example, the resolution can be increased for one axis at a time before attempting to search again. Even better yet may be to tightly interleave

searching and sampling. For example, imagine that the samples appear as an infinite, dense sequence α. The graph can be searched after every 100 points are added, assuming that neighborhoods can be defined or constructed even though the grid is only partially completed. If the search is performed too frequently, then searching would dominate the running time. An easy way make this efficient is to apply the *union-find algorithm* [243, 823] to iteratively keep track of connected components in $\mathcal{G}$ instead of performing explicit searching. If q_I and q_G become part of the same connected component, then a solution path has been found. Every time a new point in the sequence α is added, the "search" is performed in nearly[11] constant time by the union-find algorithm. This is the tightest interleaving of the sampling and searching, and results in a nice sampling-based algorithm that requires no resolution parameter. It is perhaps best to select a sequence α that contains some lattice structure to facilitate the determination of neighborhoods in each iteration.

What if we simply declare the resolution to be outrageously high at the outset? Imagine there are 100^n points in the grid. This places all of the burden on the search algorithm. If the search algorithm itself is good at avoiding local minima and has built-in multi-resolution qualities, then it may perform well without the iterative refinement of the sampling. The method of Section 5.4.3 is based on this idea by performing best-first search on a high-resolution grid, combined with random walks to avoid local minima. The algorithms of Section 5.5 go one step further and search in a multi-resolution way without requiring resolutions and neighborhoods to be explicitly determined. This can be considered as the limiting case as the number of points per axis approaches infinity.

Although this section has focused on grids, it is also possible to use other forms of sampling from Section 5.2. This requires defining the neighborhoods in a suitable way that generalizes the k-neighborhoods of this section. In every case, an infinite, dense sample sequence must be defined to obtain resolution completeness by reducing the dispersion to zero in the limit. Methods for obtaining neighborhoods for irregular sample sets have been developed in the context of sampling-based roadmaps; see Section 5.6. The notion of improving resolution becomes generalized to adding samples that improve dispersion (or even discrepancy).

5.4.3 Randomized Potential Fields

Adapting the discrete algorithms from Section 2.2 works well if the problem can be solved with a small number of points. The number of points per axis must be small or the dimension must be low, to ensure that the number of points, k^n, for k points per axis and n dimensions is small enough so that every vertex in g can be reached in a reasonable amount of time. If, for example, the problem requires 50 points per axis and the dimension is 10, then it is impossible to search all of

[11] It is not constant because the running time is proportional to the inverse Ackerman function, which grows very, very slowly. For all practical purposes, the algorithm operates in constant time. See Section 6.5.2.

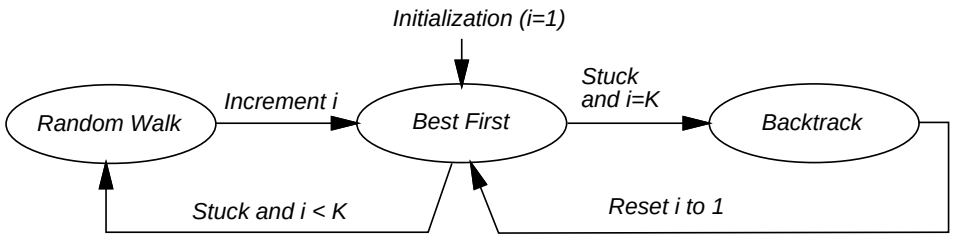


Figure 5.15: The randomized potential field method can be modeled as a three-state machine.

the 50^{10} samples. Planners that exploit best-first heuristics might find the answer without searching most of them; however, for a simple problem such as that shown in Figure 5.13a, the planner will take too long exploring the vertices in the bowl.[12]

The *randomized potential field* [70, 72, 588] approach uses random walks to attempt to escape local minima when best-first search becomes stuck. It was one of the first sampling-based planners that developed specialized techniques beyond classical discrete search, in an attempt to better solve challenging motion planning problems. In many cases, remarkable results were obtained. In its time, the approach was able to solve problems up to 31 degrees of freedom, which was well beyond what had been previously possible. The main drawback, however, was that the method involved many heuristic parameters that had to be adjusted for each problem. This frustration eventually led to the development of better approaches, which are covered in Sections 5.4.4, 5.5, and 5.6. Nevertheless, it is worthwhile to study the clever heuristics involved in this earlier method because they illustrate many interesting issues, and the method was very influential in the development of other sampling-based planning algorithms.[13]

The most complicated part of the algorithm is the definition of a *potential function*, which can be considered as a pseudometric that tries to estimate the distance from any configuration to the goal. In most formulations, there is an *attractive* term, which is a metric on $\mathcal{C}$ that yields the distance to the goal, and a *repulsive* term, which penalizes configurations that come too close to obstacles. The construction of potential functions involves many heuristics and is covered in great detail in [588]. One of the most effective methods involves constructing cost-to-go functions over $\mathcal{W}$ and lifting them to $\mathcal{C}$ [71]. In this section, it will be sufficient to assume that some potential function, $g(q)$, is defined, which is the same notation (and notion) as a cost-to-go function in Section 2.2.2. In this case, however, there is no requirement that $g(q)$ is optimal or even an underestimate of the true cost to go.

When the search becomes stuck and a random walk is needed, it is executed for some number of iterations. Using the discretization procedures of Section 5.4.2, a

[12]Of course, that problem does not appear to need so many points per axis; fewer may be used instead, if the algorithm can adapt the sampling resolution or dispersion.

[13]The exciting results obtained by the method even helped inspire me many years ago to work on motion planning.

high-resolution grid (e.g., 50 points per axis) is initially defined. In each iteration, the current configuration is modified as follows. Each coordinate, q_i, is increased or decreased by Δq_i (the grid step size) based on the outcome of a fair coin toss. Topological identifications must be respected, of course. After each iteration, the new configuration is checked for collision, or whether it exceeds the boundary of $\mathcal{C}$ (if it has a boundary). If so, then it is discarded, and another attempt is made from the previous configuration. The failures can repeat indefinitely until a new configuration in $\mathcal{C}_{free}$ is obtained.

The resulting planner can be described in terms of a three-state machine, which is shown in Figure 5.15. Each state is called a *mode* to avoid confusion with earlier state-space concepts. The VSM and LPM are defined in terms of the mode. Initially, the planner is in the BEST FIRST mode and uses q_I to start a gradient descent. While in the BEST FIRST mode, the VSM selects the newest vertex, $v \in V$. In the first iteration, this is q_I. The LPM creates a new vertex, v_n, in a neighborhood of v, in a direction that minimizes g. The direction sampling may be performed using randomly selected or deterministic samples. Using random samples, the sphere sampling method from Section 5.2.2 can be applied. After some number of tries (another parameter), if the LPM is unsuccessful at reducing g, then the mode is changed to RANDOM WALK because the best-first search is stuck in a local minimum of g.

In the RANDOM WALK mode, a random walk is executed from the newest vertex. The random walk terminates if either g is lowered or a specified limit of iterations is reached. The limit is actually sampled from a predetermined random variable (which contains parameters that also must be selected). When the RANDOM WALK mode terminates, the mode is changed back to BEST FIRST. A counter is incremented to keep track of the number of times that the random walk was attempted. A parameter K determines the maximum number of attempted random walks (a reasonable value is $K = 20$ [71]). If BEST FIRST fails after K random walks have been attempted, then the BACKTRACK mode is entered. The BACKTRACK mode selects a vertex at random from among the vertices in V that were obtained during a random walk. Following this, the counter is reset, and the mode is changed back to BEST FIRST.

Due to the random walks, the resulting paths are often too complicated to be useful in applications. Fortunately, it is straightforward to transform a computed path into a simpler one that is still collision-free. A common approach is to iteratively pick pairs of points at random along the domain of the path and attempt to replace the path segment with a straight-line path (in general, the shortest path in $\mathcal{C}$). For example, suppose $t_1, t_2 \in [0, 1]$ are chosen at random, and $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ is the computed solution path. This path is transformed into a new path,

$$\tau'(t) = \begin{cases} \tau(t) & \text{if } 0 \leq t \leq t_1 \\ a\tau(t_1) + (1-a)\tau(t_2) & \text{if } t_1 \leq t \leq t_2 \\ \tau(t) & \text{if } t_2 \leq t \leq 1, \end{cases} \tag{5.39}$$

in which $a \in [0, 1]$ represents the fraction of the way from t_1 to t_2. Explicitly,

$a = (t_2 - t)/(t_2 - t_1)$. The new path must be checked for collision. If it passes, then it replaces the old path; otherwise, it is discarded and a new pair t_1, t_2, is chosen.

The randomized potential field approach can escape high-dimensional local minima, which allow interesting solutions to be found for many challenging high-dimensional problems. Unfortunately, the heavy amount of parameter tuning caused most people to abandon the method in recent times, in favor of newer methods.

5.4.4 Other Methods

Several influential sampling-based methods are given here. Each of them appears to offer advantages over the randomized potential field method. All of them use randomization, which was perhaps inspired by the potential field method.

Ariadne's Clew algorithm This approach grows a search tree that is biased to explore as much new territory as possible in each iteration [688, 687]. There are two modes, SEARCH and EXPLORE, which alternate over successive iterations. In the EXPLORE mode, the VSM selects a vertex, v_e, at random, and the LPM finds a new configuration that can be easily connected to v_e and is as far as possible from the other vertices in $\mathcal{G}$. A global optimization function that aggregates the distances to other vertices is optimized using a genetic algorithm. In the SEARCH mode, an attempt is made to extend the vertex added in the EXPLORE mode to the goal configuration. The key idea from this approach, which influenced both the next approach and the methods in Section 5.5, is that some of the time must be spent exploring the space, as opposed to focusing on finding the solution. The greedy behavior of the randomized potential field led to some efficiency but was also its downfall for some problems because it was all based on escaping local minima with respect to the goal instead of investing time on global exploration. One disadvantage of Ariadne's Clew algorithm is that it is very difficult to solve the optimization problem for placing a new vertex in the EXPLORE mode. Genetic algorithms were used in [687], which are generally avoided for motion planning because of the required problem-specific parameter tuning.

Expansive-space planner This method [467, 844] generates samples in a way that attempts to explore new parts of the space. In this sense, it is similar to the explore mode of the Ariadne's Clew algorithm. Furthermore, the planner is made more efficient by borrowing the bidirectional search idea from discrete search algorithms, as covered in Section 2.2.3. The VSM selects a vertex, v_e, from $\mathcal{G}$ with a probability that is inversely proportional to the number of other vertices of $\mathcal{G}$ that lie within a predetermined neighborhood of v_e. Thus, "isolated" vertices are more likely to be chosen. The LPM generates a new vertex v_n at random within a predetermined neighborhood of v_e. It will decide to insert v_n into $\mathcal{G}$ with a probability that is inversely proportional to the number of other vertices

of $\mathcal{G}$ that lie within a predetermined neighborhood of v_n. For a fixed number of iterations, the VSM repeatedly chooses the same vertex, until moving on to another vertex. The resulting planner is able to solve many interesting problems by using a surprisingly simple criterion for the placement of points. The main drawbacks are that the planner requires substantial parameter tuning, which is problem-specific (or at least specific to a similar family of problems), and the performance tends to degrade if the query requires systematically searching a long labyrinth. Choosing the radius of the predetermined neighborhoods essentially amounts to determining the appropriate resolution.

Random-walk planner A surprisingly simple and efficient algorithm can be made entirely from random walks [179]. To avoid parameter tuning, the algorithm adjusts its distribution of directions and magnitude in each iteration, based on the success of the past k iterations (perhaps k is the only parameter). In each iteration, the VSM just selects the vertex that was most recently added to $\mathcal{G}$. The LPM generates a direction and magnitude by generating samples from a multivariate Gaussian distribution whose covariance parameters are adaptively tuned. The main drawback of the method is similar to that of the previous method. Both have difficulty traveling through long, winding corridors. It is possible to combine adaptive random walks with other search algorithms, such as the potential field planner [178].

5.5 Rapidly Exploring Dense Trees

This section introduces an incremental sampling and searching approach that yields good performance in practice without any parameter tuning.[14] The idea is to incrementally construct a search tree that gradually improves the resolution but does not need to explicitly set any resolution parameters. In the limit, the tree densely covers the space. Thus, it has properties similar to space filling curves [842], but instead of one long path, there are shorter paths that are organized into a tree. A dense sequence of samples is used as a guide in the incremental construction of the tree. If this sequence is random, the resulting tree is called a *rapidly exploring random tree (RRT)*. In general, this family of trees, whether the sequence is random or deterministic, will be referred to as *rapidly exploring dense trees (RDTs)* to indicate that a dense covering of the space is obtained. This method was originally developed for motion planning under differential constraints [608, 611]; that case is covered in Section 14.4.3.

[14]The original RRT [598] was introduced with a step size parameter, but this is eliminated in the current presentation. For implementation purposes, one might still want to revert to this older way of formulating the algorithm because the implementation is a little easier. This will be discussed shortly.

```
SIMPLE_RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      G.add_vertex(α(i));
4      q_n ← NEAREST(S(G), α(i));
5      G.add_edge(q_n, α(i));
```

Figure 5.16: The basic algorithm for constructing RDTs (which includes RRTs as a special case) when there are no obstacles. It requires the availability of a dense sequence, α, and iteratively connects from $\alpha(i)$ to the nearest point among all those reached by $\mathcal{G}$.

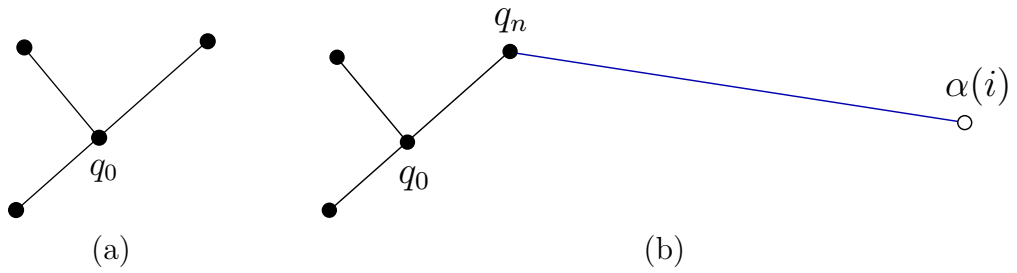


Figure 5.17: (a) Suppose inductively that this tree has been constructed so far using the algorithm in Figure 5.16. (b) A new edge is added that connects from the sample $\alpha(i)$ to the nearest point in S, which is the vertex q_n.

5.5.1 The Exploration Algorithm

Before explaining how to use these trees to solve a planning query, imagine that the goal is to get as close as possible to every configuration, starting from an initial configuration. The method works for any dense sequence. Once again, let α denote an infinite, dense sequence of samples in $\mathcal{C}$. The ith sample is denoted by $\alpha(i)$. This may possibly include a uniform, random sequence, which is only dense with probability one. Random sequences that induce a nonuniform bias are also acceptable, as long as they are dense with probability one.

An RDT is a topological graph, $\mathcal{G}(V, E)$. Let $S \subset \mathcal{C}_{free}$ indicate the set of all points reached by $\mathcal{G}$. Since each $e \in E$ is a path, this can be expressed as the *swath*, S, of the graph, which is defined as

$$S = \bigcup_{e \in E} e([0, 1]). \tag{5.40}$$

In (5.40), $e([0, 1]) \subseteq \mathcal{C}_{free}$ is the image of the path e.

The exploration algorithm is first explained in Figure 5.16 without any obstacles or boundary obstructions. It is assumed that $\mathcal{C}$ is a metric space. Initially, a vertex is made at q_0. For k iterations, a tree is iteratively grown by connecting

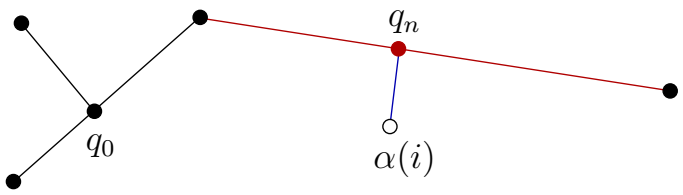
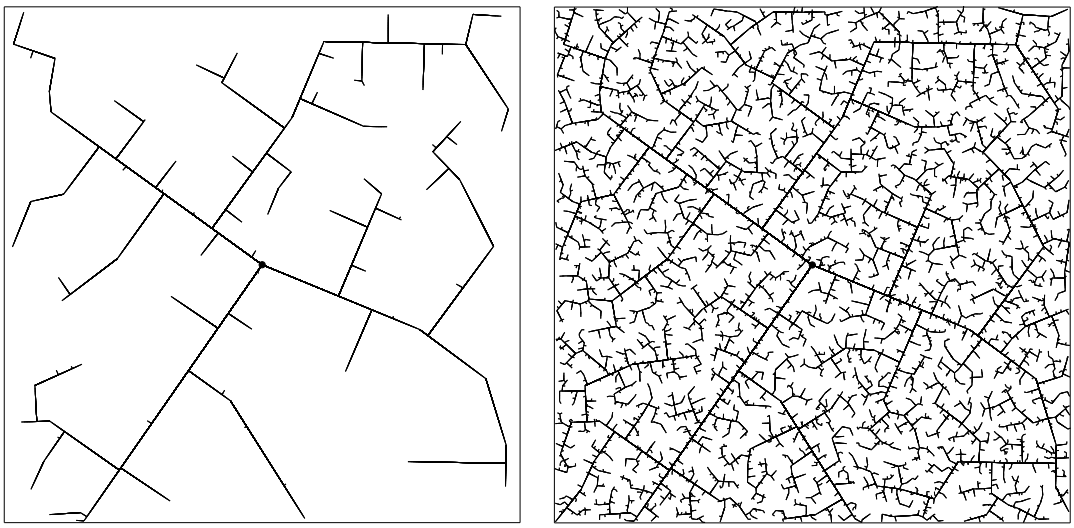


Figure 5.18: If the nearest point in S lies in an edge, then the edge is split into two, and a new vertex is inserted into $\mathcal{G}$.



45 iterations 2345 iterations

Figure 5.19: In the early iterations, the RRT quickly reaches the unexplored parts. However, the RRT is dense in the limit (with probability one), which means that it gets arbitrarily close to any point in the space.

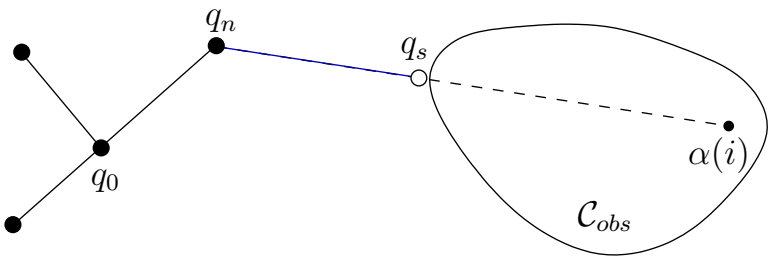


Figure 5.20: If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by the collision detection algorithm.

$\alpha(i)$ to its nearest point in the swath, S. The connection is usually made along the shortest possible path. In every iteration, $\alpha(i)$ becomes a vertex. Therefore, the resulting tree is dense. Figures 5.17–5.18 illustrate an iteration graphically. Suppose the tree has three edges and four vertices, as shown in Figure 5.17a. If the nearest point, $q_n \in S$, to $\alpha(i)$ is a vertex, as shown in Figure 5.17b, then an edge is made from q_n to $\alpha(i)$. However, if the nearest point lies in the interior of an edge, as shown in Figure 5.18, then the existing edge is split so that q_n appears as a new vertex, and an edge is made from q_n to $\alpha(i)$. The edge splitting, if required, is assumed to be handled in line 4 by the method that adds edges. Note that the total number of edges may increase by one or two in each iteration.

The method as described here does not fit precisely under the general framework from Section 5.4.1; however, with the modifications suggested in Section 5.5.2, it can be adapted to fit. In the RDT formulation, the NEAREST function serves the purpose of the VSM, but in the RDT, a point may be selected from anywhere in the swath of the graph. The VSM can be generalized to a *swath-point selection method*, SSM. This generalization will be used in Section 14.3.4. The LPM tries to connect $\alpha(i)$ to q_n along the shortest path possible in $\mathcal{C}$.

Figure 5.19 shows an execution of the algorithm in Figure 5.16 for the case in which $\mathcal{C} = [0,1]^2$ and $q_0 = (1/2, 1/2)$. It exhibits a kind of fractal behavior.[15] Several main branches are first constructed as it rapidly reaches the far corners of the space. Gradually, more and more area is filled in by smaller branches. From the pictures, it is clear that in the limit, the tree densely fills the space. Thus, it can be seen that the tree gradually improves the resolution (or dispersion) as the iterations continue. This behavior turns out to be ideal for sampling-based motion planning.

Recall that in sampling-based motion planning, the obstacle region $\mathcal{C}_{obs}$ is not explicitly represented. Therefore, it must be taken into account in the construction of the tree. Figure 5.20 indicates how to modify the algorithm in Figure 5.16 so that collision checking is taken into account. The modified algorithm appears in Figure 5.21. The procedure STOPPING-CONFIGURATION yields the nearest configuration possible to the boundary of $\mathcal{C}_{free}$, along the direction toward $\alpha(i)$. The nearest

[15] If α is uniform, random, then a *stochastic fractal* [586] is obtained. Deterministic fractals can be constructed using sequences that have appropriate symmetries.

```
RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      q_n ← NEAREST(S, α(i));
4      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
5      if q_s ≠ q_n then
6          G.add_vertex(q_s);
7          G.add_edge(q_n, q_s);
```

Figure 5.21: The RDT with obstacles.

point $q_n \in S$ is defined to be same (obstacles are ignored); however, the new edge might not reach to $\alpha(i)$. In this case, an edge is made from q_n to q_s, the last point possible before hitting the obstacle. How close can the edge come to the obstacle boundary? This depends on the method used to check for collision, as explained in Section 5.3.4. It is sometimes possible that q_n is already as close as possible to the boundary of $\mathcal{C}_{free}$ in the direction of $\alpha(i)$. In this case, no new edge or vertex is added that for that iteration.

5.5.2 Efficiently Finding Nearest Points

There are several interesting alternatives for implementing the NEAREST function in line 3 of the algorithm in Figure 5.16. There are generally two families of methods: *exact* or *approximate*. First consider the exact case.

Exact solutions Suppose that all edges in $\mathcal{G}$ are line segments in $\mathbb{R}^m$ for some dimension $m \geq n$. An edge that is generated early in the construction process will be split many times in later iterations. For the purposes of finding the nearest point in S, however, it is best to handle this as a single segment. For example, see the three large branches that extend from the root in Figure 5.19. As the number of points increases, the benefit of agglomerating the segments increases. Let each of these agglomerated segments be referred to as a *supersegment*. To implement NEAREST, a primitive is needed that computes the distance between a point and a line segment. This can be performed in constant time with simple vector calculus. Using this primitive, NEAREST is implemented by iterating over all supersegments and taking the point with minimum distance among all of them. It may be possible to improve performance by building hierarchical data structures that can eliminate large sets of supersegments, but this remains to be seen experimentally.

In some cases, the edges of $\mathcal{G}$ may not be line segments. For example, the shortest paths between two points in $SO(3)$ are actually circular arcs along $\mathbb{S}^3$. One possible solution is to maintain a separate parameterization of $\mathcal{C}$ for the purposes of computing the NEAREST function. For example, $SO(3)$ can be represented as $[0,1]^3/\sim$, by making the appropriate identifications to obtain $\mathbb{RP}^3$. Straight-

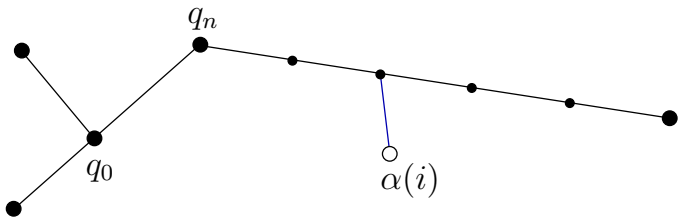


Figure 5.22: For implementation ease, intermediate vertices can be inserted to avoid checking for closest points along line segments. The trade-off is that the number of vertices is increased dramatically.

line segments can then be used. The problem is that the resulting metric is not consistent with the Haar measure, which means that an accidental bias would result. Another option is to tightly enclose $\mathbb{S}^3$ in a 4D cube. Every point on $\mathbb{S}^3$ can be mapped outward onto a cube face. Due to antipodal identification, only four of the eight cube faces need to be used to obtain a bijection between the set of all rotation and the cube surface. Linear interpolation can be used along the cube faces, as long as both points remain on the same face. If the points are on different faces, then two line segments can be used by bending the shortest path around the corner between the two faces. This scheme will result in less distortion than mapping $SO(3)$ to $[0,1]^3/\sim$; however, some distortion will still exist.

Another approach is to avoid distortion altogether and implement primitives that can compute the distance between a point and a curve. In the case of $SO(3)$, a primitive is needed that can find the distance between a circular arc in $\mathbb{R}^m$ and a point in $\mathbb{R}^m$. This might not be too difficult, but if the curves are more complicated, then an exact implementation of the NEAREST function may be too expensive computationally.

Approximate solutions Approximate solutions are much easier to construct, however, a resolution parameter is introduced. Each path segment can be approximated by inserting intermediate vertices along long segments, as shown in Figure 5.22. The intermediate vertices should be added each time a new sample, $\alpha(i)$, is inserted into $\mathcal{G}$. A parameter Δq can be defined, and intermediate samples are inserted to ensure that no two consecutive vertices in $\mathcal{G}$ are ever further than Δq from each other. Using intermediate vertices, the interiors of the edges in $\mathcal{G}$ are ignored when finding the nearest point in S. The approximate computation of NEAREST is performed by finding the closest vertex to $\alpha(i)$ in $\mathcal{G}$. This approach is by far the simplest to implement. It also fits precisely under the incremental sampling and searching framework from Section 5.4.1.

When using intermediate vertices, the trade-offs are clear. The computation time for each evaluation of NEAREST is linear in the number of vertices. Increasing the number of vertices improves the quality of the approximation, but it also dramatically increases the running time. One way to recover some of this cost is to insert the vertices into an efficient data structure for nearest-neighbor searching.

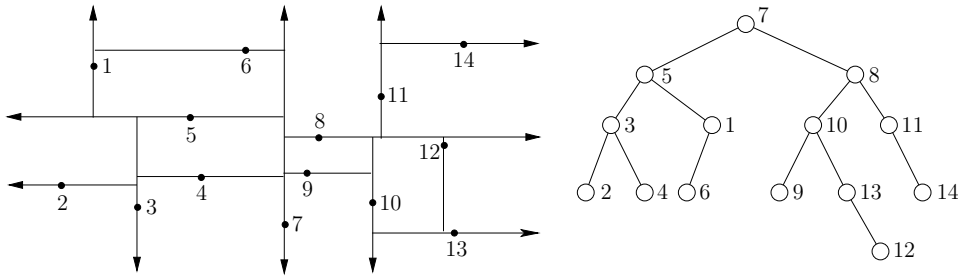


Figure 5.23: A Kd-tree can be used for efficient nearest-neighbor computations.

One of the most practical and widely used data structures is the *Kd-tree* [264, 365, 758]. A depiction is shown in Figure 5.23 for 14 points in $\mathbb{R}^2$. The Kd-tree can be considered as a multi-dimensional generalization of a binary search tree. The Kd-tree is constructed for points, P, in $\mathbb{R}^2$ as follows. Initially, sort the points with respect to the x coordinate. Take the median point, $p \in P$, and divide P into two sets, depending on which side of a vertical line through p the other points fall. For each of the two sides, sort the points by the y coordinate and find their medians. Points are divided at this level based on whether they are above or below horizontal lines. At the next level of recursion, vertical lines are used again, followed by horizontal again, and so on. The same idea can be applied in $\mathbb{R}^n$ by cycling through the n coordinates, instead of alternating between x and y, to form the divisions. In [52], the Kd-tree is extended to topological spaces that arise in motion planning and is shown to yield good performance for RRTs and sampling-based roadmaps. A Kd-tree of k points can be constructed in $O(nk \lg k)$ time. Topological identifications must be carefully considered when traversing the tree. To find the nearest point in the tree to some given point, the query algorithm descends to a leaf vertex whose associated region contains the query point, finds all distances from the data points in this leaf to the query point, and picks the closest one. Next, it recursively visits those surrounding leaf vertices that are further from the query point than the closest point found so far [47, 52]. The nearest point can be found in time logarithmic in k.

Unfortunately, these bounds hide a constant that increases exponentially with the dimension, n. In practice, the Kd-tree is useful in motion planning for problems of up to about 20 dimensions. After this, the performance usually degrades too much. As an empirical rule, if there are more than 2^n points, then the Kd-tree should be more efficient than naive nearest neighbors. In general, the trade-offs must be carefully considered in a particular application to determine whether exact solutions, approximate solutions with naive nearest-neighbor computations, or approximate solutions with Kd-trees will be more efficient. There is also the issue of implementation complexity, which probably has caused most people to prefer the approximate solution with naive nearest-neighbor computations.

5.5.3 Using the Trees for Planning

So far, the discussion has focused on exploring $\mathcal{C}_{free}$, but this does not solve a planning query by itself. RRTs and RDTs can be used in many ways in planning algorithms. For example, they could be used to escape local minima in the randomized potential field planner of Section 5.4.3.

Single-tree search A reasonably efficient planner can be made by directly using the algorithm in Figure 5.21 to grow a tree from q_I and periodically check whether it is possible to connect the RDT to q_G. An easy way to achieve this is to start with a dense sequence α and periodically insert q_G at regularly spaced intervals. For example, every 100th sample could be q_G. Each time this sample is reached, an attempt is made to reach q_G from the closest vertex in the RDT. If the sample sequence is random, which generates an RRT, then the following modification works well. In each iteration, toss a biased coin that has probability 99/100 of being HEADS and 1/100 of being TAILS. If the result is HEADS, then set $\alpha(i)$, to be the next element of the pseudorandom sequence; otherwise, set $\alpha(i) = q_G$. This forces the RRT to occasionally attempt to make a connection to the goal, q_G. Of course, 1/100 is arbitrary, but it is in a range that works well experimentally. If the bias is too strong, then the RRT becomes too greedy like the randomized potential field. If the bias is not strong enough, then there is no incentive to connect the tree to q_G. An alternative is to consider other dense, but not necessarily nonuniform sequences in $\mathcal{C}$. For example, in the case of random sampling, the probability density function could contain a gentle bias towards the goal. Choosing such a bias is a difficult heuristic problem; therefore, such a technique should be used with caution (or avoided altogether).

Balanced, bidirectional search Much better performance can usually be obtained by growing two RDTs, one from q_I and the other from q_G. This is particularly valuable for escaping one of the bug traps, as mentioned in Section 5.4.1. For a grid search, it is straightforward to implement a bidirectional search that ensures that the two trees meet. For the RDT, special considerations must be made to ensure that the two trees will connect while retaining their "rapidly exploring" property. One additional idea is to make sure that the bidirectional search is balanced [560], which ensures that both trees are the same size.

Figure 5.24 gives an outline of the algorithm. The graph $\mathcal{G}$ is decomposed into two trees, denoted by T_a and T_b. Initially, these trees start from q_I and q_G, respectively. After some iterations, T_a and T_b are swapped; therefore, keep in mind that T_a is not always the tree that contains q_I. In each iteration, T_a is grown exactly the same way as in one iteration of the algorithm in Figure 5.16. If a new vertex, q_s, is added to T_a, then an attempt is made in lines 10–12 to extend T_b. Rather than using $\alpha(i)$ to extend T_b, the new vertex q_s of T_a is used. This causes T_b to try to grow toward T_a. If the two connect, which is tested in line 13, then a solution has been found.

```
RDT_BALANCED_BIDIRECTIONAL(q_I, q_G)
 1   T_a.init(q_I); T_b.init(q_G);
 2   for i = 1 to K do
 3       q_n ← NEAREST(S_a, α(i));
 4       q_s ← STOPPING-CONFIGURATION(q_n, α(i));
 5       if q_s ≠ q_n then
 6           T_a.add_vertex(q_s);
 7           T_a.add_edge(q_n, q_s);
 8           q'_n ← NEAREST(S_b, q_s);
 9           q'_s ← STOPPING-CONFIGURATION(q'_n, q_s);
10           if q'_s ≠ q'_n then
11               T_b.add_vertex(q'_s);
12               T_b.add_edge(q'_n, q'_s);
13           if q'_s = q_s then return SOLUTION;
14       if |T_b| > |T_a| then SWAP(T_a, T_b);
15   return FAILURE
```

Figure 5.24: A bidirectional RDT-based planner.

Line 14 represents an important step that balances the search. This is particularly important for a problem such as the bug trap shown in Figure 5.13b or the puzzle shown in Figure 1.2. If one of the trees is having trouble exploring, then it makes sense to focus more energy on it. Therefore, new exploration is always performed for the smaller tree. How is "smaller" defined? A simple criterion is to use the total number of vertices. Another reasonable criterion is to use the total length of all segments in the tree.

An unbalanced bidirectional search can instead be made by forcing the trees to be swapped in every iteration. Once the trees are swapped, then the roles are reversed. For example, after the first swap, T_b is extended in the same way as an integration in Figure 5.16, and if a new vertex q_s is added then an attempt is made to connect T_a to q_s.

One important concern exists when α is deterministic. It might be possible that even though α is dense, when the samples are divided among the trees, each may not receive a dense set. If each uses its own deterministic sequence, then this problem can be avoided. In the case of making a bidirectional RRT planner, the same (pseudo)random sequence can be used for each tree without encountering such troubles.

More than two trees If a dual-tree approach offers advantages over a single tree, then it is natural to ask whether growing three or more RDTs might be even better. This is particularly helpful for problems like the double bug trap in Figure 5.13c. New trees can be grown from parts of $\mathcal{C}$ that are difficult to reach. Controlling the number of trees and determining when to attempt connections

between them is difficult. Some interesting recent work has been done in this direction [82, 918, 919].

These additional trees could be started at arbitrary (possibly random) configurations. As more trees are considered, a complicated decision problem arises. The computation time must be divided between attempting to explore the space and attempting to connect trees to each other. It is also not clear which connections should be attempted. Many research issues remain in the development of this and other RRT-based planners. A limiting case would be to start a new tree from every sample in $\alpha(i)$ and to try to connect nearby trees whenever possible. This approach results in a graph that covers the space in a nice way that is independent of the query. This leads to the main topic of the next section.

5.6 Roadmap Methods for Multiple Queries

Previously, it was assumed that a single initial-goal pair was given to the planning algorithm. Suppose now that numerous initial-goal queries will be given to the algorithm, while keeping the robot model and obstacles fixed. This leads to a *multiple-query* version of the motion planning problem. In this case, it makes sense to invest substantial time to preprocess the models so that future queries can be answered efficiently. The goal is to construct a topological graph called a *roadmap*, which efficiently solves multiple initial-goal queries. Intuitively, the paths on the roadmap should be easy to reach from each of q_I and q_G, and the graph can be quickly searched for a solution. The general framework presented here was mainly introduced in [516] under the name *probabilistic roadmaps (PRMs)*. The probabilistic aspect, however, is not important to the method. Therefore, we call this family of methods *sampling-based roadmaps*. This distinguishes them from *combinatorial roadmaps*, which will appear in Chapter 6.

5.6.1 The Basic Method

Once again, let $\mathcal{G}(V, E)$ represent a topological graph in which V is a set of vertices and E is the set of paths that map into $\mathcal{C}_{free}$. Under the multiple-query philosophy, motion planning is divided into two phases of computation:

Preprocessing Phase: During the preprocessing phase, substantial effort is invested to build $\mathcal{G}$ in a way that is useful for quickly answering future queries. For this reason, it is called a *roadmap*, which in some sense should be accessible from every part of $\mathcal{C}_{free}$.

Query Phase: During the query phase, a pair, q_I and q_G, is given. Each configuration must be connected easily to $\mathcal{G}$ using a local planner. Following this, a discrete search is performed using any of the algorithms in Section 2.2 to obtain a sequence of edges that forms a path from q_I to q_G.

```
BUILD_ROADMAP
 1  G.init(); i ← 0;
 2  while i < N
 3      if α(i) ∈ C_free then
 4          G.add_vertex(α(i)); i ← i + 1;
 5          for each q ∈ NEIGHBORHOOD(α(i),G)
 6              if ((not G.same_component(α(i), q)) and CONNECT(α(i), q)) then
 7                  G.add_edge(α(i), q);
```

Figure 5.25: The basic construction algorithm for sampling-based roadmaps. Note that i is not incremented if $\alpha(i)$ is in collision. This forces i to correctly count the number of vertices in the roadmap.

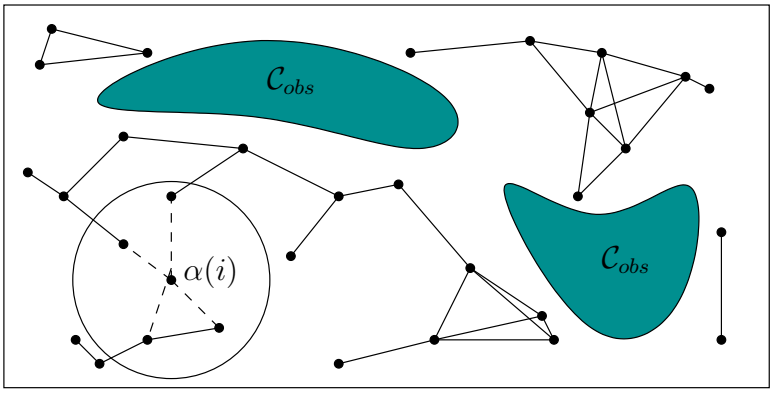


Figure 5.26: The sampling-based roadmap is constructed incrementally by attempting to connect each new sample, $\alpha(i)$, to nearby vertices in the roadmap.

Generic preprocessing phase Figure 5.25 presents an outline of the basic preprocessing phase, and Figure 5.26 illustrates the algorithm. As seen throughout this chapter, the algorithm utilizes a uniform, dense sequence α. In each iteration, the algorithm must check whether $\alpha(i) \in \mathcal{C}_{free}$. If $\alpha(i) \in \mathcal{C}_{obs}$, then it must continue to iterate until a collision-free sample is obtained. Once $\alpha(i) \in \mathcal{C}_{free}$, then in line 4 it is inserted as a vertex of $\mathcal{G}$. The next step is to try to connect $\alpha(i)$ to some nearby vertices, q, of $\mathcal{G}$. Each connection is attempted by the CONNECT function, which is a typical LPM (local planning method) from Section 5.4.1. In most implementations, this simply tests the shortest path between $\alpha(i)$ and q. Experimentally, it seems most efficient to use the multi-resolution, van der Corput–based method described at the end of Section 5.3.4 [379]. Instead of the shortest path, it is possible to use more sophisticated connection methods, such as the bidirectional algorithm in Figure 5.24. If the path is collision-free, then CONNECT returns TRUE.

The same_component condition in line 6 checks to make sure $\alpha(i)$ and q are in different components of $\mathcal{G}$ before wasting time on collision checking. This ensures that every time a connection is made, the number of connected components

of $\mathcal{G}$ is decreased. This can be implemented very efficiently (near constant time) using the previously mentioned *union-find algorithm* [243, 823]. In some implementations this step may be ignored, especially if it is important to generate multiple, alternative solutions. For example, it may be desirable to generate solution paths from different homotopy classes. In this case the condition (**not** $\mathcal{G}$.same_component($\alpha(i), q$)) is replaced with $\mathcal{G}$.vertex_degree(q) $< K$, for some fixed K (e.g., K = 15).

Selecting neighboring samples Several possible implementations of line 5 can be made. In all of these, it seems best to sort the vertices that will be considered for connection in order of increasing distance from $\alpha(i)$. This makes sense because shorter paths are usually less costly to check for collision, and they also have a higher likelihood of being collision-free. If a connection is made, this avoids costly collision checking of longer paths to configurations that would eventually belong to the same connected component.

Several useful implementations of NEIGHBORHOOD are

1. **Nearest K:** The K closest points to $\alpha(i)$ are considered. This requires setting the parameter K (a typical value is 15). If you are unsure which implementation to use, try this one.

2. **Component K:** Try to obtain up to K nearest samples from each connected component of $\mathcal{G}$. A reasonable value is $K = 1$; otherwise, too many connections would be tried.

3. **Radius:** Take all points within a ball of radius r centered at $\alpha(i)$. An upper limit, K, may be set to prevent too many connections from being attempted. Typically, $K = 20$. A radius can be determined adaptively by shrinking the ball as the number of points increases. This reduction can be based on dispersion or discrepancy, if either of these is available for α. Note that if the samples are highly regular (e.g., a grid), then choosing the nearest K and taking points within a ball become essentially equivalent. If the point set is highly irregular, as in the case of random samples, then taking the nearest K seems preferable.

4. **Visibility:** In Section 5.6.2, a variant will be described for which it is worthwhile to try connecting α to all vertices in $\mathcal{G}$.

Note that all of these require $\mathcal{C}$ to be a metric space. One variation that has not yet been given much attention is to ensure that the directions of the NEIGHBORHOOD points relative to $\alpha(i)$ are distributed uniformly. For example, if the 20 closest points are all clumped together in the same direction, then it may be preferable to try connecting to a further point because it is in the opposite direction.

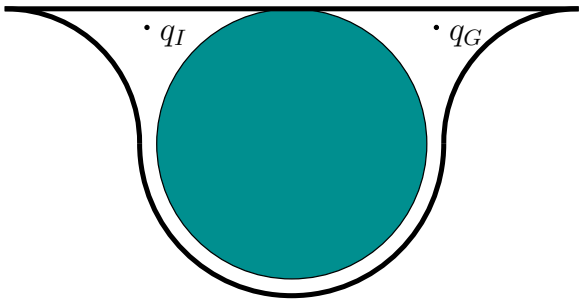


Figure 5.27: An example such as this is difficult for sampling-based roadmaps (in higher dimensional C-spaces) because some samples must fall along many points in the curved tube. Other methods, however, may be able to easily solve it.

Query phase In the query phase, it is assumed that $\mathcal{G}$ is sufficiently complete to answer many queries, each of which gives an initial configuration, q_I, and a goal configuration, q_G. First, the query phase pretends as if q_I and q_G were chosen from α for connection to $\mathcal{G}$. This requires running two more iterations of the algorithm in Figure 5.25. If q_I and q_G are successfully connected to other vertices in $\mathcal{G}$, then a search is performed for a path that connects the vertex q_I to the vertex q_G. The path in the graph corresponds directly to a path in $\mathcal{C}_{free}$, which is a solution to the query. Unfortunately, if this method fails, it cannot be determined conclusively whether a solution exists. If the dispersion is known for a sample sequence, α, then it is at least possible to conclude that no solution exists for the resolution of the planner. In other words, if a solution does exist, it would require the path to travel through a corridor no wider than the radius of the largest empty ball [600].

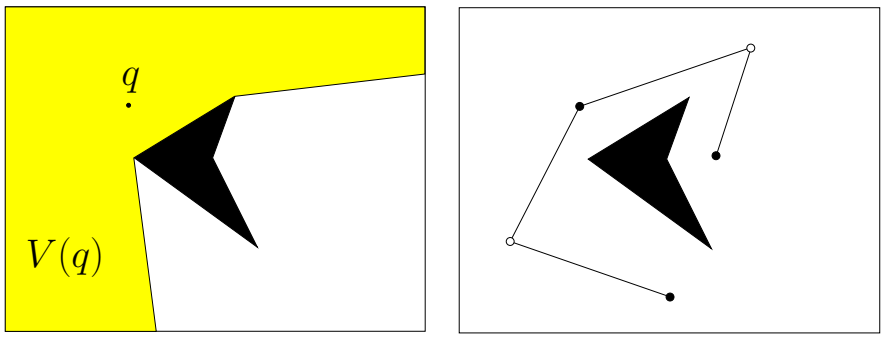
Some analysis There have been many works that analyze the performance of sampling-based roadmaps. The basic idea from one of them [69] is briefly presented here. Consider problems such as the one in Figure 5.27, in which the CONNECT method will mostly likely fail in the thin tube, even though a connection exists. The higher dimensional versions of these problems are even more difficult. Many planning problems involve moving a robot through an area with tight clearance. This generally causes narrow channels to form in $\mathcal{C}_{free}$, which leads to a challenging planning problem for the sampling-based roadmap algorithm. Finding the escape of a bug trap is also challenging, but for the roadmap methods, even traveling through a single corridor is hard (unless more sophisticated LPMs are used [479]).

Let $V(q)$ denote the set of all configurations that can be connected to q using the CONNECT method. Intuitively, this is considered as the set of all configurations that can be "seen" using line-of-sight visibility, as shown in Figure 5.28a

The ϵ-goodness of $\mathcal{C}_{free}$ is defined as

$$\epsilon(\mathcal{C}_{free}) = \min_{q \in \mathcal{C}_{free}} \left\{ \frac{\mu(V(q))}{\mu(\mathcal{C}_{free})} \right\}, \tag{5.41}$$

in which μ represents the measure. Intuitively, $\epsilon(\mathcal{C}_{free})$ represents the small frac-



(a) Visibility definition (b) Visibility roadmap

Figure 5.28: (a) $V(q)$ is the set of points reachable by the LPM from q. (b) A visibility roadmap has two kinds of vertices: guards, which are shown in black, and connectors, shown in white. Guards are not allowed to see other guards. Connectors must see at least two guards.

tion of $\mathcal{C}_{free}$ that is visible from any point. In terms of ϵ and the number of vertices in $\mathcal{G}$, bounds can be established that yield the probability that a solution will be found [69]. The main difficulties are that the ϵ-goodness concept is very conservative (it uses worst-case analysis over all configurations), and ϵ-goodness is defined in terms of the structure of $\mathcal{C}_{free}$, which cannot be computed efficiently. This result and other related results help to gain a better understanding of sampling-based planning, but such bounds are difficult to apply to particular problems to determine whether an algorithm will perform well.

5.6.2 Visibility Roadmap

One of the most useful variations of sampling-based roadmaps is the *visibility roadmap* [885]. The approach works very hard to ensure that the roadmap representation is small yet covers $\mathcal{C}_{free}$ well. The running time is often greater than the basic algorithm in Figure 5.25, but the extra expense is usually worthwhile if the multiple-query philosophy is followed to its fullest extent.

The idea is to define two different kinds of vertices in $\mathcal{G}$:

Guards: To become a *guard*, a vertex, q must not be able to see other guards. Thus, the visibility region, $V(q)$, must be empty of other guards.

Connectors: To become a *connector*, a vertex, q, must see at least two guards. Thus, there exist guards q_1 and q_2, such that $q \in V(q_1) \cap V(q_2)$.

The roadmap construction phase proceeds similarly to the algorithm in Figure 5.25. The *neighborhood function* returns all vertices in $\mathcal{G}$. Therefore, for each new sample $\alpha(i)$, an attempt is made to connect it to every other vertex in $\mathcal{G}$.

The main novelty of the visibility roadmap is using a strong criterion to determine whether to keep $\alpha(i)$ and its associated edges in $\mathcal{G}$. There are three possible cases for each $\alpha(i)$:

1. The new sample, $\alpha(i)$, is not able to connect to any guards. In this case, $\alpha(i)$ earns the privilege of becoming a guard itself and is inserted into $\mathcal{G}$.

2. The new sample can connect to guards from at least two different connected components of $\mathcal{G}$. In this case, it becomes a connector that is inserted into $\mathcal{G}$ along with its associated edges, which connect it to these guards from different components.

3. Neither of the previous two conditions were satisfied. This means that the sample could only connect to guards in the same connected component. In this case, $\alpha(i)$ is discarded.

The final condition causes a dramatic reduction in the number of roadmap vertices.

One problem with this method is that it does not allow guards to be deleted in favor of better guards that might appear later. The placement of guards depends strongly on the order in which samples appear in α. The method may perform poorly if guards are not positioned well early in the sequence. It would be better to have an adaptive scheme in which guards could be reassigned in later iterations as better positions become available. Accomplishing this efficiently remains an open problem. Note the algorithm is still probabilistically complete using random sampling or resolution complete if α is dense, even though many samples are rejected.

5.6.3 Heuristics for Improving Roadmaps

The quest to design a good roadmap through sampling has spawned many heuristic approaches to sampling and making connections in roadmaps. Most of these exploit properties that are specific to the shape of the C-space and/or the particular geometry and kinematics of the robot and obstacles. The emphasis is usually on finding ways to dramatically reduce the number or required samples. Several of these methods are briefly described here.

Vertex enhancement [516] This heuristic strategy focuses effort on vertices that were difficult to connect to other vertices in the roadmap construction algorithm in Figure 5.25. A probability distribution, $P(v)$, is defined over the vertices $v \in V$. A number of iterations are then performed in which a vertex is sampled from V according to $P(v)$, and then some random motions are performed from v to try to reach new configurations. These new configurations are added as vertices, and attempts are made to connect them to other vertices, as selected by the NEIGHBORHOOD function in an ordinary iteration of the algorithm in Figure 5.25. A recommended heuristic [516] for defining $P(v)$ is to define a statistic for each v as $n_f/(n_t + 1)$, in which n_t is the total number of connections attempted for

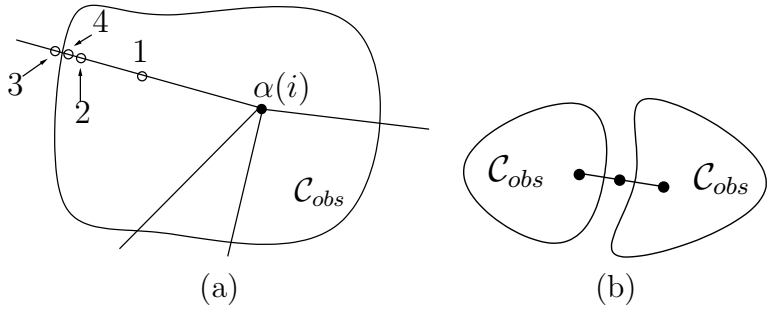


Figure 5.29: (a) To obtain samples along the boundary, binary search is used along random directions from a sample in $\mathcal{C}_{obs}$. (b) The bridge test finds narrow corridors by examining a triple of nearby samples along a line.

v, and n_f is the number of times these attempts failed. The probability $P(v)$ is assigned as $n_f/(n_t+1)m$, in which m is the sum of the statistics over all $v \in V$ (this normalizes the statistics to obtain a valid probability distribution).

Sampling on the $\mathcal{C}_{free}$ boundary [22, 26] This scheme is based on the intuition that it is sometimes better to sample along the boundary, $\partial\mathcal{C}_{free}$, rather than waste samples on large areas of $\mathcal{C}_{free}$ that might be free of obstacles. Figure 5.29a shows one way in which this can be implemented. For each sample of $\alpha(i)$ that falls into $\mathcal{C}_{obs}$, a number of random directions are chosen in $\mathcal{C}$; these directions can be sampled using the $\mathbb{S}^n$ sampling method from Section 5.2.2. For each direction, a binary search is performed to get a sample in $\mathcal{C}_{free}$ that is as close as possible to $\mathcal{C}_{obs}$. The order of point evaluation in the binary search is shown in Figure 5.29a. Let $\tau : [0,1]$ denote the path for which $\tau(0) \in \mathcal{C}_{obs}$ and $\tau(1) \in \mathcal{C}_{free}$. In the first step, test the midpoint, $\tau(1/2)$. If $\tau(1/2) \in \mathcal{C}_{free}$, this means that $\partial\mathcal{C}_{free}$ lies between $\tau(0)$ and $\tau(1/2)$; otherwise, it lies between $\tau(1/2)$ and $\tau(1)$. The next iteration selects the midpoint of the path segment that contains $\partial\mathcal{C}_{free}$. This will be either $\tau(1/4)$ or $\tau(3/4)$. The process continues recursively until the desired resolution is obtained.

Gaussian sampling [132] The Gaussian sampling strategy follows some of the same motivation for sampling on the boundary. In this case, the goal is to obtain points near $\partial\mathcal{C}_{free}$ by using a Gaussian distribution that biases the samples to be closer to $\partial\mathcal{C}_{free}$, but the bias is gentler, as prescribed by the variance parameter of the Gaussian. The samples are generated as follows. Generate one sample, $q_1 \in \mathcal{C}$, uniformly at random. Following this, generate another sample, $q_2 \in \mathcal{C}$, according to a Gaussian with mean q_1; the distribution must be adapted for any topological identifications and/or boundaries of $\mathcal{C}$. If one of q_1 or q_2 lies in $\mathcal{C}_{free}$ and the other lies in $\mathcal{C}_{obs}$, then the one that lies in $\mathcal{C}_{free}$ is kept as a vertex in the roadmap. For some examples, this dramatically prunes the number of required vertices.

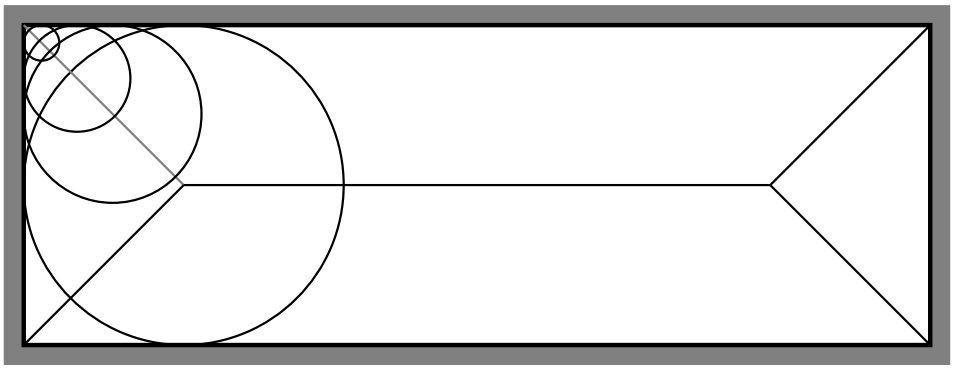


Figure 5.30: The medial axis is traced out by the centers of the largest inscribed balls. The five line segments inside of the rectangle correspond to the medial axis.

Bridge-test sampling [465] The Gaussian sampling strategy decides to keep a point based in part on testing a pair of samples. This idea can be carried one step further to obtain a *bridge test*, which uses three samples along a line segment. If the samples are arranged as shown in Figure 5.29b, then the middle sample becomes a roadmap vertex. This is based on the intuition that narrow corridors are thin in at least one direction. The bridge test indicates that a point lies in a thin corridor, which is often an important place to locate a vertex.

Medial-axis sampling [455, 635, 971] Rather than trying to sample close to the boundary, another strategy is to force the samples to be as far from the boundary as possible. Let (X, ρ) be a metric space. Let a *maximal ball* be a ball $B(x, r) \subseteq X$ such that no other ball can be a proper subset. The centers of all maximal balls trace out a one-dimensional set of points referred to as the *medial axis*. A simple example of a medial axis is shown for a rectangular subset of $\mathbb{R}^2$ in Figure 5.30. The medial axis in $\mathcal{C}_{free}$ is based on the largest balls that can be inscribed in $\mathrm{cl}(\mathcal{C}_{free})$. Sampling on the medial axis is generally difficult, especially because the representation of $\mathcal{C}_{free}$ is implicit. Distance information from collision checking can be used to start with a sample, $\alpha(i)$, and iteratively perturb it to increase its distance from $\partial\mathcal{C}_{free}$ [635, 971]. Sampling on the medial axis of $W \setminus \mathcal{O}$ has also been proposed [455]. In this case, the medial axis in $\mathcal{W} \setminus \mathcal{O}$ is easier to compute, and it can be used to heuristically guide the placement of good roadmap vertices in $\mathcal{C}_{free}$.

Further Reading

Unlike the last two chapters, the material of Chapter 5 is a synthesis of very recent research results. Some aspects of sampling-based motion planning are still evolving. Early approaches include [70, 144, 193, 280, 282, 329, 330, 658, 760]. The Gilbert-Johnson-Keerthi algorithm [388] is an early collision detection approach that helped inspire sampling-based motion planning; see [472] and [588] for many early references.

In much of the early work, randomization appeared to be the main selling point; however, more recently it has been understood that deterministic sampling can work at least as well while obtaining resolution completeness. For a more recent survey of sampling-based motion planning, see [640].

Section 5.1 is based on material from basic mathematics books. For a summary of basic theorems and numerous examples of metric spaces, see [696]. More material appears in basic point-set topology books (e.g., [451, 496]) and analysis books (e.g., [346]). Metric issues in the context of sampling-based motion planning are discussed in [21, 609]. Measure theory is most often introduced in the context of real analysis [346, 425, 546, 836, 837]. More material on Haar measure appears in [425].

Section 5.2 is mainly inspired by literature on Monte Carlo and quasi–Monte Carlo methods for numerical integration and optimization. An excellent source of material is [738]. Other important references for further reading include [191, 540, 682, 937, 938]. Sampling issues in the context of motion planning are considered in [380, 559, 600, 639, 987]. Comprehensive introductions to pure Monte Carlo algorithms appear in [341, 502]. The original source for the Monte Carlo method is [695]. For a survey on algorithms that compute Voronoi diagrams, see [54].

For further reading on collision detection (beyond Section 5.3), see the surveys in [488, 637, 638, 703]. Hierarchical collision detection is covered in [406, 638, 702]. The incremental collision detection ideas in Section 5.3.3 were inspired by the algorithm [636] and V-Clip [247, 702]. Distance computation is covered in [167, 306, 387, 406, 413, 702, 807]. A method suited for detecting self-collisions of linkages appears in [653]. A combinatorial approach to collision detection for motion planning appears in [855]. Numerous collision detection packages are available for use in motion planning research. One of the most widely used is PQP because it works well for any mess of 3D triangles [948].

The incremental sampling and searching framework was synthesized by unifying ideas from many planning methods. Some of these include grid-based search [71, 548, 620] and probabilistic roadmaps (PRMs) [516]. Although the PRM was developed for multiple queries, the single-query version developed in [125] helped shed light on the connection to earlier planning methods. This even led to grid-based variants of PRMs [123, 600]. Another single-query variant is presented in [845].

RDTs were developed in the literature mainly as RRTs, and were introduced in [598, 610]. RRTs have been used in several applications, and many variants have been developed [82, 138, 150, 200, 224, 244, 265, 362, 393, 495, 499, 498, 528, 631, 641, 642, 918, 919, 949, 979, 986]. Originally, they were developed for planning under differential constraints, but most of their applications to date have been for ordinary motion planning. For more information on efficient nearest-neighbor searching, see the recent survey [475], and [46, 47, 48, 52, 99, 230, 365, 476, 538, 758, 908, 989].

Section 5.6 is based mainly on the PRM framework [516]. The "probabilistic" part is not critical to the method; thus, it was referred to here as a *sampling-based roadmap*. A related precursor to the PRM was proposed in [390, 391]. The PRM has been widely used in practice, and many variants have been proposed [1, 23, 61, 62, 125, 161, 181, 244, 479, 544, 600, 627, 628, 740, 784, 792, 885, 900, 950, 971, 979, 995]. An experimental comparison of many of these variants appears in [380]. Some analysis of PRMs appears in [69, 467, 573]. In some works, the term PRM has been applied to virtually any

sampling-based planning algorithm (e.g., [467]); however, in recent years the term has been used more consistently with its original meaning in [516].

Many other methods and issues fall outside of the scope of this chapter. Several interesting methods based on *approximate cell decomposition* [144, 328, 646, 658] can be considered as a form of sampling-based motion planning. A sampling-based method of developing global potential functions appears in [124]. Other sampling-based planning algorithms appear in [194, 348, 417, 418, 463]. The algorithms of this chapter are generally unable to guarantee that a solution does not exist for a motion planning problem. It is possible, however, to use sampling-based techniques to establish in finite time that no solution exists [75]. Such a result is called a *disconnection proof*. Parallelization issues have also been investigated in the context of sampling-based motion planning [82, 177, 183, 257, 795].

Exercises

1. Prove that the Cartesian product of a metric space is a metric space by taking a linear combination as in (5.4).

2. Prove or disprove: If ρ is a metric, then ρ^2 is a metric.

3. Determine whether the following function is a metric on any topological space: X: $\rho(x, x') = 1$ is $x \neq x'$; otherwise, $\rho(x, x') = 0$.

4. State and prove whether or not (5.28) yields a metric space on $\mathcal{C} = SE(3)$, assuming that the two sets are rigid bodies.

5. The dispersion definition given in (5.19) is based on the worst case. Consider defining the *average dispersion*:

$$\bar{\delta}(P) = \frac{1}{\mu(X)} \int_X \min_{p \in P} \{\rho(x, p)\} dx. \tag{5.42}$$

   Describe a Monte Carlo (randomized) method to approximately evaluate (5.42).

6. Determine the average dispersion (as a function of i) for the van der Corput sequence (base 2) on $[0, 1]/\sim$.

7. Show that using the Lebesgue measure on $\mathbb{S}^3$ (spreading mass around uniformly on $\mathbb{S}^3$) yields the Haar measure for $SO(3)$.

8. Is the Haar measure useful in selecting an appropriate C-space metric? Explain.

9. Determine an expression for the (worst-case) dispersion of the ith sample in the base-p (Figure 5.2 shows base-2) van der Corput sequence in $[0, 1]/\sim$, in which 0 and 1 are identified.

10. Determine the dispersion of the following sequence on $[0, 1]$. The first point is $\alpha(1) = 1$. For each $i > 1$, let $c_i = \ln(2i - 3)/\ln 4$ and $\alpha(i) = c_i - \lfloor c_i \rfloor$. It turns out that this sequence achieves the best asymptotic dispersion possible, even in terms of the preceding constant. Also, the points are not uniformly distributed. Can you explain why this happens? [It may be helpful to plot the points in the sequence.]

11. Prove that (5.20) holds.

12. Prove that (5.23) holds.

13. Show that for any given set of points in $[0,1]^n$, a range space $\mathcal{R}$ can be designed so that the discrepancy is as close as desired to 1.

14. Suppose $\mathcal{A}$ is a rigid body in $\mathbb{R}^3$ with a fixed orientation specified by a quaternion, h. Suppose that h is perturbed a small amount to obtain another quaternion, h' (no translation occurs). Construct a good upper bound on distance traveled by points on $\mathcal{A}$, expressed in terms of the change in the quaternion.

15. Design combinations of robots and obstacles in $\mathcal{W}$ that lead to C-space obstacles resembling bug traps.

16. How many k-neighbors can there be at most in an n-dimensional grid with $1 \leq k \leq n$?

17. In a high-dimensional grid, it becomes too costly to consider all $3^n - 1$ n-neighbors. It might not be enough to consider only $2n$ 1-neighbors. Determine a scheme for selecting neighbors that are spatially distributed in a good way, but without requiring too many. For example, what is a good way to select 50 neighbors for a grid in $\mathbb{R}^{10}$?

18. Explain the difference between searching an implicit, high-resolution grid and growing search trees directly on the C-space without a grid.

19. Improve the bound in (5.31) by considering the fact that rotating points trace out a circle, instead of a straight line.

20. (Open problem) Prove there are $n+1$ main branches for an RRT starting from the center of an "infinite" n-dimensional ball in $\mathbb{R}^n$. The directions of the branches align with the vertices of a regular simplex centered at the initial configuration.

Implementations

21. Implement 2D incremental collision checking for convex polygons to obtain "near constant time" performance.

22. Implement the sampling-based roadmap approach. Select an appropriate family of motion planning problems: 2D rigid bodies, 2D chains of bodies, 3D rigid bodies, etc.

    (a) Compare the roadmaps obtained using visibility-based sampling to those obtained for the ordinary sampling method.

    (b) Study the sensitivity of the method with respect to the particular NEIGHBORHOOD method.

    (c) Compare random and deterministic sampling methods.

(d) Use the bridge test to attempt to produce better samples.

23. Implement the balanced, bidirectional RRT planning algorithm.

    (a) Study the effect of varying the amount of intermediate vertices created along edges.

    (b) Try connecting to the random sample using more powerful descent functions.

    (c) Explore the performance gains from using Kd-trees to select nearest neighbors.

24. Make an RRT-based planning algorithm that uses more than two trees. Carefully resolve issues such as the maximum number of allowable trees, when to start a tree, and when to attempt connections between trees.

25. Implement both the expansive-space planner and the RRT, and conduct comparative experiments on planning problems. For the full set of problems, keep the algorithm parameters fixed.

26. Implement a sampling-based algorithm that computes collision-free paths for a rigid robot that can translate or rotate on any of the flat 2D manifolds shown in Figure 4.5.