Motion Planning: The Essentials

Steven M. LaValle

This is the first installment of a two-part tutorial. The goal of the first part is to give the reader a basic understanding of the technical issues and types of approaches to solving the basic path planning or obstacle avoidance problem. The second installment will cover more advanced issues, including feedback, differential constraints, and uncertainty. Note that is a brief *tutorial*, rather than a comprehensive *survey* of methods. For the latter, consult recent textbooks [4], [9].

I. INTRODUCTION

Motion planning involves getting a robot to automatically determine how to move while avoiding collisions with obstacles. Its original formulation, called *The Piano Mover's Problem*, is imagined as determining how to move a complicated piece of furniture through a cluttered house. Have you ever argued about how to move a sofa up a stairwell? It has been clear for several decades that getting robots to reason geometrically about their environments and synthesize such plans is a fundamental difficulty that recurs all over robotics.

The stages of motion planning development are parellel to those of integral calculus: 1) The integration problem was clearly identified and defined; 2) perfect, exact solutions were developed for many classes of functions; 3) since these were limited to a small subset of functions that people care about, numerical integration methods were developed with great success in practice. The similar stages of motion planning were: 1) It clearly defined in the 1970s; 2) the 1980s saw the development of perfect, combinatorial solutions, which are ideal in some settings, but not practical in most; 3) the 1990s brought sampling-based methods, which are not as elegant, but offer practical solutions to modern industrial-grade problems. Over the past decade, motion planning algorithms have been widely used in robotics and automation and have furthermore found application well beyond, including the fields of virtual prototyping and computational biology.

II. PROBLEM FORMULATION

Let $\mathcal{W}$ denote the *world*, which contains a robot and obstacles. For a 2D world, $\mathcal{W} = \mathbb{R}^2$ and $\mathcal{O} \subset \mathcal{W}$ is the *obstacle region*, which has a piecewise-linear (polygonal) boundary. (The complement $\mathcal{W} \setminus \mathcal{O}$ is assumed to be a bounded open set.) The *robot* is a rigid polygon that can move through the world, but must avoid touching the obstacle region. For a 3D world, the only differences are that $\mathcal{W} = \mathbb{R}^3$, and $\mathcal{O}$ and the robot are defined with polyhedra, instead of polygons. Motion planning formulations extend well beyond rigid polygons and polyhedra, but such extensions are left to Section VI and the second part of this tutorial.

S. M. LaValle is with the Department of Computer Science, University of Illinois at Urbana-Champaign lavalle@uiuc.edu

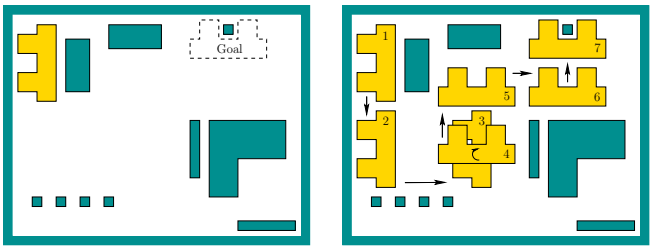

Fig. 1. A 2D example of basic path planning.

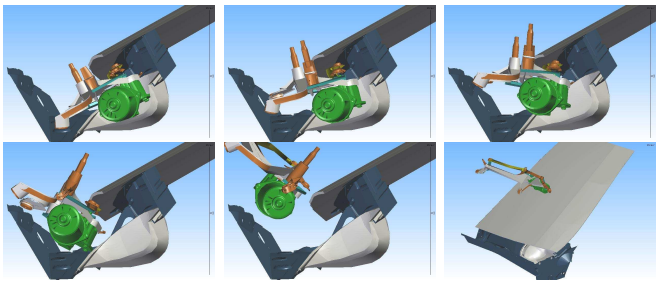

Fig. 2. A 3D automotive assembly task that involves inserting or removing a windshield wiper motor from a car body cavity. This problem was solved for clients using the path planning software of Kineo CAM.

The *basic path planning problem* is informally summarized as: Given an *initial* placement of the robot, compute how to gradually move it into a desired *goal* placement so that it never touches the obstacle region. See Figures 1 and 2 for examples.

Consider the task in terms of algorithm inputs and output.
INPUTS: An initial placement of the robot, a desired goal placement, and a geometric description of the robot and obstacle region.
OUTPUT: A precise description of how to move the robot gradually from its initial placement to the goal placement while never touching the obstacle region.

The output "description" will be a path through the set of all intermediate transformations of the robot, from start to finish.

III. LIVING IN C-SPACE

Although the motion planning problem is described in the world, it really lives in a another space: The set of all rigid-body transformations that can be applied to the robot. This is called the *configuration space* or *C-space*. Finding a solution amounts to computing a path through the part of the C-space that avoids robot-obstacle collisions.

Fig. 3. The first three are manifolds because they locally "look like" $\mathbb{R}^2$; the last two are not because at some points the dimension changes or branching occurs.

A rigid body may translate and rotate. Most people are much more familiar with performing *one* transformation to place a body into a scene, rather than thinking about *all* transformations. The notion of configuration space was the key insight to Lagrangian mechanics of rigid bodies [1], as it allowed dynamics to be expressed using the precise degrees of freedom of a body. The idea was introduced to motion planning by Lozano-Perez [12] and Udupa [17]. The C-space in physics and control theory is usually called a *Lie (pronounced "Lee") group*. In that context, which is much more widely studied than motion planning, the C-space is considered as a *differentiable manifold*, which leads to considerable technical and notational hurdles. The C-space used in motion planning requires no calculus; therefore, it is described as a *topological manifold*, which is fortunately much simpler to define and manipulate. The definition of an n-dimensional (topological) manifold $\mathcal{C}$ is a subset of $\mathbb{R}^m$ for $n \leq m$ such that every $q \in \mathcal{C}$ is contained in at least one open subset of $\mathcal{C}$ (pick a small one!) that is homeomorphic[1] to $\mathbb{R}^n$. The intuition is that in the local vicinity of every q, a manifold behaves like $\mathbb{R}^n$. It is a nicely behaved "surface". The existence of sharp corners does not even matter; however, branching or locally changing dimensions is not allowed. See Figure 3.

We now take a look at C-spaces that commonly arise in planning. Consider a 2D world. Let $\mathcal{A} \subset \mathbb{R}^2$ denote a polygonal robot. It could, for example, be all points inside of a triangle defined by vertices $(-1, 0)$, $(1, 0)$, and $(0, 1)$. We could rotate the robot counterclockwise by any $\theta \in [0, 2\pi)$ and then translate it by any $x_t \in \mathbb{R}$ in the X-direction and any $x_t \in \mathbb{R}$ in the Y direction. This allows for any possible position and orientation, and every x_t, y_t, θ combination leads to a unique robot placement. Let $q = (x_t, y_t, \theta)$ be called the *configuration*. A point $(x, y) \in \mathcal{A}$ would then appear at some $(x', y') \in \mathcal{W}$ (in the world) given by

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \qquad (1)$$

which uses the standard 3 by 3 *homogeneous transformation matrix*. The upper-left, 2-by-2 block is just a rotation matrix.

The set of all configurations $q = (x_t, y_t, \theta)$ is clearly a subset of $\mathbb{R}^3$, but to define the C-space we must take into account that $\theta \pm 2\pi$ yields equivalent rotations. We write that $\mathcal{C} = \mathbb{R}^2 \times S^1$, in which S^1 denotes a circle in the topological sense and accounts for θ (the "circle" is obtained by gluing 0 and π together). The C-space $\mathcal{C}$ is a 3D manifold, and each element is nicely described as $q = (x_t, y_t, \theta)$. Remembering that θ "wraps around" at 2π is crucial to motion planning; otherwise, an artificial barrier or redundant exploration will be introduced. If the robot is not allowed to rotate, then we obtain the *translation-only* case, and $\mathcal{C} = \mathbb{R}^2$ with $q = (x_t, y_t)$.

For the 3D world, the concepts mostly extend as you might expect. Three translation parameters x_t, y_t, z_t appear and a translation-only robot then has C-space $\mathcal{C} = \mathbb{R}^3$ with $q = (x_t, y_t, z_t)$. However, the set of 3D rotations turns out to be 3D manifold all by itself, and it is not as simple as a circle or sphere topologically. The best way to "see" its structure is to use quaternions to represent rotations. Since this a brief tutorial, only the essence is given here, and quaternion algebra is avoided here since it is not critical to motion planning. Every 3D rotation can be expressed as a rotation by an angle $\theta \in [0, 2\pi)$ about *some* fixed axis that passes through the origin. Let this axis be described by some unit vector $v = (v_1, v_2, v_3)$. This already makes it appear that there is a sphere of possible axes, and then a circle of possible angles at each place on the sphere. This collection of circles glued together around the sphere is called the Hopf fibration. Now there is another trouble. Just as 0 and 2π were equivalent in the 2D case, for the 3D case we have that v and θ produce the same rotation as $-v$ and $2\pi - \theta$. A convenient way to handle this is to define $h = (a, b, c, d)$ and assign $a = \cos(\theta/2)$, $b = v_1 \sin(\theta/2)$, $c = v_2 \sin(\theta/2)$, and $d = v_3 \sin(\theta/2)$. Note that $a^2+b^2+c^2+d^2 = 1$, meaning that h lies on a unit sphere. Furthermore, h and $-h$ are equivalent rotations. The C-space for the set of all 3D rotations is therefore nicely "visualized" as a 3-dimensional sphere–a subset of $\mathbb{R}^4$–in which opposite (called *antipodal*) points are "the same". This means that to get the set of all rotations, we can stay in the upper hemisphere ($a \geq 0$), but must be careful at $a = 0$, because opposite points on this equator are "the same". The technical term for the resulting space is *real projective 3-space*, denoted $\mathbb{RP}^3$. For the case of a 3D robot that can translate or rotation, we obtain $\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3$, which is a six-dimensional manifold. We can represent the configuration as $(x_t, y_t, z_t, a, b, c, d)$, while enforcing that $a^2 + b^2 + c^2 + d^2 = 1$. The use of quaternions means that the set of all 3 by 3 rotation matrices is parametrized by a, b, c, and d:

$$\begin{pmatrix} 2(a^2+b^2)-1 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 2(a^2+c^2)-1 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 2(a^2+d^2)-1 \end{pmatrix}. \qquad (2)$$

With different possible parametrizations of rotations, for 2D or 3D worlds, it is important to realize that if two points are "close" under one representation, they might be "far" under another. Furthermore, if there are singularities in the parametrization mapping (e.g., yaw-pitch-roll representation), the C-space might not even represent the same manifold as the set of all rotations.

[1]Homeomorphic means that for the open set, say O, there exists a continuous, bijective function $f : O \to \mathbb{R}^n$, for which the inverse f^{-1} is also continuous.

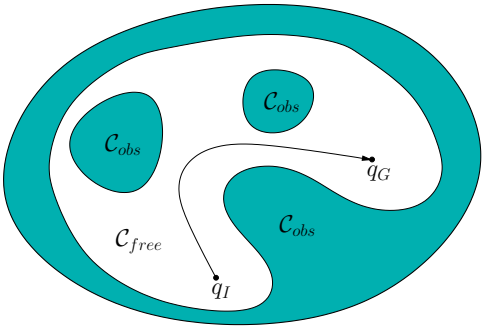

Fig. 4. In the C-space the problem looks simple: Connect q_I to q_G while remaining in $\mathcal{C}_{free}$.

Now that different possibilities for $\mathcal{C}$ have been presented, consider the parts of $\mathcal{C}$ that are prohibited due to collision. Let $\mathcal{A}(q) \subset \mathcal{W}$ denote a closed set of points in the world occupied by the robot $\mathcal{A}$ when it transformed to configuration q. A configuration $q \in \mathcal{C}$ places the robot into *collision* if and only if $\mathcal{A}(q) \cap \mathcal{O} \neq \emptyset$ (the robot and obstacle are attempting to occupy at least one common point in $\mathcal{W}$). The set of all non-colliding configurations is often called the *free space*, and is defined as

$$\mathcal{C}_{free} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} = \emptyset\}. \tag{3}$$

The complement is called the *obstacle region in C-space*: $\mathcal{C}_{obs} = \mathcal{C} \setminus \mathcal{C}_{free}$.

The problem statement of Section II seemed somewhat informal; however, using the C-space, the basic path planning problem can be precisely defined: Given a robot description $\mathcal{A}$, an obstacle description $\mathcal{O}$, a C-space $\mathcal{C}$, an *initial configuration* $q_I \in \mathcal{C}$, and a *goal configuration* q_G, compute a continuous path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ with $\tau(0) = q_I$ and $\tau(1) = q_G$. See Figure 4. A typical way to express τ is as a sequence of line segments, which ignores the particular parameter $s \in [0, 1]$, but is good enough for motion planning results. Note the path must be continuous; otherwise, the robot would appear to "teleport" from one place to another, which is obviously cheating. Gradual motions through $\mathcal{C}$ make the robot move gradually through $\mathcal{W}$.

IV. COMBINATORIAL PLANNING

Although the motion planning problem lives in the *continuous* C-space, computation is *discrete*. Therefore, if we want an algorithmic solution, we need a way to "discretize" the problem. This has led to two main schools of thought: 1) *combinatorial planning*, which thrived in the 1980s, constructs structures in the C-space that discretely and completely capture all information needed to perform planning. 2) *sampling-based planning*, developed mainly across the 1990s, uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the $\mathcal{C}_{free}$ structure. The second approach is most widely used in practice; however, the first one is far superior in many instances. It is therefore worth studying both.

To illustrate the philosophy of combinatorial planning, consider the case in which $\mathcal{W} = \mathbb{R}^2$ and contains a *point robot* ($\mathcal{A} = \{(0, 0)\}$) that cannot rotate. In this case, $\mathcal{C} = \mathbb{R}^2$, and the task is simply to "connect the dots" in the plane with a curve that avoids the obstacles; see Figure 5(a).

Here is a simple technique that contains all the essential ingredients of combinatorial planning. All methods first compute a *roadmap*, which is a graph in which each vertex is a configuration in $\mathcal{C}_{free}$ and each edge is a "simple" path through $\mathcal{C}_{free}$ that connects a pair of vertices. Here is one way to achieve this:

1) Decompose $\mathcal{C}_{free}$ into trapezoids with vertical side segments. Figure 5(b) shows the result. From each polygon vertex, an attempt is made to shoot rays upward and downward. Each ray may be immediately blocked, or it may travel until hitting another part of the obstacle boundary.
2) Place one vertex in the interior of every trapezoid. It doesn't really matter where; for simplicity, pick the centroid.
3) Place one vertex in every vertical segment. The resulting vertices are shown in Figure 5(c).
4) Connect each segment vertex to the two vertices that are in the interior of the neighboring trapezoids. Each connection forms an edge in the graph and corresponds to a straight-line path.

The result is a roadmap that appears to capture the structure of $\mathcal{C}_{free}$. How would you implement these steps? For the first step, we could iterate over each vertex and determine precisely where each upward and downward ray intersects other segments. We could then easily identify the first segment hit by the vertical ray in the above and below directions. For an example as simple as Figure 5(a), this is a fine method. However, if there are n polygonal edges in total and n is large (say, $n = 20,000$), then the method is not efficient because it takes time $O(n^2)$.

By proceeding carefully, this computation can be reduced to time $O(n \lg n)$ by employing the *plane sweep principle* [6], which underlies many decomposition algorithms used for combinatorial planning. First, sort the polygon vertices from left to right, requiring time $O(n \ln n)$. During the algorithm execution, a list of some polygon segments is maintained, sorted from top to bottom as they are stabbed by a vertical line. The method proceeds incrementally from vertex to vertex, traveling from left to right. At each step, the edge list is updated by simple insertions and deletions, which each take $O(\lg n)$ time using self-balancing binary search trees. If the edges incident to the vertex are both to the left, then the two edges are deleted from the list. If they are both to the right, they are inserted into the list (in order). Otherwise, the one to the left is deleted, and the one to the right is inserted. Thanks to this ordering, we can determine in $O(\lg n)$ time the segments directly above and below the vertex, which are first stabbed by upward and downward rays. It is furthermore simple and efficient to incrementally extend the graph as each vertex is processed. For more details, see Section 6.2.2 of [9]

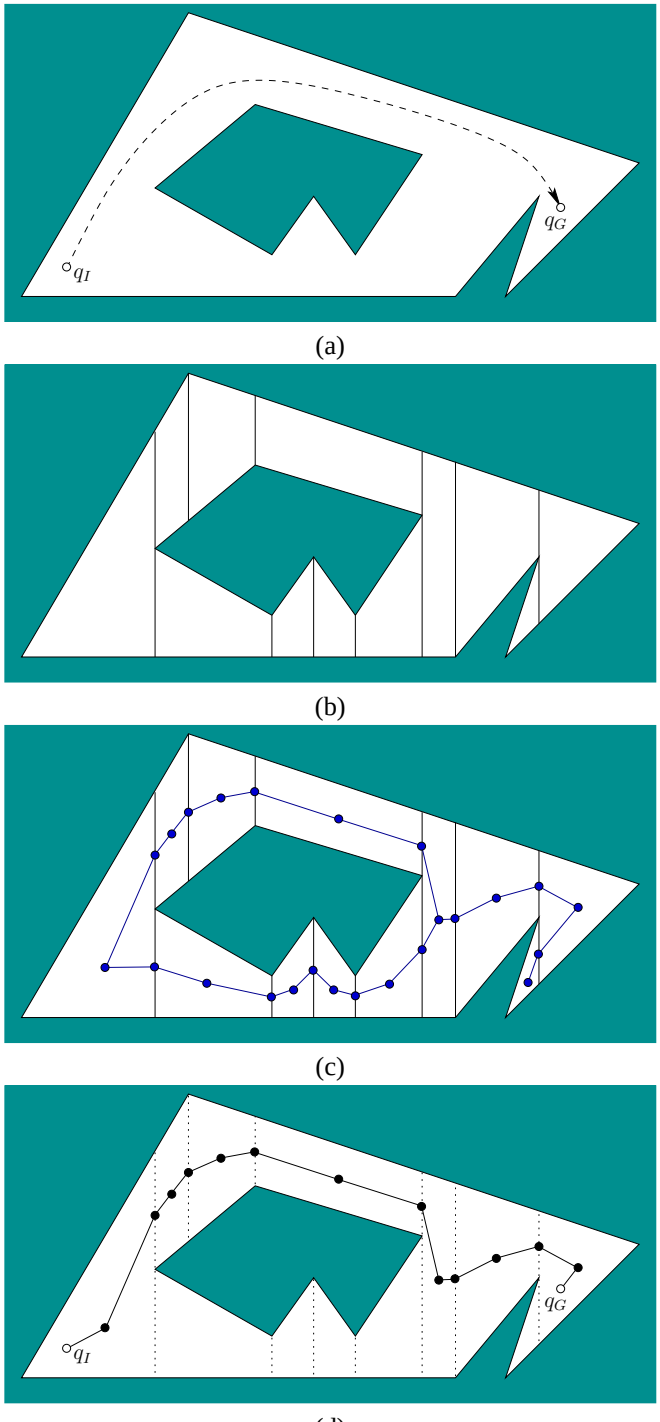

Fig. 5. A combinatorial planning illustration: a) A 2D polygonal obstacle region with proposed q_I and q_G (one possible solution is shown in a dashed path); b) the trapezoidal decomposition; c) constructing a graph by placing a vertex in every vertical edge segment and every trapezoid interior; d) connecting q_I and q_G to the graph and searching for a solution path.

1/ construct C_obs/C_free is very hard —> combinatorial explosion
2/ for a highly complex C_free, cell decomposition to obtain a 'complete' solution —> combinatorial explosion

or Section 6.1 of [6].

The roadmap is constructed without considering the query pair, q_I and q_G. Once the investment is made, the same roadmap can be used for multiple query pairs. In other words, we can easily solve numerous motion planning problems in a world that contains the same obstacles and robot. Here is a simple way to use the computed roadmap from Figure 5:

1) Find the trapezoids that contain q_I and q_G.
2) Connect q_I and q_G to the vertices in their respective trapezoids.
3) Search the graph for a path that connects q_I to q_G.

The first step can be performed trivially in $O(n)$ time by testing whether q_I (or q_G) lies in each trapezoid; this can be shaved down to $O(\lg n)$ time by developing clever hierarchical *point location* data structures [6]. The second step takes constant time, and the final step can be performed in $O(n)$ time using simple graph search algorithms such as breath-first or depth-first.

For the simple case of a point robot in a polygonal world, numerous alternative algorithms exist that yield comparable performance. We could, for example, decompose $\mathcal{C}_{free}$ into triangles instead of trapezoids. The general principles are that each cell should be easy to traverse (convex is ideal), the decomposition into cells should be easily computable, and the adjacencies between cells should be straightforward to determine. Based on these properties, a useful roadmap is obtained.

Roadmaps need not be obtained by cell decompositions. For example, a *shortest path roadmap* yields distance-optimal paths and is constructed by connecting certain pairs of vertices that can "see" each other and each have interior angle greater than π. A *maximum clearance roadmap* can also be computed efficiently. In general, a roadmap is expected to have two properties to be useful for planning: 1) *Accessibility:* It is simple to reach a point on the roadmap from any $q \in \mathcal{C}_{free}$ while trivially avoiding collisions; 2) *Connectivity-preserving:* For any pair q_1, q_2 of points that is connected to the roadmap, a path exists between them in the roadmap if and only if there was a path between q_1 and q_2. In other words, if q_2 is generally reachable from q_1, then traveling between them via the roadmap must also be possible.

It seems up to this point that combinatorial planning solutions have beautiful properties. Most importantly, they construct a *discrete* representation of the problem that *exactly* captures the solution. In other words, there are no "approximation" or "sampling" errors. These methods are called *complete*, meaning that for any input problem, they correctly determine in finite time whether or not a solution exists.

Here comes the trouble: Most motion planning problems involve robots that are not modeled as points and they can rotate in addition to translating. How many of these nice combinatorial planning ideas extend? First consider the case of a polygonal translation-only robot. If the robot $\mathcal{A}$ and obstacle $\mathcal{O}$ are convex polygons, then $\mathcal{C}_{obs}$ is a polygon in which every edge corresponds to a point-to-edge contact

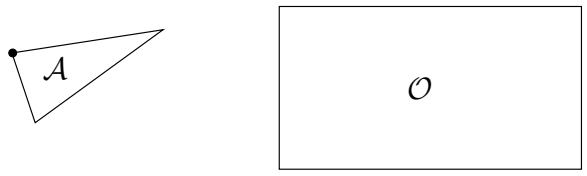

Fig. 6. A triangular robot and a rectangular obstacle.

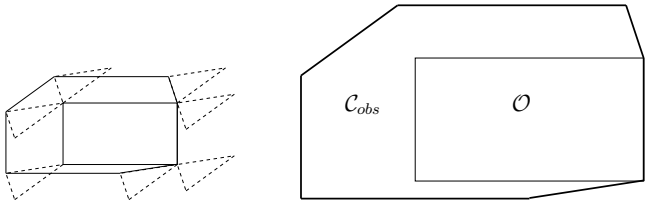

Fig. 7. (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

between $\mathcal{A}$ and $\mathcal{O}$. See Figures 6 and 7. Can you see how to achieve this by "reassembling" the edges of $\mathcal{A}$ and $\mathcal{O}$ into $\mathcal{C}_{obs}$, with the edges appearing in an ordering having to do with edge normals? Once this conversion is made, the trapezoidal decomposition approach is easily applied. If $\mathcal{A}$ and $\mathcal{O}$ are nonconvex, then they first need to be decomposed into convex pieces, to construct convex pieces of $\mathcal{C}_{obs}$. A trapezoidal decomposition algorithm could even be used for the convex decomposition of $\mathcal{A}$ and $\mathcal{O}$!

Now introduce rotation. For the translation-only case, $\mathcal{C}_{free}$ has a piecewise linear boundary because translation is a linear transformation. Unfortunately, rotation is nonlinear and commonly represented using trigonometric functions. Various ways to reparametrize rotation matrices lead to improvements; however, nonlinearity is unavoidable. For computation, polynomial parametrizations are preferred. The previous piecewise-linear representations then are replaced with *semi-algebraic* representations, meaning each "facet" of $\mathcal{A}$, $\mathcal{O}$, and $\mathcal{C}_{obs}$ is represented as the roots of implicit polynomials. Constructing $\mathcal{C}_{obs}$ in terms of polynomial roots is straightforward, but a combinatorial explosion occurs that produces far too many facets for practice (the example in Figure 6 already produces more than 70). For 3D problems, it becomes considerably worse. The next difficulty is to perform a cell decomposition. The first motion planning method to accomplish this is the cylindrical decomposition method of Schwartz and Sharir [13], which produces a number of cells that is doubly exponential in the dimension of $\mathcal{C}$. More efficient cell decomposition methods exist, and there is Canny's algorithm [3], which directly produces a roadmap through $\mathcal{C}_{free}$ in singly expoential time without a prior decomposition. These methods provide solutions to the general path planning problem; however, they are rarely even implemented due to numerical issues and inefficiency from the combinatorial explosion.

RRT(q_0)
1 G.init(q_0);
2 **repeat**
3 $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
4 $q_{near} \leftarrow$ NEAREST(G, q_{rand});
5 G.add_edge(q_{near}, q_{rand});

Fig. 8. A simple outline of the RRT algorithm.

V. SAMPLING-BASED PLANNING

Sampling-based approaches are by far the most common choice for "industrial-grade" problems because $\mathcal{C}_{obs}$ is composed of an unwieldy number of facets. They abandon the idea of explicitly characterizing $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$, and essentially leave the planning algorithm "in the dark" when exploring $\mathcal{C}_{free}$. The only "light" is provided by a collision detection algorithm, which is a black box that probes $\mathcal{C}$ to determine whether some configuration (or a small ball around it) lies in $\mathcal{C}_{free}$. These algorithms often work by hierarchically representing $\mathcal{A}$ and $\mathcal{O}$ and attempting to quickly determine collision at a course resolution [11]. Many collision detection methods are *incremental*, which means that they can yield extremely fast performance by saving information from a previous execution on a nearby configuration.

Planning algorithms then work by incrementally probing and searching $\mathcal{C}_{free}$ for a path, gradually revealing more and more of it with the collision detector. In this way, motion planning feels like using a robot with a weak sensor to explore an unknown environment. This might seem odd since $\mathcal{O}$ and $\mathcal{A}$ are given; however, the "environment" being explored is $\mathcal{C}_{free}$ (or equivalently, $\mathcal{C}_{obs}$), which is high-dimensional and prohibitive to explicitly represent. Sampling-based approaches attempt to find a solution quickly while cheating their way out of building a full "map" of $\mathcal{C}_{free}$. Don't compute more than you have to!

To get a feeling for sampling-based planning issues, we first introduce a frequently used method based on *rapidly exploring random trees (RRTs)*. Figures 8 and 9 show the algorithm and its result. The idea is to aggressively probe and explore the C-space by expanding incrementally from an initial configuration q_0. The explored territory is marked by a tree rooted at q_0. Each iteration extends the tree by adding a leaf vertex and edge that connects it to the rest of the tree. Each edge is a collision-free path between two configurations. The RRT algorithm picks a point q_{rand} *at random* in $\mathcal{C}$ (not $\mathcal{C}_{free}$), and then tries to connect the tree to it by extending the nearest point in the tree. This biases the tree toward aggressively reaching unexplored parts of $\mathcal{C}$, but eventually settling on uniform coverage.

Some implementation details are needed to clarify Figure 8. Step 1 initializes G to contain a single vertex, corresponding to q_0 and no edges. In Step 3, a random configuration generator is used to obtain $q_{rand} \in \mathcal{C}$. A random translation could be selected uniformly from a bounded region (often an axis-aligned rectangle). A random 2D rotation is obtained

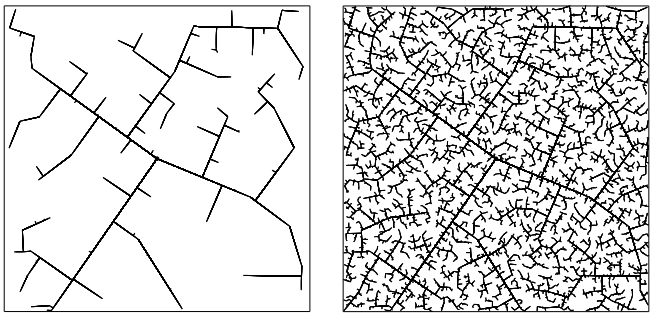

Fig. 9. In the early iterations, the RRT quickly reaches the unexplored parts. However, the RRT is dense in the limit (with probability one), which means that it gets arbitrarily close to any point in the space.

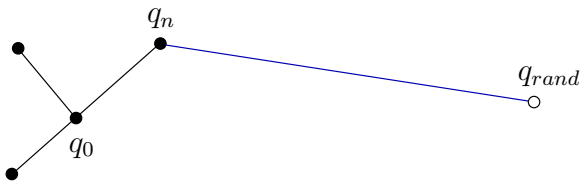

Fig. 10. A new edge is added that connects from the random sample q_{rand} to the nearest point in S, which is the vertex q_n.

easily by randomly selecting some $\theta \in [0, 2\pi)$. It turns out that selecting a uniformly random 3D rotation is technically more challenging. Here is an amazingly simple method. Choose three points $u_1, u_2, u_3 \in [0, 1]$ uniformly at random and then let [14]:

$$a = \sqrt{1-u_1}\sin 2\pi u_2 \qquad b = \sqrt{1-u_1}\cos 2\pi u_2$$
$$c = \sqrt{u_1}\sin 2\pi u_3 \qquad d = \sqrt{u_1}\cos 2\pi u_3 \tag{4}$$

in the rotation matrix (2).

What does uniform random really mean for $\mathcal{C}$? Recall from Section II that the set of transformations could be expressed in numerous ways, meaning that the notion of uniform randomness appears to be arbitrary. There is, however, a well-defined notion of uniformity based on *Haar measure*, which is beyond this tutorial; see Section 5.2 of [9]. Intuitively, if we rotate the coordinate frame on which the rotations are defined, then uniformity should be preserved. The methods for rotation above, including (4), achieve this.

Step 4 finds q_{near}, the closest point in G to q_{rand}; see Figure 10. What does it mean to be closest? This again depends on precisely how $\mathcal{C}$ is represented and implies that a distance function has been defined. The distance function $\rho : \mathcal{C} \times \mathcal{C} :\to [0, \infty)$ is formally called a *metric* and usually satisfies the following axioms for all $p, q, r \in \mathcal{C}$: 1) $\rho(p, q) \geq 0$, 2) $\rho(p, q) = 0$ if and only if $p = q$, 3) $\rho(p, q) = \rho(q, p)$, and 4) $\rho(p, q) + \rho(q, r) \geq \rho(p, r)$. In virtually all sampling-based planning algorithms, performance depends on the choice of metric. It is sometimes difficult to set the relative weights between rotational distances and translational distances; see Figure 11.

Now that "closest" has been established, which points in G are checked for being nearest to q_{rand}? The simplest is check the vertices and report the nearest one. But the closest point

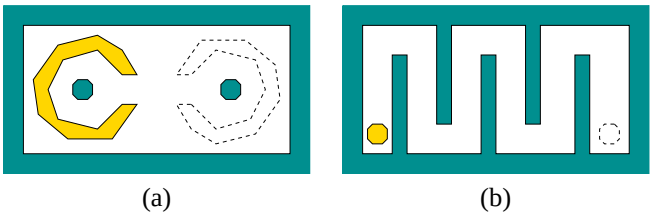

Fig. 11. Rotational vs. translation domination: (a) The task is to move the "C" shape to the right. Rotation dominates. Performance should improve if rotation is weighted heavily in the metric. (b) In this case, translation dominates, and should therefore be weighted more heavily if this fact is known in advance.

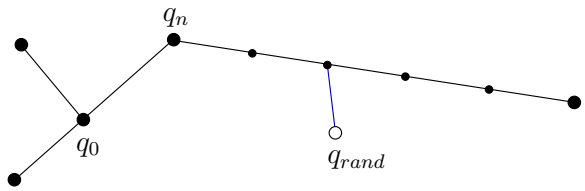

Fig. 12. For implementation ease, intermediate vertices can be inserted to avoid checking for closest points along line segments. The trade-off is that the number of vertices is increased dramatically.

among all those explored could lie along an edge. Rather than incur expensive computational cost, a common tradeoff is to check some intermediate points at regular intervals along an edge; see Figure 12. This introduces an unfortunate parameter to tune, but often simplifies implementations (it is also reasonable to avoid all of this and just use the vertices).

Finally, Step 5 extends the tree. If $\mathcal{C}_{obs}$ were empty, then an edge can be made from q_{near} to q_{rand}. If q_{near} is a vertex in G, then the endpoints of the new edge are q_{near} and q_{rand}. If q_{near} is a point along the interior of an edge, then that edge must first be split, with q_{near} introduced as an intermediate vertex. Since $\mathcal{C}_{obs}$ is usually not empty, there are two issues: 1) A collision detection algorithm makes sure that we can travel from q_{near} toward q_{rand} while staying in $\mathcal{C}_{free}$, and 2) we might not be able to reach q_{rand} without hitting $\mathcal{C}_{obs}$. If it is not possible to reach q_{rand}, then the new vertex is instead placed at the configuration q_i that gets as close as possible, as shown in Figure 13. (If no progress is possible, then no new edge and vertex are created.)

The RRT algorithm presented in Figure 8 aggressively explores $\mathcal{C}_{free}$; however, if the tree is grown from q_I, there is no consideration of q_G. Now consider ways to solve the basic path planning problem using RRTs.

Here is a simple adaptation. Start the RRT with $q_0 = q_I$ and at every 100th iteration, force $q_{rand} := q_G$ instead of

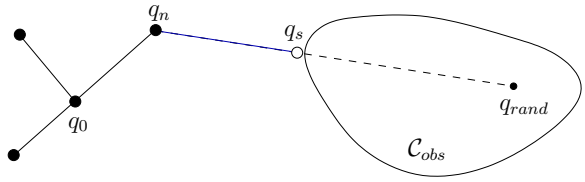

Fig. 13. If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by the collision detection algorithm.

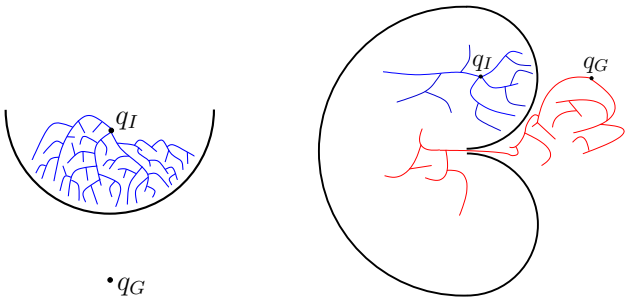

Fig. 14. The C-space obstacles may contain wells that trap planners in local minima or one-way doors that resemble bug traps.

choosing a random configuration. If q_G is reached, then a path has been found from q_I to q_G, which solves the problem. This induces a gentle bias toward the goal. At one extreme, we could pick q_G every time, making a beeline for q_G. This would fail miserably when an obstacle is reached. Figure 14(a) shows an example in which this would occur. Aggressively attempting to reach q_G by setting $q_{rand} := q_G$ in every other iteration would still work, but might waste too much effort running into $\mathcal{C}_{obs}$ instead of exploring. Therefore, a light bias, such as every 100th iteration is recommended.

For many problems, though, such a simple strategy is not enough. Figure 14(b) shows a kind of "bug trap" from which it is difficult to escape. Due to the existence of such situations, which commonly occur in practice, a *bidirectional* search is more effective and popular. The algorithm grows two RRTs: 1) G_I, rooted at q_I, and 2) G_G, rooted at q_G. Instead of always extending the trees using random configurations, half of the time is spent trying to extend each tree toward the newest vertex of the other tree. The following four iterations are repeated:

1) Generate q_{rand} and use it to extend G_I, obtaining a new leaf vertex q_{new}.
2) Force $q_{rand} := q_{new}$ and use it to extend G_G.
3) Generate a new q_{rand} and use it to extend G_G, obtaining a new leaf vertex q_{new}.
4) Force $q_{rand} := q_{new}$ and use it to extend G_I.

Steps 1 and 3 are identical to the execution in Figure 8, but for G_I and G_G, respectively. Steps 2 and 4 "trick" the RRT by using the most recent vertex from the other tree as a replacement for q_{rand}. If either of these two steps ever succeed in connecting the trees to each other, then the problem is solved. This method is quite effective for most practical problems, as aggressive exploration from q_I and q_G is balanced with trying to connect the trees to solve the problem.

An example that was solved in 2002 by the bidirectional RRT is the famous *Alpha 1.0 puzzle*, introduced by Nancy Amato and Boris Yamrom. The task is to pull apart the twisted nails, leading to an extremely narrow corridor in $\mathcal{C}_{free}$ through which the solution path must travel. The solution is illustrated in Figure 15. Most problems are not this challenging, and solutions are often found in a fraction

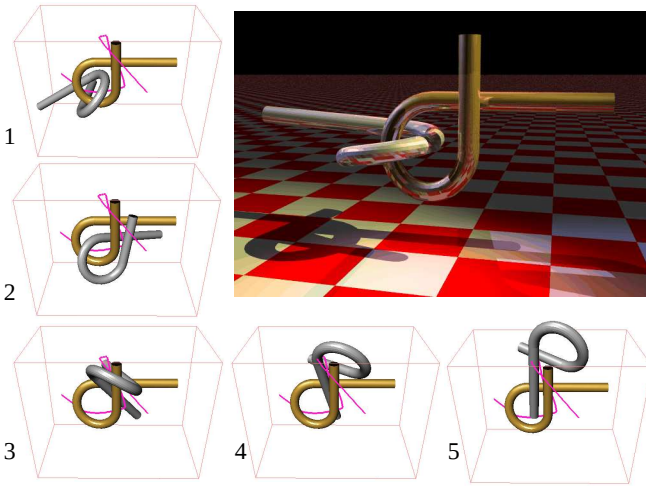

Fig. 15. The bidirectional RRT solves the Alpha 1.0 puzzle in a few minutes.

of a second. Nevertheless, there are limitations to the method as well as any sampling-based method. It is not hard to construct pathological examples that cause the algorithm to converge too slowly. In some cases, problem-specific heuristics can then be developed to recover performance.

The RRT-based methods fall into a larger family of methods called *incremental sampling and searching*, in which a graph is incrementally constructed inside of $\mathcal{C}_{free}$. Each method has a *vertex selection method*, which determines where to expand next from among vertices in the graph. After that, a *local planning method* constructs an edge from the selected vertex, thereby extending the tree. In the case of an RRT, the vertex selection method picks the vertex closest to q_{rand}. The local planning method attempts to connect the vertex to q_{rand}. As an example of an alternative incremental sampling and searching method, the Expansive Space Planner (ESP) [7] selects a vertex with probability that is inversely proportional to the number of other vertices within a ball of predetermined size. The local planning method then connects to a random configuration within the ball, but only with a probability that is inversely proportional to the number of vertices that lie within a ball centered on the random configuration. Another example that falls into this family is the randomized potential field planner [2], which implements gradient descent in $\mathcal{C}_{free}$ and uses random walks to escape local minima.

A common nuisance with sampling-based planning methods is that the produced paths are jagged as they traverse $\mathcal{C}_{free}$. This makes the solution animation jumpy; causing robots to follow such awkward paths is a comically bad idea. Therefore, *path smoothing* is usually performed to clean up solution paths. Fortunately, it is straightforward to produce a cleaner path once a jagged solution is given. A simple method is to iteratively pick a pair of points at random along the path and attempt to replace the path portion between them with a "straight line" in $\mathcal{C}_{free}$. If this survives the collision-detection verification step, then use the linear

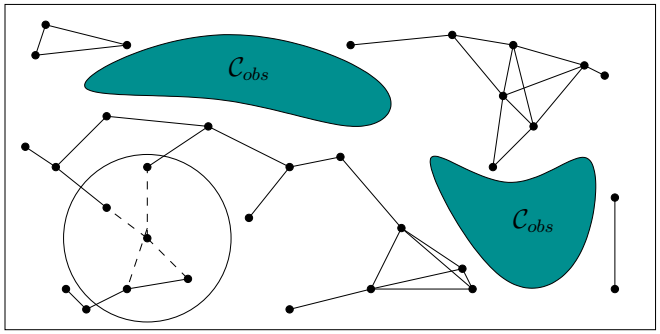

Fig. 16. The probabilistic roadmap method attempt to achieve roadmap accessibility and connectivity preservation via random sampling and connecting to nearby samples.

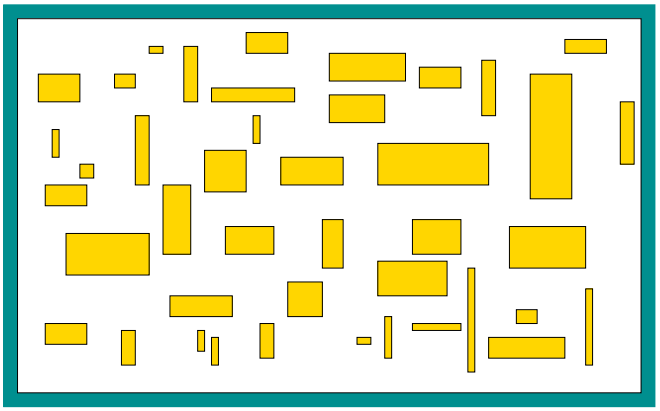

Fig. 17. Consider rearranging many rectangles, with no rotations, inside of a rectangular box in $\mathbb{R}^2$. Without a limit on the number of rectangles, the problem is NP-hard.

segment and discard the original part portion. After several dozen iterations, the path is usually much improved.

The discussion so far has focused only on *single-query* algorithms, meaning that only one q_I, q_G pair will be given so that there are no advantages of extensive precomputation. Recall from Section IV that planning problems can be quickly solved once a nice roadmap has been computed that offers the accessibility and connectivity-preserving properties. This motivates a *multiple-query* approach to sampling-based planning known as a *probabilistic roadmap* [8]. In this case, a bunch (e.g., 1000) of random configurations are chosen up front and declared to be roadmap vertices. Roadmap edges are formed by attempting to connect each configuration to all vertices within some specified radius; see Figure 16. If a roadmap can be constructed that satisfies accessibility and connectivity preservation with high probability, then it can be used to efficiently search for solutions to multiple initial-goal query pairs. One difficulty is that the roadmap may have as many edges and vertices as a high-dimensional grid [10], which provides motivation for pruning strategies that attempt to keep the good roadmap properties while reducing its size substantially. See, for example, the visibility roadmap variant [15].

To conclude, we should emphasize that a tradeoff has been made by going to sampling-based methods. Recall from Section IV that combinatorial planning leads to complete algorithms: They always find a solution if it exists; otherwise, they report failure. Since sampling-based methods solve problems without fully characterizing $\mathcal{C}_{obs}$, completeness is reduced to weaker forms. The goal is to ensure that the sampling eventually covers "all" of $\mathcal{C}$. This can be expressed in terms of *dispersion*, which is the radius of the largest empty (unsampled) ball in $\mathcal{C}$. Sampling-based approaches usually achieve *resolution completeness*, meaning that they will find a solution if one exists, but may run forever if one does not, or *probabilistic completeness*, meaning that the probability tends to one that a solution is found if one exists (otherwise, it may still run forever). For example, the RRT approaches described above lead to probabilistic completeness, partly because the dispersion is reduced to zero with probability one. Resolution completeness can be obtained by replacing the random configuration generator by a deterministic point sequence that leads to zero dispersion in $\mathcal{C}$ in the limit (for example, consider a multiresolution grid that refines forever).

The best way to learn more about sampling-based motion planning is to experiment with implementations. You could download and install a free library, such as the Open Motion Planning Library (OMPL) from Rice University, the Motion Strategy Library (MSL) from the University of Illinois, or the Motion Planning Kit (MPK) from Stanford. If you instead want to start from the basics, then at least downloading a collision detection package, such as PQP from the University of North Carolina, is recommended.

VI. DIRECT EXTENSIONS

Now that the core motion planning ideas have been explained for the case of rigid 2D or 3D robots among fixed obstacles, several straightforward extensions can be covered for which the planning methods are virtually the same.

The formulation of Section II allowed only one moving rigid body. This limited the C-space to having no more than dimension three for $\mathcal{W} = \mathbb{R}^2$ and six for $\mathcal{W} = \mathbb{R}^3$. If we allow multiple moving bodies, then there is no limit on the degrees of freedom, and hence, the dimension of $\mathcal{C}$. Consider, for example, Figure 17, in which a bunch of rectangles need to be rearranged by translation only. Each contributes two dimensions to $\mathcal{C}$. Interestingly, this problem is already NP-hard (and PSPACE-hard) if there is no maximum limit on the number of rectangles. (If the dimension of $\mathcal{C}$ is bounded in advance, then the path planning problem is solvable in time polynomial in the representation of the robot and world obstacles.)

Planning a collision-free path for multiple rigid bodies is no different conceptually to planning for a single body, once we think in terms of $\mathcal{C}$ and $\mathcal{C}_{free}$. The configuration vector $q \in \mathcal{C}$ includes coordinates to place each body. For example, for two translation-only rectangles, $q = (x_1, y_1, x_2, y_2)$ represents their position and $\mathcal{C} = \mathbb{R}^4$. The initial q_I and goal q_G configurations now express the placement of *every*

Fig. 18. The classic Puma 560 arm is a chain of three rotatable bodies (excluding the end effector) attached to a rigid base. This yields a three-dimensional C-space, which is handled by the standard planning algorithms. (Photo courtesy of the Technical University of Berlin.)

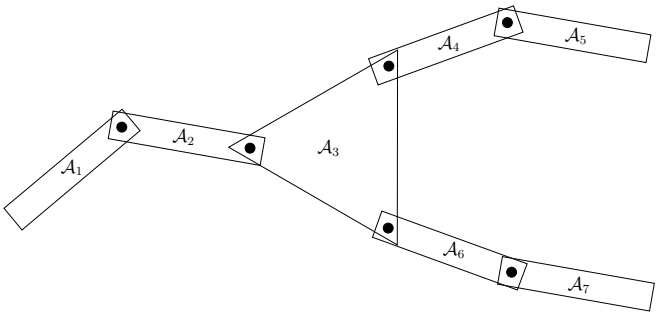

Fig. 19. Seven links are attached via rotatable joints. If each is allowed a full range of motion from 0 to 2π, then $\mathcal{C}$ is a seven-dimensional torus.

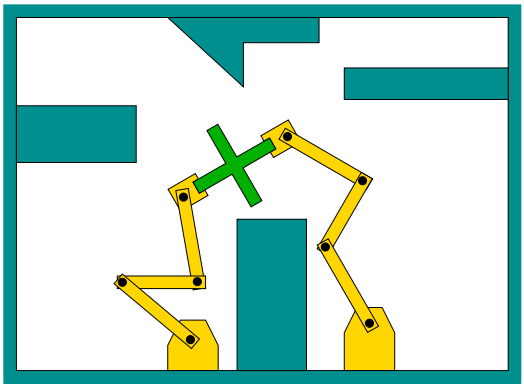

Fig. 20. Two or more arms manipulating the same object causes a closed kinematic chain.

body. Suppose there are n bodies $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_n$, with configuration parameters $q_1, \ldots, q_n$. If $\mathcal{A}_i$ is transformed into configuration q_i, it occupies $\mathcal{A}_i(q_i) \subset \mathcal{W}$ in the world. Let $q = (q_1, \ldots, q_n)$ represent the simultaneous configuration of all bodies. A configuration is collision-free, $q \in \mathcal{C}_{free}$, if and only if $\mathcal{A}_i(q_i) \cap \mathcal{O} = \emptyset$ for every i from 1 to n, and $\mathcal{A}_i(q_i) \cap \mathcal{A}_j(q_j) = \emptyset$ for every $i \neq j$. In other words, for $q \in \mathcal{C}_{free}$ there must be no body-obstacle collisions and no body-body collisions.

Once $\mathcal{C}$, q_I, q_G, and $\mathcal{C}_{free}$ are defined in this way, the methods of Sections IV and V directly apply. The only difficulty is that the dimension of $\mathcal{C}$ is large, which limits applicability of combinatorial methods and some sampling-based methods. This has motivated the development of various *decoupled* approaches, which avoid considering all bodies at once. For example, paths may be planned for each body individually, and then their motions along paths can be nicely times so that collisions are avoided. Such methods are not complete, but are practical in many settings. Alternatively, dimensionality reduction techniques, such as those based on the Johnson-Lindenstrauss Lemma, may hold promise for adapting sampling-based planning methods to directly account for all bodies simultaneously.

If bodies are allowed to contact each other, several other motion planning variants are obtained. Two will be considered here: 1) Articulated bodies, and 2) manipulation. For articulated bodies, they are attached together by joints that enable some freedom of motion between them, as shown in Figures 18 and 19. The attachment of bodies removes some of their collective degrees of freedom. Configuration coordinates express how each body is situated with respect to bodies to which it is connected. Expressions for transforming such bodies are just standard robot kinematics, covered in numerous textbooks [5], [16]. Somewhat different from standard kinematics, we are once again interested in the set of *all* possible transformations, resulting in the C-space. Once this has been defined, a manifold C-space $\mathcal{C}$ is usually obtained, on which q_I, q_G, and $\mathcal{C}_{free}$ are straightforward to define. Here, $\mathcal{C}_{free}$ includes some configurations in which there are body-body collisions, but only if these they are attached by a joint. Once defined, the methods of Sections IV and V once again apply, with the usual warning about the dimension of $\mathcal{C}$.

A more serious complication is when a collection of articulated bodies forms a loop, as shown in Figure 20. The result is called a *closed kinematic chain*, which occurs in parallel robots and if multiple robots contact the same body for manipulation. In most cases, it is difficult to explicitly characterize the set of configurations that satisfy the loop closure constraint. This makes it difficult to even parametrize paths through $\mathcal{C}$. Sampling-based planning approaches have nevertheless been developed to step through this difficult space by ensuring that loop closure is maintained while incrementally searching for a solution path.

Manipulation problems more generally require robots to determine which bodies to grasp and how to carry them to solve a problem. For example, the task might be to use a manipulator arm to stack several boxes. The degrees of freedom of boxes in addition to the robot are all included

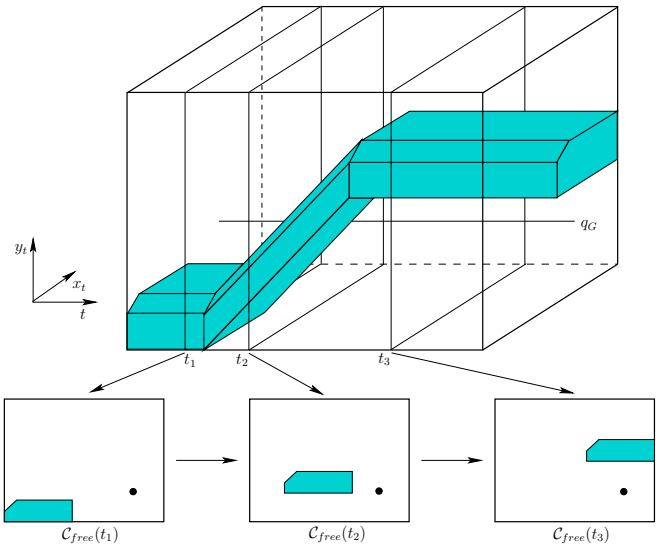

Fig. 21. A time-varying example with piecewise-linear obstacle motion. Planning through the state-time space occurs.

when defining $\mathcal{C}$. The task is expressed by specifying a configuration in which the boxes are stacked. This problem appears conceptually more challenging. Standard algorithms are often adapted to solve it by forming a hybrid C-space that includes discrete variables in addition to configuration variables. The discrete variables record modes of interaction. For example, there is a *transit mode*, when the manipulator is not carrying a body, and a *transfer mode*, when it carries a body. Heuristics are then used to determine when modes should be switched, in addition to solving the planning problem that arises in each mode.

Another variant of the basic path planning problem is to allow the obstacles to move. Let $T = [0, t_f]$ be an interval of *time*, in which t_f is some *final time*. In this case, a "snapshot" of the world can be imagined at every time $t \in T$. The obstacle region $\mathcal{O}$ becomes $\mathcal{O}(t)$. Now consider computing a collision-free path from time $t = 0$ to time $t = t_f$. This is conceptually straightforward if we construct the *configuration-time space*, $Z = \mathcal{C} \times T$. Figure 21 shows an example of how this appears. To solve the problem, path problem algorithms work in the usual way, with one exception: The path must always make forward progress through time. Combinatorial roadmap methods and incremental sampling and searching methods can be adapted without much difficulty to enforce this. It becomes considerably more challenging, however, if the robot has a maximum speed bound. This yields a constraint on the path slope through Z, which is more difficult to enforce. Finally, it is even more difficult, and practical, when there is uncertainty in predicting the future motions of the obstacles. This falls under the topic of uncertainty, which is covered in the next tutorial part.

VII. CONCLUSION

After reading this, you should hopefully have extracted the following main points. Motion planning lives in the C-space, which is the set of all transformations. Combinatorial planning solves simpler problems in a clean, elegant way, but the running time is too high for industrial-grade problems. Sampling-based planning provides practical solutions for real-world problems, but offers weaker guarantees. Performance degrades for problems in which narrow doorways in $\mathcal{C}_{free}$ are hard to find. Several extensions to the standard path planning problem expand the C-space definition and require only minor adaptations to the usual approaches. The key issue is that the C-space dimension increases, which generally raises computational complexity.

So we have seen powerful methods that generate a collision-free path automatically. Not bad. This is useful in many settings, extending well beyond robotics. But what if a robot is not able to follow the path due to differential constraints arising from kinematics and dynamics? What if we cannot predict precisely where the robot will go? What if the obstacle locations are uncertain and possibly changing? These concerns, with which every roboticist is familiar, motivate the topics in the second part of this tutorial.

REFERENCES

[1] V. I. Arnold. *Mathematical Methods of Classical Mechanics, 2nd Ed.* Springer-Verlag, Berlin, 1989.

[2] J. Barraquand, B. Langlois, and J. C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, & Cybernetics*, 22(2):224–241, 1992.

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

[4] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, 2005.

[5] J. J. Craig. *Introduction to Robotics*. Addison-Wesley, Reading, MA, 1989.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, Berlin, 2000.

[7] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *International Journal Computational Geometry & Applications*, 4:495–512, 1999.

[8] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics & Automation*, 12(4):566–580, June 1996.

[9] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Also available at http://planning.cs.uiuc.edu/.

[10] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research*, 23(7/8):673–692, July/August 2004.

[11] M. C. Lin and D. Manocha. Collision and proximity queries. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, 2nd Ed.*, pages 787–807. Chapman and Hall/CRC Press, New York, 2004.

[12] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computing*, C-32(2):108–120, 1983.

[13] J. T. Schwartz and M. Sharir. On the Piano Movers' Problem: III. Coordinating the motion of several independent bodies. *International Journal of Robotics Research*, 2(3):97–140, 1983.

[14] K. Shoemake. Uniform random rotations. In D. Kirk, editor, *Graphics Gems III*, pages 124–132. Academic, New York, 1992.

[15] T. Siméon, J.-P. Laumond., and C. Nissoux. Visibility based probabilistic roadmaps for motion planning. *Advanced Robotics Journal*, 14(6), 2000.

[16] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, New York, 2005.

[17] S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, Dept. of Electrical Engineering, California Institute of Technology, 1977.