EFFICIENTLY PLANNING COMPLIANT MOTION IN THE PLANE*

J. FRIEDMAN†, J. HERSHBERGER‡, AND J. SNOEYINK§

Abstract. Any practical model of robotic motion must cope with the uncertainty and imprecision inherent in real robots. One important model is compliant motion, in which a robot that encounters an obstacle obliquely may slide along the obstacle. In this paper, we start by investigating the geometry of compliant motion in the plane under perfect control and find a compact data structure encoding all paths to a goal. When we introduce uncertainty in control and position sensing, the same data structure allows us to find efficiently a compliant motion that reaches the goal if one exists, to compute the boundary of the non-directional backprojection of the goal, and to compute multi-step plans for sensorless robots. Our preprocessing and query approach has advantages of speed for on-line queries and flexibility for considering robots with different capabilities or initial positions in the same environment.

Key words. computational geometry, compliant motion, path hull data structure

AMS subject classifications. 68U05, 68P05

1. Introduction. Planning the motion of a robot is a practical problem with many theoretically appealing variants [3, 4, 12, 19, 32]. Given a robot's initial and goal position in some environment, there are typically many paths connecting the initial position and the goal position. The desirability of a path depends not only on properties of the path (e.g., a shortest path [33] or a high-clearance path [30]) but also on the control and sensing abilities of the robot.

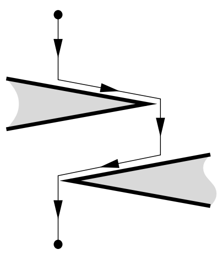


Fig. 1

We investigate a motion paradigm called *compliant motion.* A robot using compliant motion follows a "stable" path: even if the robot diverges slightly from the commanded path, it still reaches the goal. A point moving by compliant motion following a commanded direction α (with perfect control) travels through free space in direction α until it encounters an obstacle (see Figure 1). Then it slides along the obstacle until either friction is too large, or α no longer points into the obstacle. In the former case, it stops; in the latter, it resumes travel through free space in direction α. This type of motion enables the robot to "grope" its way towards the goal. In the basic model, the only way the robot can stop is by getting stuck [7, 8, 11]. In less restricted cases, some forms of goal sensing are possible [25].

When the robot's control is imperfect, the actual direction of the motion can deviate from α. We follow other researchers [7, 10, 25, 26] and assume that the deviation is bounded by some angle ϵ. For a given starting point, we look for all directions such that any motion whose instantaneous attempted direction deviates less than ϵ from the commanded direction is guaranteed to reach the goal.

* Research supported in part by Digital Equipment Corporation Systems Research Center. Preliminary versions of the material in this paper appeared in the 1989 ACM Symposium on Computational Geometry [14] and in the 1990 Scandinavian Workshop on Algorithm Theory [15].

† D. E. Shaw & Co., New York, NY.

‡ Mentor Graphics, 1001 Ridder Park Drive, San Jose, CA, USA 95131.

§ Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T 1Z4. Supported in part by an NSERC Research Grant and a fellowship from the B.C. Advanced Systems Institute.

Sliding on a boundary wall is "stable," since there is a range of directions that force the robot to slide in the same way along that wall. Given a robot's control uncertainty parameter ϵ, it may be possible to ensure that the robot always slides left (or right) along any particular wall that it encounters. Thus, it is sometimes possible to find a stable commanded direction that gets the robot all the way from the starting point to the goal. When there is no stable direction by which the robot can reach the goal, we may wish to outline a *plan,* which specifies a number of subgoals (regions of the environment), each one directly attainable from the previous one, that eventually lead to the goal. We would like to find a plan that contains as few subgoals as possible.

1.1. Previous work. In three dimensional space, compliant motion planning is a provably difficult problem. It is hard for *PSPACE* [29] and for *NEXPTIME* [4]. Although work has been done on this problem [2], most results, including our own, deal with the more tractable case of compliant motion in the plane.

Lozano-Pérez, Mason and Taylor [26] describe the *preimage backchaining* approach to compliant motion planning. Their idea is to compute the *preimage* or *backprojection* of the goal—the set of points in the environment from which the robot can reach the goal by following a single direction command. They treat the preimage as a new goal and compute its preimage. The second preimage contains all points that require a two-step plan to reach the original goal. They repeat this process until a region is found that contains the current position of the robot.

Erdmann [10] and Canny and Donald [7] describe algorithms for computing a *directional backprojection:* the set of points that reach the goal by one motion in a given direction α with a given control uncertainly ϵ. Their algorithms run in $O(n \log n)$ time and $O(n)$ space for an environment composed of n line segments (of *size* n) and a goal of constant size. Latombe [25] surveys these algorithms and other issues and approaches.

When the direction α is not given, Donald [8] presents an $O(n^4 \log n)$ algorithm for the one-step case, and an $O(n^{r^{O(1)}})$ algorithm for the r-step case, in which each motion terminates with the robot getting stuck. Briggs [1] improves the single-step bound to $O(n^2 \log n)$.

1.2. Contributions of this paper. Our approach emphasizes two features: query formulations, and late introduction of control uncertainty.

Query formulations of motion planning problems are based on the assumption that the robot moves repeatedly in the same environment. We divide the problem into two phases: preprocessing of the environment, and queries of the form "In what directions can a robot at a point p reach the goal?" For an environment with k polygonal obstacles, whose total number of vertices is n, the preprocessing time is $O(kn \log n)$ and the query time is $O(k \log n)$. Even if used for a single query, the dependence on kn (rather than n^2) allows us to describe the obstacles in greater detail, using more points, without having to pay the quadratic cost of Briggs' method [1]. For multiple queries the savings is even greater.

We assume perfect control in the preprocessing phase and build data structures that efficiently encode the entire set of paths to the goal from every point in the polygon. The magnitude of the control uncertainty is part of our query data. This allows us to change the control uncertainty of the robot dynamically, or to consider multiple robots with different control capabilities and positions, without recomputing the data structures that support compliant motion calculations. For a starting point given at query time, we can also compute the maximum possible control uncertainty under which we can still guarantee that the robot reaches the goal. This does not increase the query time.

Our basic algorithm for the single-step planning problem has a number of applications and extensions. First, we can construct the boundary of the region from which a robot with imperfect control can reach the goal in a single step in $O(kn \log n)$ time. Second, we can answer queries that specify, instead of a starting point, a region guaranteed to contain the robot's starting position. We return the set of directions in which the robot can reach the goal no matter where in the region it starts. The query time is at most $O(kn)$, but is typically less. Third, we show how to extend our one-step results to the case of sensorless robots (robots that must stick at the goal) with no change to the algorithms, data structures, or running times. Finally, we consider two polynomial variants of the multi-step planning problem: We solve a restricted version of the multi-step planning problem for sensorless robots using $O(kn^2 \log n)$ preprocessing, $O(kn^2)$ space, and $O(kn \log n)$ query time, improving Donald's exponential bound [8] in this special case. When $k = 1$, we show how to solve the multi-step problem for robots with perfect control and sensing with no change to the single-step complexity.

We employ a novel data structure called a *path hull*, which represents the convex hull of a simple polygonal chain (a *path*). Using the path hull data structure, one can find a tangent to the convex hull of the path, either from a point or parallel to a given line, in $O(\log n)$ time. Path hulls also support sequences of common maintenance operations at an amortized cost of $O(\log n)$ per operation. These sequences can consist of additions and deletions at the end of the path, and either path splits or joins (but not both splits and joins in the same sequence). We believe that path hulls are of independent interest.

The remainder of the paper is organized as follows. In the next section, we define our problem more precisely, and investigate the theoretical properties of compliant motion that we use in the algorithms. For clarity of the presentation, we describe the main algorithm in two steps. In Section 3 we present the algorithm for the special case in which the environment has no isolated obstacles (in other words, the environment is the interior of a simple polygon) and prove its correctness. In Section 4 we show how to extend the algorithm to the general case. In Section 5, we address the issues of imperfect control and position sensing, as well as the other applications of the basic algorithm. Section 6 describes in detail the path hull data structure used by the algorithms. We conclude with an open problem.

2. Fundamentals of compliant motion. In this section we define compliant motion and establish fundamental properties of paths followed by a robot doing compliant motion. We also investigate the α-backprojection, which is the set of points that reach the goal when commanded to move in direction α.

We first define notation. If a and b are points, we denote the direction from a to b by $\overrightarrow{ab}$. If α is a direction, then $\alpha^\perp$ is the direction 90° counterclockwise from α, and $-\alpha$ is the reverse direction. If α and β are directions, then the interval $[\alpha, \beta]$ denotes the range of directions from α counterclockwise to β. We later define a special direction 0 such that $\alpha < \beta$ means that $[\alpha, \beta]$ does not contain direction 0. If ϵ is an angle, then $\alpha + \epsilon$ is the direction ϵ counterclockwise from α. We use α^+ to represent a direction that is infinitesimally counterclockwise from α, so that if $\alpha^+ \leq \beta$ then $\alpha < \beta$. Similarly, α^- denotes a direction that is infinitesimally clockwise from α.

When the direction α is fixed within a given context, we generally choose the coordinate system so that α is parallel to the vector $(0, -1)$, and we say that α is pointing *down*. Within this frame of reference, the *up* direction is $-\alpha$, the *right* direction is $\alpha^{\perp}$, and the *left* direction is $-\alpha^{\perp}$.

2.1. Compliant motion trails. The environment, which we call P, is described by a set of k disjoint simple polygons that have a total of n vertices. Without loss of generality, we assume that one of the polygons, called the *outside polygon*, contains all the other polygons, called the *islands*. (We can always place a bounding polygon of constant complexity around an environment that has only islands, and insure that the algorithm never uses the new walls for sliding.) The *interior* of the environment is the interior of the outside polygon excluding the islands. None of the islands need contain any other island.

In this paper, the robot is a point. One can reduce more general robots to point robots by the *configuration space* approach of Lozano-Pérez and Wesley [27], in which a non-rotating convex robot corresponds to a reference point moving among configuration-space obstacles, and the configuration space can be computed in time linear in the size of the original environment (assuming the complexity of the robot itself is constant). The robot always moves through the interior of the environment, which we refer to as *free space*, or slides along a boundary edge of the environment (a *P-edge*).

Each P-edge has an associated *friction cone* that defines the robot's behavior when it hits the edge. Let $\overline{ab}$ be a P-edge, and assume that the interior of the environment is locally to the left of $\overrightarrow{ab}$. The friction cone of $\overline{ab}$ divides the direction interval $[\overrightarrow{ba}, \overrightarrow{ab}]$ into three zones: *zone a*, *zone b*, and the *stop zone*, which is a closed interval. See Figure 2. Suppose that a robot moving in direction α hits $\overline{ab}$. If α is in zone a, then the robot slides toward a; if α is in zone b, the robot slides toward b; and if α is in the stop zone, then the robot is stopped by friction when it hits $\overline{ab}$. Note that we do not require the friction cones to be symmetric, or even require the stop zone to include the normal to $\overline{ab}$. When the stop zone does not contain the normal, the P-edge acts as a "conveyor belt."

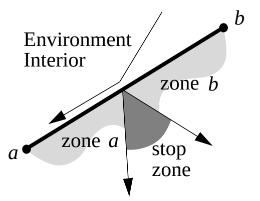


Fig. 2

When a robot sliding along a P-edge reaches an endpoint, it does one of three things: slides along an adjacent P-edge, leaves the boundary and go into free space, or gets stuck. Suppose that $\overline{ab}$ and $\overline{bc}$ are two adjacent P-edges, and the robot has just finished sliding along $\overline{ab}$ toward b. If the commanded direction α falls in zone c of P-edge $\overline{bc}$, then the robot continues to slide. If α does not point into $\overline{bc}$, then the robot goes into free space at b, following direction α. Otherwise, the robot gets stuck at b, and we say that b is a *sticky vertex*. A vertex may also be sticky if the robot reaches it from free space: If α is in zone b of $\overline{ab}$ and in zone c of $\overline{bc}$ then the robot proceeds through b as if it just slid into b along $\overline{ab}$. If this condition holds with $\overline{ab}$ and $\overline{bc}$ exchanged, then the robot proceeds through b as if it just slid into b along $\overline{bc}$. Otherwise, the robot gets stuck at b.

We introduce uncertainty in control in Section 5.1. Before then, we assume perfect control (unless we explicitly state otherwise), so a starting point p and a commanded direction α define a unique path that the robot follows, which we call a *trail* and denote by $T_\alpha(p)$. The path $T_\alpha(p)$ is a sequence of directed segments starting at p. When there is no ambiguity, we may use T_α instead of $T_\alpha(p)$.

The preprocessing phase of our algorithm works with a fixed goal on the boundary of the environment. Throughout the rest of this paper, we assume that the goal is a vertex g; however, our results also hold for a goal that is a P-edge. When $T_\alpha(p)$ contains g, we say that "α is a *good* direction for p."

We conclude this section with two properties of compliant motion trails. The first is Observation 2.1, which is in the spirit of the "kinetic framework" of Guibas, Ramshaw, and Stolfi [18].

OBSERVATION 2.1. *Let the commanded direction α point vertically downward. If the leftmost (rightmost) segment l along T_α is not the first or last segment, then l is vertical, and is directed down.*

Proof. If s and t are segments along T_α, and s has a positive horizontal component ("s is sloping to the right") while t has a negative horizontal component ("t is sloping to the left"), then there exists a free-space segment u along T_α between s and t (otherwise, there would be a sticky vertex in the middle of the trail). All free space segments in T_α are directed down. □

LEMMA 2.2. *The trail T_α has no loops.*

Proof. Suppose that T_α has a loop. If T_α goes counterclockwise around the loop, then T_α must be directed up through the rightmost portion of the loop; if T_α goes clockwise around the loop, then T_α must be directed up through the leftmost portion of the loop. In either case, we get a contradiction with Observation 2.1. □

2.2. The α-backprojection. Let α be a given direction. The *α-backprojection*, denoted by B_α, is the set of all points p for which α is a good direction. The α-backprojection plays an important role in our algorithms for compliant motion planning. In this section, we gain more insight into the structure of B_α.

Since α is fixed within the context of this section, we can assume that α points vertically downward. Thus we say that a point p is *above* point q when the vector $\overrightarrow{pq}$ has direction α. A P-edge $\overline{ab}$ of P is called *slidable towards a* if α is in zone a of $\overline{ab}$.

Figure 3 shows an example of an α-backprojection. In this figure, and in all the examples throughout this paper, the friction cones are the edge normals unless we explicitly state otherwise. Note that the boundary of B_α is composed of portions of P-edges and edges that extend through the free space of P. By the definition of sticky vertices (see Section 2.1 above), the free-space edges on the boundary of B_α are not part of B_α. (In the degenerate cases in which either of the P-edges incident to g is not slidable towards g, the free-space edge extending from g is part of B_α.) On the other hand, the P-edges on the boundary of B_α are part of B_α.

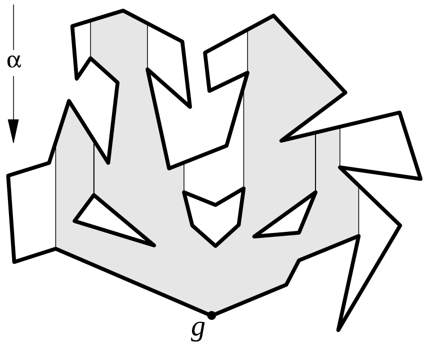


Fig. 3

Note that the α-backprojection is "almost" a simple polygon. Although some of the free-space edges that extend from islands have points of B_α on both sides, these free-space edges are not part of B_α and therefore the islands are not completely surrounded by B_α.

These properties hold not only for the example, but also for the general case, as we show in Lemma 2.4. For that proof only, we use the following definition and lemma.

The *upper envelope* of an island $\mathcal{I}$ is the set of boundary points of $\mathcal{I}$ with no points of $\mathcal{I}$ above them (with respect to a direction α). We can classify points p of the upper envelope of an island $\mathcal{I}$ into three categories:

i) *Left-inclined*: p is in B_α, and the robot slides left through p (α is pointing down).
ii) *Uninclined*: p is not in B_α, or p is the goal g.
iii) *Right-inclined*: p is in B_α, and the robot slides right through p.

LEMMA 2.3. *The upper envelope of an island $\mathcal{I}$ contains at least one uninclined point.*

Proof. Let l be the leftmost point of $\mathcal{I}$, and r be the rightmost point. Let C be the polygonal chain of the boundary of $\mathcal{I}$ counterclockwise from r to l, and let U be the upper envelope of $\mathcal{I}$. The points of U are a subset of the points of C, and form intervals along the chain C.

If l is uninclined, we're done; otherwise, l must be left-inclined. (If $\mathcal{I}$ contains g, then both l and r are uninclined.) If r is uninclined, we're also done, so r must be right-inclined. The points along U cannot change from left-inclined to right-inclined without having an uninclined point in between, and since neither category is empty, this proves the existence of an uninclined point. □

LEMMA 2.4. *The α-backprojection is a simply-connected subset of P whose interior is bounded by straight line segments. Each edge on the boundary of B_α is one of three types:*

- Bottom edges: *P-edges that have the interior of P locally above them;*
- Top edges: *portions of P-edges that have the interior of P locally below them;*
- Free-space edges: *edges that extend from a P-vertex upward all the way through the interior of P.*

Proof. The α-backprojection is connected because every point p in B_α has a path connecting p to g. We prove that B_α is simply-connected by showing that every simple loop ℓ in B_α is contractible to a point.

Notice that every maximal vertical segment in P that intersects $\ell \subset B_\alpha$ is in B_α. If ℓ contains no islands then these vertical segments cannot end inside ℓ. Thus, ℓ can be contracted to a point in its interior, which is in B_α.

On the other hand, any simple loop ℓ that contains an island contains an uninclined point on the upper envelope of each island. These upper envelopes, being monotone curves with respect to $\alpha^\perp$, can be ordered consistently with aboveness in direction α. Then there is a point $p \in \ell$ that is directly above an uninclined point on the uppermost curve, so p is not in B_α.

Since every point $p \in B_\alpha$ implies that every vertical segment in P that passes through p is also in B_α, the free-space edges on the boundary of B_α must be parallel to α and extend all the way through the interior of P. The remaining edges of B_α must be P-edges; P-edges on which the robot slides are included in B_α in their entirety. □

Another way of looking at the structure of B_α is more suitable for computing it. Consider the partition of P into trapezoids using the α-visibility graph [13]. Lemma 2.4 implies that every trapezoidal face and every vertical edge is either completely included or completely excluded from B_α.

LEMMA 2.5. *Consider the directed graph $G = (V, E)$, where*

- *the nodes V consist of the trapezoids of the α-backprojection, plus the goal g, and*
- *there is an arc from u to v if points of v slide into u.*

Then G is a tree (rooted at g).

Proof. G is connected, by Lemma 2.4. Points of each trapezoid slide into exactly one other trapezoid (or into g), so G is either a tree or contains a directed cycle. If G contained a cycle, we could exhibit a loop trail, contradicting Lemma 2.2. □

The algorithms that we present in Sections 3 and 4 use a *triangulation* of the environment P, that is, a partition of the interior of P into triangles whose vertices are P-vertices [31]. The following lemma will be used to help bound the working storage used by these algorithms.

LEMMA 2.6. *For a fixed triangulation of the environment P, every triangulation edge intersects B_α in at most k segments. (Recall that k is the number of disjoint simple polygons describing the environment.)*

Proof. First we prove a slightly stronger claim for the simple polygon case ($k = 1$): if points p and q are in B_α, and the segment $\overline{pq}$ is in free space, then the whole segment $\overline{pq}$ is in B_α. Without loss of generality, assume that p is to the left of q. Let p' be the first boundary point that $T_\alpha(p)$ hits, and let q' be the first boundary point that $T_\alpha(q)$ hits. Let C be the portion of the boundary counterclockwise from p' to q'. The polygon Q bounded by $\overline{qp}$, $\overline{pp'}$, C, and $\overline{q'q}$ is completely contained inside the environment. Now, any point r on the upper envelope U of C is along $T_\alpha(p)$ or $T_\alpha(q)$ or both, since if there were a point r on U not on either trail, the ray from r in direction $-\alpha$ would break Q into two disjoint pieces with $T_\alpha(p)$ confined to one piece and $T_\alpha(q)$ confined to the other. Because both trails are known to reach the same point g, the claim follows.

We can reduce the general case in which the outside polygon contains $k-1$ islands to the simple polygon case. It follows from Lemma 2.4 that every island has a ray extending from the island through free space in direction $-\alpha$ that is not contained in the α-backprojection. If we cut tunnels in the environment along these rays, we connect all the islands to the outside polygon without changing the α-backprojection. These tunnels divide each triangulation edge into at most k segments, and by the claim above, each such segment intersects the α-backprojection in at most one interval. □

2.3. The non-crossing theorem. We now state the key property of compliant motion paths. Let p be a point in the environment, and let α and β be arbitrary directions. The following theorem captures the fact that although the trails T_α and T_β start together at p, and may touch at other points, they do not cross. This property holds whether or not the environment has islands, and for arbitrary friction cones.

THEOREM 2.7. *Let p be a point in the environment, and let α and β be directions. Then there is a bi-infinite polygonal curve Γ that does not cross itself (it may touch itself without crossing), such that*

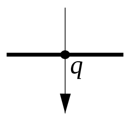
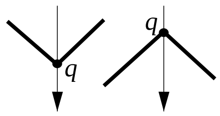
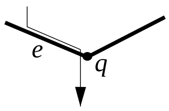
- *T_α is contained in Γ, and*
- *T_β is contained in the closure of exactly one of the open sets into which Γ partitions the plane.*

The proof of Theorem 2.7 is rather involved. The details of the proof are not used elsewhere in this paper (we need only the theorem itself); it may be easier to skim through this section at this time, and come back to it at a later reading, after acquiring a better feel for compliant motion.

Here is an overview of the proof. First, we construct Γ by extending T_α to infinity on either side (so, in fact, T_α is completely contained in Γ), and then we prove by induction that T_β stays to one side of Γ.

For a given direction γ, we define the *stabbing trail* $\widehat{T}_\gamma$ in the next two paragraphs as an infinite extension of the compliant motion trail T_γ. The stabbing trail is a polygonal chain that starts at p, has no loops, and extends to infinity. We proceed to examine the mutual behavior of the stabbing trails $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$. These two stabbing trails do not touch each other, so their concatenation can "almost" serve as our Γ. Some minor modifications to the concatenation of $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$ provides us with the final Γ.

Let us be more formal. The stabbing trail $\widehat{T}_\gamma$ is a compliant motion trail that occasionally crosses the environment boundary. In Section 2.1 we explained that every P-edge has an associated friction cone, which determines the sliding direction for a robot hitting the P-edge from the environment interior. In this section, we call this friction cone the *internal* friction cone. We rotate the internal friction cone by 180° to obtain the *external* friction cone, which determines the behavior of a hypothetical robot hitting the P-edge from the environment exterior. (The real robot cannot do this, of course.) The stabbing trail $\widehat{T}_\gamma(p)$ starts off by tracing the compliant motion of a robot starting at point p with commanded direction γ. If the compliant motion terminates at a sticky point q on a P-edge or a vertex, the stabbing trail proceeds as described in Table 1. (The first and second cases are relatively natural; we shall see later why the third case is different.)

Case	Example	Action
q is on a P-edge.		The motion simply switches sides between the interior and the exterior sides of the P-edge.
q is a sticky vertex the robot reached from free space.		We resume the motion at q (in some cases the motion resumes in the interior, while in others, in the exterior).
q is a sticky vertex, and the robot slid into q along P-edge e incident to it.		We interrupt the sliding at a point q' along e that is infinitesimally close to q.[a] We then continue by switching sides between the interior and exterior at q'.

[a] The stabbing trail $\widehat{T}_\gamma$ stops at point q' along e whose vertical projection (a projection on a line perpendicular to γ) is at a distance less than some δ from the vertical projection of q. A reasonable upper bound for δ is the smallest difference between the vertical projections of any two vertices whose vertical projections are distinct.

Table 1

When we get past the sticky point q, we resume compliant motion. If we encounter a new sticky point, we treat it like the first sticky point.

Observation 2.1 holds for stabbing trails, because a stabbing trail is nothing but a regular trail in a modified environment, in which the original P-edges are cloned (so we have two edges, the "interior" and the "exterior" edges, each with its corresponding friction cone, for each original P-edge), and the places where a stabbing trail goes through an original wall are replaced by infinitesimal holes in the new wall. Lemma 2.2 holds for stabbing trails for the same reason, applying the additional fact that a stabbing trail can slide only along one side (either the interior or the exterior) of any given edge. At some point, the stabbing trail crosses the boundary of the outside polygon for the last time and continues in a straight line to infinity.

The crux of the proof of Theorem 2.7 is the following lemma:

LEMMA 2.8. *The stabbing trails $\widehat{T}_{-\alpha}(p)$ and $\widehat{T}_{\alpha}(p)$ share no points other than p.*

Proof. We prove the lemma by assuming that $\widehat{T}_{-\alpha}$ and $\widehat{T}_{\alpha}$ do meet and deriving a contradiction. Let q be the first point (after p) along $\widehat{T}_{-\alpha}$ that is also on $\widehat{T}_{\alpha}$. Denote the portion of $\widehat{T}_{-\alpha}$ between p and q by A, and the portion of $\widehat{T}_{\alpha}$ between p and q by B. Since q is the first meeting point, A and B share p and q, but no other points.

When A and B separate at p, A goes "up" and B goes "down." More precisely, there is a line σ_p through p (σ_p is not necessarily horizontal) such that in the vicinity of p, A is above σ_p and B is below σ_p. (We may take σ_p to be the angular bisector of the initial segments of A and B.)

Observation 2.1 applies to the partial trails A and B—namely, rightmost and leftmost points, r and l, on $A \cup B$ are p or q or appear on free space edges going up on A or going down on B. We look at the cases for these extreme points and show that A and B must intersect. This contradiction establishes the lemma.

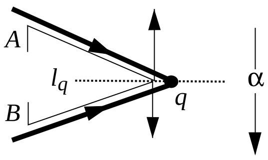


Fig. 4

First, suppose that neither r nor l is the point q. If both r and l come from one trail, say A, then the portion of A between r and l separates p below from q above because A remains between r and l, goes up at both r and l, and has no loops. But B also remains between r and l and must therefore cross A to connect p to q. On the other hand, if r comes from A and l comes from B, then consider where q is with respect to the path π from r to l that follows A to p and B to l. If q is above, then B must cross π from below to reach q; if q is below, then A must cross π from above.

Next, suppose that $r = q$. An analysis like the one above shows that no matter whether the leftmost point l is on A, is on B, or is p, the path A must reach q from above and B from below. But then the trails $\widehat{T}_{-\alpha}(p)$ and $\widehat{T}_{\alpha}(p)$ must both be sliding towards q on edges that meet at a vertex at q. As illustrated in Figure 4, however, q is a sticky vertex for both trails, and the stabbing trails would pierce their respective edges according to line 3 in Table 1, before reaching q. This contradicts their supposed behavior.□

As we mentioned at the beginning of the section, we would like to use the concatenation of $\widehat{T}_{\alpha}$ and $\widehat{T}_{-\alpha}$ as our Γ. However, we also mentioned that we want T_{α} to be completely contained in Γ, which is not the case if $\delta > 0$ (see footnote a at the bottom of Table 1). Thus we let $\delta \to 0$. This may cause Γ to become a non-simple curve: Figure 4 shows that if $\delta = 0$, the two stabbing trails may touch each other at a sticky vertex and then separate. The following lemma helps characterize the points where Γ touches itself:

LEMMA 2.9. *Let γ be an arbitrary direction, and let v be a point on the boundary of the environment that is sticky for both direction γ and direction $-\gamma$. Then v is sticky for any direction that reaches v (from another point in the environment).*

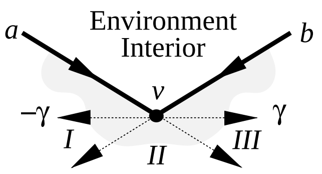


Fig. 5

Proof. Every P-edge can affect only one of the directions γ or $-\gamma$, and therefore v has to be a vertex. Furthermore, γ has to point into one of the P-edges incident to v, and $-\gamma$ has to point into the other P-edge. This leaves only one possibility, as shown in Figure 5: $\overrightarrow{av}$ is a conveyor belt at least strong enough to carry $-\gamma$ into v, and $\overrightarrow{bv}$ is a conveyor belt that carries γ into v.

Directions in zone I are forced by $-\gamma$ to slide along $\overrightarrow{av}$, and cannot slide out of v along $\overline{vb}$. Similarly, directions in zone III are forced by γ to slide along $\overrightarrow{bv}$ and are not affected by $\overrightarrow{av}$. Directions in zone II are forced by both P-edges to slide into v. Finally, any other direction that reaches v has to be either between $-\gamma$ and $\overrightarrow{va}$ but still slide towards v, or be between γ and $\overrightarrow{vb}$ but still slide towards v; in both cases the direction cannot slide out of v along the other P-edge. ☐

Lemma 2.9 implies that with Γ set to be the concatenation of the two stabbing trails $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$ (where $\delta = 0$), T_β can touch only one of the vertices where Γ touches itself (and terminate there), and T_β slides along at most one of the edges incident to that vertex.

LEMMA 2.10. *Let Γ be the concatenation of the stabbing trails $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$, with $\delta = 0$. The trail T_β never crosses Γ.*

Proof. When $\delta > 0$, the simply-connected polygonal chain Γ divides the plane into two parts, the "left" and the "right" parts (when α points down). The left half is locally on the right of $\widehat{T}_\alpha$ (when looking in the direction of progress of $\widehat{T}_\alpha$), and locally on the left of $\widehat{T}_{-\alpha}$. When $\delta = 0$, there may be more than two regions. We label the regions "left" and "right" by continuity as $\delta \rightarrow 0$.

Looking along T_β, starting from p, we can see two kinds of "events," a *Separation*—T_β separates from Γ (either at p, or after a common segment), and a *Meeting*—T_β runs into Γ. The first event is a separation. Without loss of generality, we can assume that T_β goes to the right side of Γ at that point (so β is in the range counterclockwise from α to $-\alpha$). We prove the following three claims by induction on the events:

(1) All meetings occur along P-edges.
(2) If a point $q \neq p$ on a P-edge e is common to both T_β and Γ, then T_β slides along e through q in the same direction as Γ (recall that at any point, Γ is either $\widehat{T}_\alpha$ or $\widehat{T}_{-\alpha}$).
(3) At a separation, T_β always splits off to the right side of Γ (meaning locally to the left of $\widehat{T}_\alpha$ and locally to the right of $\widehat{T}_{-\alpha}$.

Base Case. The first event. Claims (1) and (2) are trivially true, and claim (3) is true by assumption.

Induction step. Suppose that all three claims are true for the portion of T_β between p and the j^{th} event (the *old* event), and prove that the three claims hold for the portion between the j^{th} and the $(j+1)^{\text{th}}$ event (the *new* event).

Case 1. The old event is a separation. We have to prove claims (1) and (2) for the new event in this case. By the induction hypothesis, T_β split off to the right side of Γ. Since Γ goes to infinity at both ends, T_β stays on the right side until the new meeting. A free-space meeting cannot happen, since at a free-space meeting T_β crosses from the left side of Γ to the right side, so claim (1) is true at the new meeting: the meeting occurs at a point q along a P-edge e. At the meeting point q, exactly one of T_β or Γ reaches q from free space (one has to reach q from free space, or else it would not be an event; and by the argument of the previous sentence (the same argument that rules out free-space meetings), both cannot reach q from free space).

We divide Case 1 into the following subcases:

Case 1*a*. T_β reaches q from free space. First suppose that T_β hits $\widehat{T}_\alpha$. If T_α is sliding to the left (when α is pointing down) then T_β has to reach q from the left side of Γ, since β is counterclockwise from α (see Figure 6), but this is a contradiction of the induction hypothesis, so $\widehat{T}_\alpha$ slides to the right. Now, since α is in the right-sliding zone of edge e, and β is counterclockwise from α, β is also in the right-sliding zone. Similarly, we can prove that if T_β hits $\widehat{T}_{-\alpha}$, then $\widehat{T}_{-\alpha}$ has to be sliding right, which forces T_β to slide right, too.

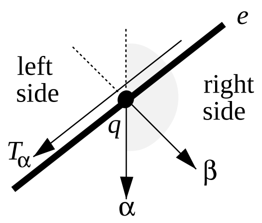


Fig. 6

Case 1*b*. T_β slides into q. This is essentially the same as Case 1*a*, except that α and β switch roles (β is counterclockwise from α).

Case 2. The old event is a meeting. We have to prove claim (2) for the common portion of T_β and Γ, and claim (3) for the new event.

We prove claim (2) by a new induction on the number of edges in the common portion. The base case is by the main induction hypothesis of this proof. For the induction step, if T_β keeps on sliding along Γ, we use the fact that $\widehat{T}_\alpha$ continues to slide right and $\widehat{T}_{-\alpha}$ continues to slide left (see Case 1 above) to conclude that they still force T_β to slide along with them. We omit the details for brevity.

For claim (3), we have a few subcases:

Case 2*a*. Both trails go into free space. By our initial assumption, T_β splits off counterclockwise of $\widehat{T}_\alpha$ and clockwise of $\widehat{T}_{-\alpha}$, so in both cases T_β splits off to the right side of Γ.

Case 2*b*. T_β goes into free space. First suppose that prior to the separation, T_β was sliding together with $\widehat{T}_\alpha$. It is not hard to verify that the only way T_β can go into free space and leave $\widehat{T}_\alpha$ sliding is if both trails were sliding to the right. This way, the environment interior was on the left-hand side when looking down $\widehat{T}_\alpha$. Since T_β split into free space it must have turned to the left-hand side when looking down $\widehat{T}_\alpha$, which is the right side of Γ. (See figure 7.) Similarly, when T_β leaves $\widehat{T}_{-\alpha}$, it turns to the right-hand side when looking down $\widehat{T}_{-\alpha}$, which is again the right side of Γ.

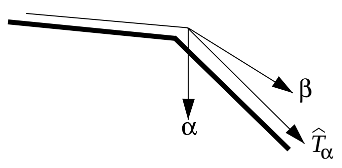


Fig. 7

Case 2*c*. Γ goes into free space, but not through a puncture (see Table 1, line 3). Similar to Case 2*b*.

Case 2d. Γ goes into free space, through a puncture. Either T_β stops at the meeting point q, or it does not and only one of $\widehat{T}_\alpha$ and $\widehat{T}_{-\alpha}$ reaches q (Lemma 2.9). In the latter case T_β and $\widehat{T}_\alpha$ (or $\widehat{T}_{-\alpha}$) are sliding to the right (else T_β would stop at q). When $\widehat{T}_\alpha$ ($\widehat{T}_{-\alpha}$) goes through the puncture at q, T_β continues to travel right, and hence splits off to the right side of Γ. $\square$

This concludes the proof of Theorem 2.7, the non-crossing theorem.

2.4. Properties of regions bounded by trails. It follows from Theorem 2.7 and Lemma 2.2 that if q is any common point of T_α and T_β other than p, then there are points $s_1, s_2, \ldots, s_j$ and $t_1, t_2, \ldots, t_j$, all common to T_α and T_β (it is possible that some of the s_i's coincide with some of the t_i's), such that:

- The points appear in the order $s_1, t_1, s_2, t_2, \ldots, s_j, t_j$ along both T_α and T_β;
- T_α does not touch T_β between s_i and t_i, and coincides with T_β between t_i and s_{i+1};
- $p = s_1$ and $q = t_j$.

This means that the portions of T_α and T_β between p and q bound a simply-connected polygonal region $R_{\alpha,\beta,q}(p)$. When α and β are both good directions (see Section 2.1) for some point p, we simply write $R_{\alpha,\beta}(p)$ instead of $R_{\alpha,\beta,g}(p)$; when p is fixed, we may write $R_{\alpha,\beta}$.

The trails T_α and T_β form the boundary of $R_{\alpha,\beta}$. In Section 4.1, we need to know more about whether T_α circumnavigates this region clockwise or counterclockwise. To answer this question, we need to define the direction 0 (zero) mentioned at the beginning of Section 2. The direction 0 is one that is not good for any point in the environment other than the goal itself. An example of such a direction is the one that points into the environment along the bisector of the two P-edges incident to the goal. This direction does not point into either P-edge incident to g, so a robot following commanded direction 0 cannot slide along them. Furthermore, in free space locally around g, it points away from g. Therefore, no compliant motion path with a programmed direction 0 can end at g. The following lemma states a consequence of the definition of direction 0:

LEMMA 2.11. *If $\alpha < \beta$ are good directions for p, then T_α traverses $R_{\alpha,\beta}$ counterclockwise (and T_β traverses it clockwise).*

Proof. Consider $\widehat{T}_0(p)$, the stabbing trail (defined in Section 2.3) starting at p with direction 0. Theorem 2.7 guarantees that $\widehat{T}_0(p)$ does not cross either T_α or T_β. Furthermore, by definition, $\widehat{T}_0(p)$ does not reach g. Since $\widehat{T}_0$ is an unbounded path, it cannot be inside $R_{\alpha,\beta}$. It follows that wherever T_α and T_β separate for the first time (usually at p), $R_{\alpha,\beta}$ lies in the interval counterclockwise of T_α and clockwise of T_β. This proves the lemma. $\square$

Combining the above properties, we conclude the section with the following theorem:

THEOREM 2.12. *Let p be a point inside P, and let $\alpha < \beta$ be good directions for p. Also, assume that $R_{\alpha,\beta}$ contains no islands of P. Then every $\gamma \in [\alpha, \beta]$ is a good direction for p. Furthermore, if $[\alpha, \beta]$ is smaller than $180°$, then every $\gamma \in [\alpha, \beta]$ is a good direction for any $q \in R_{\alpha,\beta}$.*

Proof. We prove the theorem in three steps: first, we prove that every $\gamma \in [\alpha, \beta]$ is good for p; second, we prove that if $[\alpha, \beta]$ is smaller than $180°$, then α is good for any point q along $T_\beta(p)$ and β is good for any point q along $T_\alpha(p)$; third, we prove that if $[\alpha, \beta]$ is smaller than $180°$, then both α and β are good directions for every point q inside $R_{\alpha,\beta}$. The third step gives a polygon $R_{\alpha,\beta}(q)$; applying the first step to it proves the second part of the theorem.

Let γ be a direction in $[\alpha, \beta]$. By Lemma 2.11, T_γ starts off by going into $R_{\alpha,\beta}$. A case analysis similar to that in the proof of Lemma 2.10 shows that any time T_γ hits either T_α or T_β, T_γ slides along with them. Since the interior of $R_{\alpha,\beta}$ is free of P-edges, T_γ cannot get stuck in the interior of $R_{\alpha,\beta}$. Because compliant motion trails inside a bounded region are of finite lengths (in fact, if the absolute height of the region, with respect to α, is h, and there are n P-edges totalling length l, the trail T_α cannot exceed $n \cdot h + l$ in length), and because the trail T_γ cannot get stuck inside or on the boundary of $R_{\alpha,\beta}$, it follows that T_γ has to reach g.

Continuing with the second part of the proof, let q be a point along $T_\alpha(p)$. We prove the stated claim by induction on the number of times $T_\beta(q)$ hits $T_\alpha(p)$. In the base case, if $T_\beta(q)$ does not hit $T_\alpha(p)$, then $T_\beta(q)$ must hit $T_\beta(p)$—this is because $R_{\alpha,\beta}$ is bounded, and $T_\beta(q)$ splits off to the left of $T_\alpha(p)$ and therefore into $R_{\alpha,\beta}$ ($[\alpha, \beta]$ is smaller than $180°$), and the interior of $R_{\alpha,\beta}$ is free of P-edges. Once $T_\beta(q)$ hits $T_\beta(p)$, it follows $T_\beta(p)$ all the way to g. For the inductive step, look at the first time $T_\beta(q)$ hits $T_\alpha(p)$. This hitting point has to be along a P-edge, and since $T_\alpha(p)$ slides on that P-edge, $T_\beta(q)$ slides there also. Follow the two trails together until the first time they separate, and we've reduced the number of meetings between $T_\beta(q)$ and $T_\alpha(p)$. The other direction of the claim is symmetric.

For the third part of the proof, let q be an arbitrary point strictly inside $R_{\alpha,\beta}$. Since $R_{\alpha,\beta}$ is bounded, when we extend a ray from q in direction $-\alpha$, we hit the boundary of $R_{\alpha,\beta}$, say at point r. This point cannot be along $T_\alpha(p)$: if r belonged to $T_\alpha(p)$ then so would q, but we have assumed that q is inside $R_{\alpha,\beta}$. Therefore, r must lie along $T_\beta(p)$. Applying the previous step, we conclude that α is good for r, and therefore good for q. □

Remark: If $[\alpha, \beta] > 180°$, then the second part of Theorem 2.12 does not hold. Figure 8 shows a counterexample. In the figure, the two P-edges incident to the goal are conveyor belts toward the goal. Direction β is bad for the point q.

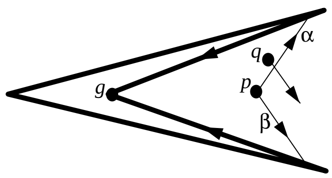


Fig. 8

3. The simple polygon algorithm. In this section we restrict ourselves to the case $k = 1$, namely, to environments that have no islands. We present an algorithm for single-step, perfect-control, perfect-sensing compliant motion planning: for an environment with n vertices and a goal vertex g. The algorithm spends $O(n \log n)$ preprocessing time to produce a linear-size data structure that represents the non-directional backprojection, which is the union of the α-backprojections for all α. This data structure subsequently enables us to produce a "single-step plan" (i.e., report all the good directions), once the starting point p is given, in $O(\log n)$ time.

Theorem 2.12 says that if P contains no islands, then the good directions of every point p inside P form a single range, bounded by the directions $start(p)$ and $stop(p)$. This range is empty for points that cannot reach the goal in a single compliant motion. If we can compute the functions $start(p)$ and $stop(p)$ for any given point p, we can easily answer queries for the one-step compliant motion planning problem.

In this section, we show how to compute two subdivisions of P, called the *start* and *stop* subdivisions, that correspond to the two functions. Each subdivision partitions the non-directional backprojection into $O(n)$ triangles and trapezoids. Each triangle or trapezoid groups together points for which we compute the corresponding function in a uniform way. By performing a point location in each subdivision, we can in $O(\log n)$ time locate the region that contains a query point, and then compute the value of the corresponding function in constant time. Once a triangulation of P is given, we compute the subdivisions in linear space and $O(\tau \log n)$ time, where τ is the number of triangulation edges that intersect the non-directional backprojection. (We can compute a triangulation of a simple polygon in $O(n \log n)$ time [16], or in linear time by a more complex algorithm [5].)

In Sections 3.1 and 3.2, we describe two algorithms that compute the α-backprojection: the set of all points for which α is a good direction. The first sweeps through the trapezoidation given by the α-visibility map; the second shows how to use an extension of the *path hull* data structure of Dobkin et al. [6] to perform the sweep without the trapezoidation. Path hulls are described in detail in Section 6. The second algorithm is a subroutine in constructing the non-directional backprojection.

Sections 3.3 through 3.5 describe and analyze the rotation algorithm for computing the start and stop subdivisions that represent the non-directional backprojection. The idea is to maintain the α-backprojection B_α as α rotates.

Most of the time (informally speaking) we can rotate α slightly without changing B_α significantly: only the free-space edges rotate. We distinguish two types of free-space edges: the *leading* free-space edges sweep over new points and the *trailing* free-space edges sweep over points that leave the α-backprojection as α turns counterclockwise. For some α's, which we call *events*, the structure of B_α may change more dramatically. Events occur when a rotating free-space edge hits a vertex, a P-edge starts or stops being slidable, or an edge turns over and goes from top to bottom. We maintain the directions of events in a priority queue. While the queue is not empty, the algorithm rotates the current direction α until just past the first scheduled event. Processing the event involves updating B_α and the list of events. Again, path hulls are used to detect events.

3.1. The trapezoid algorithm for the α-backprojection. As a warmup, we start with an easy "trapezoid algorithm" for computing B_α, given the α-visibility map of P, which is the decomposition of P into trapezoids given by cutting parallel to α through every vertex of P.

The trapezoid algorithm maintains a connected set of trapezoids that form a subset of B_α. This set of trapezoids is bounded by a "front" of free-space edges parallel to α. At each step, the algorithm tries to expand its set of trapezoids by pushing a free-space edge on the front away from the subset and into the neighboring trapezoid. To specify the action of the algorithm, let f be the free-space edge that bounds the current set of trapezoids on the left. Initially, we begin with a degenerate trapezoid of zero width bounded on both sides by the segment above the goal g.

Suppose that f, moving left, encounters a vertex v at its base. If the P-edge clockwise from v around P can slide into v, then f sweeps over its left neighboring trapezoid, adding the trapezoid to B_α. Otherwise f stops, leaves the front, and becomes a free-space edge bounding B_α (possibly of zero length if v is both top and bottom of f). See Figure 9.

If v is encountered in the middle of f, then we split f into two segments. The lower one continues to sweep left. The upper segment sweeps left if and only if the

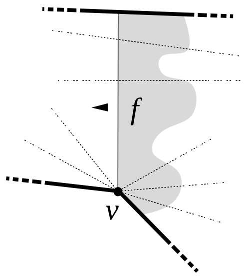


Fig. 9

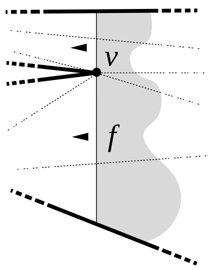


Fig. 10

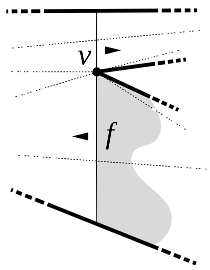


Fig. 11

P-edge clockwise from v is slidable into v. Otherwise, the upper segment stops and becomes a free-space edge bounding B_α. See Figure 10.

If v is at the top of f, then we have two cases. If one of the P-edges incident to v goes to the left, then f continues to sweep left. If both P-edges incident to v go right, then we extend f beyond v until it hits the polygon boundary. We create a new front segment for the upper portion of f which sweeps right if the upper P-edge incident to v is slidable into v; segment f continues to sweep left. See Figure 11.

The algorithm terminates when no more free-space edges can be moved.

LEMMA 3.1. *The trapezoid algorithm computes the α-backprojection in $O(n)$ steps, given the trapezoidation of P parallel to α.*

Proof. From the trapezoid containing a front edge, one can determine the next vertex encountered in constant time. The algorithm never sweeps through a trapezoid more than once, and spends constant time per trapezoid. □

3.2. Computing B_α from a triangulation. Because a triangulation can be converted into a trapezoidation in linear time [13], we could use the trapezoid algorithm of the previous section to compute an α-backprojection given a triangulation. A different algorithm is developed by Heffernan and Mitchell [20], who obtain a linear time bound using an approach that does not require a triangulation.

In this section we show how to implement the sweep of the trapezoid algorithm to compute B_α from a triangulation instead of a trapezoidation. The primary advantage of this algorithm is that it allows us to consider rotating α without the need to maintain a trapezoidation parallel to α. It also can be slightly more efficient—it can be implemented to compute B_α in time proportional to the number of triangles that intersect B_α. (Because we are anticipating the rotation algorithm, we describe an implementation that is suboptimal for computing B_α by a factor of $O(\log n)$.)

We sweep with free-space edges exactly as in the previous section. Without the trapezoidation, however, determining the next vertex that the free-space edge f hits is not so trivial. We define $L(f)$ to be the chain of left endpoints of the triangulation edges that intersect f. The order of the points along $L(f)$ is the order in which the corresponding edges cross f. Similarly, we define $R(f)$ to be the chain of right endpoints. Because the triangles that intersect f form a simple polygon, $L(f)$ and $R(f)$ are simple polygonal chains. The following lemma uses these chains to characterize the first vertex hit by a free-space edge.

LEMMA 3.2. *Let f be a free-space edge that touches P-edges t and b, which extend to the left of f, and let H be the convex hull of $L(f)$. No polygon vertex lies in the region bounded by f, t, b, and H. A similar claim holds for $R(f)$.*

Pos. of v	Action
Bottom of f	Delete v from $L(f)$ and add the left endpoints of edges that have v on their right.
Middle of f	Split $L(f)$ into two parts at v, remove v, and add to each chain the left endpoints of the appropriate edges that have v as their right endpoint.
Top of f	Remove v from $L(f)$ and add the left endpoints of the edges that have v on their right. If f extends above v, we add the left endpoints of the new edges that cross f. We also have to generate a new free-space edge f' that has v as its base.

Table 2

Proof. Suppose that a vertex v lies strictly inside the region. At least two triangulation edges are incident to v; these edges cannot cross other triangulation edges or f. If the edges incident to v all intersect the convex hull H, however, then v is a reflex vertex. Since a triangulation has no reflex vertices, no vertex exists in the region. □

Suppose that the free-space edge f moves left. (Moving right is handled symmetrically.) The first vertex v that f hits is the tangent to the hull H of $L(f)$. The algorithm maintains H using the *path hull* data structure that is described in detail in Section 6. Here it suffices to know that the structure maintains the hull of a polygonal chain and supports, in amortized logarithmic time, finding tangents to the hull, adding and deleting vertices from the ends of the chain, and spliting a chain or merging two subchains. (The split/merge bound does not allow intermixed sequences of splits and merges, but the algorithm does not require them.) All the path hulls in use at any time during the algorithm take only $O(n)$ storage altogether.

LEMMA 3.3. *Assuming that the appropriate path hull has been computed for a free-space edge, one can add trapezoids to the backprojection in amortized $O(\log n)$ time per intersected triangle.*

Proof. Let us go back to Figures 9–11, this time paying attention to the triangulation edges that cross f. When a leading free-space edge f slides to the left, the changes to $L(f)$ depend on the position of v along f, as described in Table 2.

We do similar operations to $R(f)$ for trailing free-space edges that slide to the right. Whenever a free-space edge moves and its chain changes, we compute the next event associated with it by updating the convex hull of the chain and finding the appropriate tangent to it. This takes logarithmic time per triangle intersected by B_α. □

This shows how to carry out the sweep. To begin it, erect the segment f vertically from g by walking through triangles and construct the chains $L(f)$ and $R(f)$ and their hulls.

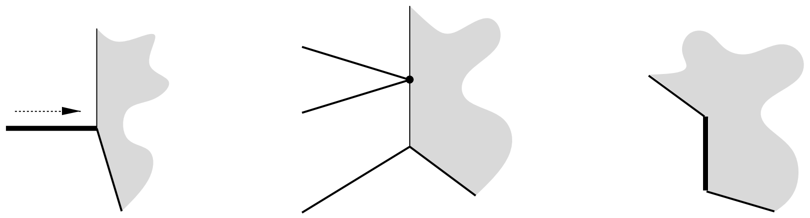
3.3. Events for the rotation algorithm. As mentioned at the beginning of Section 3, the core of the preprocessing phase is a rotation algorithm, which maintains the α-backprojection B_α while rotating α through 360°. In this section we describe the representation of B_α, the events at which it changes, and how to handle these events.

The rotation algorithm represents B_α as a sequence of edges and vertices. We define $seq(B_\alpha) = (a_1, a_2, \ldots, a_k)$ as follows. Initially, we set $seq(B_\alpha)$ to (), the empty sequence. Walking around B_α counterclockwise, starting and ending at the goal vertex

g, we append an element to $seq(B_\alpha)$ for each P-edge and for each P-vertex at the base of a free-space edge. For a P-edge e along the boundary of B_α, we append either $\underline{e}$ or $\overline{e}$, depending on whether e is a top or a bottom edge; for a P-vertex v at the base of a free-space edge, we append v. (If a bottom edge meets a top edge at a vertex, we introduce a null free-space edge at that vertex.) The polygon B_α, the trapezoid tree of B_α, and the pair $\langle \alpha, seq(B_\alpha) \rangle$ are all linear-time equivalent representations for the α-backprojection. We define events as those directions α for which $seq(B_{\alpha}-) \neq seq(B_{\alpha}+)$. The next lemma characterizes events.

LEMMA 3.4. *A direction α is an event if and only if one of the following conditions holds at this direction: (see Figure 12)*

- *a bottom edge of B_α ceases to be slidable, or a P-edge next to the base of a free-space edge becomes slidable into B_α (a "sliding event");*
- *a free-space edge encounters a P-vertex (a "visibility event");*
- *a top edge becomes a bottom edge, or a bottom edge becomes a top edge (a "turnover").*



a sliding event a visibility event a turnover

Fig. 12

Proof. If one of these events occurs, the backprojection gains or loses a vertex of the polygon, or a P-edge moves between the top and bottom of B_α. In either case $seq(B_\alpha)$ changes.

Conversely, suppose that we rotate α so that none of these conditions occurs—we argue that $seq(B_\alpha)$ does not change. Consider maintaining the vertical (that is, α) visibility graph as well as B_α. While the vertical visibility graph is unchanged, the trapezoid algorithm includes the same set of trapezoids in the backprojection (because no sliding events occur), so the sequence does not change. When the visibility graph changes, α is parallel to the line through two vertices. By assumption, the lower vertex is not the base of a free-space edge, so the higher vertex does not enter or leave the backprojection as we rotate; the vertices are not endpoints of a single P-edge, so no P-edge moves between the top and bottom of B_α. Thus $seq(B_\alpha)$ does not change. □

The rotation algorithm maintains the current direction α, the sequence $seq(B_{\alpha}+)$, and for each free-space edge f, a path hull for the left endpoints of triangulation edges that intersect f, as described in Lemma 3.2. Splits will be performed on path hulls for leading edges and merges on path hulls for trailing edges.

The algorithm uses a priority queue to determine the next event. The queue contains, for each P-edge in B_α or adjacent to it, the next direction at which the edge becomes or ceases to be slidable (for sliding events) and the direction (i.e. angle) of the P-edge itself (for turnover events). The queue also contains, for each free-space edge, the angle at which the edge hits the next vertex as α rotates (for visibility events). It follows from Lemma 3.4 that the event β at the top of the priority queue satisfies the condition $seq(B_{\alpha^+}) = seq(B_{\beta^-}) \neq seq(B_{\beta^+})$. At every step, the algorithm takes the next event off the top of the priority queue and computes the changes that have to be made to $seq(B_{\alpha^+})$ when α advances to β. In the process, the algorithm may discover and enqueue further events. The algorithm may also recognize that some events in the queue have become obsolete and dequeue them.

We now describe a single iteration of the rotation algorithm. The sequence changes during the transition from α^- to α^+ for the event α. Since α is fixed throughout the rest of the section, we define direction α to be vertically downward.

A turnover event does not involve any changes to the α-backprojection, and the changes to the sequence can be carried out in constant time. We also create or destroy a null free-space edge at the point where a top edge touches a bottom edge.

There are two kinds of events in which points are *added* to B_α: a sliding event involving a P-edge adjacent to the base of a leading free-space edge f, or a visibility event involving a leading free-space edge f. In both cases, we first rotate f to angle α and then call the procedure of Section 3.2 to add trapezoids to B_α that can now be reached because of the event on f. This may cause the scheduling of visibility events for new free-space edges, and turnover and sliding events for new P-edges.

There are two kinds of events that can cause B_α to *lose* trapezoids: a sliding event involving a bottom edge inside the α-backprojection, and a visibility event involving a trailing free-space edge. If we explicitly had a trapezoidation of B_α, then, in both cases we would need to cut a subtree off the trapezoid tree. Since we do not have the trapezoidation we need to run the algorithm of Section 3.2 in reverse to remove trapezoids from B_α.

In the case of a sliding event, suppose that bottom edge e just stopped being slidable towards its vertex u. We walk away from u on edges that slide towards u until we reach the base of a free-space edge f—all edges that we walk on will be removed from B_α. Then we rotate f to angle α, and begin sliding f in the direction of u. (Which is to the left, since f was a trailing edge.) Suppose that f encounters a vertex v—refer back to Figures 9–11 for the three cases. First, a base vertex v is easily handled as before. Second, a vertex v in the middle of f implies that anything that slides into v can no longer reach the goal. We process the upper P-edge into v as if it had a stop sliding event, moving free-space edges in until there is one based at v. Then we destroy that free-space edge, shorten f to end at v, and continue sliding f. Third, a vertex v at the top of f is easy to handle unless both incident P-edges go right. In that case, we essentially trigger a stop sliding event for the upper P-edge into v. When the portion of the backprojection that slid to v has been removed and there is a free-space edge based at v, then we merge that edge with f and continue sliding f.

In the case of a visibility event for a trailing free-space edge f, we rotate f to angle α, remove regions from B_α, if necessary, as described in the previous paragraph, construct the new free-space edge f, and continue.

In scheduling new visibility events, the algorithm must detect the first vertex hit by a rotating free-space edge. This can be done easily using path hulls as in Lemma 3.2. Each free-space edge represents the right and left chains by a path hull. However, only one of these chains is maintained per edge at any given moment: When a leading free-space edge is born, we generate its left chain. When a trailing free-space edge is born, we generate its right chain and use it to move the edge (and split it, if necessary, as described in Section 3.1) until we cannot move any further, at which time we disassemble its right chain and generate its left chain. When a leading free-space edge has to move to the right, we disassemble its left chain and generate its right chain. The following lemma shows that left and right motions are not intermixed. Thus we create the path hulls for $L(f)$ and $R(f)$ only once.

LEMMA 3.5. *A leading edge can move right only once, just before it disappears. A trailing edge can move right when it is created, but never again.*

Proof. If a leading edge moved to the right and then rotated counterclockwise, or if a trailing edge rotated and then moved right, then some points would enter the α-backprojection a second time, in violation of Theorem 2.12. □

3.4. The start and stop subdivisions. We now describe the start and stop subdivisions and their construction. The start and stop subdivisions encode the first and last angles for which points can reach the goal. Each subdivision consists of trapezoids and triangular sectors whose union is exactly the non-directional backprojection. Let p be a point in P. For the start subdivision, if p is in a trapezoid with parallel sides of direction α, then $start(p) = \alpha^+$ (see Figure 13); if p is in a sector and sees the base vertex of the sector in direction α, then $start(p) = \alpha^+$ (see Figure 14). For the stop subdivision, the corresponding direction is α^-. (In the special case in which the goal g is the base vertex of the sector containing p, then $start(p) = \alpha$ (or $stop(p) = \alpha$).)

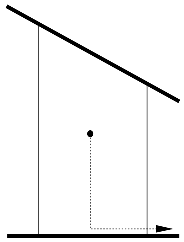


Fig. 13

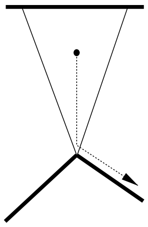


Fig. 14

Whenever the rotation algorithm moves a free-space edge, it adds regions to the start and/or stop subdivisions. When a leading edge slides left, or when a trailing edge slides right, it sweeps over trapezoids that are added to the start subdivision. Similarly, when a leading edge slides right or when a trailing edge slides left, it sweeps over trapezoids that go into to the stop subdivision.

If we rotate the direction α counterclockwise and the sequence $seq(B_\alpha)$ does not change, then each leading edge f sweeps out a triangular sector of points and adds them to B_α. A point p in the sector defined by f first enters the α-backprojection at the direction of the ray from p to the base of f. Trailing edges remove points from B_α. We include the sectors swept by leading edges in the start subdivision; those swept by trailing edges go into the stop subdivision.

In addition to α and $seq(B_\alpha)$, we maintain, for each free-space edge f, the direction of the edge at the last event involving f. Before the rotation algorithm moves a free-space edge laterally, we check if the current α is different from the direction associated with the free-space edge. If so, we "rotate" the free-space edge, adding the triangular sector to the appropriate subdivision.

Some start sectors may be generated by stop events (visibility events involving a trailing free-space edge, or stop-sliding events). This happens when a leading free-space edge disappears from the α-backprojection at a stop event, since we first create a start sector for such an edge.

3.5. Analysis of the algorithm. We now show that the running time of the rotation algorithm is $O(\tau \log n)$, where τ is the number of triangles that intersect the non-directional backprojection.

First, we show that $O(\tau)$ path hull operations are performed. When a triangulation edge enters or leaves the α-backprojection, we add its endpoints to or delete them from at most two hulls. Each vertex that enters the backprojection can cause a split; each one that leaves can cause a merge. Thus the total number of hull operations is $O(\tau)$; they take $O(\tau \log n)$ amortized time. As for space, there are $O(\tau)$ path hulls at any given moment, and each path hull can be of size $O(n)$; nevertheless, the total size of all the path hulls is $O(n)$ (this follows from Lemma 2.6). We explain in Section 6 a technique in which we allocate a fixed block of storage of size $O(n)$ for all the path hulls, and hence the working storage is $O(n)$.

Second, the total time spent on priority queue operations is also $O(\tau \log n)$. Since each P-edge enters and leaves the backprojection once, we perform a constant number of priority queue operations for each P-edge. Each time a hull changes, we compute a new tangent and add it to the queue; there are $O(\tau)$ such operations.

Finally, since each subdivision is composed of $O(\tau)$ triangles and trapezoids, we can preprocess them for point location in $O(\tau)$ time [9, 23]. At query time, we use two $O(\log \tau)$ point locations to find the regions of the start and stop subdivisions that contain the query point p. Using these regions we can determine the start and stop angles in constant time.

THEOREM 3.6. *Given a triangulated simple polygon P and a goal vertex or edge, we can build a data structure to support queries for one-step compliant motion inside P under perfect control. Queries take $O(\log \tau)$ time, preprocessing takes $O(\tau \log n)$ time and $O(n)$ space, and the data structure takes $O(\tau)$ space, where τ is the number of triangles that intersect the non-directional backprojection.*

4. Polygons with islands. Now we consider an environment P with $k - 1$ islands ($k \geq 1$), as described in Section 2.1. In this section, we show how to modify the simple polygon algorithm to handle the islands. The modified algorithm spends $O(kn \log n)$ time preprocessing the n-vertex environment P and the goal vertex g to produce a data structure with size $O(kn)$; at query time, the algorithm reports all the good directions for a given starting point p in $O(k \log n)$ time.

For any given α, the α-backprojection has the same structure as in the simple polygon case (since Lemma 2.4 does not rely on P being a simple polygon), namely, it has bottom, free-space, and top edges, and it can be computed from the trapezoids of the α-visibility graph. Also, the events that change the structure of B_α, represented by $seq(B_\alpha)$, are still the same, as stated in Lemma 3.4. Thus, we can use the rotation algorithm that was defined in Section 3 to maintain the B_α. The problem is that the good directions of a point p inside P may form several ranges, and hence as we rotate α, points may enter and leave B_α more than once. This means that the subdivisions of Section 3.4 are not subdivisions any more, but rather "piles" of possibly overlapping triangles and trapezoids. In order to build an efficient query structure for polygons with islands, we will separate each pile into layers so that every layer is indeed a subdivision.

While partitioning the trapezoids and triangles into layers leads to an efficient query structure, there is a less reductionistic way to think about the non-directional backprojection. Triangles and trapezoids of the start subdivision can be stitched together along their boundaries where a free-space edge moves from one region to another to form a simply-connected surface (due to Lemma 2.2) in space that only overlaps itself when projected to the plane. Thus, the start triangles and trapezoids give a subdivision of a subset of the universal covering space of the polygon [21]—the stop triangles and trapezoids give an alternate subdivision of the same subset. We concentrate on layers in this section, but the surface model simplifies some of the applications in Section 5.

To analyze the complexity of this amended algorithm, we need to bound the number of events that take place as we rotate α, the total size of the piles, and the storage requirements of the path hulls used during the rotation. The following lemma bounds the number of times a point can enter and leave the α-backprojection:

LEMMA 4.1. *For a given point p, there are at most k ranges that define all the good directions for p.*

Proof. Let the smallest good direction for p be α, and the largest good direction for p be ω. If the good directions for p form l ranges, then there are $l-1$ "bad" ranges between α and ω. Every such bad range must contain an island by Theorem 2.12. An island can appear in at most one bad range, since it can never be split by a trail. Therefore, $l-1$ cannot exceed the number of islands, which is $k-1$. □

We proceed as follows: in Section 4.1 we describe the layering scheme of the preprocessing phase and prove its correctness. In Section 4.2 we describe how to answer queries for the single-step, perfect control, perfect position sensing case using the data structures produced by the preprocessing phase. (Section 5 shows how to answer more realistic queries.) Finally, we analyze the complexity of this scheme in Section 4.3.

4.1. Layers for the start and stop subdivisions. Let us limit our attention to the pile that contains trapezoids and triangles that are *added* to B_α, known as the "start pile" (the treatment of the stop pile is symmetric). We need to specify the layering scheme that we use to partition this pile into subdivisions.

We use k layers, one for each of the k boundary components of the environment. Recall that a (non-turnover) event involves adding a triangular sector and zero or more trapezoids. The base vertex of a triangular sector belongs to a particular boundary component; we place the sector in the layer of that component. All the trapezoids go to the layer of the component where the *event* took place, as follows:[1]

- for a sliding event, when an edge next to B_α becomes slidable, we use the boundary component that contains that edge;
- for a visibility event, when a leading free-space edge encounters a vertex, we use the boundary component that contains the base vertex of the free-space edge.

Figure 15 illustrates the layering scheme. In this example we use idealized friction cones that are just the normals to the edges. The goal is the vertex at the bottom of the environment. On the left is the layer that corresponds to the outside polygon, and on the right is the layer that corresponds to the island. Note that some trapezoids that are based on the island go into the layer of the outside polygon, because the event in which they were created took place on the outside polygon. In order to prove the correctness of this scheme, we need to show that all the regions in any layer are disjoint. First we need the following lemmas:

LEMMA 4.2. *Suppose that p enters the α-backprojection at α^+ during an event associated with boundary component C, and suppose that $\beta > \alpha^+$ is also a good direction for p. Then C is outside $R_{\alpha^+,\beta}$.*

[1] We must treat sectors and trapezoids differently because some sectors in the start pile can be generated by stop events, as mentioned in Section 3.4.

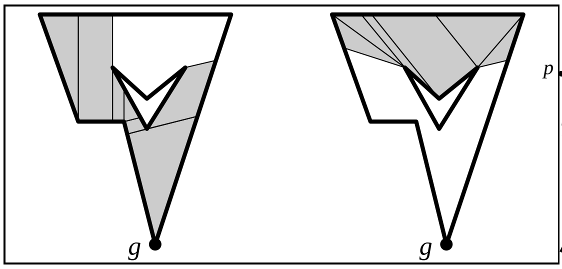


Fig. 15

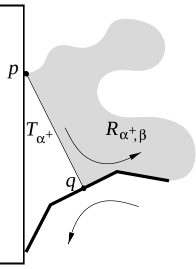


Fig. 16

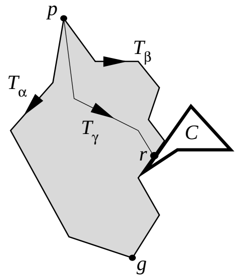


Fig. 17

Proof. We know that $p \notin B_\alpha$ and $p \in B_{\alpha+}$, so $T_{\alpha+}$ has almost the same prefix as T_α, but T_α either gets stuck or takes a wrong turn at a point q on C. The point q is arbitrarily close to a point r on C where $T_{\alpha+}$ goes through. See Figure 16.

Now, since we know that $T_{\alpha+}$ goes counterclockwise around $R_{\alpha+,\beta}$, the point q cannot be inside $R_{\alpha+,\beta}$. Hence the whole interior of C must lie outside $R_{\alpha+,\beta}$. □

Lemma 4.3. *Let p be a point in the environment, let $\alpha < \beta$ be good directions for p, and let C be a boundary component such that $R_{\alpha,\beta}$ does not contain C. Then for any $\alpha < \gamma < \beta$, the trail T_γ cannot get stuck on C at any point that is not g, the goal vertex.*

Proof. By Theorem 2.7, T_γ stays inside $R_{\alpha,\beta}$. The only way T_γ can get stuck on C is if C touches $R_{\alpha,\beta}$ where T_γ gets stuck. Suppose that T_γ gets stuck at a point r on the boundary of $R_{\alpha,\beta}$. This means that T_γ cuts $R_{\alpha,\beta}$ into two pieces: piece A is bounded by T_α and T_γ near p, and piece B is bounded by T_γ and T_β near p. If $r \neq g$, only one of these two pieces contains g. See Figure 17.

Suppose that A contains g. Again by Theorem 2.7, T_β stays in B up to r. But T_β does not get stuck at r, so it crosses to the "left" (to A) at that point. Because T_γ always slides left more easily than T_β, it cannot get stuck at r.

The other case is symmetric. □

Using these lemmas, we can now prove the correctness of the layering scheme:

Theorem 4.4. *For any layer, any two regions are disjoint.*

Proof. We prove the theorem by contradiction. Suppose that in the layer of boundary component C there are two regions that have a nonempty intersection, and suppose that p is in their intersection. One region implies α^+ as a start direction for p, and the other region implies β^+, where $\beta > \alpha$.

By Lemma 4.2, C is not contained in R_{α^+,β^+}. Since $\alpha^+ < \beta < \beta^+$, by Lemma 4.3 T_β cannot get stuck on C. However, p is in the region of β^+ because T_β *does* get stuck on C. This is a contradiction, and therefore no such point p exists. □

4.2. Queries in layers. When we are given a query point $p \in P$, we locate it in each layer of the start pile, and in each layer of the stop pile. Every region that contains p implies a corresponding direction in which p enters or leaves the α-backprojection (see Section 3.4), so when we sort the list of directions we end up with an alternating list of start-stop directions for p. This list specifies the good ranges for p, and this is the answer to the query.

4.3. Analysis. There are two quantities to consider, space and time. To help bound working storage, we have the following lemma:

Lemma 4.5. *The total size of all the path hulls at any given α is $O(kn)$.*

Proof. The total size of the path hulls is the sum, over all the free-space edges, of the number of triangulation edges and P-edges that cross each edge. Reversing the order of summation, this number is equal to the sum, over all the triangulation edges and P-edges, of the number of free-space edges that the edge crosses. There are $2n+3(k-2)$ triangulation edges and P-edges, and by Lemma 2.6, each such edge crosses at most $2k$ free-space edges. □

At any time during the algorithm, $O(kn)$ storage is needed for path hulls. Because individual path hulls may change their storage needs, statically allocating $O(kn)$ memory is not sufficient. Section 6 describes a dynamic storage allocation scheme that uses $O(kn \log n)$ storage altogether.

Obtaining the triangulation of a polygon with islands takes $O(n \log n)$ time [31, pp. 230–234] instead of the linear time possible in the simple polygon case. However, the total preprocessing time is dominated by the total number of trapezoids and sectors plus the number of turnover events (which do not contribute any regions to the pile, but have to be processed nevertheless), multiplied by the time that it takes to process each one.

LEMMA 4.6. *The total number of sectors and trapezoids in the start and stop piles is $O(kn)$.*

Proof. Each vertex of P enters B_α at most k times as α rotates. Every trapezoid or sector added to the start pile can be associated with one of these vertex entrances, and each vertex entrance has at most one trapezoid and one sector associated with it. A symmetric argument applies to the stop pile. □

Every P-edge can turn over at most twice, so there are $O(n)$ turnovers. Processing a trapezoid, a sector, or a turnover involves a path hull operation, and since the size of each path hull is $O(n)$, each operation takes $O(\log n)$ time. Therefore, the total preprocessing time is $O((kn+n)\log n) = O(kn \log n)$.

It is easy to keep track of the adjacency of the regions that we generate in each layer (since α rotates monotonically). All the regions in each layer are connected by adjacency to the island that is associated with that layer, so if we triangulate the island and add the triangles to the layer, we get a connected subdivision at no extra cost. We can preprocess this subdivision for point location using time and space linear in the size of the layer [9, 23].

Query answering involves $O(k)$ point location queries, which take $O(k \log n)$ total time, and sorting at most $2k$ numbers, which takes $O(k \log k) = O(k \log n)$ time.

THEOREM 4.7. *Given a triangulated polygonal environment P with n vertices and k islands, and a goal vertex or edge, we can build a data structure to support queries for one-step compliant motion inside P under perfect control. Queries take $O(k \log n)$ time, preprocessing takes $O(kn \log n)$ time and space, and the data structure takes $O(kn)$ space.*

5. Extensions to the main algorithm.

5.1. Imperfect control. The algorithms of Sections 3 and 4 create data structures to answer queries for the perfect control, perfect position sensing case. However, once we have formed these structures, we can easily answer more realistic queries, allowing imperfect control and imperfect position sensing.

The control uncertainty of a robot is an angular constant ϵ in the range $0 \leq \epsilon < 90°$. If the robot is programmed to move in direction α, its actual direction of motion (in free space) at any given instant lies in the range $[\alpha-\epsilon, \alpha+\epsilon]$. The path the robot follows now is not unique, although to simplify our proofs we will assume that it is piecewise differentiable. We say that a direction α is *good* for a starting point p if every path the robot may take, when commanded to move from p in direction α, is guaranteed to get the robot to the goal. We relate perfect and imperfect control in Lemma 5.2, but first we need the following lemma:

LEMMA 5.1. *Let γ be a direction, and call it "down." Given a control uncertainty $\epsilon < 90°$, let $\widetilde{T_\gamma}(p)$ be any imperfect-control trail starting at p with commanded direction γ, and let q be a point along $\widetilde{T_\gamma}(p)$ such that no sliding occurs between p and q. Let the length of $\widetilde{T_\gamma}(p)$ between p and q be L, and let the vertical distance between p and q be D. Then there is a constant c_ϵ, which depends only on ϵ, such that $L \le c_\epsilon D$.*

Proof. Since $\epsilon < 90°$, and no sliding occurs between p and q, that portion of the trail $\widetilde{T_\gamma}(p)$ is monotone with respect to γ. Therefore, the trail $\widetilde{T_\gamma}(p)$ can be described by a continuous, piecewise differentiable function $f(x)$, where x measures the vertical distance from p, and $f(x)$ measures the horizontal displacement (positive values to the right) from p. We know that

- $f(0) = 0$;
- $|f'(x)| \le \tan\epsilon$ almost everywhere.

Now we can bound the length L of $\widetilde{T_\gamma}(p)$ as follows:

$$L = \int_0^D \sqrt{dx^2 + df^2} = \int_0^D dx\sqrt{1 + |f'(x)|^2} \le \int_0^D dx\sqrt{1 + \tan^2\epsilon} = \frac{D}{\cos\epsilon}.$$

We set c_ϵ to $1/\cos\epsilon$, and the lemma follows. □

We are now ready to prove the lemma relating perfect and imperfect trails:

LEMMA 5.2. *Let p be in P, and let (α, β) be one of the (at most k) ranges of good directions for p for a robot with perfect control. Let the control uncertainty be $\epsilon < 90°$.*

- *if (α, β) is wider than 2ϵ, then any $\gamma \in (\alpha + \epsilon, \beta - \epsilon)$ is a good direction for the imperfect robot;*
- *if (α, β) is not wider than 2ϵ, then no $\gamma \in (\alpha, \beta)$ is a good direction for the imperfect robot.*

Proof. Let γ be a direction in the range $(\alpha+\epsilon, \beta-\epsilon)$. It follows from Theorem 2.12 that for any point r in $R_{\gamma-\epsilon,\gamma+\epsilon}$ the range $(\gamma - \epsilon, \gamma + \epsilon)$ is a good range under the perfect control model. Now, let $\widetilde{T_\gamma}$ be an imperfect control trail starting at p with commanded direction γ. We can prove by a case analysis similar to that of Lemma 2.10 that whenever $\widetilde{T_\gamma}$ touches the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$, it cannot cross outside of this region ($\widetilde{T_\gamma}$ cannot touch a free-space boundary edge; when $\widetilde{T_\gamma}$ touches a P-edge on the boundary, it is forced to slide in the same direction as the corresponding perfect-control trail; and when it separates from the boundary, it goes into free space inside the region).

We have to show that the robot indeed makes progress on its way to the goal. First, we consider the cases in which $\widetilde{T_\gamma}$ touches the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$. Clearly, the places where $\widetilde{T_\gamma}$ touches $T_{\gamma-\epsilon}$ are consistently sorted along both trails (because if r occurs before s along $\widetilde{T_\gamma}$, then $R_{\gamma-\epsilon,\gamma+\epsilon}(r)$ contains $R_{\gamma-\epsilon,\gamma+\epsilon}(s)$). Now, by the proof of Theorem 2.12, when $\widetilde{T_\gamma}$ travels along a portion of the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$, it slides in the same direction (never gets stuck, and never slides the wrong way, because both would imply that there is a direction in the range $(\gamma - \epsilon, \gamma + \epsilon) \subseteq (\alpha, \beta)$ that is not good for a point along the boundary of the region). Finally, whenever $\widetilde{T_\gamma}$ goes into free space, it has to hit the boundary again (this is because if r occurs before s along a free-space segment of $\widetilde{T_\gamma}$, then the area of $R_{\gamma-\epsilon,\gamma+\epsilon}(r)$ is strictly smaller than the area of $R_{\gamma-\epsilon,\gamma+\epsilon}(s)$). Since the lengths of $T_{\gamma-\epsilon}$ and $T_{\gamma+\epsilon}$ and the area of $R_{\gamma-\epsilon,\gamma+\epsilon}$ are all finite, $\widetilde{T_\gamma}$ has to reach g eventually.

Finally, we have to prove that we do not run into "Zeno's paradox," namely, that we do not have an infinitely long $\widetilde{T_\gamma}$. We use Lemma 5.1 to bound the length L of $\widetilde{T_\gamma}$ as follows:

$$L \leq c_\epsilon D + B + c_\epsilon B,$$

where D is the distance in direction γ from p to g, and B is the total length of the boundary of $R_{\gamma-\epsilon,\gamma+\epsilon}$ (that is, the sum of the lengths of $T_{\gamma-\epsilon}$ and $T_{\gamma+\epsilon}$). The first term in the bound accounts for the free-space distance; if $\widetilde{T_\gamma}$ goes farther than this in free space, it overshoots the goal. The second term accounts for the sliding. Finally, the third term accounts for the fact that some of the edges on which $\widetilde{T_\gamma}$ slides on may act as conveyor belts, and carry the robot back up a distance that the robot may later travel through free space. Hence the length of $\widetilde{T_\gamma}$ is finite, and this concludes the proof of the first part of the lemma.

As for the second claim, by definition, α and β are not good directions for p. If (α, β) is not wider than 2ϵ, then for any γ in (α, β) it is possible for the robot to follow T_α as its imperfect control path for the commanded direction γ, and therefore γ is not a good commanded direction for the imperfect robot. □

Using Lemma 5.2, it is clear how we can use the structure from Section 4 to answer queries under the imperfect control model. First, we find the good ranges under the perfect control model, as described in Section 4. Next, we increase every start direction by ϵ and decrease every stop direction by ϵ. Every range that stays nonempty is a good range under the imperfect control model. Note that we use ϵ only at query time, so the preprocessing is independent of the control uncertainty. This means that the same data structures can support several robots, with different capabilities, that operate in the same environment.

5.2. The boundary of the non-directional backprojection. Recall that the *non-directional backprojection* is the set of all points that can reach the goal in a single compliant motion. If the robot has perfect control, this set is the union of all the regions in the start pile (which is identical to the union of all the regions in the stop pile). When the control uncertainty ϵ is greater than 0, the non-directional backprojection is the set of all points that have at least one good range of size greater than 2ϵ.

We compute the boundary of the non-directional backprojection. Specifically, we compute a closed curve ν whose interior, as defined by its winding number, is exactly the non-directional backprojection. The curve ν is a simple Jordan curve on the universal covering space of the interior of P [21]. That is, the interior of ν always lies to the left of ν when ν is traversed counterclockwise, and ν contains no island of P in its interior. If $k = 1$ (the simple polygon case), then the universal covering space is just P itself, and ν is a simple curve made up of line segments and circular arcs. If $k > 1$, ν is not necessarily simple when projected down from the universal covering space to the plane.

The algorithm for computing ν is the same whether or not $k = 1$: we simply trace ν through the universal covering space. That is, we follow ν through the regions of the start and stop piles simultaneously. Whenever ν leaves one triangle or trapezoid, it enters an adjacent one, according to the adjacency relation when the two regions were created. The triangles and trapezoids of the start (stop) pile, stitched together by adjacency, form a subpolygon of the universal covering space.

To trace ν, we start a point p at g and move p counterclockwise along ν, walking through the start and stop piles simultaneously. At each step, the regions of the start and stop pile containing p dictate the line segment or circular arc that p must follow until it leaves one of the regions, as described below. The tracing algorithm adds this segment or arc to ν as it moves p. When p leaves a region, it crosses a free-space edge into an adjacent region.

Here are the three possibilities for the regions containing p, and the actions that the tracing algorithm takes in order to proceed to the next step. In each case, p moves so that the region locally to the left of ν has a good range larger than 2ϵ, as determined by the current regions of the start and stop piles, and the region locally to the right of ν does not (perhaps because ν follows the polygon boundary).

- If p lies in a triangular sector in both the start and stop pile, then ν follows a circular arc, the curve described by the apex of a fixed 2ϵ angle whose two rays pass through two fixed points (the sector bases). If the arc hits the boundary of P, ν follows that boundary as appropriate.
- If p lies in a sector in one pile and in a trapezoid in the other, then ν follows a radial line of the sector; ν may also follow the boundary of P if necessary.
- If p lies in two trapezoids, then ν follows the boundary of their intersection.

The tracing algorithm needs to go across free-space edges of the piles, which may involve changing layers. We can perform this operation in constant time if, during the preprocessing phase, we record the adjacency information of the pile regions as follows. The algorithm creates a region in either pile by rotating a free-space edge or by sweeping a free-space edge across a trapezoid. We maintain, for each free-space edge, a pointer to the region ρ of the α-backprojection immediately neighboring it; we use this pointer to doubly-link ρ with the new region over which the free-space edge sweeps, and then update the pointer to point to the new region.

The tracing algorithm takes time proportional to the number of times ν passes from one pile region to another. The following lemma shows that this quantity is $O(kn)$.

LEMMA 5.3. *Let ν be the boundary of the non-directional backprojection in the imperfect-control case. Then the tracing algorithm follows ν through each region of either pile at most four times.*

Proof. The start and stop piles are both subdivisions of a single connected portion of the universal covering space of the interior of P [21]. Viewing the piles from the perspective of the universal covering space, we see that two regions in the start and stop piles intersect if and only if they have a common point p and the trails from p to g determined by the two regions are homotopic (they wind around the islands the same way).

The tracing algorithm traces ν through the universal covering space; that is to say, the tracing algorithm is purely local, following the region adjacencies determined when the regions were created.

To enter or leave a region of either pile, ν must cross a free-space edge of the region. Since each pile subdivides part of the universal covering space, the regions of the other pile that intersect the edge are all adjacent.

Let p be a point on a free-space edge of the start pile that lies inside ν. Let α be the direction determined by the free-space edge. By Theorem 2.12 and Lemma 5.2, the direction $\alpha + \epsilon$ is good for every point in $R_{\alpha,\alpha+2\epsilon}(p)$, in particular for every point on the free-space edge between p and the base of the edge, and hence all those points lie inside ν. Thus ν intersects each free-space edge at most twice, possibly at the endpoints. Each pile region has at most four free-space edges, so the lemma follows. ◻

5.3. Uncertainty in initial position sensing. When the robot has imperfect position sensing, we may know its starting position only approximately. Formally, we know that the robot is somewhere in a (typically small) connected subset S of the environment, known as the *start region.* In this case, we need to find directions that are good for all the points in S simultaneously. Note that the answer to a query may be "unreachable" even when every point in S can reach the goal.

LEMMA 5.4. *Let S be a connected subset of the environment. If a direction α is good for all the points on the boundary of S, then α is good for all points in S.*

Proof. Let p be a point in S. If p is not a boundary point, then $T_\alpha(p)$ hits the boundary of S. ◻

The lemma means that we can find maximum and minimum angles by looking at the boundary of S. We use the following technique to find the intersection of the good directions of all the points along the boundary of S. First, we locate a point p on the boundary of S in each of the two piles. The good ranges of p correspond to at most k pairs of start/stop regions that intersect in the universal covering space as in Section 5.2. For each such pair, we trace the boundary of S in the start and stop subdivisions of the universal cover simultaneously, maintaining the interval between the "largest" start-angle α and the "smallest" stop-angle ω. We stop a tracing when we arrive back at p (and we abort a tracing, discarding the corresponding interval, if it reaches a free-space edge that bounds the non-directional backprojection). When we have completed all $O(k)$ tracings, we have generated the $O(k)$ good ranges for all the points on the boundary of S, and therefore all the points of S, under perfect control. We can shrink each such range by 2ϵ, as explained in Section 5.1, in order to obtain the result under control uncertainty.

Answering the initial query for p takes $O(k \log n)$ time, plus time proportional to the number of subdivision regions traversed during the $O(k)$ traces along the boundary of S, since we never visit the same region more than the number of times the boundary of S crosses it, and we spend constant time per triangle or trapezoid. For many "standard" starting regions, such as a disc (intersected with the environment) or a polygon with a constant number of sides, the number of triangles or trapezoids that intersect S is $O(kn)$ in the worst case, but is typically less.

5.4. Sensorless robots. We have so far implicitly assumed that the robot can detect the goal when it reaches it. If this is not the case (the robot is *sensorless* [8, 7, 11]), we can still use our data structures to answer queries.

A sensorless robot cannot detect the goal g, and so it must stick at g when it reaches it. This restricts the range of directions the robot can use; we denote the allowable range by $[\alpha, \omega]$. The range $[\alpha, \omega]$ contains the directions for which the robot cannot slide away from the goal on one of its incident edges. To answer a query for a sensorless robot with control uncertainty ϵ, with or without position uncertainty, we apply the appropriate algorithm described above, then intersect the query result with $[\alpha + \epsilon, \omega - \epsilon]$.

Once again, we can compute the boundary of the non-directional backprojection for sensorless robots: we trace along the boundary while holding an angle fixed. Whereas in Section 5.2 the angle was the start/stop range, here it is the intersection of that range with $[\alpha, \omega]$.

5.5. Convex polygonal goal regions. By minor modifications to the rotation algorithm, we can handle convex polygonal goal regions instead of single vertices or edges. We leave more general goal regions as an open problem.

To compute the α-backprojection of a convex goal polygon G we could decompose the difference $P \backslash G$ into trapezoids by the α-visibility map and then start the trapezoid algorithm of Section 3.1 with the set of trapezoids whose bottom edge are on G. To apply the rotation algorithm to maintain this α-backprojection, we need to add *turnover events* for the edges of G to the priority queue. By a slight abuse of notation, we say that a bottom-to-top turnover occurs when the rotating direction α goes from negative to positive projection on the normal to an edge of G; a top-to-bottom turnover occurs when the projection goes from positive to negative. (See Section 3.3).

Since G is assumed convex, top-to-bottom turnovers occur only at the leftmost edge, with α vertically downward. Suppose that b was the leftpoint point of G until the turnover for edge (a, b) made a the leftmost point. We handle this turnover as if it was a sliding event from b to a—we create a sector for the free-space edge that was based at b, slide the free-space edge to a, and add the left endpoints of triangulation edges whose right endpoint is a to the path hull for the free-space edge. Bottom-to-top turnovers occur only at the rightmost edge and can be handled by deleting triangulation edges from the path hull. The analysis of Section 4.3 still holds if n is changed to the total number of vertices in G and P.

5.6. The multi-step problem. A robot in an arbitrary polygonal environment may not be able to reach a goal g in a single step (this corresponds to the fact that there may be points p for which no direction is good). In this section, we consider the multi-step problem for the following cases:

- A simple-polygon environment ($k = 1$), perfect control and goal sensing. We modify the algorithm of Section 3 to handle multi-step planning within the same time and space bounds, namely, $O(n \log n)$ preprocessing time, linear-size data structures, and $O(\log n)$ query time. One can easily see that in this case, a robot can always reach g in n steps.
- Arbitrary k, sensorless (with control uncertainty), and with plans restricted to be of the form $(g_0, \alpha_1, g_1, \alpha_2, g_2, \cdots, \alpha_m, g_m)$, where $g_0 = p$, $g_m = g$, g_i is either a vertex or an edge, and any point in g_{i-1} reaches g_i via α_i. Our solution runs in $O(kn^2 \log n)$ preprocessing time, $O(kn^2)$ space, and $O(kn \log n)$ query time.

Consider the multi-step problem inside a simple polygon under perfect control and perfect goal sensing. Suppose that we computed a non-directional backprojection B from g that did not include the whole polygon P. Since B is composed of triangles and trapezoids bounded by P-edges and free-space edges, the unreached portions of P must be *bays* bounded by free-space edges, as illustrated in Figure 18.

A robot starting inside a bay bounded by a free-space edge f escapes the bay if it reaches f. We would like to run the rotation algorithm on the bay with f as the goal. However, we must first fix up the triangulation at the mouth of the bay. We need to triangulate the polygon defined by f and its left chain (or right chain). This polygon is weakly visible from f, and hence it can be triangulated in linear time by a simple algorithm [34].

Note that all the points in the triangles cut by f can reach f. At the next step, these triangles are strictly inside the non-directional backprojection of f and thus can never be cut again. We run the algorithm of Section 3 on each bay, perhaps forming smaller bays. We repeat the process until every point is in some backprojection. Since the rotation algorithm runs in $O(\tau \log n)$, and every triangle is in at most two bays,

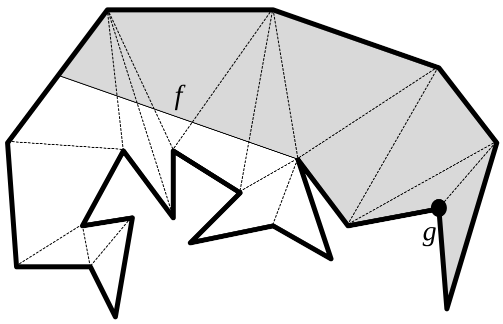


Fig. 18

the total time is only $O(n \log n)$.

With each trapezoid and triangle in the start and stop subdivisions, we store the free-space edge that is the subgoal and the number of steps to reach g. After preprocessing for point-location, we can, in $O(\log n)$ time, answer queries about the number of steps a point requires to reach g. If the robot is capable of stopping when it reaches a free-space edge f, then we can list the directions and goals for the entire path by looking at the direction and goal of each successive edge subgoal.

Solving the multi-step planning problem with goal sensing and imperfect control is complicated by the fact that the boundary of the one-step non-directional backprojection contains circular arcs. For sensorless robots with imperfect control, however, we can still say something about the multi-step planning problem. Since the robot is sensorless, we take all subgoals to be vertices or edges—after the first step, the robot moves from one vertex or edge of P to another. This suggests the following graph-based approach. In $O(kn^2 \log n)$ time and $O(kn^2)$ space, compute the non-directional backprojection of each P-vertex and each P-edge. Now we have $O(n)$ start and stop piles, and we can answer queries with each P-vertex or P-edge as a goal. Using these piles, we construct a directed graph. The nodes are the P-vertices and P-edges, and there is an arc from v_1 to v_2 if and only if v_1 can reach v_2 by a single step. For each goal v_2, we compute all arcs into it in $O(kn)$ total time, visiting each region in the start and stop piles for v_2 a constant number of times. We perform a breadth-first search on the graph starting from g, marking every node with its distance from g in the graph. We use this graph to solve the multi-step planning problem for sensorless robots. Given a query point p inside P, we find a P-vertex or P-edge that is reachable from p and is closest to g in the graph by performing at most n queries. This takes $O(kn \log n)$ time.

Note: In this case g does not have to be fixed: we simply defer the breadth-first search to the query phase, so the total time required for a query increases to $O(n^2 + kn \log n)$.

6. The path hull data structure. In this section we describe path hulls, the convex hull representation that the rotation algorithm of Sections 3 and 4 uses to find the non-directional backprojection in $O(kn \log n)$ time. These path hulls are an extension of a data structure due to Dobkin et al. [6]. A *path hull* represents the convex hull of a simple polygonal path π, and consists of two separate *semipath hulls*; the path π is the concatenation of two portions, and each semipath hull represents the convex hull of one of the two portions. The convex hull representation is simple: each semipath hull stores the vertices of the convex hull of its portion in an array.

This array supports binary search, and hence supports logarithmic-time intersection finding and tangent finding on the hull.

Path hulls support the operations required by Sections 3.2 and 3.3. In particular, for a path π with m vertices and convex hull h, a path hull that stores π and represents h supports the following operations:

1. Find tangents to h, either from a point outside h or parallel to a given line, in $O(\log m)$ time.
2. Add a vertex to either end of the path π (as long as the resulting path is simple) and update the hull in $O(\log m)$ time.
3. Delete a vertex from either end of the path and update the hull in constant amortized time, but worst-case $O(m)$ time.
4. Split π at a vertex to get two subpaths, each including the splitting vertex, and compute the convex hulls of each subpath in $O(m)$ worst-case time.
5. Merge two paths, disjoint except for a single vertex, into a single path of m vertices, and compute its convex hull in $O(m)$ worst-case time.

The worst-case time bounds quoted above are not particularly attractive; what makes path hulls useful is the amortized time bounds on intermixed sequences of these operations. In particular, if we perform an intermixed sequence of $O(m)$ operations of types 1, 2, 3, and 4 on paths with a total of m vertices, the total time required is $O(m \log m)$. The same bound holds for an intermixed sequence of $O(m)$ operations of types 1, 2, 3, and 5. We can obtain these amortized bounds because we use a pair of semipath hulls to represent each path; we balance the lengths of the two portions of the path to bound the average complexity of the operations.

The remainder of this section gives the details of path hulls. We first describe and analyze semipath hulls, from which we build path hulls. Next we tell how to implement path hulls based on the underlying semipath hull structures, characterizing precisely the operations that path hulls support. Finally, we use this characterization to prove the amortized time bounds quoted above. We defer until Section 6.4 a description of how to allocate storage for semipath hulls in our backprojection computation.

In what follows, we speak of simple paths as being directed. If π is a simple path, we distinguish between its endpoints by calling one end the *anchor* of the path and the other end the *free end* of the path. The path is directed from the anchor to the free end.

6.1. Semipath hulls. The semipath hull of a simple polygonal path represents the convex hull of the path by storing the hull vertices in a linear array; it allows vertices to be added to or deleted from the free end of the path. We use the notation $SPH(\pi, x)$ to denote the semipath hull of a simple path π anchored at x. (We have changed the nomenclature of Dobkin et al. slightly: what we call semipath hulls were called path hulls in that paper [6].)

For a simple path π with anchor x and free end v', the counterclockwise sequence of the vertices on its convex hull is a subsequence of $v' \ldots x \ldots v'$, the concatenation of the path and its reversal. The semipath hull stores the sequence of hull vertices in a deque (double-ended queue). The convex hull vertex closest to the free end of the path, call it v, appears at both ends of the deque. The subsequence property ensures that adding vertices to or deleting vertices from the free end of the path affects only the ends of the deque.

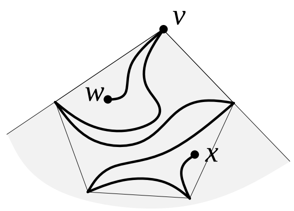


Fig. 19

When vertices are added to or deleted from the free end of the path, the semipath hull updates the deque. The update method is based on Melkman's incremental algorithm for finding the convex hull of a simple path [28], which we now describe. Each step of Melkman's algorithm adds a new vertex to the simple path, then finds the convex hull of the resulting path. As above, let v be the convex hull vertex closest to the old free end, and let w be the new free end being added. If w lies inside the angle formed by the two hull edges incident to v, then it lies inside the convex hull and does not change it. If w lies outside the old hull, the algorithm uses a Graham scan [17] to pop zero or more vertices off each end of the deque, then pushes the new vertex onto both ends of the deque.

A semipath hull consists of two data structures: a deque and a "transcript" stack. The deque contains the vertices of the convex hull, as described above, and the transcript stack records the push and pop operations needed to construct the current deque from scratch when path vertices are added one at a time. To add a vertex to the path, we use Melkman's algorithm to update the deque, and then record the update operations on the transcript stack. To delete a vertex, we play the transcript backwards: we pop operations off the transcript stack and perform their inverse operations until the deque reaches the state that existed before the vertex was added.

Allocating storage for the semipath hull is easy if we know *a priori* the maximum number of vertices that will ever belong to the path. If the upper bound on the number of vertices in the path is m, then we allocate an array of $2m - 1$ memory locations for the deque. We put the anchor vertex at the middle of the array; because there are at most $m - 1$ pushes on either end of the deque, the deque cannot overflow its array. Similarly, we allocate a block of $O(m)$ storage for the transcript stack, since the total number of operations to create the deque is at most $4m$.

The approach described so far may spend $O(m)$ time per vertex addition or deletion, since either operation may require $O(m)$ deque pops or pushes. We can avoid this overhead by using the array to do lazy pops. When we add a new vertex w, we find the tangents from w to the old convex hull using increasing-increment binary search (see below) on the array. This tells us which vertices Melkman's algorithm would pop off the deque. Instead of popping the vertices explicitly, we change the array indices that delimit the ends of the deque, as if the vertices had been popped off. When we push w onto the ends of the deque, the transcript remembers the vertex that was overwritten. To delete w and restore the deque to its former state, we simply replace w with the vertex that it overwrote, then change the deque-delimiting indices back to their previous values. Because implicitly popped array locations are altered only by reversible pushes, no information is lost during restoration. By using lazy popping, we reduce vertex addition time to $O(1)$ plus the time needed to find the tangents, and reduce vertex deletion time to $O(1)$.

We use a variant of binary search to find the tangents from the point w to the old hull. Ordinary binary search on a hull with k vertices takes $O(\log k)$ time. By using "increasing-increment" search, we reduce the search time to logarithmic in d, where d is the number of vertices that are popped off the hull by the addition of w. This form of binary search borrows the fundamental idea of finger search trees [22], but incurs little of the complexity of that data structure. We search in the array for the tangent vertices, starting at the ends of the deque (at v) and working toward the middle. Consider the search inward from the left end of the deque. By performing a constant-time test, we can tell whether the tangent vertex lies in the array interval between a query vertex and the left end of the deque. We start with the query vertex adjacent to the left end of the deque, then double the distance (measured in array indices) between the query vertex and the left end of the deque until the tangent vertex lies in the interval. We then use ordinary binary search on the interval to find the tangent vertex. The cost of the search is proportional to the logarithm of the interval size, that is, the logarithm of the number of vertices to be popped off on the left end of the deque. If d is the total number of vertices popped off by the addition of w, the cost of adding w is $O(\log(d+2))$. (The extra 2 allows for the case of no pops at all.)

Because a semipath hull stores the convex hull vertices in an array, it supports binary search algorithms to find tangents and intersections in logarithmic time. Thus we have established the following theorem.

THEOREM 6.1. *Let π be a directed simple path that is subject to addition and deletion of vertices at the free end of the path. If it is known beforehand that π will never have more than m vertices, then we can represent the convex hull of π in a semipath hull that supports (1) finding tangents to the hull, either from points outside the hull or parallel to given lines, in $O(\log m)$ time, (2) vertex addition at the free end of the path in $O(\log(d+2))$ time, where d is the number of vertices the new vertex removes from the convex hull, and (3) vertex deletion from the free end in constant time.*

6.2. Path hulls. We now discuss path hulls and their use of semipath hulls. A path hull for a path π distinguishes some vertex x of π. The vertex x divides π into two subpaths π_1 and π_2; the path hull stores $SPH(\pi_1, x)$ and $SPH(\pi_2, x)$. We use the notation $PH(\pi, x)$ to denote the path hull for π with distinguished vertex x. To make deletions at both ends of π easy, we want x to be close to the middle of π. However, repeated deletions or additions at one end may unbalance the path hull. We use amortization arguments (Theorems 6.2 and 6.3) to control the cost of imbalance.

At the beginning of Section 6 we summarized the operations that path hulls support. Now let us describe those operations in more detail, using Theorem 6.1 to bound their running times. We use h, h_1, and h_2 to denote the convex hulls of π, π_1, and π_2, and m to denote the number of vertices of π.

1. To find the tangents to h, either from a point outside h or parallel to a query line, we find the tangents to h_1 and h_2 separately, then choose the tangents to h from these four tangents. This takes $O(\log m)$ time.
2. To add a vertex to π, we add it to the semipath hull for π_1 or π_2, as appropriate. This takes $O(\log(d+2))$ time, where d is number of vertices popped from the appropriate semipath hull deque.
3. To delete a vertex w from π, we would like to delete it from exactly one of π_1 and π_2. If w is not equal to x, the middle vertex of the path hull, then deletion takes $O(1)$ time. If $w = x$, then we must rebuild the path hull, since one of the semipath hulls does not support deleting x. Suppose that π_1 is x. Then we find the middle vertex of $\pi_2 = \pi$, call it z, and build the path hull $PH(\pi, z)$, constructing the two semipath hulls by adding vertices. It takes $O(m)$ time to disassemble $SPH(\pi_2, x)$ by deleting all its vertices and then to build the two new semipath hulls. (The cost of building is $O(\sum_i \log(d_i+2))$,

where d_i is the number of vertices popped off at the i^{th} step of construction. Because $\sum_i d_i$ is at most $2m$, the construction cost is $O(m)$.) Disassembly and reconstruction also require $O(m)$ temporary storage for the vertices of π.

4. To split π at a vertex $y \in \pi_1$, we delete vertices of π_1 until y becomes the free end of π_1. Let the deleted path be π_1', let its length be m_1', and let its midpoint be z. We build the path hull $PH(\pi_1', z)$ as in the previous case. These operations take $O(m_1')$ time and temporary storage.
5. To merge two paths π_1 and π_2, we disassemble one path hull and add it to the other. Suppose that π_1 and π_2 are disjoint except for a single vertex. Let the lengths of π_1 and π_2 be m_1 and m_2, and without loss of generality assume that $m_1 \leq m_2$. We disassemble the path hull for π_1 by deleting all its vertices, then add the vertices to the path hull for π_2. This takes $O(m_1)$ time for the disassembly and $O(m_1)$ temporary storage. The time for reassembly is $O(\sum_i \log(d_i + 2))$, as in 3 above, but now $\sum_i d_i \leq 2(m_1 + m_2)$, and so the total time for merging is $O(m_1 + m_1 \log(m_2/m_1)) = O(m_1 \log(1 + m_2/m_1))$.

6.3. Analysis. Now that we have characterized the operations that we perform on path hulls, we use the characterization to bound the time required by sequences of the operations. We consider intermixed sequences of operations 1, 2, 3, and 4 or 1, 2, 3, and 5. Our time bounds are amortized: we bound the time for the whole sequence of operations, but not for individual operations. The proofs use *potential functions* to amortize the costs of the various operations; that is, we prove that for each operation, the actual running time plus the change in the potential function is $O(\log m)$. Because the potential function for sequences that include splits is different from the one for sequences that include merges, we cannot give a good bound for the time required by an intermixed sequence that includes both splits and merges. An alternating sequence of m splits and merges may take $\Theta(m^2)$ time.

THEOREM 6.2. *Given a collection of path hulls and singleton vertices that contains a total of m vertices, we can perform an intermixed sequence of $O(m)$ tangent computations, vertex additions, vertex deletions, and path splits in $O(m \log m)$ total time.*

Proof. The preceding characterization of path hull operations gives the asymptotic complexity of each. To bound the complexity of a sequence of such operations, we must be more exact in our accounting. We assign a precise cost to each operation, then define a potential function related to the costs. We define the cost of a tangent computation to be $\log m'$ for a path hull with m' vertices. The cost of a vertex addition is $\log(d+2)$, where d is the number of vertices popped off the semipath hull by the new vertex. The cost of a deletion is either 1 in the easy case, or m' in the case when a path hull with m' vertices must be rebuilt. The cost of a split is k, where k is the size of the path hull that is rebuilt.

We define a potential function on the collection of path hulls. For a single path hull $PH(\pi, x)$, let m' be the length of π and let m_1 and m_2 be the lengths of the two portions of π produced by cutting at x, so $m_1 + m_2 = m' + 1$. We define the potential of $PH(\pi, x)$ to be

$$2m' \log m' + |m_1 - m_2|.$$

The first term of the potential is used to pay for splitting costs, and the second to pay for rebalancing when necessary. The coefficients are chosen to balance the two terms against each other for the splitting operation. The potential of the whole collection of path hulls, which we call Φ, is the sum of the individual potentials. The *amortized* *cost* of an operation is its true cost plus the change in potential, $\Delta\Phi$. We show that the amortized cost of each operation is $O(\log m)$. Notice that Φ is nonnegative and bounded above by $2m\log m + m$; if we start with a collection of path hulls whose potential is nonzero, we are still assured that the total time spent on a sequence of $O(m)$ operations is $O(m \log m)$.

For a tangent computation, $\Delta\Phi$ is zero, and so the amortized cost is the same as the true cost, $\log m' = O(\log m)$.

For a vertex addition to a semipath hull with m' vertices, $\Delta\Phi$ is at most

$$\begin{aligned}
&2(m'+1)\log(m'+1) - 2m'\log m' + 1\\
&\qquad = 2\log(m'+1) + 1 + 2m'\log(1+1/m')\\
&\qquad = 2\log m' + O(1).
\end{aligned}$$

The amortized cost is $O(\log m)$.

For a vertex deletion there are two cases. If no rebuilding is needed, then $\Delta\Phi$ is at most

$$\begin{aligned}
&2(m'-1)\log(m'-1) - 2m'\log m' + 1\\
&\qquad = -2\log(m'-1) + 1 + 2m'\log(1-1/m')\\
&\qquad = -2\log m' + O(1)\\
&\qquad = O(1).
\end{aligned}$$

The amortized cost is certainly $O(\log m)$. If the path hull must be rebuilt, then the second term of the potential becomes significant. The actual cost is m', and $\Delta\Phi$ is $-2\log m' + O(1) - m'$, so the amortized cost is the same as in the first case.

For a split, the actual cost is k, where k is the size of the path hull that must be rebuilt. There are two cases depending on whether k is more or less than half of m', the length of the path being split. If $k \le m'/2$, then the first term of the potential gives what we want: $\Delta\Phi$ is at most

$$\begin{aligned}
&2(m'-k+1)\log\frac{m'-k+1}{m'} + 2k\log\frac{k}{m'} + 2\log m + k + 1\\
&\le 2\log m + 1 - k.
\end{aligned}$$

If $k > m'/2$ then the second term comes into play. The worst-case upper bound on $\Delta\Phi$ occurs when π_1, the semipath containing the splitting vertex, has size k. $\Delta\Phi$ is at most

$$\begin{aligned}
&2(m'-k+1)\log\frac{m'-k+1}{m'} + 2k\log\frac{k}{m'} + 2\log m +\\
&\qquad 1 + (m'-k) - (2k - m' - 1)\\
&\le 2\log m - 2(m'-k) - 3k + 2m' = 2\log m - k.
\end{aligned}$$

In both cases the amortized cost is $O(\log m)$. This completes the proof of the theorem. □

THEOREM 6.3. *Given a collection of path hulls and singleton vertices that contains a total of m vertices, we can perform an intermixed sequence of $O(m)$ tangent* *computations, vertex additions, vertex deletions, and path merges in $O(m \log m)$ total time.*

Proof. The idea of this proof is the same as that of the previous proof: we specify the costs of the operations precisely, define a potential function bounded by 0 and $2m \log m + m$, and show that the amortized cost of each operation is $O(\log m)$. As in the previous proof, the operation costs are $\log m'$ for tangent-finding, $\log(d+2)$ for a vertex addition, and either 1 or m' for vertex deletions, depending on whether path hulls must be rebuilt. The cost of merging two paths to produce a single path of length m' is $k \log(m'/k)$, where k is the length of the shorter of the two paths.

We define the potential of a single path hull that has two portions of lengths m_1 and m_2, with $m_1 + m_2 = m' + 1$, to be

$$2m' \log(m/m') + |m_1 - m_2|.$$

The potential for the whole collection of path hulls, which we call Φ, is the sum of the individual potentials. It satisfies $0 \le \Phi \le 2m \log m + m$.

As in Theorem 6.2, the amortized cost of a tangent computation is the same as its true cost, $\log m' = O(\log m)$. Similarly, the amortized costs of vertex additions and deletions are $O(\log(m/m')) = O(\log m)$.

For a merge, $\Delta\Phi$ is at most

$$\begin{aligned} &2(m' - k + 1) \log \frac{m' - k + 1}{m'} + 2k \log \frac{k}{m'} - 2 \log \frac{m}{m'} + k - 1 \\ &\le -2 \log \frac{m}{m'} + 2k \log \frac{k}{m'} + k \le -k \log \frac{m'}{k}. \end{aligned}$$

Hence the amortized cost of a merge is $O(1)$. □

6.4. Storage allocation for semipath hulls. In this section we discuss how to allocate storage for the semipath hulls used in the algorithms of Sections 3 and 4. In order to represent the convex hull of a simple path by a semipath hull, we must allocate a block of storage proportional in size to the length of the path. During the rotation algorithm, the paths lengthen and shorten due to vertex additions and deletions. In the face of changing path lengths, we must ensure that each semipath hull has a block of storage whose minimum size is proportional to the length of the path. We describe three solutions to the allocation problem; these solutions trade conceptual complexity for space complexity.

The first solution is trivial: simply allocate $O(n)$ storage for each semipath hull. Each semipath hull has enough space; most have an excess. This scheme takes $O(n^2)$ storage.

The second solution is less trivial: copy semipath hulls when they get too big or too small. This is the approach used when $k > 1$, that is, when there are islands inside the outer polygon (Section 4). Each semipath hull $SPH(\pi, x)$ is initially given a block of storage appropriate for a path at most two times longer or shorter than π. Suppose that the block is appropriate for a path of length m. When the length of π becomes less than $m/4$ or greater than m, we copy the semipath hull into a block of storage half or twice as big, as appropriate. The copying cost is proportional to the number of vertex additions and deletions performed on the semipath hull since the last copying operation, and so the copying does not increase the asymptotic complexity of the semipath hull operations. A straightforward scheme for block allocation uses $O(kn \log n)$ space: for each i between 0 and $\log n$, we allocate $O(kn)$ storage in blocks of size $O(2^i)$. Because there are only $O(kn)$ path vertices total, and no path is larger than $O(n)$, there are always enough blocks of the required sizes.

The third solution reduces the storage to $O(n)$ in the simple polygon case ($k = 1$). This is the trickiest solution of the three: allocate three linear-size arrays to hold all the semipath hull data, then assign disjoint subarrays to the different semipath hull data blocks. Making sure the blocks stay disjoint as the semipath hulls change is the key to this approach.

The description of this solution is divided into three parts: keeping semipath hulls stationary, sharing a deque between two arrays, and indexing the semipath hull storage.

Section 6.1 shows that a semipath hull changes very little when a vertex is added or deleted. No vertices are moved except the one being added or deleted. Thus a requirement of a storage allocation scheme is that if π' is a subpath of π with the same anchor x, then the storage for $SPH(\pi', x)$ should be a contiguous subset of that for $SPH(\pi, x)$. Because the paths we deal with are subsequences of the polygon boundary, this suggests that we number the vertices around the boundary of the polygon and assign storage based on indices. A path with endpoints numbered i and j along the boundary of P would be assigned storage with indices $i..j$. This is the idea we use, although the indexing is not quite so simple.

Before we discuss indexing, we consider storing a deque in an array fixed at one end. The deque part of $SPH(\pi, x)$ grows at both ends as new vertices are added to π, but the array locations reserved for it extend in only one direction: the location assigned to vertex x is fixed. To allow double-ended growth, we use two arrays, *right* and *left*. We split the deque array described in Section 6 into a left half and a right half. If locations $x..v$ are reserved for the deque, then we store the right half of the deque in *right*$[x..v]$, growing toward v, and the left half in *left*$[x..v]$, also growing toward v. This complicates the deque indexing, but has the properties we need. The transcript stack is easy to manage; we just store it in a transcript array in locations indexed by $x..v$, with the stack growing toward v.

The array indexing scheme we use is not vertex-based, but edge-based. In the algorithm of Section 3, we maintain a semipath hull for each free-space edge on the boundary of B_α. If each of the associated paths lay outside B_α, in the bay cut off by the free-space edge, then assigning storage based on polygon vertex indices would work—path vertices would lie in disjoint intervals of the polygon boundary. However, the path associated with a free-space edge doesn't always lie in the bay cut off by the edge, for example when a leading free-space edge is retreating (moving to the right). To cope with this difficulty, we allocate the storage for a semipath hull based on the polygon and triangulation edges that cut the free-space edge. We assign indices 1 through $4n - 6$ to the endpoints of all polygon and triangulation edges, numbered consecutively along the polygon boundary. Semipath hulls are stored in three arrays with these indices. The path associated with a free-space edge f has as many vertices as there are polygon or triangulation edges that touch f from a particular side of the path. This number is no larger than the number of edge endpoints inside the bay that f cuts off, and so we can allocate the storage for the semipath hull from that associated with these edge endpoints. In particular, the triangulation or polygon edge from f to the anchor of the path determines the fixed end of the semipath hull storage.

7. An open problem. In the standard model of compliant motion [1, 4, 7, 8, 10, 24], a vertex of the environment can be "sticky" when the robot encounters it coming from free space. In reality, however, if the robot "shakes" a little, it may get unstuck and reach the goal. In other words, it is possible that this vertex separates two good ranges for the starting point of the robot. Under the model in which the robot is able to "shake loose" when it gets stuck, it may be possible to unite these two ranges. This can produce a good range for the imperfect control case, even when no good range exists under the standard model. Figure 20 illustrates the problem: the presence of island I prevents the merging of the two good ranges for p, but if island I were not there, merging would be possible. We conjecture that it is possible to merge ranges, given the control uncertainty of the robot, in $O(kn \log n)$ preprocessing and $O(k \log n)$ query time.

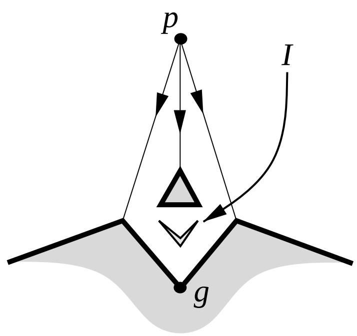


Fig. 20

Acknowledgements. We would like to thank Jean-Claude Latombe for introducing us to the problem of compliant motion planning, and to Leo Guibas for reading an early draft of this paper and for making important comments about Theorem 2.7.

REFERENCES

[1] A. J. Briggs, *An efficient algorithm for one-step planar compliant motion planning with uncertainty*, Algorithmica, 8 (1992), pp. 195–208.

[2] S. J. Buckley, *Planning and Teaching Compliant Motion Strategies*, PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[3] J. F. Canny, *A new algebraic method for robot motion planning and real geometry*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, IEEE, 1987, pp. 39–48.

[4] J. F. Canny and J. Reif, *New lower bound techniques for robot motion planning problems*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 49–60.

[5] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete and Computational Geometry, 6 (1991), pp. 485–524.

[6] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink, *An efficient algorithm for finding the CSG representation of a simple polygon*, Algorithmica, 10 (1993), pp. 1–23.

[7] B. R. Donald, *Error Detection and Recovery in Robotics*, Springer-Verlag, 1989. Lecture Notes in Computer Science 336.

[8] ———, *The complexity of planar compliant motion under uncertainty*, Algorithmica, 5 (1990), pp. 353–382.

[9] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM Journal on Computing, 15 (1986), pp. 317–340.

[10] M. A. Erdmann, *Using backprojections for fine motion planning with uncertainty*, International Journal of Robotics Research, 5 (1986).

[11] M. A. Erdmann and M. Mason, *An exploration of sensorless manipulation*, in IEEE International Conference on Robotics, 1986, pp. 1569–1574.

[12] S. Fortune and G. Wilfong, *Planning constrained motion*, in Proceedings of the 20th ACM Symposium on Theory of Computing, 1988, pp. 445–459.

[13] A. Fournier and D. Y. Montuno, *Triangulating simple polygons and equivalent problems*, ACM Transactions on Graphics, 3 (1984), pp. 153–174.

[14] J. Friedman, J. Hershberger, and J. Snoeyink, *Compliant motion in a simple polygon*, in Proceedings of the 5th ACM Symposium on Computational Geometry, 1989, pp. 175–186.

[15] ———, *Input-sensitive compliant motion in the plane*, in Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory, Springer-Verlag, 1990, pp. 225–237.

[16] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, *Triangulating a simple polygon*, Information Processing Letters, 7 (1978), pp. 175–179.

[17] R. L. Graham and F. F. Yao, *Finding the convex hull of a simple polygon*, Journal of Algorithms, 4 (1983), pp. 324–331.

[18] L. J. Guibas, L. Ramshaw, and J. Stolfi, *A kinetic framework for computational geometry*, in Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, IEEE, 1983, pp. 100–111.

[19] L. J. Guibas, M. Sharir, and S. Sifrony, *On the general motion-planning problem with two degrees of freedom*, Discrete and Computational Geometry, 4 (1989), pp. 491–521.

[20] P. J. Heffernan and J. S. B. Mitchell, *Structured visibility profiles with applications to problems in simple polygons*, in Proceedings of the 6th ACM Symposium on Computational Geometry, 1990, pp. 53–62.

[21] J. Hershberger and J. Snoeyink, *Computing minimum length paths of a given homotopy class*, in Proceedings of the 2nd Workshop on Algorithms and Data Structures, Springer-Verlag, 1991, pp. 331–342. Lecture Notes in Computer Science 519.

[22] S. Huddleston and K. Mehlhorn, *A new data structure for representing sorted lists*, Acta Informatica, 17 (1982), pp. 157–184.

[23] D. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM Journal on Computing, 12 (1983), pp. 28–35.

[24] J.-C. Latombe, *Motion planning with uncertainty: The preimage backchaining approach*, Tech. Report STAN-CS-88-1196, Department of Compuuter Science, Stanford University, Stanford, California 94305, March 1988.

[25] ———, *Robot Motion Planning*, The Kluwer international series in engineering and computer science, SECS 0124, Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[26] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor, *Automatic synthesis of fine-motion strategies for robots*, International Journal of Robotics Research, 3 (1984).

[27] T. Lozano-Pérez and M. A. Wesley, *An algorithm for planning collision-free paths among polyhedral obstacles*, Communications of the ACM, 22 (1979), pp. 560–570.

[28] A. Melkman, *On-line construction of the convex hull of a simple polyline*, Information Processing Letters, 25 (1987), pp. 11–12.

[29] B. K. Natarajan, *On Moving and Orienting Objects*, PhD thesis, Cornell University Department of Computer Science, Ithaca, N.Y., 1986.

[30] C. Ó'Dúnlaing and C. K. Yap, *A 'retraction' method for planning the motion of a disc*, Journal of Algorithms, 6 (1986), pp. 104–111.

[31] F. P. Preparata and M. I. Shamos, *Computational Geometry—An Introduction*, Springer Verlag, New York, 1985.

[32] J. T. Schwartz, M. Sharir, and J. Hopcroft, eds., *Planning, Geometry, and Complexity of Robot Motion*, Ablex Series in Artificial Intelligence, Ablex, Norwood, New Jersey, 1987.

[33] M. Sharir and A. Schorr, *On shortest paths in polyhedral spaces*, SIAM Journal on Computing, 15 (1986), pp. 193–215.

[34] G. Toussaint and D. Avis, *On a convex hull algorithm for polygons and its applications to triangulation problems*, Pattern Recognition, 15 (1982), pp. 23–29.