Three Problems about Dynamic Convex Hulls

Timothy M. Chan[*]

March 21, 2011

Abstract

We present three results related to dynamic convex hulls:

- A fully dynamic data structure for maintaining a set of n points in the plane so that we can find the edges of the convex hull intersecting a query line, with expected query and amortized update time $O(\log^{1+\varepsilon} n)$ for an arbitrarily small constant $\varepsilon > 0$. This improves the previous bound of $O(\log^{3/2} n)$.
- A fully dynamic data structure for maintaining a set of n points in the plane to support halfplane range reporting queries in $O(\log n + k)$ time with $O(\text{polylog}\, n)$ expected amortized update time. A similar result holds for 3-dimensional orthogonal range reporting. For 3-dimensional halfspace range reporting, the query time increases to $O(\log^2 n/\log\log n + k)$.
- A semi-online dynamic data structure for maintaining a set of n line segments in the plane, so that we can decide whether a query line segment lies completely above the lower envelope, with query time $O(\log n)$ and amortized update time $O(n^{\varepsilon})$. As a corollary, we can solve the following problem in $O(n^{1+\varepsilon})$ time: given a triangulated terrain in 3-d of size n, identify all faces that are partially visible from a fixed viewpoint.

1 Introduction

Dynamic planar convex hull has long been a favorite topic in classical computational geometry. The problem is to design a data structure that can maintain a set S of n points in the plane under insertions and deletions and that can answer queries about the convex hull $\text{CH}(S)$. Typical kinds of queries include:

(a) find the most extreme vertex in $\text{CH}(S)$ (i.e., the most extreme point in S) along a query direction;

(b) decide whether a query line intersects $\text{CH}(S)$ (a special case of (a));

(c) find the two vertices of $\text{CH}(S)$ that form tangents with a query point outside the hull;

(d) find the two adjacent vertices to a given vertex of $\text{CH}(S)$ (a special case of (c)).

---

[*] Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). Work supported by NSERC.

The first polylogarithmic method was discovered by Overmars and van Leeuwen (STOC'80) [27]: their *hull tree* data structure can answer queries in $O(\log n)$ time and support updates in $O(\log^2 n)$ time.

For a long time this $O(\log^2 n)$ bound had remained unsurpassed, until the author (FOCS'99) [9] proposed a new line of attack and obtained a method with $O(\log^{1+\varepsilon} n)$ amortized update time that can answer the above queries (a)–(d) in $O(\log n)$ time for any fixed constant $\varepsilon > 0$. (By balancing, one can also get $2^{O(\sqrt{\log \log n})} \log n$ query and amortized update time with this approach.) Shortly afterwards, Brodal and Jacob [5] improved the amortized update time to $O(\log n \log \log n)$ for insertions and $O(\log n \log \log \log n)$ for deletions, by following the same approach and incorporating some additional ideas. Brodal and Jacob continued much further with the approach and eventually in FOCS'02 [6] achieved the coveted $O(\log n)$ bound for both query and amortized update time. The result is optimal in standard decision tree models. Unfortunately, the final method is quite complicated—the current draft of the full paper exceeds 100 pages. Aside from this "minor" drawback, however, the dynamic planar convex hull problem would appear to be fully solved.

But is it? There is a second group of queries for which the $O(\log n)$ result currently does not apply:

(e) find the intersection of $\mathrm{CH}(S)$ with a vertical query line;

(f) decide whether a query point q lies inside $\mathrm{CH}(S)$ (a special case of (e));

(g) find the intersection of $\mathrm{CH}(S)$ with an arbitrary query line;

(h) find the two bridges (outer tangents) and the two separating tangents between two disjoint convex hulls $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$ for given point sets S_1 and S_2.

All these queries arise naturally in many applications. For example, answering *linear programming queries* in a dynamic set of halfplanes in 2-d can be reduced to the above kinds of queries by duality.

What distinguishes the first group of queries (a)–(d) from the second group (e)–(h) is that the former is *decomposable*, i.e., if S is partitioned into subsets, the answer to a query for S can be derived from the answer to the query for the individual subsets. Queries (e)–(h) do not directly satisfy the decomposability property. Instead they satisfy "LP-type" properties, which help, but with a cost: Overmars and van Leeuwen's result still holds, but the author's approach currently only gives a solution with $O(\log^{3/2} n)$ query and amortized update time [9], while Brodal and Jacob's subsequent papers do not address this group of queries at all.

(There are also other convex hull queries beyond these two groups for which Overmars and van Leeuwen's result is applicable but the author's approach is not: e.g., maintain the area or perimeter of the convex hull. We will have to ignore those types of queries here.)

Since Brodal and Jacob's papers, no further progress has been reported. (Demaine and Pătraşcu [16] had results on the word RAM model assuming integer coordinates, but the present paper will focus exclusively on results in the standard algebraic decision tree model.) We believe it is high time to revisit the problem:

Problem 1 *Design a data structure to maintain a set of points in the plane under insertions and deletions so that queries of types (e)–(h) above can be answered efficiently.*

We describe a solution with query and amortized update time $O(\log^{1+\varepsilon} n)$ for any fixed $\varepsilon > 0$ (for the inquisitive readers, the precise bound is actually $2^{O(\sqrt{\log \log n \log \log \log n})} \log n$). Although the solution follows the same approach as in the author's FOCS'99 paper (using exactly the same data structure) and requires only one major new idea in the query algorithm, we believe the result is still interesting because of the fundamental nature of the problem.

Next, we turn to another related fundamental problem: *dynamic halfplane range reporting*. The goal here is to maintain a set S of n points in the plane under insertions and deletions so that we can efficiently report all points inside a query halfplane. We let k denote the number of reported points, i.e., the output size. (A problem with similar characteristics is to find the k most extreme points, in arbitrary order, along a query direction for a given k.)

This problem can be solved using a dynamic convex hull data structure, by repeatedly finding an extreme point along the direction defined by the query halfplane, deleting it, and re-inserting it back later. This gives $O(\log n + k\log^2 n)$ query time with Overmars and van Leeuwen's data structure, or $O((1+k)\log n)$ with Brodal and Jacob's. With Overmars and van Leeuwen's structure, one can more directly obtain $O((1+k)\log n)$ query time. However, neither approach achieves linear dependency in the output size k, which is much desirable in practice:

Problem 2 *Design a data structure to maintain a set of points in the plane under insertions and deletions so that halfplane range reporting queries can be answered in* $O(\mathrm{polylog}\, n + k)$ *time.*

Static data structures achieving $O(\log n + k)$ query time are known with optimal $O(n\log n)$ preprocessing time [14, 8], but the best dynamic data structure with $O(\log n+k)$ query time currently requires $O(n^{\varepsilon})$ amortized update time [2].

We show that $O(\log n+k)$ query time is possible with polylogarithmic amortized expected update time; the precise update bound is $O(\log^{6+\varepsilon} n)$. (The expectation is with respect to internal randomization in the update algorithm; the result holds for worst-case point sets and worst-case update sequences.) The idea is to move away from both the approach by Overmars and van Leeuwen [27] and the 2-d dynamic convex hull approach initiated by the author [9], but instead adapt an approach from a different paper of the author on *3-d* convex hulls [12]. That paper presented the first data structure for dynamic 3-d convex hull queries that achieves polylogarithmic query and update time ($O(\log^2 n)$ query time and $O(\log^6 n)$ amortized expected update time). It was already noted that 3-d halfspace range reporting queries can be answered in $O(\log^2 n + k\log n)$ time with that data structure [12]. We use additional (simple) ideas to bring the query time down to $O(\log n + k)$ in 2-d.

Our ideas imply a weaker $O(\log^2 n/\log\log n + k)$ query bound for dynamic 3-d halfspace range reporting, which has applications to 2-d circular range reporting and 2-d k-nearest-neighbors queries. Note that the $O(\log^2 n/\log\log n)$ query bound is currently the best known even in the insertion-only case for $k = 1$. Perhaps more interestingly, our ideas imply the first data structure for *dynamic 3-d orthogonal range reporting* (reporting all points inside a query axis-aligned box) with $O(\log n + k)$ query time and polylogarithmic amortized expected update time, by exploiting the similarity between 3-d halfspace range reporting and 3-d dominance range reporting. Consequently, we obtain the current-record query time for dynamic orthogonal range reporting in any constant dimension $d \geq 3$ with polylogarithmic update time: $O((\log n/\log\log n)^{d-3}\log n + k)$ query time with $O(\log^{d+6+\varepsilon} n)$ amortized expected update time. Compare this to Mortensen's previous result [24], which has query time worse by almost a log factor, though with a better update bound: $O((\log n/\log\log n)^{d-1} + k)$ query time with $O(\log^{d-1-1/8+\varepsilon} n)$ update time. (See also [25] and references therein for more on dynamic orthogonal range reporting.)

Lastly, we explore a generalization of dynamic planar convex hulls: *dynamic lower envelopes of line segments.* The goal is to maintain a set S of n possibly intersecting line segments in the plane under insertions and deletions so that we can efficiently answer queries about the lower envelope $\mathrm{LE}(S)$. The lower envelope is a continuous x-monotone chain, after adding vertical extensions at "jump" points and a horizontal line at $y = \infty$. We can view $\mathrm{LE}(S)$ as the boundary of a union of unbounded trapezoids containing $(0, \infty)$. Some natural types of queries include:

(i) compute the intersection of $\mathrm{LE}(S)$ with a vertical query line;

(ii) decide whether a point is above or below $\mathrm{LE}(S)$ (a special case of (i));

(iii) decide whether a query line segment is completely below $\mathrm{LE}(S)$;

(iv) given a query ray originating from below $\mathrm{LE}(S)$, compute the first point on $\mathrm{LE}(S)$ that is hit by the ray;

(v) decide whether a query line segment q is completely above $\mathrm{LE}(S)$ (i.e., whether $\mathrm{LE}(S \cup \{q\}) = \mathrm{LE}(S)$);

(vi) given a query ray originating from above $\mathrm{LE}(S)$, compute the first point on $\mathrm{LE}(S)$ that is hit by the ray.

The problem generalizes dynamic convex hulls in more ways than one. For example, if all the segments degenerate to points, then query (iii) for lines reduces to an extreme point query. On the other hand, if all the segments are lines, then the lower envelope is dual to the upper hull, and query (v) for lines reduces to testing whether a point lies inside a convex hull.

It is already known that queries (i)–(ii) can be solved efficiently in polylogarithmic amortized time for lower envelopes of arbitrary x-monotone curves (of constant description complexity) by adapting the author's technique for dynamic 3-d convex hulls [12]. For line segments in the insertion-only case, queries (iii)–(iv) can still be solved relatively easily in polylogarithmic time by known techniques, because such queries are decomposable (a segment is completely below $\mathrm{LE}(S_1 \cup S_2)$ iff it is completely below both $\mathrm{LE}(S_1)$ and $\mathrm{LE}(S_2)$). However, despite superficial likeness, queries (v)–(vi) are nondecomposable and appear markedly more difficult even in the insertion-only case—which is all the more reason why we will study them in this paper:

Problem 3 *Design a data structure to maintain a set of line segments in the plane under insertions only so that queries of types (v)–(vi) can be answered efficiently.*

Several approaches can get sublinear $O(\sqrt{n}\,\mathrm{polylog}\, n)$ query and update time. For example, as suggested by Pankaj Agarwal (personal communication, 2002), we can divide the plane into $\sqrt{n}$ slabs and maintain a dynamic lower envelope of lines and a static lower envelope of segments in each slab; this approach works in the fully dynamic setting. Alternatively, in the insertion-only case, following a strategy from [10], we can maintain one single static lower envelope which is rebuilt after every $\sqrt{n}$ insertions.

We break the $\sqrt{n}$ barrier by presenting a new data structure with $O(\log n)$ query time for (v), $O(\log^2 n)$ query time for (vi), and $O(n^\varepsilon)$ amortized update time for any fixed $\varepsilon > 0$. (By balancing, we can also get $2^{O(\sqrt{\log n})}$ query and update time.) The result extends to the *semi-online* update setting [10, 17], where during each insertion, we know the position of the matching deletion operation in the update sequence. In particular, the semi-online case includes the *offline* case where the entire update sequence is known in advance.

Problem 3 has at least one algorithmic application: given a triangulated terrain of size n in 3-d and a given viewpoint or viewing direction q, classify each face as either "partially visible" or "completely hidden" with respect to q. This problem has obvious connections to hidden surface removal and occlusion culling: it provides a useful preprocessing step to speed up hidden-surface-removal algorithms. An advantage in studying this problem is that the output size (the number of partially visible faces) is obviously at most n, even if the entire visibility map may have quadratic combinatorial complexity. Grove, Murali, and Vitter [18] (see also [21]) for example have studied the similar problem of identifying partially visible faces among n disjoint axis-aligned rectangles in 3-d and gave an $O(n \log n)$ algorithm. The terrain case was explicitly mentioned in the 2002 paper by Kitsios *et al.* [21], who noted an $O(n^{3/2}\alpha(n) \log n)$ algorithm (or more precisely, $O(n\sqrt{k}\alpha(k) \log n)$ if there are k partially visible faces). In the appendix, we point out how the terrain problem reduces to Problem 3 for an (offline) insertion sequence and thus can be solved in time $O(n^{1+\varepsilon})$ (or more precisely, $n2^{O(\sqrt{\log n})}$) by our new data structure.

Note that the superficially similar problem of classifying each face as "partially hidden" vs. "totally visible" is easier, again due to the key issue of decomposability.

The three problems are solved in the three sections to follow, which are independent of one another and can be read in any order one wishes. The solutions to Problems 1 and 2 are more technical, being reliant on methods from previous papers [9, 12]; the solution to Problem 3 is the most original, and arguably the most interesting—it involves a clever new variant of the segment tree. This paper may serve as a "tour" of dynamic convex hull techniques, as by the end, all three current major techniques (Overmars and van Leeuwen's hull trees [27] and the author's approaches to dynamic 2-d convex hulls [9] and dynamic 3-d convex hulls [12]) will have been illustrated.

2 Dynamic Planar Convex Hulls with Line Intersection Queries

In this section, we solve Problem 1. We focus on one representative type of queries: computing the intersection of the convex hull with a vertical query line. It suffices to consider the upper hull. We work in the dual, where the problem reduces to *linear programming (LP) queries*: maintain a set S of lines in $\mathbb{R}^2$ under n insertions and deletions so that given a query direction q, we can find a point on the lower envelope $\mathrm{LE}(S)$ that is extreme along q. Denote the desired point by $\textsc{ans}(S, q)$.

We will actually solve a slight generalization of LP queries for two lower envelopes: for any two sets S and S' currently maintained, find $\textsc{ans}(S \cup S', q)$ for a query direction q. As a subroutine, we will include a solution of the previously solved problem of *lowest line queries*: given a query vertical line q, find $\mathrm{LE}(S) \cap q$. (In the primal, these correspond to extreme point queries.)

The data structure we use is identical to one from the author's previous paper [9]; the innovation lies in the query algorithm. We summarize all the properties we need about the data structure in the theorem below. (The theorem itself was obtained by a combination of a base-b version of the *logarithmic method* [3], a known deletion-only data structure [20], and a dynamic $b^{O(1)}$-ary version of the *interval tree* [4].)

Theorem 2.1 ([9]) *Fix a parameter $b \geq \log n$. For a dynamic set S of lines in $\mathbb{R}^2$ that undergoes n insertions and deletions, we can maintain a tree $\mathcal{T}(S)$ of $b^{O(1)}$ degree and $O(\log_b n)$ height in $O(\log_b n \log n)$ amortized time with the following properties:*

1. *Each node v stores a vertical slab σ_v. At each internal node v, the $b^{O(1)}$ children's slabs partition the slab at v and are maintained in a standard search tree. At a leaf v, the slab does not contain any vertex of* $\mathrm{LE}(S)$*.*
2. *Each node v stores a list S_v of $b^{O(1)}$ lines of S. The lists undergo a total of $O(n \log_b n)$ updates over time.*
3. *If $v_1, \ldots, v_i$ is a path from the root, then* $\mathrm{LE}(S) \cap \partial\sigma_{v_i}$ *coincides with* $\mathrm{LE}(S_{v_1} \cup \cdots \cup S_{v_{i-1}}) \cap \partial\sigma_{v_i}$*.*

We store each list S_v in an auxiliary data structure that supports updates in $U_0(b^{O(1)})$ amortized time, lowest line queries in $Q_{\mathrm{low0}}(b^{O(1)})$ time, and LP queries (over two sets) in $Q_{\mathrm{lp0}}(b^{O(1)})$ time. By the above theorem, the amortized update time of the whole data structure is

$$U(n) \; = \; O(\log_b n \log n + U_0(b^{O(1)}) \log_b n). \tag{1}$$

Answering lowest line queries with this data structure is relatively straightforward: We find the root-to-leaf path of all nodes $v_1, \ldots, v_\ell$ whose slabs contain the query vertical line q. The answer is the line defining $\mathrm{LE}(S) \cap \partial\sigma_{v_\ell}$, which by property 3 is the lowest among the lowest lines of $\mathrm{LE}(S_{v_i}) \cap \partial\sigma_v$ over the $O(\log_b n)$ nodes v_i. Thus, we get query time

$$Q_{\mathrm{low}}(n) \; = \; O(Q_{\mathrm{low0}}(b^{O(1)}) \log_b n). \tag{2}$$

For LP queries, we need the following locality property. This property is implicitly used in known binary search algorithms for solving LP over the intersection of two convex polygons, and related problems such as computing common tangents between linearly separated convex polygons [27, 28].

Lemma 2.2 *Let S and S' be two sets of lines in $\mathbb{R}^2$. Given two vertical lines ℓ and ℓ', and a query direction q, knowing the lowest line of S at ℓ and the lowest line of S' at ℓ' only (but not knowing S and S' themselves), we can deduce in $O(1)$ time which side of ℓ contains the* $\textsc{ans}(S \cup S', q)$ or *which side of ℓ' contains the* $\textsc{ans}(S \cup S', q)$ *(but not necessarily both).*

Proof: By applying an affine transformation, we may assume that q points vertically upward. Let $v = \mathrm{LE}(S) \cap \ell$ and $v' = \mathrm{LE}(S') \cap \ell'$. W.l.o.g., suppose ℓ is to the left of ℓ' and v is higher than v'. If ℓ' has positive slope, then the answer must be to the right of ℓ. Otherwise, the answer must be to the left of ℓ'. $\square$

By applying the property a constant number of times, we immediately obtain this slight generalization:

Corollary 2.3 *Let S and S' be two sets of lines in $\mathbb{R}^2$. Given two partitions Σ and Σ' of $\mathbb{R}^2$ into $O(1)$ disjoint vertical slabs, and a query direction q, knowing the lowest lines of S at all dividing vertical lines in Σ and the lowest lines of S' at all dividing vertical lines in Σ', we can deduce in $O(1)$ time which slab of Σ contains* $\textsc{ans}(S \cup S', q)$ or *which slab of Σ' contains* $\textsc{ans}(S \cup S', q)$*.*

We now describe how to answer an LP query, i.e., find $\textsc{ans}(S \cup S', q)$ given a data structure for S and a data structure for S'. We maintain a sequence of nodes $v_1, v_2 \ldots$ in $\mathcal{T}(S)$ and $v'_1, v'_2, \ldots$ in $\mathcal{T}(S')$ whose slabs contain $\textsc{ans}(S \cup S', q)$. We start with the root v_1 of $\mathcal{T}(S)$ and v'_1 of $\mathcal{T}(S')$ and initialize $i = i' = 1$. In each iteration, let

$$Z = S_{v_1} \cup \cdots \cup S_{v_i}, \;\; Z' = S'_{v'_1} \cup \cdots \cup S'_{v'_{i'}}, \; \text{ and } z = \textsc{ans}(Z \cup Z', q).$$

We will explain how to compute z later. In $O(\log b)$ time, locate the child slab σ of σ_{v_i} and the child slab σ' of $\sigma_{v'_{i'}}$ containing z.[1] Compute the lowest lines of Z at $\partial\sigma$ and the lowest lines of Z' at $\partial\sigma'$, by querying the $O(\log_b n)$ subsets $S_{v_1}, \ldots, S_{v_i}, S'_{v'_1}, \ldots, S'_{v'_{i'}}$ in $O(Q_{\mathrm{low0}}(b^{O(1)}) \log_b n)$ time. By property 3, these coincide with the lowest lines of S at $\partial\sigma$ and the lowest lines of S' at $\partial\sigma'$. Apply Corollary 2.3 and the partition formed by $\partial\sigma$ (into 3 vertical slabs) and the partition formed by $\partial\sigma'$; the outcome should be the same for S, S' and for Z, Z'. Therefore, in $O(1)$ time, we can deduce that σ contains $\textsc{ans}(S \cup S', q)$ *or* σ' contains $\textsc{ans}(S \cup S', q)$. In the former case, we set v_{i+1} to the child of v_i with slab σ and increment i; in the latter, we set $v'_{i'+1}$ to the child of $v'_{i'}$ with slab σ' and increment i'. When both v_i and $v'_{i'}$ are leaves, we can stop, as the answer is defined by the lowest lines at $\partial\sigma_{v_i}$ and $\partial\sigma_{v'_{i'}}$. The number of iterations is at most twice the tree height, i.e., $O(\log_b n)$.

It remains to provide a method to compute z in each iteration. The key idea is to view the $\ell = O(\log_b n)$ subsets $S_{v_1}, \ldots, S_{v_i}, S'_{v'_1}, \ldots, S'_{v'_{i'}}$ as abstract constraints in a convex programming problem. (This idea was used in some previous papers, e.g., in a geometric optimization technique by the author [11].) Specifically, the lower envelopes of these subsets are convex objects and the goal is to optimize a linear function over the intersection of ℓ convex objects. By existing randomized linear-time algorithms for LP-type problems [15, 29], the problem can be solved using $O(\ell)$ expected number of *violation tests*—deciding whether a given point lies outside a given object—and $O(\log \ell)$ expected number of *basis evaluations*—finding the optimum over the intersection of two given objects. In our setting, a violation test corresponds to testing whether a point lies above a lower envelope, which reduces to a lowest line query; a basis evaluation corresponds to an LP query over two subsets. We conclude that z can be computed in $O(\ell \cdot Q_{\mathrm{low0}}(b^{O(1)}) + \log \ell \cdot Q_{\mathrm{lp0}}(b^{O(1)}))$ expected time. The total expected time over all iterations is

$$Q_{\mathrm{lp}}(n) \;=\; O(\log_b n \cdot [Q_{\mathrm{low0}}(b^{O(1)}) \log_b n + Q_{\mathrm{lp0}}(b^{O(1)}) \log\log_b n]). \tag{3}$$

We can now obtain a near-logarithmic solution by bootstrapping. Assume the availability of a solution with $U_0(n) \le c_k (\log n)^{1+1/k}$, $Q_{\mathrm{low0}}(n) \le c_k \log n$, and $Q_{\mathrm{lp0}}(n) \le c_k (\log n)^{1+1/k} (\log\log n)^k$. We can use Overmars and van Leeuwen's method for the base case $k = 1$ (alternatively we can use a more trivial base case with some extra bootstrapping steps). Substituting into (1), (2), and (3) gives

$$\begin{aligned}
U(n) &= O(\log_b n \log n + c_k (\log b)^{1+1/k} \log_b n) \;=\; O(\log^2 n / \log b + c_k \log n (\log b)^{1/k}) \\
Q_{\mathrm{low}}(n) &= O(c_k \log b \log_b n) \;=\; O(c_k \log n) \\
Q_{\mathrm{lp}}(n) &= O(\log_b n \cdot [c_k \log b \log_b n + c_k (\log b)^{1+1/k} (\log\log b)^k \log\log_b n]) \\
&= O(c_k \log^2 n / \log b + c_k \log n (\log b)^{1/k} (\log\log n)^{k+1}).
\end{aligned}$$

Choosing b with $\log b = (\log n)^{k/(k+1)}$ yields the improved bounds $U(n) = O(c_k (\log n)^{1+1/(k+1)})$ and $Q_{\mathrm{lp}}(n) = O(c_k (\log n)^{1+1/(k+1)} (\log\log n)^{k+1})$. Thus, induction can be carried out if we set $c_{k+1} = O(c_k)$, i.e., $c_k = 2^{O(k)}$. We obtain update and query time $2^{O(k)} (\log n)^{1+1/k} (\log\log n)^k$. Setting k to an arbitrarily large constant is sufficient to give $O(\log^{1+\varepsilon} n)$. Better still, setting $k = \sqrt{\log\log n / \log\log\log n}$ gives:

---

[1] In this sentence, the two unbounded slabs from the complement of σ_{v_i} (resp. $\sigma_{v'_{i'}}$) are considered possible child slabs.

Theorem 2.4 *There is a dynamic data structure for 2-d LP queries in the lower envelope of* n *lines with* $2^{O(\sqrt{\log\log n \log\log\log n})}\log n$ *amortized update time and expected query time.*

Remarks. It is not important whether n denotes the number of updates or the size of S, since we can apply the standard trick of rebuilding whenever the size doubles or halves.

We can slightly simplified the algorithm to compute z by observing that in each iteration, Z differs by the insertion of one constraint. If z does not violate the new constraint, then z does not change. Otherwise, we know that the new z is defined by this new constraint and it suffices to solve a 1-dimensional LP-type problem (analogous to finding the minimum of ℓ abstract elements, as in [7]) instead of 2-d.

Extension to other queries. The problem of finding the intersection of the convex hull with an *arbitrary* query line dualizes to the following kind of queries: given a query point q, find a point v on LE(S) to the right of q such that the line qv has the largest slope. The problem is LP-type and Lemma 2.2 is still satisfied (by a projective transformation that depends on q, the problem can be directly reduced to the LP case). So the same method works.

The same applies to finding the intersection of the convex hull of $P \cup P'$ with an arbitrary query line for two dynamic point sets P and P'. In particular, we can compute the two bridges between two disjoint convex hulls if a separating line (e.g., a common separating tangent) is given.

The problem of finding the two common separating tangents between two disjoint convex hulls dualizes to computing the two intersections between a lower envelope of a set S of lines and an upper envelope of a set S' of lines. It suffices to find the left intersection. An analog of Lemma 2.2 (where concerning S', we replace "lower/lowest" with "upper/highest") is still satisfied, and we obtain the same result.

The same applies to LP queries over an intersection of arbitrary (lower and upper) halfplanes, i.e., the region between the lower envelope of a set S of lines and the upper envelope of a set S' of lines. Here, the answer can be one of four possibilities: the optimum for the lower envelope alone or for the upper envelope alone, or one of the two intersections between the two envelopes.

3 Dynamic Halfplane Range Reporting

In this section, we solve Problem 2, dynamic halfplane range reporting in $\mathbb{R}^2$. It suffices to consider lower halfplanes. We work in the dual, where the problem is equivalent to the following: maintain a set of lines in $\mathbb{R}^2$ under insertions and deletions so that we can report all lines below a query point. We work with the following related query problem called *k-lowest-lines queries*: given a query vertical line q and an integer k, report the k lowest lines at q, in arbitrary order. (In the primal, this corresponds to k-extreme-points queries.) Halfplane range reporting reduces to this problem by "guessing" k (e.g., see [8]): we use an increasing sequence of values for k and stop when an output line lies above the query point. If k-lowest-lines queries can be answered in $O(\log n + k)$, we can use the sequence $k = 2^i \log n,\ i = 0, 1, 2\ldots$ and obtain $O(\log n + k)$ time for a halfplane range reporting query with output size k.

The data structure we use is an adaptation of a previous one by the author, originally for dynamic 3-d convex hulls [12]. We encapsulate all the properties we need about the data structure in the theorem below. (The theorem itself was obtained by a new deletion-only "partial" data structure involving a hierarchy of *shallow cuttings*, combined with a variation of the logarithmic method. For the details, see the paper [12] along with the small changes noted in the appendix.)

Theorem 3.1 ([12]) *Fix a parameter $b \geq 2$. For a dynamic set S of lines in $\mathbb{R}^2$ that undergoes n insertions and deletions, in $O(b^{O(1)} \log^6 n)$ expected amortized time, we can maintain a collection of* cuttings $T_i^{(j)}$, $i = 1, \ldots, \lceil \log n \rceil$, $j = 1, \ldots, O(\log_b n)$ *with the following properties:*

1. *Each cutting $T_i^{(j)}$ consists of $O(2^i)$ interior-disjoint trapezoids* (cells) *containing* $(0, -\infty)$. *Each cutting does not change until its next destruction and re-creation, and the total size of all cuttings created over time is $O(b^{O(1)} n \log^4 n)$.*
2. *Each cell $\Delta \in T_i^{(j)}$ is associated with a list L_Δ of $O(n/2^i)$ lines of S. Each list undergoes deletions only after creation, and the total size of all lists created is $O(b^{O(1)} n \log^5 n)$.*
3. *If a plane h is among the k lowest lines at a vertical line q, then h is in L_Δ for some cell $\Delta \in T_{i_k}^{(j)}$ intersecting q with $i_k := \lceil \log(n/Ck)) \rceil$ for a sufficiently large constant C.*

The above theorem immediately gives a dynamic method for k-lowest-lines queries: given a query vertical line q, we simply find the cell $\Delta^{(j)} \in T_{i_k}^{(j)}$ intersecting q for each j, search for the k lowest lines in each list $L_{\Delta^{(j)}}$ by "brute force", and return the k lowest among all the lines found. The x-projection of each cutting $T_i^{(j)}$ is a 1-d subdivision, and so we can locate the cell $\Delta^{(j)}$ for each j in $O(\log n)$ by binary search. Since $|L_{\Delta^{(j)}}| = O(n/2^{i_k}) = O(k)$, the total query time is $O((\log n + k) \log_b n)$. Setting $b = 2$, we obtain $O(\log^6 n)$ amortized expected update time and $O(\log^2 n + k \log n)$ query time.

To improve the query time, we propose a simple idea: store the lists L_Δ in auxiliary data structures. Specifically, we store each L_Δ in a data structure that supports k-lowest-lines queries in $O(Q_0(|L_\Delta|) + k)$ query time and $U_0(|L_\Delta|)$ update time. The amortized expected update time becomes

$$U(n) \;=\; O(b^{O(1)} \log^6 n + \max_{m \leq n} U_0(m) \cdot b^{O(1)} \log^5 n). \tag{4}$$

Brute force search in L_Δ can be replaced by a query in an auxiliary structure. The query time becomes

$$Q(n) \;=\; O([\log n + Q_0(O(k))] \log_b n + k).$$

For example, we can use a known method with $Q_0(m) = O(m^{1-\varepsilon})$ and $U_0(m) = O(\log m)$, and set $b = 2$. This gives polylogarithmic update time and query time $Q(n) = O(\log^2 n + k^{1-\varepsilon} \log n + k) = O(\operatorname{polylog} n + k)$. The $\operatorname{polylog} n$ term in the query time is improvable by further bootstrapping, but the main bottleneck is actually the $\log n \log_b n$ term, which arises from $O(\log_b n)$ simultaneous binary searches.

We need a second idea to speed up these binary searches. We can use dynamic fractional cascading here, but we suggest a simpler alternative: *dynamic interval trees* [4, 23]. For each $i = 1, \ldots, \lceil \log n \rceil$, we store the intervals of the x-projection in cuttings $T_i^{(j)}$ over all j's in an interval tree $\mathcal{T}_i$. With interval trees, we can support insertions and deletions of intervals in $O(\log n)$ time and report all K intervals containing a query point in $O(\log n + K)$ time. We can thus maintain all the $\mathcal{T}_i$'s in additional amortized update time $O(\log n \cdot b^{O(1)} \log^4 n)$, and locate the cells $\Delta^{(j)} \in T_{i_k}^{(j)}$ over all $j = 1, \ldots, \lceil \log_b n \rceil$ in $O(\log n + \log_b n)$ time. The overall query time is

$$Q(n) \;=\; O(\log n + Q_0(O(k)) \log_b n + k). \tag{5}$$

For $m \geq \log^{1/\varepsilon} n$, we use a known method [2, 22] with $Q_0(m) = O(m^{1-\varepsilon})$ and $U_0(m) = O(\log m)$. For $m < \log^{1/\varepsilon} n$, we switch to a static method [8, 14] with $Q_0(m) = O(\log m)$ and $U_0(m) = m^{O(1)}$. Substituting into (4) and (5) and setting $b = \log^\varepsilon n$ give $U(n) = O(\mathrm{polylog}\, n)$ and

$$Q(n) \;=\; \left\{ \begin{array}{ll} O(\log n + k^{1-\varepsilon}\log n + k) & \text{if } k \geq \log^{1/\varepsilon} n \\ O(\log n + \log k \log n/\log\log n + k) & \text{if } k < \log^{1/\varepsilon} n \end{array} \right\} \;=\; O(\log n + k).$$

The update time can be improved by another bootstrapping step. This time, for the $m < \log^{1/\varepsilon} n$ case, we use $Q_0(m) = O(\log m)$ and $U_0(m) = O(\mathrm{polylog}\, m)$. This brings the amortized expected update time down to $O(\log^{6+\varepsilon} n)$.

Theorem 3.2 *There is a dynamic data structure for 2-d halfplane range reporting with $O(\log^{6+\varepsilon} n)$ amortized expected update time and $O(\log n + k)$ query time.*

Extensions to 3-d and dominance/orthogonal range reporting. For 3-d halfspace range reporting, finding the cell $\Delta^{(j)}$ reduces to 2-d point location in the xy-projection of the cutting $T_i^{(j)}$. This time, we do not know how to improve the $\log n \log_b n$ term arising from the $O(\log_b n)$ planar point location queries. For $b = \log^\varepsilon n$, the query time is now $O(\log^2 n/\log\log n + k)$.

The 3-d dominance range reporting problem can be solved by the same techniques as 3-d halfspace range reporting (due to similarity of lower envelopes of planes and lower envelopes of orthants; e.g., see [1]). Cells are now axis-aligned boxes, and the required planar point location queries are for orthogonal subdivisions. By a recent result of the author [13], 2-d orthogonal point location can be solved in $O(\log\log n)$ time with $O(n)$ preprocessing, if we are given the x- and y-ranks of the query point with respect to the given subdivision. We can again use dynamic interval trees (for x and y) to search for the x- and y-ranks of the query point with respect to $T_{i_k}^{(j)}$ over all $j = 1, \ldots, \lceil \log_b n \rceil$ in $O(\log n + \log_b n)$ time. So, the $\log n \log_b n$ term reduces to $O([\log n + \log_b n] + \log\log n \log_b n)$. For $b = \log^\varepsilon n$, the query time is thus $O(\log n + k)$.

The 3-d j-sided orthogonal range reporting problem reduces to 3-d $(j-1)$-sided range reporting by standard binary divide-and-conquer (e.g., see [24]), where the update time (but not the query time) increases by a logarithmic factor. Thus, 3-d general (6-sided) orthogonal range reporting reduces to 3-d dominance range reporting, where the update time increases by a $\log^3 n$ factor.

In higher dimensions, general orthogonal range reporting reduces to range reporting in one dimension lower by a dynamic b-ary range tree [4, 23], where the update time increases by a factor of $b^{O(1)} \log_b n$ and the query time increases by a factor of $\log_b n$. We can set $b = \log^\varepsilon n$ to keep the update time polylogarithmic.

Theorem 3.3 *There is a dynamic data structure for 3-d halfspace range reporting with $O(\log^{6+\varepsilon} n)$ amortized expected update time and $O(\log^2 n/\log\log n + k)$ query time. There is a dynamic data structure for 3-d dominance range reporting with $O(\log^{6+\varepsilon} n)$ amortized expected update time and $O(\log n + k)$ query time. There is a dynamic data structure for d-d orthogonal range reporting for any constant $d \geq 3$ with $O(\log^{d+6+\varepsilon} n)$ amortized expected update time and $O((\log n/\log\log n)^{d-3}\log n + k)$ query time.*

4 Semi-Dynamic Lower Envelopes of Line Segments

In this section, we solve Problem 3. We focus on the insertion-only case. The main problem is to maintain a set S of n line segments under insertions to answer what we call *segment queries*: decide whether a query line segment is completely above the lower envelope $\mathrm{LE}(S)$. We will also consider a slightly stronger form of queries, *ray shooting*: given a query ray originating from a point above $\mathrm{LE}(S)$, find the first point on $\mathrm{LE}(S)$ that is hit by the ray. (Segment queries can be viewed as a decision version of ray shooting queries.) Interestingly, our update algorithm will rely on our query algorithm in a crucial way.

Preliminaries. To develop intuition, it is helpful to keep in mind the special case of a segment query where the segment is a line. Here, the query is equivalent to deciding whether a line is completely above the upper hull of $\mathrm{LE}(S)$, denoted $\mathrm{UH}(\mathrm{LE}(S))$. Although $\mathrm{LE}(S)$ can change drastically in an insertion, we will show that $\mathrm{UH}(\mathrm{LE}(S))$ is easier to update.

One ingredient we use is Overmars and van Leeuwen's *hull tree* structure [27]. A hull tree for a point set consists of a pointer to a hull tree for the subset of points to the left of a dividing vertical line ℓ, a pointer to a hull tree for the subset of points to the right of ℓ, and a pointer to the bridge at ℓ (the common tangent of the two subhulls). Standard operations such as intersecting a hull with a line and merging two vertically separated hulls can be performed by binary search in logarithmic time if the heights of the hull trees are logarithmically bounded [27, 28].

Another tool we need is a standard data structure for storing a set of line segments—the *segment tree* [28]. Our data structure will involve an unusual adaptation of the segment tree. Recall the following standard definitions: given a segment s that intersects a vertical slab σ, s is *short* in σ if at least one endpoint of s is inside σ; s is *long* otherwise. In a traditional segment tree, we remove long segments, divide the given slab into two subslabs, and recursively handle short segments intersecting each of the two subslabs. Our data structure will be more intricate: first, we need to maintain auxiliary information about the long segments and use a higher-degree tree; second, and more intriguingly, because we are unable to eliminate the long segments entirely in our method, we need to pass some (hopefully small) number of long segments to the subslabs.

The data structure. Fix a parameter b to be determined later. Given a set S of line segments and a vertical slab σ with at most m endpoints inside, we describe our data structure for S and σ recursively as follows:

- Let $S_{\text{short}} = \{s \in S \mid s \text{ is short in } \sigma\}$ and $S_{\text{long}} = \{s \in S \mid s \text{ is long in } \sigma\}$.
- Store the concave chain $L = \mathrm{LE}(S_{\text{long}}) \cap \sigma$ in a standard search tree. For each $s \in S_{\text{long}}$, define its *reduced segment* $\tilde{s}$ to be the segment delimited by the leftmost point and the rightmost point on s lying on $\mathrm{LE}(S) \cap \sigma$, if such points exist. Observe that the reduced segments all lie on L. See Figure 1(left).
- If $m > 0$, divide σ into $O(b)$ subslabs $\sigma_1, \sigma_2, \ldots$ (indexed from left to right), each with at most m/b endpoints inside. For each σ_i:
  - Recursively store a data structure for a subset $S_i \subseteq S$ and the subslab σ_i, where we maintain the invariant

$$S_i \supseteq \{s \in S_{\text{short}} \mid s \text{ intersects } \sigma_i\} \cup \{s \in S_{\text{long}} \mid \tilde{s} \text{ is short in } \sigma_i\}.$$

  - Observe that there is at most one segment $s \in S_{\text{long}}$ such that $\tilde{s}$ is long in σ_i. If such an s exists, call s the *special segment* of σ_i and mark σ_i as *special.* See Figure 1(middle).

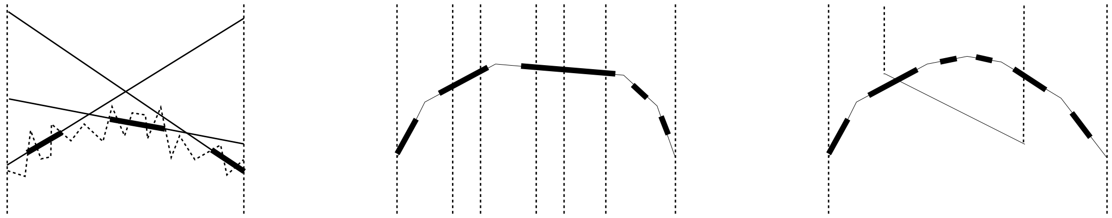

Figure 1: (Left) The long segments and their reduced segments; the lower envelope of the short segments are drawn abstractly in dotted lines. (Middle) Another example showing the reduced segments on the concave chain L; the 2nd, 4th, and 5th subslabs from the left are special. (Right) Inserting a new segment causes the recomputation of two existing reduced segments.

Clearly, S_i includes all segments that participate in $\mathrm{LE}(S) \cap \sigma_i$, except possibly for the one special segment. In particular, if σ_i is not special, then $\mathrm{LE}(S) \cap \sigma_i = \mathrm{LE}(S_i) \cap \sigma_i$. On the other hand, if σ_i is special, then no vertices inside σ_i can participate in $\mathrm{UH}(\mathrm{LE}(S) \cap \sigma)$, since the reduced special segment is an edge of this upper hull and has endpoints outside σ_i.

- Store $U = \mathrm{UH}(\mathrm{LE}(S) \cap \sigma)$ in a hull tree of logarithmic height. By the above observations, special slabs do not affect U and we have the key identity

$$U \;=\; \mathrm{UH}\left(\bigcup_{\sigma_i \text{ not special}} \mathrm{UH}(\mathrm{LE}(S_i) \cap \sigma_i)\right). \tag{6}$$

- Finally, to help speed up querying, store the subhulls $U_{j,k} = \mathrm{UH}(\bigcup_{i=j}^{k} \mathrm{UH}(\mathrm{LE}(S_i) \cap \sigma_i))$ in a hull tree of logarithmic height for each index pair j, k.

At the root, we set σ to be the entire plane. Let n denote the maximum number of segments overall.

Segment queries. We can decide whether a query segment s is completely above $\mathrm{LE}(S) \cap \sigma$ as follows:

1. Compute s', the intersection of s with the region underneath L. It remains to decide whether this clipped segment s' is completely above $\mathrm{LE}(S_{\mathrm{short}}) \cap \sigma$, or equivalently, whether s' is completely above $\mathrm{LE}(S_i) \cap \sigma_i$ for every subslab σ_i intersected by s'. (If $m = 0$, the answer is yes iff $s' = \emptyset$.)
2. Suppose that s' is short in two subslabs, say, σ_j and σ_k. (The case where s' is short in just one subslab is similar.) Directly decide whether s' is above $U_{j+1,k-1}$, and recursively decide whether s' is completely above $\mathrm{LE}(S_j) \cap \sigma$ and $\mathrm{LE}(S_k) \cap \sigma$. Then the answer to the original query is yes iff the answers to these queries are all yes.

For the analysis, note that computing s' (intersecting a line and a concave chain) and querying $U_{j+1,k-1}$ (intersecting a line and a hull) can be handled by binary search in logarithmic time [27, 28]. Letting $Q_{\mathrm{seg}}(m)$ denote the query time for a slab σ with at most m endpoints, we obtain the recurrence

$$Q_{\mathrm{seg}}(m) = 2Q_{\mathrm{seg}}(\lfloor m/b \rfloor) + O(\log n),$$

with $Q_{\mathrm{seg}}(0) = O(\log n)$. This recurrence solves to $Q_{\mathrm{seg}}(m) = O(2^{\log_b m} \log n)$.

Ray shooting queries. Given a query ray originating from above $\mathrm{LE}(S) \cap \sigma$, we can find the first point on the ray that lies on or $\mathrm{LE}(S) \cap \sigma$ as follows (we could use parametric search but the following is simpler):

> Use binary search and the segment query algorithm to identify the subslab σ_i containing the desired point. Then recurse in σ_i. (If $m = 0$, the answer can be found directly by a ray shooting query on L.)

The query time $Q_{\mathrm{ray}}(m)$ then obeys the recurrence

$$Q_{\mathrm{ray}}(m) = Q_{\mathrm{ray}}(\lfloor m/b \rfloor) + O(\log b) Q_{\mathrm{seg}}(m),$$

with $Q_{\mathrm{ray}}(0) = O(\log n)$. This yields $Q_{\mathrm{ray}}(m) = O(2^{\log_b m} \log b \log n)$.

Segment insertion. The insertion of a new segment proceeds in two stages: the two endpoints are inserted, then the segment itself. We describe the second stage first.

To insert a segment s to S, where the endpoints have already been inserted:

1. Compute the (at most two) intersections of L with $\partial\tau$, where τ denotes the region above s (an unbounded trapezoid). See Figure 1(right).

2. For each $s' \in S_{\mathrm{long}}$ defining such an intersection, recompute the reduced segment $\tilde{s'}$ and recursively insert s' to S_i for each subslab σ_i where $\tilde{s'}$ is short. Observe that other reduced segments need not be updated, as those completely inside τ are now non-existent and those completely outside τ are unchanged.

3. If s is long in σ, compute the reduced segment $\tilde{s}$ and recursively insert s to S_i for each subslab σ_i where $\tilde{s}$ is short. Update L.

4. If s is short in σ, recursively insert s to S_i for each subslab σ_i intersected by s.

5. Update the specialness marks of the subslabs.

6. Update U by repeated merging according to (6). Similarly update the $U_{j,k}$'s.

We now analyze the cost of this algorithm. Step 1 takes logarithmic time by binary search on a concave chain. There are at most two candidates for s' and thus at most four[2] recursive calls in step 2; there are at most two recursive calls in step 3. The computation of a reduced segment (steps 2 and 3) corresponds precisely to answering a ray shooting query in $\mathrm{LE}(S) \cap \sigma$ from each of the two endpoints, a problem which we have conveniently solved. The concave chain L can be updated in logarithmic (worst-case) time by splitting at the at most two intersections (as in a standard insertion-only algorithm for convex hulls [28] when dualized). Step 4 requires $O(b)$ recursive calls, but the inserted segment is short in at most two of the subslabs in these calls. Step 5 can be done in $O(b)$ time. In step 6, each hull tree U or $U_{j,k}$ can be computed naively by $O(b)$ merges of existing hull trees at the subslabs, in $O(b \log n)$ time. Note that these $O(b^2)$ trees may share common subtrees, so the resulting structure is actually a dag. We can indeed guarantee that the maximum height of the hull trees is $O(\log n)$ if the $O(b)$ merges are done in a balanced fashion, so that the height increase is $O(\log b)$ at each of the $O(\log_b m)$ levels of the recursion.

---

[2] Actually, at most two of the four endpoints of the reduced segments can change, so the number of recursive calls here can be reduced to two.

Letting $U_{\text{long}}(m)$ (resp. $U_{\text{short}}(m)$) denote the insertion time when the given segment s is long (resp. short) in σ, we obtain the following pair of recurrences:

$$\begin{aligned} U_{\text{long}}(m) &= 6U_{\text{long}}(\lfloor m/b \rfloor) + O(Q_{\text{ray}}(m) + b^3 \log n) \\ U_{\text{short}}(m) &= 2U_{\text{short}}(\lfloor m/b \rfloor) + O(r)U_{\text{long}}(\lfloor m/b \rfloor) + O(Q_{\text{ray}}(m) + b^3 \log n), \end{aligned}$$

with $U_{\text{long}}(0), U_{\text{short}}(0) = O(\log n)$. The first recurrence solves to $U_{\text{long}}(m) \leq 2^{O(\log_b m)} b^{O(1)} \log n$. The second recurrence then gives $U_{\text{short}}(m) \leq 2^{O(\log_b m)} b^{O(1)} \log n$. We conclude that a segment can be inserted in $U_{\text{seg}}(m) \leq 2^{O(\log_b m)} b^{O(1)} \log n$ time if the endpoints have already been inserted.

Endpoint insertion. To insert a new endpoint p in the slab σ, we use a standard weight-balancing technique [23, 26]:

1. Recursively insert p in the subslab σ_i containing p.
2. If the number of endpoints inside σ_i exceeds m/r, then rebuild the data structure at σ in a manner where each slab is divided into $2r$ subslabs with equal number of endpoints inside, and all segments in S are then reinserted from scratch.

Note that the rebuilding step requires $O(m)$ segment insertions but can occur only after at least $\lfloor m/2b \rfloor$ endpoint insertions. The amortized cost of each endpoint insertion, $U_{\text{endpt}}(m)$, thus satisfies

$$U_{\text{endpt}}(m) = U_{\text{endpt}}(\lfloor m/b \rfloor) + O((b/m) \cdot mU_{\text{seg}}(m)),$$

which solves to $U_{\text{endpt}}(m) \leq 2^{O(\log_b m)} b^{O(1)} \log n$.

Finally, setting $b = n^{\varepsilon}$ gives $Q_{\text{seg}}(m) = O(\log n)$, $Q_{\text{ray}}(m) = O(\log^2 n)$, and $U_{\text{seg}}(m), U_{\text{endpt}}(m) = O(n^{O(\varepsilon)})$. Alternatively, setting $r = 2^{\sqrt{\log n}}$ gives $Q_{\text{seg}}(m), Q_{\text{ray}}(m), U_{\text{seg}}(m), U_{\text{endpt}}(m) = 2^{O(\sqrt{\log n})}$.

Theorem 4.1 *There is an insertion-only data structure for Problem 3 with $O(\log n)$ time for segment queries, $O(\log^2 n)$ time for ray shooting queries, and $O(n^{\varepsilon})$ amortized insertion time. Alternatively, we can obtain $2^{O(\sqrt{\log n})}$ query and amortized insertion time.*

Extension to the semi-online dynamic case. Our data structure does not seem to cope with arbitrary deletions well, because a single deletion may reveal a large number of new reduced segments in unpredictable ways. However, we can extend the insertion-only result to the semi-online dynamic setting by applying a simple general transformation stated below, which it appears has not been explicitly noted before:

Lemma 4.2 *Given a data structure for a problem that supports insertions and the* undo *operation (i.e., deletion of the most recent element in the current set) in $O(U(n))$ time, there is a semi-online dynamic data structure with $O(U(n) \log n)$ amortized update time.*

Proof: We first consider the design of an offline data structure. Given an offline sequence of n updates, build a segment tree over the time intervals of the elements (the left/right endpoint correspond to the insertion/deletion time of an element), where the i-th leaf v_i (in left-to-right order) correspond to the i-th update in the sequence. Each time interval is stored in $O(\log n)$ nodes in the tree. Let S_v denote the subset of elements whose time intervals are stored at node v of the tree. The current set at the time of the i-th update is equal to the (disjoint) union of S_v over all nodes v from the root to the leaf v_i.

At the i-th update, let u be the lowest common ancestor between v_{i-1} and v_i. We can execute the update simply by undoing the insertions of the elements in S_v over all nodes v from v_{i-1} up to u's left child, then inserting the elements in S_v over all nodes from u's right child down to v_i. The total number of insertions/undos is equal to the total size of all the S_v's, which is $O(n \log n)$. This implies the $O(U(n) \log n)$ amortized bound.

The same reduction works in the semi-online case, by building the segment tree online. At the time of the i-th update, we know S_v for all nodes v from the root to v_i, because deletion times are given for all the elements inserted so far. □

The above lemma is applicable to our data structure, because it supports undos. The main observation is that our insertion time bound is worst-case, not amortized, if we ignore endpoint insertion. Thus, by keeping a transcript of the changes made during each insertion, we can undo an insertion with the same cost. We do not need to undo endpoint insertion, since extra endpoints do not affect the correctness of the query algorithm. (Besides, the endpoint insertion part can be deamortized by standard techniques [23, 26].) The extra log factor is absorbed by the n^{ε} or $2^{O(\sqrt{\log n})}$ bound.

Application to partial visibility in terrains. Given a triangulated terrain T in $\mathbb{R}^3$ of size n and a viewpoint q, we consider the problem of identifying all faces of T which are partially visible from q.

First, by translation, we can make q the origin. It suffices to consider the part of the terrain inside $\{x > 0\}$. Apply a projective transformation $(x, y, z) \mapsto (-1/x, y/x, z/x)$. Vertical lines are mapped to vertical lines, so the terrain remains a terrain. Lines through the origin are mapped to lines parallel to the x-axis, so the problem now becomes finding the faces that are partially visible from $(-\infty, 0, 0)$.

Project all the faces (triangles) into the xy-plane and compute a *depth order* from left to right. It is well known [19] that a depth order for disjoint convex polygons in 2-d always exists and can be computed in $O(n \log n)$ time. Let $f_1, \ldots, f_n$ be the faces (back in 3-d) in this order. Let f_i' denote the *curtain* $\mathrm{conv}(f_i \cup \{(-\infty, 0, 0)\})$ and let f_i'' be the projection of f_i' onto the yz-plane. Then f_i is partially visible iff f_i'' is not completely contained in $f_1'' \cup \cdots \cup f_{i-1}''$, i.e., $\partial f_i''$ is not completely below the upper envelope of $\partial f_1'', \ldots, \partial f_{i-1}''$. We thus arrive at a dynamic 2-d upper envelope problem with at most $2n$ segment insertions and $2n$ segment queries. By Theorem 4.1, the problem can be solved in $n 2^{O(\sqrt{\log n})}$ time.

Acknowledgement. I am grateful to Pankaj Agarwal for bringing both Problem 2 and Problem 3 (as well as the application to the terrain visibility problem) to my attention. Thanks also to Esther Ezra for discussion on Problem 3, and to Kostas Tsakalidis for asking about dynamic 3-d dominance searching.

References

[1] P. Afshani. On dominance reporting in 3D. In *Proc. 16th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 5198, Springer-Verlag, pages 41–51, 2008.

[2] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13:325–345, 1995.

[3] J. Bentley and J. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

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

[5] G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time. In *Proc. 7th Scand. Workshop Algorithm Theory*, Lect. Notes Comput. Sci., vol. 1851, Springer-Verlag, pages 57–70, 2000.

[6] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002. Current full version at https://pwgrp1.inf.ethz.ch/Current/DPCH/Journal/topdown.pdf, 2006.

[7] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete Comput. Geom.*, 22:547–567, 1999.

[8] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM J. Comput.*, 30:561–575, 2000.

[9] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[10] T. M. Chan. Semi-online maintenance of geometric optima and measures. *SIAM J. Comput.*, 32:700–716, 2003.

[11] T. M. Chan. An optimal randomized algorithm for maximum Tukey depth. In *Proc. 15th ACM-SIAM Sympos. Discrete Algorithms*, pages 423–429, 2004.

[12] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16, 2010.

[13] T. M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. In *Proc. 22nd ACM-SIAM Sympos. Discrete Algorithms*, pages 1131–1145, 2011.

[14] B. Chazelle, L. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[15] K. L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *J. ACM*, 42:488–499, 1995.

[16] E. D. Demaine and M. Pǎtraşcu. Tight bounds for dynamic convex hull queries (again). In *Proc. 23rd ACM Sympos. Comput. Geom.*, pages 354–363, 2007.

[17] D. P. Dobkin and S. Suri. Maintenance of geometric extrema. *J. ACM*, 38:275–298, 1991.

[18] E. F. Grove, T. M. Murali, and J. S. Vitter. The object complexity model for hidden line elimination. *Int. J. Comput. Geom. Appl.*, 9:207–217, 1999.

[19] L. J. Guibas and F. F. Yao. On translating a set of rectangles. In *Computational Geometry* (F. P. Preparata, ed.), in *Advances in Computing Research*, Vol. 1, JAI Press, London, England, pages 61–77, 1983.

[20] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[21] N. Kitsios, C. Makris, S. Sioutas, A. K. Tsakalidis, J. Tsaknakis, and B. Vassiliadis. An optimal algorithm for reporting visible rectangles. *Inform. Process. Lett.*, 81:283–288, 2002.

[22] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[23] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Heidelberg, 1984.

[24] C. W. Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35:1494–1525, 2006.

[25] Y. Nekrich. Dynamic range reporting in external memory. In *Proc. 21st Int. Sympos. Algorithms and Computation*, Lect. Notes Comput. Sci., vol. 6507, Springer-Verlag, pages 25–36, 2010.

[26] M. H. Overmars. *The Design of Dynamic Data Structures.* Lect. Notes in Comput. Sci., vol. 156, Springer-Verlag, 1983.

[27] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.*, 23:166–204, 1981.

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

[29] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, Lect. Notes Comput. Sci., vol. 577, Springer-Verlag, pages 569–579, 1992.

A Appendix

On Theorem 3.1 (the dynamic convex hull method from [12]). The constant b case of Theorem 3.1 follows directly from the the dynamic 3-d convex hull method in [12], after specialization to 2-d, as we now explain. We assume that the reader has the paper [12] handy. The cuttings $T_i^{(j)}$ are just the T_i's from the paper, taken over each of the $O(\log n)$ partial data structures S. The list L_Δ corresponds to $(S_{\rm live})_\Delta$.

The proof of [12, Lemma 6.1] shows that in a partial data structure, if h is among the k lowest lines of S at a vertical line q and $h \in S_{\rm live}$, then $h \cap q$ lies inside the cell $\Delta \in T_{\lceil \log(|S|/8ck)\rceil}$ that intersects q; in particular, $h \in (S_{\rm live})_\Delta$ for this cell Δ. Hence, property 3 of Theorem 3.1 follows. (Minor note: for $|S| \ll n$, indices i of the T_i's technically should be shifted to match the setup in Theorem 3.1.)

The generalization for nonconstant b requires only a few adjustments of parameters. In line 4 of construct(S) from [12, Section 3], we replace $4c'\lceil \log n\rceil$ with $(2c'/\alpha)\lceil \log n\rceil$ for an appropriate parameter α. This would guarantee that construct(S) puts at least $(1-\alpha)n$ planes in $S_{\rm live}$. In [12, Section 4], we change the definition of depth from $\lfloor \log |S_{\rm live}|\rfloor$ to $\lfloor \log_b |S_{\rm live}|\rfloor$, to ensure that the number of partial data structures at any time is $O(\log_b n)$. We merge whenever there are 2 (rather than 16) subsets of the same depth. (Note the different meaning of b in the paper.) In the proof of [12, Theorem 4.1(a)], the potential increase caused by an insertion is now given by the following expression, where $p = |S_{\rm live}^{(1)}| + |S_{\rm live}^{(2)}|$ and $b^k \le |S_{\rm live}^{(1)}|, |S_{\rm live}^{(2)}| < b^{k+1}$ (which imply $|S_{\rm live}^{(1)}|, |S_{\rm live}^{(2)}| \le \frac{b}{b+1}p$):

$$\sum_{j=0}^{\infty}(1-\alpha)\alpha^j p \log[(1-\alpha)\alpha^j p] \;-\; \sum_{i=1}^{2} |S_{\rm live}^{(i)}| \log |S_{\rm live}^{(i)}|$$

$$
\begin{aligned}
&\geq\ (1-\alpha)p\log[(1-\alpha)p]\sum_{j=0}^{\infty}\alpha^j \ - \ (1-\alpha)p\log(1/\alpha)\sum_{j=0}^{\infty}j\alpha^j \ - \ p\log[bp/(b+1)] \\
&=\ p\log[(1-\alpha)p] \ - \ [\alpha p\log(1/\alpha)]/(1-\alpha) \ - \ p\log[bp/(b+1)] \ = \ \Omega(p/b),
\end{aligned}
$$

by setting $\alpha = 1/b^{1+\varepsilon}$, for example. The rest of the amortized analysis thus goes through after readjusting bounds by polynomial factors in b.