A Dynamic Data Structure for 3-d Convex Hulls and 2-d Nearest Neighbor Queries

Timothy M. Chan*

Abstract

We present a fully dynamic randomized data structure that can answer queries about the convex hull of a set of n points in three dimensions, where insertions take $O(\log^3 n)$ expected amortized time, deletions take $O(\log^6 n)$ expected amortized time, and extreme-point queries take $O(\log^2 n)$ worst-case time. This is the first method that guarantees polylogarithmic update and query cost for arbitrary sequences of insertions and deletions, and improves the previous $O(n^\varepsilon)$-time method by Agarwal and Matoušek a decade ago. As a consequence, we obtain similar results for nearest neighbor queries in two dimensions and improved results for numerous fundamental geometric problems (such as levels in three dimensions and dynamic Euclidean minimum spanning trees in the plane).

1 Introduction

One of the classic problems in geometric data structures is *nearest neighbor search* (often dubbed the "post office" problem): preprocess a set P of n point sites so that we can quickly report the closest site to any given query point. In two dimensions, an optimal solution with $O(n \log n)$ preprocessing time and $O(\log n)$ query time has long been known and represents one of the major early achievements in computational geometry [3, 29]. However, the *dynamic* version in which we are allowed to insert and delete sites yields a much different story and is surprisingly still open: while the one-dimensional problem can be solved with $O(\log n)$ update time and $O(\log n)$ query time by standard balanced search trees, no method with the same performance is known in two dimensions.

We briefly review previous work on this dynamic 2-d problem. A straightforward sublinear method with roughly $O(\sqrt{n})$ update and query time was known early on [2]. At FOCS'92, Agarwal and Matoušek [1] presented the first improvements. In the 2-d case, they gave two methods, one with $O(\log n)$ query time and $O(n^\varepsilon)$ amortized update time, and another with $O(n^\varepsilon)$ query time and $O(\log^2 n)$ amortized update time. Here, ε is an arbitrary but fixed positive constant. There has been no further general improvement since (up to now); in particular, no existing method achieves polylogarithmic query and update time simultaneously.

Various special cases have been studied, though. When *only insertions* are allowed, a standard technique [2] yields an $O(\log^2 n)$ amortized bound for updates and queries. The same result extends to the *off-line* case (where the update sequence may contain both insertions and deletions but we are given the entire sequence in advance) and the *semi-online* case (where during the insertion of an element, we are given the time when that element will be deleted) [16].

Several researchers—Clarkson, Mehlhorn, and Seidel [12], Devillers, Meiser, and Teillaud [14], Schwarzkopf [31], and Mulmuley [27]—explored the case where the update sequence is *random* in a certain sense (for example, the insertion order is equally likely to be any permutation of the elements, and each deletion is equally likely to be applied to any of the current elements). For 2-d nearest neighbor search, updates can be performed in optimal $O(\log n)$ expected time, while queries can be answered in $O(\log^2 n)$ time [27]. These average-case analyses suggest that good running times may not be too difficult to achieve in practice. However, these analyses reveal nothing about the running time for a specific update sequence, or for an update sequence performed during the execution of an algorithm (which is hardly random). This is disappointing, as one of the motivations for designing data structures, after all, is that they can be used within algorithms to solve static problems.

In this paper, we present the first general data structure for dynamic 2-d nearest neighbor search that has polylogarithmic update and query amortized cost for any (on-line) sequence of updates. The expected amortized insertion time is $O(\log^3 n)$, the expected amortized deletion time is $O(\log^6 n)$, and the worst-case query time is $O(\log^2 n)$; the data structure can be built in $O(n \log^2 n)$ expected time. As usual, expectation is with respect to the random choices made within the algorithm. Our algorithm uses randomization in only a mild way: an adversary may even have access to the

*School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). This work has been supported by NSERC.

random choices made by earlier operations as he/she devises an update sequence. Our algorithm is simpler than Agarwal and Matoušek's [1].

Our result (like Agarwal and Matoušek's) is actually derived for a more general problem:

> Maintain a dynamic set P of points in 3-d so that we can quickly answer *extreme-point* queries for the *convex hull* of P, i.e., finding a hull vertex that is extreme along a given direction.

By a well-known lifting transformation [3], 2-d nearest neighbor queries reduce to such 3-d queries. Other kinds of queries about the 3-d convex hull can also be answered in polylogarithmic time. (See Section 5 for more details.) We comment that for the analogous dynamic 2-d convex hull problem, a polylogarithmic method was known since the early 1980s; Overmars and van Leeuwen's original $O(\log^2 n)$ time bound [28] was later improved to $O(\log^{1+\varepsilon} n)$ by this author [6] and eventually to $O(\log n)$ by Brodal and Jacob [4] for extreme-point queries (for some other kinds of convex hull queries, the best time bound is currently $O(\log^{3/2} n)$ [6]). The techniques we use for the 3-d problem turn out to be quite different from those used for the 2-d problem.

It is hard to overstate the importance of convex hulls in computational geometry. Among the many consequences of our result include improved dynamic data structures for ray shooting queries in a 3-d intersection of halfspaces, 3-d linear programming, 2-d smallest enclosing circle, 2-d diameter, 2-d bichromatic closest pair, 2-d Euclidean minimum spanning trees, and 2-d incremental width. Our result also automatically leads to improved (static) algorithms for computing the convex layers of a 3-d point set and the k-level of a 3-d set of planes. (See Section 5 for more information.)

We briefly point out the main difficulties in designing a data structure for dynamic 3-d convex hulls. Insertions can be treated by standard techniques, so we can focus on deletion. Even in the simplest case where all points are in convex position, we cannot just maintain the 3-d convex hull explicitly: although a random vertex has constant average degree (since the total degree is linear), we have no control over which vertex will be deleted next and it may have high degree. Thus, the convex hull could experience large structural change in every deletion. One idea is to initially remove all "bad" vertices of degree higher than a certain constant (since there are only a fraction of such vertices) and handle them in a separate subset (since we can answer extreme-point queries for each subset and combine the answers). This idea needs refinement, however, since the low-degree vertices themselves may have high degrees in their own convex hull. We end up using a multi-layer approach and a quantity different from degree (namely, the number of "conflict lists" that the vertex participates in). Although the description of the final method is not long (we even provide pseudocode), it does require a number of carefully orchestrated ideas, drawing inspiration from several previous approaches, for example, one of Agarwal and Matoušek's dynamic data structures [1], the author's static data structure for halfspace range reporting [7], and Eppstein's dynamic closest pair technique [18]. (See Section 5 for more discussion.)

2 Preliminaries

As noted earlier, it suffices to consider the dynamic 3-d convex hull problem, because 2-d nearest neighbor queries reduce to 3-d extreme-point queries. By symmetry, it suffices to consider the upper hull. The dual version [3, 27, 29] of the problem is more convenient to study and is equivalent to *vertical ray shooting in a lower envelope of planes*:

> Maintain a dynamic set H of n planes in 3-d so that given a vertical line q, we can quickly report the lowest plane at q, denoted by $\textsc{ans}(H, q)$.

We assume that the input planes are in general position, by standard perturbation techniques.

Our method is self-contained with the exception of an optimal static solution and one well-known geometric tool—cuttings. Given a set H of n planes and a region γ, a $(1/r)$-*cutting of* γ is a collection of disjoint open cells (tetrahedra) such that the union of the closure of the cells contains γ and each cell intersects at most $\lceil n/r \rceil$ planes of H. The *conflict list* of a cell Δ refers to the subset of all planes of H intersecting Δ and is denoted by H_Δ.

The $(\leq k)$-*level* refers to the closure of the region of all points q such that at most k planes of H are strictly below q. The following lemma states that a cutting of small size exists covering a large number of levels.

Lemma 2.1. *For any set H of n planes and parameter r, there exists an $O(1/r)$-cutting of the $(\leq n/r)$-level of H into $O(r)$ cells. We can construct this cutting, along with the conflict lists of all cells, in $O(n \log n)$ expected time. The cells are all vertical prisms (tetrahedra with a vertex at $(0, 0, -\infty)$).*

The above is a special case of Matoušek's "shallow cutting lemma" [22]. The proof was by sampling: roughly speaking, take a random sample $R \subseteq H$ of size about r, form a canonical triangulation of the arrangement of R, and return the cells that intersect the $(\leq n/r)$-level; the conflict list of each cell would then have size close to $O(n/r)$ (extra steps are required to make it truly $O(n/r)$). The $O(n \log n)$ expected running time was shown by Ramos [30]. That vertical cells are sufficient for this particular case of the shallow cutting lemma was observed earlier by the author [8, Lemma 3.1].

Our solution to the dynamic lower envelope problem is described in two sections. Section 3 contains a deletion-only data structure that solves the problem only "partially" for some queries; this section is where Lemma 2.1 comes into play. (Incidentally, Lemma 2.1 is the only place that requires randomization.) Section 4 then uses this intermediate solution to obtain the fully dynamic data structure; this section is essentially geometry-free.

3 A Partial Data Structure That Supports Deletions

We first describe an incomplete data structure that supports deletions for a set S of n planes. The data structure is incomplete in the sense that it can only handle queries when the answers belong to a certain subset $S_{\text{live}} \subseteq S$ of "live" members. Initially, we guarantee that at least half of the planes survive in S_{live}. Subsequently, each deletion can "kill" off only a small (polylogarithmic) number of additional planes from S_{live}. A static subset S_{static} is used to handle queries.

The construction of the data structure consists of logarithmically many layers, where in each layer we take a cutting of a certain size and remove "bad" planes that cause too many conflicts. The precise pseudocode is given below. (All logarithms are in base 2, and c and c' are appropriate constants.)

construct(S):
1. $S_0 = S$
2. for $i = 1, \dots, \lceil \log n \rceil$ do
3. $\quad T_i = (1/2^i)$-cutting of the $(\leq \lceil |S_{i-1}|/c2^i \rceil)$-level of S_{i-1} into $c'2^i$ vertical cells
4. $\quad S_i = S_{i-1} - \{$ all planes that intersect more than $2c'\lceil \log n \rceil$ cells of T_i $\}$
5. $\quad$for each $\Delta \in T_i$ do
6. $\quad\quad$compute conflict list $(S_i)_\Delta$ and set $j_\Delta = 0$
7. $S_{\text{live}} = S_{\text{static}} = S_{\lceil \log n \rceil}$

Each deletion of a plane—live or dead—involves examining the conflict lists that the plane participates in, and killing off members from these lists in a "gradual" way. (The motive behind this unusual strategy will be revealed later, in the proof of Lemma 3.1(c).)

delete(S, h):
1. remove h from S and (if in S_{live}) from S_{live}
2. for each i and each $\Delta \in T_i$ with $h \in (S_i)_\Delta$ do
3. $\quad$remove the $(j_\Delta + 1)$-st plane,...,$(j_\Delta + 4c)$-th plane of the list $(S_i)_\Delta$ from S_{live} (if in S_{live})
4. $\quad j_\Delta = j_\Delta + 4c$

We establish the key properties of this data structure:

LEMMA 3.1.

(a) construct *takes* $O(n \log^2 n)$ *expected time and puts at least* $n/2$ *planes in* S_{live}.

(b) *Each* delete *takes* $O(\log^2 n)$ *time and removes at most* $O(\log^2 n)$ *planes from* S_{live}.

(c) *For any vertical line* q, *if* $\text{ANS}(S, q) \in S_{\text{live}}$, *then* $\text{ANS}(S, q) = \text{ANS}(S_{\text{static}}, q)$.

Proof.

(a) Because the total number of intersecting pairs of planes of S_i and cells of T_i is less than $c'2^i \cdot n/2^i = c'n$, line 4 removes at most $n/(2\lceil \log n \rceil)$ planes per iteration. Thus, the total number of planes removed is at most $n/2$. The time bound follows by applying Lemma 2.1 for each iteration.

(b) For each i with $h \in S_i$, h intersects at most $2c'\lceil \log n \rceil$ cells $\Delta \in T_i$, and for each such cell, line 3 removes at most $4c$ planes. Thus, the total number of planes removed is $O(\log^2 n)$.

(c) Let $h^* = \text{ANS}(S, q) \in S_{\text{live}}$ and let $z^* = q \cap h^*$. If z^* lies inside some $\Delta \in T_{\lceil \log n \rceil}$, then since $|(S_{\text{static}})_\Delta| \leq 1$, $h^* = \text{ANS}(S_{\text{static}}, q)$ and we are done. Otherwise, let i be such that z^* lies inside some $\Delta \in T_i$ but outside all cells of T_{i+1} (as in Figure 1). Note that $S_{\text{live}} \subseteq S_{\text{static}} \subseteq S_i$. Since T_{i+1} contains the $(\leq \lceil |S_i|/c2^{i+1} \rceil)$-level of S_i, the number of planes of S_i that are strictly below z^* is at least $\lceil |S_i|/c2^{i+1} \rceil \geq \lceil n/c2^{i+2} \rceil$. In order for z^* to be the answer, all these planes must have been deleted from S. However, each such deletion ensures that the "next" $4c$ planes of $(S_i)_\Delta$ are not in S_{live}, until all planes of $(S_i)_\Delta$ are not in S_{live}. Since $|(S_i)_\Delta| \leq \lceil n/2^i \rceil \leq 4c \cdot \lceil n/c2^{i+2} \rceil$, after these deletions, no planes of $(S_i)_\Delta$ can indeed remain in S_{live}. In particular, $h^* \notin S_{\text{live}}$: a contradiction. $\square$

4 The Fully Dynamic Data Structure

We are now ready to present our data structure that supports both insertions and deletions for a dynamic set H of planes. We maintain a collection of $O(\log n)$

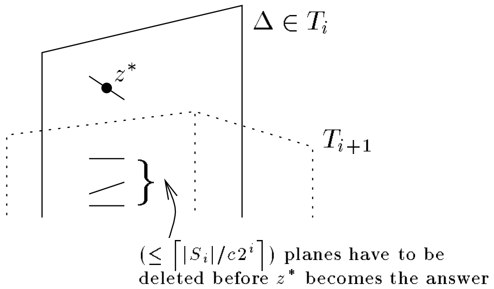

Figure 1: Proof of Lemma 3.1(c).

subsets S, each of which is stored in the data structure from Section 3. At all times, we maintain the invariant that the subsets S_{live} are disjoint with $\bigcup_S S_{\text{live}} = H$. However, the subsets S themselves (or S_{static} for that matter) are not necessarily disjoint, because of the presence of dead planes. For each subset S in the collection, we store the static lower envelope of S_{static}.

The initial data structure can be built by invoking the construction procedure from Section 3 logarithmically many times until every plane is live in some subset, as required by the invariant:

preprocess(S):
1. if $S = \emptyset$ return
2. add subset S
3. call construct(S) and preprocess($S - S_{\text{live}}$)

To insert a new plane to H, we simply add a new subset of size 1 to the collection. To ensure that the number of subsets is logarithmic, we "clean up" by repeatedly merging subsets of similar sizes (and eliminating dead planes encountered along the way). For a subset S, we define its *depth* to be $\lfloor \log |S_{\text{live}}| \rfloor$. Set $b = 16$.

insert(h):
1. add subset $S = \{h\}$ with $S_{\text{live}} = S_{\text{static}} = S$
2. while there exist b subsets $S^{(1)}, \ldots, S^{(b)}$ at the same depth do
3. remove subsets $S^{(1)}, \ldots, S^{(b)}$
4. call preprocess($S^{(1)}_{\text{live}} \cup \cdots \cup S^{(b)}_{\text{live}}$)

After the clean-up in lines 2–4 above, there can be at most $b-1$ subsets at each depth, for a total of $O(\log n)$ subsets, as claimed.

To delete a plane h from H, we invoke the deletion procedure from Section 3 to each subset S containing h. (There may be more than one subset, as h could be dead in S.) Some planes become dead in S as a result of this procedure. The key idea is to reinsert these newly deceased planes back to the data structure (into other subsets of the collection), thereby restoring the invariant.

delete(h):
1. $A = \emptyset$
2. for each subset S containing h do
3. call delete(S, h)
4. add all planes just removed from S_{live} to A
5. for each $g \in A - \{h\}$ do call insert(g)

The query algorithm is simple, as vertical ray shooting is "decomposable": just answer the query for each subset in the collection and combine the answers.

query(q):
1. $A = \emptyset$
2. for each subset S with $\textsc{ans}(S_{\text{static}}, q) \in S_{\text{live}}$ do
3. add $\textsc{ans}(S_{\text{static}}, q)$ to A
4. return $\textsc{ans}(A, q)$

The description of our data structure is complete. We now provide an amortized analysis:

Theorem 4.1.

(a) *Any sequence of* n_I insert *and* n_D delete *operations on an initial set of size* n_0 *takes total expected update time* $O(n_0 \log^2 n + n_I \log^3 n + n_D \log^6 n)$, *where* n *is the maximum size of the set at any time.*

(b) query *answers vertical ray shooting queries correctly in* $O(\log^2 n)$ *time.*

Proof.

(a) Define the *potential* of the data structure to be $\sum_S |S_{\text{live}}| \log |S_{\text{live}}|$. At any moment, the potential is upper-bounded by $n \log n$ due to disjointness of the subsets S_{live}.

The cost of preprocessing a set of n_0 planes is, by Lemma 3.1(a), $O(\sum_{j=0}^{\infty} (n_0/2^j) \log^2(n_0/2^j)) = O(n_0 \log^2 n_0)$.

Regarding insertion, consider one iteration of line 2. Say the sets $S^{(i)}$ are at depth k. The cost of this step is $O(p \log^2 p)$, like above, with $p = |S^{(1)}_{\text{live}}| + \cdots + |S^{(b)}_{\text{live}}|$. Note that $p \ge 2^{k+4}$ for our choice of $b = 16$. The increase in the potential during this step is at least

$$\begin{aligned} &\sum_{j=1}^{\infty} \frac{p}{2^j} \log \frac{p}{2^j} - \sum_{i=1}^{b} |S^{(i)}_{\text{live}}| \log |S^{(i)}_{\text{live}}| \\ &\ge\ p \log p \sum_{j=1}^{\infty} \frac{1}{2^j} - p \sum_{j=1}^{\infty} \frac{j}{2^j} - p(k+1) \\ &\ge\ p(k+4) - 2p - p(k+1)\ =\ p. \end{aligned}$$

We can thus upper-bound the insertion cost by the potential increase multiplied by a $\log^2 n$ factor.

Regarding deletion, note that line 4 adds at most $O(\log^2 n)$ planes to A per iteration, by Lemma 3.1(b), for a total of $O(\log^3 n)$ planes. The running time of lines 1–4 is simply $O(\log^3 n)$. However, the potential decreases. Noting that in general $(x+y)\log(x+y) - x\log x = y\log(x+y) + x\log(1+y/x) = y\log(x+y) + O(y)$, we see that the decrease in potential is at most

$$\begin{aligned} &|S_{\text{live}}|\log|S_{\text{live}}| - (|S_{\text{live}}| - |A|)\log(|S_{\text{live}}| - |A|) \\ &= \; O(|A|\log n) \; = \; O(\log^4 n). \end{aligned}$$

The running time of line 5 is bounded by the potential increase times $\log^2 n$, as already shown above.

Therefore, the total cost of the entire update sequence is bounded by $O(n_D \log^3 n + (\text{total potential increase})\log^2 n)$. Now,

$$\begin{aligned} &\text{total potential increase} \\ &= \; (\text{final potential} - \text{initial potential}) + \\ &\quad\; \text{total potential decrease} \\ &= \; O(n\log n - n_0 \log n_0 + n_D \log^4 n) \\ &= \; O(n_I \log n + n_D \log^4 n). \end{aligned}$$

We thus obtain the desired time bound.

(b) Correctness is easy to see: by the invariant, $\text{ANS}(H,q)$ is in S_{live} for some subset S, and by Lemma 3.1(c), $\text{ANS}(H,q) = \text{ANS}(S,q) = \text{ANS}(S_{\text{static}},q)$. The running time follows since for each of the $O(\log n)$ subsets S, $\text{ANS}(S,q)$ takes $O(\log n)$ time by known static data structures for 3-d lower envelopes (i.e., planar point location) [3, 29]. □

Note the slightly unconventional definition of n above; it is possible to make n the current size of H, by applying a standard doubling/halving amortization trick [26].

5 Remarks

Connections to previous approaches. We explain the origins of various ideas found within our data structure:

- The approach of taking $O(\log n)$ layers of samples/cuttings is akin to skip lists. More specifically, our construction in Section 3 resembles a static data structure of the author [7] for 3-d halfspace range reporting. There, conflict lists of triangulations of samples also play a vital role, but without the removal of bad planes (nor the gradual removal of planes from conflict lists).
- The idea of removing a fraction of bad planes and treating them in separate disjoint subsets stems from one of Agarwal and Matoušek's original dynamic data structures for lower envelopes [1]. Agarwal and Matoušek also used cuttings but in a top-down recursive fashion, which inevitably incurred some loss of efficiency. Our usage of cuttings of increasing sizes avoids this problem.
- The approach of decomposing a dynamic set into disjoint subsets and handling insertions by repeated merging is of course well known [2] and is sometimes called the "logarithmic method". Our data structure is more delicate, however, as we let deletions trigger multiple reinsertions (and also let dead elements remain in each subset). Traces of this nontrivial form of the logarithmic method can be seen in Eppstein's technique [18, 19] for dynamic closest-pair-type problems, and Section 4 is inspired by his technique. (Specifically, the "partial" data structure in Eppstein's closest-pair solution stores each subset as a "conga line", and a deletion triggers only one reinsertion per conga line.)

Other queries. Our data structure can handle queries other than vertical ray shooting in the lower envelope of planes. First, we can work with a dynamic *intersection of halfspaces* I instead of a lower envelope, by keeping lower halfspaces and upper halfspaces in two separate structures. We can also deal with *nonvertical ray shooting*: given a nonvertical line q that intersects I, we can return the topmost (or bottommost) point of $q \cap I$. The description and correctness proof of our query algorithm require no change at all (except for the use of the appropriate static data structures [15]).

We can answer a *membership* query (testing whether a point is inside I) by a vertical ray shooting query. We can answer a *line intersection* query (testing whether a line intersects I) by performing a ray shooting query along the line and checking whether the returned point is inside I. The query time remains $O(\log^2 n)$.

For *linear programming* queries (finding an extreme point inside I), we can use the same data structure but need to change the query algorithm. As shown by Matoušek [23], these more complicated queries reduce to ray shooting queries by a multidimensional parametric search, which increases the query time by a few logarithmic factors. It can be checked that the resulting query time is $O(\log^4 n)$. (See [5, 17] for earlier results on the semi-online case.)

Back in primal space, ray shooting and linear programming queries correspond to *gift wrapping* and *ray shooting* queries for the convex hull.

We can use also our data structure to answer *halfspace range reporting* queries (finding all points inside a query halfspace) in $O((1+k)\log^6 n)$ expected amortized time, where k denotes the output size of the query, simply by repeated deletions and reinsertions. (With more effort, the time bound can be lowered to $O(\log^2 n + k\log n)$.) Similarly, we can report the k nearest neighbors of a dynamic 2-d point set within the same bound. (See [7, 30] for earlier results on the static case.)

Applications. We list some of the many consequences of our new data structure:

- We can maintain the *smallest enclosing circle* of a dynamic 2-d point set in expected amortized time $O(\log^6 n)$, improving the previous $O(n^\varepsilon)$ bound [1]. This problem lifts to convex programming over a halfspace intersection, and can be handled again by multidimensional parametric search [23].
- We can maintain the *diameter* as well as the *bichromatic closest/farthest pair* of a dynamic 2-d point set in $O(\log^8 n)$ expected amortized time (instead of $O(n^\varepsilon)$ [1]). This follows from Eppstein's technique [18] (which increases the update time by two logarithmic factors).
- We can maintain the *Euclidean minimum spanning tree* of a dynamic 2-d point set in $O(\log^{10} n)$ expected amortized time. This is a consequence of a reduction to bichromatic closest pair, as noted by Eppstein [18] (together with known dynamic MST data structures for graphs, e.g., [21]).
- We can maintain the *all-nearest-neighbor graph* of a dynamic 2-d point set in $O(\log^6 n)$ expected amortized time. The reduction of this problem to dynamic nearest neighbors was noted previously by the author [9].
- We can maintain the *width* of a 2-d point set under *insertions* (but not deletions) in $O(\log^7 n)$ expected amortized time (instead of $O(n^\varepsilon)$), by a method of Eppstein [20] (which increases the update time by one logarithmic factor).
- We can obtain a randomized algorithm for the 3-d *convex layers* (the "convex hull peeling depth") problem that runs in $O(n\log^6 n)$ expected time. This improves Agarwal and Matoušek's $O(n^{1+\varepsilon})$ bound [1]. Our result may be compared to Chazelle's $O(n\log n)$ algorithm (1985) for the 2-d convex layers problem [10].
- We can obtain a randomized algorithm for constructing the *k-level* for a 3-d arrangement of n planes that runs in $O(n\log^2 n + f\log^6 n)$ expected time, where f denotes the output size. This improves Agarwal and Matoušek's $O(n^{1+\varepsilon}f)$ bound [1]. We can also solve the problem in $O(n\log n + nk^{3/2}\log^6 k)$ expected time by a method of the author [7] (together with the current combinatorial bound on the k-level [32]).
- We can report all *local minima* of the first k levels for a 3-d arrangement of n planes in $O(n\log^2 n + k^3\log^6 n)$ expected time, by a method of Matoušek [24].

Possible modifications and open problems. Our method is easy to implement except for the cutting step (Lemma 2.1). Unfortunately, Ramos' cutting algorithm [30] is fairly complicated. There are simpler alternatives: for example, just take a random sample R of size $O(r)$ and form the vertical decomposition of the $(\log n)$-level of R, which is provably an $O((\log^2 n)/r)$-cutting of the $(\leq n/r)$-level into $O(r\,\mathrm{polylog}\,n)$ cells, with high probability [11, 13]. This observation also implies that besides good expected time bounds, we can get good time bounds with high probability, if we don't mind a few more logarithmic factors.

There is an even simpler option that could lead to a practical implementation: just take the vertical decomposition of the lower envelope of a sample R. This envelope is only guaranteed to cover each fixed point of the $(\leq n/r)$-level with a constant probability, however. So additional modifications (for example, using multiple independent samples to lower the error probability) are required.

From a more theoretical perspective, further reductions in our time bounds would of course be worthwhile. We have not seriously attempted to minimize the number of logarithmic factors, as our initial concern is just to obtain a clean description of any polylogarithmic method. It seems likely that at least one logarithmic factor can be removed from both the insertion and deletion time, because the $O(n\log^2 n)$ construction time bound in Section 3 might be improvable: Ramos [30] actually showed how to compute $O(\log n)$ cuttings of different sizes in $O(n\log n)$ total expected time; what prevents us from immediately applying this result is that we remove bad planes each time we move on to the next cutting.

It might also be possible to de-amortize our time bounds by known tricks [26]. Our algorithm (like Agarwal and Matoušek's) uses $\Omega(n\log n)$ space (for the conflict lists in Section 3), and it is unclear whether polylogarithmic update and query time bounds are obtainable with linear space. A deterministic polylogarithmic method is also open.

Finally, we point out that Agarwal and Matoušek [1] actually described their dynamic data structures for lower envelopes in any fixed dimension. Bounds get much worse as soon as the dimension exceeds 3, even for static data structures [22, 25]. Still, one may ask whether some of the n^{ε} factors in Agarwal and Matoušek's higher-dimensional bounds can be replaced by polylogarithmic factors. Our method do not seem to yield new results here.

References

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

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

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

[4] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002.

[5] T. M. Chan. Deterministic algorithms for 2-d convex programming and 3-d online linear programming. *J. Algorithms*, 27:147–166, 1998.

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

[7] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM J. Comput.*, 30:561–575, 2001.

[8] T. M. Chan. Low-dimensional linear programming with violations. *SIAM J. Comput.*, 34:879–893, 2005.

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

[10] B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31:509–517, 1985.

[11] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[12] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3:185–121, 1993.

[13] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[14] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. *Comput. Geom. Theory Appl.*, 2:55–80, 1992.

[15] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

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

[17] D. Eppstein. Dynamic three-dimensional linear programming. *ORSA J. Comput.*, 4:360–368, 1992.

[18] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:111–122, 1995.

[19] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Experimental Algorithmics*, 5:1–23, 2000.

[20] D. Eppstein. Incremental and decremental maintenance of planar width. *J. Algorithms*, 37:570–577, 2000.

[21] J. Holm, K. de Lichtenberg, and M. Thorup. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[22] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2:169–186, 1992.

[23] J. Matoušek. Linear optimization queries. *J. Algorithms*, 14:432–448, 1993. Also with O. Schwarzkopf in *Proc. 8th ACM Sympos. Comput. Geom.*, pages 16–25, 1992.

[24] J. Matoušek. On geometric optimization with few violated constraints. *Discrete Comput. Geom.*, 14:365–384, 1995.

[25] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete Comput. Geom.*, 10:215–232, 1993.

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

[27] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

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

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

[30] E. Ramos. On range reporting, ray shooting, and k-level construction. In *Proc. 15th ACM Sympos. Comput. Geom.*, pages 390–399, 1999.

[31] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd IEEE Sympos. Found. Comput. Sci.*, pages 197–206, 1991.

[32] M. Sharir, S. Smorodinsky, and G. Tardos. An improved bound for k-sets in three dimensions. *Discrete Comput. Geom.*, 26:195–204, 2001.