Algorithmica (1995) 13: 325–345

Algorithmica
© 1995 Springer-Verlag New York Inc.

Dynamic Half-Space Range Reporting and Its Applications[1]

P. K. Agarwal[2] and J. Matoušek[3]

Abstract. We consider the half-space range-reporting problem: Given a set S of n points in $\mathbb{R}^d$, preprocess it into a data structure, so that, given a query half-space γ, all k points of $S \cap \gamma$ can be reported efficiently. We extend previously known static solutions to dynamic ones, supporting insertions and deletions of points of S. For a given parameter m, $n \le m \le n^{\lfloor d/2 \rfloor}$ and an arbitrarily small positive constant ε, we achieve $O(m^{1+\varepsilon})$ space and preprocessing time, $O((n/m^{\lfloor d/2 \rfloor}) \log n + k)$ query time, and $O(m^{1+\varepsilon}/n)$ amortized update time ($d \ge 3$). We present, among others, the following applications: an $O(n^{1+\varepsilon})$-time algorithm for computing convex layers in $\mathbb{R}^3$, and an output sensitive algorithm for computing a level in an arrangements of planes in $\mathbb{R}^3$, whose time complexity is $O((b+n) \cdot n^{\varepsilon})$, where b is the size of the level.

Key Words. Arrangements, Cuttings, Range-searching, Dynamization.

1. Introduction and Summary of Applications. We consider the *half-space range-reporting* problem: Given a set S of n points in $\mathbb{R}^d$ (d is a small constant), preprocess it so that, given a query half-space γ, the points of $S \cap \gamma$ can be reported efficiently.

For $d = 2$, a half-plane query can be answered in time $O(\log n + k)$ using $O(n)$ space and $O(n \log n)$ preprocessing, where k is the number of points reported [10]. For $d = 3$, Aggarwal *et al.* [4] have presented an $O(n \log n)$-size data structure that can answer a half-space range query in time $O(\log n + k)$ (see also [11]). For $d > 3$, Clarkson [14] gave a (randomized) solution with $O(n^{\lfloor d/2 \rfloor + \varepsilon})$ space[4] and expected preprocessing time and $O(\log n + k)$ query time. The preprocessing can be made deterministic using the results of [24]. Recently, a solution with $O(n \log n)$ preprocessing time, $O(n \log \log n)$ space, and $O(n^{1-1/\lfloor d/2 \rfloor} (\log n)^{O(1)} + k)$ query time was given in [25]. Combining these last two solutions, we can obtain a space/query time tradeoff: given a parameter m, $n \le m \le n^{\lfloor d/2 \rfloor}$, we can use $O(m^{1+\varepsilon})$ space and preprocessing to achieve $O((n/m^{1/\lfloor d/2 \rfloor}) \log n + k)$ query time. See also [8], [12], [22] for recent works on a related but more general simplex range searching problem.

---

[1] Work by the first author has been supported by National Science Foundation Grant CCR-91-06514. A preliminary version of this paper appeared in Agarwal *et al.* [2], which also contains the results of [20] on dynamic bichromatic closest pair and minimum spanning trees.

[2] Department of Computer Science, Duke University, Durham, Box 90129, NC 27708-0129, USA.

[3] Department of Applied Mathematics, Charles University, Malostranskénam 25, 118 00 Praha 1, Czech Republic.

[4] Throughout this paper, ε stands for a positive constant which can be chosen arbitrarily small with an appropriate choice of other constants in the algorithms.

Received February 19, 1992; revised December 30, 1992. Communicated by B. Chazelle.

A special case of the half-space range-reporting problems is the *half-space emptiness* problem, where we only want to decide whether the query half-space contains any point of S. For this case, Clarkson's solution [14] yields $O(\log n)$ query time with $O(n^{\lfloor d/2\rfloor+\varepsilon})$ space and preprocessing time, and Matoušek's solution [25] gives a query time of $O(n^{1-1/\lfloor d/2\rfloor}\log^{O(1)} n)$ with $O(n)$ space and $O(n \log n)$ preprocessing time. The query time of the latter algorithm can be improved to $O(n^{1-1/\lfloor d/2\rfloor}2^{O(\log^* n)})$ if we allow $O(n^{1+\varepsilon})$ preprocessing time. Numerous applications of the half-space range-reporting problem in other computational geometry problems are known and, interestingly, most of them only use the half-space emptiness variant.

All the above-mentioned data structures are static structures, that is, the set S is fixed once for all. In this paper we investigate the situation where it is necessary to modify S dynamically, which is required in many applications. Since the half-space range-reporting problem is decomposable, insertions can be handled using standard dynamization techniques (see [7] and [27]), without affecting the asymptotic running time considerably. The problem is thus how to delete points from the set. For $d = 2$, we can support deletions using the dynamic convex hull maintenance structure of Overmars and van Leeuwen [32]. For higher dimensions, Mulmuley [29] recently presented an $O(n^{\lfloor d/2\rfloor+\varepsilon})$-size data structure, that for a *random* sequence of insertions and deletions, answers an empty half-space query in $O(\log n)$ time with high probability and performs an update operation in $O(n^{\lfloor d/2\rfloor 1+\varepsilon})$ expected amortized time. (However, for some specific update sequences the performance can be bad.)

In this paper we give dynamic counterparts of the half-space range-reporting structures of [14] and [25]. If we sacrifice something in the query answering efficiency, the nearly linear space solution of [25] can be dynamized easily. However, dynamizing Clarkson's structure turns out to be more complicated; we modify it, using a technique due to Chazelle *et al.* [12]. We obtain the following results:

THEOREM 1.1. *Given a set S of n points in $\mathbb{R}^d$ $(d \geq 3)$, the half-space range-reporting problem can be solved with the following performance*:

(i) *$O(n \log n)$ space and preprocessing, $O(n^{1-1/\lfloor d/2\rfloor+\varepsilon} + k)$ query time, and $O(\log^2 n)$ amortized update time.*[5]

(ii) *$O(n^{\lfloor d/2\rfloor+\varepsilon})$ space and preprocessing, $O(\log n + k)$ query time, and $O(n^{\lfloor d/2\rfloor-1+\varepsilon})$ amortized update time.*

(iii) *For a parameter m, $n \leq m \leq n^{\lfloor d/2\rfloor}$, $O(m^{1+\varepsilon})$ space and preprocessing time,*

$$O\left(\frac{n}{m^{1/\lfloor d/2\rfloor}}\log m\right)$$

query time, and $O(m^{1+\varepsilon}/n)$ amortized update time.

---

[5] By saying that a data structure for a set S of n points has amortized update time $f(n)$, we mean that, starting with a current structure storing S, an arbitrary sequence of at most n insertions and deletions can be performed in $nf(n)$ time. We believe that with some more effort, the amortized bounds could be turned into worst-case ones (for a single update operation), using the standard techniques. However, we feel that this goal is not worth the effort at this point.

Next, we discuss several applications of these data structures to various problems (not trying to provide an exhaustive list). Most of the problems are just stated, since more detailed presentations are given in [3] and [23]. For some problems, solutions are described in the second half of this paper.

Ray Shooting. Given a convex polyhedron P in $\mathbb{R}^d$ defined as the intersection of n half-spaces, preprocess it into a data structure, so that the first intersection point of the boundary of P and a query ray can be computed quickly. In another paper [3], we describe a data structure for this problem with $O(m^{1+\varepsilon})$ space ($n \le m \le n^{\lfloor d/2 \rfloor}$), $O((n/m^{1/\lfloor d/2 \rfloor}) \log^5 n)$ query time, and $O(m^{1+\varepsilon}/n)$ amortized update time (for inserting or deleting a half-space determining P). The dynamic solution uses our dynamic half-space emptiness data structure. If the ray originates inside P, the query time can be improved by a $\log^3 n$ factor. Further minor improvements in the query time are discussed in [23]. As we will see later, several other problems can also be reduced to answering ray-shooting queries in a convex polyhedron.

Half-space range reporting has also been applied to some other ray-shooting problems, see [3].

Nearest/Farthest-Neighbor Queries. Given a set S of n points in $\mathbb{R}^d$, preprocess it into a data structure, so that the k nearest (or k farthest) neighbors of a query point can be determined quickly (k can be a part of the query). The static version of the problem is well studied [4]. Since the problem is decomposable, it can be easily dynamized efficiently if we allow only insertions, and if we are willing to pay an extra logarithmic factor in the query time.

The problem however becomes much harder if we allow deletions. For $d = 2$, the query and the update time for the best-known nearest- (or farthest-) neighbor algorithm are $O(\sqrt{n} \log n)$. (Of course, if we allow linear update time, a query can be answered in $O(\log n)$ time.) We are not aware of any efficient solution in higher dimensions except in some special cases, e.g., the closest pair in a set of n points can be maintained in $O(\log^d n \log \log n)$ amortized time [34] (see also [35]). Recently Mulmuley [29] showed that if the sequence of insertions/deletions is random, then a k-nearest-neighbor query can be answered in $O(k \log n)$ expected time, and a point can be inserted or deleted in amortized time $O(n^{\lceil d/2 \rceil - 1 + \varepsilon})$ with high probability (other variants of the algorithm give slightly better expected performance bounds, but not with high probability, see [29]). Again, the algorithm is not guaranteed to perform very well on an arbitrary sequence of insertions and deletions.

In [3] we gave a data structure of size $O(m^{1+\varepsilon})$ that could answer a k-nearest- (or k-farthest-) neighbor query in time $O((n/m^{1/\lceil d/2 \rceil}) \log^{O(1)} n + k \log^2 n)$. Using the dynamic half-space range reporting structure, we get, as usual, amortized update time $O(m^{1+\varepsilon}/n)$ for a dynamic version of this data structure. The same bounds hold if we want to determine the k farthest neighbors. Let us remark that if $m = n^{\lceil d/2 \rceil}$, the query time can be improved to $O(\log^2 n + k \log n)$, which matches Mulmuley's bound for $k = \Omega(\log n)$.

Dynamic Linear Optimization Queries. Let H be a set of n linear constraints (half-spaces) in $\mathbb{R}^d$. We wish to preprocess H, so that the optimal point with respect to a query objective function (hyperplane) can be computed efficiently.

For $d = 2$, the dynamic version of the above problem can be solved, efficiently using the Overmars and van Leeuwen algorithm for maintaining convex hulls [32]. For $d = 3$, Eppstein [19] recently gave a data structure for the special case in which the sequence of insertions/deletions is known in advance. For $d \le 4$, assuming a random sequence of updates, Mulmuley [30] gave an algorithm that answers a query in polylogarithmic time with high probability, and updates the set of hyperplanes in expected amortized time $O(\log n)$ and $O(n \log^2 n)$ for $d = 3$, 4, respectively.

It was shown in [23] that a data structure for the half-space emptiness problem (satisfying certain mild assumptions, met by our data structure) can be used for answering linear optimization queries, with a polylogarithmic number of half-space emptiness queries needed for answering an optimization query. Hence, given a parameter m, $n \le m \le n^{\lceil d/2 \rfloor}$, a data structure can be maintained for a set of n linear constraints in $\mathbb{R}^d$, with $O(m^{1+\varepsilon})$ space and $O(m^{1+\varepsilon}/n)$ amortized update time, so that the optimal solution for a query objective function can be computed in time $O((n/m^{1\lfloor d/2 \rfloor}) \log^{O(1)} n)$. The same algorithm can be used to maintain the convex hull of a set of points implicitly, so that various queries, e.g., whether a hyperplane intersects the convex hull of S, or whether a query point lies in the convex hull of S, etc., can be answered efficiently.

Smallest Enclosing Disk Maintenance. A corollary of the previous result is that a smallest enclosing disk of a set of n points in $\mathbb{R}^d$ can be maintained in amortized $O(n^{1-1/(\lceil d/2 \rceil+1)+\varepsilon})$ time per insertion or deletion of a point of S [26].

Maintaining Bichromatic Closest Pair and Diameter. Given a set of red points and another set of blue points in $\mathbb{R}^d$, maintain a closest red–blue pair of points. Agarwal *et al.* [1] gave an $O(n^{2-2/(\lceil d/2 \rceil+1)+\varepsilon})$-time algorithm to compute a bichromatic closest pair of a set of n points in $\mathbb{R}^d$. However, no efficient dynamic algorithm was known for this problem. Recently Eppstein [20] has shown that a dynamic data structure for nearest-neighbor searching can be used to maintain a bichromatic closest pair efficiently. Plugging our data structure into his algorithm, a bichromatic closest pair in $\mathbb{R}^d$ can be maintained in time $O(n^{1-2/(\lceil d/2 \rceil+1)+\varepsilon})$. It also yields a dynamic algorithm for maintaining a Euclidean minimum spanning tree of n points in the plane with $O(\sqrt{n} \log n)$ update time.

The diameter of a set S of points is the distance between the farthest pair of points in S. It is well known that the diameter of a set of n points in the plane can be computed in time $O(n \log n)$. The algorithm of Agarwal *et al.* [1] can be modified to compute the diameter as well. Recently, Chazelle *et al.* [9] presented an $O(n^{1+\varepsilon})$-time algorithm to compute the diameter of a set of points in $\mathbb{R}^3$. Supowit [36] presented a data structure that could update the diameter of a set of n points in the plane in $O(\sqrt{n} \log n)$ amortized time if only deletions were allowed. However, no efficient algorithm was known if both insertions and deletions are allowed. The algorithm of Eppstein [20] can be modified for maintaining the diameter of a set of points. In particular, the diameter of a set of n points in $\mathbb{R}^d$ can be maintained in time $O(n^{1-2/(\lceil d/2 \rceil+1)+\varepsilon})$.

Convex Layers in Dimension 3. For a finite set $S \subset \mathbb{R}^d$, its convex layers C_1,

$C_2, \ldots$ are defined inductively as follows: Let $CH(S)$ denote the set of points lying on the boundary of the convex hull of S. Let $S_0 = S$ and, for $i \geq 1$, $C_i = CH(S_{i-1})$ and $S_i = S_{i-1} - (C_i \cap S_{i-1})$. In Section 3 we present an $O(n^{1+\varepsilon})$-time algorithm for computing the convex layers of a set of n points in $\mathbb{R}^3$; the previously best-known algorithm required $O(n^{3/2} \log n)$ time.

Levels in Arrangements, Higher-Order Voronoi Diagrams. Let H be a set of n hyperplanes in $\mathbb{R}^d$. The *level* of a facet f in $\mathscr{A}(H)$ is the number of hyperplanes of H lying strictly above f. The *k-level* $\Pi_k = \Pi_k(H)$ in the arrangement of H is the set of (the closures of) facets whose level is k. Let $|\Pi_k|$ denote the total number of faces of all dimensions in Π_k (the complexity of Π_k). In Section 4 we present an output-sensitive algorithm for computing a level in dimension 3, with time complexity $O((b + n)n^\varepsilon)$, where b is the complexity of the level.

As a corollary, we can compute the kth-order Voronoi diagram of n points in the plane in time $O(n^{1+\varepsilon} k)$.

Our algorithm for computing a level can be extended to higher dimensions, though the running time becomes worse.

2. Dynamizing Half-Space Range Reporting. In this section we prove Theorem 1.1. We begin by observing that the half-space range-reporting problem is decomposable, i.e., the answers for two disjoint point sets S_1 and S_2 can be combined into an answer for $S_1 \cup S_2$ in constant time. It is known that a data structure for a decomposable problem, which supports delete operations, can be converted into another data structure that supports insertions as well [27]. In particular,

THEOREM 2.1. *Given a set S of n points, let $\Psi(S)$ be a data structure for the half-space range-reporting problem that supports deletions. Let $P(n)$ be the preprocessing time, $Q(n) + O(k)$ the query time, and $D(n)$ the amortized deletion time of $\Psi(S)$. Let β be a parameter, not necessarily constant. Then another data structure can be constructed that answers a half-space range-reporting query in time $O(k) + \sum_{i=1}^{\lceil \log_\beta n \rceil} Q(\beta^i)$, can insert a point in amortized time $O(\sum_{i=0}^{\lceil \log_\beta n \rceil} P(\beta^{i+1})/\beta^i)$, and can delete a point in amortized time $D(n) + O(\log n + P(n)/n)$. The size and preprocessing time of the new data structure remain the same.*

A proof of a similar claim is given in [27] (see also [31]), but for the sake of completeness we present the proof here.

PROOF. Let S be the set of input points in $\mathbb{R}^d$. The current set S is partitioned into $t = \lceil \log_\beta n \rceil$ subsets $S_1, \ldots, S_t$, such that S_i contains at most $(\beta - 1)\beta^i$ points. For each $i \leq t$, we maintain $\Psi(S_i)$ and a counter r_i (varying in range from 0 to $\beta - 1$). Let γ be a query half-space. To report the points of $S \cap \gamma$, we query all $\Psi(S_i)$ with γ and form the union of the (pairwise disjoint) answers.

Next consider the deletion of a point from S. As we delete points from S, we periodically reconstruct the entire structure. We remember the number of points deleted from S since it was constructed the last time, and we always reconstruct

$\Psi(S)$ anew after deleting $n'/2$ points, where n' is the number of points in S when the structure was constructed the last time (the entire structure may be reconstructed while inserting a point too; see below). When reconstructing the data structure with an n-point current set S, we let $t = \lceil \log_\beta n \rceil$. We set $S_t = S$ and $S_i = \emptyset$ for $i < t$, and construct $\Psi(S_t)$. We set the counter $r_i = 0$ for $i < t$ and $r_t = \lceil |S_t|/\beta^t \rceil$. The time spent in periodic reconstruction of the structure can be subsumed by charging an additional $O(P(n)/n)$ time to each insert/delete operation.

While deleting a point p, if the data structure is not reconstructed, we proceed as follows. We find in $O(\log n)$ time the set S_i that contains p, delete p from S_i, and then modify $\Psi(S_i)$ accordingly. The total amortized running time of a delete operation is thus at most $D(n) + O(\log n + P(n)/n)$.

We insert a point to S using the following procedure:

Algorithm INSERT (S, p)
$P \leftarrow \{p\}$, $i \leftarrow 0$
while $r_i = \beta - 1$ **do**
  $P \leftarrow P \cup S_i$
  $S_i \leftarrow \emptyset$, $r_i \leftarrow 0$
  $i \leftarrow i + 1$
end while
$S_i \leftarrow P$, $r_i \leftarrow r_i + 1$
Construct $\Psi(S_i)$

It is easily seen that $|S_i| < \beta^{i+1}$. Since we spend at most $P(\beta^{i+1})$ time in constructing S_i and each S_i is constructed only after performing n/β^i insert operations, the claim follows. □

In view of the above theorem, it suffices to describe half-space reporting data structures that support only delete operations.

2.1. The Case of Almost Linear Space. We now describe how to dynamize the data structure of [25]. The query time of our data structure is slightly worse than that of [25], but our data structure is somewhat simpler. For simplicity, let us assume that the points in S are in general position. We need some definitions.

DEFINITION 2.2. Let S be a set of n points. A hyperplane h is called *k-shallow* (with respect to S), if one of the open half-spaces determined by h contains at most k points. A *simplicial partition* for S is a collection $\Pi = \{(S_1, \Delta_1), (S_2, \Delta_2), \ldots, (S_t, \Delta_t)\}$, where $S_1, \ldots, S_t$ form a disjoint partition of S and each Δ_i is a simplex containing S_i.

We use the following result:

THEOREM 2.3 [25]. *Let S be a set of n points in $\mathbb{R}^d$ $(d \geq 3)$, and let k, r be parameters such that $k \leq n/r$, $1 \leq r < n$. Then a simplicial partition* $\Pi = \{(S_1, \Delta_1), \ldots, (S_t, \Delta_t)\}$

for S can be computed such that $t = O(r)$, $|S_i| \le 2n/r$, and that every k-shallow hyperplane intersects at most $c_1 r^{1-1/\lfloor d/2 \rfloor}$ simplices of Π for $d > 3$, and at most $c_1 \log r$ simplices for $d = 3$. Here c_1 is a constant depending only on d. If r is bounded by a constant, the computation can be performed in $O(n)$ time.

We construct a partition tree $\mathcal{T} = \mathcal{T}(S)$ on S using the above theorem. Each node v of $\mathcal{T}$ is associated with a simplex Δ_v and a subset $S_v \subseteq S$. If u is the root of $\mathcal{T}$, then $S_u = S$ and $\Delta_u = \mathbb{R}^d$.

Let v be a node of $\mathcal{T}$. If $|S_v| \le c$, for some suitable constant c, then v is a leaf (storing the list of points of S_v). Otherwise, we proceed as follows: Let r be a sufficiently large constant. We set $n_v = |S_v|$ and $k_v = n_v/r$, and construct a simplicial partition $\Pi_v = \{(S_v^1, \Delta_v^1), \ldots, (S_v^t, \Delta_v^t)\}$ for S_v as in Theorem 2.3; $t = O(r)$ and $|S_v^i| \le 2n_v/r$. We create a child w_i for each simplex Δ_v^i, and set $\Delta_{w_i} = \Delta_v^i$, $S_{w_i} = S_v^i$. We also store S_v at v. We recursively construct the structure at each w_i. The depth of $\mathcal{T}$ is obviously $O(\log n)$, it uses $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

For maintaining $\mathcal{T}$ under deletions of points from S, we also store a counter d_v with every node v. This counter is set to $|S_v|/2r$ when we build the subtree rooted at v anew. Let us describe how a point is deleted from S. Every point of S is stored in exactly one node at each level of the tree, so to delete a point p from S, we follow a path in $\mathcal{T}$ starting from its root. At each interior node $v \in \mathcal{T}$, we first delete p from S_v and decrement d_v by one. If $d_v > 0$, we proceed with deletion of p from the appropriate child of v. If $d_v = 0$, we rebuild the whole subtree rooted at v anew, using the current set S_v. At a leaf node v, we simply delete p from the list.

Since the initial value of d_v is $n_v/2r$ and the time spent in reconstructing the subtree rooted at v is $c' n_v \log n_v$ for some constant c', we can amortize the time spent in reconstruction by changing $2c'r \log n_v$ time to each delete operation. We visit $O(\log n)$ nodes of $\mathcal{T}$ to delete a point and we spend $O(1)$ time at a node v if the subtree rooted at v is not constructed, so the overall amortized time in deleting a point is $O(\log^2 n)$.

By construction, each n_v/r-shallow hyperplane with respect to the initial set of points S_v intersects the simplices of at most $\kappa = c_1 r^{1-1/\lfloor d/2 \rfloor}$ ($\kappa = c_1 \log r$ for $d = 3$) children of v. Furthermore, we construct the subtree rooted at v after deleting $n_v/2r$ points, therefore at any moment a hyperplane, $(n_v/2r)$-shallow with respect to the current set S_v, intersects the simplices of at most κ children.

Let γ be a query half-space. We search $\mathcal{T}$ in a top-down fashion, starting from the root. At each node v, we do the following: If v is a leaf, we test all points of S_v for membership in γ. If v, on the other hand, is an interior node, we classify each of its children w according to the relation of the simplex Δ_w to γ (Δ_w may lie completely outside γ, completely inside γ, or cross the bounding hyperplane h of γ). If the number of children whose simplices cross h exceeds κ, we may infer that h is not $(n_v/2r)$-shallow for the current S_v, and thus at least $n_v/2r$ points of S_v lie in γ. We can afford to test every point of S_v for membership in γ and charge the time to the reported points; each point is charged at most $2r = O(1)$ units of time. Otherwise, we recursively search at all children w for which Δ_w intersects h,

and we also report all points in children whose simplices are completely contained in γ.

The algorithm correctly reports all points of $S \cap \gamma$. Let $Q(n)$ denote the maximum query time for a partition tree $\mathscr{T}$, whose current set consists of at most n points, not counting the time spent on point membership testing at interior nodes (with more than κ children simplices crossed by h). Then

$$Q(n) \le c_1 r^{1-1/\lfloor d/2 \rfloor} Q\left(\frac{2n}{r}\right) + O(r).$$

For a large enough constant r, the solution of this recurrence is known to be $O(n^{1-1/\lfloor d/2 \rfloor + \varepsilon})$ (ε tending to 0 with $r \to \infty$). For $d = 3$, we get $Q(n) = O(n^{\varepsilon})$. Hence, we have

LEMMA 2.4. *Given a set of n points in $\mathbb{R}^d$, we can preprocess it in time and space $O(n \log n)$, so that a half-space range query can be answered in time $O(n^{1-1/\lfloor d/2 \rfloor + \varepsilon} + k)$, and that a point can be deleted in $O(\log^2 n)$ amortized time.*

Choosing $\beta = 2$ in Theorem 2.1, the above Lemma implies Theorem 1.1(i).

2.2. The Case of Logarithmic Query time. In this section we consider dynamizing the half-space range-reporting data structure due to Clarkson [14]. First we explain a dynamic data structure for the half-space emptiness problem, and then we indicate the extension needed for handling the half-space range-reporting queries.

We work in a dual setting. Let H be the set of n hyperplanes dual to the points in S. In the dual setting, answering an empty half-space query for S reduces to determining whether a query point p lies above all hyperplanes of H. We refer to the dual problem as the *upper envelope problem* for H.

Let us begin by explaining a deterministic and a slightly improved counterpart of the basic randomized construction used in Clarkson's method.

Let H be a collection of hyperplanes in $\mathbb{R}^d$, and let $k, r \le n$ be parameters. For the sake of simplicity, we again assume that the hyperplanes of H are in general position. In this situation a collection Ξ of simplices with disjoint interiors is called a $(1/r)$-*cutting* for the $(\le k)$-*level* of H, provided that the simplices of Ξ cover all points of level at most k (with respect to H, i.e., all points in $\mathbb{R}^d$ with at most k hyperplanes of H lying strictly above them), and that each simplex of Ξ is intersected by at most n/r hyperplanes of H. We apply the following result:

THEOREM 2.5 (Shallow Cutting Lemma [25]). *Let H, k, r be as above, with $k = O(n/r)$. Then a $(1/r)$-cutting Ξ for the $(\le k)$-level of H, consisting of $s(r) = O(r^{\lfloor d/2 \rfloor})$ simplices, exists. For $r \le n^{\alpha}$ (where $\alpha > 0$ is a certain constant, dependent on the dimension), such a cutting can be computed in $O(n \log r)$ time.*

For every simplex $\Delta \in \Xi$, we say that a hyperplane $h \in H$ is *relevant* for Δ if it lies above Δ or intersects Δ. Let H_{Δ} denote the collection of hyperplanes relevant for Δ. We observe that any Δ with $|H_{\Delta}| > n/r + k$ cannot contain

any point of level at most k, because at most n/r hyperplanes intersect Δ, and thus more than k of its relevant hyperplanes must lie completely above Δ. We can therefore drop Δ from Ξ. Henceforth we assume $|H_\Delta| \le n/r + k$ for every $\Delta \in \Xi$.

Clarkson's original structure for the upper envelope problem is a tree-like structure, built recursively as follows (we present it in a slightly different manner, anticipating the changes needed for dynamization): If the number of hyperplanes in H is smaller than a suitable constant, then we simply store the list of hyperplanes of H; this will be a leaf node. A query is answered by testing the query point against each hyperplane of H.

If, on the other hand, H is large, we choose a suitable parameter r (a sufficiently large constant in Clarkson's original construction), and compute a $(1/r)$-cutting Ξ for the (≤ 0)-level of H, consisting of $s(r) = O(r^{\lfloor d/2 \rfloor})$ simplices. We store the cutting Ξ at the root of the data structure, and, for every $\Delta \in \Xi$, we recursively build a subtree corresponding to the data structure for H_Δ. The space $S(n)$ occupied by this data structure obeys the recursion

$$S(n) \le O(r^{\lfloor d/2 \rfloor}) + O(r^{\lfloor d/2 \rfloor}) S\left(\frac{n}{r}\right),$$

which for a sufficiently large but constant value of r solves to $O(n^{\lfloor d/2 \rfloor + \varepsilon})$.

A query with a point p on H is answered as follows: We determine whether p belongs to some simplex of Ξ at the root of $\Psi(H)$. If there is no such simplex, then there is a hyperplane of H lying above p (since Ξ covers all points of level 0), and we can conclude that p does not lie in the upper envelope of H. Otherwise, let $\Delta \in \Xi$ be the simplex containing p. We recursively query the data structure for H_Δ with p. Since the depth of recursion is at most $O(\log n)$ and we spend a constant time at each node for finding the simplex that contains p, the overall query time is $O(\log n)$.

Let us examine what obstacles prevent a direct dynamization of this data structure (recall that we are only interested in deletions). The first one is that if we delete a hyperplane h from H, the level of some points changes from 1 to 0. Since Ξ does not cover all points of level 1 with respect to H, some points of level 0 with respect to $H \setminus \{h\}$ may not be covered by Ξ. This problem can be resolved rather easily if we use a $(1/r)$-cutting Ξ for, say, the $(\le n/r)$-level of H (instead of (≤ 0)-level). Since the level of any point changes by at most one if we delete a hyperplane, such a Ξ covers all points of level 0 with respect to the current H even after n/r deletions; then we can rebuild the data structure anew for the current H.

At this point, another and more serious problem arises. When deleting a hyperplane from the root of the data structure, we must propagate this deletion into the children of the root, their children, etc., until the leaves. It is easy to see that an average hyperplane in H intersects about $1/r$ of the simplices of the cutting Ξ, i.e., $O(r^{\lfloor d/2 \rfloor - 1})$ simplices. When deleting such a hyperplane, the deletion is propagated into $O(r^{\lfloor d/2 \rfloor - 1})$ children, and we obtain a recurrence with the "right" solution. However, some of the hyperplanes may intersect many more simplices (it can be shown that this cannot be avoided in general). If it happens and these bad hyperplanes are always among the first n/r hyperplanes deleted before the

data structure is reconstructed, we get quite a bad overall performance. We circumvent this problem using a technique borrowed from [12]. The main idea is to partition H into subsets so that the hyperplanes within each subset H_i are "good" in the sense that a $(1/r)$-cutting of size $O(r^{\lfloor d/2 \rfloor})$ for H_i exists so that each hyperplane in H_i intersects $O(r^{\lfloor d/2 \rfloor - 1})$ simplices of the cutting. We preprocess each H_i recursively.

In order to obtain $O(\log n)$ query time, we choose r to be a very small power of n (instead of a constant). If r is not constant, locating a point in a $(1/r)$-cutting Ξ is no longer trivial. We extend all the facets of the simplices in Ξ to hyperplanes, and use a point-location structure for a hyperplane arrangement with a polynomial space and logarithmic query time (such as the one in [13]), obtaining

LEMMA 2.6. *Let Ξ be a cutting in $\mathbb{R}^d$ consisting of $O(r^{\lfloor d/2 \rfloor})$ simplices. A data structure that can determine the simplex of Ξ containing the query point in $O(\log r)$ time, using $O(r^c)$ space and preprocessing (c is a constant depending only on d), exists.*

Let us give a more formal description of the dynamic data structure. Let us fix a small positive constant δ throughout the construction (the ε in the resulting performance bounds as well as the constants hidden in the asymptotic notation depend on δ).

The data structure is periodically rebuilt from scratch after deleting some of the hyperplanes present at the moment of the previous global reconstruction. We use m to denote the number of hyperplanes in H when the data structure was constructed last time, and n to denote the number of hyperplanes in the current H. We set $r = m^{\delta}$; this setting remains valid all the time between two global reconstructions. We assume that m is large enough.

We reconstruct the data structure from scratch after deleting $m/2r$ hyperplanes from H. Thus $n \geq m(1 - 1/2r)$. Let $P(m)$ denote the time spent in preprocessing H. The time spent for these global reconstructions can be amortized by charging $2rP(m - m/2r)/m \leq 2rP(n)/n$ time to each delete operation. As we will see, this contribution to the amortized deletion time is well within the claimed limit.

The data structure for H (whose substructures are also periodically reconstructed during the deletions, but the reconstruction of substructures does not change the value of r) is a recursively defined tree, denoted by $\Psi(H)$. Let us describe how a subtree for a subset $G \subseteq H$, denoted by $\Psi(G)$, is built. If $v = |G| \leq r$, then $\Psi(G)$ is a leaf node. We preprocess G in time $O(r^{\lfloor d/2 \rfloor + \varepsilon})$ for the upper envelope problem using Clarkson's static data structure. Thus, for a leaf node, the upper envelope problem for G can be solved in $O(\log r)$ time.

Let us now assume $v > r$. The root of the tree $\Psi(G)$ stores the following items:

- A partition of G into disjoint subsets $G_1, G_2, \ldots, G_t$ to be described later.
- For every $i = 1, 2, \ldots, t$, a cutting Ξ_i and a point location structure for Ξ_i as in Lemma 2.6.
- For every $i = 1, 2, \ldots, t$ and every simplex $\Delta \in \Xi_i$, a pointer to a subtree of the form $\Psi(G_{i,\Delta})$, where $G_{i,\Delta}$ is the subset of hyperplanes of G_i relevant for Δ.

- For each hyperplane $h \in G_i$, the list L_h of simplices $\Delta \in \Xi_i$ for which h is relevant.
- A counter *dcount* (used for the deletion algorithm).

After the construction of $\Psi(G)$ (before any deletions takes place), these objects have the following properties:

1. The counter *dcount* is set to $v/2r$.
2. For every i, the simplices of Ξ_i cover all points of level at most v/r with respect to G.
3. For every i and $\Delta \in \Xi_i$, $|G_{i,\Delta}| \le 2v/r$.
4. For every hyperplane $h \in G_i$, h intersects at most

$$\kappa = C_1 \cdot r^{\lfloor d/2 \rfloor - 1 + \delta} \tag{2.1}$$

simplices of Ξ_i, where C_1 is the same constant as in (2.3) below.

Notice that property 3 implies that the depth of $\Psi(G)$ is $O(\log_r v) = O(1/\delta)$.

The construction of these objects proceeds by induction on i. Suppose that collections $G_1, G_2, \ldots, G_{i-1} \subseteq G$ have already been constructed. Let $\bar{G}_i = G \setminus (G_1 \cup \cdots \cup G_{i-1})$, $v_i = |\bar{G}_i|$. Let

$$r_i = r\frac{v_i}{v} \quad \text{and} \quad k = \frac{v}{r} = \frac{v_i}{r_i}.$$

We compute a $(1/r_i)$-cutting Ξ_i of size $O(r_i^{\lfloor d/2 \rfloor})$ for the $(\le k)$-level of $\bar{G}_i$, as mentioned in Theorem 2.5, and preprocess Ξ_i for point-location queries using Lemma 2.6. For every $\Delta \in \Xi_i$, let $\bar{G}_{i,\Delta}$ denote the collection of hyperplanes of $\bar{G}_i$ relevant for Δ (recall that a hyperplane is relevant for Δ if it passes above or through Δ). As explained above, we assume

$$|\bar{G}_{i,\Delta}| \le \frac{v_i}{r_i} + k = \frac{2v_i}{r_i} = \frac{2v}{r}. \tag{2.2}$$

A hyperplane $h \in \bar{G}_i$ is called *good* if it is relevant for at most κ hyperplanes of $\bar{G}_i$. Since

$$\sum_{\Delta \in \Xi} |\bar{G}_{i,\Delta}| \le O(r_i^{\lfloor d/2 \rfloor}) \cdot \frac{2v_i}{r_i} \le C_1 v_i \left(\frac{v_i r}{v}\right)^{\lfloor d/2 \rfloor - 1}, \tag{2.3}$$

where C_1 is a constant appearing in the bound on the size of Ξ_i, the number of "bad" hyperplanes in $\bar{G}_i$ is at most

$$\frac{\left(\sum_{\Delta \in \Xi_i} |\bar{G}_{i,\Delta}|\right)}{\kappa} \le \frac{v_i}{r^\delta} \cdot \left(\frac{v_i}{v}\right)^{\lfloor d/2 \rfloor - 1} \le \frac{v_i}{r^\delta}.$$

Let G_i be the set of good hyperplanes in $\bar{G}_i$ and $\bar{G}_{i+1} = \bar{G}_i - G_i$.

If $|\bar{G}_{i+1}| > v/r$, we continue the above-described construction inductively for $i+1$. Otherwise Ξ_{i+1} consists of a sufficiently large simplex and $G_{i+1} = \bar{G}_{i+1}$. This finishes the construction of the objects stored at the root of $\Psi(G)$. The appropriate subtrees $\Psi(G_{i,\Delta})$ are constructed recursively.

The above construction guarantees $v_i/v_{i+1} \geq r^\delta$, and it finishes when $v_i \leq v/r$, therefore $t \leq 1/\delta$. (For $d > 3$, we could get a much better bound for t—about $\log_{\lfloor d/2 \rfloor}(1/\delta)$—by a more careful calculation.) It is easily seen that properties 1–4 are guaranteed by the construction.

Let us estimate the space $S(v)$ needed for $\Psi(G)$. If $v \leq r$, then the space required is $O(r^{\lfloor d/2 \rfloor + \varepsilon})$. Otherwise, we need $O(tr^c)$ space to store the point-location structures for $\Xi_1, \ldots, \Xi_t$ and $O(tvr^{\lfloor d/2 \rfloor - 1})$ space to store the lists L_h for each $h \in G$. Since $|G_{i,\Delta}| \leq 2v/r$, we get the following recurrence:

$$S(v) = \begin{cases} O(r^{\lfloor d/2 \rfloor + \varepsilon} & \text{for} \quad v \leq r, \\ O(vr^{\lfloor d/2 \rfloor - 1} + \sum_{i \leq t, \Delta \in \Xi_i} S\left(\dfrac{2v}{\gamma}\right) & \text{for} \quad v > r. \end{cases}$$

Since each Ξ_i consists of $O(r^{\lfloor d/2 \rfloor})$ simplices and t is bounded by a constant, the solution of the above recurrence is $S(v) = O((rv)^{\lfloor d/2 \rfloor + \varepsilon})$. By a similar computation, we get $P(v) = O((rv)^{\lfloor d/2 \rfloor + \varepsilon})$ for the time required for building $\Psi(G)$.

Answering a Query. We answer an upper envelope query recursively. Let p be the query point. Suppose we are at the root v of a subtree $\Psi(G)$. If v is a leaf node, we answer the query in $O(\log r)$ time using Clarkson's structure stored at v. Otherwise, we compute a partial answer A_i for every $i = 1, 2, \ldots, t$, and return YES (p does lie above all hyperplanes of G) if and only if all these answers are YES. To compute the answer A_i, we determine in time $O(\log r)$ the simplex $\Delta \in \Xi_i$ that contains p. If there is no such simplex, we can conclude that p does not lie in the upper envelope of G, and set A_i to NO. Otherwise, A_i is set to the answer returned by the data structure $\Psi(G_{i,\Delta})$.

The depth of the overall tree $\Psi(H)$ is $O(1/\delta)$ and the branching degree in this query-answering process is always at most $t = O(1)$. This implies that only $O(1)$ nodes are visited. Since we spend $O(\log r)$ time at each node, the total query time is $O(\log n)$. The correctness of the algorithm is discussed after we explain the deletion algorithm.

Deleting a Hyperplane. Let us describe the algorithm for deleting a hyperplane h from H. We visit $\Psi(H)$ in a top-down fashion, and at the root v of each subtree $\Psi(G)$ visited, we do the following: If v is a leaf, we rebuild the structure stored at v. If v is not a leaf, we first decrement the counter *dcount* stored at v by one. If *dcount* becomes zero, we rebuild $\Psi(G)$ (for the current G). Otherwise, we find the i with $h \in G_i$. We delete h from G_i, and then recursively delete h from all subtrees of the form $\Psi(G_{i,\Delta})$ with $\Delta \in L_h$. By construction, $|L_h| \leq \kappa$.

The deletion algorithm guarantees that properties 3 and 4 always hold, and 2 is replaced by a weaker one:

2′. For each i, the simplices of Ξ_i cover all points of level at most $v/2r$ with respect to the current set G.

The correctness of the query-answering algorithm follows immediately from 2′ (actually even a weaker version of 2′, where all points of level 0 are covered, would suffice; this stronger form anticipates the extension to half-space range reporting) and the above discussion.

In order to finish the proof of Theorem 1.1(ii), it suffices to estimate the amortized deletion time. To this end, let $D(v, k)$ denote the maximum time needed for deleting k hyperplanes from the data structure $\Psi(G)$ with $|G| = v$, starting at the moment $\Psi(G)$ was built anew. As noted above, a deletion of one hyperplane from G propagates into at most κ children of the root of $\Psi(G)$, thus before the next complete reconstruction of $\Psi(G)$, there are at most $(v/2r)\kappa$ deletions in the children. We thus get the following recurrences:

$$D(v, k) \le \begin{cases} O(kr^{\lfloor d/2 \rfloor + \varepsilon}) & \text{for } v \le r, \\ D\left(v, \dfrac{v}{2r}\right) + P\left(v - \dfrac{v}{2r}\right) + D\left(v - \dfrac{v}{2r}, k - \dfrac{v}{2r}\right) & \text{for } v \le r, \quad k > \dfrac{v}{2r}, \\ \max\left\{\sum_j D\left(\dfrac{2v}{r}, k_j\right) \middle| \sum_j k_j \le k\kappa\right\} + O(k \log v) & \text{for } v > r, \quad k \le \dfrac{v}{2r}. \end{cases}$$

The above recurrence implies

$$D(v, k) \le C^{\log_r v} k v^{\lfloor d/2 \rfloor - 1 + \delta} r^{\lfloor d/2 \rfloor + \varepsilon},$$

for some large enough constant C. In particular, we get $D(m, m) = O(m^{\lfloor d/2 \rfloor + \delta})$, where the constant c_2 is independent of δ.

Hence, we obtain a data structure that, using $O(n^{\lfloor d/2 \rfloor + \varepsilon})$ space and preprocessing, can answer an empty half-space query in time $O(\log n)$ and can delete a point in $O(n^{\lfloor d/2 \rfloor - 1 + \varepsilon})$ amortized time. Plugging this result into Theorem 2.1 and choosing $\beta = n^\delta$, we obtain

THEOREM 2.7. *Given a set S of n points in $\mathbb{R}^d$, we can preprocess it in $O(n^{\lfloor d/2 \rfloor + \varepsilon})$ time and space, so that an empty half-space query can be answered in $O(\log n)$ time, and a point can be inserted to or deleted from S in amortized time $O(n^{\lfloor d/2 \rfloor - 1 + \varepsilon})$.*

Half-Space Reporting. We now briefly describe how to modify the above procedure for half-space range reporting (actually its dual version—reporting hyperplanes of H lying above a query point). Let v and G be as above. We preprocess G for the half-space range-reporting problem using Clarkson's (static) data structure and store it at v as its secondary structure. It is easy to check that $S(v)$, $P(v) = O((rv)^{\lfloor d/2 \rfloor + \varepsilon})$.

When we reconstruct $\Psi(G)$, we also reconstruct the secondary structure with the current G. However, when we delete a hyperplane h we just update the primary structure as described earlier. The time spent in deleting a hyperplane obviously remains the same. Notice that we update the secondary structure only when we reconstruct $\Psi(G)$, so the secondary structure will store $\tilde{G}$, the set of hyperplanes in G when $\Psi(G)$ was constructed last time, not the current set of hyperplanes.

As for answering a query, if v is a leaf, we report, in time $O(\log \nu + k_v) = O(\log r + k_v)$, all k_v hyperplanes of G lying above p using the structure stored at v. If v is not a leaf, then, for each $i \le t$, we determine in time $O(\log r)$ the simplex $\Delta_i \in \Xi_i$ that contains p. If Δ_i is defined for all $i \le t$, then we recursively search in each $\Psi(\bar{G})$ with p. Otherwise, i.e., there is some i such that p does not lie in any simplex of Ξ_i, we query the secondary structure stored at v and compute in time $O(\log \nu + \tilde{k}_v)$ all $\tilde{k}_v$ hyperplanes of $\tilde{G}$ that lie above p. We then report the subset of these hyperplanes that have not been deleted; let k_v denote the number of hyperplanes reported.

If there is an $i \le t$ such that no simplex of Ξ_i contains p, then, by property 2′, at least $\nu/2r$ hyperplanes of (current) G lie above p, i.e., $k_v \ge \nu/2r$. We have deleted at most $\nu/2r$ points since $\Psi(G)$ was last constructed, so $\tilde{k}_v \le k_v + \nu/2r \le 2k_v$, which implies that we spent $O(\log \nu + k_v)$ time to query the secondary structure. We leave it to the reader to verify that the overall query time is $O(\log n + k)$, where k is the number of points reported. Hence, we have

LEMMA 2.8. *We can preprocess a set S of n points in $\mathbb{R}^d$ using $O(n^{\lfloor d/2 \rfloor + \varepsilon})$ space and preprocessing, so that all k points lying in query half-space can be reported in time $O(\log n + k)$ and that a point can be deleted from S in amortized time $O(n^{\lfloor d/2 \rfloor - 1 + \varepsilon})$.*

Plugging this result into Theorem 2.1 and choosing $\beta = n^{\delta}$, we obtain Theorem 1.1(ii).

It remains to establish Theorem 1.1(iii). To this end we first build a partition tree as in the proof of part (i) for S, but we terminate the recursive construction when the number of points in a node v drops below $s = (m/n)^{1/(\lfloor d/2 \rfloor - 1)}$. We then preprocess S_v using Lemma 2.8. We omit the analysis of this combined data structure.

REMARK 2.9. For $d = 3$, we do not have to combine the two data structures. We can either use Theorem 1.1(i) to get a data structure of size $O(n \log n)$, with $O(n^{\varepsilon} + k)$ query time and $O(\log^2 n)$ amortized update time, or we can use Theorem 1.1(ii) to obtain a data structure of size $O(n^{1+\varepsilon})$, with $O(\log n + k)$ query time and $O(n^{\varepsilon})$ amortized update time.

3. Computing Convex Layers. In this section we present an $O(n^{1+\varepsilon})$-time algorithm for computing the convex layers of a set of n points in $\mathbb{R}^3$. Recall that the convex layers of S are defined iteratively as follows: compute the convex hull of S, throw away the points of S lying on the boundary of the hull, and repeat the same step until S becomes empty. (See Figure 1 for an example of convex layers in the plane.)

In two dimensions the dynamic convex hull data structure of Overmars and van Leeuwen [32], which maintains a convex hull of a set of n points in $O(\log^2 n)$ time per insert or delete operation, gives an obvious algorithm for computing the convex layers in $O(n \log^2 n)$ time. Later Chazelle observed that although, in

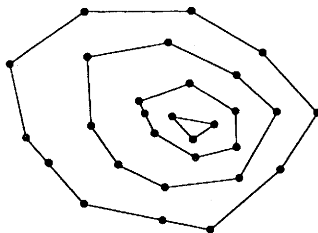

Fig. 1. Convex layers in two dimensions.

general, deleting a point requires $O(\log^2 n)$ time, deleting all points lying on the boundary convex hull can be performed in $O(\log n)$ amortized time per delete operation. Recently, Hershberger and Suri [21] have shown that if only deletions are allowed, a point can be deleted from the convex hull in $O(\log n)$ amortized time. Both of these approaches yield an optimal $O(n \log n)$-time algorithm for computing the convex layers of a set of n points in the plane.

In three dimensions an explicit representation of the convex hull cannot be maintained efficiently, since deletion or insertion of a point may require a linear number of changes in the convex hull structure. Nevertheless, our results on dynamic data structures for the half-space emptiness problem provide an implicit dynamic representation of the convex hull, supporting efficient procedures for answering various queries concerning the convex hull. These procedures yield an efficient algorithm for computing the convex layers in three dimensions. The time complexity of the previously best-known algorithm for this problem was $O(n^{3/2} \log n)$.

Our algorithm is a variant of the gift-wrapping method. For the sake of clarity, we present the algorithm in a dual setting. Let H denote the set of planes dual to the points in S, and let $U(H)$ (resp. $L(H)$) denote the upper (resp. lower) envelope of $\mathscr{A}(H)$, i.e., the boundary of the unbounded cell in the arrangement of H lying above (resp. below) all planes of H. Let H^U (resp. H^L) denote the set of planes appearing in $U(H)$ (resp. $L(H)$). The convex hull of S corresponds to $U(H)$ and $L(H)$ in the sense that there is a one-to-one mapping between the vertices (resp. edges, faces) of the convex hull of S and faces (resp. edges, vertices) of $U(H)$ and $L(H)$. In the dual setting, the problem of computing the convex layers thus reduces to computing the upper and lower envelopes of H, deleting the planes of $H^U \cup H^L$ from H, and repeating the process until H becomes empty.

Let H_i denote the set of planes deleted from H in the ith repetition of this step, i.e., the planes dual to the points of the ith layer of S. We show that after $O(n^{1+\varepsilon})$ initial preprocessing, the ith step (detection and deletion of the planes of H_i) can be accomplished in $O(|H_i| n^{\varepsilon})$ (amortized) time; this implies an $O(n^{1+\varepsilon})$ total running time of the algorithm.

We describe how to compute $U(H)$; $L(H)$ can be computed in an analogous way. For each plane $h \in H$, we pick up the half-space lying above h, and we set up a dynamic data structure for ray shooting inside the convex polyhedron determined by these half-spaces (according to [3], as explained in the introduction). The query and the amortized update time are no worse than $O(n^{\varepsilon})$. For the sake

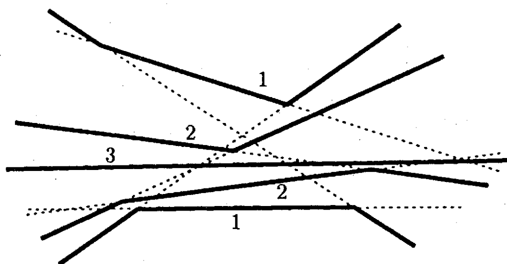

Fig. 2. Convex layers in the dual setting.

of simplicity we assume that the planes of H are in general position, that is, no three planes share a common line and no four planes share a common point.

We compute the 1-skeleton of $U(H)$ (i.e., the graph formed by the edges and vertices of $U(H)$), by traversing its edges in the depth-first manner. We maintain the set Q of vertices of $U(H)$ that we have already visited, so that we do not visit a vertex more than once.

Let us assume that we are at a vertex v of $U(H)$ formed by the intersection of three planes h_1, h_2, and h_3. We add v to Q. The three edges $e_{i,j}$ $(1 \le i < j \le 3)$ incident to v are portions of the lines $h_i \cap h_j$. Without loss of generality assume that we arrived at v through the edge $e_{1,2}$. We now have to determine the other endpoints of the edges $e_{2,3}$, $e_{1,3}$; we describe it for $e_{1,3}$. Let ρ denote the ray emanating from v and containing $e_{1,3}$; it can be computed in $O(1)$ time. We compute the first plane $h_l \in H \setminus \{h_1, h_3\}$ intersected by $\rho_{1,3}$, using our dynamic data structure: we delete h_1 and h_3 from the structure, perform a ray-shooting query with ρ, and insert h_1, h_3 back.[6] If ρ does not intersect any plane of H, $e_{1,3}$ is an unbounded edge of $U(H)$. Otherwise, $v' = h_1 \cap h_3 \cap h_l$ is the other endpoint of $e_{1,3}$. If v' has not been visited earlier, we recursively search at v', and then proceed with the other edge $e_{2,3}$ in a similar manner.

We visit a vertex of $U(H)$ only once and we perform two delete operations, two insert operations, and two ray-shooting queries at each vertex visited, so the above algorithm requires $O(n^\varepsilon)$ time per vertex of $U(H)$. Since the 1-skeleton of $U(H)$ is a planar graph, the running time is $O(|H^U| n^\varepsilon)$.

We still have to describe how to compute an initial vertex in $U(H)$ in order to start the depth-first search. We can discover such a vertex by shooting a constant number of rays: the first one, shot from plus infinity along the z-axis, finds a point p_0 on a face of $U(H)$, the next one, shot from p_0 inside that face, finds an edge, and finally shooting along that edge detects a vertex (of course, some of the rays could be unbounded; we can handle this by considering vertices and edges at infinity). We thus spend $O(n^\varepsilon)$ time in computing the initial vertex.

[6] Actually we can modify the ray-shooting structure in such a way that the deletions are unnecessary, namely, that the first plane crossed by the query ray is reported.

Hence, we can conclude that

THEOREM 3.1. *The convex layers of a set of n points in* $\mathbb{R}^3$ *can be computed in time* $O(n^{1+\varepsilon})$.

REMARK 3.2. The above technique does not give an efficient algorithm in higher dimensions, because the time spent at each layer is proportional to the number of vertices in the envelopes (not to the number of facets), and the number of vertices in a polytope defined by n hyperplanes in $\mathbb{R}^d$ can be as large as $\Omega(n^{\lfloor d/2 \rfloor})$ in the worst case.

4. Computing Levels in Plane Arrangements. Levels in hyperplane arrangements play an important role in several geometric problems, including higher-order Voronoi diagrams [17] and half-space range searching [11].

It is well known that $\sum_{j=1}^{k} |\Pi_j| = O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$ for an arrangement of n hyperplanes in $\mathbb{R}^d$ [15], and that this bound is tight in the worst case. Hence, the expected complexity of j-level with j randomly chosen in range from 1 to k is no worse than $O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil - 1})$, but only little is known about the worst-case complexity of a single level (see [17] for older results and references for dimension 2, and [6], [33], [37], [5], and [16] for recent results).

Since the complexity of a k-level varies a lot, a natural question is whether it can be computed in an output-sensitive fashion. Edelsbrunner and Welzl [18] showed that a level in an arrangement of n lines in the plane can be computed in time $O(n \log n + b \log^2 n)$, where b is the actual size of the level. Recently, Mulmuley [28] presented a randomized algorithm for computing all levels $\Pi_1, \ldots, \Pi_k$, for a given k, whose expected running time is $O(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$ for $d \geq 4$ and $O(nk^2 \log (n/k))$ for $d = 3$. We are not aware of any efficient algorithm for computing a single level in arrangements of planes in $\mathbb{R}^3$. Here we present an output-sensitive algorithm with $O(n^{\varepsilon})$ cost for every feature of the k-level. Our technique also extends to higher dimensions, though we can only obtain weaker bounds:

THEOREM 4.1.

(i) *Given a set H of n planes in* $\mathbb{R}^3$ *and an integer* $k < n$, *the k-level in the arrangement of H can be computed in time* $O((n + b)^{1+\varepsilon})$, *where b is the actual size of the level.*
(ii) *Given a collection of n hyperplanes in* $\mathbb{R}^d$ $(d \geq 4)$ *and a positive integer* $k < n$, *the k-level can be computed in time*

$$O(n^{1+\varepsilon} + \min\{bn^{1-2/(\lfloor d/2 \rfloor + 1)+\varepsilon}, b^{d/(d+1)} n^{d/(d+1)+\varepsilon}\}).$$

It is well known that computing the kth-order Voronoi diagram in $\mathbb{R}^d$ can be reduced to computing the k-level in an arrangement of n hyperplanes in $\mathbb{R}^{d+1}$. Thus, we get

Corollary 4.2. *The* kth *order Voronoi diagram of a set of* n *points in the plane can be computed deterministically in time* $O((b+n)n^{\varepsilon}) = O(n^{1+\varepsilon}k)$, *where* b *is the actual size of the diagram.*

We begin by proving Theorem 4.1(i). We basically follow the same approach as in the previous section for computing the convex layers. That is, we traverse the 1-skeleton of Π_k by following its edges in a depth-first manner. Assuming that the planes of H are in general position, every vertex of $\mathscr{A}(H)$ is the intersection point of three planes. There are three edges of Π_k incident to each vertex of Π_k of the level. Hence, having arrived at a vertex v through one of its edges, it suffices to determine the other endpoints of the remaining two edges and recursively search from these vertices, provided that they have not been visited earlier.

We use a similar mechanism as in [18]. For a vertex $v = h_1 \cap h_2 \cap h_3$ of the level, let $H_l(v)$ (resp. $H_u(v)$) denote the set of planes of H lying strictly below (resp. above) v (thus, by our general position assumptions, each plane of H except for h_1, h_2, and h_3 appears in $H_l(v)$ or $H_u(v)$). Throughout the depth-first search algorithm, we maintain the following invariant:

> Whenever we visit a vertex v, we have at our disposal a data structure $\Lambda_l(v)$ for answering ray-shooting queries inside the upper envelope of $H_l(v)$, and a similar data structure $\Lambda_u(v)$ for the lower envelope of $H_u(v)$.

Suppose that we have arrived at a vertex $v = h_1 \cap h_2 \cap h_3$ along one of its edges, $e_{1,2} \subset h_1 \cap h_2$. We query both $\Lambda_l(v)$ and $\Lambda_u(v)$ with a ray $\rho_{1,3}$ originating at v and going along $h_1 \cap h_3$ in an appropriate direction. This gives us the first plane h_2' hit by the ray $\rho_{1,3}$ (if no such plane exists, $\rho_{1,3}$ determines an unbounded edge of the 1-skeleton). We then check whether the vertex $v' = h_1 \cap h_2' \cap h_3$ has already been visited. If the answer is "no," we recursively search at v'. After returning to the vertex v, we perform a similar action with the ray $\rho_{2,3}$ originating in v and going within $h_2 \cap h_3$. After returning to v again, we return to the vertex from which we originally came to v.

It remains to show how to maintain the invariant (the ray-shooting data structures). This is quite simple: any two vertices v and v' joined by an edge share two of the triple of defining planes. Thus, when passing from a vertex $v = h_1 \cap h_2 \cap h_3$ to $v' = h_1' \cap h_2 \cap h_3$, we delete h_1' from either $\Lambda_l(v)$ (if h_1' is below v) or $\Lambda_u(v)$ (if h_1' is above v), and we insert h_1 to the appropriate one of $\Lambda_l(v)$, $\Lambda_u(v)$, obtaining $\Lambda_l(v')$ and $\Lambda_u(v')$. Hence, for the depth-first search we need to perform at most four ray-shooting queries and at most four insert/delete operations at each vertex of the 1-skeleton.

Again, we can find in $O(n^{\varepsilon})$ time the first vertex on Π_k from which we initiate the depth-first search. Hence, the total running time of the algorithm is $O((n+b)n^{\varepsilon})$, which proves Theorem 4.1(i).

The above algorithm can be extended to higher dimensions. Since the degree of each vertex of Π_k is d in $\mathbb{R}^d$, we need to perform $2(d-1)$ ray-shooting queries and four insert/delete operations per vertex. The total running time is therefore

$$O\left(m^{1+\varepsilon} + b\left(\frac{m^{1+\varepsilon}}{n} + \frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^2 n\right)\right),$$

where m is a parameter, $n \le m \le n^{\lfloor d/2 \rfloor}$. Choosing

$$m = n^{2\lfloor d/2 \rfloor/(\lfloor d/2 \rfloor + 1)},$$

the running time of the above procedure becomes

$$O(n^{1+\varepsilon} + bn^{1-2/(\lfloor d/2 \rfloor+1)}) + \varepsilon). \tag{4.1}$$

If b is large, the running time can be improved by using the same technique but a different data structure. Recall that at each step we are given a ray containing an edge of Π_k and we want to determine the first hyperplane of H hit by this ray. If we allow $O(m^{1+\varepsilon})$ $(n \le m \le n^d)$ space and preprocessing, then a ray-shooting query among a collection of n hyperplanes can be answered in time $O((n/m^{1/d}) \log^2 n)$ [3]. Although the query time of this structure is considerably higher, the advantage of this approach is that we do not perform insert/delete operations. If b is the number of vertices in Π_k, the running time is

$$O\left(m^{1+\varepsilon} + b \frac{n}{m^{1/d}} \log^2 n\right).$$

Setting $m = n^{d/(d+1)} b^{d/(d+1)}$, the running time becomes

$$O(n^{d/(d+1)+\varepsilon} b^{d/(d+1)} + n^{1+\varepsilon}). \tag{4.2}$$

Choosing the value of m in this second approach is somewhat tricky, because we do not know the value of b in advance. We guess the value of b, say $\bar{b}$. Initially, we set $\bar{b} = n^{(2d - \lfloor d/2 \rfloor + 1)/(\lfloor d/2 \rfloor + 1)}$ and run the first algorithm. If it runs for more than $c_1 n^{2d/(\lfloor d/2 \rfloor + 1)}$ steps, for some appropriate constant $c_1 > 0$, we stop. We double the value of $\bar{b}$, set $m = n^{d/(d+1)} \bar{b}^{d/(d+1)}$, and run the second algorithm. If it stops within $c_1 n^{d/(d+1)} \bar{b}^{d/(d+1)}$ steps, we are done. Otherwise, we repeat the above step. It is easily seen that if we restarted the algorithm at least once, then the total running time is the same as in (4.2). This finishes the proof of Theorem 4.1(ii).

References

[1] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete & Computational Geometry*, **6** (1991), 407–422.

[2] P. K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic half-space reporting, proximity problems, and geometric minimum spanning trees. *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, 1992, pp. 80–89.

[3] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, **22** (1993), 794–806.

[4] A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrieval problems by compacting Voronoi diagrams. *Proc. 22nd ACM Symposium on Theory of Computing*, 1990, pp. 331–340.

[5] N. Alon, I. Bárány, Z. Füredi, and D. Kleitman. Point selections and weak ε-nets for convex hulls. *Combinatorics, Probability, & Computing*, **1** (1992), 189–200.

[6] B. Aronov, B. Chazelle, H. Edelsbrunner, L. Guibas, M. Sharir, and R. Wenger. Points and

triangles in the plane and halving planes in the space. *Discrete & Computational Geometry*, **6** (1991), 435–442.

[7] J. Bentley and J. Saxe. Decomposable searching problems, I: Static-to-dynamic transformation. *Journal of Algorithms*, **1** (1980), 301–358.

[8] B. Chazelle. Lower bounds on the complexity of polytope range searching. *Journal of the American Mathematical Society*, **2** (1989), 637–666.

[9] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Diameter, width, closest line-pair, and parametric searching. *Discrete & Computational Geometry*, **10** (1993), 183–196.

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

[11] B. Chazelle and F. P. Preparata. Half-space range searching: An algorithmic application of k-sets. *Discrete & Computational Geometry*, **1** (1986), 83–93.

[12] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, **8** (1992), 407–430.

[13] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, **2** (1987), 195–222.

[14] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, **17** (1988), 830–847.

[15] K. L. Clarkson and P. Shor. New applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, **4** (1989), 387–421.

[16] T. Dey and H. Edelsbrunner. Counting triangle crossings and halving planes. *Discrete & Computational Geometry*, **12** (1994), 281–289.

[17] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, 1987.

[18] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM Journal on Computing*, **15** (1986), 271–284.

[19] D. Eppstein. Dynamic three-dimensional linear programming. *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 94–103.

[20] D. Eppstein. Fully dynamic maintenance of Euclidean minimum spanning trees and maxima of decomposable functions. *Discrete & Computational Geometry*, to appear.

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

[22] J. Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, **8** (1992), 315–334.

[23] J. Matoušek and O. Schwarzkopf. Linear optimization queries. *Proc. 8th ACM Symposium on Computational Geometry*, 1992, pp. 16–25.

[24] J. Matoušek. Approximations and optimal geometric divide-and-conquer. *Proc. 23rd ACM Symposium on Theory of Computing*, 1991, pp. 506–511.

[25] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, **2**(3) (1992), 169–186.

[26] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, **12** (1983), 720–732.

[27] K. Mehlhorn. *Multi-Dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1985.

[28] K. Mulmuley. On levels in arrangements and Voronoi diagrams. *Discrete & Computational Geometry*, **6** (1991), 307–338.

[29] K. Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling. *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 216–227.

[30] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 180–186.

[31] M. Overmars. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.

[32] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, **23** (1981), 166–204.

[33] J. Pach, W. Steiger, and E. Szemerédi. An upper bound on the number of planar k-sets. *Discrete & Computational Geometry*, **7** (1992), 109–123.

[34] M. Smid. Maintaining minimal distances of a point set in polylogarithmic time. *Discrete & Computational Geometry*, **7** (1992), 415–431.

[35] C. Schwarz and M. Smid. An $O(n \log n \log \log n)$ algorithm for online closest pair problem. *Proc. 3rd ACM–SIAM Symposium on Discrete Algorithms*, 1992, pp. 517–526.

[36] K. Supowit. New techniques for some dynamic closest-point and farthest-point problems. *Proc. 1st ACM–SIAM Symposium on Discrete Algorithms*, 1990, pp. 84–90.

[37] S. Vrećica and R. Živaljević. The colored tverberg's problem and complexes of injective functions. *Journal of Combinatorial Theory, Series A*, **61** (1992), 309–318.