OUTPUT-SENSITIVE CONSTRUCTION OF CONVEX HULLS

By

TIMOTHY MOON-YEW CHAN

B.A., Rice University, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October, 1995

Abstract

The construction of the convex hull of a finite point set in a low-dimensional Euclidean space is a fundamental problem in computational geometry. This thesis investigates efficient algorithms for the convex hull problem, where complexity is measured as a function of both the size of the input point set and the size of the output polytope.

Two new, simple, optimal, output-sensitive algorithms are presented in two dimensions and a simple, optimal, output-sensitive algorithm is presented in three dimensions. In four dimensions, we give the first output-sensitive algorithm that is within a polylogarithmic factor of optimal. In higher fixed dimensions, we obtain an algorithm that is optimal for sufficiently small output sizes and is faster than previous methods for sublinear output sizes; this result is further improved in even dimensions.

Although the focus of the thesis is on the convex hull problem, applications of our techniques to many related problems in computational geometry are also explored, including the computation of Voronoi diagrams, extreme points, convex layers, levels in arrangements, and envelopes of line segments, as well as problems relating to ray shooting and linear programming.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I wish to thank my research supervisor, Jack Snoeyink, for providing many helpful comments on various drafts of this work, sharing many enlightening discussions, and introducing me to the exciting subject of computational geometry. His guidance, encouragement, and friendship are invaluable to me. I am also much indebted to David Kirkpatrick and Nick Pippenger for their advice and support. I would also like to thank my University Examiners, Maria Klawe and Maurice Queyranne.

To the entire Computer Science Department, I would like to extend my gratitude for providing a wonderful environment for my graduate study at the University of British Columbia. The financial support of the Killam Trusts and the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

Finally, I am deeply thankful to my parents for their love, patience, and encouragement, and for all that they have done for my sake; this work is dedicated to them.

Chapter 1

Introduction

Computational geometry [Ede87, Mul93, O'Ro94, PS85] studies the design and analysis of algorithms for solving geometric problems. One central problem that has received considerable attention in the area is the problem of constructing convex hulls. The importance of the problem stems not only from its many applications (such as to pattern recognition, statistics, and image processing) but also from its usefulness as a tool for solving a variety of problems in computational geometry. The concept of convex hulls is well-studied in mathematics and is appealing both mathematically and intuitively: given a point set, its convex hull is simply the smallest convex set that encloses it.

In the following section, we define the convex hull problem in more precise terms. In Section 1.3, we describe the type of algorithms seeked in this thesis, namely, *output-sensitive algorithms.* Section 1.4 gives a brief review of previous convex hull algorithms. Finally, in Section 1.5, we outline the results obtained in this thesis.

1.1 The Convex Hull Problem

Let $P = \{p_1, \ldots, p_n\}$ be a set of n points in d-dimensional Euclidean space E^d, where $d \geq 2$ is a small fixed constant. An assumption we make throughout the thesis is that the input points are in *general position.* This means that no $d+1$ points of P lie in a common hyperplane; in certain places, we also require that no d points of P lie in a vertical hyperplane. (In this thesis, the terminology "vertical," "above/below," "upward/downward,"

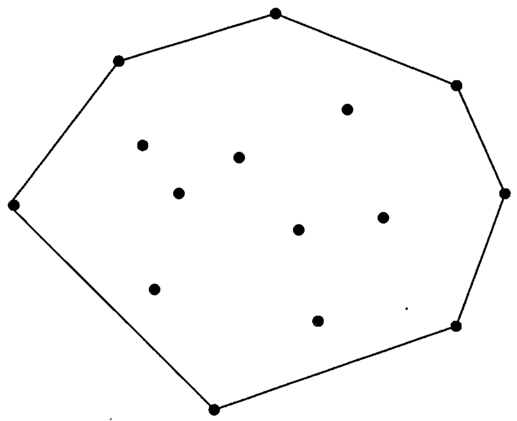

Figure 1.1: The boundary of the convex hull of a planar point set.

and "highest/lowest" are always with respect to the last coordinate.) There are general *perturbation methods* [EM90, EC92] to cope with point sets not in general position. The idea behind these methods is to eliminate degenerate configurations by applying an arbitrarily small perturbation to the input; the perturbation is done only conceptually, so each primitive operation on the perturbed input has to be simulated.

Recall that a set $S \subseteq E^d$ is *convex* if for every $p, q \in S$, the line segment $\overline{pq}$ is contained in S. The *convex hull* of P, denoted by $\mathrm{conv}(P)$, is the "smallest" convex set containing P, that is, the intersection of all convex sets containing P. Equivalently, it can be defined as the intersection of all halfspaces containing P. Alternatively, it is the set of all convex combinations of P: $\mathrm{conv}(P) = \{\sum_{i=1}^{n} \alpha_i p_i \ : \ \sum_{i=1}^{n} \alpha_i = 1, \ \alpha_1, \ldots, \alpha_n \geq 0\}$.

An intersection of a finite set of (closed) halfspaces is called a *polytope*. By a well-known fact [Grü67, MS71], $\mathrm{conv}(P)$ is a polytope. For example, if $d = 2$, it is a convex polygon (see Figure 1.1). We can represent the boundary of the polygon by its sequence of edges in, say, counterclockwise order.

In higher dimensions, we can describe the boundary of the polytope by its *faces*:

Suppose that $\mathcal{P} \subseteq E^d$ is a polytope with a non-empty interior. If h is a hyperplane that intersects the boundary of $\mathcal{P}$ but not its interior, then $h \cap \mathcal{P}$ is a *face* of $\mathcal{P}$. A face is a *j-face* ($0 \leq j < d$) if it has dimension j—that is, if it is contained in some j-flat but not in a $(j-1)$-flat. A $(d-1)$-face is called a *facet*, a $(d-2)$-face is called a *ridge*, a 1-face is called an *edge*, and a 0-face is called a *vertex*. The faces of $\mathcal{P}$, together with the empty set $\emptyset$ and $\mathcal{P}$ itself, form a lattice under inclusion, and the union of the facets is the boundary of $\mathcal{P}$. Thus, we can represent the boundary of $\mathcal{P}$ by the Hasse diagram corresponding to this lattice of faces.

The vertices of $\text{conv}(P)$ belong to the point set P and are also called the *extreme points* of P. The following statements are equivalent given a point $p \in P$: (i) p is extreme in P; (ii) $p \notin \text{conv}(P-\{p\})$; (iii) $\text{conv}(P-\{p\}) \neq \text{conv}(P)$; and (iv) there exists a vector $\xi \in E^d$ such that $\xi \cdot p > \xi \cdot q$ for all $q \in P$ $(q \neq p)$. In (iv), we say that p is *extreme/maximal along direction* ξ (or minimal along $-\xi$). The convex hull of P is the same as the convex hull of the extreme points.

"Constructing the convex hull of P" then means producing a complete representation of the boundary of $\text{conv}(P)$—the ordered sequence of edges if $d = 2$, or the facial lattice structure if $d > 2$. This construction problem is the main topic of this thesis.

A closely related and equally important problem is the computation of an intersection of a set of halfspaces $H = \{h_1, \ldots, h_n\}$. Denote this intersection by $\bigcap H$. The well-known *linear programming problem* seeks a point in $\bigcap H$ that is maximal along a given direction. In contrast, the *halfspace intersection problem* asks for a complete representation of $\bigcap H$. (In the context of linear programming, each halfspace in H is called a *(linear) constraint*, a point in $\bigcap H$ is called a *feasible solution*, and the intersection $\bigcap H$ is called the *region of feasibility*.)

To solve the halfspace intersection problem, we can first find a point in the interior of $\bigcap H$ (if the intersection is non-empty), using linear programming techniques [Meg84].

By translation, we can move this point to the origin o so that each halfspace h_i is of the form $\{x \in E^d : \xi_i \cdot x \leq 1\}$ for some vector $\xi_i \in E^d$. Then there is a one-to-one correspondence [Ede87] between the j-faces of $\bigcap H$ and the $(d-j-1)$-faces of the convex hull $\operatorname{conv}(\{\xi_1, \ldots, \xi_n\})$. This shows that computing intersections of halfspaces and computing the convex hulls are in fact equivalent problems.

Notice that we have just applied a form of *duality* (or *polarity*) when we map a halfspace $\{x \in E^d : \xi \cdot x \leq 1\}$ to a point $\xi \in E^d$. Duality [CGL85, Ede87] is extremely important in computational geometry as it allows one to transform a problem involving points to a problem involving halfspaces/hyperplanes and vice versa. Sometimes we may gain more insight into the geometry of a problem by examining the problem in both its primal and dual setting.

For a more in-depth exposition of the concepts discussed so far, we refer the reader to the standard computational geometry textbooks [Ede87, Mul93, O'Ro94, PS85].

1.2 The Number of Faces of the Convex Hull

Given a set P of n points in E^d, how many faces can $\operatorname{conv}(P)$ have? For $d = 2$, the number is clearly at most $2n$, since $\operatorname{conv}(P)$ is a polygon with at most n vertices and at most n edges. For $d = 3$, Euler's formula implies that the number of edges and facets relate linearly to the number of vertices, so the total number of faces is also $O(n)$. However, for $d = 4, 5$, the number can be quadratic in n, and for $d \geq 6$, it can be as high as $\Theta(n^{\lfloor d/2 \rfloor})$, as the following theorem shows.

Theorem 1.2.1 *Let d be a fixed constant and n be any number.*

(i) Every n-point set $P \subseteq E^d$ in general position has at most $O(n^{\lfloor d/2 \rfloor})$ faces in its convex hull.

(ii) There exists an n-point set $P \subseteq E^d$ in general position that has $\Omega(n^{\lfloor d/2 \rfloor})$ faces in its convex hull.

Proof: We first prove the upper bound in (i). Since the number of faces of the convex hull is at most 2^d times the number of facets, it suffices to bound the number of facets in conv(P). Since the number of j-faces is at most $\binom{n}{j+1} = O(n^{j+1})$ (as a j-face is incident to $j+1$ vertices), it suffices to bound the number of facets in terms of the number of $(\lfloor d/2 \rfloor - 1)$-faces in conv$(P)$. If we dualize points to halfspaces as described in Section 1.1, then our task is to bound the number of vertices in terms of the number of $\lceil d/2 \rceil$-faces in an intersection $\bigcap H$ of n halfspaces. This is done by charging vertices to $\lceil d/2 \rceil$-faces in $\bigcap H$ as follows.

Consider a vertex v of $\bigcap H$. By the general position assumption, there are precisely d edges incident to v. Orient the edges in an upward direction. Then we can find either $\lceil d/2 \rceil$ edges oriented towards v, or $\lceil d/2 \rceil$ edges oriented away from v. In the former case, these $\lceil d/2 \rceil$ edges define a $\lceil d/2 \rceil$-face σ that has v as its highest vertex. In the latter case, the $\lceil d/2 \rceil$ edges define a $\lceil d/2 \rceil$-face σ that has v as its lowest vertex. In either cases, we charge v to σ. Since each face has a unique highest/lowest vertex (assuming no degeneracies), at most two vertices are charged to a $\lceil d/2 \rceil$-face. Thus, the number of vertices is no more than twice the number of $\lceil d/2 \rceil$-faces, and (i) is proved.

To prove the lower bound in (ii), we choose a point set P on the moment curve: $P = \{M(t_1), \ldots, M(t_n)\}$, where $t_1, \ldots, t_n$ are n distinct real numbers and $M(t) = (t, t^2, \ldots, t^d) \in E^d$ for any t. Observe that a hyperplane can pass through at most d of the points in P, since a function that is linear in $t, t^2, \ldots, t^d$ is a polynomial in t of degree $\leq d$ and thus has at most d distinct zeroes. Therefore, the points in P are in general position. We now show that there are at least $\binom{n}{\lfloor d/2 \rfloor} = \Omega(n^{\lfloor d/2 \rfloor})$ faces in conv(P). In fact, we prove a stronger property (the $\lfloor d/2 \rfloor$-*neighborly* property) about the polytope conv(P): every $\lfloor d/2 \rfloor$-subset Q of P defines a ($\lfloor d/2 \rfloor - 1$)-face of conv(P).

The proof of this property is not difficult. Given a subset $Q \subseteq P$ with $|Q| \leq \lfloor d/2 \rfloor$, we need to show that there is a hyperplane h such that all points of Q lie on h and all points of $P - Q$ lie strictly on one side of h. This follows if we define the function

$$F(t) = \prod_{M(t_i) \in Q} (t - t_i)^2$$

and observe that F is linear in $t, t^2, \ldots, t^d$, $F(t_i) = 0$ for all $M(t_i) \in Q$, and $F(t_i) > 0$ for all other t_i's. $\square$

For the point set P used in the above lower-bound argument, conv(P) is called a *cyclic polytope*. Using a tighter upper-bound analysis, McMullen [McM70, MS71] in fact showed that the cyclic polytopes (or more generally, the $\lfloor d/2 \rfloor$-neighborly polytopes) attain the maximum number of j-faces among all d-dimensional, n-vertex polytopes, for any $0 < j < d$. This result is the celebrated *Upper Bound Theorem* in polytope theory.

1.3 Output-Sensitive Algorithms

Since the number of faces of the convex hull can be as large as $\Theta(n^{\lfloor d/2 \rfloor})$ by Theorem 1.2.1, any algorithm for constructing the convex hull needs to spend at least $\Omega(n^{\lfloor d/2 \rfloor})$ time in the worst case just to write out a representation of the hull. Hence, $\Omega(n^{\lfloor d/2 \rfloor})$ is automatically a lower bound for the convex hull problem.

However, for point sets that occur in practice, the convex hull has usually a small number of faces. For example, if the points in P are chosen independently at random from a uniform distribution on a convex r-gon in E^2, then the expected number of faces of the convex hull is $O(r \log n)$ [RS63]. For points chosen uniformly and independently from the interior of a d-dimensional hypercube, the expected number of vertices of the convex hull is $O(\log^{d-1} n)$ [BK+78]. For points from the interior of a d-dimensional ball, the expected number of faces is $O(n^{(d-1)/(d+1)})$ [Ray70], which is far from the pessimistic $O(n^{\lfloor d/2 \rfloor})$ bound. Even when the points are chosen on the surface of the unit paraboloid so that all points are extreme (which is the case in applications to *Voronoi diagrams* [Aur91]), the expected number of faces in the convex hull is still $O(n)$ if the points are lifted from a uniformly distributed point set in the interior of a $(d-1)$-dimensional ball [Dwy91].

In analyzing the performance of convex hull algorithms, it is therefore desirable to take into account the number of faces of the convex hull. From now on, this number will be denoted f (although in a few occasions f is also used to denote a facet; this should be clear from the context). Our goal is to develop efficient algorithms for constructing convex hulls, where the measure of efficiency is asymptotic worst-case running time as a function of both n (the *input size*) and f (the *output size*). In general, algorithms with complexity measured in terms of not only the input size, but also the output size, are said to be *output-size sensitive*, or *output-sensitive* for short. Ideally, one should need to spend less work when the output size is small.

Besides the number of faces f, a related parameter one could use to measure the complexity of convex hull algorithms is the number of hull vertices or extreme points. This is denoted by h (when it does not represent a hyperplane or halfspace; again, this should be clear from the context). While the number f ranges from $\Theta(1)$ to $\Theta(n^{\lfloor d/2 \rfloor})$, the number h ranges from $\Theta(1)$ to n. For $d \leq 3$, f and h are asymptotically equivalent, so for historical reasons, we will use h instead of f. For $d \geq 4$, we will return to using f.

Remark: As stated in the beginning of Section 1.1, we frequently need to assume that the input points are in general position for our algorithms to work properly; when this assumption does not hold, we have to rely on perturbation methods [EM90, EC92]. However, perturbing the input points may cause the output size to increase, so we need to redefine f to be the maximum number of faces of conv(P') over all "perturbations" P' of P. For $d \leq 3$, we can simply redefine h to be the number of input points on the boundary of the convex hull (since these are the only points that can become a vertex after perturbation). These redefinitions are important only when there are a large number of degeneracies. For our two- and three-dimensional convex hull algorithms, we are able to handle degenerate cases directly, so redefining h is unnecessary.

1.4 Previous Convex Hull Algorithms

The convex hull problem has had a long history going back to the beginning of computational geometry and has been an intensively studied subject even up to the present day. Early papers dealt primarily with the planar case $d = 2$. The first $O(n \log n)$-time algorithm in E^2 was given by Graham [Gra72] and is commonly known as *Graham's scan.* In terms of n alone, Graham's algorithm is (asymptotically) optimal, since an $\Omega(n \log n)$ lower bound for the convex hull problem can be obtained by a reduction from sorting. Using more complex arguments [Ben83, Yao81], the $\Omega(n \log n)$ lower bound applies also to the weaker problem of identifying the vertices of convex hull (the extreme points). However, all of these lower bound arguments assume that the number of hull vertices h is at least a fraction of n, so it is conceivable that there is an algorithm that beats Graham's scan if h is substantially smaller than n. This was indeed shown by Jarvis [Jar73], who gave a simple $O(nh)$-time algorithm, dubbed *Jarvis's march.* This algorithm is thus output-sensitive.

After the publication of Graham's and Jarvis's algorithm, a number of different convex hull algorithms (as well as variants on previous algorithms) were proposed in E^2. We mention here two divide-and-conquer algorithms [PS85]—"MergeHull" and "QuickHull"—modeled after the sorting algorithms MergeSort and QuickSort. The former divide-and-conquer algorithm, due to Preparata and Hong [PH77], runs in $O(n \log n)$ time. The latter algorithm, discovered independently by several researchers around the late 1970s, runs in $O(nh)$ time in the worst case, but is usually faster in practice (as its name suggests). At the time, no asymptotic improvement to the original bounds by Graham and Jarvis was known, so the true complexity, in terms of n and h, of the convex hull problem in E^2 remained unresolved. Finally, in 1986 Kirkpatrick and Seidel [KS86] gave the definitive answer in a paper entitled "The Ultimate Planar Convex Hull Algorithm?" by giving an output-sensitive $O(n \log h)$-time algorithm and providing a matching lower bound. It would appear that Kirkpatrick and Seidel's optimal algorithm is thus the "ultimate" convex hull algorithm for $d = 2$—or is it?

The convex hull problem in its three-dimensional setting ($d = 3$) has also been studied intensively. Preparata and Hong [PH77] presented the first $O(n \log n)$-time algorithm in E^3, based on divide-and-conquer. The first output-sensitive algorithm is the *gift-wrapping method* of Chand and Kapur [CK70], which works in arbitrary dimensions and is a generalization of Jarvis's march in two dimensions (although historically, Chand and Kapur's method appeared before Jarvis's). As analyzed by Swart [Swa85], the method runs in $O(nh)$ time.

For a long time the gift-wrapping method was the only output-sensitive algorithm known for three-dimensional convex hulls. Then, in their seminal work on randomization techniques in computational geometry [CS89], Clarkson and Shor gave an algorithm with an optimal $O(n \log h)$ expected running time, where the expectation is based solely on the random choices made by the algorithm [Mul93]. Afterwards, Edelsbrunner and Shi [ES91] proposed a deterministic algorithm with an $O(n \log^2 h)$ running time, by following the paradigm of Kirkpatrick and Seidel. An optimal $O(n \log h)$-time deterministic algorithm was finally obtained when Chazelle and Matoušek [CM95] applied recently-developed derandomization techniques to Clarkson and Shor's convex hull algorithm. (A different randomized $O(n \log h)$ algorithm was recently reported by Clarkson [Cla94] and, according to his paper, can also be derandomized.) The resulting algorithm is not very practical, since derandomization techniques tend to be complicated, using tools like the method of conditional probabilities. The problem of finding a simple optimal output-sensitive algorithm for computing convex hulls in E^3 thus remained.

For dimensions $d \geq 4$, much attention was directed to devising efficient worst-case convex hull algorithms. The gift-wrapping method by Chand and Kapur [CK70] was shown to run in $O(n^{\lfloor d/2 \rfloor + 1})$ time [Swa85] in the worst case. Seidel [Sei81] improved this time bound to $O(n^{\lceil d/2 \rceil})$, using a different approach called the *beneath-beyond method.* In a later paper [Sei86], Seidel exploited a *shelling order* to obtain a second algorithm with an $O(n^{\lfloor d/2 \rfloor} \log n)$ worst-case running time. The randomized incremental construction technique of Clarkson and Shor [CS89] yields an algorithm with expected $O(n^{\lfloor d/2 \rfloor})$ time, which is optimal for worst-case output. Another randomized algorithm with the same complexity was given by Seidel [Sei91]. A deterministic $O(n^{\lfloor d/2 \rfloor})$-time algorithm was finally obtained by Chazelle [Cha93b]; his method is based on Clarkson and Shor's randomized solution combined with new ideas on derandomization.

Although Chazelle's algorithm has settled the complexity of the convex hull problem in arbitrary fixed dimension for worst-case output, the output-sensitive complexity is far from resolved. Among the algorithms we have discussed, only the gift-wrapping method and Seidel's second deterministic algorithm are output-sensitive. For an f-face output (recall that the number f is $\Omega(1)$ and $\Theta(n^{\lfloor d/2 \rfloor})$), Swart proved an $O(nf)$-time bound for the gift-wrapping method, and Seidel proved an $O(n^2 + f \log n)$-time bound for his algorithm. Note the substantial improvement of these bounds over Chazelle's $O(n^{\lfloor d/2 \rfloor})$ bound for small values of f.

In terms of n and f, $\Omega(n \log f + f)$ is the only lower bound known (the first term is a consequence of Kirkpatrick and Seidel's two-dimensional lower bound). It is suspected that there is an algorithm with running time close to $O(n \log f + f)$, but finding one appeared difficult and remained an outstanding problem, even for $d = 4$. There were some improvements to the upper bound: Matoušek [Mat93] showed that the running time of Seidel's method [Sei86] can be brought down to $O(n^{2-2/(\lfloor d/2 \rfloor+1)+\varepsilon} + f \log n)$ if data structures for linear programming queries are used. (Here and throughout this thesis, $\varepsilon > 0$ denotes an arbitrarily small but fixed constant.) The n^ε factor can even be reduced to $\log^{O(1)} n$ using Matoušek's static structures. Further improvements seem to require new ideas.

Remark: In this brief look at previous convex hull algorithms, we have not mentioned results (e.g. [BS78, Dwy91]) that assume a certain probability distribution on the input, since we are not focussing on "average-case" complexity. Moreover, we have discussed only sequential algorithms; see [AGR94] for a recent work on parallel convex hull algorithms in a fixed dimension. We have not examined the dynamic maintenance problem [OvL81], nor considered the construction of convex hull of geometric objects other than point sets.

1.5 Results in This Thesis

In this thesis, we obtain new methods for the output-sensitive construction of convex hulls. An outline of our results is as follows. In the planar case, we discover two simple algorithms that compute the convex hull in optimal $O(n \log h)$ time. The first one can be viewed as a simplification of Kirkpatrick and Seidel's "ultimate" algorithm and the second one uses a grouping idea to speed up Jarvis's march. Considering the long history and the fundamental nature of the planar convex hull problem, it is surprising that these two simple algorithms have been left undetected. The second algorithm is also interesting from a practical viewpoint, as it does not require a linear-time subroutine for finding medians, unlike Kirkpatrick and Seidel's original solution. More importantly, this algorithm can be extended to yield an optimal $O(n \log h)$-time method for constructing convex hulls in E^3. This 3-d algorithm does not require the complex derandomization tools used by Chazelle and Matoušek's optimal algorithm, and uses relatively simple data structures, namely, Dobkin-Kirkpatrick hierarchies [DK83, DK90]. These results are described in Chapter 2.

Then in Chapter 3, we present what we regard as the central part of the thesis: an output-sensitive convex hull algorithm in E^4 that runs in $O((n+f)\log^2 f)$ time. This is the first algorithm in four dimensions that is within logarithmic factors of optimal over the whole range of output sizes f. This result has an important consequence as it immediately leads to an output-sensitive algorithm for computing three-dimensional Voronoi diagrams, which have numerous applications [Aur91, OBS92]. Although our 4-d convex hull algorithm is the most intricate algorithm we study, its guiding principle is nevertheless the same simple divide-and-conquer strategy used by our first planar convex hull algorithm.

In Chapter 4, we enter into higher-dimensional space. We observe that the gift-wrapping method can be improved using the data structures for ray shooting queries in polytopes developed by Agarwal and Matoušek [AM93] and refined by Matoušek and Schwarzkopf [MS93]. Together with the grouping idea from our second planar convex hull algorithm, this implies an $O(n \log f + (nf)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ time bound for the construction of convex hulls in E^d. If $f = O(n^{1/\lfloor d/2 \rfloor} / \log^K n)$ for a sufficiently large K, then the $O(n \log f)$ term dominates; hence, our method is optimal for small output sizes. Furthermore, our time bound improves Matoušek's previous $O(n^{2-2/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n + f \log n)$ bound for sublinear output size f, i.e., for $f = O(n/\log^K n)$. We manage to improve this result further in even dimensions by combining these higher-dimensional techniques with our 4-d divide-and-conquer algorithm. The running time obtained is $O((n + (nf)^{1-1/\lceil d/2 \rceil} + fn^{1-2/\lceil d/2 \rceil}) \log^{O(1)} n)$.

Our techniques are not limited to the convex hull problem. We apply these to many problems in computational geometry throughout the thesis. These applications include Voronoi diagrams (as mentioned above), envelopes of line segments, enumeration of extreme points, convex layers and depths of point sets, levels in arrangements of hyperplanes, and linear programming with few violated constraints. We hope that these "digressions" demonstrate the important role that convex hulls have in computational geometry.

Chapter 5 summarizes our work and concludes with open problems and remarks on directions for further research.

Remark: Most of the results of this thesis have been presented in conference papers, and their full versions have been submitted for publication in journals. See [CSY95b] for the simplification of Kirkpatrick and Seidel's algorithm and its extension to four dimensions. A dual version of the 4-d algorithm in the halfspace-intersection setting is described in [CSY95a]. Most of our higher-dimensional results appear in [Cha95b]; specialization to 2-d and 3-d can be found in [Cha95a].

Chapter 2

Two- and Three-Dimensional Convex Hulls

In this chapter, we present two $O(n \log h)$-time convex hull algorithms in the plane E^2, the second of which is also extended to E^3 with the same optimal time complexity. Although algorithms with this time complexity were known, our methods are simpler than the previous and also illustrate the basic ideas to be used in the rest of this thesis for constructing convex hulls in higher dimensions. In particular, the first planar convex hull algorithm we present, which can be considered as a simplification of the original optimal method of Kirkpatrick and Seidel [KS86], serves as the basis of the four-dimensional output-sensitive algorithm in the next chapter.

2.1 A Simplified "Ultimate Planar Convex Hull Algorithm"

This section describes a simple $O(n \log h)$ convex hull algorithm in the plane. Since our algorithm can be viewed as a simplification of Kirkpatrick and Seidel's planar convex hull algorithm, we first sketch here their method for comparison.

Given an n-point set $P \subseteq E^2$, we want to construct the convex hull of P. It suffices to compute just the *upper hull* of P, consisting of the sequence of hull edges that have an upward normal vector. Then the lower hull can be computed in a similar manner by reflection and the convex hull can be obtained by joining these two hulls.

Kirkpatrick and Seidel's algorithm constructs the upper hull of P as follows: (i) find a point $p^* \in P$ with the median x-coordinate, (ii) compute the edge $\overline{p_1 p_2}$ of the upper hull that intersects the vertical line through p^* ($x[p_1] < x[p^*] < x[p_2]$), and (iii) recursively compute the upper hull of all points left of (and including) p_1 and the upper hull of all points right of (and including) p_2.

To find the edge $\overline{p_1p_2}$ (the *bridge*) that intersects a given vertical line in step (ii), Kirkpatrick and Seidel used a prune-and-search procedure, similar to the prune-and-search linear programming algorithm of Dyer [Dye84] and Megiddo [Meg83b, Meg84]. (In fact, bridge-finding can be formulated as a linear program in dual space.) Here is a high-level description what is involved in this prune-and-search procedure: First, points are paired, the slope of the line through each pair is calculated, and the median slope m is computed. Then the upper-hull vertex p_m with a supporting line of slope m is found. A comparison involving p_m and the given vertical line is then performed, which allows one point in half of the pairs be pruned. This step eliminates 1/4 of the points and the procedure is repeated.

This ends our brief sketch of Kirkpatrick and Seidel's algorithm. As a summary, we can say that their algorithm has two levels: the lower level is a prune-and-search procedure, and on top of that is a divide-and-conquer method. Our main observation is that we can get a simpler algorithm if we combine these two levels into one, i.e., if we use pruning directly for divide-and-conquer rather than for searching. As a result, we can skip the step that computes the point p^* with median x-coordinate and avoid actually searching for the bridge at each recursive step.

2.1.1 The prune-and-divide algorithm in the plane

We now give the details of our simplified planar convex hull algorithm. As in Kirkpatrick and Seidel's algorithm, only the upper hull of the given n-point set $P \subseteq E^2$ is computed. We first pair the points of P arbitrarily and calculate the slope of the line through each pair. We then find the median slope m and compute the upper-hull vertex p_m that has a

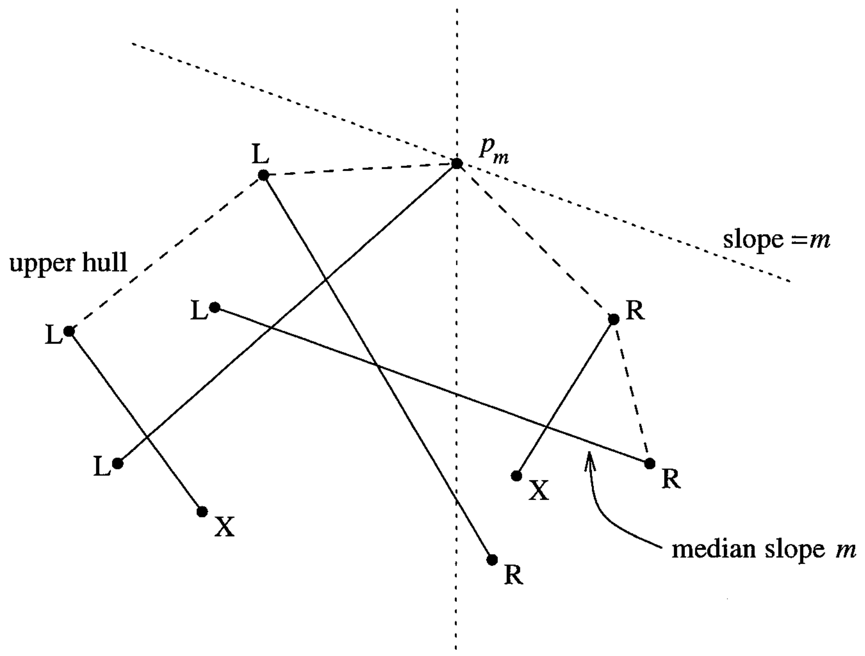

Figure 2.2: Pairing and pruning points in the plane. Points marked L belong to P_ℓ, points marked R belong to P_r, and points marked X belong to neither sets.

supporting line of slope m; this vertex can be computed by taking the maximum along a projection of P parallel to m. The x-coordinate of p_m is then used to divide P into two parts: P_ℓ, which contains p_m and all points to its left, and P_r, which contains p_m and all points to its right.

Now, if a pair has slope less than m, then the right point in the pair cannot participate in the upper hull of P_ℓ and thus can be pruned from P_ℓ. Similarly, if a pair has slope greater than m, then its left point in the pair cannot participate in the upper hull of P_r and can be pruned from P_r. Since half $(n/4)$ of the pairs have slope less than the median m and half have slope greater than m, pruning ensures that P_ℓ and P_r each contain at most $3n/4$ points. We then recursively compute the upper hull of P_ℓ and P_r. See Figure 2.2 for an example.

The pseudocode of the algorithm is given below. For convenience, we assume that the leftmost and rightmost points p_ℓ and p_r of P have been identified and we let n be the cardinality of the set $P^\bullet = P - \{p_\ell, p_r\}$ instead. In the interest of practical efficiency, line 1 has been added to the algorithm; it does not affect asymptotic worst-case performance.

```
Algorithm DivideHull2d($P^\bullet, p_\ell, p_r$)
[ Given $n$-point set $P^\bullet \subseteq E^2$ and points $p_\ell, p_r \in E^2$ such that $x[p_\ell] < x[p] < x[p_r]$ for
  all $p \in P^\bullet$, return the sequence of edges of the upper hull of $P = P^\bullet \cup \{p_\ell, p_r\}$. ]
1. discard points from $P^\bullet$ that lie below $\overline{p_\ell p_r}$
2. if $P^\bullet = \emptyset$ then return $\langle \overline{p_\ell p_r} \rangle$
   if $P^\bullet = \{p\}$ then return $\langle \overline{p_\ell p}, \overline{p p_r} \rangle$
3. arbitrarily choose $\lfloor n/2 \rfloor$ disjoint pairs $\{\{s_1, t_1\}, \ldots, \{s_{\lfloor n/2 \rfloor}, t_{\lfloor n/2 \rfloor}\}\}$ from $P^\bullet$
       and order each pair so that $x[s_i] < x[t_i]$
4. let $m_i = (y[t_i] - y[s_i]) \,/\, (x[t_i] - x[s_i])$, $i = 1, \ldots, \lfloor n/2 \rfloor$
       and $m$ = median of $\langle m_1, \ldots, m_{\lfloor n/2 \rfloor} \rangle$
5. let $p_m$ = point in $P$ that maximizes $y[p_m] - m \cdot x[p_m]$
6. let $P_\ell^\bullet = \{p \in P^\bullet : x[p] < x[p_m]\} - \{t_i : m_i \leq m\}$
       $P_r^\bullet = \{p \in P^\bullet : x[p] > x[p_m]\} - \{s_i : m_i \geq m\}$
7. if $p_m = p_r$ then return DivideHull2d($P_\ell^\bullet, p_\ell, p_r$)
   if $p_m = p_\ell$ then return DivideHull2d($P_r^\bullet, p_\ell, p_r$)
   otherwise return the concatenation of
       DivideHull2d($P_\ell^\bullet, p_\ell, p_m$) and DivideHull2d($P_r^\bullet, p_m, p_r$)
```

Remark: It is not difficult to modify `DivideHull2d()` to work for point sets P not in general position. When there are more than one point in P that maximize $y[p_m] - m \cdot x[p_m]$ in line 5, we simply pick the leftmost one. When two points in a pair share the same x-coordinate, we can eliminate the bottom one.

2.1.2 Analysis of the prune-and-divide algorithm in the plane

Let $T(n,h)$ be the running time of algorithm `DivideHull2d()` on a point set with $n+2$ points (i.e., n points excluding p_ℓ and p_r) and $h+1$ upper-hull vertices (i.e., h upper-hull edges). By noting that median-finding (line 4) can be done in linear time, we obtain the following recurrence for $T(n,h)$, where c denotes a constant:

$$T(n,h) \le \begin{cases} c & \text{if } n \le 1 \\ T(n_\ell, h) + cn & \text{if } n \ge 2 \text{ and } h_r = 0 \\ T(n_r, h) + cn & \text{if } n \ge 2 \text{ and } h_\ell = 0 \\ T(n_\ell, h_\ell) + T(n_r, h_r) + cn & \text{if } n \ge 2 \text{ and } h_\ell, h_r \ge 1 \end{cases}$$

for some $0 \le n_\ell,\ n_r \le \lceil 3n/4 \rceil$ and $h_\ell, h_r \ge 0$ with $n_\ell + n_r < n$ and $h_\ell + h_r = h$.

Using the concavity of the logarithm, one can then prove that $T(n,h) = O(n \log h)$ by induction. Here, we observe an alternative proof that is perhaps simpler as it avoids the use of induction. The proof is more general and provides better insight into recurrences of this kind by examining their *recursion trees.*

Let T be a rooted tree in which each node ν is assigned a cost $c(\nu) \in [0,\infty)$. We say that the cost function c is *α-fading* for a constant $\alpha \in (0,1)$ if $c(\mu) \le \alpha\, c(\nu)$ for every node μ and its parent ν. As part of the analysis of their 3-d output-sensitive convex hull algorithm, Edelsbrunner and Shi [ES91, Lemma 3.1] proved that the total cost in such a tree is asymptotically bounded by the per-level cost times the logarithm of the number of nodes. Their proof uses a path compression operation that transforms T into a balanced tree. We give a simple, short proof of their result that avoids path compression altogether; we then improve the bound to depend on the number of leaves rather than the number of nodes.

Lemma 2.1.1 *In a recursion tree T with m nodes and ℓ leaves and an α-fading cost function c, if the sum of the costs at each level is bounded by C, then the sum of the costs of all nodes in T is (i) at most $C(\log_{1/\alpha} m+2)$ and (ii) at most $C(\log_{1/\alpha} \ell+1+1/(1-\alpha))$.*

Proof: Number the levels of the tree 0, 1, 2, ...with the root at level zero. Let $k = \lfloor \log_{1/\alpha} m \rfloor$. The sum of the costs at levels $0, 1, \ldots, k$ is bounded by $C(k+1) \leq C(\log_{1/\alpha} m + 1)$. Furthermore, by the α-fading property, each node on a level greater than k has cost bounded by $C\alpha^{k+1} \leq C/m$; hence, the sum of the costs at level $k+1, k+2, \ldots$ is bounded by C. Part (i) follows.

To prove part (ii), we choose $k = \lfloor \log_{1/\alpha} \ell \rfloor$ instead. As before, the sum of the costs at levels $0, 1, \ldots, k$ is bounded by $C(k+1) \leq C(\log_{1/\alpha} \ell + 1)$. Thus, we just have to account for the costs of nodes at levels greater than k. Note that each node belongs to some root-to-leaf path in T. By the α-fading property, the sum of the costs at levels $k+1, k+2, \ldots$ along such a path is bounded by

$$C\alpha^{k+1} + C\alpha^{k+2} + \ldots = \frac{C\alpha^{k+1}}{1-\alpha} \leq \frac{C}{(1-\alpha)\ell}.$$

Since there are ℓ root-to-leaf paths in total, the sum of the costs at levels $k+1, k+2, \ldots$ is bounded by $C/(1-\alpha)$. Part (ii) follows. □

With Lemma 2.1.1, it is now easy to show that the running time of algorithm `DivideHull2d()` is $O(n \log h)$. Consider the recursion tree generated by the calls to `DivideHull2d()`. It is clear that the sum of the costs at each level of the tree is bounded by cn and that the cost function satisfies the (3/4)-fading property. Since the number of leaves is at most h (as a new edge is discovered at every leaf), Lemma 2.1.1(ii) immediately implies that the total cost of the algorithm is bounded by $cn \log_{4/3} h + O(n)$.

The storage requirement of the algorithm is clearly linear. We have thus shown:

Theorem 2.1.2 *Algorithm* `DivideHull2d()` *computes the* $(h+1)$*-vertex upper hull of an* $(n+2)$*-point set* $P \subseteq E^2$ *in* $O(n \log h)$ *time and* $O(n)$ *space.*

Remarks:

1. Compared to the algorithm by Kirkpatrick and Seidel, `DivideHull2d()` is faster

by a constant factor: if finding the median of n numbers takes bn time, then in the worst case, Kirkpatrick and Seidel's algorithm spends $3bn\log_2 h + O(n)$ time and our algorithm spends $\frac{1}{2}bn\log_{4/3} h + O(n) \approx 1.2bn\log_2 h + O(n)$ time in median-finding (which is the most costly operation in both algorithms).

2. Besides viewing `DivideHull2d()` as a simplification of Kirkpatrick and Seidel's algorithm, one can also view `DivideHull2d()` as a variant of the "QuickHull" algorithm [PS85], since QuickHull recursively uses an extreme vertex to divide a convex hull into two. But as pruning is not done in QuickHull, its worst-case complexity can be $\Theta(nh)$. We learned recently that Wenger [Wen94] has proposed a randomized version of QuickHull that performs pruning. His algorithm, with an $O(n\log h)$ expected running time, is similar to ours, except that finding the median slope in line 4 is replaced by randomly selecting a slope. In the next section, we give another $O(n\log h)$ algorithm that avoids median-finding but is deterministic.

2.2 An Optimal Convex Hull Algorithm in Two and Three Dimensions

We now give a different $O(n\log h)$-time convex hull algorithm in the plane, along with its extension in three dimensions. In the worst case, this 2-d algorithm is faster than the method from the previous section since this algorithm does not perform median-finding operations. The extension in 3-d is also simpler than the previous optimal derandomization method of Chazelle and Matoušek [CM95]. Our idea here is to improve Jarvis's march and the gift-wrapping method by applying a common *grouping* trick. This grouping idea can be applied to other problems besides the construction of convex hulls, and we will consider one example later in Section 2.3. Some variants of the method for convex hulls are discussed in Section 2.2.3.

2.2.1 The group-and-wrap algorithm in the plane

Let $P \subseteq E^2$ be a set of $n \geq 3$ points. The algorithm *Jarvis's march* [Jar73, O'Ro94, PS85] computes the h vertices of the convex hull one at a time, in counterclockwise (ccw) order, by a sequence of h *wrapping steps*, as follows: if p_{k-1} and p_k are the previous two vertices computed, then the next vertex p_{k+1} is set to be the point $p \in P$ that maximizes the angle $\angle p_{k-1}p_kp$ with $p \neq p_k$. One wrapping step can obviously be done in $O(n)$ time by scanning all n points; with an appropriate initialization the method constructs the entire convex hull in $O(nh)$ time.

We observe that a wrapping step can be done faster if we preprocess the points. Choose a parameter m between 1 and n and partition P into $\lceil n/m \rceil$ groups each of size at most m. Compute the convex hull of each group in $O(m \log m)$ time by, say, Graham's scan [Gra72]. This gives us $\lceil n/m \rceil$ possibly overlapping convex polygons each with at most m vertices, after a preprocessing time of $O(\frac{n}{m}(m \log m)) = O(n \log m)$. Now, a wrapping step can be done by scanning all $\lceil n/m \rceil$ polygons and computing tangents or supporting lines of the polygons through the current vertex p_k, as shown in Figure 2.3. Since tangent finding takes logarithmic time for a convex polygon by binary or Fibonacci search [CD87, PS85] (the dual problem is to intersect a convex polygon with a ray), the time required for a wrapping step is then $O(\frac{n}{m} \log m)$. As h wrapping steps are needed to compute the hull, the total time of the algorithm becomes $O(n \log m + h(\frac{n}{m} \log m)) = O(n(1 + h/m) \log m)$.

The following is the pseudocode of the algorithm just described. The procedure always runs within $O(n(1 + H/m) \log m)$ time and successfully returns the list of edges of conv(P) in ccw order when $H \geq h$.

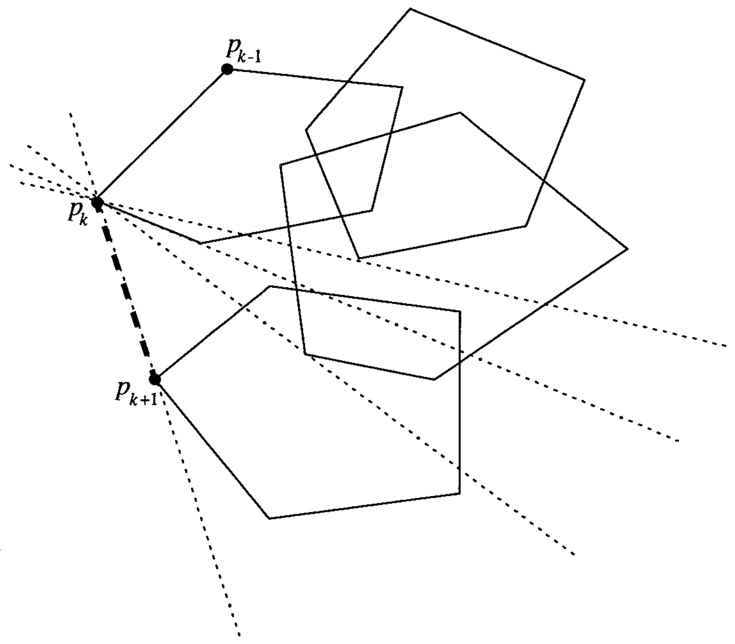

Figure 2.3: Wrapping a set of $\lceil n/m \rceil$ convex polygons of size m.

Algorithm `GroupHull2d`(P, m, H), where $P \subseteq E^2$, $3 \le m \le n$, and $H \ge 1$

```
1.  partition $P$ into subsets $P_1, \ldots, P_{\lceil n/m \rceil}$ each of size at most $m$
2.  for $i = 1, \ldots, \lceil n/m \rceil$ do
3.      compute conv$(P_i)$ by Graham's scan and store its vertices in an array
        in ccw order
4.  $p_0 \leftarrow (0, -\infty)$
5.  $p_1 \leftarrow$ the rightmost point of $P$
6.  for $k = 1, \ldots, H$ do
7.      for $i = 1, \ldots, \lceil n/m \rceil$ do
8.          compute the point $q_i \in P_i$ that maximizes $\angle p_{k-1} p_k q_i$  $(q_i \neq p_k)$
            by performing a binary search on the vertices of conv$(P_i)$
9.      $p_{k+1} \leftarrow$ the point $q$ from $\{q_1, \ldots, q_{\lceil n/m \rceil}\}$ that maximizes $\angle p_{k-1} p_k q$
10.     if $p_{k+1} = p_1$ then return $\langle \overline{p_1 p_2}, \ldots, \overline{p_{k-1} p_k}, \overline{p_k p_1} \rangle$
11. return *incomplete*
```

By choosing $m = H$, the complexity of the algorithm is then $O(n(1+H/m)\log m) = O(n\log H)$. Since the value of h is not known in advance, we use a sequence of H's to "guess" its value as shown below (the same strategy is used in Chazelle and Matoušek's algorithm [CM95]):

```
Algorithm GroupHull2d($P$), where $P \subseteq E^2$
1.  for $t = 1, 2, \ldots$ do
2.      $L \leftarrow$ GroupHull2d($P, m, H$), where $m = H = \min\{2^{2^t}, n\}$
3.      if $L \neq$ incomplete then return $L$
```

The procedure stops with the list of hull edges as soon as the value of H in the for-loop reaches or exceeds h. The number of iterations in the loop is $\lceil \log\log h \rceil$ (using base-2 logarithms), and the t-th iteration takes $O(n\log H) = O(n2^t)$ time. Therefore, the total running time of the algorithm is $O(\sum_{t=1}^{\lceil \log\log h \rceil} n2^t) = O(n2^{\lceil \log\log h \rceil + 1}) = O(n\log h)$. The storage requirement is clearly linear.

Theorem 2.2.1 `GroupHull2d()` *computes the h-vertex convex hull of an n-point set $P \subseteq E^2$ in $O(n\log h)$ time using $O(n)$ space.*

Remark: We can handle point sets that are not in general position as follows: when there are more than one point q that maximize the angle $\angle p_{k-1}p_kq$ in line 9 of `GroupHull2d`(P, m, H), pick the point q that is farthest from p_k; use the same rule to break ties in line 8.

2.2.2 The group-and-wrap algorithm in three dimensions

Let $P \subseteq E^3$ be a set of $n \geq 4$ points. Assuming that the points are in general position, we know (by Euler's formula) that there are precisely $2h-4$ facets (triangular faces) of the convex hull if there are h hull vertices. It suffices to construct these $2h-4$ hull facets; with the aid of a dictionary, we can easily generate the list of vertices and edges, together with the facial lattice structure of $\text{conv}(P)$, in additional $O(h \log h)$ time.

The higher-dimensional analogue of Jarvis's march is Chand and Kapur's *gift-wrapping method* [CK70, PS85, Swa85], which computes the hull facets one at a time as follows: from a given facet f, we generate its three adjacent facets f_j by performing a wrapping step about each of the three edges e_j of f $(j = 1, 2, 3)$. Here, a *wrapping step* about e_j is to compute a point $p_j \in P$ that maximizes the angle between f and $\text{conv}(e_j \cup \{p_j\})$ with $p_j \notin e_j$. Since such a step can be done in $O(n)$ time, we can find the facets adjacent to f in $O(n)$ time. Assuming an initial facet f_0 is given (which can be found in two wrapping steps), a breadth-first or depth-first search can then generate all facets of the convex hull. Using a dictionary to detect duplication, we can ensure that each facet is processed once. This implies that the algorithm performs $3(2h - 4)$ wrapping steps and thus runs in $O(nh)$ time.

We can use the same grouping idea from the previous subsection to improve the time complexity to optimal $O(n \log h)$ while maintaining linear space. The calls to Graham's scan (line 3 of `GroupHull2d`(P, m, H)) are now replaced by calls to Preparata and Hong's three-dimensional convex hull algorithm [PH77], which has the same complexity. To extend line 8 to three dimensions, we need to calculate tangents or supporting planes of 3-dimensional polytopes through a given line. In order to obtain the same running time as in the previous subsection, we need a method to perform each of these tangent operations in logarithmic time. A data structure that can do precisely this is the polyhedral hierarchy of Dobkin and Kirkpatrick [DK83, DK90].

A *polyhedral hierarchy* can be defined as a monotone sequence of 3-dimensional polytopes $\mathcal{P}_1 \subset \mathcal{P}_2 \subset \ldots \subset \mathcal{P}_\ell$ with the property that each connected component (or *cap*) of $\mathcal{P}_{k+1} - \mathcal{P}_k$ is of constant complexity. Each $\mathcal{P}_k$ is called a *level* of the hierarchy. If $\mathcal{P}_1$ has constant size and $\mathcal{P}_\ell = \mathcal{P}$, then the sequence is said to be an *inner hierarchical representation* of $\mathcal{P}$. Similarly, if $\mathcal{P}_1 = \mathcal{P}$ and $\mathcal{P}_\ell$ has constant size, then it is an *outer hierarchical representation* of $\mathcal{P}$. For any 3-dimensional polytope $\mathcal{P}$ with m vertices, Dobkin and Kirkpatrick showed that an inner/outer hierarchical representation of $O(\log m)$ levels exists and can be computed in $O(m)$ time. The polytope at each level is not explicitly stored in the representation; instead, pointers between two adjacent levels are provided so that one can easily traverse up or down the hierarchy.

The hierarchical representation provides a very useful data structure for manipulating with polytopes in 3-space. For instance, by "walking up" the inner hierarchy, we can find the tangent of a given polytope passing through a given line in $O(\log m)$ time. (The binary-search solution to the tangent-finding problem for convex polygons can be seen as an implicit use of the hierarchy.) By "walking down" the outer hierarchy, we can also solve the dual problem of intersecting a polytope with a ray in $O(\log m)$ time. There are many other applications of the Dobkin-Kirkpatrick hierarchy; for example, we mention a polylogarithmic algorithm by Eppstein [Epp91] for detecting whether three polytopes in E^3 have a common intersection.

We can now give the pseudocode of our three-dimensional convex hull algorithm. By using the Dobkin-Kirkpatrick hierarchical representation to store the polytopes in line 3 (which require only linear-time preprocessing), we can perform line 11 in logarithmic time for each polytope $\operatorname{conv}(P_i)$. The analysis of this algorithm is thus identical to that of the two-dimensional algorithm.

```
Algorithm GroupHull3d($P, m, H$), where $P \subseteq E^3$, $4 \le m \le n$, and $H \ge 1$
 1.  partition $P$ into subsets $P_1, \ldots, P_{\lceil n/m \rceil}$ each of size at most $m$
 2.  for $i = 1, \ldots, \lceil n/m \rceil$ do
 3.      compute conv($P_i$) by Preparata and Hong's algorithm and store it in
         a Dobkin-Kirkpatrick hierarchy
 4.  $F, Q \leftarrow \{f_0\}$, where $f_0$ is some initial facet of conv($P$)
 5.  for $k = 1, \ldots, 2H - 4$ do
 6.      if $Q = \emptyset$ then return $F$
 7.      pick some $f \in Q$ and set $Q \leftarrow Q - \{f\}$
 8.      let $e_j$ be the edges of $f$  ($j = 1, 2, 3$)
 9.      for $j = 1, 2, 3$ do
10.          for $i = 1, \ldots, \lceil n/m \rceil$ do
11.              compute the point $q_i \in P_i$ that maximizes the angle between $f$ and
                 conv($e_j \cup \{q_i\}$) by searching the hierarchy of conv($P_i$)
12.          $p_j \leftarrow$ the point $q$ from $\{q_1, \ldots, q_{\lceil n/m \rceil}\}$ that maximizes the angle between
                 $f$ and conv($e_j \cup \{q\}$)  ($q \notin e_j$)
13.          $f_j \leftarrow$ conv($e_j \cup \{p_j\}$)
14.          if $f_j \notin F$ then
15.              $F \leftarrow F \cup \{f_j\}$, $Q \leftarrow Q \cup \{f_j\}$
16.  return *incomplete*
```

We can use a queue or a stack to implement Q and a dictionary to implement F. As there are only $O(h)$ dictionary operations, they can be carried out in $O(h \log h)$ time. In fact, more clever implementations of the gift-wrapping method via a *shelling order* [Sei86] replace the need for dictionaries with just a priority queue.

As before, we choose the group size $m = H$ and guess the value of h with a sequence of H's:

```
Algorithm GroupHull3d($P$), where $P \subseteq E^3$
1.  for $t = 1, 2, \ldots$ do
2.      $L \leftarrow$ GroupHull3d($P, m, H$), where $m = H = \min\{2^{2^t}, n\}$
3.      if $L \ne$ *incomplete* then return $L$
```

Theorem 2.2.2 `GroupHull3d()` *computes the h-vertex convex hull of an n-point set $P \subseteq E^3$ in $O(n \log h)$ time using $O(n)$ space.*

Remark: We can handle point sets that are not in general position as follows: In line 8, let $e_j = \overline{a_j b_j}$ with a_j and b_j oriented in a counterclockwise order around f. When there are more than one point q that maximize the angle between f and $\text{conv}(e_j \cup \{q\})$ in line 12 of `GroupHull3d`(P, m, H), pick the point q that maximizes the angle $\angle b_j a_j q$; and if there are more than one q that achieve this maximum, pick the one farthest from a_j. Use the same rule to break ties in line 11. For degenerate point set, it is easier to keep track of edges rather than facets, since facets can be convex polygons rather than triangles. So, make F and Q sets of edges instead and in line 15, add the oriented edges $\overrightarrow{b_j a_j}$ and $\overrightarrow{a_j q}$ to F and Q. Although we may not have a complete description of the facet incident to these two edges, we know the equation of the plane containing the facet; this equation is sufficient to perform wrapping about these edges.

2.2.3 Refinements of the group-and-wrap method

In this subsection, we suggest ideas on possible improvements that may make algorithms `GroupHull2d()` and `GroupHull3d()` run even faster in practice.

Idea 1. First, points found to be in the interior of $\text{conv}(P_i)$ in line 3 of `GroupHull2d`(P, m, H) or `GroupHull3d`(P, m, H) can be eliminated from further consideration. This may potentially save work during future iterations of the algorithm, although it does not affect the worst-case complexity.

Idea 2. In `GroupHull2d`(P) and `GroupHull3d`(P), we choose the group size $m = H$ so as to balance the $O(n \log m)$ preprocessing cost and the $O(H(\frac{n}{m} \log m))$ cost for the

$O(H)$ wrapping steps. Alternatively, we can choose $m = \min\{H \log H,\ n\}$ or reversely set $H = m/\log m$. This choice of m does not affect the first cost except in the lower-order terms, but it reduces the second cost from $O(n \log H)$ to $O(n)$ and thus results in a smaller constant factor overall.

Idea 3. With Idea 2, the dominant cost of algorithm GroupHull2d(P, m, H) lies in the preprocessing, i.e., the computation of the convex hulls of the groups in line 3. To reduce this cost, we may consider reusing hulls computed from the previous iteration and merging them as the group size is increased. Suppose m' is the previous group size. Since the convex hull of two convex polygons can be computed in linear time (the dual problem is to intersect two convex polygons), we can compute the convex hull of $\lceil m/m' \rceil$ convex m'-gons in $O(m \log(m/m'))$ time by the standard "MergeHull" divide-and-conquer algorithm [PS85]. Thus, the $\lceil n/m \rceil$ hulls in line 3 can be constructed in $O(n \log(m/m'))$ rather than $O(n \log m)$ time. The same can be said for the three-dimensional case, but merging two 3-dimensional polytopes, though possible in linear time [Cha92], is more expensive.

Idea 4. In GroupHull2d(P), we use the sequence of group size $m = 2^{2^t}$, $t = 1, 2, \ldots$, to guess h. The improvements from Ideas 2 and 3 in fact permit us to choose slower growing sequences and still retain optimal $O(n \log h)$ complexity. For example, one possible sequence is simply $m = 2^t$, $t = 2, 3, \ldots$, which corresponds to doubling the group size after each iteration. Note that a coarser sequence approximates h less well while a denser sequence requires more iterations. We may try to optimize the worst-case constant factor and lower-order terms using sequences with different growth rates. We suggest the sequence $m = 2^{t^2}$, $t = 2, 3, \ldots$

Idea 5. E. Welzl has observed that the binary search in line 8 of algorithm `GroupHull2d`(P, m, H) can be replaced by a simpler linear search without changing the time complexity of the algorithm. The following monotonicity property provides the justification: during the course of the algorithm, the variable q_i in line 8 can only advance in the ccw direction along $\text{conv}(P_i)$ for each fixed i. As a result, the h-vertex convex hull of p convex polygons with a total of n vertices can be computed in $O(n + hp)$ time by gift-wrapping; the two-polygon ($p = 2$) version of the algorithm is in fact the dual of an intersection algorithm by O'Rourke et al. [OC+82] (see also [O'Ro94, PS85]). The total cost of `GroupHull2d`(P, m, H) can then be reduced to $O(n \log m + H(n/m))$ time, which is a $\log m$ factor saving in the second term. Although the overall constant factor is unaffected by the saving if Idea 2 is employed (as the first term is the dominant one), the linear search is easier to implement. There does not seem to be an analogous simplification in three dimensions.

2.3 Application: Lower Envelopes of Line Segments in the Plane

In the previous section, we have presented optimal output-sensitive convex hull algorithms in both E^2 and E^3. Besides simplicity, the approach of the previous section has the advantage that it is applicable to a variety of other problems. This section gives one application that illustrates this idea particularly well.

Consider the problem of computing the *lower envelope* $\mathcal{L}(S)$ of a set S of n line segments in the plane, which we define as the boundary of $\bigcup_{s \in S} \hat{s}$ where $\hat{s}$ is the trapezoid formed by extending the segment s vertically to $+\infty$. See Figure 2.4 for an example. (Convex hulls correspond to lower envelopes of lines in the dual.) Let h be the output size, i.e., the number of edges in the envelope; it is known that h is at most $O(n\alpha(n))$ [HS86]. Hershberger [Her89] has given a worst-case optimal algorithm that

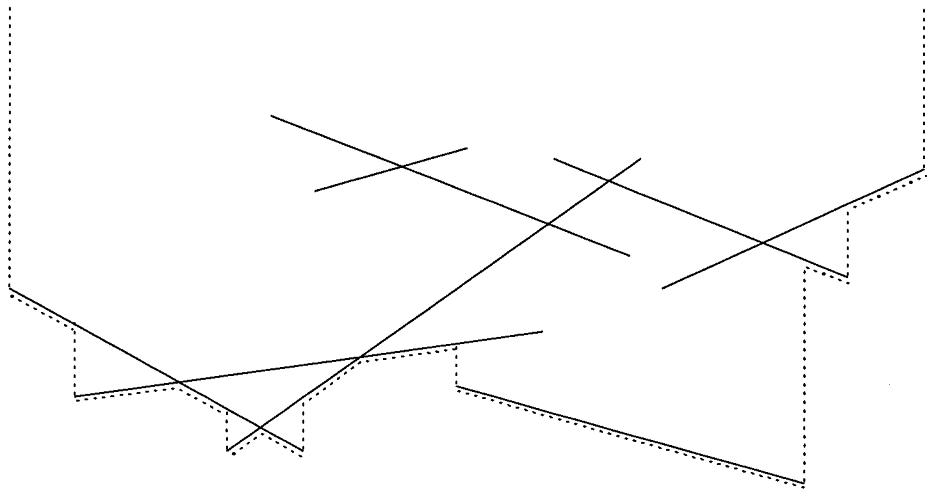

Figure 2.4: The lower envelope of a set of line segments. (Shown in dotted lines.)

computes lower envelopes in $O(n \log n)$ time. We now describe how his algorithm can be made output-sensitive with our technique.

First, observe that we can trace the h edges in $\mathcal{L}(S)$ from left to right by performing h *ray shooting operations*, where a ray shooting operation is: given a ray ρ emanating from a point on or beneath $\mathcal{L}(S)$, find the first trapezoid $\hat{s}$ $(s \in S)$ that ρ crosses. As such an operation can be done in $O(n)$ time, this gives us a naïve $O(nh)$ method, like Jarvis's march. To improve the running time, partition S into $\lceil n/m \rceil$ groups each of at most m segments and compute the lower envelope of each group by Hershberger's algorithm; this takes $O(n \log m)$ time in total. Using known data structures such as [CE+91, GH+87, HeS93], we can perform ray shooting under each of these $\lceil n/m \rceil$ envelopes in $O(\log m)$ time after $O(m\alpha(m))$ preprocessing (the ray shooting methods can be simplified in our case since envelopes are monotone). This implies that the h ray shooting operations on $\mathcal{L}(S)$ can be done in $O(h(\frac{n}{m} \log m))$ time. Choosing an appropriate group size m and guessing the output size h give us an optimal output-sensitive

$O(n \log h)$ algorithm for computing the lower envelope.

Theorem 2.3.1 *The h-edge lower envelope of a set of n line segments in E^2 can be constructed in $O(n \log h)$ time.*

Other applications of the same approach, including higher-dimensional ones, can be found in Chapter 4. In many cases, our grouping technique, combined with appropriate data structures, reduces $O(n \log n)$ terms in the running time to $O(n \log h)$, where h represents output size.

Chapter 3

Four-Dimensional Convex Hulls

In the previous chapter, we have presented some optimal output-sensitive algorithms for constructing convex hulls in two and three dimensions. Although there is not a drastic difference between the $O(n \log h)$ performance of these algorithms and the performance of the $O(n \log n)$ algorithms, in higher-dimensional space output-sensitivity can make a real difference.

This chapter considers the four-dimensional space E^4, where we give a near-optimal output-sensitive convex hull algorithm with an $O((n + f) \log^2 f)$ running time. The algorithm is therefore quite efficient for the whole range of output sizes f from $\Theta(1)$ to $\Theta(n^2)$. For example, when $f = \Theta(n)$, the algorithm runs in $O(n \log^2 n)$ time, which is a significant improvement over the $O(n^2)$ running time of a worst-case optimal algorithm [Sei81]. (The previous output-sensitive method by Seidel [Sei86], combined with Matoušek's improvement [Mat93], achieves $O(n^{4/3} \log^{O(1)} n)$ time in this case.)

The basic strategy behind our algorithm is divide-and-conquer. In order to obtain an output-sensitive method, the subproblems we solve cannot have asymptotically more faces than the original polytope. Therefore, we make each subproblem compute some restricted portion of the original polytope. For each of the subproblems defined, we show that sufficiently many input points can be removed without changing the subproblem. As in quicksort-like recursions, the merge step is trivial once we have devised a partitioning scheme for dividing a problem into subproblems. We have already seen this type of recursion before, namely, in the planar algorithm of Section 2.1; in fact, our 4-d algorithm is an extension of this 2-d algorithm. Further extension of the method to higher dimensions will be given in the next chapter in Section 4.6.

Our result on 4-d convex hulls has an important application, namely, to the computation of Voronoi diagrams in 3-space. We will examine this application in Section 3.3.

3.1 Preliminaries on the Divide-and-Conquer Construction of Convex Hulls

Before we give our four-dimensional convex hull algorithm, we first set up notation, introduce key concepts concerning the divide-and-conquer computation of convex hulls, and then describe a tool that we need.

3.1.1 The upper hull

For the most part, our 4-d algorithm is described in an arbitrary constant dimension d. In this setting, one can then tell when the four-dimensionality of the problem is used so that extensions to higher dimensions can be described more easily later. Figures are drawn for the case $d = 3$ only.

Given a set $P \subseteq E^d$ of n points in general position, our goal is compute the facial structure of the convex hull $\text{conv}(P)$. Following the approach of Section 2.1, we focus our attention only on the *upper hull*, consisting of faces of the convex hull with an upward normal vector (see Figure 3.5(a)). The upper hull of P can be thought of as the bounded faces of $\text{conv}(P \cup \{(0, \ldots, 0, -\infty)\})$. Once we have a method for computing the upper hull of P, we can also compute the lower hull of P in a similar manner by reflection and join the two hulls to form the convex hull of P.

Notation. Let $F(P)$, $R(P)$, and $V(P)$ be the set of all facets, ridges, and vertices (respectively) of the upper hull of P.

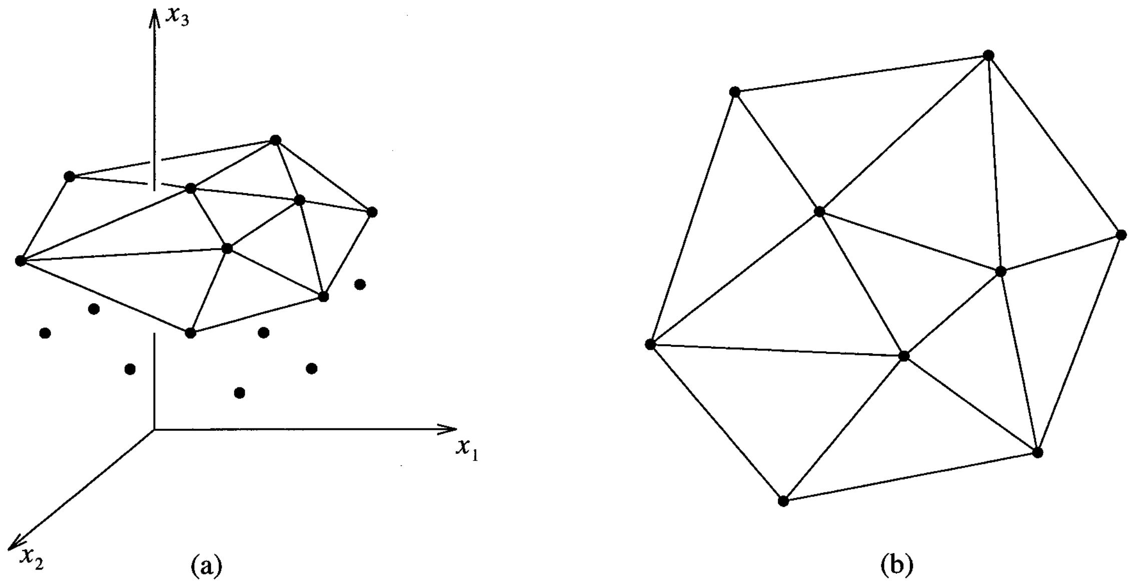

Figure 3.5: (a) The upper hull of a point set in E^3 and (b) the vertical projection of its facets.

To simplify representational issues, we require our algorithm to output only the set $F(P)$ of all facets of the upper hull of P. From this set, we can then generate all faces and build the complete lattice structure of the faces (the Hasse diagram) using a dictionary in $O(|F(P)| \log |F(P)|)$ time; this additional cost will be absorbed in the cost of the algorithm. Our algorithm for computing $F(P)$ is based on divide-and-conquer: to compute all the facets in $F(P)$, we partition $F(P)$ into suitable subsets and recursively compute these subsets of facets.

3.1.2 Facets and their duals

The following provides a simple characterization of the set of facets $F(P)$. First by the non-degeneracy assumption, $F(P)$ consists only of $(d-1)$-dimensional simplices with vertices all from P. Let f be such a simplex and let $h(f)$ denote the unique hyperplane containing f. Then $f \in F(P)$ iff all points of P lie on or below $h(f)$.

We now introduce some useful notation used throughout the chapter.

Notation. Let $\downarrow$ denote the *vertical projection* operator: $p\downarrow = (x_1, \ldots, x_{d-1})$ if $p = (x_1, \ldots, x_d)$, and $P\downarrow = \{p\downarrow : p \in P\}$ for any $P \subseteq E^d$. Given $P \subseteq E^d$ and $S \subseteq E^{d-1}$, let $P_{|S} = \{p \in P : p\downarrow \in S\}$ be the *restriction of* P *to* S. Let $\operatorname{int} S$ denote the interior of S and ∂S denote the boundary of S.

Observe that the vertical projection of the facets in $F(P)$ forms a collection of $(d-1)$-dimensional simplices in E^{d-1} that have disjoint interiors, that is, $\operatorname{int}(f\downarrow) \cap \operatorname{int}(f'\downarrow) = \emptyset$ for any two distinct facets $f, f' \in F(P)$. In fact, the vertical projection of all faces of the upper hull forms a *simplicial complex.* For example, if $d = 3$, then $\{f\downarrow : f \in F(P)\}$ forms a triangulation in the plane, as shown in Figure 3.5(b). Thus, one possible divide-and-conquer approach is to use these vertical projections to partition $F(P)$.

An equally natural approach is to use the vertical projections of the facets' *duals* to partition $F(P)$. For each $f \in F(P)$, we can define a point $f^D \in E^d$ via the standard duality transformation of [Ede87]: if the hyperplane $h(f)$ is given by $\{(x_1, \ldots, x_d) : x_d = 2\,\xi_1 x_1 + \ldots + 2\,\xi_{d-1} x_{d-1} - \xi_d\}$, then we let $f^D = (\xi_1, \ldots, \xi_d)$. The projection of the dual $f^D\downarrow = (\xi_1, \ldots, \xi_{d-1})$ is geometrically just the gradient of $h(f)$ (ignoring the scalar multiple 2); for example, if $d = 2$, then this is just the slope.

To allow us to speak about the two divide-and-conquer approaches more succinctly, we make the following definitions:

Definition. Given sets $S, \Delta \subseteq E^{d-1}$, let $F_S(P) = \{f \in F(P) : f\downarrow \subseteq S\}$ be the *primal restriction of* $F(P)$ *to* S and $F^{\Delta}(P) = \{f \in F(P) : f^D\downarrow \in \Delta\}$ be the *dual restriction of* $F(P)$ *to* Δ.

Before we describe our divide-and-conquer algorithm, we first need to introduce a general tool for geometric divide-and-conquer known as the $(1/r)$-*cutting.*

3.1.3 Cuttings for divide and conquer

Let H be a set of n hyperplanes. A *cutting* in E^d is a covering of E^d with closed (possibly unbounded) simplices with disjoint interiors; the *size* of the cutting is the number of simplices. A cutting Ξ is a $(1/r)$-cutting if any simplex of Ξ intersects at most n/r hyperplanes of H. The $(1/r)$-cutting and its relatives have recently become a popular tool in the design of divide-and-conquer algorithms for many geometric problems; see [AGR94, Cha93b] for examples in the context of convex hulls.

In [CF90], Chazelle and Friedman showed that a $(1/r)$-cutting of size $O(r^d)$ exists for any finite set of hyperplanes in E^d. Since then, many researchers, notably Matoušek [Mat91b, Mat91c] and Chazelle [Cha93a], have looked into the problem of how such a cutting can be constructed deterministically. The following theorem is one of the results that have been established, using derandomization techniques. We remark that improvements to this theorem were known, but since we need it only for the special case when r is constant, it is more than sufficient for our purposes.

Theorem 3.1.1 ([Mat91a, Theorem 6.1]) *Given n hyperplanes in a fixed dimension d, a $(1/r)$-cutting of size $O(r^d)$ can be computed in $O(nr^d)$ time.*

3.2 A Prune-and-Divide Convex Hull Algorithm in Four Dimensions

We are now ready to provide the details of our four-dimensional convex hull algorithm, which is an extension of the planar algorithm `DivideHull2d()` from Section 2.1. First, here is a high-level description how the extension works.

Recall that in line 4 of algorithm `DivideHull2d()`, the median of a set of $n/2$ numbers is computed. Since the median can be thought of as a one-dimensional $(1/2)$-cutting from Section 3.1.3, we extend this step to d dimensions by computing the $(1/2)$-cutting of a set of $n/2$ hyperplanes in E^{d-1}. In line 5, a vertex p_m, used for dividing the hull, is computed by taking the maximum of a set of numbers formed by projecting the input points along a direction of slope m. Of course, the maximum of a set of numbers can be interpreted as the upper hull of a one-dimensional point set. In d dimensions, p_m then becomes a collection of ridges computed by projecting the input points along certain directions and taking the upper hulls of the resulting $(d-1)$-dimensional point sets. For $d = 4$, these upper hulls are 3-dimensional and are therefore of linear size.

In the 2-d algorithm, p_m divides the upper hull of P into two parts: the portion of the hull to the left of p_m and the portion to the right of p_m. Observe that the left hull is also the portion with slope less than m, and similarly the right hull is the portion with slope greater than m. We have thus used p_m to partition the upper hull in two ways: (i) by x-coordinate and (ii) by slope. The restriction of the upper hull with x-coordinate inside a given interval is just primal restriction, in the terminology of Section 3.1.2, and the restriction of the upper hull with slope within a given interval is just dual restriction. In our extension to 4-d, we adopt the same strategy of using both primal and dual restrictions to partition the upper hull.

In the planar case, dividing the point set by x-coordinate ensures that the two subproblems do not share any input points except for the vertex p_m, and pruning by slope ensures that each of the two subproblems has at most $3/4$ of the input points. In the same manner for E^4, dividing by primal restrictions controls the sum of the sizes of the subproblems and pruning by dual restrictions guarantees that no subproblem receives more than a fixed fraction of the input. Subproblems can now share more than one input point, but we argue that the number of points shared is proportional to the size of the output. The analysis then follows from an application of the general cost lemma (Lemma 2.1.1) from Section 2.1.2: primal dividing bounds the per-level cost of the recursion tree and dual pruning ensures the α-fading property.

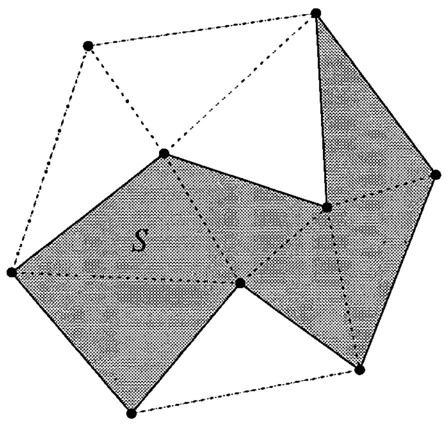

Figure 3.6: A simple region S of the point set in Figure 3.5.

3.2.1 Primal dividing

Our convex hull algorithm computes the primal restrictions of $F(P)$ to certain regions in E^{d-1} recursively. The regions are not arbitrary but are of a special form that we call *simple regions.*

Definition. A set $S \subseteq E^{d-1}$ is a *simple region* of P if it is the vertical projection of a union of facets in $F(P)$.

Figure 3.6 shows an example of a simple region. A simple region S may be disconnected. There may even exist a non-empty open $(d-1)$-ball centered on ∂S, of an arbitrarily small radius, whose intersection with $\operatorname{int} S$ is not homeomorphic to any $(d-1)$-ball. This intersection cannot be empty however, as S is a union of full-dimensional simplices; in particular, this rules out "spikes," e.g., $(d-2)$-simplices that are attached to the boundary.

The following lemma lists some useful properties concerning simple regions and primal restrictions. Part (a) is an identity that follows from definition and is important for proving other parts of the lemma. Parts (b) and (c) discuss when points can be removed without changing the primal restriction. Parts (d) and (e) provide bounds on the number of vertices and ridges restricted to a simple region. Finally, (f) and (g) describe properties of the boundary of a simple region.

Lemma 3.2.1 *Let S be a simple region of P. The following statements are true:*

(a) $\bigcup_{f \in F_S(P)} f\!\downarrow = S$. *(The projection of the facets of the primal restriction to S covers the region S.)*

(b) If $Q \subseteq P$ contains all vertices of the facets in $F_S(P)$, then S is a simple region of Q and $F_S(P) = F_S(Q)$. (Points that do not contribute to facets in $F_S(P)$ can be removed from P without affecting $F_S(P)$.)

(c) S is a simple region of the restricted point set $P_{|S}$ and the restricted facets $F_S(P_{|S}) = F_S(P)$.

(d) $|V(P_{|S})| \leq d\,|F_S(P)|$. *(The number of facets in the primal restriction gives a bound for the number of vertices.)*

(e) $|\{r \in R(P) : r\!\downarrow \subseteq S\}| \leq d\,|F_S(P)|$. *(The number of facets in the primal restriction gives a bound for the number of ridges.)*

(f) ∂S is the vertical projection of a union of ridges in $R(P)$. Thus, we can represent ∂S as a set of at most $d\,|F_S(P)|$ ridges.

(g) $P_{|\partial S} = \{v : v \text{ is a vertex of some ridge } r \text{ in } \partial S\}$, *and* $|P_{|\partial S}| \leq d\,|F_S(P)|$. *(The number of vertices in the boundary ∂S is bounded.)*

Proof: Recall that $\{\text{int}\,(f\!\downarrow) : f \in F(P)\}$ are disjoint. Then (a) is immediate from the definition of the primal restriction $F_S(P)$.

To prove (b), first note that $F_S(P) \subseteq F_S(Q)$ follows directly from the hypothesis. This implies that S is a simple region of Q. Now, (a) says that $\bigcup_{f \in F_S(P)} f{\downarrow} = S = \bigcup_{f \in F_S(Q)} f{\downarrow}$. We therefore must have equality: $F_S(P) = F_S(Q)$.

Statement (c) is a direct consequence of (b).

To prove (d), let p be a point in $V(P_{|S})$. Since the projection $p{\downarrow} \in S$, by (a) we have $p{\downarrow} \in f{\downarrow}$ for some facet $f \in F_S(P)$. Since $p \in V(P_{|S})$, p must be a vertex of f. Then (d) follows as each facet has d vertices incident on it (by the non-degeneracy assumption).

To prove (e), observe that each ridge r with $r{\downarrow} \subseteq S$ is incident on some facet in $F_S(P)$, by (a). Then (e) follows as each facet has d ridges incident on it.

The first part of (f) is immediate from the definition of a simple region. The cardinality bound is just a consequence of (e).

The first part of (g) follows from (f) and the non-degeneracy assumption. In particular, this implies that $P_{|\partial S} \subseteq V(P)$ and consequently, $P_{|\partial S} \subseteq V(P_{|S})$. So the second part follows from (d). $\square$

To compute the primal restriction $F_S(P)$ for a simple region S of P, our divide-and-conquer algorithm first subdivides S into smaller simple regions $\{S_i\}$ with disjoint interiors and then recursively computes $F_{S_i}(P)$ for each of the S_i's. In computing $F_{S_i}(P)$ we may consider only those input points that belong to $P_{|S_i} = P_{|\operatorname{int} S_i} \cup P_{|\partial S_i}$ by Lemma 3.2.1(c). Since $\operatorname{int} S_i$ are disjoint, the points shared between subproblems are points restricted to the boundary of the S_i's and we can bound the size of these boundaries in terms of the number of output facets by Lemma 3.2.1(f,g).

Note that when $d = 2$, the boundary of a connected simple region consists of just two points. In higher dimensions, the boundary becomes more complex and its manipulation demands more care.

3.2.2 Dual pruning

To find a good strategy for subdividing a simple region, we switch to dual space. We show how to partition E^{d-1} into a constant number of simplices such that in computing the dual restriction of $F(P)$ to each of the simplices, a fraction of the points of P can be pruned.

Lemma 3.2.2 *In $O(|P|)$ time, one can find closed simplices $\Delta_1, \ldots, \Delta_k \subseteq E^{d-1}$ and $P_1, \ldots, P_k \subseteq P$ such that (i) $\bigcup_{i=1}^{k} \Delta_i = E^{d-1}$ and $\{\operatorname{int} \Delta_i\}$ are disjoint, (ii) $F^{\Delta_i}(P) = F^{\Delta_i}(P_i)$, and (iii) $|P_i| \leq \alpha |P|$, for all $i = 1, \ldots, k$. Here, k and $0 < \alpha < 1$ are both constants depending only on d (assuming that $|P|$ exceeds a certain constant).*

Proof: A proof of this lemma in the dual setting can be found in Edelsbrunner's exposition [Ede87] of Megiddo's linear programming algorithm [Meg84]. The algorithm is based on the prune-and-search paradigm, and this lemma represents its "prune step." In Megiddo's original approach, the constant k is quite large and α is very close to 1. We observe an alternative solution using results on cuttings.

Let $|P| = n$. First, form the set H of dual hyperplanes by mapping each point $p = (p_1, \ldots, p_d)$ of P to the hyperplane $\{(\xi_1, \ldots, \xi_d) : \xi_d = 2p_1\xi_1 + \ldots + 2p_{d-1}\xi_{d-1} - p_d\}$. Then each facet f of the upper hull of P corresponds to a vertex of the lower envelope of H. (The *lower envelope* of H consists of faces of the polytope $\mathcal{P} = \{\xi \in E^d : \xi \text{ is below every hyperplane of } H\}$.) Furthermore, if $\Delta \subseteq E^{d-1}$, then a facet f in the dual restriction $F^{\Delta}(P)$ corresponds to a vertex of the lower envelope that has vertical projection in Δ.

Arbitrarily pair the n hyperplanes in H, compute the intersection of each pair, and vertically project these intersections. This gives us $n/2$ hyperplanes in E^{d-1}. Compute a constant-sized $(1/2)$-cutting $\{\Delta_i\}$ of these $(d-1)$-dimensional hyperplanes by Theorem 3.1.1. Consider a simplex Δ_i from the cutting. Half of the $n/2$ pairs have an intersection whose vertical projection lies completely outside Δ_i. For such a pair, one of the two hyperplanes cannot participate in the restriction of the lower envelope of H to Δ_i and is thus redundant. Therefore, in computing the dual restriction $F^{\Delta_i}(P)$, $n/4$ of the points can be pruned. This proves the lemma with $\alpha = 3/4$. □

3.2.3 Converting from dual to primal

In this subsection we show how to convert the partitioning $\{\Delta_i\}$ of the dual space obtained from Lemma 3.2.2 into a partitioning in the primal space. Specifically, we show that for any simplex $\Delta \subseteq E^{d-1}$, we can define a region $S_\Delta = S_\Delta(P) \subseteq E^{d-1}$ such that the primal restriction of $F(P)$ to S_Δ is the same as the dual restriction of $F(P)$ to Δ (i.e., $F_{S_\Delta}(P) = F^\Delta(P)$).

We start with the case when Δ is just a halfspace. By a transformation of coordinates, we can make Δ the halfspace $\{(\xi_1, \ldots, \xi_{d-1}) : \xi_1 \leq 0\}$. Define a projection $\pi_\Delta : E^d \to E^{d-1}$ that sends a point $(x_1, \ldots, x_d)$ to $(x_2, \ldots, x_d)$. Consider the upper hull of the $(d-1)$-dimensional point set $\pi_\Delta(P)$: its facets are projection of ridges in the upper hull of P, that is, $F(\pi_\Delta(P)) \subseteq \{\pi_\Delta(r) : r \in R(P)\}$. Let "boundary" B_Δ be the union of all $r\downarrow$ with $\pi_\Delta(r) \in F(\pi_\Delta(P))$. Then B_Δ is monotone in the first coordinate: given any $(\xi_1, \ldots, \xi_{d-1}) \in E^{d-1}$, there is at most one point $(\xi'_1, \ldots, \xi'_{d-1}) \in B_\Delta$ with $\xi_2 = \xi'_2, \ldots, \xi_{d-1} = \xi'_{d-1}$. We define S_Δ to be the region right of B_Δ, that is, $S_\Delta = \{(\xi_1, \ldots, \xi_{d-1}) : (\xi'_1, \xi_2, \ldots, \xi_{d-1}) \in B_\Delta \text{ for some } \xi'_1 \leq \xi_1\}$, as in Figure 3.7(a).

The following lemma can now be established for a halfspace Δ by verifying definitions.

Lemma 3.2.3 $F_{S_\Delta}(P) = F^\Delta(P)$.

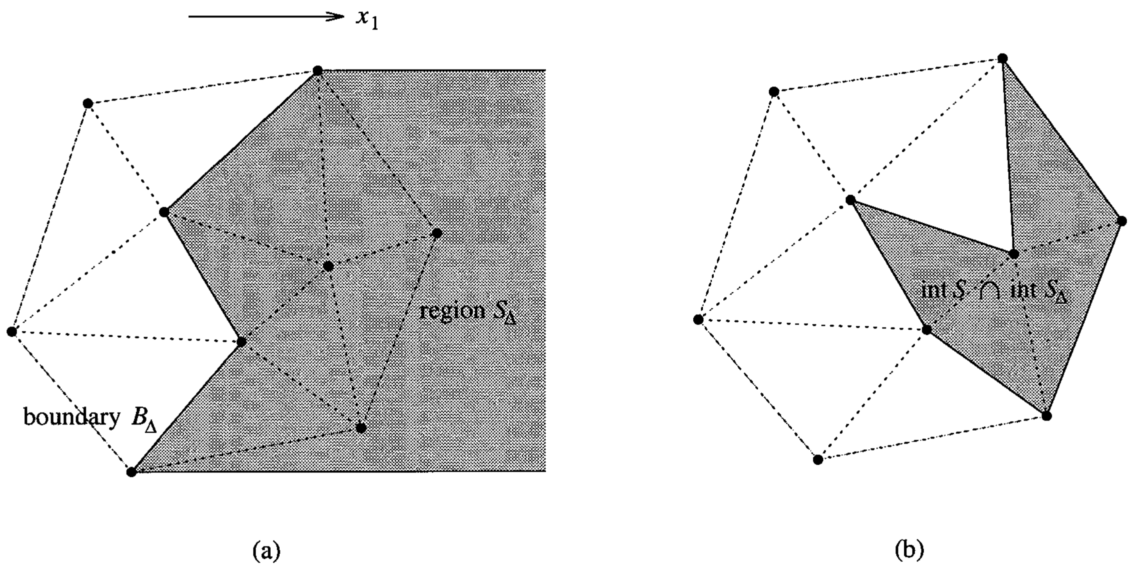

Figure 3.7: (a) The region S_Δ and (b) the intersection of its interior with the interior of the simple region S from Figure 3.6.

Remark: In the dual setting, as described in the proof of Lemma 3.2.2, the $(d-1)$-dimensional upper hull of the projected point set $\pi_\Delta(P)$ corresponds to the $(d-1)$-dimensional intersection of the lower envelope of H with $\partial\Delta$. Thus, ridges appearing in B_Δ correspond to edges of the lower envelope of H that intersect $\partial\Delta$.

We now extend the definition of S_Δ to the case when Δ is a simplex rather than a halfspace: Since Δ is a simplex, write Δ as an intersection of a constant number of halfspaces $\{\delta_j\}$. Then define S_Δ to be a region with interior $\bigcap_j \operatorname{int} S_{\delta_j}$. It is not difficult to see that Lemma 3.2.3 holds for simplices Δ as well.

3.2.4 Specializing for $d = 4$

We now show that the region S_Δ as defined in the previous subsection satisfies some nice computational properties if $d = 4$. We first consider the case in which Δ is a halfspace.

For $d = 4$, the projected point set $\pi_\Delta(P)$ is 3-dimensional. We can compute the facets of the upper hull $F(\pi_\Delta(P))$, and thus, the boundary B_Δ, in $O(|P| \log |V(P)|)$ time by Theorem 2.2.2. This permits computations involving the region S_Δ to be done efficiently, such as deciding if a point lies in the interior of S_Δ.

Lemma 3.2.4 *Suppose that $d = 4$. Then the restricted point set $P_{|\operatorname{int} S_\Delta}$ can be computed in $O(|P| \log |V(P)|)$ time using $O(|P|)$ space.*

Proof: Compute the facets of the 3-dimensional upper hull $F(\pi_\Delta(P))$ and store $\{\pi_\Delta(r){\downarrow} : \pi_\Delta(r) \in F(\pi_\Delta(P))\}$, which is a set of $O(|V(P)|)$ triangles in E^2 with disjoint interiors, in a planar point location structure [EGS86, Kir83, Pre90, ST86]; this takes $O(|P| \log |V(P)|)$ time. For each $p \in P$, we can then test whether $p{\downarrow} \in \operatorname{int} S_\Delta$ in logarithmic time by finding a facet $\pi_\Delta(r)$ of $F(\pi_\Delta(P))$ with $\pi_\Delta(p){\downarrow} \in \pi_\Delta(r){\downarrow}$ and then determining which side of $r{\downarrow}$ the point $p{\downarrow}$ lies on. $\square$

Another operation on the region S_Δ that we need is that of intersecting S_Δ with a simple region (see Figure 3.7(b)). To ensure that the resulting region is simple, we intersect their interiors only. We represent a simple region S by its boundary, which is a set of $O(|F_S(P)|)$ ridges by Lemma 3.2.1(f). We assume that each ridge r of ∂S is given an orientation to indicate which side of $r{\downarrow}$ the region S lies on.

Lemma 3.2.5 *Suppose that $d = 4$. Given the boundary ∂S for a simple region S of P, one can construct the boundary $\partial S'$ for a new simple region S' of P with $\operatorname{int} S' = \operatorname{int} S \cap \operatorname{int} S_\Delta$ in $O((|P| + |F_S(P)|) \log |V(P)|)$ time using $O(|P| + |F_S(P)|)$ space.*

Proof: Call a subset of S a *subregion* if it is the closure of a connected component of $E^{d-1} - (\partial S \cup B_\Delta)$. A subregion is a simple region, so we can define the new simple region S' to be the union of all subregions contained in S_Δ. The boundary $\partial S'$ is made up of

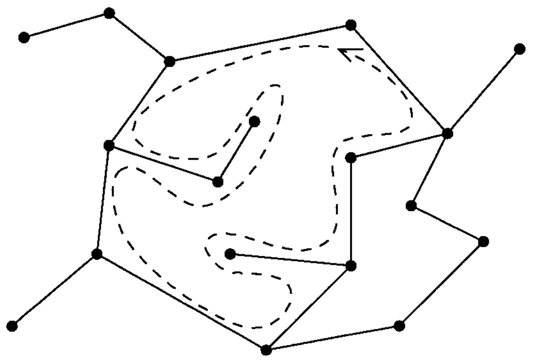

Figure 3.8: Tracing a boundary component ($d = 3$).

boundary components, which are connected components of the boundary of subregions. To decide whether a given boundary component B contributes to $\partial S'$, take a point q near B but inside the region bounded by B (this requires an examination of the orientation of a ridge in B), and then test if $q \in \operatorname{int} S_\Delta$ using the point location method from the previous lemma. Therefore, to compute $\partial S'$, it suffices to produce all the boundary components. This is done using depth-first search as follows.

We first record the ridges of the boundaries ∂S and B_Δ in a dictionary; as the $(d-1)$-dimensional upper hull $F(\pi_\Delta(P))$ and the boundary B_Δ can be computed in $O(|P| \log |V(P)|)$ time for $d = 4$, this takes $O((|P| + |F_S(P)|) \log |V(P)|)$ time. We make two copies of a ridge to represent the two "sides" of a ridge and assign different orientations to them. We then generate all the $(d-3)$-subfaces of these ridges, and for each such $(d-3)$-face σ, we create the list of ridges incident to σ in sorted order and store σ in a dictionary for $(d-3)$-faces. The ordering of these ridges is based on the angles made by their vertical projections with a fixed hyperplane through $\sigma\!\downarrow$ in E^{d-1}.

Then, given an (oriented) ridge in a boundary component B, we can identify its $d-1$ adjacent ridges in B in constant time by following pointers. (Here, two oriented ridges

r_1 and r_2 are *adjacent* in B if there is a common $(d-3)$-subface σ incident on both ridges and there is no other ridge r' in B that σ is incident on, such that $r'\!\downarrow$ lies within the angle range defined by $r_1\!\downarrow$ and $r_2\!\downarrow$ around $\sigma\!\downarrow$.) Using a depth-first search to visit the adjacent ridges recursively, we can then trace all ridges that belong to the same boundary component B, as indicated in Figure 3.8. All boundary components can then be generated by ensuring that all ridges in ∂S are visited. The time required by the depth-first search is proportional to the number of ridges in ∂S and B_Δ and is therefore only $O(|F_S(P)| + |V(P)|)$.

Note: if both sides of a ridge appear in the boundary $\partial S'$, then the ridge can be removed. □

It remains to extend the above lemmas to the case when Δ is a simplex rather than a halfspace. Recall that we have defined S_Δ such that $\operatorname{int} S_\Delta = \bigcap_j \operatorname{int} S_{\delta_j}$, if Δ is written as an intersection of a constant number of halfspaces $\{\delta_j\}$. By applying Lemmas 3.2.4 and 3.2.5 to each halfspace δ_j individually, we see that the lemmas are also true for the simplex Δ.

3.2.5 The prune-and-divide algorithm in four dimensions

We now have all the pieces needed for an output-sensitive convex hull algorithm in E^4. Let `Dual-Partition`(P) represent a dual partitioning $\{(P_i, \Delta_i)\}_{i=1}^{k}$ obtained from Lemma 3.2.2. Let `Restrict-Interior`(P, Δ) represent the restricted point set $P_{|\operatorname{int} S_\Delta}$ as computed by Lemma 3.2.4 and let `Restrict-Boundary`(P, B, Δ) be the boundary of the simple region S' returned in Lemma 3.2.5 for $B = \partial S$. The following provides an outline of our recursive algorithm.

```
Algorithm DivideHull4d($P^\bullet, B$)
[ Given $P^\bullet = P_{|\operatorname{int} S}$ and $B = \partial S$ for a simple region $S$ of a point set $P \subseteq E^4$ where
  $B \neq \emptyset$ is represented as a set of (oriented) ridges, return the set of facets $F_S(P)$. ]
 1.  $P \leftarrow P^\bullet \cup \{v : v$ is a vertex of some ridge $r$ in $B\}$
 2.  if $|P| \leq n_0$ for a constant $n_0$ then return $F_S(P)$ in constant time
 3.  $\{(P_i, \Delta_i)\}_{i=1}^k \leftarrow$ Dual-Partition($P$) by computing a (1/2)-cutting (Lemma 3.2.2)
 4.  for $i = 1, \ldots, k$ do
 5.      $P_i^\bullet \leftarrow P_i \cap P^\bullet \cap$ Restrict-Interior($P, \Delta_i$) by computing a 3-d upper hull
                 and performing 2-d point location (Lemma 3.2.4)
 6.      $B_i \leftarrow$ Restrict-Boundary($P, B, \Delta_i$) by computing a 3-d upper hull and
                 performing depth-first search on the boundary ridges (Lemma 3.2.5)
 7.  return $\bigcup \{$DivideHull4d($P_i^\bullet, B_i$) $: B_i \neq \emptyset\}$
```

We first argue that the algorithm indeed computes the primal restriction $F_S(P)$. In the first line of the algorithm, we reset P to the point set $P_{|\operatorname{int} S} \cup P_{|\partial S} = P_{|S}$, according to Lemma 3.2.1(g); the justification is provided by Lemma 3.2.1(c): $F_S(P) = F_S(P_{|S})$. Line 2 provides the base case. Line 3 gives us a constant number of simplices $\{\Delta_i\}$ with disjoint interiors, covering E^{d-1}; for each Δ_i, we are also given a subset P_i of P, of cardinality at most $\alpha\,|P|$, such that $F^{\Delta_i}(P) = F^{\Delta_i}(P_i)$.

Let S_i denote the simple region with interior $\operatorname{int} S \cap \operatorname{int} S_{\Delta_i}$. Since $F_{S_{\Delta_i}}(P) = F^{\Delta_i}(P)$ by Lemma 3.2.3, we know that the S_i's have disjoint interiors and that their union is S. Furthermore, as $F_{S_i}(P) \subseteq F_{S_{\Delta_i}}(P) = F^{\Delta_i}(P) = F^{\Delta_i}(P_i)$, all facets in $F_{S_i}(P)$ have vertices from P_i, which implies that $F_{S_i}(P) = F_{S_i}(P_i)$ by Lemma 3.2.1(b).

In line 5 we set $P_i^\bullet = P_i \cap P^\bullet \cap P_{|\operatorname{int} S_{\Delta_i}} = P_{i\,|\operatorname{int} S_i}$ and in line 6 we let B_i be the boundary of S_i. Then line 7 returns $\bigcup_i F_{S_i}(P_i) = \bigcup_i F_{S_i}(P) = F_S(P)$, as claimed.

Having argued that DivideHull4d() correctly computes the primal restriction $F_S(P)$, we can use the algorithm to compute the set $F(P)$ of all facets of the upper hull. The initial simple region S_0 we use is just the convex hull of $P\!\downarrow$, which can be computed using the 3-dimensional algorithm from Section 2.2. Thus, by letting $P^\bullet = \{p \in P :$

$p{\downarrow}$ is not a vertex of $S_0\}$ and $B = \partial S_0$, a call to `DivideHull4d`$(P^\bullet, B)$ then returns $F(P)$, as desired.

3.2.6 Analysis of the prune-and-divide algorithm in four dimensions

We now analyze the running time of the algorithm. We do so by counting the cost of the recursion tree produced by the calls to `DivideHull4d`(), in a way similar to our analysis of `DivideHull2d`() in Section 2.1.2. Let n be the number of input points and f be the number of facets of the upper hull. Let P_ν and S_ν denote the input point set and the simple region associated with a node ν of the recursion tree. Let $n_\nu = |P_{\nu\,|\,\mathrm{int}\,S_\nu}|$ and $f_\nu = |F_{S_\nu}(P_\nu)|$.

By Lemmas 3.2.2, 3.2.4, and 3.2.5, the non-recursive part of the algorithm (lines 1–6) requires $O((|P_\nu| + |F_{S_\nu}(P_\nu)|)\log|V(P_\nu)|) = O((|P_\nu| + f_\nu)\log f_\nu)$ time at node ν, since $|V(P_\nu)| = |V(P_{\nu\,|S_\nu})| \leq d\,|F_{S_\nu}(P_\nu)|$ by Lemma 3.2.1(d). To get the total running time, we just have to sum this cost over all nodes in the recursion tree.

We first analyze the cost contributed by the $O(|P_\nu|\log f_\nu)$ term. By Lemma 3.2.2(iii), this cost is α-fading, so we can apply Lemma 2.1.1. To sum the costs on a given level of the tree, we write $|P_\nu| = |P_{\nu\,|\,\mathrm{int}\,S_\nu}| + |P_{\nu\,|\partial S_\nu}| \leq n_\nu + df_\nu$ by Lemma 3.2.1(g). Since the S_ν's have disjoint interiors over all nodes ν of one level, we have $\sum_\nu n_\nu < n$ and $\sum_\nu f_\nu = f$ for each level of the recursion tree. This gives us an $O((n+f)\log f)$ bound on the cost-per-level. The tree has $O(f)$ leaves, as each leaf discovers at least one facet (note that $F_{S_\nu}(P_\nu) = \emptyset$ only if $\partial S_\nu = \emptyset$ by definition of a simple region). Lemma 2.1.1(ii) says that the total contribution is $O((n+f)\log^2 f)$.

Next we analyze the cost contributed by the $O(f_\nu \log f_\nu)$ term. This cost may not be α-fading, so we cannot apply Lemma 2.1.1. But since $\sum_\nu f_\nu = f$ and the recursion tree has depth at most $\log_{1/\alpha} n$ by Lemma 3.2.2(iii), we can bound the sum of these costs by $O(f \log f \log n)$, which never dominates $O((n+f)\log^2 f)$.

We conclude that the total running time of the algorithm is $O((n+f)\log^2 f)$. Total space is $O(n+f)$ as long as we free up the space used to store the boundary B before we make the recursive calls in line 7.

Theorem 3.2.6 *Algorithm* `DivideHull4d()` *computes the f-face upper hull of an n-point set $P \subseteq E^4$ in $O((n+f)\log^2 f)$ time and $O(n+f)$ space.*

Remarks:

1. Algorithm `DivideHull4d()` can be considered as a primal-based divide-and-conquer algorithm since it recursively computes the primal restriction of $F(P)$ to a simple region. Alternatively, one may consider an algorithm that computes the dual restriction of $F(P)$ to a simplex recursively. This dual-based approach is perhaps less complex since simplices are easier to handle than boundaries of simple regions. However, the problem with this approach is that the dual analogue of Lemma 3.2.1(b) is not true in general: one can construct a point set $P \subseteq E^3$ and a triangle $\Delta \subseteq E^2$ with $F^\Delta(P) \neq F^\Delta(Q)$ for $Q = \{v : v \text{ is a vertex of some facet } f \in F^\Delta(P)\}$.

2. Although the cutting techniques used for dual pruning in our algorithm have been well-studied, our strategy for primal dividing appears new. This strategy provides a simple way to guarantee that the total problem size at any level of the recursion is $O(n+f)$; it would be difficult to obtain such a bound using the existing cutting techniques alone. Previously, primal dividing was used only in two and three dimensions, notably in the algorithms of Kirkpatrick and Seidel [KS86] and Edelsbrunner and Shi [ES91], and our algorithm can be regarded as an extension of these approaches. In fact, the three-dimensional version of our algorithm simplifies Edelsbrunner and Shi's algorithm in the same way as our two-dimensional algorithm `DivideHull2d()` simplifies Kirkpatrick and Seidel's (see the first remark after Theorem 2.1.2). The "contour"-based approach used in a recent parallel 3-d convex hull algorithm by Amato, Goodrich, and Ramos [AGR94]

can also be interpreted as a form of primal dividing. There, contours play a role similar to the lower-dimensional upper hulls of $\pi_\Delta(P)$ in Section 3.2.3 and are used to ensure that the total problem size at any level of the recursion remains $O(n)$; but since they describe their method in the dual setting, its geometry is less apparent in some places.

3. `DivideHull4d()` can return not only $F(P)$ but also a point location structure for the set $\{f\downarrow : f \in F(P)\}$ of tetrahedra in E^3. We simply maintain the recursion tree and store the planar point location structures from Lemma 3.2.4 at every node; this requires $O((n+f)\log f)$ space. Then we can find a facet of which the vertical projection contains our given query point by just following a path down the recursion tree. Since the tree has depth at most $\log_{1/\alpha} n$, the query time is $O(\log^2 n)$. For small output size f, we can further reduce the space and query time bound to $O(f \log f)$ and $O(\log^2 f)$ by first calling `DivideHull4d()` to identify the vertices $V(P)$ and then building the point location structure for $V(P)$ instead of P (as $F(V(P)) = F(P)$). We thus achieve the same performance as Goodrich and Tamassia's 3-d point location structure [GT91].

4. *Some practical issues.* With no additional work, `DivideHull4d()` can return the incidence structure between facets and ridges; this fact can be used to reduce the number of dictionary operations needed. Moreover, with an appropiate choice of coordinate system, it is not necessary to compute the upper and lower hulls separately; we choose to do so here merely because vertical projections are easier to visualize. In Section 3.2.3, a transformation of coordinates is used to define the projection π_Δ for an arbitrary halfspace Δ; this is used only to simplify presentation and such a transformation needs not be carried out explicitly. Finally, we should mention that degeneracies may occur in the projected point set $\pi_\Delta(P)$ even though the point set P is itself non-degenerate; in such a situation, we may wish to apply a perturbation to Δ.

5. In the next chapter, we describe one way that algorithm `DivideHull4d()` can be extended to higher dimensions. Since this extension requires higher-dimensional data

structures for ray shooting and linear programming queries, we postpone its discussion until we come to Section 4.6. See that section for the difficulties that arise when the dimension is beyond 4.

3.3 Application: Three-Dimensional Voronoi Diagrams

In this section we discuss one important application of our four-dimensional convex hull algorithm, namely the output-sensitive computation of Voronoi diagrams in three dimensions. Let P be a set of n point sites in E^d. The *Voronoi region of a site* $p \in P$ consists of all points $q \in E^d$ such that q is closer to p than to any other point in P (with respect to Euclidean distance). The *Voronoi diagram of* P is the collection of all Voronoi regions (see Figure 3.9). The Voronoi diagram is an extremely useful structure in computational geometry, since it is a powerful technique for dealing with problems related to proximity, such as answering *nearest neighbor queries.* Furthermore, the dual of the Voronoi diagram is the *Delaunay triangulation*, another fundamental geometric structure with many applications. See the survey by Aurenhammer [Aur91] or the book by Okabe, Boots, and Sugihara [OBS92] for more on these and other applications.

We first describe how an algorithm for constructing convex hulls automatically yields an algorithm for constructing Voronoi diagrams in one dimension lower by "lifting" the sites. The connection between convex hulls and Voronoi diagrams is well known: it was first noted by Brown [Bro80] and further developed by Edelsbrunner and Seidel [ES86]. We also describe how to get an output-sensitive algorithm for computing a portion of the Voronoi diagram clipped to a polytope.

For a given site $p = (p_1, \ldots, p_d) \in P$, we can use a "lifting map" [Ede87, ES86] to define a halfspace p^* in E^{d+1}: $p^* = \{(x_1, \ldots, x_{d+1}) : x_{d+1} \geq 2p_1x_1 + \ldots + 2p_dx_d - p \cdot p\}$. It is well known that the Voronoi regions are just the vertical projection of

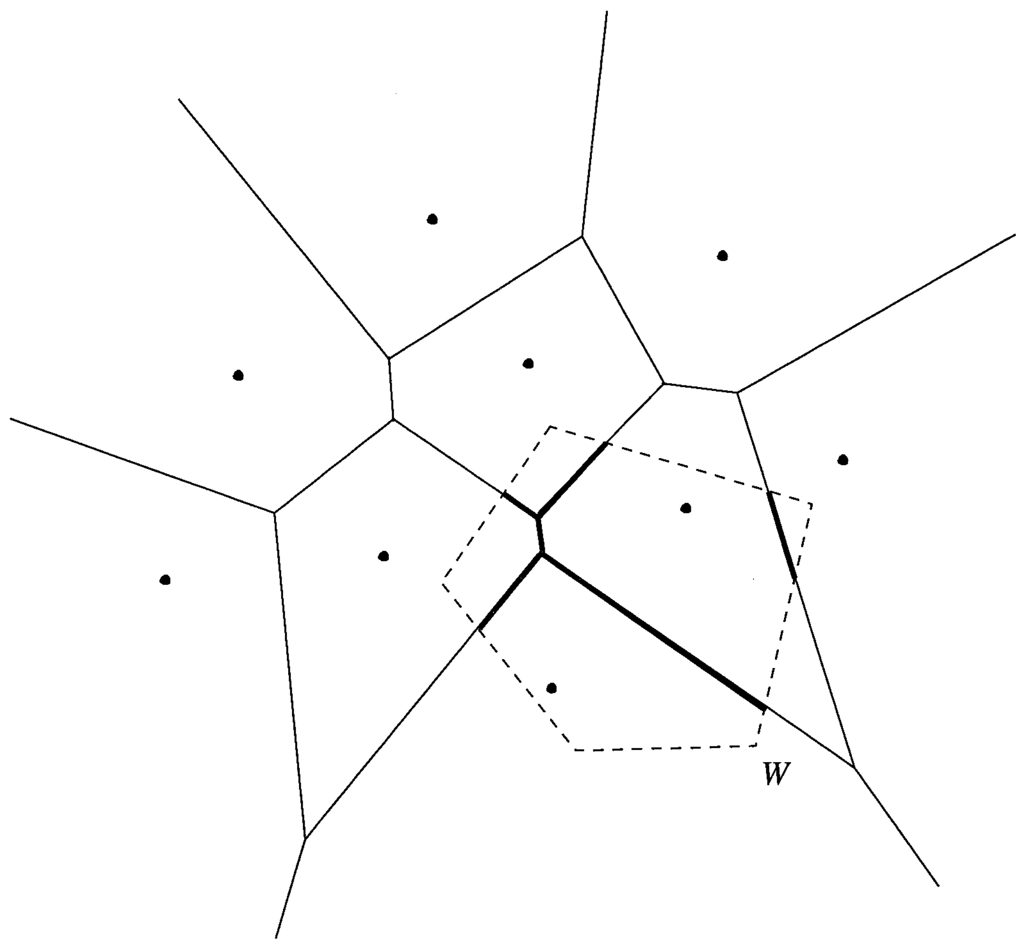

Figure 3.9: The Voronoi diagram of a planar point set. Bold lines indicate edges of the diagram clipped to the polygon W.

the facets of the polytope $\bigcap_{p\in P} p^*$. Thus, the computation of a Voronoi diagram in E^d is reduced to the computation of an intersection of halfspaces in E^{d+1}. Since the computing an intersection of halfspaces is equivalent to the computing convex hulls by duality (Section 1.1), Theorem 3.2.6 has the following consequence:

Theorem 3.3.1 *The Voronoi diagram of n point sites in E^3 can be computed in $O((n+f)\log^2 n)$ time and $O(n+f)$ space, where f is the size of the Voronoi diagram ($\Omega(n) = f = O(n^2)$).*

In certain applications, only the portion of a Voronoi diagram lying in a given area is needed, and the size of this portion may be much smaller than the size of the entire Voronoi diagram. What we want is then an output-sensitive algorithm to compute the *Voronoi diagram of P clipped to a region W*, defined simply as the collection of all non-empty intersections of the Voronoi regions with W (see Figure 3.9).

Suppose that W is a k-dimensional polytope $(\bigcap \Gamma) \cap F$, where Γ is a set of m halfspaces and F is a k-flat in E^d. Lift each halfspace $\gamma \in \Gamma$ to a vertical halfspace denoted by γ^* and lift the k-flat to a vertical $(k+1)$-flat denoted by F^*. Then the clipped Voronoi diagram is just the vertical projection of the facets of the polytope $\bigcap_{p\in P} p^* \cap \bigcap_{\gamma\in\Gamma} \gamma^* \cap F^*$. Thus, the clipped Voronoi diagram can be computed by constructing the intersection of the halfspaces $\{p^* \cap F^* : p \in P\} \cup \{\gamma^* \cap F^* : \gamma \in \Gamma\}$ inside the $(k+1)$-flat F^*. If $k = 3$, we can use Theorem 3.2.6 to compute the intersection of these $n+m$ halfspaces of dimension $k+1$.

Theorem 3.3.2 *Let $d \geq 4$ be a constant. The Voronoi diagram of n point sites in E^d clipped to a 3-dimensional polytope defined by m halfspaces can be computed in $O((n+m+f)\log^2 f)$ time and $O(n+m+f)$ space, where f is the size of the clipped Voronoi diagram ($\Omega(1) = f = O(n^2)$).*

Chapter 4

Higher-Dimensional Convex Hulls

In this chapter, we consider the convex hull problem in E^d where the dimension d is any fixed constant. We begin by generalizing the two- and three-dimensional convex hull algorithms in Section 2.2. Recall that these algorithms are based on the gift-wrapping method. Since one can interpret a wrapping step as performing a *ray shooting query* in dual space, we recast the grouping technique of Section 2.2 in a more general setting in terms of ray shooting queries (Section 4.1). With a little more work, the same technique can actually be applied to *linear programming queries* as well (Section 4.2). Data structures for answering ray shooting and linear programming queries are then our main tools in higher-dimensional space.

Using known data structures for ray shooting queries in polytopes by Agarwal, Matoušek and Schwarzkopf [AM93, MS93], we obtain an $O(n \log f + (nf)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$-time convex hull algorithm in E^d. Using known data structures for linear programming queries by Matoušek [Mat93], we also obtain an $O(n \log^{O(1)} h + (nh)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$-time algorithm for identifying the *extreme points* (i.e., the vertices of the convex hull) of a set of n points in E^d. (Recall that $f = O(n^{\lfloor d/2 \rfloor})$ is the number of hull faces and $h \leq n$ is the number of hull vertices/extreme points.) These output-sensitive results are described in Section 4.3.

Ray shooting and linear programming queries turn out to have numerous applications, and we examine how our techniques lead to improved results in some of these applications in Sections 4.4 and 4.5. Finally, using ray shooting and linear programming queries in our

4-d divide-and-conquer convex hull algorithm from Chapter 3, we obtain an extension of the algorithm that computes d-dimensional convex hulls in $O((n + (nf)^{1-1/\lceil d/2 \rceil} + fn^{1-2/\lceil d/2 \rceil}) \log^{O(1)} n)$ time for any even $d > 4$ in Section 4.6.

4.1 Ray Shooting Queries

We first investigate the problem of ray shooting in a convex polytope. Given a collection H of n (closed) halfspaces in E^d, where each halfspace contains a known point, say, the origin o, a *ray shooting query* is to determine the first bounding hyperplane h of $\bigcap H$ that is crossed by a query ray originating from $\bigcap H$ (a ray *crosses* a hyperplane h if it intersects h but is not contained in h).

In two dimensions, the ray shooting problem can be solved as follows: first compute the polygon $\bigcap H$ and store its vertices in an array in counterclockwise order; then a query can be done by a simple binary search. Observe that computing the intersection $\bigcap H$ is equivalent to computing a convex hull in the dual space, and thus takes $O(n \log n)$ time by Graham's scan for example [Gra72]; and the binary search takes $O(\log n)$ time. Hence, this method requires $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time.

The same preprocessing time, space, and query time can be obtained in three dimensions: in the preprocessing, compute the polytope $\bigcap H$ by the dual of Preparata and Hong's convex hull algorithm [PH77] and construct its Dobkin-Kirkpatrick hierarchical representation [DK83, DK90]; then a query can be answered in logarithmic time (see Section 2.2.2).

Our first observation is that a preprocessing time/query time tradeoff is possible using the grouping idea from Section 2.2. Using this observation, we can perform q queries in $O(n \log q)$ time rather than $O(n \log n)$ time for small q's.

Lemma 4.1.1 *There is a (static) data structure for ray shooting in a polytope defined by a set H of n halfspaces in E^2 or E^3 with $O(n \log m)$ preprocessing time, $O(n)$ space, and $O((n/m) \log m)$ query time, where m is a parameter between 1 and n.*

Proof: Partition H into $\lceil n/m \rceil$ subsets ("groups") $H_1, \ldots, H_{\lceil n/m \rceil}$, each of size at most m and build the above structures for each H_i. The total preprocessing time is $O(\frac{n}{m}(m \log m)) = O(n \log m)$, and the space complexity remains $O(n)$. Now, ray shooting is a *decomposable* problem [BS80], i.e., the answer to a query on $H' \cup H''$ can be computed from the answers to the queries on H' and H'' in constant time. Therefore, a query on H can be computed directly by querying on each H_i, taking $O((n/m) \log m)$ time. □

Corollary 4.1.2 *An (online) sequence of q ray shooting queries in a polytope defined by a set H of n halfspaces in E^2 or E^3 can be performed in $O(n \log q + q \log n)$ time and $O(n)$ space.*

Proof: By Lemma 4.1.1, the total time needed to answer q queries is $O(n \log m + q(\frac{n}{m} \log m))$, where $1 \leq m \leq n$. Choose $m = q$ when $q \leq n$ and choose $m = n$ when $q > n$. □

For d-dimensional polytopes with $d > 3$, Agarwal and Matoušek [AM93] were the first to obtain efficient ray shooting data structures. Subsequently, Matoušek and Schwarzkopf [MS93] proposed a simpler and slightly faster approach, with the results shown in Table 4.1.

Structure 1 in the table originates from a data structure by Matoušek [Mat92] for the problem of preprocessing an n-point set $P \subseteq E^d$ for halfspace range queries. (A *halfspace range query* on P is to report all points of P that lie in a given query halfspace.)

Structures	preprocessing time, space	update time (amortized)	ray shooting query time [MS93]	linear programming query time [Mat93]
1	$n \log n,\ n$	N/A	$n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n$	$n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n$
2	$m \log^{O(1)} n$	N/A	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{2d+1} n$
3	$n^{\lfloor d/2 \rfloor} \log^{O(1)} n$	N/A	$\log n$	$\log^{d+1} n$
1′	$n \log n,\ n$	$\log^2 n$	$n^{1-1/\lfloor d/2 \rfloor+\varepsilon}$	$n^{1-1/\lfloor d/2 \rfloor+\varepsilon}$
2′	$m^{1+\varepsilon}$	$m^{1+\varepsilon}/n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n$	$\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{2d+1} n$
3′	$n^{\lfloor d/2 \rfloor+\varepsilon}$	$n^{\lfloor d/2 \rfloor-1+\varepsilon}$	$\log n$	$\log^{d+1} n$

Table 4.1: Known data structures for ray shooting queries in polytopes and linear programming queries. For Structures 2 and 2′, m is a parameter between n and $n^{\lfloor d/2 \rfloor}$. The bounds are all deterministic, with the big-Oh notation omitted.

A collection $\{(P_1, \Delta_1), \ldots, (P_k, \Delta_k)\}$ is called a *simplicial partition* of size k if (i) $\bigcup_i P_i = P$, (ii) the P_i's are disjoint, and (iii) Δ_i is a simplex containing P_i for each i. The key behind Matoušek's structure is the following *Partition Theorem*: given $1 < r < n/2$, there exists a simplicial partition $\{(P_i, \Delta_i)\}$ of size $O(r)$, with $n/r \le |P_i| \le 2n/r$, such that any hyperplane with fewer than n/r points of P on one side crosses at most $O(r^{1-1/\lfloor d/2 \rfloor} + \log r)$ of the simplices Δ_i. A method for constructing such a simplicial partition is described in [Mat92]. This theorem suggests a data structure for storing the point set P called the *partition tree*: the simplices Δ_i are stored at the root of the tree, and a subtree is generated in a recursive fashion for each of the point sets P_i's.

Matoušek and Schwarzkopf [MS93] observed that the partition tree can be used to answer ray shooting queries on the polytope $\bigcap H$ if we dualize the halfspaces in H to points and augment each node of the partition tree with a $(1/r)$-*net* [HW87, Mat91c]. Choosing $r = n^\gamma$ for a suitable constant $\gamma > 0$, they then obtained a linear-space structure that can be built in $O(n \log n)$ time and can answer ray shooting queries in $O(n^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} n)$ time.

Structure 3 in Table 4.1 is obtained in a different manner. Here, the key concept is a shallow cutting. A *0-level shallow cutting* for H is a covering of $\bigcap H$ with simplices with disjoint interiors; the size of Ξ is the number of simplices. Furthermore, Ξ is a *0-level shallow* $(1/r)$*-cutting* if any simplex of Ξ intersects at most n/r of the bounding hyperplanes of H. In [Mat92], it is shown that a 0-level shallow $(1/r)$-cutting of size $O(r^{\lfloor d/2 \rfloor})$ exists.

Following an earlier randomized algorithm of Clarkson [Cla87] for the polytope membership problem, Matoušek and Schwarzkopf defined the following tree-like structure: a 0-level shallow $(1/r)$-cutting for H is stored at the root of the tree, and for each simplex Δ in the cutting, a subtree is generated recursively for the set of halfspaces that do not completely contain Δ. If one sets $r = n^{\gamma}$, such a structure can be built in $O(n^{\lfloor d/2 \rfloor} \log^{O(1)} n)$ time. It was shown by Matoušek and Schwarzkopf [MS93] that a ray shooting query can be answered in logarithmic time with this structure.

Finally, a combination of the linear-space approach used in Structure 1 and the large-space approach used in Structure 3 yields a continuous tradeoff between preprocessing and query time: with $O(m \log^{O(1)} n)$ preprocessing, one can answer a ray shooting query in $O((n/m^{1/\lfloor d/2 \rfloor}) \log n)$ time, for any $n \leq m \leq n^{\lfloor d/2 \rfloor}$. This is named Structure 2 in Table 4.1.

We observe here that the grouping scheme used in Lemma 4.1.1 can in fact be used to obtain further preprocessing time/query time tradeoffs for Structure 1.

Lemma 4.1.3 *There is a (static) data structure for ray shooting in a polytope defined by a set H of n halfspaces in E^d $(d > 3)$ with $O(n \log m)$ preprocessing time, $O(n)$ space, and $O((n/m^{1/\lfloor d/2 \rfloor}) \log^{O(1)} m)$ query time, where m is a parameter between 1 and n.*

Proof: By partitioning H into $\lceil n/m \rceil$ groups as in Lemma 4.1.1 and using Structure 1 to store each group, the preprocessing time becomes $O(\frac{n}{m}(m \log m)) = O(n \log m)$ and query time becomes $O(\frac{n}{m}(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)) = O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{O(1)} m)$. □

Corollary 4.1.4 *A sequence of q ray shooting queries in a polytope defined by a set H of n halfspaces in E^d $(d > 3)$ can be performed in $O(n \log q + (nq)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n + q \log n)$ time (using $O(n + (nq)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ space).*

Proof:

CASE I. $q \leq n^{1/\lfloor d/2 \rfloor} / \log^K n$, where K is a sufficiently large constant. Use Lemma 4.1.3's modification of Structure 1 with $m = (q \log^K q)^{\lfloor d/2 \rfloor}$ $(1 \leq m \leq n)$. Then the running time is

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

CASE II. $n^{1/\lfloor d/2 \rfloor} < q < n^{\lfloor d/2 \rfloor}$. Use Structure 2 with $m = (nq)^{1-1/(\lfloor d/2 \rfloor+1)}$ $(n \leq m \leq n^{\lfloor d/2 \rfloor})$. Then the running time is

$$O\left(m \log^{O(1)} n + q \frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n\right) = O((nq)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n).$$

CASE III. $q \geq n^{\lfloor d/2 \rfloor} \log^K n$. Use Structure 3. Then the running time is

$$O\left(n^{\lfloor d/2 \rfloor} \log^{O(1)} n + q \log n\right) = O(q \log n).$$

□

Remark: In some applications, the number of queries q may not be known in advance. In that case, the parameter m cannot be set directly. This problem can be avoided by breaking the q queries into k clusters of $q_1, \ldots, q_k$ queries, where $q_1, q_2, \ldots$ is a known sequence and $q_1 + \ldots + q_{k-1} < q \leq q_1 + \ldots + q_k$. For example, in Case I of the proof of

Corollary 4.1.4, if we choose the sequence $q_i = 2^{2^i}$ $(i = 1, 2, \ldots)$, then the total running time is $O(\sum_{i=1}^{k} n \log q_i) = O(\sum_{i=1}^{\lceil \log \log q \rceil} n 2^i) = O(n \log q)$, as before. (Logarithms are in base 2.) Similarly, in Case II, we see that the complexity remains unchanged by choosing the sequence $q_i = n^{1/\lfloor d/2 \rfloor} 2^i$ $(i = 1, 2, \ldots)$. This method of guessing the value of q using an increasing sequence is analogous to the method used in Section 2.2 for guessing the output size.

We now discuss dynamic ray shooting in polytopes, where halfspaces may be inserted or deleted. In two dimensions, a data structure by Overmars and van Leeuwen [OvL81] has $O(n \log n)$ preprocessing time, $O(n)$ space, $O(\log^2 n)$ update time, and $O(\log n)$ query time; the data structure uses a balanced binary search tree and concatenable queue operations to maintain the dual convex hull. It is straightforward to extend Lemma 4.1.1 to get a method with $O(n \log m)$ preprocessing time, $O(n)$ space, $O((n/m) \log^2 m)$ update time, and $O((n/m) \log m)$ query time $(1 \leq m \leq n)$. We can then obtain a dynamic planar version of Corollary 4.1.2:

Lemma 4.1.5 *A sequence of q ray shooting queries in a polygon defined by a dynamic set H of at most n halfplanes in E^2, and q insertions/deletions on H can be performed in $O(n \log q + q \log^2 n)$ time and $O(n)$ space.*

Proof: By the above method, the total time needed to perform q queries and updates is $O(n \log m + q(\frac{n}{m} \log^2 m))$, where $1 \leq m \leq n$. Choose $m = q \log q$ when $q \leq n/\log n$ and choose $m = n$ otherwise. $\square$

In higher dimensions, Matoušek and Schwarzkopf [MS93] have provided dynamic versions of their data structures, as shown in the bottom half of Table 4.1. Structure 1′ uses the dynamic partition trees from Agarwal and Matoušek [AM91] and is similar to the static Structure 1, except that $(1/r)$-nets are replaced by more robust simplicial partitions and the parameter r is set to a large enough constant rather than the number n^γ. This causes the polylogarithmic factor in the query time to increase to an n^ε factor. Structure $3'$ is based on Structure 3 and again uses dynamization techniques from [AM91]. Higher-level shallow cuttings are used, and the parameter r is also set to a constant; as a result, space and preprocessing time is now $O(m^{1+\varepsilon})$ instead of $O(m \log^{O(1)} n)$. Finally, Structure $2'$ combines the approaches in the other two dynamic structures to yield a continuous tradeoff.

We can apply our grouping scheme to get a preprocessing time/query time tradeoff for Structure $1'$. This modification is easily shown to have $O(n \log m)$ preprocessing time, $O(n)$ space, $O((n/m) \log^2 m)$ amortized update time, and $O(n/m^{1/\lfloor d/2 \rfloor - \varepsilon})$ query time $(1 \leq m \leq n)$. As a consequence, we get the following:

Lemma 4.1.6 *A sequence of q ray shooting queries in a polytope defined by a dynamic set H of at most n halfspaces in E^d $(d > 2)$ can be performed in*

(i) $O(n \log q + (nq)^{1-1/(\lfloor d/2 \rfloor + 1) + \varepsilon} + qn^{1-2/(\lfloor d/2 \rfloor + 1) + \varepsilon})$ time, if the number of insertions/deletions is $O(q)$;

(ii) $O(n \log^2 n + (nq)^{1-1/(\lfloor d/2 \rfloor + 1) + \varepsilon} + q \log n)$ time, if the number of insertions/deletions is $O(n)$.

Proof: For (i), we consider three cases:

CASE I. $q \leq n^{1/\lfloor d/2 \rfloor - \varepsilon}$. Use the above modification of Structure $1'$ with $m^{1/\lfloor d/2 \rfloor - \varepsilon} = q$ $(1 \leq m \leq n)$. Then the running time is

$$O\left(n \log m + q\frac{n}{m} \log^2 m + q\frac{n}{m^{1/\lfloor d/2 \rfloor - \varepsilon}}\right) = O(n \log q).$$

CASE II. $n^{1/\lfloor d/2 \rfloor} < q < n$. Use Structure $2'$ with $m = (nq)^{1-1/(\lfloor d/2 \rfloor + 1)}$ $(n \leq m \leq n^{\lfloor d/2 \rfloor})$. Then the running time is

$$O\left(m^{1+\varepsilon} + q\frac{m^{1+\varepsilon}}{n} + q\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n\right) = O((nq)^{1-1/(\lfloor d/2 \rfloor + 1) + \varepsilon}).$$

CASE III. $q \geq n$. Use Structure 2′ with $m = n^{2-2/(\lfloor d/2 \rfloor+1)}$ $(n \leq m \leq n^{\lfloor d/2 \rfloor})$. Then the running time is

$$O\left(m^{1+\varepsilon} + q\frac{m^{1+\varepsilon}}{n} + q\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log n\right) = O(qn^{1-2/(\lfloor d/2 \rfloor+1)+2\varepsilon}).$$

For (ii), we perform a similar analysis. We use Structure 1′ when $q \leq n^{1/\lfloor d/2 \rfloor - \varepsilon}$, Structure 2′ with $m = (nq)^{1-1/(\lfloor d/2 \rfloor+1)}$ when $n^{1/\lfloor d/2 \rfloor} < q < n^{\lfloor d/2 \rfloor}$, and Structure 3′ when $q \geq n^{\lfloor d/2 \rfloor + \varepsilon}$. □

4.2 Linear Programming Queries

In this section, we apply similar techniques to those of the previous section to answer linear programming queries. Given a collection H of n halfspaces in E^d, each containing the origin o, a *linear programming query* is to determine the vertex v of the polytope $\bigcap H$ that maximizes $\xi \cdot v$ for a query vector $\xi \in E^d$.

We begin by extending the grouping technique of Lemmas 4.1.1 and 4.1.3 to handle linear programming with a small number of queries. This is not trivial because linear programming, unlike ray shooting, is not a decomposable problem.

Lemma 4.2.1 *There is a dynamic data structure for linear programming queries on a set H of n halfplanes in E^2 with $O(n \log m)$ preprocessing time, $O(n)$ space, and $O((n/m)\log^2 m)$ update and query time, where m is a parameter between* 1 *and* n.

Proof: We consider the static case first. Partition H into $\lceil n/m \rceil$ groups $H_1, \ldots, H_{\lceil n/m \rceil}$, each of size at most m, compute the convex polygon $\Pi_i = \bigcap H_i$ for each i, and store each of them in an ordered array. The total preprocessing time is then $O(\frac{n}{m}(m \log m)) = O(n \log m)$, while space is linear. Reichling [Rei88a] showed that in $O(k \log^2 m)$ time, one can detect whether the intersection of k convex m-gons is empty, and if not, report the point in the intersection that is extreme in a given direction ξ; his method is based on Megiddo's prune-and-search technique. Using Reichling's algorithm on the $k = \lceil n/m \rceil$ polygons $\Pi_1, \ldots, \Pi_{\lceil n/m \rceil}$, we can answer a linear programming query in $O((n/m) \log^2 m)$ time.

The dynamic part can be proved using Overmars and van Leeuwen's data structure [OvL81] to store each of the H_i's, which requires $O((n/m) \log^2 m)$ update time; Reichling's time bound still applies. $\square$

As a result of this lemma, q linear programming queries in the plane can be answered in $O(n \log q)$ time for $q \leq n/\log n$.

Corollary 4.2.2 *A sequence of q linear programming queries and q insertions/deletions on a dynamic set H of at most n halfplanes in E^2 can be performed in $O(n \log q + q \log^2 n)$ time and $O(n)$ space.*

In higher dimensions, Matoušek [Mat93] obtained data structures for linear programming queries achieving the complexities shown in Table 4.1. His approach is as follows: He first showed how a multidimensional parametric search technique can reduce the problem of answering linear programming queries to that of answering halfspace-emptiness queries with witness. (In the dual setting, a *halfspace-emptiness query* on H is to determine whether a given query point p belongs to $\bigcap H$, and if not, provide a *witness* halfspace $h \in H$ that does not contain p.) Then he obtained data structures for the halfspace-emptiness problem using techniques completely analogous to those used in the ray shooting problem. (In fact, the halfspace-emptiness is a special case of ray shooting.)

We now show how to obtain a preprocessing time/query time tradeoff for Structure 1 in the case of linear programming queries. The bounds we get are similar to those obtained in Lemma 4.1.3, except for an extra polylogarithmic factor in n in the query time; this causes an additional $O(n \log\log n)$ term in the overall time bound.

Lemma 4.2.3 *There is a (static) data structure for linear programming queries on a set H of n halfspaces in E^d $(d > 2)$ with $O(n \log m)$ preprocessing time, $O(n)$ space, and $O((n/m^{1/\lfloor d/2 \rfloor}) \log^{O(1)} m \log^d n)$ query time, where m is a parameter between 1 and n.*

Proof: In this proof, we assume that the reader is familiar with Matoušek's technique [Mat93].

We consider the halfspace-emptiness queries first. Partition H into $\lceil n/m \rceil$ groups $H_1, \ldots, H_{\lceil n/m \rceil}$, each of size at most m. For each of the H_i's, we build a data structure [Mat93] with $O(m \log m)$ preprocessing time and $O(m)$ space, so that each halfspace-emptiness query on H_i can be answered in $O(\log m)$ parallel steps using $O(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)$ processors. The total preprocessing time is then $O(\frac{n}{m}(m \log m)) = O(n \log m)$ and the space requirement remains linear. Since the halfspace-emptiness problem is decomposable, a query on H can be performed in $\tau(n, m) = O(\log m)$ parallel steps using $\pi(n, m) = O(\frac{n}{m}(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)) = O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{O(1)} m)$ processors; or sequentially, in $t(n, m) = O(\frac{n}{m}(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)) = O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{O(1)} m)$ time.

Matoušek has shown that any data structure for halfspace-emptiness queries (satisfying some reasonable conditions) can be used to answer linear programming queries by a multidimensional version of Megiddo's parametric search method [Meg83a]. The resulting query time is given by $O(t(n, m)\tau(n, m)^d \log^d \pi(n, m))$, which, in our case, is $O(\frac{n}{m^{1/\lfloor d/2 \rfloor}} \log^{O(1)} m \log^d n)$. □

Corollary 4.2.4 *A sequence of q linear programming queries on a set H of n halfspaces in E^d $(d > 2)$ can be performed in $O(n \log\log n + n \log q + (nq)^{1-1/(\lfloor d/2 \rfloor + 1)} \log^{O(1)} n + q \log^{d+1} n)$ time.*

Proof: The proof is as in Corollary 4.1.4, except that for Case I ($q \leq n^{1/\lfloor d/2 \rfloor} / \log^K n$) we use Lemma 4.2.3 with $m = (q \log^K n)^{\lfloor d/2 \rfloor}$ ($1 \leq m \leq n$). The running time for Case I now becomes

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

□

Remark: Again, the complexity remains the same even if the value of q is not known in advance. (Use the sequence $q_i = (\log n)^{2^i}$ ($i = 1, 2, \ldots$) for Case I.)

Since Matoušek's data structures can be made dynamic (see Table 4.1), the following analogue of Lemma 4.1.6 is straightforward:

Lemma 4.2.5 *A sequence of q linear programming queries on a dynamic set H of at most n halfspaces in E^d ($d > 2$) can be performed in*

(i) $O(n \log\log n + n \log q + (nq)^{1-1/(\lfloor d/2 \rfloor+1)+\varepsilon} + qn^{1-2/(\lfloor d/2 \rfloor+1)+\varepsilon})$ *time, if the number of insertions/deletions is $O(q)$;*

(ii) $O(n \log^2 n + (nq)^{1-1/(\lfloor d/2 \rfloor+1)+\varepsilon} + q \log^{d+1} n)$ *time, if the number of insertions/deletions is $O(n)$.*

Finally, we observe that for the semidynamic case, where there are no deletions, Lemma 4.2.5(ii) may be improved somewhat.

Lemma 4.2.6 *A sequence of q linear programming queries and n insertions on an initially empty set of halfspaces in E^d can be performed in $O(n \log^2 n + (nq)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n + q \log^{O(1)} n)$ time.*

Proof: As in the proof of Lemma 4.2.3, we consider the halfspace-emptiness problem first. Since, this problem is decomposable, the techniques by Bentley and Saxe [BS80] may be applied to convert a static structure to a semidynamic one (which increases building time and query time by a logarithmic factor). We then apply Matoušek's parametric search to use this structure for answering linear programming queries. The resulting time bound is only a polylogarithmic factor increase on the static bound in Corollary 4.2.4. $\square$

In Section 4.7, we show how to eliminate the $\log n$ factors in Lemma 4.2.3, and thus remove the $n \log\log n$ term from both Corollary 4.2.4 and Lemma 4.2.5(i), if *randomization* is allowed.

4.3 A Convex Hull Algorithm and an Extrema Algorithm in Any Fixed Dimension

In this section, we apply the results from Sections 4.1 and 4.2 to obtain output-sensitive algorithms for the problem of constructing convex hulls and the related problem of enumerating extreme points, in any fixed dimension.

We first show that the f-face convex hull of an n-point set in E^d can be constructed by performing $O(f)$ ray shooting queries in a d-dimensional polytope defined by n halfspaces. The algorithm that we use is just the well-known gift-wrapping method [CK70, PS85, Swa85] dualized, since a *wrapping step* (recall Section 2.2) corresponds to shooting a ray in the dual polytope. If the ray shooting queries are performed directly by scanning the halfspaces, then we get an $O(nf)$-time bound. We observe that this can be improved using the data structures from Section 4.1.

Theorem 4.3.1 *The convex hull of a set P of n points in E^d can be constructed in $O(n \log f + (nf)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ time (and $O(n + (nf)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ space), where f is the number of hull faces.*

Proof: In the dual setting, our problem becomes computing an intersection of a set H of n halfspaces in E^d (assumed to be in general position), each containing the origin o. It suffices to compute the vertices of the intersection $\bigcap H$, from which one can easily generate the complete facial lattice structure of $\bigcap H$ in $O(f \log f)$ time by a dictionary.

First, an initial vertex v_0 can be found by performing d ray shooting queries in $\bigcap H$, since shooting a ray from o gives a point in a $(d-1)$-face, and shooting a ray from a point in a j-face inside its affine hull (i.e., inside the j-flat containing the face) gives a point in a $(j-1)$-face $(1 \leq j < d)$. Furthermore, given a vertex v, the vertices adjacent to v in the *1-skeleton* (the graph formed by the vertices and edges of $\bigcap H$) can be found by performing d ray shooting queries: if $h_1, \ldots, h_d$ are the hyperplanes defining v, then shoot a ray from v along each of the d lines formed by intersecting $d-1$ hyperplanes from $\{h_1, \ldots, h_d\}$.

Since the 1-skeleton is connected, we can use a depth-first search or a breadth-first search to visit all vertices of $\bigcap H$; we can ensure that each vertex is visited only once by using a dictionary to detect replication (as in the 3-d algorithm in Section 2.2.2). This shows that the vertices of $\bigcap H$ can be computed by performing $O(f)$ ray shooting queries in $\bigcap H$. The theorem then follows by applying Corollaries 4.1.2 and 4.1.4 (recall that $f = O(n^{\lfloor d/2 \rfloor})$). □

Next, we consider the problem of finding the h extreme points of an n-point set P in E^d. Although in two and three dimensions this problem has the same complexity as the convex hull problem, the extreme point problem in higher dimensions is generally a less demanding problem, since we are required to output only the vertices of the convex hull, not the entire facial lattice structure.

Determining whether a given point is extreme is reducible to solving a linear program on a set of n dual halfspaces. By testing all n points in P, we can find all the extreme points in n linear programming queries. Since Megiddo [Meg84] showed that a linear program can be solved in linear time for any constant d, the extreme point problem can be solved in quadratic time, as is well known. Using the data structures from Section 4.2 (Corollary 4.2.4), the n linear programming queries can in fact be solved in $O(n^{2-2/(\lfloor d/2\rfloor+1)}\log^{O(1)} n)$ time, which implies a subquadratic bound for enumerating extreme points, as Matoušek [Mat93] has observed. We now show that the bound can be improved to an output-sensitive one (i.e., one that depends on both n and h).

As we have noted, n linear programming queries on n halfspaces are sufficient to solve the extreme point problem. We observe that the number of queries or the number of halfspaces can be reduced if h is small. Specifically, we give a simple algorithm that finds the extreme points using h queries on n halfspaces together with n queries on h halfspaces. Using Megiddo's linear programming algorithm, this leads to an $O(nh)$-time extrema algorithm which was also discovered recently by Clarkson [Cla94] and Ottmann et al. [OSS95]. We observe that the time bound can be further improved using the results from Section 4.2.

Theorem 4.3.2 *The h extreme points of a set P of n points in E^d can be identified in $O(n\log^{d+1} h + (nh)^{1-1/(\lfloor d/2\rfloor+1)+\varepsilon})$ time or in $O(n\log^{O(1)} h + (nh)^{1-1/(\lfloor d/2\rfloor+1)}\log^{O(1)} n)$ time.*

Proof: Without loss of generality, assume that the origin o is in the interior of $\mathrm{conv}(P)$. Consider the following incremental algorithm, which is essentially the same as Clarkson's algorithm [Cla94] and the algorithm by Ottmann et al. [OSS95]:

```
Algorithm Extrema($P$)
1. $Q \leftarrow \emptyset$
2. for each $p \in P$ (in any order) do
3.     if $p \notin Q$ and $p \notin \mathrm{conv}(Q \cup \{o\})$ then
4.         if $p$ is an extreme point of $P$ then
5.             $Q \leftarrow Q \cup \{p\}$
6.         else find the facet $f$ of $\mathrm{conv}(P)$ that intersects ray $\overrightarrow{op}$
7.             let $v$ be a vertex of $f$ that is not in $Q$
8.             $Q \leftarrow Q \cup \{v\}$
9. return $Q$
```

Observe that v must exist in line 7, because otherwise all vertices of f would be in Q; since $p \in \mathrm{conv}(f \cup \{o\})$, this would imply that $p \in \mathrm{conv}(Q \cup \{o\})$: a contradiction with line 3. It is then clear that the algorithm correctly returns the set of extreme points of P.

We now analyze the cost of the algorithm. Note that line 3 can be done by solving a linear program on Q in the dual and lines 4 and 6 can be done by solving a linear program on P in the dual. (Line 7 takes constant time since each facet has d vertices by the general position assumption.) Observe that although line 3 is executed n times, lines 4–8 are executed only h times since each execution adds a new point to Q. Thus, the algorithm requires h linear programming queries on P, a static set of size n, and n linear programming queries on Q, a semidynamic set of size at most h.

By Corollary 4.2.4, the h queries on P can be done in $O(n \log\log n + n \log h + (nh)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ time. By Lemma 4.2.5(ii), the n queries and h insertions on Q can be done in $O((nh)^{1-1/(\lfloor d/2 \rfloor+1)+\varepsilon} + n \log^{d+1} h)$ time. The total running time is then $O(n \log\log n + n \log^{d+1} h + (nh)^{1-1/(\lfloor d/2 \rfloor+1)+\varepsilon})$.

Notice that when $h \le n^{\alpha}$ for a constant $\alpha < (1/\lfloor d/2 \rfloor)^2$, the number of hull faces is $O(n^{1/\lfloor d/2 \rfloor - \varepsilon})$; so we can compute the entire convex hull in optimal $O(n \log h)$ time and

$O(n)$ space by Theorem 4.3.1. This allows us to remove the $O(n \log\log n)$ term in the time bound. The first part of the theorem is thus proved, and the second part follows similarly, using Lemma 4.2.6 instead of Lemma 4.2.5(ii) for Q. $\square$

Theorem 4.3.2 has an interesting corollary: the h-vertex convex hull of an n-point set in d dimensions can be constructed in $O(n \log^{O(1)} h + h^{\lfloor d/2 \rfloor})$ time. In terms of n and h, this bound is within a polylogarithmic factor of optimal in the worst case for the convex hull problem, since $\Omega(n \log h + h^{\lfloor d/2 \rfloor})$ is a lower bound. This bound is not output-sensitive though, since the output size f can range anywhere from $\Theta(h)$ to $\Theta(h^{\lfloor d/2 \rfloor})$.

Corollary 4.3.3 *The convex hull of a set P of n points in E^d can be constructed in $O(n \log^{O(1)} h + h^{\lfloor d/2 \rfloor})$ time, where h is the number of hull vertices.*

Proof: Compute the extreme points by Theorem 4.3.2 and then construct the convex hull of these h points by Chazelle's algorithm [Cha93b] in $O(h^{\lfloor d/2 \rfloor})$ time (note that when $h = \Omega(n^{1/\lfloor d/2 \rfloor})$, we have $h^{\lfloor d/2 \rfloor} = \Omega((nh)^{1-1/(\lfloor d/2 \rfloor + 1)})$). $\square$

4.4 Application: Convex Layers and Depths

We now discuss an application of the output-sensitive convex hull and extrema algorithms from the previous section. Let $P \subseteq E^d$ be a n-point set. The *convex layers* (or "onion" layers) of P are defined iteratively as follows: layer 1 is convex hull of P, and if layer i is non-empty, then layer $i+1$ is defined as the convex hull of the points of P that are not vertices of the previous layers $1, \ldots, i$ (see Figure 4.10). If P_i denotes the vertices of the i-th layer, then P_{i+1} can be expressed recursively as the extreme points of $P - (P_1 \cup \ldots \cup P_i)$. Every point $p \in P$ is a vertex of exactly one layer; that is, there is an unique i for which $p \in P_i$. This index i is called the *depth of* p.

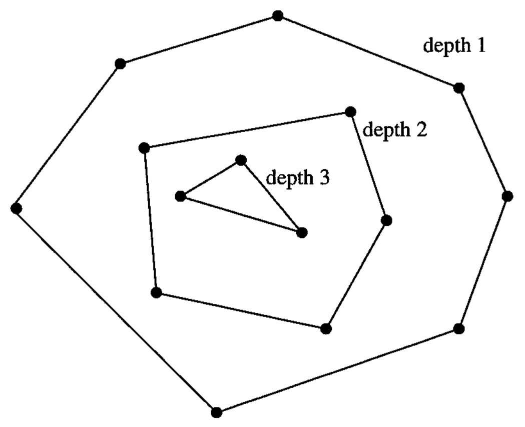

Figure 4.10: The convex layers of a planar point set.

Two problems come to mind: (i) computing the facial structure of each of the convex layers, and (ii) computing just the vertices of the convex layers. The *convex layers problem* here refers to the first problem. We call the second problem the *depth problem*, because it is equivalent to finding the depths of all the points. As Preparata and Shamos [PS85] described, one motivation of these problems is from robust estimation in the area of statistics.

For $d = 2$, Chazelle [Cha85] gave an optimal $O(n \log n)$-time algorithm for the convex layers problem (and thus, the depth problem as well). For $d = 3$, a near-optimal $O(n^{1+\varepsilon})$-time method was provided by Agarwal and Matoušek [AM93]. For $d \geq 4$, the problems have been less well studied. As the total number of faces over all convex layers can range from $\Theta(n)$ to $\Theta(n^{\lfloor d/2 \rfloor})$, the output size of the convex layers problem can be huge in higher dimensions. In applications where we only need to know which point lies in which layer, the depth problem is therefore of more interest.

Since an $O(n^3)$-time solution to the depth problem in any constant dimension d can be easily derived using linear programming, Edelsbrunner [Ede87, Problem 10.3(c)] asked whether there is an $o(n^3)$ method for $d \geq 4$. By repeatedly computing extreme points and removing points, one can solve the depth problem in $O(n^2)$ linear programming queries and $O(n)$ deletions on n halfspaces, which by Lemma 4.2.5(ii), take $O(n^{3-3/(\lfloor d/2 \rfloor+1)+\varepsilon})$ time. However, as Ottmann et al. [OSS95] have pointed out recently, a simpler and better solution is to apply an $O(nh)$-time extrema algorithm (like the one in Section 4.3) to compute the vertices of each layer. Since the sum of the number of extreme points over all layers is just n, the vertices of all layers are identified in only $O(n^2)$ time.

Here, we show how a hybrid of the convex hull and extrema algorithms from Section 4.3 leads to an improved subquadratic solution to the depth problem for any d; for example, the time bound obtained is $O(n^{8/5+\varepsilon})$ for $d = 4$ or $d = 5$. We then show how this result implies an output-sensitive algorithm for the convex layers problem.

Theorem 4.4.1 *The depth of all points in a set P of n points in E^d can be computed in $O(n^{2-\beta+\varepsilon})$ time, where $\beta = 2/(\lfloor d/2 \rfloor^2 + 1)$.*

Proof: We iteratively compute the vertices of the i-th layer ($i = 1, 2, \ldots$) as follows. We use the convex hull algorithm in Theorem 4.3.1 to construct the i-th layer, but as soon as more than n^β vertices are discovered in the layer, we stop the computation and switch to the extrema algorithm in Theorem 4.3.2 to compute the vertices of the layer. We then remove the vertices of the i-th layer from P and proceed to the $(i+1)$-st layer. At the end, we will have the depths of every point in P. For the calls to the convex hull algorithm, we will use a dynamic ray shooting data structure instead of a static one so that structures don't have to be rebuilt as points are removed from P after each iteration; for the calls to the extrema algorithm, however, we will leave the data structures unchanged.

Let h_i denote the number of vertices of the i-th layer ($\sum_i h_i = n$). We first analyze the cost of the calls to the convex hull algorithm in Theorem 4.3.1, which involve a number of ray shooting queries and n deletions on a dynamic set of at most n halfspaces; the number of queries is proportional to the number of facets discovered. Since we stop the computation in a layer when n^β vertices are found, we make at most $O(\min\{h_i^{\lfloor d/2\rfloor}, n^{\beta\lfloor d/2\rfloor}\})$ queries for the i-th layer. The total number of queries is then asymptotically bounded by

$$\begin{aligned}\sum_{h_i\le n^\beta} h_i^{\lfloor d/2\rfloor} + \sum_{h_i>n^\beta} n^{\beta\lfloor d/2\rfloor} &\le n^{\beta(\lfloor d/2\rfloor-1)}\left(\sum_{h_i\le n^\beta} h_i\right) + n^{\beta(\lfloor d/2\rfloor-1)}\left(\sum_{h_i>n^\beta} n^\beta\right)\\ &\le n^{\beta(\lfloor d/2\rfloor-1)}\sum_i h_i \le n^{1+\beta(\lfloor d/2\rfloor-1)}.\end{aligned}$$

By Lemma 2.6(ii), we see that the cost of these queries is $O((n^{2+\beta(\lfloor d/2\rfloor-1)})^{1-1/(\lfloor d/2\rfloor+1)+\varepsilon})$ $= O(n^{2-\beta+c\varepsilon})$ by our choice of β (where c is an appropriate constant).

Next, we analyze the cost of the calls to the extrema algorithm in Theorem 4.3.2. Note that the extrema algorithm is called only for the layers i with $h_i > n^\beta$, and the number of h_i's with $h_i > n^\beta$ is at most $n^{1-\beta}$ (since $\sum_i h_i = n$). Ignoring logarithmic factors, the cost is then

$$\begin{aligned}\sum_{h_i>n^\beta}\left(n + (nh_i)^{1-1/(\lfloor d/2\rfloor+1)}\right) &\le n\left(\sum_{h_i>n^\beta} 1\right) + n^{1-1/(\lfloor d/2\rfloor+1)}\left(\sum_{h_i>n^\beta} h_i^{1-1/(\lfloor d/2\rfloor+1)}\right)\\ &\le n\,(n^{1-\beta}) + n^{1-1/(\lfloor d/2\rfloor+1)}\left(\sum_{h_i>n^\beta} h_i\right)^{1-1/(\lfloor d/2\rfloor+1)}\left(\sum_{h_i>n^\beta} 1\right)^{1/(\lfloor d/2\rfloor+1)}\\ &\le n^{2-\beta} + n^{1-1/(\lfloor d/2\rfloor+1)}\left(n^{1-1/(\lfloor d/2\rfloor+1)}\right)\left(n^{1-\beta}\right)^{1/(\lfloor d/2\rfloor+1)} = O(n^{2-\beta}),\end{aligned}$$

by Hölder's inequality. Therefore, the entire method runs in $O(n^{2-\beta+c\varepsilon})$ time. □

Corollary 4.4.2 *The convex layers of a set P of n points in E^d can be constructed in $O(n^{2-\beta+\varepsilon} + f\log n)$ time, where $\beta = 2/(\lfloor d/2\rfloor^2+1)$ and f is now the total number of faces in all convex layers.*

Proof: Let P_i be the vertices of layer i (i.e., the points of depth i) and let h_i and f_i be the number of vertices and faces of the layer ($\sum_i h_i = n$, $\sum_i f_i = f$). We first compute

P_i for all i in $O(n^{2-\beta+\varepsilon})$ time by Theorem 4.4.1 and then construct the convex hull of each P_i using Seidel's algorithm [Sei86] with Matoušek's improvement [Mat93]. The total time needed is $O(n^{2-\beta+\varepsilon}+\sum_i(h_i^{2-2/(\lfloor d/2\rfloor+1)+\varepsilon}+f_i\log h_i))=O(n^{2-\beta+\varepsilon}+f\log n)$. □

Remarks:

1. A worst-case optimal convex layers algorithm for $d\geq 4$ is not difficult to get: just use an $O(nh)$-time extrema algorithm to compute the vertices of each layer and use Chazelle's convex hull algorithm [Cha93b] to construct the layers; then the running time is $O(n^2+n^{\lfloor d/2\rfloor})$.

2. For a more direct output-sensitive convex layers algorithm, we can simply do the following: iteratively use the convex hull algorithm in Theorem 4.3.1 to construct the i-th layer ($i=1,2,\ldots$) and delete points from P that are vertices of a layer after each iteration. This method is the same as the one by Agarwal and Matoušek [AM93] (proceedings version) for the three-dimensional case. It requires $O(f)$ ray shooting queries and n deletions, and by Lemma 4.1.6(ii), takes $O((nf)^{1-1/(\lfloor d/2\rfloor+1)+\varepsilon})$ time, which is superior to the bound in Corollary 4.4.2 only when the output size f is near linear (recall $\Omega(n)=f=O(n^{\lfloor d/2\rfloor})$).

4.5 Further Applications: Levels in Arrangements and Linear Programming with Violations

This section describes some more applications of the results in Sections 4.1 and 4.2. Problems considered here include the output-sensitive construction of an interesting geometric structure known as the *k-level* [Ede87], and also a variant of the familiar linear programming problem.

We first consider the construction of a k-level in a hyperplane arrangement. Given a

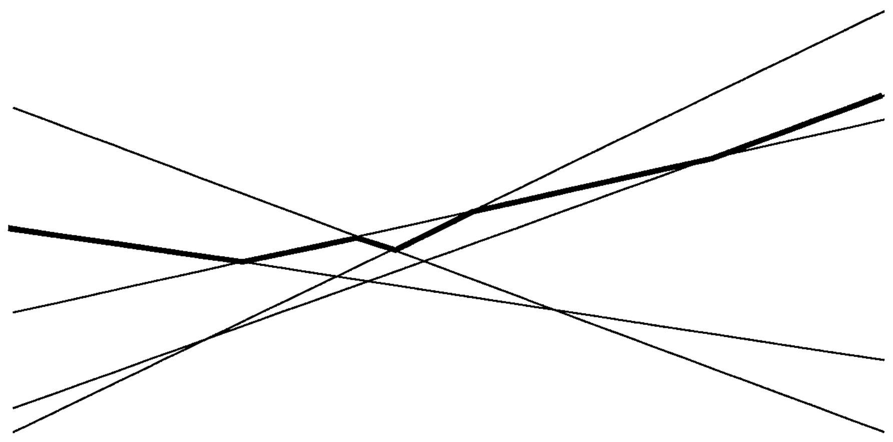

Figure 4.11: The boundary of the 1-level in an arrangement of lines. (Shown in bold.)

set H of n hyperplanes in E^d, the *k-level* in the arrangement $\mathcal{A}(H)$ is defined as the set of all points in E^d that have at most k hyperplanes of H above it $(0 \leq k < n)$. The 0-level (the *upper envelope*) is just the dual of a convex hull. Figure 4.11 shows an example of a k-level with $k = 1$. Let f denote the size (combinatorial complexity) of the k-level.

In the plane, an output-sensitive algorithm for constructing the k-level was given by Edelsbrunner and Welzl [EW86]. We improve its running time from $O(n \log n + f \log^2 n)$ to $O(n \log f + f \log^2 n)$. (Alternatively, Cole, Sharir, and Yap [CSY87] gave an algorithm that runs in $O(n \log n + (n+f) \log^2 k)$ time.) In higher dimensions, Agarwal and Matoušek [AM93] proposed a depth-first search method based on ray shooting queries, which runs in $O(n \log n + f^{1+\varepsilon})$ time for $d = 3$ (actually they state a weaker $O((n+f)n^{\varepsilon})$ bound). We improve this to $O(n \log f + f^{1+\varepsilon})$ and analyze the running time in higher dimensions. Tight worst-case bounds on the size f of a k-level is currently not known even for $d = 2$; thus, achieving output-sensitivity is particularly important here.

Theorem 4.5.1 *A k-level in an arrangement $\mathcal{A}(H)$ of n hyperplanes in E^d can be constructed in* *(i)* $O(n \log f + f \log^2 n)$ *time, if* $d = 2$*;*

(ii) $O(n \log f + f^{1+\varepsilon})$ *time, if* $d = 3$*;*

(iii) $O(n \log f + (nf)^{1-1/(\lfloor d/2 \rfloor+1)+\varepsilon} + fn^{1-2/(\lfloor d/2 \rfloor+1)+\varepsilon})$ *time, if* $d \geq 4$*;*

where f *is the output size.*

Proof: The depth-first search algorithm by Agarwal and Matoušek [AM93] (proceedings version) constructs the k-level using $O(f)$ polytope ray shooting queries and $O(f)$ insertions/deletions on two dynamic sets of at most n halfspaces. (In two dimensions, their algorithm is the same as Edelsbrunner and Welzl's [EW86].) Hence, the theorem follows from Lemmas 4.1.5 and 4.1.6(i). □

Next, we consider the following problem: given a set H of n hyperplanes in E^d, a direction ξ and a number $0 \leq k < n$, find a point in the k-level of $\mathcal{A}(H)$ that is minimal along ξ; in other words, find a minimal point that lies on or above all but at most k of the hyperplanes in H. This is the feasible case of the *linear programming problem with at most k violated constraints.* If $k = 0$, it is just an ordinary linear programming problem and can be solved in $O(n)$ time by Megiddo's algorithm [Meg84]. We are interested in the case when $k > 0$ is a small integer; that is, we allow only a few violations of the constraints.

For this k-violation problem, Matoušek [Mat94] has devised a method that not only finds the minimum but also enumerates all $O(k^d)$ local minima in the $(\leq k)$-levels. His method is a depth-first search procedure that uses linear programming queries. It runs in $O(n \log n + k^2 \log^2 n)$ time if $d = 2$ and $O(n \log n + k^{3+\varepsilon})$ time if $d = 3$; the running time is $O(n \log n)$ if $d \geq 4$ and k is sufficiently small. We show how the $O(n \log n)$ terms in these bounds can be reduced to $O(n \log k)$ in two dimensions or to $O(n \log \log n + n \log k)$

in higher dimensions. As we will see in Section 4.7, the extra $O(n \log\log n)$ term can even be removed using randomization.

Theorem 4.5.2 *The linear programming problem on n constraints in E^d with at most k violations for the feasible case can be solved in*

(i) $O(n \log k + k^2 \log^2 n)$ *time, if* $d = 2$*;*

(ii) $O(n \log\log n + n \log k + k^{3+\varepsilon})$ *time, if* $d = 3$*;*

(iii) $O(n \log\log n + n \log k)$ *time, if* $d \geq 4$ *and* $k^d \leq n^{1/\lfloor d/2 \rfloor - \varepsilon}$.

Proof: The depth-first search algorithm by Matoušek [Mat94] solves this problem using $O(k^d)$ linear programming/membership queries and $O(k^d)$ insertions/deletions on two dynamic sets of at most n halfspaces. (A membership query is just a special case of a ray shooting query.) Hence, part (i) of the theorem follows from Lemma 4.1.5 and Corollary 4.2.2, and parts (ii) and (iii) follow from Lemmas 4.1.6(i) and 4.2.5(i). □

Remark: As Roos and Widmayer [RW94] observed, the planar feasible linear programming problem with at most k violated constraints can be solved in $O(n \log^2 n)$ time, if one uses an $O(n \log n)$ slope selection algorithm [CS+89] and performs binary search. Roos and Widmayer also studied a related but different planar problem, which in our terminology corresponds to finding the y-maximum point on the *boundary* of the k-level. They give a slightly improved $O(n \log n + k \log^2 k)$-time method for this problem; however, this method does not seem to apply to our k-violation linear programming problem.

The techniques here may also be applicable to the *infeasible case* of linear programming with k violated constraints (which has applications to separation and transversal problems [ERvK93]), or to the *smallest k-enclosing circle problem*; see Matoušek's paper [Mat94].

As we now show, Matoušek's approach on linear programming with violations can in fact be used to improve an output-sensitive algorithm by Mulmuley [Mul90] for constructing $(\leq k)$-*levels*—i.e., the i-level for all $i = 0, 1, \ldots, k$—of a non-redundant arrangement $\mathcal{A}(H)$ of n hyperplanes in E^d. Here, $\mathcal{A}(H)$ is *non-redundant* if for every $h \in H$, the upper envelope of $H - \{h\}$ coincides with the upper envelope of H.

Mulmuley's algorithm can be regarded as a generalization of Seidel's output-sensitive convex hull algorithm [Sei86]. While Seidel's algorithm constructs convex hulls in $O(n^2 + f \log n)$ time, Mulmuley's algorithm constructs $(\leq k)$-levels in $O(n^2 k^{d-1} + f \log n)$ time. In both cases, f denotes output size. Matoušek [Mat93] had already shown how to reduce the first term of the running time of Seidel's algorithm to $O(n^{2-2/(\lfloor d/2 \rfloor+1)+\varepsilon})$ (actually, $O(n^{2-2/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$). Here, we show how to reduce the first term of the running time of Mulmuley's algorithm to $O(n^{2-2/(\lfloor d/2 \rfloor+1)+\varepsilon} k^{d-1})$. Using the lifting map from Section 3.3, this result can be used in the construction of *Voronoi diagrams of order* $0, 1, \ldots, k$ in one dimension lower; see [ES86, Mul90].

Theorem 4.5.3 *We can compute i-levels in a non-redundant arrangement $\mathcal{A}(H)$ of n hyperplanes in E^d for all $i = 0, 1, \ldots, k$ in $O(n^{2-2/(\lfloor d/2 \rfloor+1)+\varepsilon} k^{d-1} + f \log n)$ time, where f is the output size.*

Proof: Let $L_i(H)$ denote the boundary of the i-level in $\mathcal{A}(H)$ and let f_i be its size ($\sum_{i=0}^{k} f_i = f$). For each $h \in H$, let $H_h = \{h \cap h' : h' \in H - \{h\}\}$, which is a set of $(d-1)$-dimensional hyperplanes.

Mulmuley [Mul90] gave an algorithm which constructs the facial structure of $L_i(H)$ in $O((f_i + f_{i-1}) \log n)$ time, given the following information:

1. the local minima in $L_i(H) - L_{i-1}(H)$ (along some predefined direction),

2. the local minima in $L_i(H_h) - L_{i-1}(H_h)$ for each $h \in H$, and
3. the facial structure of $L_{i-1}(H)$.

Matoušek [Mat94] has shown that the local minima of all i-levels in $\mathcal{A}(H)$ ($i = 0, 1, \ldots, k$) can be enumerated by performing $O(k^d)$ linear programming/membership queries and $O(k^d)$ insertions/deletions on two dynamic sets of at most n halfspaces. Similarly, the local minima of all i-levels in $\mathcal{A}(H_h)$ ($i = 0, 1, \ldots, k$) can be computed using $O(k^{d-1})$ linear programming/membership queries and $O(k^{d-1})$ insertions/deletions, for each $h \in H$. Observe that we do not need separate structures to store each H_h as the data structures from Section 4.2 can perform linear programming queries restricted to any j-flats [Mat93]. The total number of queries and updates is then $O(k^d + nk^{d-1}) = O(nk^{d-1})$. By Lemmas 4.1.6(i) and 4.2.5(i), this takes $O(n^{2-2/(\lfloor d/2 \rfloor + 1)+\varepsilon} k^{d-1})$ time.

Thus, items 1 and 2, for all $i = 0, 1, \ldots, k$, can be computed in $O(n^{2-2/(\lfloor d/2 \rfloor + 1)+\varepsilon} k^{d-1})$ time. Now, Mulmuley's algorithm can be used to construct the facial structure of $L_0(H), L_1(H), \ldots, L_k(H)$ incrementally, in additional $O(f \log n)$ time. $\square$

Remark: In the worst case, the total size f of the ($\leq k$)-levels can be as large as $\Theta(n^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil})$ and this bound is tight [CS89]. Currently, algorithms for constructing the ($\leq k$)-levels that are optimal for worst-case output are known in dimension 2 by Everett et al. [ERvK93] and in dimensions ≥ 4 by Mulmuley [Mul91]. The latter result of Mulmuley is randomized.

Before we close this section, we remark that further applications of our ideas are possible. For example, Theorem 4.3.1 can be extended to compute the intersection of a convex hull with a j-flat in an output-sensitive manner; in the dual, this corresponds to computing projections (*shadows*) of an intersection of halfspaces. More generally, we can obtain output-sensitive bounds for computing *skeletons in a halfspace intersection*, or with the known methods for ray shooting in a collection of hyperplanes [AM93], *skeletons in a hyperplane arrangement*; see [Ede87, Chapter 9]. With suitable data structures, this applies to arrangements of different objects as well, such as line segments in the plane.

4.6 The Prune-and-Divide Convex Hull Algorithm in Even Dimensions

In this section, we return to the convex hull problem. We show that Theorem 4.3.1 can be further improved in even dimensions. The method is an extension of the divide-and-conquer algorithm `DivideHull4d()` given in Chapter 3. To describe the extension, let us first recall the terminology and notation from Chapter 3 and particularly, Sections 3.2.3 and 3.2.4. It suffices to provide higher-dimensional analogues of Lemmas 3.2.4 and 3.2.5 from Section 3.2.4, since other parts of the algorithm work in any fixed dimension. As in Section 3.2.4, we may assume that Δ is a halfspace.

Recall that in the algorithm in Chapter 3, the facets $F(P)$ of the upper hull of a point set $P \subseteq E^d$ are constructed by recursively computing the primal restriction $F_S(P)$ to various simple regions S of P. To divide a simple region into smaller simple regions, $(d-1)$-dimensional upper hulls of the projection $\pi_\Delta(P) \subseteq E^{d-1}$ for certain halfspaces Δ are used.

The main difficulty that arises when $d > 4$ is that we do not have control on the size of these $(d-1)$-dimensional upper hulls, i.e., the number of facets in $F(\pi_\Delta(P))$. For $d \leq 4$, the size is $O(|V(P)|)$, which we can bound by $O(|F_S(P)|)$ if $P{\downarrow} \subseteq S$ by Lemma 3.2.1(d). For $d > 4$, we can only bound the size by $O(|F(P)|)$ and this can be much larger than the actual output size $|F_S(P)|$, since there may exist facets in $F(P)$ with vertical projection outside S even if $P{\downarrow} \subseteq S$. To overcome this difficulty, we do not construct all the hull facets in $F(\pi_\Delta(P))$ but instead apply the results in Sections 4.1 and 4.2 to perform queries on $F(\pi_\Delta(P))$.

Lemma 4.6.1 *Suppose that $d > 4$. Then the restricted point set $P_{|\operatorname{int} S_\Delta}$ can be computed in $O((|P| + (|P|\,|V(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P|)$ time.*

Proof: As in the proof of Lemma 3.2.4, we can test whether $p{\downarrow} \in \operatorname{int} S_\Delta$ for a given point $p \in P$ by finding a facet $\pi_\Delta(r)$ of $F(\pi_\Delta(P))$ with $\pi_\Delta(r){\downarrow} \in \pi_\Delta(r){\downarrow}$ and then determining which side of $r{\downarrow}$ the point $p{\downarrow}$ lies on. This facet can be found by performing a linear programming query on a polytope $\mathcal{P}$ defined by $|P|$ dual halfspaces in E^{d-1} corresponding to the $|P|$ points in the projected point set $\pi_\Delta(P) \subseteq E^{d-1}$. As we need $|P|$ queries for each point $p \in P$, the cost is $O(|P|^{2-2/\lceil d/2\rceil} \log^{O(1)} |P|)$ by Corollary 4.1.4.

To further reduce the running time, we first identify the vertices $\pi_\Delta(v)$ of $V(\pi_\Delta(P))$ ($v \in V(P)$); using the extrema algorithm from Section 4.3, this requires at most $O((|P| + (|P|\,|V(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |V(P)|)$ time by Theorem 4.3.2 (remember that the point set $\pi_\Delta(P)$ is just of dimension $d-1$). Because $F(\pi_\Delta(P)) = F(V(\pi_\Delta(P)))$, we only need the halfspaces corresponding to these $\leq |V(P)|$ vertices to define our polytope $\mathcal{P}$. The cost of the $|P|$ queries is then no more than $O((|P| + (|P|\,|V(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P|)$, according to Corollary 4.1.4. □

Lemma 4.6.2 *Suppose that $d > 4$. Given ∂S for a simple region S of P, one can construct $\partial S'$ for a new simple region S' of P with $\operatorname{int} S' = \operatorname{int} S \cap \operatorname{int} S_\Delta$ in $O((|P| + |F_S(P)| + (|P|\,|F_S(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P|)$ time.*

Proof: As in the proof of Lemma 3.2.5, it suffices to compute all the boundary components. Then we can select which boundary components contribute to the boundary $\partial S'$—that is, which boundary components correspond to a subregion inside S_Δ—by the techniques of the previous lemma. This requires at most $O(|F_S(P)|)$ linear programming queries on $|P|$ halfspaces in E^{d-1}, and thus can be done within $O((|P| + |F_S(P)| + (|P|\,|F_S(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P|)$ time.

The boundary computation is again based on depth-first search, but now we cannot generate all the ridges of the boundary B_Δ in advance as we cannot afford to compute all the facets in $F(\pi_\Delta(P))$; rather, a ridge is generated when it is needed.

As before, we store the ridges in ∂S (but not B_Δ) and their $(d-3)$-subfaces in a dictionary and sort these ridges around each $(d-3)$-face σ. Suppose B is a boundary component and we are given a ridge r in B (with its orientation). We first describe how we can generate the $d-1$ ridges that are adjacent to r in B.

These adjacent ridges can be classified into two types: (i) ones that are in ∂S, and (ii) ones that are in the boundary B_Δ. We deal with the adjacent ridges that are in ∂S first. A ridge adjacent to r must share a common $(d-3)$-subface, so let us consider one $(d-3)$-subface σ of r. Look up the dictionary to see if σ is a $(d-3)$-face of ∂S. If so, by performing a binary search on the list of ridges that σ is incident on, we can identify a candidate ridge in ∂S that r may be adjacent to. Repeating this procedure for every $(d-3)$-subface σ of r, we get all the ridges in ∂S that r may be adjacent to.

Next we deal with the adjacent ridges that are in the boundary B_Δ. Again we consider a $(d-3)$-subface σ of r. Determine whether $\pi_\Delta(\sigma)$ is a ridge of $R(\pi_\Delta(P))$; this test can be reduced to a linear programming query on a $(d-1)$-dimensional polytope defined by $|P|$ dual halfspaces. If the test is true, the linear programming query can be used to get a facet $\pi_\Delta(r')$ of $F(\pi_\Delta(P))$ $(r' \in R(P))$ that $\pi_\Delta(\sigma)$ is incident on. Then a ray shooting query in dual space can be used to find (if it exists) the other facet $\pi_\Delta(r'')$ of $F(\pi_\Delta(P))$ $(r'' \in R(P))$ that $\pi_\Delta(\sigma)$ is also incident on. These ridges r' and r'' in B_Δ give two possible candidates for the adjacent ridges of r. We repeat this procedure for every $(d-3)$-subface σ of r.

We now have a list of possible candidates for ridges that may be adjacent to r in B. By performing some local tests, we can deduce which of these ridges are actually adjacent. As in the proof of Lemma 3.2.5, we can then trace the complete boundary component B by visiting the adjacent ridges recursively in a depth-first manner. To compute all boundary components, we ensure that all ridges in ∂S are visited.

To evaluate total time needed by this computation, observe that the number of ridges visited by the depth-first search is $O(|F_S(P)|)$ by Lemma 3.2.1(e) since we only generate ridges r with $r{\downarrow} \subseteq S$. The work is then dominated by $O(|F_S(P)|)$ linear programming and ray shooting queries in E^{d-1}, which, by Corollaries 4.1.4 and 4.2.4, require no more than $O((|P| + |F_S(P)| + (|P|\,|F_S(P)|)^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P|)$ time. $\square$

To get a convex hull algorithm in E^d, we just have to replace Lemmas 3.2.4 and 3.2.5 by Lemmas 4.6.1 and Lemma 4.6.2 in the algorithm outline for `DivideHull4d()` from Section 3.2.5. We follow the same notation from Section 3.2.6 to analyze the running time.

By Lemmas 4.6.1 and 4.6.2, the non-recursive part of the algorithm now takes $O((|P_\nu| + |F_{S_\nu}(P_\nu)| + (|P_\nu|\,(|V(P_\nu) + F_{S_\nu}(P_\nu)|))^{1-1/\lceil d/2\rceil}) \log^{O(1)} |P_\nu|) = O((|P_\nu| + f_\nu + (|P_\nu|\,f_\nu)^{1-1/\lceil d/2\rceil}) \log^{O(1)} n)$ time at node ν, if we recall that $|V(P_\nu)| = |V(P_{\nu\,|S_\nu})| \leq d\,|F_{S_\nu}(P_\nu)|$ by Lemma 3.2.1(d).

To sum this cost, we recall, from Section 3.2.6, that $\sum_\nu n_\nu < n$ and $\sum_\nu f_\nu = f$ over every level of the recursion tree. Since $|P_\nu| = |P_{\nu\,|\operatorname{int} S_\nu}| + |P_{\nu\,|\partial S_\nu}| \leq n_\nu + df_\nu$ by Lemma 3.2.1(g), we also have $\sum_\nu |P_\nu| \leq n + df$. Using Hölder's inequality, we obtain the following cost-per-level bound, ignoring polylogarithmic factors:

$$\begin{aligned}
\sum_\nu \left(|P_\nu| + f_\nu + (|P_\nu|\,f_\nu)^{1-1/\lceil d/2\rceil}\right) &= O\left(n + f + n^{1-2/\lceil d/2\rceil} \sum_\nu \left(|P_\nu|^{1/\lceil d/2\rceil} f_\nu^{1-1/\lceil d/2\rceil}\right)\right) \\
&= O(n + f \;+\; n^{1-2/\lceil d/2\rceil}(n+f)^{1/\lceil d/2\rceil} f^{1-1/\lceil d/2\rceil}) \\
&= O(n + (nf)^{1-1/\lceil d/2\rceil} + fn^{1-2/\lceil d/2\rceil}).
\end{aligned}$$

Summing over all $O(\log_{1/\alpha} n)$ levels, we get $O((n + (nf)^{1-1/\lceil d/2\rceil} + fn^{1-2/\lceil d/2\rceil}) \log^{O(1)} n)$ as the total running time.

Theorem 4.6.3 *Let $d > 4$ be a constant. The convex hull of an n-point set in E^d can be computed in $O((n + (nf)^{1-1/\lceil d/2 \rceil} + fn^{1-2/\lceil d/2 \rceil}) \log^{O(1)} n)$ time.*

For odd dimensions d, this method is of no use since it is more complicated and not better than the convex hull algorithm in Section 4.3; however, for even dimensions d, we do obtain improvement over previous results for a certain range of f. For example, when $f = \Theta(n)$, our previous method (and also Matoušek's method [Mat93]) achieves $O(n^{2-2/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$ time; the method here achieves $O(n^{2-2/\lceil d/2 \rceil} \log^{O(1)} n)$ time. In general, if the output size is linear or sublinear, Theorem 4.6.3 provides the best upper bound currently known for the convex hull problem, ignoring polylogarithmic factors.

4.7 Appendix: Using Randomization in Linear Programming Queries

In this appendix, we describe how randomization can be used to improve the results from Section 4.2.

Consider the following linear programming problem: given k preprocessed convex polytopes $\Pi_1, \ldots, \Pi_k \subseteq E^d$, each defined by m halfspaces containing the origin o, compute the vertex v of $\Pi_1 \cap \ldots \cap \Pi_k$ that maximizes $\xi \cdot v$ for a given $\xi \in E^d$. Suppose that linear-space static structures (Structure 1) from Table 4.1 are used to store these polytopes. As is demonstrated in the proof of Lemma 4.2.3, a direct application of Matoušek's multi-dimensional parametric search technique would yield an $O(k\, m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m \log^d k)$-time solution. Here we describe how the $\log^d k$ factor can be eliminated by using Sharir and Welzl's randomized algorithm for generalized linear programming [SW92] (which is based on Seidel's linear programming algorithm [Sei91]). This in turn improves the query time in Lemma 4.2.3.

We first observe that the problem of finding an extremum in a non-empty intersection of k convex objects in E^d belongs to the class of *LP-type problems of combinatorial* *dimension* d as defined by Sharir and Welzl [SW92]. Sharir and Welzl presented a simple randomized algorithm for solving LP-type problems of fixed combinatorial dimension that requires an expected number of $O(k)$ *primitive operations*. The primitive operations, in our case, are: (i) to test whether a given point lies inside one of the objects (*violation tests*), and (ii) to find the extremum in an intersection of $d+1$ of the objects (*basis computations*).

For our application, the objects are polytopes. A violation test is simply a membership query and costs $O(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)$ time. A basis computation involves solving our linear programming problem on $d+1$ of the polytopes; since the number of polytopes is now constant, we can apply our previous method, via Matoušek's parametric search, to solve this problem in $O(m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)$ time. Because $O(k)$ violation tests and basis computations are expected to be performed by Sharir and Welzl's algorithm (the expected number of basis computations is actually only $O(\log^d k)$ [Wel91]), we obtain a randomized $O(k\, m^{1-1/\lfloor d/2 \rfloor} \log^{O(1)} m)$-time solution to our linear programming problem on k polytopes.

The above method carries through if the polytopes are stored in linear-space dynamic structures (Structure 1′ from Table 4.1); we simply replace the $\log^{O(1)} m$ factors with m^{ε}. With slightly more effort, we can even remove the assumption that the polytopes all contain the origin; the method can detect whether k preprocessed polytopes have a common intersection.

Note that in the two-dimensional case both violation tests and basis computations can be performed in $O(\log m)$ time. Thus, Sharir and Welzl's algorithm achieves expected $O(k \log m)$ time, which is an improvement over the previous $O(k \log^2 m)$ algorithm by Reichling [Rei88a], as used in our proof of Lemma 4.2.1. It is also interesting to compare the techniques here with those used in the previous deterministic and randomized methods by Reichling [Rei88b] and Eppstein [Epp91] for the three-dimensional problem.

The (expected) query time in Lemma 4.2.3 can now be improved to $O((n/m^{\lfloor d/2 \rfloor}) \log^{O(1)} m)$ since it uses $k = \lceil n/m \rceil$. As a consequence, the $O(n \log\log n)$ term in Corollary 4.2.4 can be eliminated; the same is true for the dynamic case (Lemma 4.2.5(i)), which leads to corresponding improvements in Theorem 4.5.2.

dimension	running time	references
2	$O(n \log h)$	[KS86]
3	$O(n \log h)$	[CM95]
4	$O(nf)$	[CK70, Swa85]
	$O(n^{4/3} \log^{O(1)} n + f \log n)$	[Mat93, Sei86]
$d > 4$	$O(nf)$	[CK70, Swa85]
	$O(n^{2-2/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n + f \log n)$	[Mat93, Sei86]
2	$O(n \log h)$	Theorems 2.1.2 and 2.2.1
3	$O(n \log h)$	Theorem 2.2.2
4	$O((n + f) \log^2 f)$	Theorem 3.2.6
	$O(n \log f + (nf)^{2/3} \log^{O(1)} n)$	Theorem 4.3.1
$d > 4$	$O(n \log f + (nf)^{1-1/(\lfloor d/2 \rfloor+1)} \log^{O(1)} n)$	Theorem 4.3.1
	$O((n + (nf)^{1-1/\lceil d/2 \rceil} + fn^{1-2/\lceil d/2 \rceil}) \log^{O(1)} n)$	Theorem 4.6.3

Table 5.2: Summary of output-sensitive results for the convex hull problem. The top half of the table shows the best running time achieved by previous algorithms, and the second half shows the running time achieved by the algorithms of this thesis.

In trying to answer the question of whether $\Theta(n \log f + f)$ is the true complexity of the convex hull problem, we may consider some special cases, for instance, when output size is very small or very large. For sufficiently small values of f, we do indeed attain optimal $O(n \log f)$ time in arbitrary dimension (Theorem 4.3.1). A theoretically interesting question is whether $O(f)$ time can be achieved for sufficiently large values of f. The method by Seidel [Sei86] achieved $O(f \log n)$ time for f super-quadratic.

A result not listed in Table 5.2 that is worth mentioning here is that we can construct the h-vertex convex hull in E^d in $O(n \log^{O(1)} h + h^{\lfloor d/2 \rfloor})$ time (Corollary 4.3.3). Thus, there is a near-optimal convex hull algorithm if the output polytope is of the worst kind, i.e., if $f = \Theta(h^{\lfloor d/2 \rfloor})$. Unfortunately, polytopes of this kind are rare in practice, so this result is more of a theoretical nature. (Unlike Chazelle's method [Cha93b], this is at least sensitive to the number of hull vertices.)

Note that all the algorithms listed in Table 5.2 are deterministic. Can randomization lead to faster or simpler output-sensitive algorithms for higher-dimensional convex hulls?

Most of our new algorithms in Table 5.2, especially the four- and higher-dimensional algorithms, assume general position in the input. To what extent can this assumption be removed?

This thesis has explored new techniques for solving the convex hull problem. Along the way, we have also touched on applications to many related problems. For instance, we have obtained output-sensitive algorithms for computing Voronoi diagrams (Theorems 3.3.1 and 3.3.2), extreme points (Theorem 4.3.2), convex layers (Corollary 4.4.2), k-levels (Theorems 4.5.1 and 4.5.3), and even envelopes of line segments (Theorem 2.3.1). We have also obtained improved algorithms for non-constructive problems such as computing depths in a point set (Theorem 4.4.1) and linear programming with few violated constraints (Theorem 4.5.2). Theorem 4.4.1 on depths provides a nice example on how output-sensitive algorithms can be used to produce better worst-case algorithms for certain problems not requiring output-sensitivity.

Our discussion on these related problems raises more interesting questions. For one, is our extrema algorithm (Theorem 4.3.2) close to optimal? A weaker question is: in terms of n alone, is $\Theta(n^{2-2/(\lfloor d/2\rfloor+1)})$ near the true complexity of the extreme point problem? We do not know of even an $\Omega(n^{1+\varepsilon})$ lower bound on the problem. Obtaining nontrivial lower bounds for problems of this type may be a topic of future research. Another open question is: can the depth problem be solved in close to $O(n^{2-2/(\lfloor d/2\rfloor+1)})$ time, like the extreme point problem? One can also consider improving our results on convex layers and k-levels; for example, can we construct four-dimensional k-levels in $O((n+f)^{1+\varepsilon})$ time, as we can for convex hulls? Our result on linear programming with violations may also be improved; for example, can one get an $O(n \log k)$-time algorithm for all $0 \leq k < n$ in the planar (feasible) case?

Bibliography

[AM91] P. K. Agarwal and J. Matoušek. Dynamic half-space reporting and its applications. Technical Report CS-1991-43, Duke University, 1991.

[AM93] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22:764–806, 1993. Also in *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 517–526, 1992.

[AGR94] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 683–694, 1994.

[Aur91] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[Ben83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual Symposium on Theory of Computing*, pages 80–86, 1983.

[BK+78] J. L. Bentley, H. T. Kung, M. Schkolnick and C. D. Thompson. On the average number of maxima in a set of vectors. *Journal of the Association for Computing Machinery*, 25:536–543, 1978.

[BS80] J. L. Bentley and J. Saxe. Decomposable searching problems I: static-to-dynamic transformations. *Journal of Algorithms*, 1:301–358, 1980.

[BS78] J. L. Bentley and M. I. Shamos. Divide-and-conquer for linear expected time. *Information Processing Letters*, 7:87–91, 1978.

[Bro80] K. Q. Brown. *Geometric transforms for fast geometric algorithms.* PhD thesis, Carnegie–Mellon University, Pittsburg, Penn., 1980.

[Cha95a] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. Submitted to *Discrete & Computational Geometry.*

[Cha95b] T. M. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. Submitted to *Discrete & Computational Geometry.* Also in *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, pages 10–19, 1995.

[CSY95a] T. M. Chan, J. Snoeyink, and C.-K. Yap. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 282–291, 1995.

[CSY95b] T. M. Chan, J. Snoeyink, and C.-K. Yap. Primal dividing and dual pruning: output-sensitive construction of 4-d polytopes and 3-d Voronoi diagrams. Submitted to *Discrete & Computational Geometry.*

[CK70] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the Association for Computing Machinery*, 17:78–86, 1970.

[Cha85] B. Chazelle. An optimal algorithm for computing convex layers. *IEEE Transactions on Information Theory*, IT-31:509–517, 1985.

[Cha92] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM Journal on Computing*, 21:671–696, 1992.

[Cha93a] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9:145–158, 1993.

[Cha93b] B. Chazelle. An optimal convex hull algorithm for point sets in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993.

[CD87] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *Journal of the Association for Computing Machinery*, 34:1–27, 1987.

[CE+91] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. In *Proceedings of the 18th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, volume 510, Springer-Verlag, pages 661–673, 1991.

[CF90] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[CGL85] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CM95] B. Chazelle and J. Matoušek. Derandomizing an output-sensitive convex hull algorithm in three dimensions. *Computational Geometry: Theory and Applications*, 5:27–32, 1995.

[Cla87] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, 2:195–222, 1987.

[Cla94] K. L. Clarkson. More output-sensitive geometric algorithms. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 695–702, 1994.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4:387–421, 1989.

[CS+89] R. Cole, J. Salowe, W. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM Journal on Computing*, 18:792–810, 1989.

[CSY87] R. Cole, M. Sharir, and C.-K. Yap. On k-hulls and related problems. *SIAM Journal on Computing*, 16:61–77, 1987.

[DK83] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical Computer Science*, 27:241–253, 1983.

[DK90] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra: a unified approach. In *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, volume 443, Springer-Verlag, pages 440–413, 1990.

[Dwy91] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6:343–367, 1991.

[Dye84] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM Journal on Computing*, 13(1):31–45, 1984.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15:317–340, 1986.

[EM90] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[ES86] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete & Computational Geometry*, 1:25–44, 1986.

[ES91] H. Edelsbrunner and W. Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM Journal on Computing*, 20:259–277, 1991.

[EW86] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM Journal on Computing*, 15:271–284, 1986.

[EC92] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proceedings of the Eighth Annual ACM Symposium on Computational Geometry*, pages 74–82, 1992.

[Epp91] D. Eppstein. Dynamic three-dimensional linear programming. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 488–494, 1991.

[ERvK93] H. Everett, J.-M. Robert, and M. van Kreveld. An optimal algorithm for computing ($\leq k$)-levels, with applications to separation and transversal problems. In *Proceedings of the Ninth Annual ACM Symposium on Computational Geometry*, pages 38–46, 1993.

[GT91] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 523–533, 1991.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[Grü67] B. Grünbaum. *Convex Polytopes.* John Wiley & Sons, London, 1967.

[GH+87] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[HS86] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6:151–177, 1986.

[HW87] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete & Computational Geometry*, 2:127–151, 1987.

[Her89] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 33:169–174, 1989.

[HeS93] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: shoot a ray, take a walk. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 54–63, 1993.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.

[KS86] D. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986.

[Mat91a] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 505–511, 1991.

[Mat91b] J. Matoušek. Cutting hyperplane arrangements. *Discrete & Computational Geometry*, 6:385–406, 1991.

[Mat91c] J. Matoušek. Efficient partition trees. In *Proceedings of the Seventh Annual ACM Symposium on Computational Geometry*, pages 1–9, 1991.

[Mat92] J. Matoušek. Reporting points in halfspaces. *Computational Geometry: Theory and Applications*, 2:169–186, 1992.

[Mat93] J. Matoušek. Linear optimization queries. *Journal of Algorithms*, 14:432–448, 1993. Also with O. Schwarzkopf in *Proceedings of the Eighth Annual ACM Symposium on Computational Geometry*, pages 16–25, 1992.

[Mat94] J. Matoušek. On geometric optimization with few violated constraints. In *Proceedings of the Tenth Annual ACM Symposium on Computational Geometry*, pages 312–321, 1994.

[MS93] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete & Computational Geometry*, 10(2):215–232, 1993.

[McM70] P. McMullen. The maximal number of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[MS71] P. McMullen and G. C. Shephard. *Convex Polytopes and the Upper Bound Conjecture*. Cambridge University Press, 1971.

[Meg83a] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the Association for Computing Machinery*, 30:852–865, 1983.

[Meg83b] N. Megiddo. Linear time algorithm for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12:759–776, 1983.

[Meg84] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the Association for Computing Machinery*, 31:114–127, 1984.

[Mul90] K. Mulmuley. Output sensitive construction of levels and Voronoi diagrams in R^d of order 1 to k. In *Proceedings of the 22rd Annual ACM Symposium on Theory of Computing*, pages 322–330, 1990.

[Mul91] K. Mulmuley. On levels in arrangements and Voronoi diagrams. *Discrete & Computational Geometry*, 6:307–338, 1991.

[Mul93] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1993.

[OBS92] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, England, 1992.

[O'Ro94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[OC+82] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19:384–391, 1982.

[OSS95] T. Ottmann, S. Schuierer, and S. Soundaralakshmi. Enumerating extreme points in higher dimensions. To appear in *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science*, 1995.

[OvL81] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[Pre90] F. P. Preparata. Planar point location revisited. *International Journal of Foundations of Computer Science*, 1(1):71–86, 1990.

[PH77] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the Association for Computing Machinery*, 20:87–93, 1977.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[Ray70] H. Raynaud. Sur l'enveloppe convexe des nuages des points aléatoires dans R^n, I. *Journal of Applied Probability*, 7:35–48, 1970.

[Rei88a] M. Reichling. On the detection of a common intersection of k convex objects in the plane. *Information Processing Letters*, 29:25–29, 1988.

[Rei88b] M. Reichling. On the detection of a common intersection of k convex polyhedra. In *Computational Geometry and its Applications*, Lecture Notes in Computer Science, volume 333, Springer-Verlag, pages 180–186, 1988.

[RS63] A. Rényi and R. Sulanke. Über die konvexe Hülle von n zufällig gerwähten Punkten I. *Z. Wahrsch. Verw. Gebiete*, 2:75–84, 1963.

[RW94] T. Roos and P. Widmayer. k-Violation linear programming. *Information Processing Letters*, 52:109–114, 1994.

[ST86] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the Association for Computing Machinery*, 29:669–679, 1986.

[Sei81] R. Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical Report 81-14, Department of Computer Science, University of British Columbia, Vancouver, B.C., 1981.

[Sei86] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 404–413, 1986.

[Sei91] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[SW92] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proceedings of the Ninth Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, volume 577, Springer-Verlag, pages 569–579, 1992.

[Swa85] G. F. Swart. Finding the convex hull facet by facet. *Journal of Algorithms*, 6:17–48, 1985.

[Wel91] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer (Ed.), *New Results and New Trends in Computer Science*, Lecture Notes in Computer Science, volume 555, Springer-Verlag, pages 359–370, 1991.

[Wen94] R. Wenger. Randomized quick hull. Manuscript, 1994.

[Yao81] A. C. Yao. A lower bound to finding convex hulls. *Journal of the Association for Computing Machinery*, 28:780–787, 1981.