Efficient Construction of Convex Hulls from Spherical Distributions in Higher Dimensions

by

Wm. M. Stewart

B.Sc., University of New Brunswick, 1983

M.Sc., University of New Brunswick, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in the Faculty of Computer Science

This thesis is accepted.

_____

Dean of Graduate Studies and Research

THE UNIVERSITY OF NEW BRUNSWICK

February, 1992

Abstract

The incremental construction of convex hulls is investigated. Given a set of n points S in $\Re^d$ in general position, a data structure is given and an algorithm implemented to add a point to an existing convex hull in $\Theta(d^3)$ expected time in the number of new facets created. The storage to store a convex hull is $\Theta(dm)$ in the number of facets m. When the set of points S can be preprocessed the convex hull can be constructed directly. For the online problem a Two-Flat algorithm is described to identify interior points or a visible facet as required by walking a circuit of facets in the convex hull defined by an intersection with a 2-flat. The performance of the algorithms are investigated for the construction of cyclic polytopes, and convex hulls of sets distributed on (distribution A) and in (distribution B) d-dimensional spheres. Cyclic polytopes can be constructed in $\Theta(d^3m)$ time, or $\Theta(d^2m)$ time with an increase in storage by a factor of d, where the Two-Flat algorithm has negligible cost even in low dimensions. The preprocessing construction of convex hulls from distribution A creates less than $2dm$ facets in total for the sets studied. For convex hulls from distribution A the Two-Flat algorithm becomes more efficient as the dimension rises, and for $d \geq 5$ increases the total processing by less than a factor of two; an argument is given to support the conjecture that the Two-Flat algorithm has worst-case

complexity O $(m^{1/(d-1)})$ for distributions A and B. The online construction of convex hulls from distribution A can then be done in $\Theta(d^4 m)$ time, or $\Theta(d^3 m)$ time by increasing the storage by a factor of d. Convex hulls in three through ten dimensions are constructed with 24 megabytes of available space for storage of the data structure, limiting n to approximately 350, 120, 60, and 45 in dimensions 7, 8, 9, and 10 respectively. For sets of points distributed in spheres, interior points should be removed in a preprocessing step whenever the number of vertices of the final convex hull is signifigantly less than n. Generation of Delauney triangulations of some real-world surveying data by Brown's method is also investigated.

Contents

List of Figures

List of Tables

Acknowledgments

I will be eternally grateful to Joseph Horton, to whom I owe so much and can never thank enough. Thanks also to William Knight for answering so many complicated questions in simple words, to Lev Goldfarb for introducing me to Morris Kline and quantum mechanics, to Uday Gujar for encouraging me at the very beginning, to Dana Wasson for his generous support, to Rod Cooper for many stimulating political arguments, to Jane Fritz for being a unique role model, to Arlene Stocek just for being a wonderful human being, to Kirby Ward for invaluable assistance with software, and to UNB Computing Services for their endless patience. Thank you all from the bottom of my heart.

Chapter 1

Introduction

The convex hull problem is a central one in the area of computational geometry, with application to stock cutting and allocation, intersection of spheres, robotics and graphics, generation of voronoi diagrams, and other problems. In many cases, particularly in the plane, generation of the convex hull is a necessary preprocessing step for some other algorithm.

The convex hull of a finite set S of n points in d dimensions is an intuitively natural structure with three equivalent definitions commonly used to define it.

For example, the convex hull of S may be defined to be the intersection of all convex sets containing S, or, alternatively, as the smallest convex set containing S. More formally, the following definition suits our purposes best.

Definition 1 *The convex hull* $\mathrm{conv}\,(S)$ *of a set of* n *points* $S = \{p_1, \ldots, p_n\}$ *in* $\Re^d$ *is the closed convex set of all convex combinations of the points of* S. *That is:*

$$\mathrm{conv}\,(S) = \{\alpha_1 p_1 + \cdots + \alpha_n p_n \mid \alpha_1 + \cdots + \alpha_n = 1,\ \alpha_i \geq 0\}$$

In one dimension the convex hull of a set of points S from the real line is simply the closed segment $[\min(S), \max(S)]$. The solution to the problem is $\Omega(n)$ and an optimal algorithm is just a linear search.

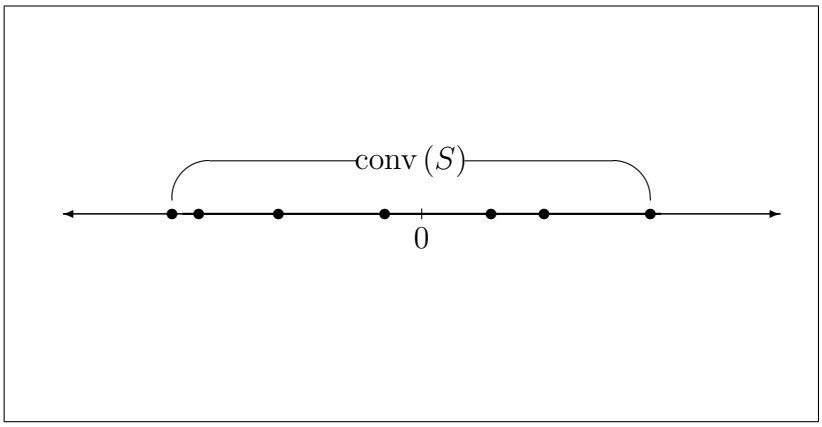

Figure 1.1: Convex Hull in the Real Line

In the plane $\mathrm{conv}\,(S)$ is the smallest convex polygon containing the set of points S, and the solution is usually represented by an ordered set of the polygon's edges.

A lower bound to the time complexity of the planar problem is found by noting that sorting is transformable to the planar convex hull problem.

This can be done by placing the points of the set S on a parabola, so that a solution to the convex hull problem produces an ordered set of edges that correspond to an ordering of the set S by first coordinate. The planar convex hull problem therefore has a lower bound of $\Omega(n \log n)$.

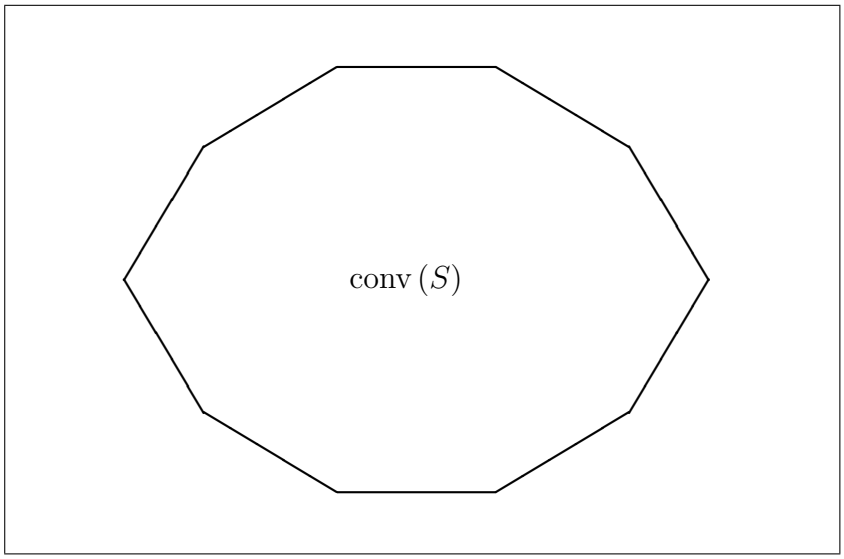

Figure 1.2: Convex Hull in the Plane

On the other hand, even though all known planar convex hull algorithms return an ordered set of edges, it is worth noting that the $\Omega(n \log n)$ bound can also be derived using a result concerning maximal vectors [17]. The lower bound then doesn't rely on the condition that the set of convex hull edges found by an algorithm must be ordered, and also applies to any algorithm, even algorithms which simply return an unordered set of edges or just a list of the extreme points of S.

The Graham Scan [12] was the first optimal convex hull algorithm developed for two dimensions. Graham's algorithm sorts the points by angle around any point interior to the set S (say, the centroid of any three non-collinear points of S), and then constructs the convex hull point by point in the order defined by the sort. For each addition, only a constant amount of backtracking is required overall to delete edges previously added and superceded by the current point. The complexity of the initial sort is therefore the major operation of the algorithm, and the Graham Scan has optimal complexity of $\Theta(n \log n)$.

A number of subsequent algorithms have been developed for the two dimensional problem, each with computational advantages for particular distributions of points. For example, Jarvis's algorithm [14] has expected complexity $\mathrm{O}\left(n^{4/3}\right)$ for points distributed uniformly in a circle (note that $n^{4/3} < n \log n$ up to approximately 1000). Eddy's algorithm [11] is $\mathrm{O}\left(n\right)$ for points distributed uniformly in a convex polygon, uniformly in a circle, and normally in the plane. And the Akl-Toussaint algorithm [2] is $\mathrm{O}\left(n\right)$ for points distributed uniformly in a square.

In three dimensions, $\mathrm{conv}\left(S\right)$ is a three dimensional polytope or bounded polyhedron with polygonal facets. Geodesic domes (and their interior) are typical examples of convex hulls in $\Re^3$. Whether represented by points, edges,

or facets, a three dimensional hull can be stored in only $O(n)$ space, but the $\Omega(n \log n)$ lower bound on the time complexity is inherited from the plane.

Only one known algorithm, Preparata and Hong's Divide and Conquer algorithm, achieves optimality in three dimensions [17]. This algorithm first divides the given set of points S into two disjoint sets S_1 and S_2. This can be done by any of several methods in linear time, such as splitting the set by first coordinate. The algorithm then constructs the convex hulls $\text{conv}(S_1)$ and $\text{conv}(S_2)$ recursively. The final step merges $\text{conv}(S_1) \cup \text{conv}(S_2)$ in linear time. This merge is done by constructing the 'cylindrical' set of facets joining the two disjoint hulls with a linear algorithm. A classic example of the divide-and-conquer paradigm, this algorithm has $O(n \log n)$ complexity since the number of facets of the three dimensional convex hull is no more than $O(n)$. A much simpler version of Preparata and Hong's algorithm is also given for the planar case.

Unfortunately, the more complex facial data structure of higher dimensional convex hulls has so far prevented the development of a general convex hull merge procedure for any dimension greater than three. Although it may provide some promise, the divide and conquer paradigm has not yet been applied to the general case, and a general procedure independent of dimension must be expected to be diffucult.

Basic assumptions, definitions, and theorems required for the development of the subsequent material are given next in chapter 2. In chapter 3 algorithms for the higher dimensional convex hull problem are discussed. In chapter 4 the data structure used to represent a convex hull is described together with an incremental update algorithm to add a point to an existing convex hull. In chapter 5 two cases are discussed, one for the case when the set is known and can be preprocessed, and one for the online case when the points arrive from an external source and a Two-Flat algorithm is used to identify interior vertices or a visible facet as required. In chapter 6 we discuss implementation issues, the behavior of the algorithms in practice for the construction of cyclic polytopes and convex hulls of points distributed in and on spheres, expected complexities, space considerations, an application, and related matters. Conclusions are given in chapter 7. Appendix A includes the listing of the software in PL/1. Appendix B contains tables of data.

Chapter 2

Definitions and Theorems

In this chapter the basic assumptions, definitions, and theory required in the following chapters is described.

Consider a set of points $S = \{p_1, \ldots, p_n\}$ in $\Re^d$, for d greater than 2. We make the following assumption

Assumption The set S is in general position, i.e., no $k+2$ points appear in any $k - flat$.

The vertices of the convex hull, also called the extreme points, are the fundamental building blocks of convex hulls.

Definition 2 *An* extreme point *p of S is not the convex combination of any*

other two or more other points of S. The set of all extreme points is denoted $\mathrm{ext}(S)$.

Convex hulls are equivalent to closed convex polytopes of finite point sets.

Theorem 1 (Brondsted, theorem 7.2) *The polytope $P = \mathrm{conv}\,(S)$ is the convex polytope of the extreme points of S.*

Convex hulls and convex polytopes, or just hull and polytope, will therefore be referred to interchangably throughout the following.

The facial structure of convex polytopes has been much investigated, and forms the basic theory of many important areas such as linear programming.

Definition 3 *A* face *F of a convex polytope P is a convex subset such that, for any two distinct points y and z in P, with $]y, z[\cap F$ not empty, then $[y, z]$ lies wholly in F.*

That is, for any two points in the polytope, if the interior of the line segment connecting those two points intersects a face F, then the entire line segment must lie in F. This definition implies that the faces of a convex polytope are closed convex sets, and that they lie on the boundary of the hull.

A face F is called a k-face when $\dim(F) = k$. By convention the (-1)-face is the empty set and the d-face is the entire polytope. These two faces

are called the *improper* faces of P.

The *proper* faces of P are those of dimension 0 through $d-1$, or points through *facets.* For example, a convex hull in the plane has two types of proper faces—points and edges. A convex hull in three dimensions has three types of proper faces—points, edges, and triangular facets. In four dimensions a convex hull has four types of proper faces—points, edges, triangles, and tetrahedronal facets. And so on.

One property that makes convex hulls such natural structures is that each proper face of a hull is itself a lower dimensional convex hull.

Theorem 2 (Brondsted, theorem 7.3) *Every proper k-face F of a convex polytope P is itself a k-polytope, and* $\text{ext}(F) \subset \text{ext}(S)$.

Because the set of points is assumed to be in general position, every proper k-face is not only a k-polytope but also a k-simplex defined by $k+1$ points of the set S. A k-face F will therefore be represented by its $k+1$ extreme points $\text{ext}(F) \subset S$.

The faces of main interest for our purposes will be the three proper faces of dimension $(d-1)$, $(d-2)$, and $(d-3)$, which we call facets, subfacets, and subsubfacets respectively.

The following well known theorem regarding facets and subfacets forms the basis of Chand and Kapur's Giftwrapping algorithm, Jarvis's algorithm,

Preparata and Hong's divide and conquer algorithm, as well as the algorithms presented in this thesis, imposing a simple and useful structure on convex polytopes.

Theorem 3 *Each subfacet of a convex polytope is contained in exactly two* neighboring *facets.*

Since each facet of a d-polytope is a d-simplex, each has d subfacets and d neighboring facets, one neighbor for each subfacet. We are assured that the d neighbors are distinct as a consequence of simple convexity.

In higher dimensions it is difficult to give any consistent characterization of an 'average' convex hull independent of an underlying application, since the number of facets of a polytope varies very widely with d and n and depends heavily on the distribution of the given set of points S.

In the best case, the number of facets may be as little as $\mathrm{O}(1)$. On the other hand, when the points of S fall on a moment curve of the following form

$$(t, t^2, t^3, \ldots, t^d),\ t \in \Re$$

we call $\mathrm{conv}(S)$ a *cyclic polytope*, and encounter a particularly difficult distribution. Cyclic polytopes constitute the worst case convex hulls from the point of view of space, with $\mathrm{O}(n^{\lfloor d/2 \rfloor})$ facets [20]. Even neglecting the signifigant constant factors, it can be readily seen that cyclic polytopes cannot

be generated in practice for all but very small n and d. Fortunately, cyclic polytopes are considered to be special objects that are unlikely to arise in all but the most specialized applications.

However, the number of convex hull facets has been shown to be much lower than $\mathrm{O}\left(n^{\lfloor d/2 \rfloor}\right)$ for a wide range of so-called 'average' distributions of points, such as sets of points uniformly distributed in and on convex bodies [10]. For example, for sets of points distributed in or on spheres the number of facets of the convex hull is only $\mathrm{O}\left(n\right)$ for fixed d. These kinds of distributions are generally considered to be more likely to arise in actual applications, and therefore are of more practical interest than the extreme case typified by cyclic polytopes.

Even for these spherical distributions though the number of facets rises very quickly with the dimension d, in practice quickly exceeding the capacity of on-line storage even for relatively small problems, such as a hundred points in ten dimensions. It is this rapid growth in the size of the output for non-trivial distributions that makes the convex hull problem in higher dimensions particularly difficult. Some problems with relatively small input size, such as 500 points distributed on a sphere in $\Re^{100}$, must be considered practically insoluble in the foreseeable future simply because of the extreme size of the output.

Chapter 3

The Higher Dimensional Problem

In higher dimensions, four and greater, there are three common representations of convex hulls that may be appropriate for any given problem or application.

The first and simplest type of problem is the *vertex enumeration* problem. This problem concerns itself with finding just the vertices or extreme points $\text{ext}(S)$ of the convex hull, or, equivalently, eliminating all interior points.

A straightforward solution to the vertex enumeration problem can be found simply by n invocations of a linear programming algorithm, one invocation for each point of the set S. For example, to determine whether or not

the last point p_n is a vertex of the convex hull or an interior point, we can determine whether or not a feasible solution exists for the following linear programming problem, where each point p_i of S is considered to be a column vector, and the coefficients α_i lie in $\Re$.

$$\begin{aligned} \text{Subject to} \quad & \alpha_1 p_1 + \cdots + \alpha_{n-1} p_{n-1} = p_n \\ & \alpha_1 + \cdots + \alpha_{n-1} = 1 \\ & \alpha_i \geq 0 \end{aligned}$$

The objective function, and whether it is to be maximized or minimized, is actually irrelevant, since all that is required is a feasible solution. If a feasible solution can be found, then it will be an expression of the point p_n as a convex combination of some other $d+1$ points of S, which immediately implies that it must be interior to $\text{conv}\,(S)$. On the other hand, if no feasible solution to the problem can be found, then p_n is not the convex combination of any other $d+1$ points of the set S. Assuming that S is in general position, the point p_n must then be an extreme point of the set.

By repetition of this procedure for every point of the set S, placing each point on the right hand side in turn, the solution to the vertex enumeration problem in can be found in $\text{O}\,(n^2)$ time. Since the number of facets m of convex hulls of many distributions is only $\text{O}\,(n)$, one might conclude that an

$\mathrm{O}(n^2)$ solution to the vertex enumeration problem is not good, but it will be shown later that there is a wide range of n and d for some distributions for which it may be acceptable.

The convex hull problem is usually framed as a *facet enumeration* problem, where we are interested in finding just the facets of the hull—the proper faces of highest dimension, or in the most general way as the *facial graph* problem which finds the faces of the convex hull of all dimensions with relationships of inclusion, also called the Hasse diagram [20], that is, where each k-dimensional face of the convex hull has a pointer to each of the $(k+1)$-dimensional faces in which it is contained. The facial graph is particularly useful for applications in which the question must be asked: 'do these given $k+1$ points define a k-face of the convex hull $\mathrm{conv}(S)$?'.

Chand and Kapur's Giftwrapping algorithm [7] finds the facets of the convex hull in $\mathrm{O}(n\ m)$ time for fixed dimension, where m is the actual number of facets of the convex hull to be found, and was the first convex hull algorithm for the higher dimensional case. The algorithm starts with one facet of the hull, which can be easily found by a recursive invocation of the giftwrapping algorithm itself, and then the set of subfacets of this first face are added to an initially empty list of subfacets. For each subfacet of this list until empty the giftwrapping procedure is then employed. For example,

for a set in general position in $\Re^3$, all facets are triangles. The algorithm wraps a halfplane around each edge in the subfacet list until it hits a point in the set, thereby identifying a new facet, and possibly adding new edges to the subfacet list. Similarly, in four dimensions, the algorithm wraps around triangular subfacets to find new tetrahedronal facets. The algorithm repeats the procedure until the subfacet list is empty and the entire convex hull has been found.

Seidel developed two facet finding algorithms. His 'shelling' algorithm [22] finds the facets of the hull in $\mathrm{O}\left(n^2 + m \log n\right)$ time for fixed dimension by a traversal of a shelling line through the convex hull. For example, in three dimensions we might define a shelling line that passes through the interior of the polytope and intersects two facets on opposite sides of the hull. The set of facets can then be enumerated in shelling order, i.e., in the order they become visible on an outward traversal of the shelling line on one side and in the order they disappear on an inward traversal of the shelling line on the other side. The algorithm maintains the boundary of the current set of visible facets in a manner that allows the identification of the next facet at logarithmic cost, with no more than n iterations of the simplex algorithm to handle special cases.

Seidel other facet finding algorithm is randomized and incremental with

expected time complexity $O(n^{\lfloor d/2 \rfloor})$ [23]. The addition of each new point necessitates the use of a linear programming algorithm as a preprocessing step, after which the facet graph can be updated in a straightforward manner (with a radix sort to maintain the data structure) in time $O(n + N)$ where n is the number of vertices and N is the number of new facets added.

Clarkson and Shor's facet finding algorithm uses Las Vegas random sampling methods in the dual space to generate the convex hull in $O(m \log m)$ expected time for fixed dimension [8]. A bipartite conflict graph is maintained to speed up the algorithm, maintaining the relationship between each halfplane and the edges of the hull not in the halfplane. Clarkson has also developed an online version of the algorithm, which maintains a set of simplices interior to the hull to enable the identification of interior points by traversal of a line through the interior of the polytope.

Avis's $O(dnm)$ Pivoting algorithm for vertex enumeration of arrangments and polyhedra can be used with modification for the facet enumeration problem [1]. In the dual space, the algorithm starts at an optimum vertex and reverses Bland's pivoting rule to lexicographically backtrack through the rest of the vertices. The algorithm does not require extra space, and is output-sensitive. This algorithm may be particularly good, but the only known implementation in Mathematica makes some space and time trade-offs

that compromize its efficiency.

Kallay's incremental 'beneath-beyond' algorithm [20] finds the facial graph, is online, and has complexity $\mathrm{O}\left(n^{\lfloor d/2 \rfloor + 1}\right)$. By identifying vertices as *reflexive* and *non-reflexive*, the algorithm traverses the facial graph from the bottom up, separating the faces of the hull into three classes and processing as appropriate: faces that belong to the new hull unmodified, faces that must be deleted, and faces that must make a dimensional step-up to include the new point added to the hull.

Seidel also developed an incremental algorithm for the facial graph problem which operates in the dual space [21], with complexity $\Theta(n^{\lfloor d/2 \rfloor})$ for even d and $\mathrm{O}\left(n^{\lfloor d/2 \rfloor + 1}\right)$ for odd d. By separating faces of the dual facial graph into six colours, each point (dual plane) updates the dual facial graph by a similar dimensional step-up as used by Kallay's algorithm.

Chapter 4

The Incremental Paradigm

In this chapter we describe a data structure and update algorithm to add a new point to a convex polytope.

The incremental paradigm was first applied to the construction of non-planar convex hulls by Dijkstra in *The Discipline of Programming* [9] for three dimensions. Dijkstra's algorithm represents the convex hull as a planar graph by its set of edges and stores a range of additional information to facilitate the ordering of the set of edges cyclically around each facet. By a traversal of this data structure the old edges are deleted and new edges are added as required for the addition of each new point of the set. Dijkstra does not give the complexity of his algorthm, but it can be reasonably estimated as $\mathrm{O}\,(n^2)$.

The incremental paradigm for convex hull construction may appear to contain a built-in inefficiency, in that it uses less information than is available. For example, at each stage of construction the convex hull might be updated to include a point that is not present in the final hull and will be superceded by later additions. There may be reason to believe that a global approach, taking all points of the set into consideration at once, may be able to utilize extra information to perform better.

At the same time, the incremental paradigm recommends itself for three reasons:

1. It is a conceptually simple approach that can be easily understood and promises to result in a practical algorithm that can be readily implemented.

2. An incremental update procedure is the required operation for a solution to the online problem, when the entire set of points is not known beforehand and an efficent update is the only practical approach.

3. For many distributions the number of vertices of the hull is $\mathrm{O}(n)$, so an efficient incremental construction may well be the best choice.

4.1 Data Structure

The data structure that will be used to represent a convex polytope is just its set of facets P. Each facet F in P will be represented by its d extreme points from S. Rather than defining a new notation like $\mathrm{Vert}(F)$, in order to simplify the following discussion we refer to a facet F both as a concept and as a set of d points from S, as will be clear from the context.

The following information will also be stored for each facet F—its d subfacets, pointers to each of its d neighbors, and the equation of its halfspace equation, which will be denoted as follows:

- $\mathrm{Subf}_1^F, \ldots, \mathrm{Subf}_d^F$, where each subfacet is represented by a unique combination of $d-1$ of the d points of F.

- $\mathrm{Neig}_1^F, \ldots, \mathrm{Neig}_d^F$, pointers ordered so that Subf_i^F is shared with Neig_i^F

$$F \cap \mathrm{Neig}_i^F = \mathrm{Subf}_i^F \ .$$

- Half^F , the inequality of the halfspace bounded by the hyperplane containing F, and oriented so that $S \subset \mathrm{Half}^F$.

The subfacets can be generated in a straightforward manner at minimal cost. The pointers to neighboring facets will be set by procedures to be described in the following sections. The up-front 'cost' of creating a facet

F in P is therefore bounded by the cost of creating the halfspace equation, which we assume can be done in $\Theta(d^3)$ expected time from the d points of F by finding the solution to a $d \times d$ system of simultaneous linear equations. That is, if $q_1, \ldots q_d$ are column vectors representing the points of F, and c is a column vector of any non-zero constant c, we solve

$$(q_1 \cdots q_d)(x) = (c)$$

for x and and then represent Half^F in the usual manner by the pair (x, c). Proper orientation of the halfspace is then ensured, by checking the sign of the computation $< x, p > - c$ for any point p interior to the convex hull so far constructed, and if the sign is negative then setting $\text{Half}^F = (-x, -c)$. Given the halfspace equation of a facet, testing whether or not a point p is in Half^F then takes only $\Theta(d)$ time by checking for a positive sign in $< x, p > - c$. This will be a common operation throughout the following.

There are of course faster methods for generation of the halfspace equation, with complexities fractionally less than $\text{O}\,(d^3)$, but they have not been implemented for the following reasons: the complexity of the implementation for general dimensions; the possibility of loss of precision; and it will be shown later that the cost of setting the neighbor pointers puts an $\Omega(d^3)$ lower bound on the creation of each facet in any case.

The subfacets Subf_i^F will not be actually stored, and will instead be de-

fined implicitly by the following convention: The points of a facet

$$F = \{q_1, \ldots, q_d\}$$

will be stored in ascending order by their index in S, and then

$$\text{Subf}_i^F = F \setminus \{q_i\}$$

This convention was implemented and facilitates several $\Theta(d)$ operations later.

Since the remaining data structure (pointers to extreme points, pointers to neighbors, and halfspace equation) can all be stored in $\Theta(d)$ space, this convention allows us to reduce the storage per facet from $\Theta(d^2)$ to $\Theta(d)$, and the total storage for a convex hull from $\Theta(d^2 m)$ to $\Theta(dm)$, where m is the total number of facets of the convex hull.

4.2 Updating a Convex Hull

In this section an algorithm is described to implement the operation $P \Leftarrow P \cup \{p\}$ for arbitrary dimension greater than two where P is a convex hull represented by the data structure given above and p is a new point outside P. The case when p is inside P will be addressed in chapter 5. One facet K for which p is not in Half^K is assumed to be known, and it is shown how to find K in chapter 5.

The update of a convex hull can be visualized as follows. Imagine that the facets of the convex hull P are opaque, and that our eye is positioned at the new point p somewhere outside P. In the most general terms, the addition of p to P requires the following operations:

1. Delete all facets of P visible from p.

2. Add a set of new facets to P, each containing the new point p.

3. Set all links between neighboring facets to maintain the data structure.

This describes the essential steps of an update procedure in any dimension. That is, the existing polytope P is split by the update into two sets: the set of visible facets for which p is not in Half^F that must be deleted because they do not include the new point p, and the set of nonvisible facets for which p is in Half^F that will remain because they do include the new point p.

The set of new facets that must be added to P all contain the new point p, and so can be thought of as a sort of multi-dimensional cap to cover the set of visible facets.

Finally, to complete the update and maintain our data structure, we must ensure that all adjacencies between neighboring facets of the new hull are recorded.

With respect to the data structure of P, the update may be broken down into the following six activities:

1. Identify the set of visible facets V

$$V = \{ F \mid F \in P, p \notin \text{Half}^F \}$$

2. Identify the boundary between the set of visible facets V and the set of nonvisible facets, which will be a set of subfacets called the horizon H. Each subfacet f in the horizon is shared by two neighboring facets of P, one visible facet and one nonvisible facet.

3. Add to P a cap C of new facets with p at the apex, where each facet in C is defined by the union of p and a subfacet of the horizon:

$$C = \{ F \mid F = f \cup \{p\}, f \in H \}$$

4. Set all links between nonvisible facets of P on the horizon H and neighboring facets in C.

5. Set all links between neighboring facets of the cap C itself.

6. Delete the visible set V.

Therefore, the update procedure considers four sets of faces

- the set of visible facets V

- the set of subfacets in H
- the set of nonvisible facets on H
- the cap of new facets C

and two types of links between neighboring facets that must be set to maintain the data structure of $P \cup \{p\}$:

- between facets of C and nonvisible facets on the horizon H;
- between facets of the cap C itself.

To efficiently implement the update procedure it is convenient to combine several steps of the update into one operation, simplifying the operation by dividing it into two phases. In phase one of the procedure steps 1 through 4 above are performed in one pass: identify the visible set V, identify the horizon H, create the cap C, and link C to P. In phase two the set of visible facets V is traversed to perform step 5—interlinking neighboring facets in the cap C itself. When phase two is complete the visible set V is no longer required and may be deleted, thereby completing the update.

4.2.1 Phase One

Recall that we assume that one visible facet K is known before the update begins. Given K, the primary feature of our data structure—links between

the neighboring facets—can be used to traverse the entire visible set V and identify it, also identify the horizon H, create the cap C, and link C to the nonvisible facets of P on the horizon in one operation.

This procedure is performed recursively as a breadth-first traversal of V as follows. First, put K in V. Then invoke Traverse(K) as described below.

Traverse(K):

- Scan the neighboring facets of K:
  - When a neighboring facet G is found to be visible and is not already in V, then put G in V and recursively Traverse(G).
  - When a neighboring facet G is found to be nonvisible then the subfacet $f = K \cap G$ is a subfacet of the horizon. A new cap facet $T = f \cup \{p\}$ can immediately be created, and T and G can be interlinked across their common subfacet f.

This procedure is described more completely by algorithm Traverse in figure 4.1. At the conclusion of the traversal, steps 1 through 4 of the update procedure are complete, the visible set V has been identified, and the cap C has been created and linked to the nonvisible facets on the horizon H.

Recall the convention that the points defining a facet shall be stored in ascending order by index in S and that $F \cap \mathrm{Neig}_i^F = \mathrm{Subf}_i^F$. This enables an

```
Algorithm Traverse ( F )
  for i = 1, ..., d do
    G ← Neig_i^F
    if G ∉ V then
      if p ∉ Half^G then          { G is visible from p. }
        V ⇐ V ∪ G
        Traverse ( G )            { Recurse. }
      else
                { Create T = {p} ∪ (F ∩ G). }
        T ← Create_Facet (p, F, G )
        Neig_d^T ← G
        j ← Find_Common_Subfacet ( G, T )
        Neig_j^G ← T
        C ⇐ C ∪ T
      end if
    end if
  end for
end Traverse
```

Figure 4.1: Phase 1 – Identify V and H, Create C, Link C to P.

$\Theta(d)$ time for several basic operations by merge-type procedures, including Create_Facet (excluding creation of the halfspace equation) and interlinking neighboring facets T and G.

For example, for a given value of i during the operation of Traverse we know that $F \cap G = \text{Subf}_i^F$, and when T is created it is $T = \text{Subf}_i^F \cup \{p\}$. Therefore if $F = (q_1, \ldots, q_d)$ then

$$T = (q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_d, p)$$

since p is the latest addition to the set and has the highest index in S so far (note that the same analysis will apply to both the preprocessing and online cases), an $\Theta(d)$ operation.

To interlink facets T and G one can first set $\text{Neig}_d^T = G$ directly, since by the above definition of facet T it is clear that $T \cap G = \text{Subf}_d^T$. The value of j such that $\text{Neig}_j^G = F$ can then be found by an $\Theta(d)$ merge-type operation of the points of G and T by the routine Find_Common_Facet of figure 4.2.

The complexity of phase 1 as implemented by the Traverse algorithm can be calculated by dividing the processing into two types: encounters with visible facets, and encounters with nonvisible facets on the horizon which cause the creation of a cap facet:

- The first encounter with each visible facet F costs $\Theta(d)$ for the test $p \notin \text{Half}^F$. There may be at most $d-1$ subsequent visits to F from

Algorithm Find_Common_Facet (F, G)

{ *Find* j *such that* $Subf_j^F$ *is shared by* F *and* G. }

$j \leftarrow 0$

$a \leftarrow 1$

$b \leftarrow 1$

do while $j = 0$

if $F(a) = G(b)$ then

$a \leftarrow a + 1$

$b \leftarrow b + 1$

else

if $F(a) < G(b)$ then

$a \leftarrow a + 1$

else

$j \leftarrow b$

end if

end if

if $a > d$ then $j \leftarrow d$

end

return (j)

end Find_Common_Subfacet

Figure 4.2: Finding a common subfacet

other neighboring facets during the traversal through V, and each such encounter will be only $\Theta(1)$ to determine that F has already been visited and placed in V. The complexity of this portion of phase one is therefore $\Theta(d\,|V\,|)$.

- Each encounter with a nonvisible facet F on the horizon costs $\Theta(d)$ for the test $p \in \text{Half}^F$. The subsequent creation of a new cap facet costs $\Theta(d^3)$. Linking the new cap facet to the nonvisible facet across their common subfacet is $\Theta(d)$. The complexity of this portion of phase one of the update is therefore $\Theta(d^3\,|C\,|)$.

The overall complexity of phase one is therefore $\Theta(d\,|V\,| \;+\; d^3\,|C\,|)$.

When phase one is complete it remains to interlink the cap facets with each other, performed by phase two.

4.2.2 Phase Two

The major remaining requirement to maintain the data structure is the setting of all required links between neighboring facets in the cap C itself.

This procedure must be efficient since there are many more links to be set between cap facets themselves in phase two than there were between the cap and the original convex hull P in phase one. Whereas there are exactly $|C\,|$ links to be set between C and P in phase one, there are $(d-1)\cdot|C\,|/2$

links to be set between neighboring facets of the cap in phase two.

Lemma 1 *Exactly $d-1$ of the neighbors of every cap facet are also cap facets.*

Proof: Consider any cap facet T. Only one subfacet of T does not contain p, that is, subfacet $f = T \setminus \{p\}$, and f is shared with a nonvisible facet on the horizon. The other $d-1$ subfacets of T do contain p, and only cap facets contain p, so $d-1$ neighboring facets of T must be in the cap. □

A straightforward procedure to interlink neighboring cap facets which was implemented and uses a kind of 'scrabble' sort procedure, described as follows.

First, store all the subfacets of the facets of the cap corresponding to unset neighbor links in a two dimensional array, with one subfacet in each row, where each subfacet is defined by the indices of its $d-1$ extreme points from S in ascending order. In an associated pointer array, carry pointers for each subfacet to the cap facet it belongs to. Each subfacet in the array will appear twice, once for each of the two cap facets that have it in common.

Sort the array in row major order, sorting the pointer array at the same time. At the conclusion of the sort the array will be a list of duplicate rows, 1 and 2, 3 and 4, 5 and 6, etc., where each pair of rows is a pair of identical

subfacets shared by two neighboring cap facets. All that remains is to scan the pointer array and interlink each successive pair of cap facets across their common subfacet.

The above procedure is simple and can be implemented to run very quickly with an efficient sorting utility in $\Theta(|C| \cdot \log |C|)$ expected time, but has a major drawback that mitigates against its use and led to the search for an alternative approach, i.e., the procedure requires $\Theta(d^2 |C|)$ extra storage at $\Theta(d)$ storage for $d-1$ subfacets of each cap facet. Since the major constraint on the construction of many convex hulls is space this scrabble sort procedure was dropped in favor of the following procedure.

A better approach that avoids the requirement of extra storage sets each required link between neighboring cap facets by walking a circuit of neighboring facets in P that all contain the same subsubfacet. First, note that such a circuit exists.

Lemma 2 *There is a circuit of neighboring facets in a d-polytope P around each subsubfacet of P.*

Proof: Consider any subsubfacet ϕ contained in any facet F. Now, ϕ is defined by $d-2$ points so in one lower dimension is a *subfacet* of the $(d-1)$-polytope F and therefore contained in exactly two facets f and g of F by theorem 3. But f and g are really subfacets in P, so ϕ is contained in

exactly two subfacets of F. This implies that exactly two of F's neighbors also contain ϕ. The same argument holds for each of these two neighbors (exactly two of their neighbors contain ϕ), so the path through neighboring facets around ϕ is one-dimensional. The path must be closed because the number of facets of P is finite. $\square$

Figure 4.3 shows two examples of a subsubfacet circuit, one around the point of a portion of a 3-polytope, and one around the edge of a portion of a 4-polytope.

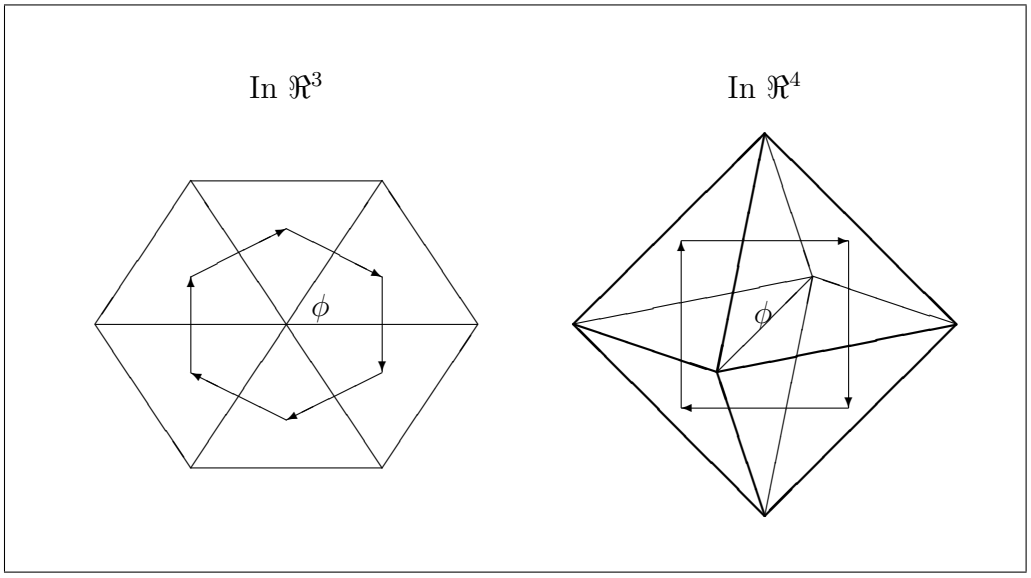

Figure 4.3: Subsubfacet Circuits in $\Re^3$ and $\Re^4$

Note that no claim is made that the circuit around a subsubfacet is unique or that other disjoint circuits may not exist, although convexity does imply

it. It suffices for our purposes that at least one such circuit exists.

These circuits can be used to interlink the neighboring facets of the cap as follows. For each facet T of the cap with $\text{Neig}_j^T = \emptyset$, find the neighboring cap facet U that shares Subf_j^T by walking a subsubfacet circuit as follows—refer to figure 4.4 for an example in $\Re^3$:

1. Now, T is a facet of the cap, so $T = f \cup \{p\}$ where f is a subfacet of the horizon shared by a visible facet F and nonvisible facet G. Now p is contained in Subf_j^T so we can write $\text{Subf}_j^T = \phi \cup \{p\}$ where ϕ is a subsubfacet in f. By inclusion ϕ must be contained in F and G, so F and G are two neighboring facets on a circuit around ϕ. Start walking this circuit around ϕ starting at F and moving through the visible set V.

2. Since the circuit crosses the horizon at f it must cross it again. Stop when the first nonvisible facet Z is found and let the previous (visible) facet be Y. Now Y and Z share a subfacet y in the horizon, so there must be a cap facet $U = y \cup \{p\}$ linked to Z across y.

3. Now ϕ is contained in Y and Z, so ϕ is contained in y and U. But p is in U as well, so $\phi \cup \{p\}$ is in U. But $\text{Subf}_j^T = \phi \cup \{p\}$, so U shares Subf_j^T with T. Interlink T and U across Subf_j^T.

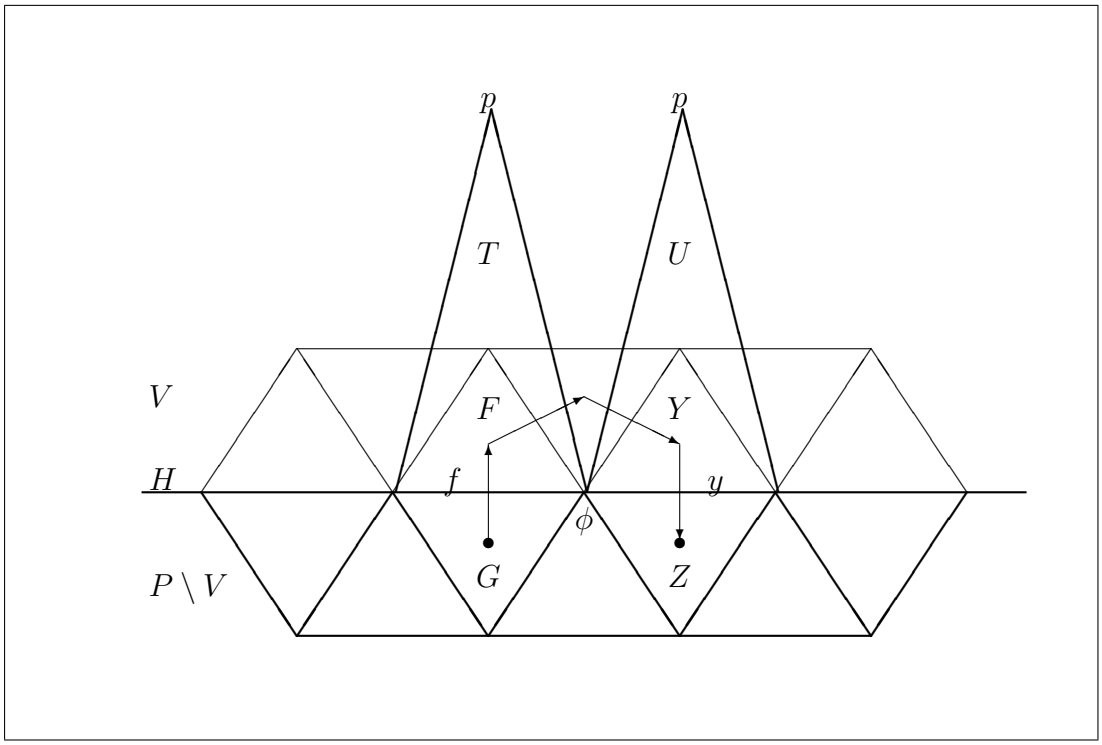

Figure 4.4: Interlinking Cap Facets in $\Re^3$

This procedure is performed by algorithm Interlink of figure 4.5. The two pointers Visible_Neighbor^T and Nonvisible_Neighbor^T are two extra pointers attached to each facet that are set by Create_Facet when T is created. Next_Facet finds the next facet in the circuit by an $\Theta(d)$ merge-type operation on the points of F, G, and ϕ (see procedure Rotate in appendix A). The facets T and U can then be interlinked by $\mathrm{O}(d)$ invocations of Find_Common_Facet.

There are $d-1$ neighbor links to be set for each cap facet in phase two, and two links are set by each invokation of Interlink, so the procedure must be performed exactly $|C|(d-1)/2$ times in all to completely interlink all neighboring facets of the cap.

The overall complexity of phase two of the update can be found by dividing the processing performed by the Interlink algorithm into two types: processing charged to facets of the visible set V during traversals around subsubfacet circuits, and processing charged to facets of the cap C at the end of each traversal, as follows:

- Each traversal of a subsubfacet circuit is through the visible set V, and each step in the circuit takes $\Theta(d)$ time. Now, a subsubfacet is defined by $d-2$ points, so each facet contains $\binom{d}{d-2} = \Theta(d^2)$ distinct subsubfacets. Therefore, there can be at most $\mathrm{O}(d^3|V|)$ processing of facets

Algorithm Interlink (T, j)

$\phi \leftarrow \text{Subf}_j^T \setminus \{p\}$

$F \leftarrow \text{Visible\_Neighbor}^T$

$G \leftarrow \text{Nonvisible\_Neighbor}^T$

while $F \in V$ do { *Walk around* ϕ. }

$X \leftarrow$ Next_Facet (F, G, ϕ) { *An* $O(d)$ *operation.* }

$G \leftarrow F$

$F \leftarrow X$

end while

$l \leftarrow$ Find_Common_Facet(G, F)

$U \leftarrow \text{Neig}_l^F$

$\text{Neig}_j^T \leftarrow U$

$l \leftarrow$ Find_Common_Facet(T, U)

$\text{Neig}_l^U \leftarrow T$

end Interlink

Figure 4.5: Interlinking Neighboring Cap Facets.

of the set V by Interlink in phase two, since each facet/subsubfacet pair defines a unique circuit.

- The remaining processing performed during phase two is the actual interlinking of neighboring cap facets at the end of each traversal. Each pair of links costs $\Theta(d)$ to set by Find_Common_Facet, and $d-1$ links must be set for each facet in C, for an overall complexity of $\Theta(d^2|\,C\,|)$.

The overall complexity of phase two as performed by algorithm Interlink is therefore $\mathrm{O}\,(d^3\,|\,V\,|+d^2\,|\,C\,|)$. This may not appear to be a major improvement over the scrabble sort method, but we shall see that it works suprisingly well in practice, and it facilitates the following presentation of the overall complexity of the update algorithm without requiring extra space.

4.3 Overall Complexity

The complexity of phase one is

$$\Theta(d\,|\,V\,|\;+\;d^3\,|\,C\,|).$$

The complexity of phase two is

$$\mathrm{O}\,(d^3\,|\,V\,|+d^2\,|\,C\,|)$$

and the cost of deleting the visible set V at the conclusion of the update is minor. The overall complexity for all processing performed during an update

Algorithm Update (P, p, K)

$V \Leftarrow \emptyset$

$C \Leftarrow \emptyset$

Traverse (K) { *Phase 1.* }

for each $T \in C$ and $\text{Neig}_i^T = \emptyset$ { *Phase 2.* }

Interlink (T, i)

end for

$P \Leftarrow P \setminus V$ { *Clean-up.* }

end Update

Figure 4.6: Update P with point p.

is therefore

$$\mathrm{O}\left(d^3\,|V\,| + d^3\,|C\,|\right)$$

Fortunately, the $d^3|V\,|$ term can be amortized away by the following argument.

The original cost to create each facet F of the visible set is already assumed to be $\Theta(d^3)$ to create the halfspace equation Half^F, and the entire visible set V is deleted at the end of the update. The one-time $\mathrm{O}\left(d^3\right)$ cost per facet of V incurred during the update can therefore be charged to its original creation, thereby amortizing this term away and leaving an overall update complexity of $\Theta(d^3|C\,|)$ in the size of the cap C.

This amortization is possible because of the assumption that creation of a halfspace equation is an $\Theta(d^3)$ operation, even though more sophisticated techniques of solving systems of linear equations are available with fractionally better performance. For our purposes these faster methods are not useful, at least with regard to improving the complexity of the update, since the $\mathrm{O}(d^3|V|)$ cost incurred during phase 2 depends on the complexity of the subsubfacet traversals and would not be reduced by a faster procedure of calculating halfspace equations.

However, it will be shown later that for some distributions of points the expected complexity of subsubfacet traversals is less than $\mathrm{O}(d^3|V|)$, and that a space/time trade-off can then reduce the expected complexity of the update for these sets to $\Theta(d^2|C|)$.

Chapter 5

Preprocessing and Online Constructions

In this chapter two methods are described to construct convex hulls with the data structure and update algorithm of the last chapter.

The first method is appropriate for the case when the entire set of points S is known beforehand and can be preprocessed with a lexicographic sort. This appraoch was first used by Seidel for his dual-space incremental algorithm.

The second method can be used for the online problem when the points arrive one by one from an external source and the convex hull must be quickly generated from the existing convex hull and each new point as it arrives. For this problem a Two-Flat algorithm is described to traverse a circuit of

neighboring facets in the hull in order to identify interior vertices, and to supply a first visible facet K to start the update when the new point is an exterior point.

5.1 The Preprocessing Case

When the set of points S is entirely known and can be preprocessed, the convex hull of S can be constructed almost directly by the successive invocation of the update algorithm for each point of S in turn.

The procedure starts with an initial simplicial convex hull P which can be easily generated from the first $d+1$ points of the set S. We are gaurenteed that this first convex hull will have positive volume and is therefore nondegenerate, since no $d+1$ points of S lie in the same hyperplane by our assumption of general position; in other words, absolutely any $d+1$ points of S will form a simplex of dimension d. The procedure is straightforward, amounting to no more than the construction of a d-simplex with links to neighboring facets set appropriately, in $\Theta(d^4)$ time (see Create_Simplex in appendix A).

Given the first simplicial convex hull P, the complete convex hull can be built by adding the remaining $n-(d+1)$ points to P, one by one, with the update algorithm of the last chapter. But first the set must be pre-

processed to facilitate the subsequent construction. Fortunately, this step is not complex, and can be carried out just by sorting the set S lexicographically. This can be done quite efficiently by any of a number of library sorts, but we chose a particularly fast implementation of Sedgewick's quicksort [19]. The expected complexity is therefore $\mathrm{O}(n \log dn)$ with a very small constant, greatly dominated by the remainder of the algorithm, and can be considered to be practically free for all but the most trivial problems.

This preprocessing step is required to satisfy the two conditions of an update:

1. Each new point p to be added to P is known to be exterior to the polytope, so that the update is actually required; and

2. A first visible facet K is known to start the breadth-first search of V in phase one of the update.

The lexicographic sort of S can be shown to ensure that each new point to be added to the hull is outside the convex hull so far constructed, i.e., point p_i is gaurenteed to not be in $\mathrm{conv}(p_1, \ldots, p_{i-1})$ for every i, by the following observation. When the set of points S is sorted in lexicographic order a hyperplane can always be found that separates $p_i = (x_{i1}, \ldots, x_{id})$ from the set of points $\{p_1, \ldots, p_{i-1}\}$, thereby guarenteeing that p_i is not interior to $\mathrm{conv}(p_1, \ldots, p_{i-1})$. Assuming that no two points share the first

coordinate, this separating hyperplane may be directly given as the one with first coordinate equal to x_{i1}. In the case that up to at most $d-1$ other points do share first coordinate with p_i, then the hyperplane can always be perturbed slightly to achieve the same result. Therefore, after the sort, no point lies inside the convex hull of the points preceeding it.

This preprocessing step also satisfies the second condition, ensuring that a first visible facet K from the existing convex hull P can be found to begin phase one of the update. This can be seen to be satisfied by considering that at least one facet of the cap C created during the update $P \Leftarrow P \cup \{p_{i-1}\}$ must be visible with respect to the current point p_i. That is, the line segment $\overline{p_{i-1}\, p_i}$ must intersect $\text{conv}\,(p_1, \ldots, p_{i-1})$ in only one point due to their lexicographic ordering, namely p_{i-1}, so p_{i-1} must be visible from p_i. Convexity then implies that no vertex of a convex hull can be visible from some other point in space without at least one facet containing that vertex also being visible, so it follows that at least one facet of P containing p_{i-1} is also visible from p_i.

But the facets of P containing the point p_{i-1} are exactly those facets of the cap C created during the previous update $P \Leftarrow P \cup \{p_{i-1}\}$ (as also noted by Seidel [23]). For any given update $P \Leftarrow P \cup \{p_{i-1}\}$, the first visible facet K can then be identified in either one of two essentially equivalent ways:

- Check each facet of the cap as it is created for visibililty with respect

to the next point of the set, for $i = d+2, \ldots, n-1$, or

- Keep the cap C from the previous update to be scanned for a visible facet as the first step of the next update.

As a matter of clarity of presentation the second approach is preferred, but for efficiency of implementation the first approach was used. In either case, the identification of the first visible facet K is at most an $\mathrm{O}(d|C|)$ operation in the size of the previous cap C. This cost may be charged against and subsumed by the original $\mathrm{O}(d^3|C|)$ cost to create the cap C in the first place, thereby amortizing the cost away.

The preprocessing construction is shown in figure 5.1. Before the first update of the point p_{d+2} the facets of the initial simplicial polytope are assigned to the cap C, thereby ensuring that a first visible facet K for the first update can be easily found by checking all $d+1$ facets of the initial simplex.

The remaining points of the set can then be simply added to P one by one and the cap C passed forward each time to provide the first visible facet for the next update, until every point has been added and the final convex hull $P = \mathrm{conv}(S)$ is completely generated.

Algorithm Preprocessing_Construction (S, d, n, P)

Sort (S) { *In lexicographic order.* }

$P \Leftarrow \text{conv}\,(p_1, \ldots, p_{d+1})$

$C \Leftarrow P$

for $i = d + 2, \ldots, n$ do

$F \leftarrow$ Scan_Cap (C) { *Find a facet visible from* p_i. }

$V \leftarrow \emptyset$ { *Re-initialize.* }

$C \leftarrow \emptyset$

Update (P, p_i, F)

end for

end Convex

Figure 5.1: Preprocessing Construction

5.2 The Online Case

The online problem is by its nature essentially a real-time problem, where the points of the set S arrive one at a time from an external source and we wish to maintain the convex hull of the points so far received.

With a sufficiently fast algorithm, abundant resources, and sufficiently large interarrival rate, it might well be possible to reconstruct the convex hull from scratch after the arrival of each new point, thereby providing an acceptable 'pseudo-online' solution that solves a series of apparently unrelated convex hull problems of sizes $1, 2, \ldots, n$. For any combination of algorithm and resources, however, there will be a potential case where for a sufficiently large number of points n and sufficiently short interarrival rate this procedure will be insufficient, that is, too slow. It is the always possible existence of such a case that provides the rationale for the requirement of a true online algorithm, an algorithm that can update the existing convex hull quickly without complete preprocessing.

We assume that at least $d + 1$ points of the incoming set S are known from which an initial nondegenerate simplicial convex hull P of dimension d can be created in the same manner used to construct an initial hull for the preprocessing construction by Create_Simplex, after which any finite number of new points arrive in arbitrary order.

Before beginning, the two assumptions upon which the update algorithm relies must be satisfied, given below in a slightly different form:

1. Each new point p that arrives can be identified as interior to P and discarded, in which case the update isn't required;

2. If p is an exterior point then at least one visible facet K can be found in acceptable time to begin the update.

In other words, when p is interior to P, which can certainly occur when the points arrive online and may lie anywhere in space, then the event needs to be identified to avoid the update, and when p is exterior to P a first visible facet K is required to begin the search of the visible set V in phase one.

```
Algorithm Online_Update ( P, p)
  F ← Two_Flat ( P, p )          {Find first visible facet.}
  if F ≠ ∅ then                  { p is exterior to P. }
     Update ( P, p, F )          { Update P. }
  end if
end Online_Update
```

Figure 5.2: The Online Update

The algorithm used to satisfy the first condition will also satisfy the sec-

ond condition.

With regard to the second condition, it is particularly important to be able to find the first visible facet K without having to process a significant proportion of the facets of the existing convex hull P, an approach that would be very costly with a complexity that would quickly approach the cost of reconstructing the convex hull from scratch, thereby invalidating the rationale for having an online procedure. Therefore an efficient method of finding the first visible facet K is required that avoids an examination of all facets of the existing hull.

The method of solving this problem used in the last section for the pre-processing case is clearly not applicable here, since there is no guarentee that a facet of the cap created by the previous update will be visible from the new point under consideration. The new point may be able so see many facets of P, only one facet of P, or, if an interior point, no facets at all.

Given a new point p and a polytope P, both conditions can be satisfied by a Two-Flat algorithm which traverses a circuit of neighboring facets defined by the intersection of P and a 2-flat. The method was developed because it appeared to have potential for efficient implementation and performance, and is shown later to work well for dimensions greater than four.

First, note that the intersection of a 2-flat and a d-dimensional convex

hull P produces a circuit of neighboring facets.

Lemma 3 *The non-empty intersection of a 2-flat Q with the interior of a d-polytope P is a circuit of neighboring facets in P.*

Proof: Let F be any facet of P intersected by Q. Assuming nondegeneracy, the intersection of a 2-flat and a $(d-1)$-flat is a 1-flat, so the intersection of Q with F is a line segment and the two endpoints of this line segment lie in two of the subfacets of F. Therefore Q intersects exactly two neighbors of F. The same analysis holds for each of these two neighbors and their neighbors in turn, so the instersection of Q and P defines a one-dimensional path. Since the number of facets of P is finite the path is closed. In fact, the intersection is a convex polygon. □

A 2-flat circuit is particularly useful to satisfy the two update requirements when it is defined so that the 2-flat Q passes through both the newly arrived point p and some known facet F of the existing convex hull. Let us define Q in this manner and then traverse the resulting circuit of neighboring facets, beginning at the first facet F and proceeding in either direction, and examining each facet in the circuit when encountered for visibility with respect to the new point p. The traversal must then have one of two possible outcomes:

- The circuit completes, returning to the starting facet F without having found any facet in the circuit that is visible from p; or
- Some facet in the circuit is found to be visible from p.

In the first case we can conclude that the new point p lies entirely inside the convex hull P, and therefore may be discarded. This follows directly from the following observation: the two-dimensional polygon formed by the intersection of Q and P lies entirely inside P, and if all facets in the circuit are nonvisible from p then p is interior to the polygon.

In the second case the new point p is clearly outside the convex hull P since a facet of P has been found that is visible from p. The traversal may then be terminated and the visible facet, say K, passed back to the main algorithm to begin the search of the visible set.

This procedure is described more precisely by the following:

Two-Flat Algorithm:

1. Define a 2-flat Q passing through points p, q, and r, where p is the new point, q is the centroid of the first facet F in P, and r is any other point not on the line $\overline{pq}$. Walk the circuit defined by Q, starting at F, in either direction. Stop when either
   (a) A facet K in the circuit is found to be visible from p, in which case return K; or

(b) The walk returns to F, in which case return $\emptyset$.

The creation of the 2-flat Q and the visibility tests for each facet in the circuit are straightforward. The primary operation required for the implementation of the traversal is the movement from one facet to the next, which can be carried out in $\Theta(d^3)$ time as follows.

Say F is the current facet in the circuit. Then there are d candidate neighboring facets of F to be visited next. Exactly two of F's neighbors are also intersected by Q and one of these neighbors has just been visited before proceeding to F, so there are $d-1$ candidate neighbors of F to be visited next.

If the next facet in the circuit is facet G then F and G share a common subfacet with a non-empty intersection with Q. Unfortunately, no $\Theta(d)$ merge-type procedure can be employed here to find this subfacet. We must find G by examining each of the $d-1$ candidate subfacets of F to determine which one is intersected by Q.

The primary operation of the 2-flat circuit traversal is therefore a test to determine whether or not a subfacet has a non-empty intersection with a 2-flat, which can be carried out as follows. The 2-flat Q is defined by three points p, q, and r as described above. A given subfacet of F, say f, is defined by $d-1$ points. If Q intersects f then the intersection will occur in a point,

and there will be a convex combination of the points defining f that define that point and an affine combination of the points defining Q that define the same point. There will always be a point of intersection between aff(f) and Q. Therefore, by finding the point of intersection between aff(f) and Q and then checking to see if that point falls within f itself (that is, is not only an affine combination but also a convex combination) the test for intersection will be complete.

This test may be carried out by solving a system of simultaneous linear equations $A\,X = B$, where A and B are defined as follows. Without loss of generality, let the subfacet f be defined by the points $p_1, \ldots, p_{d-1}$, and let each of these points be a column vector represented by its coordinates. Let p, q, and r be the points defining the 2-flat Q, these points also being column vectors represented by their coordinates. The point of intersection between aff(f) and Q can then be found by solving the following $(d+2) \times (d+2)$ system of simultaneous linear equations for X*

*Thanks to Wm. Knight for the formulation of this SSLE.

$$\left|\begin{array}{ccc|ccc} p_1 & \cdots & p_{d-1} & -p & -q & -r \\ & & & & & \\ & & & & & \\ & & & & & \\ \hline 1 & \cdots & 1 & 0 & 0 & 0 \\ 0 & \cdots & 0 & -1 & -1 & -1 \end{array}\right| \quad X \quad = \quad \left|\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 1 \\ -1 \end{array}\right|$$

The first $d-1$ coordinates of X are the coefficients of the combination of the points of f defining the intersection between aff(f) and Q. The last three coordinates of X are the negative of the coefficients of the combination of the points of Q defining the same point of intersection. The first d rows of A and B assure that both combinations define the same point of intersection, that is, that when they are subtracted they leave the origin. The last two rows of A and B assure that both combinations are indeed affine.

The actual point of intersection is of little interest. After finding X it remains to determine whether or not the intersection point lies not only in aff(f) but in f itself, which may be done by determining whether or not the first $d-1$ coordinates of X are all positive; if so, then the 2-flat Q intersects f itself, if not, then it doesn't.

Finally note that checking to see if the last three coordinates of X are all positive is insufficient, since the point of intersection in Q is irrelevant to the

test.

The expected average complexity of each step in the 2-flat circuit is therefore $\Theta(d^4)$ for $\Theta(d)$ tests to make at $\Theta(d^3)$ per test. This complexity can be reduced by a factor of d, since each subfacet tested differs from the one before by only one point, and changing any single row or column of an SSLE and finding the new solution can be done in only $\Theta(d^2)$ expected time. Instead of recalculating a new solution to a new SSLE up to $d-1$ times, we can instead solve the first SSLE as usual and then generate each of the up to $d-2$ subsequent solutions by changing the appropriate column of the matrix A in $\Theta(d^2)$ time. This revision would reduce the average expected complexity of each step in the 2-flat circuit from $\Theta(d^4)$ to $\Theta(d^3)$. This revision was implemented but no significant savings were observed, at least for the sets studied, since the revised method had somewhat more overhead. In any case the length of each circuit will be shown later to be relatively short in higher dimensions with a minor cost for $d \geq 5$. The original $\Theta(d^4)$ method was therefore retained for its greater precision. Nevertheless, it is expected that the revised approach could be of significant value for very large n and sufficiently high d.

5.3 Building the Facial Graph

Finally, note that the facial graph $\mathcal{G}$ can be generated from the set of facets of a convex hull P in a straightforward manner in $\Theta(|\mathcal{G}| \log |\mathcal{G}|)$, as follows.

First, put the highest dimensional face P in the facial graph, then add the facets with pointers to P. Add the subfacets of each facet to $\mathcal{G}$ with pointers to the facet in which they are contained, unless the subfacet has already been added in which case only add the pointer. Repeat this process for each k-face for subsubfacets through edges, adding its $(k-1)$-faces to $\mathcal{G}$ with pointers to the k-face it is contained in, unless it is already in $\mathcal{G}$ in which case just add the pointer. Finally, point the empty set at each vertex.

The major operation at the consideration of each k-face f is therefore the determination of whether or not f has already been added to $\mathcal{G}$, and if so its position in $\mathcal{G}$, which can be done in $\Theta(\log |\mathcal{G}|)$ time by the usual methods.

Chapter 6

Results

In this chapter we discuss implementation issues, behavior of the algorithms for the generation of convex hulls of points from spherical distributions and cyclic polytopes, behavior of the Two-Flat algorithm and conjectured complexity, performance of the Interlink procedure, space considerations, an application, and related matters.

6.1 Implementation Issues

The algorithms were initially implemented in Fortran, since the bulk of the processing is mathematical. However, the language is unwieldy for involved programming, so the software was ported to PL/1 which supports recursion, pointer based variables, and various other programming advantages. APL

was considered but found unsuitable for obtaining timing results.

Although not as widely used as some other programming languages, PL/1 is exceptionately well supported with optimizing compilers, interactive debuggers, a large library of mathematical and array functions, and remains among the best of the third generation languages if you can pretend that there are only two numeric data types—fixed binary and float binary. Its only serious drawbacks would seem to be a fashionable fixation with semicolons and the lack of an endif statement. It has, for example, done at least one thing that many have lamented Fortran never did—learned from APL and introduced array operations, as in

```
a ( i, * ) = sum ( b ( j, * ) + c ( *, k ) )
```

This has the obvious advantages of making code much shorter, much clearer, and therefore more robust.

On the other hand, the unusually limited portability of PL/1 compilers makes the software difficult to use outside an IBM environment. This drawback has been partially alleviated since the preprocessing algorithm has been translated to the programming language C and linked with a three dimensional graphics package by Gupta [13]. We expect to translate the online algorithm into C as well, so that the software will be accessible from any package with access to C procedures to make it available on a wide range of

non-IBM systems.

Most of the runs were done on an IBM-3090. A few in higher dimensions were done on an IBM-9121 which was found to be between 23 and 25 percent faster for the algorithms presented here, and so the timings were adjusted accordingly. All timings are given in seconds. Vector processing was not used. All data shown is obtained from single runs—that is, they are not the result of several runs that have been averaged, thereby showing the stability of the algorithm and the data collected.

The generation of cyclic polytopes is investigated later in section 6.3, where it is shown that they present precision problems. More extensive investigation is made of two distributions we will call distribution A and B, where A is drawn from points uniformly distributed on the surface of a d-sphere centered around the origin and distribution B is drawn from a sets of points uniformly distributed inside the same d-sphere, as two representative distributions of convex hulls of sets in general position. Whenever we speak of spherical distributions in general we mean sets of points from any independent distribution of points around the center of a d-sphere—for example, normally distributed.

Each point of these sets was generated by first setting each coordinate equal to a normally distributed random variable centred at zero and then

normalizing the vector to length one. The vector of a point generated in this manner then has a uniformly random direction around the origin. For distribution A, the vector was then given a length of 100, to uniformly distribute the points on the surface of the sphere. For distribution B the vector was given a length of 100 and then multiplied by $u^{1/d}$, where u is a uniformly distributed variable between zero and one, to uniformly distribute the points throughout the interior of the sphere.

The accuracy of the implementation of the algorithms was tested as follows. First, known convex hulls in three and four dimensions were generated. Second, the convex hulls were found to be the same for the same set S whether generated by the preprocessing algorithm or the online algorithm. Third, the minimum angle between the normals of adjacent facets was calculated to assure that the software was actually returning convex objects.

6.2 Behavior in Practice

Before assessing the practical utility of the algorithm, what is the optimal possible performance? In theory, an optimal construction of a convex hull with m facets would be only $\Theta(dm)$, assuming a minimum cost of $\Omega(d)$ on the generation of each facet.

Since the worst case, cyclic polytopes, have a very large number of facets

$m = \mathrm{O}\,(n^{\lfloor d/2 \rfloor})$, we might expect convex hulls from other distributions to grow rapidly with the dimension as well. This is intuitively plausible, since it might be expected that there are more ways for points to be adjacent and connected to other points in the set in higher dimensions than in lower dimensions.

On the other hand, Dwyer [10] has noted that a surprisingly large number of 'average' distributions have only $m = \mathrm{O}\,(n)$, including the spherical distributions A and B and other uniform distributions in figure 6.1.

Uniform Distributions	m	Ref.
on sphere (A)	$\Theta(n)$	[6]
in sphere (B)	$\Theta(n^{(d-1)/(d+1)})$	[18]
in any polytope	$\Theta(\log^{d-1} n)$	[10]
in any convex body	$\mathrm{O}\,(n^{(d-1)/(d+1)})$	[3]

Figure 6.1: $\mathrm{O}\,(n)$ distributions.

This suggests that for fixed d an optimal convex hull algorithm for spherical distributions would be only $\Theta(n)$. However, for both distributions A and B the number of facets m is heavily influenced by the dimension d. Buchta [6] has shown that the expected number of facets m for distribution A is

$$\frac{2}{d}\,\gamma_{(d-1)^2}\,\gamma_{d-1}^{-(d-1)}\,n(1+o(1))$$

while, for distribution B, m is

$$\frac{2}{d}\,\gamma_{(d-1)^2}\,\gamma_{d-1}^{-(d-1)}\,v(1+o(1))$$

where v is the number of vertices of the hull and

$$\gamma_x \;=\; \frac{\Gamma(x+1)}{2^{x+1}((\Gamma(1+\frac{x}{2}))^2}$$

For both distributions m goes to infinity very quickly, if somewhat more slowly for distribution B than for distribution A. Some values are given in table 6.1 as calculated from Maple for the values of Buchta's equations.

d	$\frac{2}{d}\,\gamma_{(d-1)^2}\,\gamma_{d-1}^{-(d-1)}$	$\approx$
3	2	2
4	$\frac{24}{35}\pi^2$	$6.77 \cdot 10^0$
5	$\frac{286}{9}$	$3.17 \cdot 10^1$
6	$\frac{1296000}{676039}\pi^4$	$1.87 \cdot 10^2$
7	$\frac{12964479}{10000}$	$1.30 \cdot 10^3$
8	$\frac{3442073600000}{322476036831}\pi^6$	$1.03 \cdot 10^4$
9	$\frac{4155610296921974}{45956640625}$	$9.04 \cdot 10^4$
10	$\frac{10083790436267520000000}{109701233401363445369}\pi^8$	$8.72 \cdot 10^5$

Table 6.1: Values for Buchta's equations.

Using Stirling's formula

$$x! \;\approx\; \sqrt{2\pi x}\left(\frac{x}{e}\right)^x$$

with relative error (from [16]) of

$$\left(1+\frac{1}{12x}+\frac{1}{288x^2}-\frac{139}{5140x^3}+\mathrm{O}\left(\frac{1}{x^4}\right)\right)$$

we can approximate γ_x as

$$\frac{\sqrt{2\pi x}(x/e)^x}{2^{x+1}\left[\sqrt{\pi x}(x/2e)^{x/2}\right]^2}$$

simplifying to

$$\frac{\sqrt{2\pi x}(x/e)^x}{2^{x+1}\pi x(x/2)^x e^{-x}}=\frac{1}{\sqrt{2\pi x}}$$

We can then approximate

$$\frac{2}{d}\,\gamma_{(d-1)^2}\,\gamma_{d-1}^{-(d-1)} \quad \text{where} \quad \gamma_x = \frac{\Gamma(x+1)}{2^{x+1}((\Gamma(1+\frac{x}{2}))^2}$$

for odd dimension d as

$$\begin{aligned}
&\frac{2}{d}\frac{1}{\sqrt{2\pi(d-1)^2}}\left(\frac{1}{\sqrt{2\pi(d-1)}}\right)^{-(d-1)}\\
&\qquad=\frac{2\cdot(\sqrt{2\pi(d-1)})^{d-1}}{d\cdot(d-1)\sqrt{2\pi}}\\
&\qquad=\frac{2\cdot(\sqrt{2\pi})^{d-1}\cdot(d-1)^{(d-1)/2}}{d\cdot(d-1)\cdot\sqrt{2\pi}}\\
&\qquad=\frac{2\cdot(\sqrt{2\pi})^{d-2}\cdot(d-1)^{(d-3)/2}}{d}
\end{aligned}$$

with some constant error. A similar caluclation can be made for even d.

The actual number of hull facets m for convex hulls from distributions A and B and dimensions three through ten is shown in figure 6.2. In dimensions three through six m is within ten percent of the number predicted

by Buchta's equations for distribution A. For larger d, the number of facets m is somewhat less than predicted by Buchta's equations due to a delay in asymptotic behavior for values of n that are relatively low for the dimension.

The number of facets m is slightly greater for distribution A in lower dimensions, since for distribution B the number of vertices v of the convex hull is less than n. This comes from a result by Raynaud [18], who showed that the expected number of vertices of a convex polytope drawn from distribution B is $O\left(n^{(d-1)/(d+1)}\right)$. The actual number of vertices v for convex hulls from distribution B is shown in figure 6.3. For an actual set of 1600 points in $\Re^6$ the number of vertices is 1189.

The similarity between distributions A and B in higher dimensions has been called the orange-peel phenomenon, where the percentage of orange in the peel goes up as the orange gets larger. That is, keeping the number of points n and the radius of the sphere r constant, the percentage of the sphere in an outer layer of constant width increases exponentially with d. Therefore, for any given n, there is a dimension d where all n points of a set from distribution B can be expected to be vertices of the final convex hull. One might say that as the dimension increases, distribution B draws closer to distribution A.

This data points out the primary challenge to the efficient construction

Figure 6.2: Hull Facets: m by n.

Figure 6.3: Extreme points: v by n.

of convex hulls from these distributions or any spherical distribution where the number of vertices v draws closer to n as the dimensions rises—the very large output size compared with input size. We can expect that even an optimal algorithm with $\Theta(dm)$ complexity will quickly be bound by space before time if the output is to be stored.

The online memory available for the collection of the data presented here was limited to 32 megabytes. After overhead, 24 megabytes of memory was available for storage of the convex hull data structure. For dimensions seven through ten and distributions A and B, convex hulls were generated approx-

imately as large as could be generated within this bound.

6.2.1 Preprocessing Constructions

We now discuss the behavior of the preprocessing algorithm for sets of points from distributions A and B and dimensions three through ten.

Throughout we assume that the $\mathrm{O}(dn \log n)$ complexity of the preprocessing of the set of points S by lexicographic sort is negligible and may be ignored. This term is greatly dominated by the number of facets m that are processed by the rest of the algorithm. Furthermore, rather than link in an external sorting utility, the optimized quicksort recommended by Sedgewick [19] was implemented. Sedgwick's algorithm has an extremely short inner loop—test and increment—and therefore an unusually small constant. Even in three dimensions this preprocessing step typically takes a small fraction of one percent of the overall processing time.

The cpu timing results for the preprocessing construction of convex hulls of sets from distributions A and B are shown in figure 6.4. The performance of the construction for these sets seems to be close to linear in n. This can be considered fairly efficient, at least for distribution A, suggesting that the algorithm has time within a constant multiple of n for fixed d.

In lower dimensions for distribution B, where the number of vertices and

Figure 6.4: Timing results: Cpu-time by n.

facets is less than $\mathrm{O}(n)$ the algorithm cannot be considered as good. In fact, one might say that this algorithm is output-size insensitive, since the timing results are very similar for both distributions without regard to the number of facets in the final hull.

The behavior of the algorithm suggests a constant time update for fixed d, or, at least, an overall constant time update for both distributions A and B. Notwithstanding the evidence of the timing results, that this should be so is not obvious, since it implies that the average cost of adding a point to a convex hull in this manner is constant for fixed d, unrelated to the number of points n.

Consider the nature of the caps C_i created during the preprocessing construction of sets from distribution A or B. Without loss of generality, we may assume that no point of S shares the first coordinate with any other point, so that the lexicographic ordering is equivalent to a sort on first coordinate x_1. Each intermediate convex hull $P_i = \mathrm{conv}(p_1, \ldots, p_i)$ is then the convex hull of the subset of S on one side of a $(d-1)$-dimensional halfplane H, orthogonal to x_1 and passing through p_i. As the halfplane H is moved along the first coordinate, then each new point added to the hull is the extreme point of a d-dimensional hemisphere.

For example, for distribution A each intermediate hull P_i would be the

convex hull of a set of points uniformly distributed on the curved surface of the hemisphere as defined above. That is, $P_{n/4}$ would be the convex hull of a set of $n/4$ points uniformly distributed on the curved surface of (a little more than) a quarter sphere, $P_{n/2}$ would be the convex hull of a set of points uniformly distributed on just about a half sphere, and $P_{3n/4}$ would be the convex hull of a set of points distributed uniformly on the curved surface of (a little less than) a three-quarter sphere. For distribution B, each intermediate hull would be the convex hull of a set of points distributed uniformly inside a similar sequence of hemispheres.

There are therefore two types of points added to the hull. For distribution A each new point p_i is of the first type, uniformly distributed on the boundary between the curved and flat sides of the hemisphere P_i. In the limit for distribution B, $\mathrm{O}\left(n^{(d-1)/(d+1)}\right)$ of the additions will be vertices of the first type, and the remainder will lie on the flat side of the hemisphere.

A plausible conjecture would then be that the local neighborhood of each of type of vertex would be similar, that is, the average number of facets in the caps of type one vertices would be similar, and the average number of facets in the caps of type two vertices would be similar. This is what we find overall, at least for dimensions three through six, where the average number of facets a is shown in figure 6.5 and can be seen to be almost constant for

fixed dimension d. Furthermore, a does not differ much between distributions A and B. For $d > 6$, asymptotic behavior has not yet set in, although it can be seen to be starting in $\Re^7$.

This analysis leads to the following measure of efficiency for preprocessing constructions of convex hulls from distribution A. From Buchta we know that the expected number of facets of a set of n points is

$$m = c\,n$$

where c is constant for fixed d as defined in section 6.2. The complexity of a preprocessing construction of $\mathrm{conv}\,(S)$ is a function of the total number of facets t that are created, and

$$t \;= d + 1 + \sum_{i=d+2}^{n} |\,C_i\,| \;=\; \mathrm{O}\,(na)$$

The total number of facets t created for example sets from distributions A and B are shown in figure 6.6, where they can be seen to be close to linear in n. By comparing the total number of facets t with the size of the output m with the relation $n \times t/m$ in figure 6.7, it can be seen that the algorithm performs much better in lower dimensions for distribution A than for distribution B. For distribution A the total number of facets t is no more than $2\,d\,m$, at least for the sets studied. In other words, the average cap size for the preprocessing construction of convex hulls from distribution A for these sets is $\mathrm{O}\,(2dc)$, or up to about $2d$ times optimal.

Figure 6.5: Cap sizes: a by n.

Figure 6.6: Total facets: t by n.

Figure 6.7: Relative efficiency: t/m by n.

Finally, a purely empirical but practical measure of the efficiency of the algorithm is the relation $n \times$ cpu-time$/m$ which measures the time in seconds required for each facet in the output, shown in figure 6.8.

By this measure, the algorithm is much more efficent for distribution A than for B, taking much less time per facet of the final hull for the sets studied. On the other hand, the tendency appears to be towards equivalence, since the cost per facet for distribution B actually goes down as the dimension goes up, caused by the tendency of distribution B to act like distribution A in higher dimensions. That is, for constant n, the times for both distributions will approach each other as the dimension rises.

6.2.2 Online Constructions

In this section we discuss the behavior of the online construction of sets of points from distributions A and B and dimensions three through ten, including timing results and expected complexity, the number of total facets created, and related matters.

First behavior of the Two-Flat algorithm is discussed, used to identify interior vertices or a first visible facet as required for an incremental addition of each new point.

For each invocation of the Two-Flat algorithm for each point the 2-flat

Figure 6.8: Time per facet: Cpu-time/m by n.

Q was defined by three points—the new point, the centroid of the first facet F stored in the facet list, and the centroid of the closest neighboring facet G of F to p_i. Because a 2-flat can intersect the centroids of two neighboring facets without intersecting their common subfacet, Q was then shifted so as to assure it would. Finally, no circuit was followed past halfway or 180 degrees, since it would then complete and the point is guaranteed to be an interior point, thereby removing up to a factor of 2 from the length of a traversal when the new point is inside P. That is, if p is the new point, q is a point on the 2-flat in the first facet in the circuit, and r is a point on the 2-flat in the i'th facet in the circuit, then when the sign of triangle $\triangle pqr$ changes the point p must be inside the polytope and the traversal is terminated.

The complexity of the algorithm remains open. Several avenues of analysis were explored to obtain an upper bound. The worst case was chosen for simplicity, which for distributions A and B would be where the 2-flat cuts through the centre of the sphere and the number of facets that are involved in the intersection can be expected to be greatest.

The first approach to the analysis involved projecting the set of points S onto the 2-flat, and then bounding the number of facets by bounding the number of points within a certain distance of the boundary of the circle. Since the number of points close to the boundary goes down with d this

approach was at first thought promising, but an appropriate distance was not found.

Another avenue tried to measure the expected length of the segment formed by the intersection of the 2-flat with the average facet, but the characterization of an average facet was the primary difficulty. Other avenues included measuring the expected length of the sequence of edges between corresponding vertices of the facets in the circuit, and measuring the expected maximum $(d-1)$-dimensional area of the facets in the circuit.

However, in practice the performance of the Two-Flat algorithm is shown to be good in figure 6.9, where the average length of each traversal l is shown to be relatively small. For constant n the average length l actually goes down for sets from distribution A as the dimension increases, even though the number of facets m goes up rapidly, dropping to an average of approximately 25 steps per point for $n = 3000$ in $\Re^5$. From this data we conclude that the algorithm is an appropriate choice for the task.

Given that m rises very rapidly with d, a more informative measure of the algorithms efficiency is the relation $n \times w/m$, where w is the total number of steps taken by the Two-Flat algorithm for all points, measuring the proportion of facets processed by the Two-Flat algorithm compared to the size of the output, shown in figure 6.10.

Figure 6.9: Average 2-flat traversal: l by n.

By this measure, the Two-Flat algorithm can be seen to become much more efficient as the dimension rises, and for distribution A, the ratio w/m falls to 1 for $\Re^5$. Since the overall complexity of the Two-Flat traversals for all points can be done in $\Theta(d^3w)$ expected time, for $d > 4$ the contribution of this algorithm for distribution A is less than the processing required to produce the output. From this data we conclude that for dimensions five and greater the Two-Flat algorithm adds less than a factor of 2 to the overall complexity, and can therefore be considered to be minor to the remainder of the processing.

Finally, we present an argument to support the conjecture that the complexity of the algorithm is $\mathrm{O}\,(m^{1/(d-1)})$ for the distributions studied.

First, note that the length of the worst case circuit is $\mathrm{O}\,(m)$ in $\Re^2$. In three dimensions, it seems natural to expect a 2-flat through the center of the sphere to intersect something like $\mathrm{O}\,(\sqrt{m})$ facets, since the intersection is a one-dimenional line cutting through a two-dimensional area.

For general dimension d, keeping the radius r constant, the $(d-1)$-dimensional area of the surface of a sphere rises with $\mathrm{O}\,(r^{d-1})$, while the length of the one-dimensional circumference of the sphere stays constant at $2\pi r$. This line of reasoning suggests that the number of facets intersected by a worst-case 2-flat might be something like $\mathrm{O}\,(m^{1/(d-1)})$.

Figure 6.10: Relative two-flat efficiency: w/m by n.

If the facets were regular, all with equal size and shape, this argument could be extended almost immediately to give the $\mathrm{O}\left(m^{1/(d-1)}\right)$ upper bound. Unfortunately, there is no absolute guarantee immediately apparent that a number of long, thin facets won't be encountered, which when cut along their short side will extend the length of the circuit, even though this eventuality doesn't arise in practice.

Now we examine the overall performance of online constructions. The timing results for the online construction of sets of points from distributions A and B are shown in figure 6.11. Particularly in higher dimensions the online construction is significantly faster than the preprocessing construction. Again, the timing results appear to be close to linear in n, but with a smaller constant factor.

Assuming that the complexity of the Two-Flat algorithm is minor and can be neglected dimensions five and greater, the following analysis explains this behavior for distribution A.

A special case of the online problem for sets of points from distribution A has a complexity advantage. When the points arrive in random order, then the expected total number of facets t created by an online construction can be shown to be $\Theta(dm)$ as follows.

Since S is from distribution A, all n points of the set are vertices of

Figure 6.11: Cpu-timings: Cpu-time by n.

conv (S), and from Buchta [6] we know that

$$m = cn$$

where c is constant for fixed d. Now let a be the average number of facets around a vertex of conv (S). Then the following relationship holds: the number of vertices times a must equal the number of facets times d, or

$$na = md\,.$$

Solving the first and second equation for a then gives

$$a = cd$$

so the expected average number of facets a is bounded for fixed dimension d and not dependent on n or m.

In other words, the average size of each cap C_i does not increase with i. This property is familiar in $\Re^3$ where the number of facets of a convex hull from distribution A is $2n - 4$, so $c \approx 2$ and the average number of facets around each vertex is approximately $2 \cdot d = 6$.

The total number of facets t created by an online construction of the convex hull of a set of points from distribution A is therefore

$$t \;=\; (d+1) + \sum_{i=d+2}^{n} |\,C_i\,|$$

so

$$t \;=\; \Theta(na) \;=\; \Theta(ncd) \;=\; \Theta(dm)$$

The same analysis holds for other spherical distributions, except that the number of facets m is a function of cv instead of cn. For example, for distribution B the expected total number of facets created will be $t = \Theta(dcn)$, but the expected number of facets in the final hull m is less than cn whenever all n points of the set are not vertices of the final convex hull.

This is the behavior of the construction in practice. The relative efficiency of the online construction as measured by total facets created t divided by number of facets in the final hull m, $n \times t/m$, is shown in figure 6.12. For distribution A it can be seen that $t/m \approx d$, so d times as many facets are created than will be present in the final convex hull.

Assuming that the Two-Flat procedure can be neglected in dimensions $d \geq 5$, and an $\Theta(d^3)$ cost to create each facet, the expected complexity of the online construction of convex hulls from distribution A is therefore $\Theta(d^4 m)$. In the following section it will be shown that an additional factor of d can be dropped from this complexity.

Finally, the empirical measure of efficiency $n \times$ cpu-time$/m$ is shown in figure 6.13, measuring the actual cost in cpu-time for the production of each facet in the output, where it can be seen again that the online construction out performs the preprocessing construction in higher dimensions. For both distributions the cost per facet is actually dropping as the dimension increases

Figure 6.12: Relative efficiency: t/m by n.

for the sets studied, for distribution A to about 0.0025 seconds for $n = 3000$ in $\Re^5$.

6.2.3 The Interlink Procedure

The performance of the construction of convex hull is primarily a function of the creation of facets, but is also affected by the processing performed by the Interlink algorithm, which links neighboring cap facets together across their common subfacets.

Each invocation of Interlink causes a walk around a subsubfacet through the visible set. Even though the cost of the walk has been amortized away, the actual performance of the task might remain of interest. If it turned out to be too inefficient, the $\mathrm{O}(|C|\log|C|)$ scrabble sort procedure to perform the same task could be preferable (see section 4.2.2), even though it requires $\Theta(d^2|C|)$ extra space.

One might assume that the length of each walk would rise rapidly with the dimension, but in fact the average number of facets r in each subsubfacet rotation remains constant as shown in figures 6.14—6.15 for both the preprocessing and online cases and distributions A and B. For all cases, $r < 4$. This implies that rotation around a subsubfacet in arbitrary dimension does not differ much from the three dimensional case, where there are on average

Figure 6.13: Cost per facet: Cpu-time/m by n.

six facets around each subsubfacet (point), and we would expect the average number of facets in each rotation to be less than or equal to four.

For spherical distributions the time complexity of each subsubfacet rotation can then be given as $\Theta(d)$. This allows us to reduce the expected time complexity of the online construction of convex hulls from distribution A as follows.

Recall that phase two of the update has complexity $\mathrm{O}\,(d^3|\,V\,| + d^2|\,C\,|)$, and that the $d^3|\,V\,|$ term was calculated by bounding the number of facet/subsubfacet pairs in the visible set V. Now the results in figures 6.14–6.15 show that the average length of each subsubfacet rotation is constant with n and d, so the average expected complexity of all $d \cdot |\,C\,|/2$ subsubfacet traversals can be given more exactly as $\Theta(d^2|\,C\,|)$ instead of $\mathrm{O}\,(d^3|\,V\,|)$. The overall expected complexity of phase two is therefore just $\Theta(d^2|\,C\,|)$.

Now each new cap facet T created is the union of a new point p and a subfacet of an existing nonvisible facet G on the horizon. Therefore, T and G differ by only one point, and the halfspace equation Half^T can be created in $\Theta(d^2)$ time from the system of linear equations used to create Half^G. This would drop the complexity of phase one of the update also to $\Theta(d^2|\,C\,|)$. Given that the online construction of convex hulls from distribution A creates $t = \Theta(dm)$ facets, the overall expected complexity of the algorithm can be

Figure 6.14: Interlink performance: r by n.

Figure 6.15: Interlink performance: r by n.

reduced to

$$\Theta(d^3 m)$$

by trading off an increase in storage from $\Theta(dm)$ to $\Theta(d^2 m)$ to store the SSLE used to generate the halfspace equation of each facet. The advisability of this method will depend on which constraint is in shortest supply, time or space.

6.3 Construction of Cyclic Polytopes

Recall from section 2 that cyclic polytopes are worst-case convex hulls, defined for example by a set of points that fall on a moment curve:

$$(\, t, t^2, t^3, \ldots, t^d \,),\ t \in \Re$$

All cyclic polytopes of n points in d-space are combinatorially equivalent, are the worst-case convex hulls in terms of view of size with $\mathrm{O}\,(n^{\lfloor d/2 \rfloor})$ facets, and can be expected to be more difficult to construct than convex hulls from spherical distributions.

The space complexity of convex hulls results in a curious feature: as we can see by the presence of the floor operator, there is a step between consecutive odd-even dimensions in which the number of facets increases by a factor of n. That is, cyclic polytopes in consecutive even-odd dimensions,

2 and 3, 4 and 5, 6 and 7, and so on are of the same order. This feature is familiar in $\Re^2$ and $\Re^3$, where the worst-case number of facets is n and $2n-4$ respectively.

The vertices of the cyclic polytopes were generated from the distribution

$$p_i = (i, i^2, \ldots, i^d)$$

on the moment curve $(t, t^2, t^3, \ldots, t^d)$, using integer arithmetic, and then randomly ordered for online constructions.

Construction of cyclic polytopes from this distribution quickly revealed that they constitute a particularly difficult case, limited to very low values of n and d, above which the minimum angle between the normals of adjacent facets of the polytope becomes zero. When this happens, the actual convexity of the final polytope can no longer be assured, and shortly thereafter concavities were found to occur between adjacent facets. This happens very quickly, at $n = 30$ in six dimensions, $n = 20$ in seven dimensions, and only $n = 16$ in eight dimensions.

Alternative combinatorially equivalent distributions were tried to alleviate this problem, using smaller sequences of ascending powers and smaller sequences of bases, as well as alternative mathematical procedures from IMSL and LINPACK for generation of the facet equations, always using original data, but without significantly increasing the range of n.

From this we conclude that convex hulls of this type are particularly difficult to construct, not so much because of space or time as might be expected, but because of the near coplanarity of adjacent facets. A more practical measure of the complexity of this type of problem would be the number of bits needed to maintain the required precision.

Therefore, in the following we only present data for cyclic polytopes that are correctly generated for three through eight dimensions. Figure 6.16 shows the exponential rise in the number of facets m with d, where one can see the clustering of even/odd dimensions 4 & 5 and 6 & 7.

Figure 6.16: Cyclic polytopes: m by n.

First we show a similar result as given in [21], that the incremental construction of a cyclic polytope is O $(n^{\lfloor d/2 \rfloor})$ for even dimensions and O $(n^{\lfloor d/2 \rfloor + 1})$

for odd dimensions. This can be done by finding an upper bound on the size of a cap by the construction of a vertex figure.

Lemma 4 *The number of facets in cap* $|C_i|$ *of a cyclic d-polytope is at most* $\mathrm{O}\left((i-1)^{\lfloor (d-1)/2 \rfloor}\right)$.

Proof: Consider a vertex figure $\mathcal{V}$ of vertex p_i of intermediate hull P_i, formed by the intersection of a hyperplane H with P_i separating p_i from the other vertices. Now V is therefore a $(d-1)$-polytope, and its vertices and facets are defined by the intersection of H with the edges and facets containing p_i respectively. There are at most $i-1$ edges containing p_i in P_i, and therefore at most $i-1$ vertices in $\mathcal{V}$. In the worst case therefore, $\mathcal{V}$ can have at most $\mathrm{O}\left((i-1)^{\lfloor (d-1)/2 \rfloor}\right)$ facets. So there are at most $\mathrm{O}\left((i-1)^{\lfloor (d-1)/2 \rfloor}\right)$ facets in P_i containing p_i. □

With a bound on the size of a cap, we can then bound the time complexity of $\Theta(n)$ such updates as follows.

The algorithm is $\mathrm{O}\left(d^3 t\right)$, where t is the total number of facets created or

$$t \;=\; d+1+\sum_{i=d+2}^{n} |C_i|$$

Now we know that $|C_i| = \mathrm{O}\left((i-1)^{\lfloor (d-1)/2 \rfloor}\right)$, so

$$t \;=\; \mathrm{O}\left(\, n^{\lfloor (d-1)/2 \rfloor + 1}\right)$$

But when d is even then

$$\lfloor (d-1)/2 \rfloor + 1 \;=\; \lfloor d/2 \rfloor$$

so the overall algorithm has optimal $\Theta(n^{\lfloor d/2 \rfloor})$ complexity in even dimensions and $\mathrm{O}\left(n^{\lfloor d/2 \rfloor + 1}\right)$ complexity when d is odd.

The actual performance of both the preprocessing and online methods of construction as measured by $n \times \text{Cputime}$ is shown in figure 6.17.

Both algorithms perform equally well for the even dimensions, in terms of time, but the online construction is markedly faster for odd dimensions because the preprocessing algorithm creates a much larger total number of facets t in odd dimensions as shown in figure 6.18. For example, the online construction appears to create an almost linear number of facets in $\Re^3$.

To explain this behavior, the actual number of facets surrounding each point p_i of the set was calculated, showing the pattern shown in table 6.2. For even dimensions, the number of facets around each vertex is always equal for all n vertices of the polytope. For the odd dimensions the number of facets around each vertex is equal for $n - (d+1)$ of the vertices in the middle of the set, while the remaining vertices on each end of the curve have a variable number with the two extreme vertices having the most.

Assuming that this pattern is repeated in higher dimension (which may be likely but not necessarily certain) then preprocessing constructions must

Figure 6.17: Cpu timings: Cpu-time by n.

Figure 6.18: Total facets: t by n.

		Number of facets around each vertex										
d	n	p_1	p_2	p_3	p_4	p_5	$\cdots$	p_{n-4}	p_{n-3}	p_{n-2}	p_{n-1}	p_n
2	n	2					$\cdots$					2
3	n	$n-1$	3	4			$\cdots$			4	3	$n-1$
4	26	46					$\cdots$					46
	27	48					$\cdots$					48
	28	50					$\cdots$					50
	29	52					$\cdots$					52
5	26	275	65	85	84		$\cdots$		84	85	65	275
	27	299	68	89	88		$\cdots$		88	89	68	299
	28	324	71	93	92		$\cdots$		92	93	71	324
	29	350	74	97	96		$\cdots$		96	97	74	350
6	26	462					$\cdots$					462
	27	506					$\cdots$					506
	28	552					$\cdots$					552
	29	600					$\cdots$					600
7	16	275	155	191	183	184	$\cdots$	184	183	191	155	275
	17	352	187	232	223	224	$\cdots$	224	223	232	187	352
	18	442	222	277	267	268	$\cdots$	268	267	277	222	442
	19	546	260	326	315	316	$\cdots$	316	315	326	260	546

Table 6.2: Cap sizes for cyclic polytopes.

create many more convex hull facets than are in the final hull, since each point added to the polytope is an endpoint.

For example, in $\Re^3$ we can imagine a cyclic polytope as a spiral with the vertices falling on the curve (t, t^2, t^3). If a light source were placed at $z = \infty$ then half of the polytope would be lit and half would be dark. The $n - 2$ facets on top of the polytope would be lit and all contain the point p_n, and the $n - 2$ facets on the bottom of the polytope would be dark and all contain the point p_1. When this polytope is constructed by adding the points in lexicographic order, every point p_i added to the polytope requires the deletion of $i - 3$ facets and the creation of $i - 1$ new facets, resulting in an overall $\Theta(n^2)$ expected complexity even though the number of facets in the final hull is only $2n - 4$.

In other words, it is possible in odd dimensions to add the points in an order such that the size of each cap will be on the order of the size of the entire hull, resulting in an algorithm with $\Theta(nm)$ expected complexity, and when the points of the set are added in the order they appear on the moment curve this worst-case is achieved.

Note that this is not necessarily the result of the lexicographic sort. It would be possible for the set of points to be oriented so that the sort places the $d + 1$ endpoints first in the set, leaving the remaining updates with constant

cap sizes, but without knowing beforehand which vertices were endpoints it is not clear how this orientation could be arranged.

For even dimensions on the other hand, no cap can be more than $\mathrm{O}(m/n)$, so the order of arrival of the points is irrelevant to the analysis of complexity. Furthermore, the data in table 6.2 suggests that in even dimensions each cap $|C_i|$ has the same number of facets as surround any of the other $i-1$ vertices, so that the order of arrival truly does not affect the total number of facets created.

For the online construction the total number of steps w taken by the Two-Flat algorithm is quite small (see tables of data in appendix B), and the ratio w/m is even smaller for these sets than for spherical distributions. For cyclic polytopes or similar distributions the Two-Flat algorithm appears to be an excellent choice, and we can assume that its contribution to the remainder of the algorithm can be neglected even in low dimensions.

Assuming that the pattern shown in table 6.2 is repeated in higher dimensions, then the complexity of the online construction can be shown to be within a constant multiple of optimal for fixed d by a similar analysis as for the online construction of convex hulls from spherical distribution A.

That is, given a cyclic polytope with n vertices, the number of vertices

times the average number of facets around each vertex equals md, or

$$na = md$$

and

$$a = md/n$$

From table 6.2 we find that for fixed d in both even and odd dimensions the number of facets surrounding each vertex is constant for $\Theta(n)$ points, and the endpoints in odd dimensions can be ignored since there are only a constant number and the total number of facets in their caps can be no more than $\mathrm{O}(m)$. The complexity of the construction is then a function of the total number of facets created t, which can be bounded by

$$t \;=\; \mathrm{O}(na) \;=\; \mathrm{O}(dm)$$

The relative performance of both the preprocessing and online constructions is shown in figure 6.19 by the relation $n \times t/m$. The preprocessing construction performs more poorly than the online construction, but becomes more efficient in consecutive odd/even dimensions.

The online construction performs somewhat better, partly driven by lower order terms. That is, for spherical distribution A the cap size stays constant for fixed d but for cyclic polytopes $\sum_{i=d+2}^{n} |C_i|$ is generally much less than na, since $i^{\lfloor (d-1)/2 \rfloor}$ is generally much less than $n^{\lfloor (d-1)/2 \rfloor}$.

Figure 6.19: Relative efficiency: t/m by n.

The better than expected complexity for the online construction of odd-dimensional cyclic polytopes is also affected by the fact that from table 6.2 we see that most vertices of an odd-dimensional cyclic polytope of n points are in fewer than m/n facets. For example, in three dimensions, $n-(d+1)$ of the vertices are in exactly four facets, less than the overall average of six. The average cost of an addition in odd dimensions is therefore less than $\mathrm{O}\,(m/n)$. This has a curious result.

For the online construction the ratio t/m is actually dropping for consecutive dimensions d and $d+2$ in figure 6.19 to some value under 2. From this data we conclude that in practice this construction is very close to optimal in both n and d. That is, in practice S will probably not be the vertices of a perfect cyclic polytope, but rather a set of points from a very similar distribution with similar characteristics. The data shown here then suggests that the complexity of the Two-Flat traversals is negligible, the total number of facets created t gets closer to m as d rises, and the expected complexity of the construction therefore approaches $\Theta(d^3m)$.

By making the same space/time trade-off made in section 6.2.3 to generate halfspace equations in $\Theta(d^2)$ time from existing data, the expected complexity of the online construction of cylic polytopes in both even and odd dimensions can be reduced to $\Theta(d^2m)$, which is a better result than that

obtained for spherical distributions by a factor of d.

6.4 Space Requirements

Recall from section 4.1 that the space required for storage of each convex hull facet is $\Theta(d)$ due to the implicit definition of subfacets. The storage actually required for a facet with 64-bit storage for floating point values is $16d + 36$ bytes, or $128d + 288$ bits.

With respect to temporary storage during construction, the space to store the visible set is required throughout an update in order to carry out the subsubfacet traversals during the Interlink phase. In the worst-case a given visible set could include all but one facet of the existing convex hull. This points out a weakness of the incremental paradigm for specialized sets of points that are processed in an unfortunate order, where it could be possible that intermediate hulls P_i will have many more facets than are actually in the final hull P_n. The same concern does not apply to the true online problem, where the arrival order of the points cannot be controlled.

For either a preprocessing or online construction, an upper bound on the amount of memory required for the construction of a convex hull is clearly less than twice as much as required for storage of the largest intermediate

convex hull, since the maximum storage required is the maximum for all i of

$$|P_i| + |C_{i+1}| < |P_i| + |P_{i+1}|$$

For the online construction of convex hulls from distribution A the expected amount of storage required will be $d|P_{n-1}| + d|C_n|$. If the expected number of facets of a hull of i vertices is ci where c is as defined by Buchta's equations, then we know that the expected size of $|C_i|$ is cd from section 6.2.2, and so the expected storage required will be

$$\Theta(dc(n-1) + d(dc)) \;=\; \Theta(dc(n+d-1))$$

In other words, expected amount of storage required for the online construction of a convex hull of n points from distribution A is the same as is required just to store a convex hull of $n + d - 1$ points, which may be useful when n is close to d and the dimension is large.

6.5 An Application

The algorithms developed here were applied to a real world problem, the Delauney triangulation of two dimensional surveying data along the Saint John river, with mixed results.

The construction of the Delauney triangulation of a set of points in the plane is a common precursor of many algorithms in computational geome-

try. The Delauney triangulation is defined as follows: assuming that no four points lie on any one circle, the circumcircles of the points of each triangle do not include any other point. The Delauney traingulation is generally preferred over other triangulations for many applications because of its regularity, i.e., it has the maximum minimum angle over all triangulations.

The Delauney triangulation is also the dual of the Voronoi diagram, the set of polygons consisting of the portion of the plane closest to each point p of the set S. The Voronoi diagram is a very useful structure, providing optimal solutions to a number of proximity problems, such as the closest pair, nearest neighbor search, all nearest neighbors, and planar convex hull problems [20]

An $\Theta(n \log n)$ divide and conquer algorithm is given in [20] for the planar Voronoi diagram problem, and, by extension, the Delauney triangulation problem. However, Brown has also showed that the d-dimensional Delauney triangulation is transformable from the $(d+1)$-dimensional convex hull problem [5]. This can be done by putting the d-dimensional set of points onto a $(d+1)$-dimensional paraboloid, that is

1. Make each point $p = (c_1, \ldots, c_d)$ of the d-dimensional input set S into the point $p^* = (\, c_1, \ldots, c_d, \, |p|^2 \,)$ of $(d+1)$-dimensional set S^*;

2. Find conv (S^*);

3. Delete the 'top' facets of the polytope, i.e., those facets with a negative

value in the last coordinate of the halfspace equation;

4. Project the remaining facets back into $\Re^d$.

The procedure outlined above was implemented to generate the 2-dimensional Delanauy triangulation, i.e., to find the convex hull of the 2-dimensional set of points S projected on the three dimensional paraboloid $z = x^2 + y^2$, delete the top facets, and then project the set of facets back into the plane. The procedure was then applied to real data collected by a local surveying company.

Generation of the triangulation for the first set of data was successful, for $n = 20509$, and taking a minute and a half cpu time in total.

However, for the second set of data, $n = 43621$, the deletion of facets with upward normals failed. By generating the 2-dimensional hull of the set, it was found to have only 28 vertices, because, as it was later disovered, it was drawn from data collected along a river valley. In other words, some long thin triangles on the edge of the set had normals with z coordinate very close to zero. This suggest that this method may be impractical for the generation of Delauney triangulations of these types of sets.

On the other hand, this is not necessarily a barrier to the construction of Delauney triangulations of these types of sets by this procedure with some alterations, especially considering its successful generation of the Delauney

triangulation of the first set of 20509 points, which was later found to have only 64 extreme points in the plane.

For large sets of points of this type with long, relatively straight sides, one possible approach would be to add extra points to the set in appropriate places in order to make the boundary more regular. Alternatively, the set could be divided into partitions of an appropriate size—that is, small enough that the points of the problem facets are low enough on the parabola where the curve is more pronounced to have the correct sign in the z coordinate. The optimal merge algorithm given for the planar divide-and-conquer Voronoi diagram problem could then be used with modification to stitch the partitions together.

Chapter 7

Conclusions

The algorithms presented here provide perhaps the best online solution to the higher dimensional convex hull problem for sets of points from spherical distributions. The algorithms were tested in three through ten dimensions, and have been demonstrated to provide fast, reliable solutions up to a limit of twenty-four megabytes storage to store the final data structure.

7.1 Observations

The online construction consistently performed better than the preprocessing construction by every measure, primarily because the average cap size when the points are added in lexicographic from spherical distributions are approximately twice as large as when the points are added in random order.

The online construction is therefore preferred even when the set is entirely known beforehand.

The incremental paradigm as implemented here for the online problem generates the convex hull of sets of points distributed uniformly on spheres in $\Theta(d^4m)$ expected time, with $\Theta(dm)$ space required to store the convex hull, and $\Theta(dm\frac{n+d-1}{n})$ expected space required for the construction.

The average length of each subsubfacet traversal remains less than four with both n and d in all cases. Each pair of links between neighboring cap facets can therefore be set in $\Theta(d)$ expected time, and the overall cost to add a cap of $|C|$ facets to the hull can be done in $\Theta(d^2|C|)$ expected time, by making a space trade-off and increasing the storage required to store the hull to $\Theta(d^2m)$ in order to generate facet equations in $\mathrm{O}(d^2)$ time from existing data. The expected time complexity of the algorithm can thereby be reduced to $\Theta(d^3m)$.

In practice, at least for the sets investigated here, the primary constraint may well be space before time. When building convex hulls of sets of points distributed on spheres in high dimensions an increase in storage requirements by a factor of d will often be unacceptable, and the original algorithm will probably be preferred in most cases.

For sets of points distributed inside spheres, like distribution B, the re-

sults collected suggest that the most practical course in most cases is to remove interior points in a preprocessing step, and then construct the convex hull of the remaining vertices in $\Theta(d^4cv) = \Theta(d^4m)$ expected time where v is the number of vertices and c is the constant defined by Buchta's equations. Any method for this procedure, even the $O(n^2)$ linear programming method given in chapter 3, will be acceptable whenever c is large compared to v. For example, some convex hulls of sets of points normally distributed inside spheres were also investigated, where the algorithms were found to be hopelessly inefficient without a preprocesser to remove interior vertices from the set.*

From table 6.1 it can be seen that in $\Re^{10}$ the number of points n would have to be approximately 872000 before n^2 would become larger than m. In other words, an $O(n^2)$ procedure for removal of interior points would be minor compared to the remaining processing in higher dimensions for all but extremely large values of n, where the size of the output makes the problem practically intractable in any case.

Assuming for the moment that we have unlimited space, the other side of the above argument is that for fixed d the construction of the convex hull provides an algorithm that is linear in n for the vertex enumeration

*A reliably fast method for this procedure that does not require linear programming machinery is given in [24].

problem, and the removal of interior points might be most efficiently done for very large sets by finding the facets of the convex hull. For example, for a set S of $n = 10^{1000}$ points distributed uniformly inside a sphere in $\Re^{10}$, the identification of the vertices of S by finding the facets of the convex hull in $\Theta(n)$ time might well be faster than alternative methods. However, it is clear that this constitutes a case well outside of the capability of present technology, and regardless of advances in processing speed and storage it will remain true that there will always be a value of d for which it will be faster to remove interior vertices in a preprocessing step than to possibly add them to the hull incrementally and delete them later.

Assuming that the conjecture that the complexity of the Two-Flat algorithm is $\mathrm{O}\left(m^{1/(d-1)}\right)$ by the argument given in section 6.2.2, the overall complexity of the online construction of convex hulls from distribution A is then $\mathrm{O}\left(d^4m + d^3nm^{1/(d-1)}\right)$.

The results in section 6.2.2 show that the Two-Flat algorithm provides a fully acceptable solution to the two requirements of an online incremental update, identifying interior vertices or a visible facet as required, and that it becomes markedly more efficient as the dimension rises. For $d \geq 5$ for distribution A it traverses in total fewer facets than are in the final hull, so in dimensions greater than four the Two-Flat algorithm increases the overall

complexity of the algorithm by less than a factor of two, and can then be considered to be minor.

For cyclic polytopes the Two-Flat algorithm works even better than for spherical distributions. For these worst-case convex hulls the expected average length of a traversal to a visible facet is relatively short, and so the Two-Flat algorithm can be considered to be an appropriate choice for these types of distributions as well, contributing a minor cost to the overall processing even in low dimensions.

The online construction of cyclic polytopes created far fewer total facets t compared to the output size m. In even dimensions this behavior is caused by lower order terms, i.e., by the fact that the average cap size is generally much less than m/n for $i < n$. For odd dimensions this behavior is also caused by the fact that the average cap size of most of the vertices interior to the curve is less than m/n even for the final polytope. For example, the cap size of $n - 4$ of the vertices of a 3-dimensional cyclic polytope is only four, even though the overall average cap size must be approximately six (see table 6.2). As a result, the ratio of total facets created t divided by the output size m for consecutive (even or odd) dimensions d and $d + 2$ falls to some value less than 2, so that $t = \Theta(m)$. The expected complexity of the online construction of cylic polytopes can then be given as $\Theta(d^3m)$, or $\Theta(d^2m)$

with the same space/time trade-off as discussed for spherical distributions, which is actually better by a factor of d than the result obtained for spherical distributions.

At the same time, it must be noted that the construction of cyclic polytopes was shown to be particularly difficult in practice, not because of time or even space, but due to the near coplanarity of adjacent facets.

7.2 Open Questions

The major open question is the order of average case complexity of the Two-Flat algorithm. In practice the method works well for the sets of points studied, with time requirements well within acceptable limits, but obtaining an upper bound, even just for spherical distributions, would be desirable. As well, a reliable method of dealing with degeneracy would be desirable.

A more reliable solution for the Delauney triangulation problem by Brown's method would probably be directly usable by agencies with real-world requirements. As given the method works well, but for sets with long thin triangles on the edge of the set we encounter precision problems. One of three approaches is possible: greatly increase the floating point precision, add new points to the set, or break the set into small enough pieces to be solved correctly and then merge them together with the optimal algorithm

given in [20]. Actually, since the time of the merge algorithm is a function of the number of triangles on the edges of the sets, the complexity of the merge should be somewhat better than linear.

The utility of the algorithm for problems in very high d using external storage would be of interest, where asymptotic behavior in much higher dimensions could be investigated. A version of the algorithm of this type would be most appropriate for a stand-alone machine with very large external storage where time was not a constraint.

The results obtained for the Delauney triangulation problem and cyclic polytopes suggest that an important factor in the complexity of convex hull finding algorithms, even for sets in general position, is the number of bits required to maintain the needed precision for differentiation of the normals of adjacent facets. This implies that the generation of convex hulls of sets not in general position may be fraught with difficulties, where the angle between adjacent facets is zero. Assuming infinite precision, the algorithm as given will work correctly—that is, it will simply break non-simplicial facets into several adjacent simplicial facets. Lacking sufficient precision, perturbation methods may prove sufficient.

Appendix A

Software

The following PL/1 program constructs convex hulls for the preprocessing case or the online case, as selected by the argument proc_type.

Rather than using pointers, a self-initializing doubly linked list is used. That is, the list is initialized as new nodes are requested. This approach was chosen to avoid storage overflow, and to minimize the possibility of anomolies in the cpu timing results caused by repeated dynamic memory allocation. The recursion of Traverse is unwrapped.

The program exits gracefully on the following errors: overflow of available storage; degeneracy in the Two-Flat procedure; and failure to create a halfspace equation, signaled by a determinant of zero.

```
 Convex : procedure
    ( S, n, d, S_index, P, p_size, m, proc_type, error_code ) ;

/*--------------------------------------------------------------------
|                                                                     |
|  Name:                                                              |
|                                                                     |
|     Convex                                                          |
|                                                                     |
|  Description:                                                       |
|                                                                     |
|     Find conv(S). Return a linked set of facets P.                  |
|                                                                     |
|  Usage:                                                             |
|                                                                     |
|     call Convex ( S, n, d, S_index, P, p_size, m, proc_type,        |
|                   error_code )                                      |
|                                                                     |
|     S ( n, d )   - Set of n points in Rd.                           |
|                                                                     |
|     n            - Size of S.                                       |
|                                                                     |
|     d            - Dimension.                                       |
|                                                                     |
|     S_index      - Permutation index of S if sorted.                |
|                                                                     |
|     P ( * )      - Set of facets.                                   |
|                                                                     |
|      vertices ( d ) - Indices of points of facet.                   |
|      neighbor ( d ) - Pointers to neighboring facets.               |
|      h_y      ( d ) - Coefficients of halfspace equation (h_y,h_b)  |
|      h_b            - RHS constant of halfspace equation.           |
|      h_length       - Length of halfspace vector h_y.               |
|      non_visible_facet - Set on creation, used by Interlink.        |
|      visible_facet  - Set on creation, used by Interlink.           |
|      in_v           - Set if facet is put in visible set V.         |
|      f_link         - Forward link.                                 |
|      b_link         - Backward link.                                |
|      t_link         - Temporary link, used to put facets in the     |
|                       visible set V and the cap C.                  |
|                                                                     |
|     p_size       - Size of P data structure. Must be as large as    |
|                    the maximum number of facets required.           |
|                                                                     |
|     m            - Number of facets in the convex hull.             |
|                                                                     |
|     error_code   -                                                  |
|        0 = No error.                                                |
```

```
|       1 = Overflow of facet data structure.                          |
|       2 = Wrong path taken by Two_Flat.                              |
|       3 = Failure of Create_Facet.                                   |
|                                                                      |
|    proc_type    -                                                    |
|       1 = Preprocessing construction.                                |
|       2 = Online construction.                                       |
|                                                                      |
|                                                                      |
|  Logical Program Structure:                                          |
|                                                                      |
|    Convex_initialization                                             |
|       Sort1                      (if proc_type is preprocessing)     |
|       Create_simplex                                                 |
|          Get_new_facet                                               |
|          Create_H                                                    |
|             Lneqs                                                    |
|                                                                      |
|    Two_Flat                      (if proc_type is online)            |
|       Lneqs                                                          |
|                                                                      |
|    Traverse                                                          |
|       Create_and_link_new_facet                                      |
|          Get_new_facet                                               |
|          Find_common_subfacet                                        |
|          Create_H                                                    |
|             Lneqs                                                    |
|                                                                      |
|    Interlink                                                         |
|       Rotate                                                         |
|       Find_common_subfacet                                           |
|                                                                      |
|    Delete_V                                                          |
|                                                                      |
|    Convex_termination                                                |
|                                                                      |
|  Reference:                                                          |
|      An Algorithm to Find the Facets of the Convex Hull in           |
|      Higher Dimensions, SCCCG Proceedings, August 1990.              |
|                                                                      |
|  Author:                                                             |
|    Wm. M. Stewart                                                    |
|    Computer Science                                                  |
|    University of New Brunswick                                       |
|    Fredericton, New Brunswick                                        |
|    Canada, E3B 5A3                                                   |
|                                                                      |
|  Creation Date:                                                      |
```

```
|     4 December 1989                                                  |
|                                                                      |
|  Last Revision Date:                                                 |
|     5 September 1991                                                 |
|                                                                      |
|  Copyright:                                                          |
|     None.                                                            |
|                                                                      |
----------------------------------------------------------------------*/

/*----------------------------------------------------------------------
|                                                                      |
| Variable dictionary:                                                 |
|                                                                      |
|    avail             - Head of available list of facets in P.        |
|    c_header          - Head of list of cap facets in C.              |
|    centroid ( d )    - Centroid of first simplex P, used to set      |
|                        the correct sign of halfspace equations.      |
|    d2                - Constant set to d+2, for declaration of       |
|                        arrays of size d+2.                           |
|    f                 - A facet.                                      |
|    i                 - Index of current point in S.                  |
|    k                 - Visible facet found by two_flat algorithm.    |
|    link_full         - Switch, turned on when the dynamic            |
|                        initialization of the linked list of facets   |
|                        finishes.                                     |
|    link_init_counter - Counter for dynamic initialization of the     |
|                        linked list of facets.                        |
|    p ( d )           - Current point.                                |
|    stack ( p_size )  - Stack to unwrap the recursion of Traverse.    |
|    subsubfacet_size  - Constant set to d-2, for declaration of       |
|                        subsubfacet by Interlink.                     |
|    v_header          - Head of list of visible facets in V.          |
|                                                                      |
|----------------------------------------------------------------------*/
```

```
declare

   s ( *, * )                  float binary ( 21 ),
   n                           fixed binary ( 31 ),
   d                           fixed binary ( 31 ),
   s_index ( * )               fixed binary ( 31 ),

   1 P ( * )                   controlled,
      2 vertices ( * )         fixed binary ( 31 ),
      2 neighbor ( * )         fixed binary ( 31 ),
      2 h_y      ( * )         float binary ( 53 ),
      2 h_b                    float binary ( 53 ),
      2 h_length               float binary ( 53 ),
      2 non_visible_facet      fixed binary ( 31 ),
      2 visible_facet          fixed binary ( 31 ),
      2 in_v                   bit ( 1 ),
      2 f_link                 fixed binary ( 31 ),
      2 b_link                 fixed binary ( 31 ),
      2 t_link                 fixed binary ( 31 ),

   p_size                      fixed binary ( 31 ),
   m                           fixed binary ( 31 ),
   proc_type                   fixed binary ( 31 ),
   error_code                  fixed binary ( 31 ),


   avail                       fixed binary ( 31 ),
   c_header                    fixed binary ( 31 ),
   centroid ( d )              float binary ( 53 ),
   d2                          fixed binary ( 31 ),
   f                           fixed binary ( 31 ),
   i                           fixed binary ( 31 ),
   k                           fixed binary ( 31 ),
   link_full                   bit          (  1 ),
   link_init_counter           fixed binary ( 31 ),
   p ( d )                     float binary ( 53 ),
   stack ( p_size )            fixed binary ( 31 ) ctl,
   subsubfacet_size            fixed binary ( 31 ),
   v_header                    fixed binary ( 31 ) ;

allocate stack ;
```

```
     /* Create initial simplex, centroid, initialize s_index, etc. */

 call convex_initialization ;

     /* Add points i=d+1...n. */

 i  =  d+1 ;

 do while (  i < n  &  error_code = 0  ) ;

    i       =  i + 1 ;
    p ( * ) =  s ( i, * ) ;

    if proc_type = 2 then k = two_flat ;

    if k > 0 then do ;                  /* Update P with p_i. */

       f            =  k ;
       k            =  0 ;

       in_v   ( f ) = '1'b ;
       t_link ( f ) =  0 ;
       v_header     =  f ;

       c_header     =  0 ;

       call traverse ( f ) ;

       if error_code = 0 then do ;

          call interlink ;
          call delete_v ;

       end ;

       end ;

    end ;

 end ;

 if error_code ^= 0 then do ;
    put skip list
        ( '*** ERROR ***   Error_code =', error_code, 'I = ', i ) ;
 end ;

return ;
```

```
1traverse : procedure ( f ) ;

 /*-------------------------------------------------------------------
 |  Traverse and identify the set of visible facets V, create the    |
 |  cap C, and link C to the nonvisible facets on the horizon.       |
 -------------------------------------------------------------------*/

  declare
     stack_top                    fixed binary ( 31 ),
     f                            fixed binary ( 31 ),
     g                            fixed binary ( 31 ),
     j                            fixed binary ( 31 ),
     more                         bit ( 1 ),
     sum                          builtin,
     t                            fixed binary ( 31 ) ;
```

```
stack_top =  0 ;
j         =  0 ;
more      = '1'b ;

do while ( more & error_code = 0 ) ;

   j = j + 1 ;

   g = neighbor ( f, j ) ;

   if ^ in_v ( g ) then

      if sum (h_y(g,*)*p(*)) - h_b(g) > 0.0 then do ;

         in_v    ( g )       = '1'b ;
         t_link  ( g )       =  v_header ;
         v_header            =  g ;

         stack_top           =  stack_top + 1 ;
         stack ( stack_top ) =  g ;
         end ;

      else do ;

         t            = create_and_link_new_facet ;
         t_link ( t ) = c_header ;
         c_header     = t ;

      end ;

   if j = d then do ;

      if stack_top = 0 then
         more      = '0'b ;
      else do ;
         f         =  stack ( stack_top ) ;
         stack_top =  stack_top - 1 ;
         j         =  0 ;
      end ;

   end ;

end ;

return ;
```

```
1create_and_link_new_facet : procedure ;

 /*-------------------------------------------------------------------
 |   Create a new facet and link to neighbor on the horizon.         |
 -------------------------------------------------------------------*/

  declare
     jj                          fixed binary ( 31 ),
     t                           fixed binary ( 31 ) ;

  t =  get_new_facet ;

  if error_code = 0 then do ;

     vertices ( t, * )       =  vertices ( f, * ) ;
     non_visible_facet ( t ) =  g ;
     visible_facet     ( t ) =  f ;

     do jj = j to d - 1 ;
        vertices ( t, jj ) = vertices ( t, jj+1 ) ;
     end ;

     vertices ( t, d ) = i ;

     neighbor ( t, d ) = g ;

     jj                 = find_common_subfacet ( t, g ) ;
     neighbor ( g, jj ) = t ;

     call create_h ( t ) ;

     if proc_type = 1  &  i < n  &  k = 0  then
     if sum (h_y(t,*)*s(i+1,*)) - h_b(t) > 0.0 then k = t ;

  end ;

 return ( t ) ;

 end create_and_link_new_facet ;

 end traverse ;
```

```
1interlink : procedure ;

 /*-------------------------------------------------------------------
 |   Interlink the new facets of the cap with each other.            |
 -------------------------------------------------------------------*/

  declare
     c_size                       fixed binary ( 31 ),
     j                            fixed binary ( 31 ),
     t                            fixed binary ( 31 ),

     /* the variables below are used by rotate.     */

     a                            fixed binary ( 31 ),
     b                            fixed binary ( 31 ),
     g                            fixed binary ( 31 ),
     h                            fixed binary ( 31 ),
     l                            fixed binary ( 31 ),
     phi ( subsubfacet_size )     fixed binary ( 31 ),
     x                            fixed binary ( 31 ),
     w                            fixed binary ( 31 ) ;

  c_size = 0 ;
  t      = c_header ;

  do while ( t ^= 0 ) ;

     c_size = c_size + 1 ;

     do j = 1 to d-1 ;

        if neighbor ( t, j ) = 0 then call rotate ;

     end ;

     t = t_link ( t ) ;

  end ;

  m  = m + c_size ;

 return ;
```

```
1rotate : procedure ;

 /*-------------------------------------------------------------------
 |   Rotate around a subsubfacet phi.                                 |
 -------------------------------------------------------------------*/

  do l = 1 to j-1 ;
     phi ( l ) = vertices ( t, l ) ;
  end ;

  do l = j to d-2 ;
     phi ( l ) = vertices ( t, l+1 ) ;
  end ;

  g = non_visible_facet ( t ) ;
  h = visible_facet     ( t ) ;

  do while ( in_v ( h ) ) ;

     a = 1 ;
     b = 1 ;
     x = 0 ;

     do while ( b <= d-2 & x = 0 ) ;

        if vertices ( h, a ) = phi ( b ) then do ;
           a = a + 1 ;
           b = b + 1 ;
           end ;
        else
           if neighbor ( h, a ) ^= g then
              x = neighbor ( h, a ) ;
           else
              a = a + 1 ;

     end ;

     do while ( x = 0 ) ;
        if neighbor ( h, a ) ^= g then
           x = neighbor ( h, a ) ;
        else
           a = a + 1 ;
     end ;

     g = h ;
     h = x ;

  end ;
```

```
 l                 = find_common_subfacet ( g, h ) ;
 w                 = neighbor ( h, l ) ;
 neighbor ( t, j ) = w ;

 l                 = find_common_subfacet ( t, w ) ;
 neighbor ( w, l ) = t ;

return ;

end rotate ;

end interlink ;
```

```
1find_common_subfacet : procedure ( f, g ) ;

 /*--------------------------------------------------------------------
 |   Find j such that sbf^F_j is shared by F and G.                    |
 --------------------------------------------------------------------*/

  declare
     a                            fixed binary ( 31 ),
     b                            fixed binary ( 31 ),
     f                            fixed binary ( 31 ),
     g                            fixed binary ( 31 ),
     j                            fixed binary ( 31 ) ;

  j = 0 ;
  a = 1 ;
  b = 1 ;

  do while ( j = 0 ) ;

     if vertices ( f, a ) = vertices ( g, b ) then do ;
        a = a + 1 ;
        b = b + 1 ;
        end ;
     else
        if vertices ( f, a ) < vertices ( g, b ) then
           a = a + 1 ;
        else
           j = b ;

     if a > d then j = d ;

  end ;

 return ( j ) ;

 end find_common_subfacet ;
```

```
1two_flat : procedure returns ( fixed binary ( 31 ) ) ;

 /*--------------------------------------------------------------------
 |   Find a first visible facet by 2-flat intersection, defined by   |
 |   p, the centroid of the first facet in P, and the neighboring    |
 |   facet G of F closest to p. The triangle defined by these three  |
 |   points is then shifted to ensure it passes through the subfacet |
 |   shared by F and G, and then made into an isoceles.              |
 --------------------------------------------------------------------*/

  declare
     a ( d2, d2 )                float binary ( 53 ),
     b ( d2 )                    float binary ( 53 ),
     b_save ( d2 )               float binary ( 53 ),
     d_save                      fixed binary ( 31 ),
     det                         float binary ( 53 ),
     distance                    float binary ( 53 ),
     distance_t                  float binary ( 53 ),
     f                           fixed binary ( 31 ),
     float                       builtin,
     g                           fixed binary ( 31 ),
     h                           fixed binary ( 31 ),
     half_path_switch            bit ( 1 ),
     ii                          fixed binary ( 31 ),
     intersect_switch            bit ( 1 ),
     j                           fixed binary ( 31 ),
     jj                          fixed binary ( 31 ),
     l                           fixed binary ( 31 ),
     last_angle                  float binary ( 53 ),
     last_last_angle             float binary ( 53 ),
     llast_last_angle            float binary ( 53 ),
     last_f                      fixed binary ( 31 ),
     llast_f                     fixed binary ( 31 ),
     last_g                      fixed binary ( 31 ),
     llast_g                     fixed binary ( 31 ),
     length1                     float binary ( 53 ),
     length2                     float binary ( 53 ),
     length                      float binary ( 53 ),
     mod                         builtin,
     more                        bit ( 1 ),
     next_angle                  float binary ( 53 ),
     next_facet                  fixed binary ( 31 ),
     q1  ( d )                   float binary ( 53 ),
     q1t ( d )                   float binary ( 53 ),
     q2  ( d )                   float binary ( 53 ),
     q2t ( d )                   float binary ( 53 ),
     q3  ( d )                   float binary ( 53 ),
     start_facet                 fixed binary ( 31 ),
```

```
   sum                          builtin,
   sqrt                         builtin,
   v1 ( d )                     float binary ( 53 ),
   v2 ( d )                     float binary ( 53 ) ;

f            = b_link ( 1 ) ;         /* First facet in P. */

if sum (h_y(f,*)*p(*)) - h_b(f) > 0.0 then return ( f ) ;

jj       = 1 ;
g        = neighbor ( f, jj ) ;
distance = (sum(h_y(g,*) * p(*)) - h_b(g)) / h_length(g) ;

do j = 2 to d ;

   g          = neighbor ( f, j ) ;
   distance_t = (sum(h_y(g,*) * p(*)) - h_b(g)) / h_length(g) ;

   if distance_t > distance then do ;
      distance = distance_t ;
      jj       = j ;
   end ;

end ;

g            = neighbor ( f, jj ) ;

    /* Find the midpoints of f, g, and their common subfacet. */

q1t ( * ) = 0.0 ;
q2t ( * ) = 0.0 ;
q3  ( * ) = 0.0 ;

do j = 1 to d ;
   q1t ( * ) = q1t ( * ) + s ( vertices ( f, j ), * ) ;
   q2t ( * ) = q2t ( * ) + s ( vertices ( g, j ), * ) ;
end ;

do j = 1 to d-1 ;
   l        = mod ( jj + j - 1, d ) + 1 ;
   q3 ( * ) = q3 ( * ) + s ( vertices ( f, l ), * ) ;
end ;

q1t ( * ) = q1t ( * ) / float ( d ) ;
q2t ( * ) = q2t ( * ) / float ( d ) ;
q3  ( * ) = q3  ( * ) / float ( d-1 ) ;

     /* Set q1 and q2. Make triangle (p,q1,q2) an isoceles. */
```

```
q1 ( * )  = q3 ( * ) + 3 * ( q2t ( * ) - q1t ( * ) ) ;
q2 ( * )  = q3 ( * ) - 3 * ( q2t ( * ) - q1t ( * ) ) ;
q1t ( * ) =  q1 ( * ) - p ( * ) ;
q2t ( * ) =  q2 ( * ) - p ( * ) ;

length1 = sqrt ( sum ( q1t ( * ) * q1t ( * ) ) ) ;
length2 = sqrt ( sum ( q2t ( * ) * q2t ( * ) ) ) ;

if length1 > length2 then
   q2 ( * ) = p ( * ) + ( length1 / length2 ) * q2t ( * ) ;
else do ;
   q1 ( * ) = p ( * ) + ( length2 / length1 ) * q1t ( * ) ;
end ;

do ii = 1 to d ;
   b_save ( ii ) = 0.0 ;
end;

b_save ( d+1 ) = 1.0 ;
b_save ( d+2 ) = 1.0 ;

last_f           = f ;
llast_f          = f ;
last_g           = g ;
llast_g          = g ;
start_facet      = f ;
last_angle       = 2.0 ;
last_last_angle  = last_angle ;
llast_last_angle = last_angle ;
half_path_switch = '1'b;

v1 ( * ) = q3 ( * ) - p ( * ) ;
length   = sqrt ( sum ( v1 ( * ) * v1 ( * ) ) ) ;
v1 ( * ) = v1 ( * ) / length ;

   /* Walk around 2-flat path. */

do while ( error_code = 0  &  g ^= start_facet &
           half_path_switch  &
           sum (h_y(g,*)*p(*)) - h_b(g) < 0.0  ) ;

   next_facet   = 0 ;

   do ii = 1 to d while ( next_facet = 0 ) ;

      h = neighbor ( g, ii ) ;
```

```
   if h ^= f then do ;

      do j = 1 to d-1 ;
         l = mod ( ii + j - 1, d ) + 1 ;
         do jj = 1 to d ;
            a ( jj, j ) = s ( vertices ( g, l ), jj ) ;
         end ;
         a ( d+1, j ) = 1.0 ;
         a ( d+2, j ) = 0.0 ;
      end ;

      do j = 1 to d ;
         a ( j, d   ) = -p  ( j ) ;
         a ( j, d+1 ) = -q1 ( j ) ;
         a ( j, d+2 ) = -q2 ( j ) ;
      end ;

      do j = d to d+2 ;
         a ( d+1, j ) = 0.0 ;
         a ( d+2, j ) = 1.0 ;
      end ;

      d_save  = d + 2 ;
      b ( * ) = b_save ( * ) ;

      call lneqs ( b, det, a, d_save ) ;

      if det = 0.0 then do ;
         error_code = 2 ;
         end ;
      else do ;
         intersect_switch = '1'b ;
         do j = 1 to d-1 ;
            if b ( j ) < 0.0 then intersect_switch = '0'b ;
         end ;
         if intersect_switch then next_facet = h ;
      end ;

   end ;

end ;

if next_facet = 0 then error_code = 2 ;

if error_code = 0 then do ;
   llast_f = last_f ;
   last_f  = f ;
   f       = g ;
```

```
      llast_g = last_g ;
      last_g  = g ;
      g       = next_facet ;

      /* Determine if path has gone more than half way around. */

      v2 ( * )   = ( b ( d   ) * p  ( * ) +
                     b ( d+1 ) * q1 ( * ) +
                     b ( d+2 ) * q2 ( * ) ) ;
      v2 ( * )   = v2 ( * ) - p ( * ) ;

      length     = sqrt ( sum ( v2 ( * ) * v2 ( * ) ) ) ;
      v2 ( * )   = v2 ( * ) / length ;
      next_angle = sum ( v1 ( * ) * v2 ( * ) )  ;
      if next_angle <= last_last_angle then do ;
         llast_last_angle = last_last_angle ;
         last_last_angle  = last_angle ;
         last_angle       = next_angle ;
         end ;
      else do ;
         half_path_switch = '0'b ;
      end ;

   end ;

 end ;

 if error_code ^= 0  |  ^half_path_switch  |
    sum (h_y(g,*)*p(*)) - h_b(g) < 0.0 then do ;
    g = 0 ;
 end ;

 return ( g ) ;

end two_flat ;
```

```
1delete_v : procedure ;

 /*--------------------------------------------------------------------
 |   Delete the set of visible facets V.                               |
 --------------------------------------------------------------------*/

  declare
     f                           fixed binary ( 31 ),
     v_size                      fixed binary ( 31 ) ;

  v_size = 0 ;
  f      = v_header ;

  do while ( f ^= 0 ) ;

     f_link ( b_link ( f ) ) = f_link ( f ) ;
     b_link ( f_link ( f ) ) = b_link ( f ) ;
     f_link ( f )            = avail ;
     avail                   = f ;

     v_size                  = v_size + 1 ;

     f                       = t_link ( f ) ;

  end ;

  m  = m - v_size ;

 return ;

 end delete_v ;
```

```
1convex_initialization : procedure ;
 /*--------------------------------------------------------------------
 |   Initialize S_index. If proc_type=1 sort S lexicographically.    |
 |   Create the initial simplex and centroid of the simplex.         |
 --------------------------------------------------------------------*/
  declare
     d3                          fixed binary ( 31 ),
     float                       builtin,
     j                           fixed binary ( 31 ),
     k                           fixed binary ( 31 ),
     least_p ( d, d )            float binary ( 21 ),
     max_p   ( d, d )            float binary ( 21 ) ;

  error_code       = 0 ;
  subsubfacet_size = d - 2 ;
  d2               = d + 2 ;

  do j = 1 to n ;
     s_index        ( j )  =  j ;
  end ;

  if proc_type = 1 then call sort1 ( s, s_index, 1, n, d ) ;

  centroid ( * ) = 0.0 ;

  do j = 1 to d + 1 ;
     centroid ( * ) = centroid ( * ) + s ( j, * ) ;
  end ;

  centroid ( * ) = centroid ( * ) / float ( d + 1 ) ;

       /* Initialize linked list of facets P. */

  m                 =  0 ;
  f_link ( 1 )      =  1 ;
  b_link ( 1 )      =  1 ;
  link_init_counter =  1 ;
  link_full         = '0'b ;
  avail             =  0 ;

  d3 = d + 1 ;

  call create_simplex ;

  m  =  d+1 ;

 return ;
```

```
1Create_simplex : procedure ;
 /*-------------------------------------------------------------------
 |   Create initial simplex from first d+1 points of S.              |
 -------------------------------------------------------------------*/
  declare
     f                            fixed binary ( 31 ),
     f_index_list ( d3 )          fixed binary ( 31 ),
     i                            fixed binary ( 31 ),
     j                            fixed binary ( 31 ),
     l                            fixed binary ( 31 ),
     sum                          builtin,
     temp                         fixed binary ( 31 ) ;

  k = 0 ;

  do i = 1 to d + 1 ;
     f_index_list ( i ) = get_new_facet ;
  end ;

  do i = 1 to d + 1 ;

     l  =  0 ;
     f  =  f_index_list ( i ) ;

     do j = 1 to d + 1 ;
        if  i ^=  j then do ;
           l                  =  l + 1 ;
           vertices ( f, l )  =  j ;
           neighbor ( f, l )  =  f_index_list ( j ) ;
        end ;
     end ;

     call create_h ( f ) ;

          /* If preprocessing case, set first visible facet K. */

     if proc_type = 1  &  k = 0  &  n > d+1  then
     if sum (h_y(f,*)*s(d+2,*)) - h_b(f) > 0.0 then
        k = f ;

  end ;

  return ;

 end create_simplex ;

 end convex_initialization ;

1get_new_facet : procedure returns ( fixed binary ( 31 ) ) ;
```

```
/*--------------------------------------------------------------------
|   Get a new facet from available storage.                          |
--------------------------------------------------------------------*/

 declare
    last_node                    fixed binary ( 31 ),
    t                            fixed binary ( 31 ) ;

 if ^ link_full then do ;
    link_init_counter = link_init_counter + 1 ;
    t                 = link_init_counter ;
    if link_init_counter >= p_size then link_full = '1'b ;
    end ;
 else
    if avail ^= 0 then do ;
       t     = avail ;
       avail = f_link ( avail ) ;
       end ;
    else do ;
       error_code = 1 ;
       t          = 1 ;
    end ;

 last_node            = b_link ( 1 ) ;
 f_link ( t )         = 1 ;
 b_link ( t )         = last_node ;
 f_link ( last_node ) = t ;
 b_link ( 1 )         = t ;

 neighbor ( t, * )    =  0 ;
 in_v ( t )           = '0'b;

return ( t ) ;

end get_new_facet ;
```

```
1create_h : procedure ( f ) ;
 /*--------------------------------------------------------------------
 |   Create the plane equation for facet F.                           |
 --------------------------------------------------------------------*/
  declare
     a ( d, d )                  float binary ( 53 ),
     b ( d )                     float binary ( 53 ),
     dd                          fixed binary ( 31 ),
     det                         float binary ( 53 ),
     f                           fixed binary ( 31 ),
     j                           fixed binary ( 31 ),
     sqrt                        builtin,
     sum                         builtin ;

  do j = 1 to d ;
     a ( j, * )  =  s ( vertices ( f,j ), * ) ;
  end ;

  b ( * )   =  1.0 ;
  h_b ( f ) =  1.0 ;
  dd        = d ;

  call lneqs ( b, det, a, dd ) ;

  if det = 0.0 then

     error_code = 3 ;

  else do ;

     h_y ( f, * )  =  0.0 ;
     h_y ( f, * )  =  b ( * ) ;

     h_length(f) = sqrt ( sum ( h_y(f,*) ** 2 ) ) ;

     if sum (h_y(f,*) * centroid(*)) - h_b(f) > 0.0 then do ;

        h_y ( f, * )  =  -h_y ( f, * ) ;
        h_b ( f )     =  -h_b ( f ) ;

     end ;

  end ;

  return ;

 end create_h ;
```

```
1sort1 : procedure ( a, b, s, e, d ) ;

 /*--------------------------------------------------------------------
 |   Name:                                                            |
 |      Quicksort                                                     |
 |   Description                                                      |
 |      This subroutine implement the quicksort algorithm to          |
 |      sort a floating point matrix A in lexicographic order.        |
 |      As well, array B (probably an index array) is sorted.         |
 |   Reference                                                        |
 |      Sedgewick, R. (1978) "Implementing Quicksort Algorithms",     |
 |      Communications of the ACM, Volume 21, Number 10, October.     |
 --------------------------------------------------------------------*/

  declare
     a ( *, * )                  float binary ( 21 ),
     b ( * )                     fixed binary ( 31 ),
     d                           fixed binary ( 31 ),
     done                        bit ( 1 ),
     e                           fixed binary ( 31 ),
     i                           fixed binary ( 31 ),
     j                           fixed binary ( 31 ),
     k                           fixed binary ( 31 ),
     l                           fixed binary ( 31 ),
     m                           fixed binary ( 31 ),
     lstack ( 100 )              fixed binary ( 31 ),
     r                           fixed binary ( 31 ),
     rstack ( 100 )              fixed binary ( 31 ),
     s                           fixed binary ( 31 ),
     stackp                      fixed binary ( 31 ),
     tempa ( d )                 float binary ( 21 ),
     tempb                       fixed binary ( 31 ),
     trunc                       builtin ,
     v ( d )                     float binary ( 21 ),
     x                           fixed binary ( 31 );

  l      =  s ;
  r      =  e ;
  m      = 10 ;
  done   = '0'b ;
  stackp =  0 ;

  if  r - l + 1 <= m  then done = '1'b;

  do while ( ^ done ) ;

     x              = trunc ( ( l + r ) / 2 ) ;
```

```
tempa          = a ( x, * ) ;
a ( x, * )     = a ( l + 1, * ) ;
a ( l + 1, * ) = tempa ;

tempb       = b ( x ) ;
b ( x  )    = b ( l + 1 ) ;
b ( l + 1 ) = tempb ;

do k = 1 to d while ( a ( l + 1, k )  =  a ( r, k ) ) ;
end ;

if  a ( l + 1, k )  >  a ( r, k )  then do;

   tempa          = a ( l + 1, * ) ;
   a ( l + 1, * ) = a ( r, * ) ;
   a ( r, * )     = tempa ;

   tempb       = b ( l + 1 ) ;
   b ( l + 1 ) = b ( r ) ;
   b ( r )     = tempb ;

end ;

do k = 1 to d while ( a ( l, k ) = a ( r, k ) ) ;
end ;

if  a ( l, k ) > a ( r, k )  then do ;
   tempa       = a ( l, * ) ;
   a ( l, * )  = a ( r, * ) ;
   a ( r, * )  = tempa ;
   tempb       = b ( l ) ;
   b ( l )     = b ( r ) ;
   b ( r )     = tempb ;
end ;

do k = 1 to d while ( a ( l + 1, k ) = a ( l, k ) ) ;
end ;

if  a ( l + 1, k ) > a ( l, k )  then do;
   tempa          = a ( l + 1, * ) ;
   a ( l + 1, * ) = a ( l, * ) ;
   a ( l, * )     = tempa ;
   tempb          = b ( l + 1 ) ;
   b ( l + 1 )    = b ( l ) ;
   b ( l )        = tempb ;
end ;

i = l + 1 ;
```

```
j = r ;
v = a ( l, * ) ;

do while ( j >= i ) ;

   i = i + 1 ;

   do k = 1 to d while ( a ( i, k ) = v ( k ) ) ;
   end ;

   do while ( a ( i, k ) < v ( k ) ) ;
      i = i + 1 ;

      do k = 1 to d while ( a ( i, k ) = v ( k ) ) ;
      end ;

   end ;

   j = j - 1 ;

   do k = 1 to d while ( a ( j, k ) = v ( k ) ) ;
   end ;

   do while ( a ( j, k ) > v ( k ) ) ;
      j = j - 1 ;

      do k = 1 to d while ( a ( j, k ) = v ( k ) ) ;
      end ;

   end ;

   if  j < i then goto exit_loop1 ;

   tempa       = a ( i, * ) ;
   a ( i, * )  = a ( j, * ) ;
   a ( j, * )  = tempa ;
   tempb       = b ( i ) ;
   b ( i )     = b ( j ) ;
   b ( j )     = tempb ;

end ;
exit_loop1:

tempa       = a ( l, * ) ;
a ( l, * )  = a ( j, * ) ;
a ( j, * )  = tempa ;
tempb       = b ( l ) ;
b ( l )     = b ( j ) ;
```

```
   b ( j )     = tempb ;

   if  j - l <= m  &  r - i + 1 <= m  then do ;

      if  stackp = 0  then do ;
         done = '1'b ;
         end ;
      else do ;
         l      = lstack ( stackp ) ;
         r      = rstack ( stackp ) ;
         stackp = stackp - 1 ;
      end ;
      end ;

   else do ;

      if  j - l <= m  |  r - i + 1 <= m  then do ;

         if  j - l >= r - i + 1  then do ;
            r = j - 1 ;
            end ;
         else do ;
            l = i ;
         end ;
         end ;

      else do ;

         stackp = stackp + 1 ;

         if j - l >= r - i + 1 then do;
            lstack ( stackp ) = l ;
            rstack ( stackp ) = j - 1 ;
            l                 = i ;
            end ;
         else do ;
            lstack ( stackp ) = i ;
            rstack ( stackp ) = r ;
            r                 = j - 1 ;
         end ;

      end ;

   end ;

end ;

do i = 2 to e ;
```

```
    tempa = a ( i, * ) ;
    tempb = b ( i ) ;
    j     = i - 1 ;

    do k = 1 to d while ( a ( j, k ) = tempa ( k ) ) ;
    end ;

    do while ( a ( j, k ) > tempa ( k ) ) ;
       a ( j+1, * ) = a ( j, * ) ;
       b ( j+1 )    = b ( j ) ;
       j            = j - 1 ;
       if j <= 0 then goto exit_loop2 ;

       do k = 1 to d while ( a ( j, k ) = tempa ( k ) ) ;
       end ;

    end ;
    exit_loop2 :

    a ( j+1, * ) = tempa ;
    b ( j+1 )    = tempb ;

 end ;

return ;

end sort1;

end convex ;
```

Appendix B

Tables of Data

B.1 Preprocessing–Distribution A

d	n	m	t	$rt(d-1)/2$	cputime
3	250	496	2007	6536	0.543416
3	500	996	4446	14792	1.224330
3	750	1496	7058	23740	1.950036
3	1000	1996	9732	32936	2.763071
3	1250	2496	12457	42336	3.417683
3	1500	2996	15329	52324	4.248258
3	1750	3496	18293	62680	5.034505
3	2000	3996	21179	72724	5.840454
3	2250	4496	24102	82916	6.646776
3	2500	4996	27039	93164	7.447639
3	2750	5496	30119	103984	8.305676
3	3000	5996	33150	114608	9.169383
4	250	1560	7685	37153	2.745369
4	500	3196	18461	90611	6.711613
4	750	4891	30279	149107	10.937673
4	1000	6496	42457	209703	15.460477
4	1250	8214	55167	272845	20.132629
4	1500	9896	68519	339025	25.038315
4	1750	11591	82613	409655	29.919739
4	2000	13289	96627	479563	35.079025
4	2250	14957	110677	549882	40.350449
4	2500	16640	124509	618770	45.862183
4	2750	18287	139277	692929	50.744171
4	3000	20012	153883	765829	56.330063

d	n	m	t	$rt(d-1)/2$	cputime
5	300	7378	40389	262314	18.791428
5	600	15744	103619	677680	48.910309
5	900	24434	172073	1127636	80.405075
5	1200	33524	248938	1634921	117.762329
5	1500	42384	329709	2168329	156.817886
5	1800	51326	412743	2717369	195.172089
5	2100	60324	500935	3301065	237.037643
6	200	18136	90813	736153	52.722855
6	400	44054	265447	2164813	155.539566
6	600	71346	480749	3931811	283.673340
6	800	99142	713329	5839879	421.298340
6	1000	130285	971865	7961037	581.956299
6	1100	145701	1105301	9058099	530.039551
7	50	6610	20979	200704	14.681265
7	100	24094	96936	937929	68.439758
7	120	32314	139162	1349604	99.021515
7	150	46244	211837	2059613	150.767731
7	200	72616	361909	3526119	259.167236
7	250	101114	537473	5247616	387.031006
7	300	132534	743447	7266659	538.060059
7	350	164398	966930	9462136	703.233887
8	20	1167	2305	24676	1.910583
8	40	10573	31127	347430	26.405563
8	60	28920	96751	1087860	82.418610
8	80	52996	201611	2277690	173.387970
8	100	82769	336955	3815351	292.082520
8	120	116980	510287	5790055	442.375977

d	n	m	t	$rt(d-1)/2$	cputime
9	10	10	10	0	0.006658
9	20	1780	3088	37131	3.033577
9	30	9260	20320	254504	20.342789
9	40	25898	70059	895134	57.613419
9	50	49676	154761	1991673	128.336334
9	60	84600	278690	3593019	231.765076
10	13	84	105	1126	0.092323
10	18	1231	1933	25724	1.810435
10	23	5033	9591	133339	9.185479
10	28	13296	30075	426825	29.143326
10	33	27255	66233	945960	64.449890
10	38	49660	128621	1848796	125.584793
10	43	77314	209909	3024909	205.908890

B.2 Preprocessing–Distribution B

d	n	v	m	t	$rt(d-1)/2$	cputime
3	250	59	114	1599	5192	0.452103
3	500	86	168	3353	11037	0.955377
3	750	104	204	5101	16886	1.460204
3	1000	121	238	6871	22766	2.009170
3	1250	136	268	8608	28453	2.474198
3	1500	141	278	10391	34453	2.964301
3	1750	156	308	12163	40436	3.495768
3	2000	158	312	13856	46085	3.987372
3	2250	165	326	15611	51943	4.499757
3	2500	168	332	17414	57960	5.038630
3	2750	180	356	19179	63940	5.523833
3	3000	183	362	21017	70142	6.049145
4	250	140	760	6395	31262	2.316963
4	500	214	1189	14215	70092	5.196110
4	750	270	1554	22353	110723	8.228250
4	1000	321	1847	30043	149251	11.151294
4	1250	346	1991	38343	190628	14.354426
4	1500	384	2244	46961	233804	17.486008
4	1750	422	2448	55265	275340	20.568893
4	2000	461	2671	63981	318951	23.916840
4	2250	490	2846	72853	363531	26.877502
4	2500	516	2997	81359	406099	30.351440
4	2750	551	3211	90067	449754	33.489288
4	3000	588	3433	99009	494431	36.875534

d	n	v	m	t	$rt(d-1)/2$	cputime
5	300	230	4564	34962	228514	16.349777
5	600	391	8230	83776	551276	39.364044
5	900	519	11166	135111	891118	63.648026
5	1200	626	13672	187901	1240920	89.348999
5	1500	745	16256	245184	1620883	117.161957
5	1800	844	18380	300824	1990639	143.531052
5	2100	930	20340	357494	2366603	172.412323
5	2400	1020	22474	413259	2736260	199.320847
6	200	195	15523	86643	703768	50.346786
6	400	362	32469	232765	1901768	136.099869
6	600	526	49374	420485	3448334	248.721420
6	800	677	66271	616039	5057182	365.856689
6	1000	810	80776	828633	6811090	506.701172
6	1200	947	94686	1032433	8490308	497.943604
6	1400	1063	107509	1254213	10320815	608.731445
6	1600	1189	121247	1478645	12172286	715.531494
7	50	50	6274	20149	192610	14.068970
7	150	150	42612	203319	1975734	144.201508
7	250	249	89902	515051	5033958	369.023926
7	350	347	138110	881886	8633461	635.695313
7	400	395	163540	1085299	10630730	636.922607
8	20	20	1153	2299	24684	1.894699
8	40	40	10247	30679	343148	26.105667
8	60	60	27309	96549	1086473	82.606674
8	80	80	49154	192865	2177189	165.814514
8	100	100	77843	320619	3628435	277.994629
8	120	120	110371	488999	5546534	424.716064

d	n	v	m	t	$rt(d-1)/2$	cputime
9	10	10	10	10	0	0.006631
9	20	20	1734	3105	37247	3.054101
9	30	30	9384	21421	269418	21.443268
9	40	40	26560	70578	900389	58.057816
9	50	50	52348	151170	1935393	124.732651
9	60	60	87846	272392	3502585	225.889145
10	13	13	84	107	1162	0.096219
10	18	18	1214	1923	25482	1.819850
10	23	23	5226	9115	124635	8.707877
10	28	28	13094	26337	369387	25.454758
10	33	33	26610	59983	851104	58.148590
10	38	38	47281	123077	1770392	120.009735
10	43	43	72831	207745	2999508	203.585541

B.3 Online—Distribution A

d	n	m	t	$rt(d-1)/2$	w	cputime
3	300	596	1659	4844	3493	1.241896
3	600	1196	3450	10208	9981	3.229308
3	900	1796	5252	15616	18181	5.657092
3	1200	2396	7089	21164	27852	8.451158
3	1500	2996	8932	26736	38748	11.581150
3	1800	3596	10740	32168	50697	14.940383
3	2100	4196	12550	37608	64377	18.745392
3	2400	4796	14359	43044	78504	22.638809
3	2700	5396	16158	48440	94611	27.069321
3	3000	5996	17978	53920	111547	31.497559
4	250	1560	5633	26538	2454	2.945844
4	500	3196	12051	57073	6468	6.821815
4	750	4891	18489	87563	11136	10.988282
4	1000	6496	25001	118606	16875	15.669538
4	1250	8214	31643	150123	23318	20.664337
4	1500	9896	38453	182619	30041	25.812790
4	1750	11591	45191	214672	37292	30.916367
4	2000	13289	51813	246160	44884	36.367722
4	2500	16640	65029	308992	61379	47.806137
4	3000	20012	78361	372508	78846	59.642883

d	n	m	t	$rt(d-1)/2$	w	cputime
5	300	7378	28969	184534	3091	15.084981
5	600	15744	66594	426477	8423	35.598633
5	900	24434	106232	681744	14715	58.051895
5	1200	33524	146978	943901	21637	81.347794
5	1500	42384	188669	1212495	29550	105.831482
5	1800	51326	230905	1484715	37534	130.833496
5	2100	60324	274271	1764630	45882	155.623627
5	2400	69268	318081	2047516	54734	183.067596
5	2700	78352	361651	2328732	63871	209.402374
5	3000	87588	404962	2607842	73638	235.348267
6	200	18136	74385	597674	1673	44.460373
6	400	44054	195985	1580393	4952	117.849823
6	600	71346	331619	2678345	8582	200.793365
6	800	99142	480137	3883016	12936	293.006836
6	1000	130285	629925	5094030	17710	388.180908
6	1100	145701	707409	5722396	20205	350.739258
7	50	6610	20645	197377	183	14.933167
7	150	46244	185440	1793876	1091	134.506302
7	250	101114	444147	4307495	2478	321.758789
7	350	164398	742458	7204490	4186	538.721191
8	20	1167	2201	23268	21	1.830634
8	40	10573	26901	296603	100	22.897263
8	60	28920	87231	973810	208	74.741302
8	80	52996	175545	1970722	351	151.859161
8	100	82769	294069	3310222	518	256.305176
8	120	116980	437191	4929963	777	384.577881

d	n	m	t	$rt(d-1)/2$	w	cputime
9	10	10	10	0	0	0.006309
9	20	1780	3294	40213	19	3.280164
9	30	9260	20710	259715	41	20.946259
9	40	25898	64680	819656	80	65.850830
9	50	49676	141407	1805630	162	146.061829
9	60	84600	254083	3256009	227	263.967285
10	20	2498	4199	57086	14	4.004929
10	24	6367	12543	175468	25	12.153897
10	28	13296	27485	387677	42	26.701096
10	36	38850	94019	1344909	76	92.007553
10	40	61597	154283	2213666	87	150.984787
10	44	83391	227551	3280842	98	224.667068
10	46	95785	269585	3892733	110	266.585205
10	47	103705	289019	4170524	111	284.633545

B.4 Online—Distribution B

d	n	v	m	t	$rt(d-1)/2$	w	cputime
3	300	69	134	637	1919	2915	0.806791
3	600	95	186	993	3043	7253	1.878896
3	900	115	226	1190	3647	11970	3.012170
3	1200	131	258	1410	4334	17231	4.320048
3	1500	141	278	1569	4828	22614	5.616176
3	1800	155	306	1729	5323	28433	7.032525
3	2100	160	316	1903	5900	34138	8.389588
3	2400	163	322	1998	6212	39971	9.745375
3	2700	177	350	2124	6599	45854	11.151486
3	3000	183	362	2256	7024	51790	12.654961
4	250	140	760	3221	15172	2653	2.160418
4	500	214	1189	5477	25922	6501	4.486297
4	750	270	1554	7697	36571	10857	6.993716
4	1000	321	1847	9601	45662	15610	9.571816
4	1250	346	1991	11197	53461	20743	12.202708
4	1500	384	2244	12735	60850	26232	14.940354
4	1750	422	2448	14003	66920	31796	17.615082
4	2000	461	2671	15241	72851	37687	20.493759
4	2250	490	2846	16529	79045	43960	23.440979
4	2500	516	2997	17711	84799	50103	26.459335
4	2750	551	3211	19001	90994	56174	29.235458
4	3000	588	3433	20159	96433	62411	32.199295

d	n	v	m	t	$rt(d-1)/2$	w	cputime
5	300	230	4564	22844	146602	3081	12.438333
5	600	391	8230	44411	285661	8422	25.996475
5	900	519	11166	64509	415769	14621	39.245941
5	1200	626	13672	79663	513510	21515	50.200943
5	1500	745	16256	96873	625196	28954	63.107910
5	1800	844	18380	113354	732363	36575	75.418076
5	2100	930	20340	128467	830474	44411	87.613846
5	2400	1020	22474	143992	931058	52883	100.390213
5	2700	1096	24260	156560	1012706	61643	111.682068
5	3000	1180	25782	166855	1079238	70885	122.667404
6	200	195	15523	64555	517605	1658	38.808563
6	400	362	32469	154465	1245358	5044	94.302505
6	600	526	49374	259827	2101140	8987	159.467392
6	800	677	66271	360307	2916214	13190	222.528656
6	1000	810	80776	450473	3647161	18039	285.375488
6	1200	947	94686	543361	4401941	23153	276.147461
6	1400	1063	107509	625801	5070517	28306	320.179932
6	1600	1189	121247	712809	5776299	33640	365.990234
7	50	50	6274	19043	181527	167	13.584641
7	150	150	42612	163858	1580408	1182	118.494812
7	250	249	89902	383554	3710932	2573	279.927490
7	350	347	138110	645931	6264269	4284	472.402832
7	400	395	163540	785757	7624358	5226	462.365234
8	20	20	1153	2247	23845	21	1.885442
8	40	40	10247	25353	279192	129	21.555115
8	60	60	27309	83137	928707	257	71.356918
8	80	80	49154	170159	1912281	492	147.271713
8	100	100	77843	290223	3271170	682	256.129639
8	120	120	110371	417245	4703536	861	363.381104

d	n	v	m	t	$rt(d-1)/2$	w	cputime
9	10	10	10	10	0	0	0.006267
9	20	20	1734	3227	39504	16	3.209914
9	30	30	9384	20189	252331	41	20.400818
9	40	40	26560	63737	805735	94	65.109177
9	50	50	52348	146883	1876707	159	150.302200
9	60	60	87846	255994	3277826	225	261.567871
10	14	14	168	221	2561	4	0.202362
10	18	18	1214	2091	28482	17	2.017103
10	22	22	4216	7607	105268	26	7.336512
10	26	26	9369	18963	266433	45	18.373535
10	30	30	18211	38763	547803	63	37.664398
10	34	34	30366	69481	988469	71	67.512741
10	38	38	47281	112979	1611951	85	109.992798
10	42	42	67098	177341	2547368	108	173.505081
10	46	46	94270	238143	3408047	133	232.763702

B.5 Cyclic Polytopes—Preprocessing

d	n	m	t	$rt(d-1)/2$	cputime
3	4	4	4	0	0.001030
3	5	6	8	10	0.002053
3	6	8	13	24	0.003512
3	8	12	26	64	0.007136
3	10	16	43	120	0.011638
3	12	20	64	192	0.018139
3	14	24	89	280	0.025338
3	16	28	118	384	0.032897
3	18	32	151	504	0.041984
3	20	36	188	640	0.051871
3	22	40	229	792	0.062946
3	24	44	274	960	0.076086
3	26	48	323	1144	0.089547
3	27	50	349	1242	0.098679
3	28	52	376	1344	0.106371
3	29	54	404	1450	0.113280
3	30	56	433	1560	0.120610
3	35	66	593	2170	0.166673
3	40	76	778	2880	0.218483
3	45	86	988	3690	0.276179
3	50	96	1223	4600	0.344103
3	55	106	1483	5610	0.419160
3	60	116	1768	6720	0.495610

d	n	m	t	$rt(d-1)/2$	cputime
4	5	5	5	0	0.001554
4	6	9	11	21	0.003616
4	7	14	19	51	0.006324
4	9	27	41	138	0.014135
4	11	44	71	261	0.024333
4	13	65	109	420	0.038004
4	15	90	155	615	0.053521
4	17	119	209	846	0.073394
4	19	152	271	1113	0.094908
4	21	189	341	1416	0.119319
4	23	230	419	1755	0.146569
4	25	275	505	2130	0.179451
4	27	324	599	2541	0.212464
4	28	350	649	2760	0.233201
4	29	377	701	2988	0.252488
4	30	405	755	3225	0.261692
4	35	560	1055	4545	0.365456
4	40	740	1405	6090	0.488134
4	45	945	1805	7860	0.629660
4	50	1175	2255	9855	0.787995

d	n	m	t	$rt(d-1)/2$	cputime
5	6	6	6	0	0.001980
5	7	12	15	45	0.006013
5	8	20	29	123	0.012332
5	10	42	76	410	0.032887
5	12	72	155	925	0.069315
5	14	110	274	1732	0.125261
5	16	156	441	2895	0.203850
5	18	210	664	4478	0.309138
5	20	272	951	6545	0.444201
5	22	342	1310	9160	0.615529
5	24	420	1749	12387	0.826264
5	26	506	2276	16290	1.075981
5	27	552	2575	18515	1.214857
5	28	600	2899	20933	1.360541
5	29	650	3249	23552	1.537246
6	7	7	7	0	0.002840
6	8	16	19	72	0.009097
6	9	30	39	202	0.020064
6	11	77	111	700	0.059635
6	13	156	239	1622	0.130934
6	15	275	439	3096	0.242777
6	17	442	727	5250	0.403492
6	19	665	1119	8212	0.624999
6	21	952	1631	12110	0.913770
6	23	1311	2279	17072	1.285749
6	25	1750	3079	23226	1.754661
6	26	2002	3541	26790	2.022758
6	27	2277	4047	30700	2.330220
6	28	2576	4599	34972	2.630606
6	29	2900	5199	39622	2.993076

d	n	m	t	$rt(d-1)/2$	cputime
7	8	8	8	0	0.003610
7	9	20	24	120	0.014097
7	10	40	54	370	0.034063
7	11	70	104	814	0.068408
7	12	112	181	1528	0.120396
7	13	168	293	2600	0.198245
7	14	240	449	4130	0.305663
7	15	330	659	6230	0.453205
7	16	440	934	9024	0.646744
7	17	572	1286	12648	0.897388
7	18	728	1728	17250	1.212815
8	9	9	9	0	0.004854
8	10	25	29	170	0.020524
8	11	55	69	540	0.052780
8	12	105	139	1219	0.110019
8	13	182	251	2339	0.202097
8	14	294	419	4055	0.339410
8	15	450	659	6545	0.540579
8	16	659	989	10020	0.810363

B.6 Cyclic Polytopes—Online

d	n	m	t	$rt(d-1)/2$	w	cputime
3	4	4	4	0	0	0.000990
3	5	6	7	6	2	0.002172
3	6	8	11	16	3	0.003088
3	8	12	23	52	8	0.006797
3	10	16	31	72	16	0.010500
3	12	20	35	76	18	0.011853
3	14	24	45	104	25	0.015534
3	16	28	58	144	33	0.020531
3	18	32	87	248	50	0.032309
3	20	36	67	156	85	0.035957
3	22	40	82	204	92	0.042650
3	24	44	106	288	98	0.048420
3	26	48	141	416	102	0.057017
3	27	50	164	502	104	0.065460
3	28	52	130	360	131	0.059807
3	29	54	151	438	150	0.069180
3	30	56	171	512	160	0.074983
3	35	66	145	378	214	0.082689
3	40	76	153	380	302	0.115612
3	45	86	245	718	359	0.137382
3	50	96	238	660	459	0.170282
3	55	106	261	722	510	0.175350
3	60	116	302	856	567	0.194608

d	n	m	t	$rt(d-1)/2$	w	cputime
4	5	5	5	0	0	0.001723
4	6	9	11	21	2	0.003841
4	7	14	19	51	2	0.006383
4	9	27	41	138	11	0.016736
4	11	44	71	261	13	0.025938
4	13	65	109	420	14	0.038923
4	15	90	155	615	24	0.057481
4	17	119	209	846	36	0.081409
4	19	152	271	1113	63	0.112305
4	21	189	341	1416	75	0.140079
4	23	230	419	1755	87	0.168936
4	25	275	505	2130	105	0.206749
4	27	324	599	2541	129	0.247895
4	28	350	649	2760	121	0.262470
4	29	377	701	2988	153	0.291356
4	30	405	755	3225	166	0.311846
4	35	560	1055	4545	253	0.443878
4	40	740	1405	6090	332	0.594186
4	45	945	1805	7860	438	0.770495
4	50	1175	2255	9855	483	0.965787

d	n	m	t	$rt(d-1)/2$	w	cputime
5	6	6	6	0	0	0.001987
5	7	12	14	36	2	0.005472
5	8	20	28	114	4	0.012373
5	10	42	67	333	6	0.030335
5	12	72	116	598	9	0.051666
5	14	110	183	980	18	0.084550
5	16	156	271	1497	17	0.121147
5	18	210	456	2772	23	0.208220
5	20	272	456	2497	66	0.229717
5	22	342	630	3609	44	0.290301
5	24	420	824	4858	39	0.374031
5	26	506	1140	7059	44	0.521688
5	27	552	1382	8825	44	0.638772
5	28	600	1171	6908	110	0.572909
5	29	650	1394	8504	101	0.670661
6	7	7	7	0	0	0.002567
6	8	16	19	72	1	0.009057
6	9	30	39	202	4	0.020544
6	11	77	111	700	5	0.059572
6	13	156	239	1622	12	0.132046
6	15	275	439	3096	18	0.247712
6	17	442	727	5250	17	0.402760
6	19	665	1119	8212	22	0.623799
6	21	952	1631	12110	33	0.911985
6	23	1311	2279	17072	42	1.282188
6	25	1750	3079	23226	41	1.725435
6	26	2002	3541	26790	41	1.987620
6	27	2277	4047	30700	42	2.273267
6	28	2576	4599	34972	40	2.576137
6	29	2900	5199	39622	45	2.924273

d	n	m	t	$rt(d-1)/2$	w	cputime
7	8	8	8	0	0	0.003614
7	9	20	23	105	1	0.013326
7	10	40	53	355	3	0.034004
7	11	70	92	649	6	0.060239
7	12	112	160	1243	5	0.106505
7	13	168	245	1968	6	0.163203
7	14	240	362	3000	14	0.247197
7	15	330	510	4310	12	0.346789
7	16	440	694	5954	18	0.478780
7	17	572	958	8479	17	0.657053
7	18	728	1354	12511	18	0.941994
7	19	910	1900	18251	19	1.350279
8	9	9	9	0	0	0.004789
8	10	25	29	170	1	0.020888
8	11	55	69	540	3	0.054424
8	12	105	139	1219	6	0.111768
8	13	182	251	2339	5	0.203203
8	14	294	419	4055	10	0.348711
8	15	450	659	6545	15	0.549171

Bibliography

[1] Avis, D.; Fukuda, K. (1991) *Pivoting Algorithms for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra*, Proc. 7th Symposium on Computational Geometry, ACM, June.

[2] Akl, S. G.; Toussaint, G. T. (1978) *A Fast Convex Hull Algorithm*, Info. Proc. Lett. 7, n. 5, 219-222.

[3] Barany, I.; Larman, D. G.; (1988) *Convex Bodies, Economic Cap Coverings, Random Polytopes*, Mathematika, 35(2):274-291, June.

[4] Brondsted, A. (1983) *An Introduction to Convex Polytopes*, Springer-Verlag, New York.

[5] Brown, K. Q. (1979) *Voronoi Diagrams from Convex Hulls,* Information Processing Letters, vol. 9, no. 5, 16 December.

[6] Buchta, C.; Muller, F.; Tichy, R. F. (1985) *Stochastical Approximation of Convex Bodies*, Math. Annal., 271,225-235.

[7] Chand, D. R.; Kapur, S. S. (1970) *An Algorithm for Convex Polytopes*, J.ACM, 17, 78-86.

[8] Clarkson, K.; Shor, P. (1989) *Applications of Random Sampling in Computational Geometry II*, 4:387-421.

[9] Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A.

[10] Dwyer, R. A. (1990) *Kinder, Gentler Average-Case Analysis for Convex Hulls and Maximal Vectors*, SIGACT News, 21, 64-71.

[11] Eddy, W. F. (1977) *Algorithm 523 CONVEX, A New Convex Hull Algorithm for Planar Sets*, Collected Algorithms from ACM, 523P1-523P6.

[12] Graham, R. L. (1972) *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*, Inf. Proc. Lett., 1, 132-133.

[13] Gupta, S. (1991) *Stewart's Algorithm for the Convex Hull in d-Dimensions,* CS4993 Report, Computer Science, U.N.B.

[14] Jarvis, R. A. (1973) *On the Identification of the Convex Hull of a Finite Set iof Points in the Plane*, Info. Proc. Lett. 2, no. 1, 18-21.

[15] Kirkpatrick, D. G.; Seidel, R. (1986) *The Ultimate Planar Convex Hull Algorithm?*, SIAM J. Computing, 15:287-299.

[16] Graham, R; Knuth, D.; Patashnik, O. (1989) *Concrete Mathematics*, Addison-Wesley, p. 438

[17] Preparata, F. P.; Hong, S. J. (1977) *Convex Hulls of Finite Sets in Two and Three Dimensions*, CACM, vol. 20, 87-93.

[18] Raynaud, H. (1970) *Sur le'enveloppe convexe des nuages des points aleatoires dan* $\Re^n$*,* I. J. Appl. Prob. 7, 35-48.

[19] Sedgewick, R. (1978) *Implementing Quicksort Algorithms,* CACM, vol. 21, no. 10, October.

[20] Shamos, M. I.; Preparata, F. P. (1985) *Computational Geometry*, Springer-Verlag, New York, 131-134.

[21] Seidel, R. (1981) *A Convex Hull Algorithm Optimal for Point Sets in Even Dimensions*, Tech. Rep. 81-14, Dept. of C.S., Univ. of B.C., British Columbia, Canada.

[22] Seidel, R. (1986) *Constructing Higher-Dimensional Convex Hulls at Logarithmic Cost per Face*, Proc. 18th ACM Symp. on Theory of Computing, 404-413.

[23] Seidel, R. (1990) *Linear Programming and Convex Hulls Made Easy,* Proc. Sixth Annual Symp. Computational Geometry, pp. 211-215

[24] Stewart, W. M. (1987) *Three New Algorithms for the d-Dimensional Convex Hull Problem,* Master's thesis, University of New Brunswick.

[25] Stewart, Wm. M. (1990) *An Algorithm to Find the Facets of the Convex Hull in Higher Dimensions,* SCCCG Proceedings, pp. 252-256, August.

[26] Stewart, Wm. M. (1989) *A Convex Hull Algorithm Optimal in Even Dimensions,* APICS Proceedings, pp. 133-142, November.

[27] Swart, G. F. (1985) *Finding the Convex Hull Facet by Facet*, J. Algorithms 6, 17-48