A Complete Implementation for Computing General Dimensional Convex Hulls

Ioannis Z. Emiris

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Complete Implementation for Computing General Dimensional Convex Hulls

Ioannis Z. Emiris

N° 2551

Mai 1995

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

A Complete Implementation for Computing General Dimensional Convex Hulls

Ioannis Z. Emiris [*]

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet SAFIR

Rapport de recherche n ° 2551 — Mai 1995 — 22 pages

Abstract: We study two important, and often complementary, issues in the implementation of geometric algorithms, namely exact arithmetic and degeneracy. We focus on integer arithmetic and propose a general and efficient method for its implementation based on modular arithmetic. We suggest that *probabilistic modular arithmetic* may be of wide interest, as it combines the advantages of modular arithmetic with randomization in order to speed up the lifting of residues to an integer. We derive general error bounds and discuss the implementation of this approach in our general-dimension convex hull program. The use of perturbations as a method to cope with input degeneracy is also illustrated. We present the implementation of a computationally efficient scheme that, moreover, greatly simplifies the task of programming. We concentrate on *postprocessing*, often perceived as the Achilles' heel of perturbations. Starting in the context of a specific application in robotics, we examine the complexity of postprocessing and attempt to delimit the cases where perturbations become a hindrance rather than an enhancement. Lastly, we discuss the visualization capabilities of our software and illustrate them for problems in computational algebraic geometry.

Key-words: Exact integer computation, probabilistic modular arithmetic, general-dimension convex hull, symbolic perturbation, postprocessing, visualization.

(Résumé : tsvp)

[*] Projet SAFIR, I.N.R.I.A., B.P. 93, 06902 Sophia-Antipolis, France. emiris@sophia.inria.fr

Une Implantation Complète pour Construire l'Enveloppe Convexe en Dimension Arbitraire

Résumé : Nous étudions deux problèmes importants, et souvent complémentaires, concernant l'implantation des algorithmes géométriques: l'arithmétique exacte et la singularité des données. En particulier, nous proposons une méthode efficace et générale pour effectuer l'arithmétique sur les entiers laquelle est fondée sur l'arithmétique modulaire (par rapport à un premier). Nous proposons une approche d'*arithmétique modulaire probabiliste*, qui combine les avantages de l'arithmétique dans un champ fini avec une technique aléatoire afin d'améliorer l'efficacité du théorème du reste Chinois. Nous démontrons des bornes générales sur la probabilité d'erreur lesquelles sont utilisées dans notre programme pour construire l'enveloppe convexe en dimension arbitraire. La perturbation, vue comme une méthode pour éviter les singularités. est aussi illustrée au sein de notre programme à travers un schéma efficace pour simplifier la programmation. Ici, nous sommes plutôt intéressés par la phase de *post-traitement* (postprocessing), souvent regardé comme le talon d'Achille des perturbations. Motivés par une application concrète en robotique, nous examinons la complexité du post-traitement et nous essayons de définir les cas dans lesquels la méthode des perturbations n'est pas rentable. Finalement, nous décrivons l'aptitude à la visualisation de notre logiciel en utilisant un exemple dans la géométrie algébrique algorithmique et nous donnons les lieux où le logiciel est accessible.

Mots-clé : Arithmétique exacte sur les entiers, arithmétique modulaire probabiliste, enveloppe convexe en dimension arbitraire, perturbation formelle, post-traitement, visualisation.

1 Introduction

This paper aspires to contribute in the ongoing debate on implementation issues in computational geometry. We focus on two ubiquitous, and often complementary, questions, namely exact arithmetic and input degeneracy.

The idea behind modular arithmetic is that the bulk of the computation is transferred to a domain where it can be carried out efficiently. Then the solutions are transformed back to the domain of the original problem. In the case of exact integer (or rational) arithmetic, most of the computation is carried out over fixed-precision integers.

We review modular arithmetic and present a method for speeding up the transformation of the residues to an integer result. This method may not be new to computer algebra experts but it appears to be largely unknown within the computational geometry community. We derive new bounds on the error probabilities and give a concrete example of its use in our general-dimension implementation of the Beneath-Beyond algorithm. We indicate instances of particular interest of this approach, namely when tight a priori bounds on the results exist or when modular arithmetic is coupled with floating-point arithmetic.

Algorithm designers often choose to ignore special cases. One reason is that the generic problem typically allows us to concentrate on the essential features of the problem. In other cases the treatment of special instances might be disproportionally demanding compared to the frequency they occur. So more often than not, degenerate cases are simply assumed not to arise. A comprehensive case analysis at the implementation stage is often so laborious and prone to programming errors that the perturbation method is warranted.

We sketch the implementation of an existing scheme under which the primitives of interest can be efficiently computed. In addition, the simplicity of the scheme makes it attractive for practical use. An important issue with perturbations is postprocessing. Postprocessing strongly depends on the problem as well as the perturbation employed. Since there exist problems for which the complexity of postprocessing is significant, we may conclude that, although perturbations may dramatically simplify certain algorithms, they are not panacea. This is the point we attempt to illustrate by examining in depth the requirements for a specific robotics application and the modifications needed for our program to produce the desired output.

Our implementation has or is being used in a variety of applications:

- Collision detection in dynamic simulation and animation [LMC94]. Objects are "wrapped up" in their convex hull. If, for any two objects their convex hulls do not intersect, then we have decided quite efficiently that these objects do not collide. This will be the case most of the time, yielding significant time savings. On the other hand, if the two convex hulls penetrate, further examination is needed.

- Prediction of poses for industrial parts on a conveyor belt and computation of stable grasps. This is the robotics application studied in section 5.

- Mixed subdivisions used in solving systems of nonlinear equations. This example is used to illustrate the visualization capabilities of our software (see section 6).

The paper is organized as follows. The following section reviews related work in the area. Section 3 discusses modular arithmetic for exact integer computation and focuses on probabilistically lifting the residues to an integer. Section 4 offers an overview of the implementation and of the perturbation employed and concentrates on the evaluation of the primitive operations. Postprocessing is explained in section 5 and visualization of results is described in section 6. Section 7 provides pointers for accessing the software. A summary of the discussion and some open questions appear in section 8.

2 Related Work

This section refers to alternative approaches to computing convex hulls and reviews perturbation schemes as well as previous work on exact modular arithmetic.

There is a variety of convex hull algorithms and programs. In the past few years, there have been some implementations of general dimension convex hull algorithms, e.g. K. Clarkson's and [BDH93, BMS94]. As far as algorithms are concerned, we restrict attention to Beneath-Beyond. This is an incremental method that repeats the following step: given the convex hull of a subset of the points, it adds one point and updates the convex hull. If the point lies outside the given polytope, the set of visible facets must be identified. The algorithm's efficiency depends heavily on how this process is carried out; there are several different approaches, see e.g. [CS89, Sei91, BDS$^+$92].

Perturbation schemes are extensively compared in [ECS95]; here we discuss only the most relevant approaches to our own. The first systematic approach in computational geometry is due to Edelsbrunner [Ede86, EW86, EG86]. It is called Simulation of Simplicity (SoS) and was generalized by Edelsbrunner and Mücke in [EM90]. Their implementation of a three and four dimensional convex hull algorithm was the basis for our own program. More recently, SoS was implemented in conjunction with three-dimensional Delaunay triangulations [Müc95].

Every input coordinate $x_{i,j}$ is perturbed into

$$x_{i,j}(\epsilon) = x_{i,j} + \epsilon^{2^{i\delta - j}}, \qquad \text{where } \delta > d \text{ and } 1 \gg \epsilon > 0.$$

d is the dimension and ϵ is a symbolic infinitesimal. Intuitively, raising ϵ to such a high power distinguishes between any two coordinates in the context of a wide variety of algorithms. One exception is an inconsistency in the case of the InSphere primitive. The main drawback is that SoS incurs a very high complexity overhead. For instance, deciding the sign of a $d \times d$ perturbed determinant requires the calculation of $\Omega(2^d)$ minors in the worst case, though fewer on average.

Yap [Yap90b, Yap90a] deals with the more general setting in which branching occurs at rational expressions and, even, arbitrary analytic functions [Sei94]. For input variables $\mathbf{x} = (x_1, \dots, x_N)$, the scheme considers all power products $w = \prod_{i=1}^N x_i^{e_i}$, $e_i \geq 0$. Under a total *admissible* ordering $\leq_A$ we have $w \leq_A w' \Rightarrow ww'' \leq_A w'w''$, for any products w, w' and w'' and the power products can be sorted in $1 \leq_A w_1 \leq_A w_2 \leq_A \dots$. Each branching polynomial $f(\mathbf{x})$ is now associated with an infinite list of polynomials

$$S(f) \;=\; (f, f_{w_1}, f_{w_2}, \dots) \;:\; \text{ where } f_w \text{ is the partial derivative with respect to } w.$$

The sign of f under the perturbation is taken to be the sign of the first polynomial in $S(f)$ with a nonzero value and this can always be found after a finite number of evaluations. The merit of the scheme is its generality. However, the worst-case complexity to evaluate $S(f)$ is exponential in N, although the average-case complexity is significantly lower. When the input is organized as n points in $\mathbb{R}^d$, the worst-case complexity of determinant evaluation is exponential in d.

To reduce the complexity overhead, the following scheme was presented in [EC95]:

$$x_{i,j}(\epsilon) = x_{i,j} + \epsilon\, i^j, \qquad \text{where } 1 \gg \epsilon > 0.$$

The perturbation applies to the Ordering (or Sorting), Sidedness (or Orientation), Transversality and InSphere primitives. The worst-case complexity overhead is $\mathcal{O}(\log d)$ under the algebraic computational model.

This scheme was subsequently optimized with respect to the bit size of the perturbation quantities in [EC92]:

$$x_{i,j}(\epsilon) = x_{i,j} + \epsilon\,(i^j \bmod q), \qquad \text{where prime integer } q > n. \tag{1}$$

This is the scheme implemented; section 4 examines its application to the Ordering and Sidedness primitives. Since we can select q to be very close to n, the perturbation quantities have size $\lceil \log n \rceil$.

A very relevant piece of work in the context of implemented perturbations and postprocessing is by Dobrindt, Mehlhorn and Yvinec [DMY93]. They proposed an efficient scheme specifically for coping with degenerate intersections between a convex and a general polyhedron in three dimensions. The vertices of the convex polyhedron are guaranteed to be perturbed in a specific direction with respect to the given facets, so this is a typical instance of a *controlled* perturbation. An important merit of this work in the current context is that it discusses postprocessing in detail in order to recover the exact solution. This leads to authors to conclude against suggesting the use of perturbations in this case.

We must mention that the basic existential question on the perturbation method is still open. After a variety of perturbation schemes have been proposed, recent work argues against the general applicability of this method [BMS94, Sch94] motivated by the observation that the difficulty and complexity of postprocessing may dominate that of the entire program.

Different approaches as well as implementations for exact arithmetic already exist, e.g. [DS88, FvW93, Yap93], though none considers the approach adopted in this paper. Probabilistic Chinese remaindering, which is our approach, was originally proposed in the context of multivariate interpolation by Manocha and Canny [MC93].

3 Probabilistic Modular Arithmetic

This section discusses a modular method for carrying out exact arithmetic over the integers, thus minimizing the need for arbitrary-precision integers. We also suggest a technique that speeds up computation significantly when the computed quantity is near zero and we are ready to accept a very small probability of error. This technique has been implemented in our program and found to be quite efficient in practice. Although the method is not really new, it is not widely used in computational geometry. We wish to suggest that this is an interesting approach for geometric implementations in general and it should be preferable in a variety of situations. Moreover, it can be used with rational data with the same asymptotic complexity [DST88].

Modular arithmetic. First we present the general context for modular arithmetic and how it allows for exact integer arithmetic. The simplest case is when a set of values is given defining a single integer a. For instance we may wish to compute the determinant, given the entries of a square matrix. In general, the answer consists of a sequence $c_0, \ldots, c_d$ of integers. An example encountered in the convex hull implementation is the calculation of the characteristic polynomial coefficients, given the coefficients of an arbitrary square matrix. In any case, the scalar or vector result is defined through a sequence of arithmetic operations which commute with taking residues.

Here are the main stages of the modular method for exact integer computation:

- The given integer quantities are mapped to their residues modulo a set of primes.
- The relevant computation is performed within each finite field defined by every one of the primes.
- The integer answer a or $c_0, \ldots, c_d$ is computed by the results in each finite field.

The last step relies on the Chinese remainder theorem. An alternative technique, based on Hensel's lifting, is discussed in [Lau82]. In order for the entire process to be deterministic, a bound on the magnitude of the final answer must be known. This is used to calculate the cardinality of finite fields.

There are two ways to transform a set of residues to the unique integer that corresponds to them. The first was introduced by Lagrange and requires that all residues be available. We have implemented the second method, due to Newton, because it is *incremental* in the sense that every time a new residue is known a new answer is computed to correspond to the previous residues as well as the new one. Both techniques are described in [Lau82].

Without loss of generality, we consider the case where the answer consists of a single integer $a \in \mathbb{Z}$. Then we let

$$a_j = a \bmod p_j \qquad \text{and} \qquad a^{(k)} = a \bmod (p_1 \cdots p_k),\ k \geq 1.$$

The advantage of Newton's algorithm for computing $a^{(k)}$ is that it requires only a_k and $a^{(k-1)}$, along with some precomputed quantities that depend only on the list of primes: at the k-th step, the method requires the inverse of $(p_1 \cdots p_{k-1})$ modulo p_k. Let

$$s_k \equiv \left(\prod_{i=1}^{k-1} p_i \pmod{p_k} \right)^{-1} \Leftrightarrow s_k \prod_{i=1}^{k-1} p_i \equiv 1 \pmod{p_k}.$$

It is reasonable to assume that integer s_k has been precomputed and stored for every p_k in the list of primes. Using the fact that $(p_1 \cdots p_{k-1})$ and p_k are relatively prime, it can be shown that

$$a^{(k)} = a^{(k-1)} + \left((a_k - a^{(k-1)}) s_k \bmod p_k \right) \prod_{i=1}^{k-1} p_i.$$

Let $M(b) = \mathcal{O}(b \log b \log\log b)$ denote the bit complexity to execute any one of the four basic arithmetic operations between b-bit integers [AHU74]. Let k denote the number of finite fields $\mathbb{Z}_{p_i}$, for distinct primes $p_1, \ldots, p_k$, necessary to carry out a particular computation. The first and third stage have each bit complexity $\mathcal{O}(M(k) \log k)$ [AHU74].

The middle stage is the actual computation within each $\mathbb{Z}_{p_i}$. Since all primes p_i have constant bit size independent of the size of the answer, the bit complexity of this phase is k times the algebraic complexity of the desired computation. We have made the implicit assumption that a sufficiently long list of primes is available at the beginning of the computation and that obtaining p_i from the list is a constant-time operation. Both hypotheses are easy to guarantee by using some upper bound on the input size and by storing an array of primes.

It is observed in several experiments that the maximum number of primes is usually too large and often the answer is reconstructed earlier. In other words, it is typically the case that some $j < k$ exists such that $a^{(j)} = a$, where k is the theoretically secure number of finite fields to be used. Intuitively, as more primes are used, more information is gathered. To take advantage of this phenomenon, we introduce probabilistic criteria for identifying stage j where the computation can stop.

Probabilistic lifting. In the rest of this section we examine randomized methods for speeding up the transformation of a set of residues to an integer. This process is also called *lifting*, since the residues may be thought of as projections of the integer to different finite fields.

We assume that all primes are larger than 2, i.e., $p_i \geq 3$ for all i. The representation of a finite field is possible in several different ways. In this subsection we use positive integers in $[0, p_i)$ to represent $\mathbb{Z}_{p_i}$. Later we switch to an equivalent representation using the integers in $[-\lfloor p_i/2 \rfloor, \lfloor p_i/2 \rfloor]$. To simplify notation we also restrict attention to computing non-negative integers and later extend discussion to the general case.

Let $\pi_k = \prod_{i=1}^{k} p_i$ and define integer β by $\beta = \lfloor R/\pi_k \rfloor$, $R \in \mathbb{Z}_{>0}$.

Lemma 3.1 *Let $a \in \mathbb{Z}$ be uniformly distributed in interval $[0, R)$, for some positive $R \in \mathbb{Z}$. The probability that $a^{(k)} = a^{(k-1)}$ and $a^{(k)} \neq a$, for a fixed k is bounded by $1/p_k$.*

Proof If $\pi_k \geq R$ then $a^{(k)} = a$ and the probability of error is zero. Now we consider the case $\pi_k < R$, which implies $\beta \geq 1$. We partition the interval $[0, R)$ to β disjoint intervals of size π_k and a, possibly empty, interval of size less than π_k. A finer partition is depicted below, where the double brackets or parentheses have no different meaning than the simple ones, except that they mark those subintervals of length π_{k-1} where the hypothesis is satisfied:

$$\begin{aligned} [0, R) \;=\; & [0, \pi_k) \;\cup\; [[\pi_k, \pi_k + \pi_{k-1})) \;\cup\; [\pi_k + \pi_{k-1}, 2\pi_k) \;\cup\; \cdots \\ \cdots \cup\; & [[(\beta-1)\pi_k, (\beta-1)\pi_k + \pi_{k-1})) \;\cup\; [(\beta-1)\pi_k + \pi_{k-1}, \beta\pi_k) \;\cup \cdots \\ & \cdots \cup\; [[\beta\pi_k, \beta\pi_k + \pi_{k-1})) \;\cup\; [\beta\pi_k + \pi_{k-1}, R). \end{aligned}$$

The last two intervals may be empty or the last interval may be empty and the second to last interval may be of length shorter than π_{k-1}. Formally,

$$a^{(k)} = a^{(k-1)} \neq a \Leftrightarrow a \in \left[\alpha \prod_{i=1}^{k} p_i,\ \alpha \prod_{i=1}^{k} p_i + \prod_{i=1}^{k-1} p_i \right), \qquad \text{for some } \alpha \in \mathbb{Z}_{>0} \,:\, \alpha \leq \beta. \tag{2}$$

Note that $\alpha > 0$ and $a > \pi_k$, because otherwise a lies in the very first interval and $a^{(k)} = a$.

There are $\beta - 1$ marked subintervals of length π_{k-1} and one last marked interval of length at most π_{k-1}, therefore the total length of the marked intervals is at most $\beta\pi_{k-1}$. Since $R \geq \beta\pi_k$, the probability that a lies in any marked subinterval is bounded by $(\beta\pi_{k-1})/(\beta\pi_k)$ and the claim follows. □

This lemma expresses the probability that the incremental transformation of a set of residues to an integral value incorrectly stops after a fixed step k. Assuming that K denotes the maximum number of residues that may be considered for a particular a, the following lemma bounds the probability that the transformation stops incorrectly at *any* step. K can be computed from a priori bounds on the size of a, and since we are restricting attention to positive integers, this means $\pi_K \geq R$. For a more convenient expression of the bound we require that most of the primes are larger than K, which is always the case for applications of tractable size.

Lemma 3.2 *Let $a \in \mathbb{Z}$ be uniformly distributed in interval $[0, R)$, for $R \in \mathbb{Z}_{>0}$ and assume $p_2, \ldots, p_{K-1} > K$. Then, the probability that $a^{(k)} = a^{(k-1)}$ and $a^{(k)} \neq a$, for any $k \in \{2, \ldots, K\}$, is bounded by $(K-2)/p_{\min}$, where $p_{\min}$ is the prime of minimum magnitude in $\{p_2, \ldots, p_{K-1}\}$.*

Proof By lemma 3.1, the probability that some $k \in \{2, \ldots, K-1\}$ leads to an incorrect answer is at most $1/p_k$. For $k = K$ this probability is zero. Hence, the overall probability is

$$1 - \prod_{k=2}^{K-1} \left(1 - \frac{1}{p_k}\right) \leq \sum_{k=2}^{K-1} \frac{1}{p_k},$$

by the relation $p_i > K$. The claim follows. □

Now we turn to the more general case that the answer consists of a sequence of $d+1$ values. Notice that d is here a mere parameter that could take any value; later, when we apply these results to the specific problem of convex hull computation, it will denote the geometric space dimension.

Let $c_0, \ldots, c_d$ be the integer values, let $p_1, \ldots, p_k$ be a sequence of primes as above and let

$$c_{ij} = c_i \bmod p_j \qquad \text{and} \qquad c_i^{(k)} = c_i \bmod (p_1 \cdots p_k), \quad k \geq 1.$$

To simplify notation, let $c = (c_0, \ldots, c_d)$, $c_j = (c_{0j}, \ldots, c_{dj})$ and $c^{(k)} = (c_0^{(k)}, \ldots, c_d^{(k)})$. Then $c^{(k)} = c^{(k-1)}$ is equivalent to $c_i^{(k)} = c_i^{(k-1)}$ for all $i \in \{0, \ldots, d\}$.

Theorem 3.3 *Let $c_0, \ldots, c_d \in \mathbb{Z}$ be independently and uniformly distributed in range $[0, R)$ for some $R > 0$ and assume $p_2, \ldots, p_{K-1} > K$. Then the probability that $c^{(k)} = c^{(k-1)}$ and $c^{(k)} \neq c$, for some $k \in \{2, \ldots, K-1\}$, is bounded by $(K-2)/p_{\min}$, where $p_{\min} = \min\{p_2, \ldots, p_{K-1}\}$.*

Proof The conditions of the lemma are equivalent to:

$$\exists j \in \{0, \ldots, d\} \, : \, c_j^{(k)} = c_j^{(k-1)}, \; c_j^{(k)} \neq c_j \qquad \text{and} \qquad c_i^{(k)} = c_i^{(k-1)}, \; \forall i \in \{0, \ldots, d\} \, : \, i \neq j.$$

In the worst case, all entries in $c^{(k)}$ are identical and the relations above hold for some k. Then the claim follows by lemma 3.2. □

We wish to emphasize the fact that this approach is particularly efficient when the result is known a priori to have small magnitude. Moreover, the present technique can be coupled with some adaptive floating-point computation which gives fast accurate results for values away from zero. An important example is sign determination.

Lastly, it must be underlined that integer computation may be efficiently implemented with a double-precision floating-point type. An argument for this approach is the fact that most current architectures provide more bits in the mantissa of a double-precision floating-point type than in a single-precision integer (53 as compared to 32). Furthermore, calculations over the doubles are of comparative speed, if not faster, than integer calculations.

4 Implementation

This section briefly describes our implementation of the Beneath-Beyond algorithm [Ede87] for constructing the convex hull of a finite point set in arbitrary dimension. More precisely, given n points in $\mathbb{Z}^d$, the program returns the exact volume and a triangulation of the facets of the convex hull of these points.

Floating-point input is not excluded. In fact, for a specific application in robotics described in section 5, we wrote a preprocessor to convert floating-point coordinates to integers by multiplying by a power of 10. It is possible to find the smallest power so that the integer inputs include all significant digits.

Beneath-Beyond algorithm. We first discuss the generic Beneath-Beyond algorithm, i.e., the algorithm designed under the assumption that every $k+1$ input points define a nonempty k-simplex. The algorithm constructs the convex hull of the input points in an incremental fashion, by constructing the convex hull of k points, for every $k \in \{d+1, \ldots, n\}$. Passing from k to $k+1$ points requires identifying all facets of the constructed polytope that are *visible* from the $(k+1)$-st point. These are exactly the facets that can be linked to the point with a straight line which does not intersect the polytope.

The main point that distinguishes the different variants of Beneath-Beyond is the way to search for the visible facets. In the "vanilla" version which we have implemented, the following observation is used: If the points are sorted on some coordinate, then there is always one facet of the k-th hull that has the k-th point as a vertex and is visible by the $(k+1)$-st point. In other words, we can always find a visible facet among those added in the previous stage, therefore we may simply search through all of the latter facets until a visible one is found. Then, we examine adjacent facets and recurse on those that are themselves visible.

Sorting the points also implies that every new point is external with respect to the existing partial convex hull. This approach requires that the input points have distinct coordinates along the chosen axis, a condition which is simulated by our perturbation. More efficient approaches exist for enumerating the visible facets, e.g. [CS89, Sei91, BDS$^+$92, BDH93].

The visibility test consists of deciding whether a facet defined by $d+1$ points is visible by another point. If the points defining every facet of the current hull are given in *positive orientation*, then it is possible to implement the visibility test as a single Orientation test. The Orientation primitive is well-known and also defined below. The set of $d+1$ vertices have positive orientation if, when projected on the hyperplane of the facet, they make the d-dimensional Orientation primitive take a positive value.

Instead, we choose to implement the visibility test with two Orientation tests, one with respect to the new point and one with respect to a point that is known to be in the convex hull. Thus we do not have to sort the facet vertices. If this order, however arbitrary, is not changed during execution, then we can store at the facet the sign of the first Orientation test with respect to an interior point. Then the total number of Orientation tests per facet is just one more than the total number of visibility tests involving this facet.

Perturbation. The computationally efficient perturbation scheme (1), mentioned in section 2, is used in order to simulate genericity of the input. It has been introduced in [EC92] and later developed in [ECS95]. We mention below the asymptotic bounds on the complexity overhead incurred by this scheme. Proofs of all claims made below can be found in both publications. $MM(t) = \mathcal{O}(t^{2.376})$ [CW90] is the matrix multiply complexity, which is asymptotically equal to the complexity of inverting a matrix [vzG88].

The perturbation uses a prime q to change the k-th coordinate of the i-th input point as follows, where $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, d\}$:

$$x_{i,k}(\epsilon) = x_{i,k} + \epsilon\,(i^k \bmod q), \qquad \text{where } 1 \gg \epsilon > 0 \text{ and } q > n. \tag{1}$$

To minimize bit complexity we choose q to be the smallest prime that exceeds n. The perturbation is

- symbolic, in the sense that the exact amount by which every point is perturbed depends on variable ϵ which may be thought of as taking different values for different inputs,

- conceptual, because there is no symbolic computation required,
- infinitesimal, since ϵ is arbitrarily small, and
- computationally efficient with respect to a family of primitives (including Ordering and Sidedness), in the sense that the asymptotic bit complexity of evaluating these primitives under the perturbation is higher than the complexity of the original primitives at most by a logarithmic factor in d.

The combinatorial part of the program is not aware of the perturbation, and this is one of the advantages of the perturbation method. Namely, to add a perturbation scheme to a given program the implementor need only rewrite the parts computing the primitives. This is indeed what we had to do, given the convex hull program of E. Mücke from the University of Illinois at Urbana-Champagne. This was a PASCAL program computing convex hulls in three and four dimensions using SoS (see section 2) and arbitrary-precision integers.

The two main primitive operations are now examined in the light of the chosen perturbation. The Sorting or Ordering primitive is implemented on the first axis and requires nothing more than sorting, with some mechanism for breaking ties. For any two points x_{i_1} and x_{i_2} we must decide the sign of $x_{i_1,1} - x_{i_2,1}$, or, under the perturbation, the sign of

$$x_{i_1,1}(\epsilon) - x_{i_2,1}(\epsilon) \;=\; x_{i_1,1} + \epsilon\, i_1 - x_{i_2,1} - \epsilon\, i_2 \qquad 1 \le i_1 \ne i_2 \le n. \tag{3}$$

Degeneracy with respect to this primitive occurs exactly when $x_{i_1,1} = x_{i_2,1}$, in which case the test reduces to comparing i_1 versus i_2. Since the two points are distinct, there is always a unique ordering.

The second and more important primitive deciding visibility of facets is known as Orientation or Sidedness. Given is a facet of the partial convex hull, defined by points $x_{i_1}, \ldots, x_{i_d}$, and the new point $x_{i_{d+1}}$. The hyperplane of the facet in $\mathbb{R}^d$ defines two halfspaces: the primitive asks which halfspace contains the new point. This halfspace does not contain the partial hull if and only if the facet is visible. The halfspace of $x_{i_{d+1}}$ with respect to the facet is determined by the sign of the determinant of

$$\Lambda_{d+1} = \begin{bmatrix} 1 & x_{i_1,1} & x_{i_1,2} & \cdots & x_{i_1,d} \\ 1 & x_{i_2,1} & x_{i_2,2} & \cdots & x_{i_2,d} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{i_{d+1},1} & x_{i_{d+1},2} & \cdots & x_{i_{d+1},d} \end{bmatrix}.$$

The degenerate case is $\det \Lambda_{d+1} = 0$, which happens exactly when $x_{i_1}, \ldots, x_{i_d+1}$ lie on the same hyperplane. The perturbed matrix is denoted by $\Lambda_{d+1}(\epsilon)$ and is obtained by substituting $x_{i_k,j}$ by $x_{i_k,j} + \epsilon(i_k^j \bmod q)$. It can be shown that the perturbed determinant is never zero since it has at least one nonzero term, as seen in the following expression:

$$\det \Lambda_{d+1}(\epsilon) = \det \Lambda_{d+1} + \left(\epsilon^k \text{ terms }, 1 \le k \le d-1\right) \;+\; \epsilon^d \det V,$$

where V resembles a Vandermonde matrix:

$$V = \begin{bmatrix} 1 & i_1 \bmod q & \cdots & i_1^d \bmod q \\ 1 & i_2 \bmod q & \cdots & i_2^d \bmod q \\ \vdots & \vdots & & \vdots \\ 1 & i_{d+1} \bmod q & \cdots & i_{d+1}^d \bmod q \end{bmatrix} \qquad \Rightarrow \det V \equiv \prod_{k>l\ge 1}^{d+1} (i_k - i_l) \not\equiv 0 \pmod{q}.$$

The merit of the specific perturbation scheme is that it reduces sign determination for the perturbed determinant to a characteristic polynomial computation, as shown in a general context by [ECS95, Th. 13]. Here we obtain

$$\det \Lambda_{d+1}(\epsilon) = \frac{1}{1}(-1)^{d+1} \det V \det(M - \epsilon I). \tag{4}$$

The sign of the lowest-power nonzero term in the ϵ-polynomial is the sign of the perturbed determinant. Here I is the $(d+1) \times (d+1)$ identity matrix and M is another $(d+1) \times (d+1)$ matrix with entries among the input coordinates:

$$M = -V^{-1}L \qquad \text{where} \qquad L = \begin{bmatrix} 0 & x_{i_1,1} & \dots & x_{i_1,d} \\ \vdots & \vdots & & \vdots \\ 0 & x_{i_{d+1},1} & \dots & x_{i_{d+1},d} \end{bmatrix}.$$

The crucial property of the perturbation is the linearity in ϵ. This leads to expression 4, which demonstrates that there is no explicit symbolic computation, and to an efficient computation of $\det \Lambda_{d+1}(\epsilon)$.

Theorem 4.1 [ECS95, Th. 15, 17] *Perturbation (1) is valid with respect to Ordering and Sidedness. Suppose that there exist n distinct input parameters and that determinant sign determination is implemented by determinant evaluation. Then the perturbation increases the bit complexity of Ordering along the first coordinates by a factor of 2 and the asymptotic worst case bit complexity of Sidedness by a factor of $\mathcal{O}(\log d)$.*

The $\mathcal{O}(\log d)$ factor follows from a bound of $MM(d)\,\mathcal{O}(\log d)$ on the complexity of computing all coefficients of the characteristic polynomial of an arbitrary square matrix [KG85]. However, this algorithm is nontrivial and we have, instead, implemented a simpler $\mathcal{O}(d^3)$ method.

In fact, each Sidedness test starts with computing $\det \Lambda_{d+1}$. If this is nonzero then there is no need for the perturbed primitive, which is more expensive. The computation of this determinant is also implemented by probabilistic modular arithmetic. The program tries different primes $p_1, \dots, p_k$, for some $k \geq 1$, until one of the following happens:

- either $\det \Lambda_{d+1} \equiv 0 \pmod{p_1}$
- or $\det \Lambda_{d+1} \bmod \prod_{i=1}^{k} p_i = \det \Lambda_{d+1} \bmod \prod_{i=1}^{k-1} p_i$.

If the second case occurs and $\det \Lambda_{d+1} \bmod \prod_{i=1}^{k} p_i \neq 0$, then the determinant sign is accepted. Otherwise, the perturbed predicate is computed.

The $\mathcal{O}(d^3)$ steps for computing $\det \Lambda_{d+1}(\epsilon)$ and their complexities are listed below. The algorithms can be found in [Wil65].

- Compute the determinant of V and its inverse by LU decomposition and back-substitution, with total complexity $5(d+1)^3/6 + \mathcal{O}(d^2)$.
- Multiply V and L to obtain M in time $(d+1)^3$.
- Compute the upper Hessenberg form of M in time $5(d+1)^3/6 + \mathcal{O}(d^2)$.
- Compute the companion matrix of $M - \epsilon I$ in time $(d+1)^3/6 + \mathcal{O}(d^2)$.

Hence the total arithmetic complexity is

$$2\frac{5}{6}\,(d+1)^3 + \mathcal{O}(d^2) \;=\; 2\frac{5}{6}\,d^3 + \mathcal{O}(d^2).$$

Arithmetic. We now turn to the implementation of these primitives. The program runs over the integers and requires exact arithmetic, in order to evaluate the perturbed predicates. This is a strict requirement, since any approximation may lead to inconsistencies in the perturbed configuration. We have implemented the probabilistic modular method discussed in detail in section 3.

As seen in that section, the size of the chosen primes determines their cardinality. We are mostly working with 32-bit machines, therefore, in order to use single-precision arithmetic for the bulk of the computation, we would choose primes close to 2^{32}. However, this complicates the handling of the overflow bit during addition and subtraction. So we choose primes smaller than 2^{31}. Of course, multiplication still requires some extra words to store the intermediate values before projecting in the current finite field.

The chosen method for carrying out arithmetic calls for a modular arithmetic package to execute the bulk of the computation and an arbitrary-precision integer package for transforming the residues of some integer result to an integer value. The first package is based on an implementation by D. Manocha written at U. C. Berkeley, which included an implementation of the modular multiplication routine in the SUN assembly language. For portability, we have added a C version of this routine, which is often at the bottleneck of the computation. We have also used the Long Integer Arithmetic package of H. Rosenberg from the University of Illinois at Urbana-Champaign in order to carry out the Chinese remaindering operations.

As described above, the transformation of a set of residues to an integer value stops when two consecutive primes lead to the same scalar or vector value. The probability of error depends on the maximum number of primes needed, so we proceed to compute bounds on this number. Such bounds are actually computed by the program using arbitrary-precision integer, in order to give the option to the user to run with zero probability of error.

In what follows we derive bounds on $\det \Lambda_{d+1}$ and on the coefficients of $\det \Lambda_{d+1}(\epsilon)$ as well as on the number of primes needed for each computation. We denote by $s > 0$ the maximum number of bits in any input coordinate, i.e., $x_{k,j} \in (-2^s, 2^s)$. We had considered integer values in range $[0, R)$, whereas now we have to account for their sign as well. This range maps bijectively to $(\lfloor R/2 \rfloor, \lceil R/2 \rceil]$.

The Ordering primitive is trivial and does not require any modular arithmetic.

Theorem 4.2 *In computing* $\det \Lambda_{d+1}$ *by modular arithmetic, the maximum number* K *of 31-bit primes needed is*

$$30K \geq s(d+1) + 1 + \frac{d+1}{2} \lg(d+1).$$

Proof The magnitude of $\det \Lambda_{d+1}$ is, by Hadamard's inequality (see e.g. [Mig82]),

$$|\det \Lambda_{d+1}| \leq \prod_{k=1}^{d+1} \sqrt{1 + \sum_{j=2}^{d+1} x_{i_k,j-1}^2} \leq (d+1)^{\frac{d+1}{2}} \prod_{k=1}^{d+1} x_{i_k,\max}, \qquad \text{where } x_{i_k,\max} = \max\{1, x_{i_k,1}, \ldots, x_{i_k,d}\}. \tag{5}$$

This bound is indeed computed by the program. The magnitude of the determinant is bounded by $\lfloor R/2 \rfloor$.

$$R < 2^{s(d+1)+1}(d+1)^{\frac{d+1}{2}}.$$

The maximum number of primes K for this computation is determined by $\pi_K \geq R > \pi_{K-1}$. So it suffices to have $\pi_k \geq R$. Applying $\lg p_i > 30$ for all primes $p_1, \ldots, p_K$ completes the proof. □

If we restrict attention to $d \leq 14$ and $s \leq 31$ we obtain $K \geq 17$.

For the perturbed determinant we must compute the coefficients of the characteristic polynomial of M.

Theorem 4.3 *In computing the characteristic polynomial of M by modular arithmetic, the maximum number K of 31-bit primes is given by*

$$30\,K \geq d \max\{s, d \lg n\} + 2 + \frac{d}{2} \lg d,$$

where s is the maximum bit size of the input coordinates.

Proof From expression (4) the characteristic polynomial is the product of the ϵ-polynomial $\det \Lambda_{d+1}(\epsilon)$ multiplied by $\det V > 1$. Therefore the maximum coefficient size of $\det \Lambda_{d+1}(\epsilon)$ provides an upper bound on the size of the characteristic polynomial coefficients. Each entry of $\Lambda_{d+1}(\epsilon)$ is a sum of an input point coordinate and a perturbation quantity, hence its bit size is the maximum of $1+s$ and $1+d\lg n$. All coefficients of $\det \Lambda_{d+1}(\epsilon)$ are sums of determinants of order at most d that have entries of bit size $1+\max\{s, d\lg n\}$. Hadamard's bound leads to an expression similar to (5) for every coefficient c_i:

$$|c_i| \leq \prod_{k=1}^{d} \sqrt{\sum_{j=1}^{d} 2^{1+\max\{s,d\lg n\}}} = d^{d/2} \left(2^{1+\max\{s,d\lg n\}}\right)^d.$$

If $\lfloor R/2 \rfloor$ bounds the magnitude of each coefficient, then

$$R < 2^{d\max\{s,d\lg n\}+2} d^{d/2}$$

and the claim follows. □

Usually $K > d^2 \lg n/31$ suffices. For $s \leq 31$, $d \leq 14$ and $n \leq 10000$ imply $K \geq 88$, whereas $d \leq 8$ and $n \leq 1000$ give $K \geq 22$. For $s \leq 7$, $d \leq 6$ and $n \leq 100$ the theorem yields $K \geq 9$, which is the largest number of primes required for the examples of table 4. In our implementation $K = 15$.

Complexity. The current version is in Ansi-C and includes about 1000 lines of code for the main combinatorial part, 600 lines for implementing the primitives over finite fields and 1400 lines for the modular and exact integer arithmetic package.

Table 4 is copied from [ECS95] to show the performance of the program on a SparcStation 10/41 with one 40 MHz processor, 32 MBytes of memory and a rating of 50 SpecInt92. d and n stand for the dimension and the number of input points respectively and all coordinates are integers in $(-100, 100)$. The output of the program is the rational volume and a list of facets; postprocessing is discussed in the next section. The user CPU running times are rounded down to an integer number of seconds.

For fixed d and n we have experimented with inputs of various degrees of degeneracy. The last three columns are headed by the fraction ρ of Orientation tests whose evaluation is not zero on the original input and which are carried out as determinant calculations. Thus, the first column corresponds to random inputs with practically all tests being generic. The other extreme has inputs

		user CPU running time		
d	n	$\rho \simeq 1$ (random)	$\rho \simeq .5$	$\rho = 0$ (coincident)
4	54	4s		24s
4	100	8s	40s	1m 6s
4	200	19s	1m 11s	2m 8s
4	500	53s		7m 39s
3	100	0s		11s
4	100	8s	40s	1m 6s
5	100	43s	4m 7s	5m 19s
6	100	4m 18s		45m 15s
7	100	29m 1s		

Table 1: Performance of the Convex Hull Volume implementation.

with coincident points, constructed to study the program's performance when all Orientation tests reduce to a characteristic polynomial. The middle column corresponds to point sets comprised of both random and coincident points.

In analyzing these results it must be remembered that the program's complexity depends on the number of facets in the partial convex hulls. In the worst case, the hull of n points in d dimensions has $\mathcal{O}(n^{\lfloor d/2 \rfloor})$ facets. However, the expected number of facets for points selected randomly as above is proportional to $\log^{d-1} n$ [BKST78]; this is verified by our experimental results.

5 Postprocessing

This section starts by describing the output of the implementation discussed above and then focuses on the postprocessing required for recovering a different description of the convex hull. We report on our experience with implementing this postprocessing in three dimensions.

The exact volume of the convex hull is obtained without postprocessing because the problem mapping $\mathbb{Z}^{dn} \rightarrow \mathbb{Q}$ is continuous everywhere [ECS95, Prop. 4].

At any stage of the incremental construction, the perturbation causes every region between the point being added and the existing hull to be partitioned into d-dimensional simplices. Each, possibly empty, simplex is defined by the new point and one of the visible facets. Clearly, the sum of the simplex volumes gives the amount of volume "added" at this stage, where the sum of all such volumes gives the convex hull volume. Every simplex volume is expressed as a determinant $\det \Lambda_{d+1}$. To compute the exact volume we simply sum all volumes for which $\det \Lambda_{d+1} \neq 0$. When the program computes $\det \Lambda_{d+1}(\epsilon)$, the volume is given by the constant term of this polynomial.

To specify the facet output we introduce some terminology. Let the *extreme* points of a given point set in $\mathbb{R}^d$ be those that *strictly* maximize the inner product with some d-vector, i.e., they are not expressible as a convex combination of the other points. The extreme input points are exactly the vertices of the convex hull. Due to the perturbation, though, the output vertex set may be a proper superset of the extreme points. This happens when the vertex set contains points that are not extreme but simply *extremal*, i.e., they simply maximize the inner product with a certain vector. In other words, extremal points lie in the interior of the convex hull of some face.

There are two reasons why non-extreme points may be reported as vertices.

- First, the perturbation does not allow any $d+1$ points to lie on the same hyperplane. In particular, whenever there are $d+1$ extreme coplanar points they will define more than one facet. As a result, the output polytope is *simplicial.*
- Second, there may be non-extreme points that appear as vertices simply because of the perturbation. For instance, any 3 collinear extremal points in $\mathbb{R}^d$, for any d, may give rise to more than one edge.

Depending on the application, a different description of the convex hull may be desired. Here we set the following goals:

- Eliminate all faces defined by $k+1$ points having zero k-dimensional volume, for any $k \in \{1, \ldots, d-1\}$.
- Merge all facets lying on the same $(d-1)$-dimensional hyperplane.
- Inform the user if the convex hull has zero d-dimensional volume and return a lower dimensional convex hull satisfying the preceding conditions.

Postprocessing in general dimension can be based on the computation of each facet's normal. Code for computing these normals and a specific application where the lower hull was needed are discussed in the section 6. Theoretical issues, including a complexity analysis, can be found in [ECS95].

We turn, now, to three-dimensional convex hulls used in computing the pose of an industrial part that falls on a conveyor belt. The algorithm of [WRG92] reads a set of points defining the part and computes its convex hull facets. Then it projects these facets on a sphere in order to compute the probability of the object landing on this facet. This information is used by a gripper in order to grasp and move the part from the belt.

The input comes in floating-point numbers: we calculate the smallest power of 10 that produces signed integers of at most 31 bits and multiply the data by this power. For this application the precision is always sufficient.

The goal of postprocessing is to construct a polytope whose 2-dimensional facets are the convex hull of all coplanar input points; such facets may be defined by an arbitrary number of points. Moreover, no coincident vertices are allowed and instances of more than 2 collinear points should be substituted by the endpoints of their segment. Coincident points are dealt with at the initial phase of sorting along an axis, where we simply check the other two coordinates in order to eliminate duplicates.

Since we are concerned only with the specific low-dimensional case, we choose to maintain the necessary planarity information as we build the hull. This is straightforward because of the way the Orientation primitive has been implemented: if $\det \Lambda_4$ vanishes, then we know that only because of the perturbation the query point appears to be off the given plane. Hence we can mark the appropriate new facets as being on the same plane as the old facet.

Visibility is implemented as suggested in the description of the Beneath-Beyond algorithm in [Ede87]: All facets of the partial hull are colored white at the beginning of each update. Then, those that are examined with respect to the point being added are colored either red or blue, according to whether they are visible or not. To include the planarity information we introduce colors pink and cyan which indicate facets that are visible or invisible only due to the perturbation, i.e., $\det \Lambda_4 = 0$.

Moreover, facets carry an extra field denoting the plane on which they lie. Every plane is assigned a distinct positive integer when a triangle is found on it. This integer serves as the plane's identification and indexes an array that points to some facet on this plane. One-dimensional facets, also called *slivers*, are assigned the negatives of the two plane identifications corresponding to the planes intersecting at this facet. Notice that slivers are different from edges since they are defined by three points. The two planes defining a sliver are uniquely defined, so the same pair of planes corresponds to all collinear slivers.

Formally, we define the *plane information* of every facet to be as follows.

- Every 2-dimensional facet contains a field $(l, 0)$, where $l \in \mathbb{Z}_{>0}$ indicates the unique 2-dimensional plane containing the facet.

- Every sliver contains a field $(-l_1, -l_2)$, where $l_1 \neq l_2 \in \mathbb{Z}$ and $l_1 > 0$ is always a plane identifier, while $l_2 \geq 0$ may be zero. If both identifiers are negative, they indicate the two unique 2-dimensional planes the intersection of which contains the sliver. Slivers for which only one plane $l > 0$ is known have plane information $(-l, 0)$.

This information also distinguishes between two-dimensional facets and slivers.

Special cases handle inputs defining a one or two-dimensional convex hull.

A useful observation is that when a facet is red or blue it cannot be a sliver. Given a red (or pink) and a blue (or cyan) facet and the new point, the case analysis below indicates how to obtain the plane information of the new facet. Below l, l_1 and l_2 are all positive.

- New plane: If the given facets are red and blue then the new facet has pair $(l, 0)$, where l corresponds to a new plane.

- Plane information from the pink facet: If the pink facet has plane information $(l, 0)$ then the new facet has plane information $(l, 0)$ or $(-l, 0)$, depending on whether it is two-dimensional or not. If the pink facet has plane information $(-l_1, -l_2)$ and the new facet is a sliver, then its plane information is $(-l_1, -l_2)$.

- Plane information from the blue or cyan facet: If the given facets are red and cyan and the latter has plane information $(l, 0)$, or if the facets are a pink sliver and a cyan facet with pair $(l, 0)$, then the new facet has $(l, 0)$.

- The last case is not defined as it involves too much detail. It covers the cases of a red facet and a cyan sliver, pink and cyan slivers and, finally, pink sliver and blue facet.

Once the three-dimensional polytope is built, coplanar facets are merged and, for each plane, a two-dimensional problem is solved to clear the non-extreme points.

Maintaining the coplanarity information, merging facets and running the two-dimensional subproblems requires an additional 500 lines of code. It seems that in arbitrary dimension this approach to postprocessing would be suboptimal. Moreover, it is doubtful whether the perturbation method would be the best alternative when a triangulation of the hull surface is not desired because the case analysis would be too complicated. After all, introducing a lengthy case analysis defeats the purpose of perturbations. In general, when the complexity of postprocessing becomes significant, the value of the perturbation technique should be reexamined against alternative strategies.

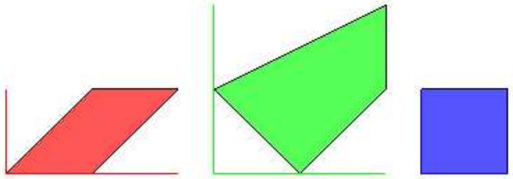

Figure 1: The red, green and blue original Newton polygons.

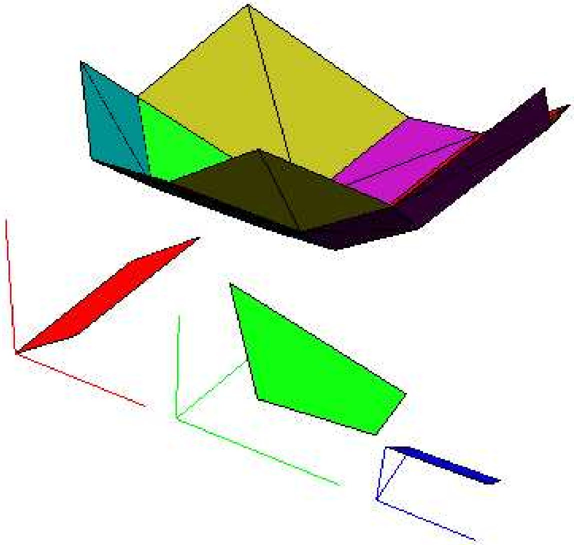

Figure 2: The lifted polygons and the lower hull of their 3-dimensional Minkowski sum.

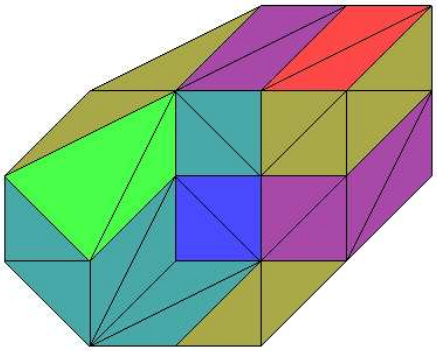

Figure 3: A mixed subdivision of the two-dimensional Minkowski sum.

6 Visualization

Our implementation has an option in three and four dimensions for producing output in `oogl OFF` format, which is compatible with GEOMVIEW. Below we illustrate this capability by examining a specific application in computational algebraic geometry.

The problem we consider here comes from *sparse elimination theory*, a relatively new area of computational algebraic geometry which exploits the monomial structure of polynomial systems in order to obtain tighter bounds and faster algorithms for their manipulation [GKZ94]. The algebraic questions are formulated in geometric terms by means of the *Newton polytope* of a polynomial, which is a convex polytope in $\mathbb{R}^n$, where n denotes the number of variables. The Newton polytope captures the sparse structure of the given polynomial and, in particular, their volume depends on the number of nonzero terms.

An important object in the study of systems of $n+1$ polynomials is the *sparse*, or *Newton, resultant.* Two of the algorithms that construct this resultant [CE93, CP93] rely on a *mixed subdivision* of the Minkowski sum of the $n+1$ corresponding Newton polytopes. Recall that the Minkowski sum of two convex polytopes A and B is convex polytope $A+B = \{a+b | a \in A, b \in B\}$. Moreover, it has been conjectured [Emi94] that from this subdivision an exact matrix formula for the sparse resultant can be obtained. Mixed subdivisions are also instrumental in defining *sparse homotopies* [HS, VVC94] and computing *mixed volumes* [EC94].

Our publicly available implementation in

`ftp://robotics.eecs.Berkeley.edu/pub/MixedVolume/subdiv`

computes a mixed subdivision of $n+1$ Newton polytopes in n dimensions by using the convex hull program as a callable library. The algorithm first lifts the input polytopes to a hyperplane in $n+1$ dimensions, then takes the lower hull of the lifted Minkowski sum which is projected into $\mathbb{R}^n$ along the $(n+1)$-st coordinate axis. The convex hull program computes all facets of the lifted sum and a postprocessing step checks their normal vectors in order to keep only those on the lower hull. Lower hull facets project bijectively to maximal cells in the mixed subdivision of the n-dimensional Minkowski sum.

In the example examined here, $n = 2$ and we use color to code the cells and express their "mixedness". We use the color vectors of GEOMVIEW which define a unique color given three rational coordinates in $[0,1]$ corresponding to the amount of red, green and blue. Figure 1 shows the three original polygons in distinct coordinate frames. Figure 2 shows the lower hull of the Minkowski sum corresponding to the three shown lifted polygons, colored respectively red, green and blue. The respective color vectors are $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$. Notice that all facets are triangulated, as a side effect of our convex hull program, and that shading is used in three dimensions.

Figure 3 shows the maximal cells in the mixed subdivision of the two-dimensional sum, where the red, green and blue cells are translated copies of the original polygons. Notice again that the cells are triangulated since they result from projecting the lower hull facets along the x_3-axis. All other cells are *mixed*, in the sense that they are the Minkowski sum of two edges of distinct input polygons: mixed cells generated by a red and a green edge have color vector $(0.5, 0.5, 0)$ and are painted reddish-green; cells generated by a red and a blue edge have vector $(0.5, 0, 0.5)$ and appear as violet.

7 Availability

Our general dimension program is publicly available from

`ftp://robotics.eecs.Berkeley.edu/pub/ConvexHull`.

This directory contains

- a README file,
- all sources in Ansi-C,
- a makefile for SUN, DEC ALPHA, SGI and HP architectures.
- subdirectory `/lib` where the program is organized as a library to be used with other software,
- subdirectory `/envelope` which provides the routines called by the subdivision program and which distinguish between upper and lower envelope facets,
- subdirectory `/tests` containing input files and the outputs produced under various options,
- and subdirectory `/papers` containing some of the relevant publications.

These directories are also available through

`http://www.inria.fr/safir/SAFIR/Ioannis.html`.

8 Conclusion

The paper proposes a probabilistic modular method for exact arithmetic and argues for its general applicability in computational geometry implementations. We have computed bounds on the probability of error and have illustrated the overall approach with its advantages and disadvantages in the context of our convex hull program. We also suggest combining this technique with floating-point computation which gives fast accurate results in complementary cases.

The second contribution of this paper regards the perturbation method. The convex hull program is described in detail and, in particular, the use of a computationally efficient method is discussed that handles degenerate cases. Probably the main limitation of perturbations is the need for postprocessing: we consider this issue in the specific context of three-dimensional convex hulls and describe our software to address this problem. Several advantages and drawbacks of the perturbation are mentioned. We think that perturbations can greatly enhance the power of certain geometric programs, but they may also become a boomerang if applied without discrimination.

Lastly, we have discussed the visualization capabilities of our software in connection to a specific application in computational algebraic geometry and have reported on the software's availability.

Acknowledgment

I thank John Canny for introducing me to Newton polytopes and perturbation schemes as well as for his insightful ideas.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[BDH93] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hull. Tech. Report GCG53, Geometry Center, Univ. of Minnesota, July 1993.

[BDS$^+$92] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Appliactions of random sampling to on-line algorithms in computational geometry. *Discr. and Comput. Geometry*, 8:51–71, 1992.

[BKST78] J.L. Bentley, H.T. Kung, M. Schkolnick, and C.D. Thompson. On the average number of maxima in a set of vectors. *J. ACM*, 25:536–543, 1978.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM–SIAM Symp. on Discr. Algorithms*, pages 16–23, 1994.

[CE93] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. Intern. Symp. Applied Algebra, Algebraic Algor. and Error-Corr. Codes, Lect. Notes in Comp. Science 263*, pages 89–104, Puerto Rico, 1993. Springer Verlag.

[CP93] J. Canny and P. Pedersen. An algorithm for the Newton resultant. Technical Report 1394, Computer Science Dept., Cornell University, 1993.

[CS89] K.L. Clarkson and P.W. Shor. Appliactions of random sampling in computational geometry, II. *Discr. and Comput. Geometry*, 4:387–422, 1989.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9:251–280, 1990.

[DMY93] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete framework for the intersection of a general polyhedron with a convex one. In *Proc. 3rd Workshop Algorithms Data Struct., Lect. Notes Comp. Science 709*, pages 314–324. Springer-Verlag, Berlin, 1993.

[DS88] D. Dobkin and D. Silver. Recipes for geometric and numerical analysis – part i: An empirical study. In *Proc. ACM Symp. on Computational Geometry*, pages 93–105, 1988.

[DST88] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra.* Academic Press, London, 1988.

[EC92] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. ACM Symp. on Computational Geometry*, pages 74–82, 1992.

[EC94] I.Z. Emiris and J.F. Canny. Efficient incremental algorithms for the sparse resultant and the mixed volume. Technical Report 839, Computer Science Division, U.C. Berkeley, 1994. Submitted for publication.

[EC95] I.Z. Emiris and J.F. Canny. A general approach to removing degeneracies. *SIAM J. Computing*, 24(3), 1995. To appear. A preliminary version in *Proc. IEEE Symp. Foundations of Comp. Sci.,* 1991, pp. 405–413.

[ECS95] I.Z. Emiris, J.F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica, Spec. issue on comp. geometry in manufacturing*, 1995. To appear.

[Ede86] H. Edelsbrunner. Edge-skeletons in arrangements with applications. *Algorithmica*, 1:93–109, 1986.

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

[EG86] H. Edelsbrunner and L.J. Guibas. Topologically sweeping an arrangement. In *Proc. ACM Symp. Theory of Comp.*, pages 389–403, 1986.

[EM90] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9(1):67–104, 1990.

[Emi94] I.Z. Emiris. *Sparse Elimination and Applications in Kinematics.* PhD thesis, Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, December 1994.

[EW86] H. Edelsbrunner and R. Waupotitsch. Computing a ham-sandwich cut in two dimensions. *J. Symbolic Comput.*, 2:171–178, 1986.

[FvW93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. ACM Symp. on Computational Geometry*, pages 163–172, 1993.

[GKZ94] I.M. Gelfand, M.M. Kapranov, and A.V. Zelevinsky. *Discriminants and Resultants.* Birkhäuser, Boston, 1994.

[HS] B. Huber and B. Sturmfels. A polyhedral method for solving sparse polynomial systems. *Math. Comp.* To appear. A preliminary version presented at the Workshop on Real Algebraic Geometry, Aug. 1992.

[KG85] W. Keller-Gehrig. Fast algorithms for the characteristic polynomial. *Theor. Comp. Sci.*, 36:309–317, 1985.

[Lau82] M. Lauer. Computing by homomorphic images. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 139–168. Springer-Verlag, Wien, 2nd edition, 1982.

[LMC94] M.C. Lin, D. Manocha, and J. Canny. Efficient contact determination for dynamic environments. In *Proc. IEEE Intern. Conf. Robotics and Automation*, pages 602–608, 1994.

[MC93] D. Manocha and J. Canny. Multipolynomial resultant algorithms. *J. Symbolic Computation*, 15(2):99–122, 1993.

[Mig82] M. Mignotte. Some useful bounds. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 259–263. Springer-Verlag, Wien, 2nd edition, 1982.

[Müc95] E.P. Mücke. Detri 2.2: A robust implementation for 3D Delaunay triangulations. Presented at the Geometric Software Workshop, The Geometry Center, Minneapolis. Sources available from `ftp://cs.uiuc.edu/pub/edels/geometry`, January 1995.

[Sch94] P. Schorn. Degeneracy in geometric computation and the perturbation approach. *The Computer Journal*, 37(1):35–42, 1994.

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

[Sei94] R. Seidel. The nature and meaning of perturbations in geometric computing. In *Proc. 11th Symp. Theoret. Aspects Computer Science, Lect. Notes Computer Science 775*, pages 3–17. Springer-Verlag, 1994.

[VVC94] J. Verschelde, P. Verlinden, and R. Cools. Homotopies exploiting Newton polytopes for solving sparse polynomial systems. *SIAM J. Numerical Analysis*, 31(3):915–930, 1994.

[vzG88] J. von zur Gathen. Algebraic complexity theory. In J. Traub, editor, *Annual Review of Computer Science*, pages 317–347. Annual Reviews, Palo Alto, Cal., 1988.

[Wil65] J. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford Univ. Press, London, 1965.

[WRG92] J. Wiegley, A. Rao, and K. Goldberg. Computing a statistical distribution of stable poses for a polyhedron. In *Proc. 30th Annual Allerton Conf. on Comm. Control and Computing*, U.Ill. Urbana-Champaign, 1992.

[Yap90a] C.-K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comp. Sys. Sci.*, 40:2–18, 1990.

[Yap90b] C.-K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10:349–370, 1990.

[Yap93] C.-K. Yap. Towards exact geometric computation. In *Proc. Canadian Conf. on Computational Geometry*, 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399