Faster Geometric Algorithms via Dynamic Determinant Computation

Vissarion Fisikopoulos and Luis Peñaranda

University of Athens, Dept. of Informatics & Telecommunications, Athens, Greece
{vfisikop,lpenaranda}@di.uoa.gr

Abstract. Determinant computation is the core procedure in many important geometric algorithms, such as convex hull computations and point locations. As the dimension of the computation space grows, a higher percentage of the computation time is consumed by these predicates. In this paper we study the sequences of determinants that appear in geometric algorithms. We use dynamic determinant algorithms to speed-up the computation of each predicate by using information from previously computed predicates.

We propose two dynamic determinant algorithms with quadratic complexity when employed in convex hull computations, and with linear complexity when used in point location problems. Moreover, we implement them and perform an experimental analysis. Our implementations outperform the state-of-the-art determinant and convex hull implementations in most of the tested scenarios, as well as giving a speed-up of 78 times in point location problems.

Keywords: computational geometry, determinant algorithms, orientation predicate, convex hull, point location, experimental analysis.

1 Introduction

Determinantal predicates are in the core of many important geometric algorithms. Convex hull and regular triangulation algorithms use *orientation* predicates, the Delaunay triangulation algorithms also involve the *in-sphere* predicate. Moreover, algorithms for exact volume computation of a convex polytope rely on determinantal volume formulas. In general dimension d, the orientation predicate of $d+1$ points is the sign of the determinant of a matrix containing the homogeneous coordinates of the points as columns. In a similar way, the volume determinant formula and in-sphere predicate of $d+1$ and $d+2$ points respectively can be defined. In practice, as the dimension grows, a higher percentage of the computation time is consumed by these core procedures. For this reason, we focus on algorithms and implementations for the exact computation of the determinant. We give particular emphasis to division-free algorithms. Avoiding divisions is crucial when working on a ring that is not a field, *e.g.*, integers or polynomials. Determinants of matrices whose elements are in a ring arise in combinatorial problems [21], in algorithms for lattice polyhedra [4] and secondary polytopes [23] or in computational algebraic geometry problems [12].

L. Epstein and P. Ferragina (Eds.): ESA 2012, LNCS 7501, pp. 443–454, 2012.

Our main observation is that, in a sequence of computations of determinants that appear in geometric algorithms, the computation of one predicate can be accelerated by using information from the computation of previously computed predicates. In this paper, we study orientation predicates that appear in convex hull computations. The convex hull problem is probably the most fundamental problem in discrete computational geometry. In fact, the problems of regular and Delaunay triangulations reduce to it.

Our main contribution is twofold. First, we propose an algorithm with quadratic complexity for the determinants involved in a convex hull computation and linear complexity for those involved in point location problems. Moreover, we nominate a variant of this algorithm that can perform computations over the integers. Second, we implement our proposed algorithms along with division-free determinant algorithms from the literature. We perform an experimental analysis of the current state-of-the-art packages for exact determinant computations along with our implementations. Without taking the dynamic algorithms into account, the experiments serve as a case study of the best implementation of determinant algorithms, which is of independent interest. However, dynamic algorithms outperform the other determinant implementations in almost all the cases. Moreover, we implement our method on top of the convex hull package `triangulation` [6] and experimentally show that it attains a speed-up up to 3.5 times, results in a faster than state-of-the-art convex hull package and a competitive implementation for exact volume computation, as well as giving a speed-up of 78 times in point location problems.

Let us review previous work. There is a variety of algorithms and implementations for computing the determinant of a $d \times d$ matrix. By denoting $O(d^\omega)$ their complexity, the best current ω is 2.697263 [20]. However, good asymptotic complexity does not imply good behavior in practice for small and medium dimensions. For instance, LinBox [13] which implements algorithms with state-of-the-art asymptotic complexity, introduces a significant overhead in medium dimensions, and seems most suitable in very high dimensions (typically > 100). Eigen [18] and CGAL [10] implement decomposition methods of complexity $O(n^3)$ and seem to be suitable for low to medium dimensions. There exist algorithms that avoid divisions such as [24] with complexity $O(n^4)$ and [5] with complexity $O(nM(n))$ where $M(n)$ is the complexity of matrix multiplication. In addition, there exists a variety of algorithms for determinant sign computation [8,1]. The problem of computation of several determinants has also been studied. TOPCOM [23], the reference software for computing triangulations of a set of points, efficiently precomputes all orientation determinants that will be needed in the computation and stores their signs. In [15], a similar problem is studied in the context of computational algebraic geometry. The computation of orientation predicates is accelerated by maintaining a hash table of computed minors of the determinants. These minors appear many times in the computation. Although, applying that method to the convex hull computation does not lead to a more efficient algorithm.

Our main tools are the Sherman-Morrison formulas [27,3]. They relate the inverse of a matrix after a small-rank perturbation to the inverse of the original matrix. In [25] these formulas are used in a similar way to solve the dynamic transitive closure problem in graphs.

The paper is organized as follows. Sect. 2 introduces the dynamic determinant algorithms and the following section presents their application to the convex hull problem. Sect. 4 discusses the implementation, experiments, and comparison with other software. We conclude with future work.

2 Dynamic Determinant Computations

In the *dynamic determinant problem*, a $d \times d$ matrix A is given. Allowing some preprocessing, we should be able to handle updates of elements of A and return the current value of the determinant. We consider here only non-singular updates, *i.e.*, updates that do not make A singular. Let $(A)_i$ denote the i-th column of A, and e_i the vector with 1 in its i-th place and 0 everywhere else.

Consider the matrix A', resulting from replacing the i-th column of A by a vector u. The Sherman-Morrison formula [27,3] states that $(A + wv^T)^{-1} = A^{-1} - \frac{(A^{-1}w)(v^T A^{-1})}{1+v^T A^{-1} w}$. An i-th column update of A is performed by substituting $v = e_i$ and $w = u - (A)_i$ in the above formula. Then, we can write A'^{-1} as follows.

$$A'^{-1} = (A + (u - (A)_i)e_i^T)^{-1} = A^{-1} - \frac{(A^{-1}(u - (A)_i))\ (e_i^T A^{-1})}{1 + e_i^T A^{-1}(u - (A)_i)} \qquad (1)$$

If A^{-1} is computed, we compute A'^{-1} using Eq. 1 in $3d^2 + 2d + O(1)$ arithmetic operations. Similarly, the matrix determinant lemma [19] gives Eq. 2 below to compute $det(A')$ in $2d + O(1)$ arithmetic operations, if $det(A)$ is computed.

$$det(A') = det(A + (u - (A)_i)e_i^T) = (1 + e_i^T A^{-1}(u - (A)_i) det(A) \qquad (2)$$

Eqs. 1 and 2 lead to the following result.

Proposition 1. [27] *The dynamic determinant problem can be solved using $O(d^\omega)$ arithmetic operations for preprocessing and $O(d^2)$ for non-singular one column updates.*

Indeed, this computation can also be performed over a ring. To this end, we use the adjoint of A, denoted by A^{adj}, rather than the inverse. It holds that $A^{adj} = det(A)A^{-1}$, thus we obtain the following two equations.

$$A'^{adj} = \frac{1}{\det(A)}(A^{adj} \det(A') - (A^{adj}(u - (A)_i))\ (e_i^T A^{adj})) \qquad (3)$$

$$det(A') = det(A) + e_i^T A^{adj}(u - (A)_i) \qquad (4)$$

The only division, in Eq. 3, is known to be exact, *i.e.*, its remainder is zero. The above computations can be performed in $5d^2 + d + O(1)$ arithmetic operations for Eq. 3 and in $2d + O(1)$ for Eq. 4. In the sequel, we will call *dyn_inv* the dynamic determinant algorithm which uses Eqs. 1 and 2, and *dyn_adj* the one which uses Eqs. 3 and 4.

3 Geometric Algorithms

We introduce in this section our methods for optimizing the computation of sequences of determinants that appear in geometric algorithms. First, we use dynamic determinant computations in incremental convex hull algorithms. Then, we show how this solution can be extended to point location in triangulations.

Let us start with some basic definitions from discrete and computational geometry. Let $\mathcal{A} \subset \mathbb{R}^d$ be a pointset. We define the *convex hull* of a pointset $\mathcal{A}$, denoted by $\text{conv}(\mathcal{A})$, as the smallest convex set containing $\mathcal{A}$. A hyperplane *supports* $\text{conv}(\mathcal{A})$ if $\text{conv}(\mathcal{A})$ is entirely contained in one of the two closed half-spaces determined by the hyperplane and has at least one point on the hyperplane. A *face* of $\text{conv}(\mathcal{A})$ is the intersection of $\text{conv}(\mathcal{A})$ with a supporting hyperplane which does not contain $\text{conv}(\mathcal{A})$. Faces of dimension $0, 1, d-1$ are called vertices, edges and facets respectively. We call a face f of $\text{conv}(\mathcal{A})$ *visible* from $a \in \mathbb{R}^d$ if there is a supporting hyperplane of f such that $\text{conv}(\mathcal{A})$ is contained in one of the two closed half-spaces determined by the hyperplane and a in the other. A k-simplex of $\mathcal{A}$ is an affinely independent subset S of $\mathcal{A}$, where $dim(\text{conv}(S)) = k$. A *triangulation* of $\mathcal{A}$ is a collection of subsets of $\mathcal{A}$, the *cells* of the triangulation, such that the union of the cells' convex hulls equals $\text{conv}(\mathcal{A})$, every pair of convex hulls of cells intersect at a common face and every cell is a simplex.

Denote $\boldsymbol{a}$ the vector $(a, 1)$ for $a \in \mathbb{R}^d$. For any sequence C of points $a_i \in \mathcal{A}$, $i = 1 \dots d+1$, we denote A_C its orientation $(d+1) \times (d+1)$ matrix. For every a_i, the column i of A_C contains $\boldsymbol{a}_i$'s coordinates as entries. For simplicity, we assume general position of $\mathcal{A}$ and focus on the Beneath-and-Beyond (BB) algorithm [26]. However, our method can be extended to handle degenerate inputs as in [14, Sect. 8.4], as well as to be applied to any incremental convex hull algorithm by utilizing the dynamic determinant computations to answer the predicates appearing in point location (see Cor. 2). In what follows, we use the dynamic determinant algorithm *dyn_adj*, which can be replaced by *dyn_inv* yielding a variant of the presented convex hull algorithm.

The BB algorithm is initialized by computing a d-simplex of $\mathcal{A}$. At every subsequent step, a new point from $\mathcal{A}$ is inserted, while keeping a triangulated convex hull of the inserted points. Let t be the number of cells of this triangulation. Assume that, at some step, a new point $a \in \mathcal{A}$ is inserted and T is the triangulation of the convex hull of the points of $\mathcal{A}$ inserted up to now. To determine if a facet F is visible from a, an orientation predicate involving a and the points of F has to be computed. This can be done by using Eq. 4 if we know the adjoint matrix of points of the cell that contains F. But, if F is visible, this cell is unique and we can map it to the adjoint matrix corresponding to its points.

Our method (Alg. 1), as initialization, computes from scratch the adjoint matrix that corresponds to the initial d-simplex. At every incremental step, it computes the orientation predicates using the adjoint matrices computed in previous steps and Eq. 4. It also computes the adjoint matrices corresponding to the new cells using Eq. 3. By Prop. 1, this method leads to the following result.

Algorithm 1: Incremental Convex Hull ($\mathcal{A}$)

Input : pointset $\mathcal{A} \subset \mathbb{R}^d$
Output : convex hull of $\mathcal{A}$
sort $\mathcal{A}$ by increasing lexicographic order of coordinates, *i.e.*, $\mathcal{A} = \{a_1, \ldots, a_n\}$;
$T \leftarrow \{\, d\text{-face of } conv(a_1, \ldots, a_{d+1})\}$; // $conv(a_1, \ldots, a_{d+1})$ is a d-simplex
$Q \leftarrow \{\, \text{facets of } conv(a_1, \ldots, a_{d+1})\}$;
compute $A^{adj}_{\{a_1,\ldots,a_{d+1}\}}, \det(A_{\{a_1,\ldots,a_{d+1}\}})$;
foreach $a \in \{a_{d+2}, \ldots, a_n\}$ **do**
  $Q' \leftarrow Q$;
  foreach $F \in Q$ **do**
    $C \leftarrow$ the unique d-face s.t. $C \in T$ and $F \in C$;
    $u \leftarrow$ the unique vertex s.t. $u \in C$ and $u \notin F$;
    $C' \leftarrow F \cup \{a\}$;
    $i \leftarrow$ the index of u in A_C;
    // both $det(A_C)$ and A_C^{adj} were computed in a previous step
    $det(A_{C'}) \leftarrow det(A_C) + (A_C^{adj})^i(\boldsymbol{u} - \boldsymbol{a})$;
    if $\det(A_{C'})\det(A_C) < 0$ ***and*** $\det(A_{C'}) \neq 0$ **then**
      $A^{adj}_{C'} \leftarrow \frac{1}{\det(A_C)}(A_C^{adj}\det(A_{C'}) - A_C^{adj}(\boldsymbol{u} - \boldsymbol{a})(e_i^T A_C^{adj}))$;
      $T \leftarrow T \cup \{d\text{-face of } conv(C')\}$;
      $Q' \leftarrow Q' \ominus \{(d-1)\text{-faces of } C'\}$; // symmetric difference
  $Q \leftarrow Q'$;
return Q;

Theorem 1. *Given a d-dimensional pointset all, except the first, orientation predicates of incremental convex hull algorithms can be computed in $O(d^2)$ time and $O(d^2 t)$ space, where t is the number of cells of the constructed triangulation.*

Essentially, this result improves the computational complexity of the determinants involved in incremental convex hull algorithms from $O(d^\omega)$ to $O(d^2)$. To analyze the complexity of Alg. 1, we bound the number of facets of Q in every step of the outer loop of Alg. 1 with the number of $(d-1)$-faces of the constructed triangulation of $\mathrm{conv}(\mathcal{A})$, which is bounded by $(d+1)t$. Thus, using Thm. 1, we have the following complexity bound for Alg. 1.

Corollary 1. *Given n d-dimensional points, the complexity of BB algorithm is $O(n \log n + d^3 nt)$, where $n \gg d$ and t is the number of cells of the constructed triangulation.*

Note that the complexity of BB, without using the method of dynamic determinants, is bounded by $O(n \log n + d^{\omega+1} nt)$. Recall that t is bounded by $O(n^{\lfloor d/2 \rfloor})$ [28, Sect.8.4], which shows that Alg. 1, and convex hull algorithms in general, do not have polynomial complexity. The schematic description of Alg. 1 and its coarse analysis is good enough for our purpose: to illustrate the application of dynamic determinant computation to incremental convex hulls and to

quantify the improvement of our method. See Sect. 4 for a practical approach to incremental convex hull algorithms using dynamic determinant computations.

The above results can be extended to improve the complexity of geometric algorithms that are based on convex hulls computations, such as algorithms for regular or Delaunay triangulations and Voronoi diagrams. It is straightforward to apply the above method in orientation predicates appearing in point location algorithms. By using Alg. 1, we compute a triangulation and a map of adjoint matrices to its cells. Then, the point location predicates can be computed using Eq. 4, avoiding the computation of new adjoint matrices.

Corollary 2. *Given a triangulation of a d-dimensional pointset computed by Alg. 1, the orientation predicates involved in any point location algorithm can be computed in $O(d)$ time and $O(d^2t)$ space, where t is the number of cells of the triangulation.*

4 Implementation and Experimental Analysis

We propose the *hashed dynamic determinants* scheme and implement it in C++. The design of our implementation is modular, that is, it can be used on top of either geometric software providing geometric predicates (such as orientation) or algebraic software providing dynamic determinant algorithm implementations. The code is publicly available from `http://hdch.sourceforge.net`.

The hashed dynamic determinants scheme consists of efficient implementations of algorithms *dyn_inv* and *dyn_adj* (Sect. 2) and a hash table, which stores intermediate results (matrices and determinants) based on the methods presented in Sect. 3. Every $(d-1)$-face of a triangulation, *i.e.*, a common facet of two neighbor cells (computed by any incremental convex hull package which constructs a triangulation of the computed convex hull), is mapped to the indices of its vertices, which are used as keys. These are mapped to the adjoint (or inverse) matrix and the determinant of one of the two adjacent cells. Let us illustrate this approach with an example, on which we use the *dyn_adj* algorithm.

Example 1. Let $A = \{a_1 = (0,1), a_2 = (1,2), a_3 = (2,1), a_4 = (1,0), a_5 = (2,2)\}$ where every point a_i has an index i from 1 to 5. Assume we are in some step of an incremental convex hull or point location algorithm and let $T = \{\{1,2,4\}, \{2,3,4\}\}$ be the 2-dimensional triangulation of $\mathcal{A}$ computed so far. The cells of T are written using the indices of the points in $\mathcal{A}$. The hash table will store as keys the set of indices of the 2-faces of T, *i.e.*, $\{\{1,2\},\{2,4\},\{1,4\}\}$ mapping to the adjoint and the determinant of the matrix constructed by the points a_1, a_2, a_4. Similarly, $\{\{2,3\},\{3,4\},\{2,4\}\}$ are mapped to the adjoint matrix and determinant of a_2, a_3, a_4. To insert a_5, we compute the determinant of a_2, a_3, a_5, by querying the hash table for $\{2,3\}$. Adjoint and determinant of the matrix of a_2, a_3, a_4 are returned, and we perform an update of the column corresponding to point a_4, replacing it by a_5 by using Eqs. 3 and 4.

To implement the hash table, we used the Boost libraries [7]. To reduce memory consumption and speed-up look-up time, we sort the lists of indices that form the

hash keys. We also use the *GNU Multiple Precision arithmetic library* (GMP), the current standard for multiple-precision arithmetic, which provides integer and rational types `mpz_t` and `mpq_t`, respectively.

We perform an experimental analysis of the proposed methods. All experiments ran on an Intel Core i5-2400 3.1GHz, with 6MB L2 cache and 8GB RAM, running 64-bit Debian GNU/Linux. We divide our tests in four scenarios, according to the number type involved in computations: (a) rationals where the bit-size of both numerator and denominator is 10000, (b) rationals converted from `doubles`, that is, numbers of the form $m \times 2^p$, where m and p are integers of bit-size 53 and 11 respectively, (c) integers with bit-size 10000, and (d) integers with bit-size 32. However, it is rare to find in practice input coefficients of scenarios (a) and (c). Inputs are usually given as 32 or 64-bit numbers. These inputs correspond to the coefficients of scenario (b). Scenario (d) is also very important, since points with integer coefficients are encountered in many combinatorial applications (see Sect. 1).

We compare state-of-the-art software for exact computation of the determinant of a matrix. We consider LU decomposition in CGAL [10], optimized LU decomposition in Eigen [18], LinBox asymptotically optimal algorithms [13] (tested only on integers) and Maple 14 `LinearAlgebra[Determinant]` (the default determinant algorithm). We also implemented two division-free algorithms: Bird's [5] and Laplace expansion [22, Sect.4.2]. Finally, we consider our implementations of *dyn_inv* and *dyn_adj*.

We test the above implementations in the four coefficient scenarios described above. When coefficients are integer, we can use integer exact division algorithms, which are faster than quotient-remainder division algorithms. In this case, Bird, Laplace and *dyn_adj* enjoy the advantage of using the number type `mpz_t` while the rest are using `mpq_t`. The input matrices are constructed starting from a random $d \times d$ matrix, replacing a randomly selected column with a random d vector. We present experimental results of the most common in practice input scenarios (b), (d) (Tables 1, 2). The rest will appear in the full version of the paper. We stop testing an implementation when it is slow and far from being the fastest (denoted with '-' in the Tables).

On one hand, the experiments show the most efficient determinant algorithm implementation in the different scenarios described, without considering the dynamic algorithms. This is a result of independent interest, and shows the efficiency of division-free algorithms in some settings. The simplest determinant algorithm, Laplace expansion, proved to be the best in all scenarios, until dimension 4 to 6, depending on the scenario. It has exponential complexity, thus it is slow in dimensions higher than 6 but it behaves very well in low dimensions because of the small constant of its complexity and the fact that it performs no divisions. Bird is the fastest in scenario (c), starting from dimension 7, and in scenario (d), in dimensions 7 and 8. It has also a small complexity constant, and performing no divisions makes it competitive with decomposition methods (which have better complexity) when working with integers. CGAL and Eigen implement LU decomposition, but the latter is always around two times faster. Eigen is the fastest implementation in scenarios (a) and (b), starting from di-

Table 1. Determinant tests, inputs of scenario (b): rationals converted from `double`. Each timing (in milliseconds) corresponds to the average of computing 10000 (for $d < 7$) or 1000 (for $d \geq 7$) determinants. Light blue highlights the best non-dynamic algorithm while yellow highlights the dynamic algorithm if it is the fastest over all.

d	Bird	CGAL	Eigen	Laplace	Maple	*dyn_inv*	*dyn_adj*
3	.013	.021	.014	.008	.058	.046	.023
4	.046	.050	.033	.020	.105	.108	.042
5	.122	.110	.072	.056	.288	.213	.067
6	.268	.225	.137	.141	.597	.376	.102
7	.522	.412	.243	.993	.824	.613	.148
8	.930	.710	.390	–	1.176	.920	.210
9	1.520	1.140	.630	–	1.732	1.330	.310
10	2.380	1.740	.940	–	2.380	1.830	.430
11	–	2.510	1.370	–	3.172	2.480	.570
12	–	3.570	2.000	–	4.298	3.260	.760
13	–	4.960	2.690	–	5.673	4.190	1.020
14	–	6.870	3.660	–	7.424	5.290	1.360
15	–	9.060	4.790	–	9.312	6.740	1.830

mension 5 and 6 respectively, as well as in scenario (d) in dimensions between 9 and 12. It should be stressed that decomposition methods are the current standard to implement determinant computation. Maple is the fastest only in scenario (d), starting from dimension 13. In our tests, Linbox is never the best, due to the fact that it focuses on higher dimensions.

On the other hand, experiments show that *dyn_adj* outperforms all the other determinant algorithms in scenarios (b), (c), and (d). On each of these scenarios, there is a threshold dimension, starting from which *dyn_adj* is the most efficient, which happens because of its better asymptotic complexity. In scenarios (c) and (d), with integer coefficients, division-free performs much better, as expected, because integer arithmetic is faster than rational. In general, the sizes of the coefficients of the adjoint matrix are bounded. That is, the sizes of the operands of the arithmetic operations are bounded. This explains the better performance of *dyn_adj* over the *dyn_inv*, despite its worse arithmetic complexity.

For the experimental analysis of the behaviour of dynamic determinants used in convex hull algorithms (Alg. 1, Sect. 3), we experiment with four state-of-the-art exact convex hull packages. Two of them implement incremental convex hull algorithms: `triangulation` [6] implements [11] and `beneath-and-beyond (bb)` in `polymake` [17]. The package `cdd` [16] implements the double description method, and `lrs` implements the gift-wrapping algorithm using reverse search [2]. We propose and implement a variant of `triangulation`, which we will call `hdch`, implementing the hashed dynamic determinants scheme for dimensions higher than 6 (using Eigen for initial determinant and adjoint or inverse matrix computation) and using Laplace determinant algorithm for lower dimensions. The main difference between this implementation and Alg. 1 of Sect. 3 is that it does not sort the points and, before inserting a point, it performs a point location. Thus, we can take advantage of our scheme in two places: in the orientation predicates appearing in the point location procedure and in the ones that

Table 2. Determinant tests, inputs of scenario (d): integers of bit-size 32. Times in milliseconds, averaged over 10000 tests. Highlighting as in Table 1.

d	Bird	CGAL	Eigen	Laplace	Linbox	Maple	*dyn_inv*	*dyn_adj*
3	.002	.021	.013	.002	.872	.045	.030	.008
4	.012	.041	.028	.005	1.010	.094	.058	.015
5	.032	.080	.048	.016	1.103	.214	.119	.023
6	.072	.155	.092	.040	1.232	.602	.197	.033
7	.138	.253	.149	.277	1.435	.716	.322	.046
8	.244	.439	.247	–	1.626	.791	.486	.068
9	.408	.689	.376	–	1.862	.906	.700	.085
10	.646	1.031	.568	–	2.160	1.014	.982	.107
11	.956	1.485	.800	–	10.127	1.113	1.291	.133
12	1.379	2.091	1.139	–	13.101	1.280	1.731	.160
13	1.957	2.779	1.485	–	–	1.399	2.078	.184
14	2.603	3.722	1.968	–	–	1.536	2.676	.222
15	3.485	4.989	2.565	–	–	1.717	3.318	.269
16	4.682	6.517	3.391	–	–	1.850	4.136	.333

appear in construction of the convex hull. We design the input of our experiments parametrized on the number type of the coefficients and on the distribution of the points. The number type is either rational or integer. From now on, when we refer to rational and integer we mean scenario (b) and (d), respectively. We test three uniform point distributions: (i) in the d-cube $[-100, 100]^d$, (ii) in the origin-centered d-ball of radius 100, and (iii) on the surface of that ball.

We perform an experimental comparison of the four above packages and `hdch`, with input points from distributions (i)-(iii) with either rational or integer coefficients. In the case of integer coefficients, we test `hdch` using `mpq_t` (`hdch_q`) or `mpz_t` (`hdch_z`). In this case `hdch_z` is the most efficient with input from distribution (ii) (Fig. 1(a); distribution (i) is similar to this) while in distribution (iii) both `hdch_z` and `hdch_q` perform better than all the other packages (see Fig. 1(b)). In the rational coefficients case, `hdch_q` is competitive to the fastest package (not shown for space reasons). Note that the rest of the packages cannot perform arithmetic computations using `mpz_t` because they are lacking division-free determinant algorithms. Moreover, we perform experiments to test the improvements of hashed dynamic determinants scheme on `triangulation` and their memory consumption. For input points from distribution (iii) with integer coefficients, when dimension ranges from 3 to 8, `hdch_q` is up to 1.7 times faster than `triangulation` and `hdch_z` up to 3.5 times faster (Table 3). It should be noted that `hdch` is always faster than `triangulation`. The sole modification of the determinant algorithm made it faster than all other implementations in the tested scenarios. The other implementations would also benefit from applying the same determinant technique. The main disadvantage of `hdch` is the amount of memory consumed, which allows us to compute up to dimension 8 (Table 3). This drawback can be seen as the price to pay for the obtained speed-up.

A large class of algorithms that compute the exact volume of a polytope is based on triangulation methods [9]. All the above packages compute the

Table 3. Comparison of `hdch_z`, `hdch_q` and `triangulation` using points from distribution (iii) with integer coefficients; `swap` means that the machine used swap memory

$\|\mathcal{A}\|$	d	`hdch_q`		`hdch_z`		`triangulation`	
		time (sec)	memory (MB)	time (sec)	memory (MB)	time (sec)	memory (MB)
260	2	0.02	35.02	0.01	33.48	0.05	35.04
500	2	0.04	35.07	0.02	33.53	0.12	35.08
260	3	0.07	35.20	0.04	33.64	0.20	35.23
500	3	0.19	35.54	0.11	33.96	0.50	35.54
260	4	0.39	35.87	0.21	34.33	0.82	35.46
500	4	0.90	37.07	0.47	35.48	1.92	37.17
260	5	2.22	39.68	1.08	38.13	3.74	39.56
500	5	5.10	45.21	2.51	43.51	8.43	45.34
260	6	14.77	1531.76	8.42	1132.72	20.01	55.15
500	6	37.77	3834.19	21.49	2826.77	51.13	83.98
220	7	56.19	6007.08	32.25	4494.04	90.06	102.34
320	7	`swap`	`swap`	62.01	8175.21	164.83	185.87
120	8	86.59	8487.80	45.12	6318.14	151.81	132.70
140	8	`swap`	`swap`	72.81	8749.04	213.59	186.19

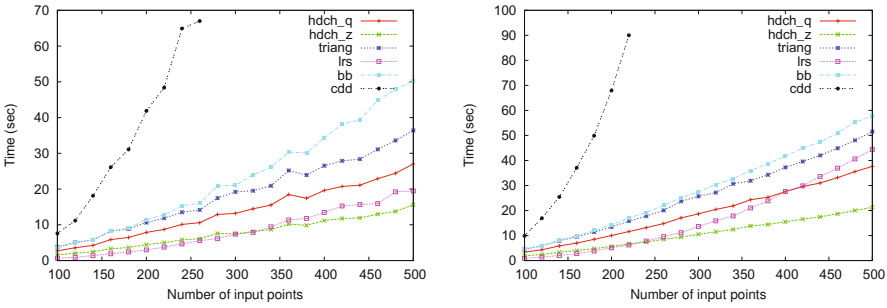

Fig. 1. Comparison of convex hull packages for 6-dimensional inputs with integer coefficients. Points are uniformly distributed (a) inside a 6-ball and (b) on its surface.

volume of the polytope, defined by the input points, as part of the convex hull computation. The volume computation takes place at the construction of the triangulation during a convex hull computation. The sum of the volumes of the cells of the triangulation equals the volume of the polytope. However, the volume of the cell is the absolute value of the orientation determinant of the points of the cell and these values are computed in the course of the convex hull computation. Thus, the computation of the volume consumes no extra time besides the convex hull computation time. Therefore, `hdch` yields a competitive implementation for the exact computation of the volume of a polytope given by its vertices (Fig. 1).

Finally, we test the efficiency of hashed dynamic determinants scheme on the point location problem. Given a pointset, `triangulation` constructs a data structure that can perform point locations of new points. In addition to that, `hdch` constructs a hash table for faster orientation computations. We perform tests with

Table 4. Point location time of 1K and 1000K (1K=1000) query points for `hdch_z` and `triangulation` (`triang`), using distribution (iii) for preprocessing and distribution (i) for queries and integer coefficients

	d	$\lvert\mathcal{A}\rvert$	preprocess time (sec)	data structures memory (MB)	# of cells in triangulation	query time (sec) 1K	query time (sec) 1000K
`hdch_z`	8	120	45.20	6913	319438	0.41	392.55
`triang`	8	120	156.55	134	319438	14.42	14012.60
`hdch_z`	9	70	45.69	6826	265874	0.28	276.90
`triang`	9	70	176.62	143	265874	13.80	13520.43
`hdch_z`	10	50	43.45	6355	207190	0.27	217.45
`triang`	10	50	188.68	127	207190	14.40	14453.46
`hdch_z`	11	39	38.82	5964	148846	0.18	189.56
`triang`	11	39	181.35	122	148846	14.41	14828.67

`triangulation` and `hdch` using input points uniformly distributed on the surface of a ball (distribution (iii)) as a preprocessing to build the data structures. Then, we perform point locations using points uniformly distributed inside a cube (distribution (i)). Experiments show that our method yields a speed-up in query time of a factor of 35 and 78 in dimension 8 to 11, respectively, using points with integer coefficients (scenario (d)) (see Table 4).

5 Future Work

It would be interesting to adapt our scheme for gift-wrapping convex hull algorithms and implement it on top of packages such as [2]. In this direction, our scheme should also be adapted to other important geometric algorithms, such as Delaunay triangulations.

In order to overcome the large memory consumption of our method, we shall exploit hybrid techniques. That is, to use the dynamic determinant hashing scheme as long as there is enough memory and subsequently use the best available determinant algorithm (Sect. 4), or to clean periodically the hash table.

Another important experimental result would be to investigate the behavior of our scheme using filtered computations.

Acknowledgments. The authors are partially supported from project "Computational Geometric Learning", which acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 255827. We thank I.Z. Emiris for his advice and support, as well as M. Karavelas and E. Tsigaridas for discussions and bibliographic references.

References

1. Abbott, J., Bronstein, M., Mulders, T.: Fast deterministic computation of determinants of dense matrices. In: ISSAC, pp. 197–203 (1999)
2. Avis, D.: lrs: A revised implementation of the reverse search vertex enumeration algorithm. In: Polytopes - Combinatorics and Computation, Oberwolfach Seminars, vol. 29, pp. 177–198. Birkhäuser-Verlag (2000)

3. Bartlett, M.S.: An inverse matrix adjustment arising in discriminant analysis. The Annals of Mathematical Statistics 22(1), 107–111 (1951)
4. Barvinok, A., Pommersheim, J.E.: An algorithmic theory of lattice points in polyhedra. New Perspectives in Algebraic Combinatorics, 91–147 (1999)
5. Bird, R.: A simple division-free algorithm for computing determinants. Inf. Process. Lett. 111, 1072–1074 (2011)
6. Boissonnat, J.D., Devillers, O., Hornus, S.: Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension. In: SoCG, pp. 208–216 (2009)
7. Boost: peer reviewed C++ libraries, `http://www.boost.org`
8. Brönnimann, H., Emiris, I., Pan, V., Pion, S.: Sign determination in Residue Number Systems. Theor. Comp. Science 210(1), 173–197 (1999)
9. Büeler, B., Enge, A., Fukuda, K.: Exact volume computation for polytopes: A practical study (1998)
10. CGAL: Computational geometry algorithms library, `http://www.cgal.org`
11. Clarkson, K., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom.: Theory & Appl. 3, 185–212 (1993)
12. Cox, D.A., Little, J., O'Shea, D.: Using Algebraic Geometry. Graduate Texts in Mathematics. Springer, Heidelberg (2005)
13. Dumas, J.G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B., Turner, W., Villard, G.: Linbox: A generic library for exact linear algebra. In: ICMS, pp. 40–50 (2002)
14. Edelsbrunner, H.: Algorithms in combinatorial geometry. Springer-Verlag New York, Inc., New York (1987)
15. Emiris, I., Fisikopoulos, V., Konaxis, C., Peñaranda, L.: An output-sensitive algorithm for computing projections of resultant polytopes. In: SoCG, pp. 179–188 (2012)
16. Fukuda, K.: cddlib, version 0.94f (2008), `http://www.ifor.math.ethz.ch/~fukuda/cdd_home`
17. Gawrilow, E., Joswig, M.: Polymake: a framework for analyzing convex polytopes, pp. 43–74 (1999)
18. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), `http://eigen.tuxfamily.org`
19. Harville, D.A.: Matrix algebra from a statistician's perspective. Springer, New York (1997)
20. Kaltofen, E., Villard, G.: On the complexity of computing determinants. Computational Complexity 13, 91–130 (2005)
21. Krattenthaler, C.: Advanced determinant calculus: A complement. Linear Algebra Appl. 411, 68 (2005)
22. Poole, D.: Linear Algebra: A Modern Introduction. Cengage Learning (2006)
23. Rambau, J.: TOPCOM: Triangulations of point configurations and oriented matroids. In: Cohen, A., Gao, X.S., Takayama, N. (eds.) Math. Software: ICMS, pp. 330–340. World Scientific (2002)
24. Rote, G.: Division-free algorithms for the determinant and the Pfaffian: algebraic and combinatorial approaches. Comp. Disc. Math., 119–135 (2001)
25. Sankowski, P.: Dynamic transitive closure via dynamic matrix inverse. In: Proc. IEEE Symp. on Found. Comp. Sci., pp. 509–517 (2004)
26. Seidel, R.: A convex hull algorithm optimal for point sets in even dimensions. Tech. Rep. 81-14, Dept. Comp. Sci., Univ. British Columbia, Vancouver (1981)
27. Sherman, J., Morrison, W.J.: Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. The Annals of Mathematical Statistics 21(1), 124–127 (1950)
28. Ziegler, G.: Lectures on Polytopes. Springer (1995)