Computational Geometry 54 (2016) 1–16

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Faster geometric algorithms via dynamic determinant computation ☆

Vissarion Fisikopoulos [a,*], Luis Peñaranda [b,*]

[a] Université Libre de Bruxelles, CP 216, Boulevard du Triomphe, 1050 Brussels, Belgium
[b] Universidade Federal do Rio de Janeiro, Departamento de Ciência da Computação, Av. Athos de Silveira Ramos 274, Rio de Janeiro 21941-916, Brazil

ARTICLE INFO

Article history:
Received 17 December 2014
Received in revised form 16 July 2015
Accepted 2 December 2015
Available online 14 December 2015

Keywords:
Determinant algorithms
Orientation predicate
Volume computation
Rank-1 updates
Experimental analysis

ABSTRACT

The computation of determinants or their signs is the core procedure in many important geometric algorithms, such as convex hull, volume and point location. As the dimension of the computation space grows, a higher percentage of the total computation time is consumed by these computations. In this paper we study the sequences of determinants that appear in geometric algorithms. The computation of a single determinant is accelerated by using the information from the previous computations in that sequence.

We propose two dynamic determinant algorithms with quadratic arithmetic complexity when employed in convex hull and volume computations, and with linear arithmetic complexity when used in point location problems. We implement the proposed algorithms and perform an extensive experimental analysis. On one hand, our analysis serves as a performance study of state-of-the-art determinant algorithms and implementations. On the other hand, we demonstrate the supremacy of our methods over state-of-the-art implementations of determinant and geometric algorithms. Our experimental results include a 20 and 78 times speed-up in volume and point location computations in dimension 6 and 11 respectively.

1. Introduction

Computing the sign of a determinant, or in other words evaluating a determinant *predicate*, is in the core of many important geometric algorithms. For example, convex hull algorithms use *orientation* predicates, and Delaunay triangulation algorithms involve *in-sphere* predicates. Furthermore, the computation of the value of a determinant, or in other words a determinant *construction*, is also important in some geometric algorithms. For example, the exact volume computation of a convex polytope using either triangulation or sign decomposition method relies on the computation of the volume of simplices, which reduces to computing the value of a determinant.

In other words predicates encapsulate decisions in contrast to constructions that involve computation of new numerical values. In general dimension d, the orientation predicate of $d+1$ points is the sign of the determinant of a matrix containing

☆ This work is partially supported by 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. The main part of this work was done while the authors were at the University of Athens.

* Both authors are corresponding authors.
E-mail addresses: vfisikop@ulb.ac.be (V. Fisikopoulos), luisp@ufrj.br (L. Peñaranda).

http://dx.doi.org/10.1016/j.comgeo.2015.12.001
0925-7721/

the homogeneous coordinates of the points as columns. On the other hand, the volume of a simplex is the value of the determinant of a matrix containing the homogeneous coordinates of the $d+1$ vertices of the simplex. In practice, as the dimension grows, a higher percentage of the computation time is consumed by these core procedures.

In this paper, we study effective algorithms and implementations for the computation of the determinant predicates and constructions that appear in geometric computations. The model we follow is the exact computation paradigm presented in [1] and advocated by the Computation Geometry Algorithms Library (CGAL) [2], a state-of-the-art library for geometric computations. Note that in geometric algorithms the naive use of floating point arithmetic may lead to incorrect results [3]. There are two main scenarios regarding exactness. The first provides exact predicates but not necessarily exact constructions while the second provides both exact predicates and exact constructions. In this paper we study the second scenario. We give a particular emphasis on exact division and division-free algorithms. Avoiding divisions is crucial when working on a ring that is not a field, *e.g.*, integers or polynomials.

The main idea of our approach is to study the *sequence* of computations of determinants or signs of determinants that appear in geometric algorithms. A single computation can be accelerated by using the information from the previous computations in this sequence. The essential case is the sequence of computations of the orientation predicates that appear in convex hull algorithms. The convex hull problem is probably the most fundamental problem in discrete and computational geometry. In fact, the problems of regular, Delaunay triangulations and Voronoi diagrams reduce to it by computing a convex hull in one dimension higher [4]. Additionally, in the course of an incremental convex hull algorithm like Beneath-and-Beyond [5] we compute the volume of the polytope as a by-product of the computation. See [6] for a survey on volume computation and relevant implementations.

Since we will study *in practice* the performance of geometric and algebraic algorithms, it is important to classify the test cases. Especially, one of the parameters we will use is the dimension. We will refer throughout the paper to dimensions $d < 5$ as *low*, to dimensions $5 \leq d \leq 25$ as *medium* and to dimensions $d > 25$ as *high*, unless otherwise stated. In our experiments, we focus on medium dimensions for determinant computations and "small to medium" for geometric algorithms, *i.e.*, 6 to 11 depending on the application.

1.1. Previous work

There is a variety of algorithms and implementations for computing the determinant of a $d \times d$ matrix. Let us denote by $O(d^{\omega})$ the complexity of matrix multiplication. First, we consider the case where the matrix has values from a field. For $\omega > 2$, an algorithm for matrix multiplication imply an algorithm for determinant computation with the same ω [7]. The best current ω is 2.3728639 [8].

An important class of determinant computation algorithms are the algorithms which use *exact divisions*, *i.e.*, divisions known to have remainder zero. An application of them is the computation of the determinant of a matrix with integer entries using only integer arithmetic. A typical example of this is Bareiss algorithm [9].

Division-free algorithms form another category. They use no divisions at all, *e.g.*, when matrix coefficients are elements of an abstract commutative ring. The best current ω in this category is 2.697263 [10]. Here, it is worth mentioning a family of determinant algorithms that use combinatorial approaches. They were introduced by Mahajan and Vinay [11], and are based on *clow* (closed ordered walk) sequences. Several similar methods with complexity $O(d^4)$ are surveyed by Rote [12]. Based on the idea of clow sequences Bird introduced a simpler algorithm that uses matrix operations [13]. Its complexity is $O(dM(d))$, where $M(d)$ is the complexity of matrix multiplication. Urbańska conceived a method that uses fast matrix multiplication [14] to obtain a complexity $O(d^{3.03})$ [15]. However, in practice when d is small, Bird's algorithm behaves better than other division-free algorithms, as it will be discussed in Section 4.3.

Determinants of matrices over a ring arise in combinatorial problems [16], in algorithms for lattice polyhedra [17] and secondary polytopes [18] or in computational algebraic geometry problems [19]. A special case of the latter is the computation of resultant polytopes that have applications in polynomial system solving [20] and geometric modeling [21].

Good asymptotic complexity does not imply good behavior in practice for low and medium dimensions. For instance, LinBox [22], which implements algorithms with state-of-the-art asymptotic complexity, introduces a significant overhead in low and medium dimensions, and seems most suitable in high dimensions (see Section 4.3 for more details).

Eigen [23] implements LU decomposition, of complexity $O(d^3)$, and seems to be suitable for low and medium dimensions. Eigen was designed with floating-point computations in mind, where it uses hardware floating-point vectors to attain great speed.

In addition, there exists a variety of algorithms for determinant sign computation [24–28]. Kaltofen and Villard [29] present a complete survey on the matter. One tool commonly used for sign computations is *filtering*: arithmetic operations are done using fixed-precision floating-point interval arithmetic, switching to exact arithmetic only when the sign is unknown. Filtered computations are widely used because they provide a simple approach to avoid performing exact operations in many cases. While filtered computation performs well in low dimensions, there is no experimental study on the efficiency of current methods in medium dimensions (see Section 4.6).

The problem of computing sequences of determinants has also been studied. TOPCOM [18] is the reference software for enumerating all regular triangulations of a set of points in general dimension. It efficiently pre-computes all orientation determinants that will be needed in the computation and stores their signs. Emiris et al. [30] study a similar problem in the context of computational algebraic geometry. In particular, the computation of several regular triangulations for different lifting functions. The computation of orientation predicates is accelerated by maintaining a hash table with the computed minors of the determinants. These minors appear many times in the computation. However, this method does not provide considerable acceleration when applied to the case of a single convex hull computation.

Our approach utilizes the Sherman–Morrison formulas [31,32]. They relate the inverse of a matrix after a small-rank perturbation to the inverse of the original matrix. Other applications of these formulas include solving the dynamic transitive closure problem in graphs [33] and studying the effect of new links on Google Page Rank [34].

1.2. Contribution

We design algorithms that perform dynamic determinant updates and achieve quadratic complexity for the determinants involved in incremental convex hull or volume computation algorithms and linear complexity for determinants involved in point location algorithms. Interestingly, we propose a variant of these algorithms that can perform computations over the integers. Our main technical tool is Sherman–Morrison formulas. As far as we know this is the first application of these formulas to geometric algorithms.

We implement the proposed algorithms along with division-free determinant algorithms from the literature. We perform an extensive experimental analysis of the current state-of-the-art packages for exact determinant computations along with our implementations. Without taking the dynamic determinant algorithms into account, our experiments present a result of independent interest: they serve as a *survey* of state-of-the-art determinant algorithms and implementations. In the division-free case, with matrices containing very large integer values, we show that the simple and not-widely used algorithm due to Bird [13] outperforms state-of-the-art implementations in dimensions $6 < d < 10$, while providing a very competitive performance for higher dimensions. Dynamic algorithms start outperform all the other tested determinant implementations in dimension 6 when the input has small bit-size. For larger bit-size, the dynamic algorithms become competitive in larger dimensions ($d > 23$ in our tests).

We adapt our implementations to work with geometric algorithms, thus providing exact predicates and constructions. A natural geometric context to test our method is exact volume computation, where it yields very competitive implementations. For instance, we obtain an up to 20 times speed-up comparing with state-of-the-art packages in dimension 6. We also provide experimental results showing that our method improves the running time of convex hull and point location implementations with respect to other exact implementations. Another interesting feature of our method is that it takes advantage of multiple precision integer, as opposed to rational, arithmetic when the input coordinates are integral (*e.g.* lattice polytopes).

Overview of the paper The paper is organized as follows. Section 2 introduces the dynamic determinant algorithms and the following section presents their application to geometric algorithms. Section 4 discusses the implementation, experiments, and comparison with other software. We end up with conclusions and future work.

A preliminary version of the results of this paper appeared [35]. In this final version, we include new results on volume computations and experiments on real practical scenarios. We also present more experimental results on determinant and convex hull computations and discuss issues as filtering and memory consumption. Overall, we provide an improved and more detailed presentation of our method.

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, that is, updates that do not make A singular. This assumption is sufficient for our method as we explain at the end of Section 3.2.

The Sherman–Morrison formula [31,32] states that

$$\left(A + wv^T\right)^{-1} = A^{-1} - \frac{(A^{-1}w)(v^T A^{-1})}{1 + v^T A^{-1} w}, \tag{1}$$

where A a $d \times d$ matrix and v, w vectors of dimension d. Let A' be the matrix resulting from replacing the i-th column of A by a vector u. Also 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. An i-th column update of A is performed by substituting $v = e_i$ and $w = u - (A)_i$ in Equation (1). Then, we can write A'^{-1} as follows:

$$A'^{-1} = \left(A + (u - (A)_i)e_i^T\right)^{-1} = A^{-1} - \frac{\left(A^{-1}(u - (A)_i)\right)(e_i^T A^{-1})}{1 + e_i^T A^{-1}(u - (A)_i)}, \tag{2}$$

where e_i^T is simply selecting row i. If A^{-1} is computed, we compute A'^{-1} using Equation (2). The computation is performed as follows:

$$h_1 = A^{-1}(u - (A)_i) \tag{3}$$

$$h_2 = h_1/(1 + (h_1)^i) \tag{4}$$

$$H_3 = h_2\,(A^{-1})^i \tag{5}$$

$$A'^{-1} = A^{-1} - H_3 \tag{6}$$

where $(A)^i, (h_1)^i$ denote the i-th row of A and the i-th element h_1 respectively. The intermediate results are the d-dimensional vectors h_1, h_2 and the $d \times d$ matrix H_3. Hence, the equations (3), (4), (5), (6) are computed in $d^2 + d$, $d + O(1)$, d^2, d^2 arithmetic operations respectively and thus $3d^2 + 2d + O(1)$ in total.

The matrix determinant lemma [36] states that

$$\det\left(A + wv^T\right) = \left(1 + v^T A^{-1} w\right) \det(A) \tag{7}$$

which yields the following equation

$$\det\left(A'\right) = \det\left(A + (u - (A)_i)e_i^T\right) = \left(1 + e_i^T A^{-1}(u - (A)_i)\right) \det(A)\,. \tag{8}$$

Using Equation (8) we compute $\det\left(A'\right)$ in $2d + O(1)$ arithmetic operations, if $\det(A)$ is known. Equations (2) and (8) lead to the following result.

Proposition 1. *(See [31].) 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. The preprocessing consists in the computation of A^{-1} and $\det(A)$.*

Then we show how this computation can 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 $A^{\mathrm{adj}} = \det(A)A^{-1}$, thus we obtain the following two equations.

$$A'^{\,\mathrm{adj}} = \frac{1}{\det(A)}\left(A^{\mathrm{adj}} \det(A') - \left(A^{\mathrm{adj}}(u - (A)_i)\right)\,\left(e_i^T A^{\mathrm{adj}}\right)\right) \tag{9}$$

$$\det(A') = \det(A) + e_i^T A^{\mathrm{adj}}(u - (A)_i) \tag{10}$$

The only division in Equation (9) is known to be exact, *i.e.*, its remainder is zero. If the computation follows the order of operations as determined by the parenthesis in Equations (9), (10) then the computation can be performed in $5d^2 + d + O(1)$ arithmetic operations for Equation (9) and in $2d + O(1)$ for Equation (10). In the sequel, we will call *dyn_inv* the dynamic determinant algorithm that uses Equations (2) and (8), and *dyn_adj* the one that uses Equations (9) and (10).

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 utilize dynamic determinants, as described in the previous section, in incremental convex hull algorithm; they form one of the basic classes of convex hull algorithms. Then, we show how this solution can be extended to other geometric algorithms such as point locations in triangulations and volume computations.

3.1. Preliminaries

Let us start with some basic definitions from discrete geometry. Let $\mathcal{A} \subset \mathbb{R}^d$ be a set of n points. We define the *convex hull* of a pointset $\mathcal{A}$, denoted by $\mathrm{conv}(\mathcal{A})$, as the smallest convex set containing $\mathcal{A}$. A hyperplane *supports* $\mathrm{conv}(\mathcal{A})$ if $\mathrm{conv}(\mathcal{A})$ 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 $\mathrm{conv}(\mathcal{A})$ is the intersection of $\mathrm{conv}(\mathcal{A})$ with a supporting hyperplane that does not contain $\mathrm{conv}(\mathcal{A})$. Faces of dimension 0 and $d-1$ are called *vertices* and *facets* respectively. We call a face f of $\mathrm{conv}(\mathcal{A})$ *visible* from $a \in \mathbb{R}^d$ if there is a supporting hyperplane that contains f such that $\mathrm{conv}(\mathcal{A})$ 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 the convex hull of an affinely independent subset S of $\mathcal{A}$, where $\dim(\mathrm{conv}(S)) = k$. A *triangulation* of $\mathcal{A}$ is a collection of simplices of $\mathcal{A}$, called the *cells* of the triangulation, such that the union of the simplices equals $\mathrm{conv}(\mathcal{A})$ and every pair of simplices intersect at a common face or have an empty intersection. We define the *orientation matrix* A_C of a set C of points $\{a_1 \dots a_{d+1}\} \subset \mathbb{R}^d$ to be the $(d+1) \times (d+1)$ matrix such that for every a_i, the column i of A_C contains $\vec{a}_i$'s coordinates as entries, where $\vec{a}_i$ is the homogeneous vector $(a_i, 1)$.

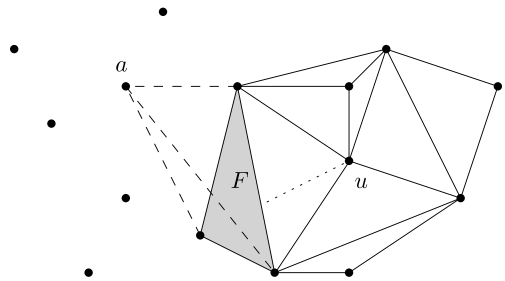

Fig. 1. The course of an incremental convex hull algorithm in 3 dimensions.

Algorithm 1: Incremental convex hull ($\mathcal{A}$).

Input : pointset $\mathcal{A} \subset \mathbb{R}^d$
Output : convex hull of $\mathcal{A}$

```
sort A by increasing lexicographic order of coordinates, i.e., A = {a_1, ..., a_n};
T ← {a_1, ..., a_{d+1}};
Q ← facets of conv(a_1, ..., a_{d+1});
foreach a ∈ {a_{d+2}, ..., a_n} do
    Q' ← Q;
    foreach F ∈ Q do
        C ← the unique d-face s.t. C ∈ T and F ∈ C;
        u ← the unique vertex s.t. u ∈ C and u ∉ F;
        C' ← F ∪ {a};
        // det(A_C) and A^adj were computed in a previous step
        det(A_C') ← (det(A_C) after updating u with a using Equations (9), (10));
        if det(A_C') det(A_C) < 0 then
            T ← T ∪ {d-face of conv(C')};
            Q' ← Q' ⊖ {(d − 1)-faces of C'}; // symmetric difference
        end
    end
    Q ← Q';
end
return Q;
```

3.2. Incremental convex hull

For simplicity, we assume general position of $\mathcal{A}$ and present our method for the Beneath-and-Beyond (BB) algorithm [5]. However, our method can be extended to handle degenerate inputs as in [37, §8.4], and can be applied to more efficient incremental convex hull algorithms (*e.g.*, [38]) by utilizing the dynamic determinant computations to answer the predicates appearing in point location (Corollary 2). A clarification of this claim is our implementation in Section 4 which first handles degenerate inputs in practice and second is faster compared to other software. 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. This choice is supported by our experiments where we show that *dyn_adj* is faster than *dyn_inv* in all the tested dimensions.

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 vertices of F has to be computed (Fig. 1). That is, we have to compute the sign of the determinant of the matrix A_C, where C is the set of vertices of F union with a. If we know the adjoint and the determinant of the orientation matrix of a cell of T that contains F, this can be done by applying Equation (10). If F is on the boundary, this cell is unique (*e.g.*, (F, u) in Fig. 1) otherwise we arbitrarily select one of the two cells that contain F.

Algorithm 1, as initialization, computes from scratch the adjoint matrix and the determinant of the orientation matrix A_C, where C contains the vertices of the initial d-simplex. At every incremental step, it first computes the orientation predicates using the adjoint matrices and determinants computed in previous steps using Equation (10). Second, it computes the adjoint and determinant of the orientation matrices of the new cells using Equation (9). By Proposition 1, this method leads to the following result.

Proposition 2. *Given a d-dimensional pointset the first orientation predicate of incremental convex hull algorithms is computed in $O(d^{\omega})$ time, and all the others in $O(d^2)$ time in total $O(d^2 t)$ space, where t is the number of cells of the constructed triangulation.*

Essentially, this result improves the computational complexity of the predicates involved in incremental convex hull algorithms from $O(d^{\omega})$ to $O(d^2)$ by using more space and dynamic determinant updates. Recall that $O(d^{\omega})$ is the current best complexity (Section 1). To analyze the complexity of Algorithm 1, we bound the number of facets of Q in every step of the outer loop of Algorithm 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 Lemma 2, we have the following complexity bound for Algorithm 1, where we assume that $n \gg d$ to hide the preprocessing complexity $O(d^{\omega})$.

The method of dynamic determinants increases the *space complexity* of BB from $O(nd)$ numbers and $O(td)$ references to $O(td^2)$ numbers and $O(td)$ references. The numbers stored by the two methods are different. The original method stores only point coefficients while ours stores additionally determinants and the inverse and adjoint matrices. The bit-sizes of those numbers are different. The $O(nd)$ point coefficients are part of the input. Let τ be a bound on their bit-sizes. From Hadamard's inequality [39] the value of the determinant of a matrix A is bounded by

$$|\det(A)| \leq 2^{\tau d} d^{d/2}.$$

It follows that the bit-size of the computed determinants is $O(d(\tau + \log d)$, which becomes $O(d\tau)$ under the standard assumption $\tau \gg d$. Since the absolute values of the elements of the adjoint and inverse of A are bounded by the determinant of submatrices of A, the above bound also holds for the bit-size of the elements of the adjoint and inverse matrices.

Corollary 1. *Given n d-dimensional points whose coefficients bit-size is bounded by τ, the complexity of BB algorithm is $O(n \log n + d^3 nt)$, where $n \gg d$, $\tau \gg d$ and t is the number of cells of the constructed triangulation. The consumed space is $O(td^2)$ numbers of bit-size at most $O(d\tau)$ and $O(td)$ references.*

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 $O(n^{\lfloor d/2 \rfloor})$ [40, §8.4], which shows that Algorithm 1, and convex hull algorithms in general, do not have polynomial complexity in n and d. The schematic description of Algorithm 1 and its coarse analysis is good enough for our purpose: to elucidate the application of dynamic determinants to incremental convex hull computation and to quantify the improvements using this method. See Section 4 for a practical approach to incremental convex hull algorithms using dynamic determinant computations.

In Section 2 we have addressed only non-singular updates. Here we show that this will not limit our method to handle degenerate cases. In a degenerate case, the determinant of an orientation matrix will be zero if the points in the orientation test span a space of dimension less than d. However, in this case, we do not have to update the adjoint or the determinant of the orientation matrix (which would be equivalent to a singular update operation), since no new cell is going to be created.

3.3. Point location and volume computation

The above results can be used to improve the efficiency of geometric algorithms that use convex hull computations. One way of computing *Delaunay triangulations* in $\mathbb{R}^d$ and their dual *Voronoi diagrams* is to compute the convex hull of the points lifted on the paraboloid in $\mathbb{R}^{d+1}$. For generic liftings, the above construction leads to regular triangulations.

Another important geometric problem where our method could be applied is *exact volume computation*, since one of the two major classes of volume computation algorithms is based on triangulation methods [6]. To elucidate this, observe that in Algorithm 1 we can compute the volume of the polytope by summing up the volumes of all full dimensional simplices in the resulting triangulation. Indeed, the volume of a simplex is the absolute value of the determinant of its orientation matrix. The difference of an incremental convex hull and a volume computation algorithm using a triangulation method is that the former needs to evaluate determinant predicates (*i.e.*, know only the sign of determinants), while the latter needs determinant constructions (*i.e.*, compute the value of determinants).

As mentioned above, more efficient incremental convex hull algorithms (*e.g.*, the work of Clarkson and Shor [38]) do not sort the input points, they use instead *point location* methods to find the position of the point that is going to be inserted into the convex hull. It is straightforward to apply our scheme in orientation predicates appearing in *point location* algorithms, that perform orientation tests with respect to the facets of the triangulation. The orientation predicates queried by a point location algorithm can be computed using Equation (10), if the adjoint and determinant of the orientation matrices of the cells of the triangulation have been precomputed. That yields the following result.

Corollary 2. *Given a triangulation of a d-dimensional pointset computed by an incremental convex hull algorithm like Algorithm 1, the orientation predicates involved in point location algorithms that perform orientation tests with respect to the facets of the triangulation can be computed in $O(d)$ arithmetic operations, using $O(d^2 t)$ numbers of maximum bit-size $O(d\tau)$ as space, where t is the number of cells of the triangulation and τ bounds the bit-sizes of the numbers, as in Corollary 1.*

4. Implementation and experimental analysis

4.1. Software design

We implemented in C++ the methodology described above, which we call *hashed dynamic determinants*. The scheme consists of efficient implementations of algorithms *dyn_inv* and *dyn_adj* (Section 2) and a hash table, which stores intermediate results such as matrices and determinants. Note that since our implementation computes values of determinants and not only their sign it cannot take advantage of filtering techniques (Section 4.6).

The design of our implementation is *modular*. It can be used by an algebraic software, providing dynamic determinant algorithm implementations. Moreover it can be used by a geometric software providing exact geometric predicates and constructions (*e.g.*, orientation and volume). Here we focus on geometric software that implements incremental convex hull algorithms, which essentially compute a triangulation. Our implementation is independent of the data-structures used by the geometric software. The use of the hash table as an additional data-structure is a way to provide the user with an interface to the new determinant computation without modifying its own data structure. In practice, hash tables have constant insertion and retrieval times, and thus our approach does not introduce a significant overhead in computing time while remaining modular.

The hashing scheme works as follows. Assume that the input points are indexed as $\{a_1, \ldots, a_n\}$. We use as *hash keys* the tuples of indices of the $(d-1)$-faces of the triangulation. Each $(d-1)$-face is mapped to one of the two cells (*i.e.*, d-faces) of the triangulation that it belongs to. The selection between the two cells is arbitrary and does not affect the efficiency of the method. For every cell we also store the adjoint and the determinant of the matrix that corresponds to its vertices' coordinates. In the course of geometric algorithms a given point b should be tested for orientation with respect to a hyperplane defined by points that are locally indexed as $a_1, \ldots, a_d$. Querying the hash table for the tuple $(a_1, \ldots, a_d)$ we obtain the adjoint and the determinant of the matrix with entries the coordinates of $a_1, \ldots, a_d$ and one more point c. Thus, the requested orientation determinant is computed by updating c with b applying Equations (9) and (10). The following 2-dimensional example illustrates our approach.

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 conv(A) computed so far. The cells of T are indexed using the indices of the points in $\mathcal{A}$. For each cell, the hash table will store as keys the set of indices of the 2-faces of the cell, *e.g.*, for the cells $\{1, 2, 4\}$ the keys are $\{\{1, 2\}, \{2, 4\}, \{1, 4\}\}$ 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 in T one should compute the orientation determinant of a_2, a_3, a_5 to determine whether the facet $\{2, 3\}$ is visible from a_5 and hence should be connected to construct a new cell $\{2, 3, 5\}$. Similar computations are performed for the other facets. By querying the hash table for $\{2, 3\}$ the adjoint and the determinant of the matrix of a_2, a_3, a_4 are returned. Then, we perform an update of the column corresponding to point a_4, replacing it by a_5 and apply Equations (9) and (10) to compute the adjoint and the determinant of the new cell. Finally, the two new keys $\{2, 5\}, \{3, 5\}$ are added to the hash table and are mapped to the new cell $\{2, 3, 5\}$.

The hash table has been implemented using the Boost libraries [41]. To reduce memory consumption and speed-up look-up time, we sort the lists of indices that form the hash keys. We 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.

The geometric software we interface with our implementation is the `CGAL` package `Triangulation` [42,43], which implements an incremental convex hull algorithm. The difference between this implementation and Algorithm 1 of Section 3 is that `Triangulation` does not sort the points along one coordinate but along a d-dimensional Hilbert curve and performs a fast point location at every insertion. Thus, we can take advantage of our scheme in two places: (a) in the orientation predicates appearing in the *point location* procedure, and (b) in the ones that appear in the *construction of the convex hull*.

We call `hdch` the modification of `Triangulation` with hashed dynamic determinants. On the technical part, we provide a modification of the `CGAL` Kernel were the call to the determinant is replaced by a functor which implements the dynamic determinant formulas and has access to the hash table. The hash table is completely hidden from the interface. We use `Eigen` for initial determinant and adjoint or inverse matrix computation and Laplace determinant algorithm for dimensions lower than 6.

4.2. Experimental setup

All experiments ran on an Intel Core i5-2400 3.1 GHz, with 6 MB L2 cache and 8 GB 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 10 000,

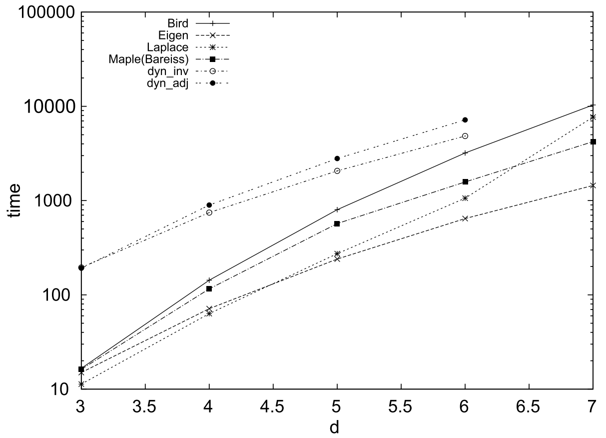

Fig. 2. Determinant experiments, inputs of scenario (a). Each timing (in milliseconds) corresponds to the average of computing 10 000 determinants.

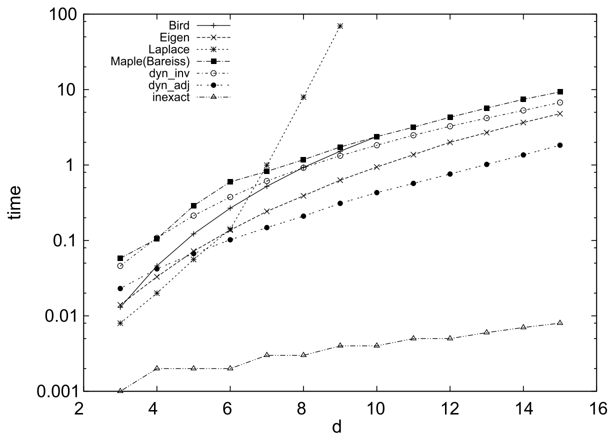

Fig. 3. Determinant experiments, inputs of scenario (b). Each timing (in milliseconds) corresponds to the average of computing 10 000 (for $d < 7$) or 1000 (for $d \geq 7$) determinants.

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 10 000, 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 (Section 1).

4.3. Determinant computation experiments

We compare state-of-the-art software for exact computation of the determinant of a $d \times d$ matrix in the four coefficient scenarios described above. When coefficients are integers, we can use integer exact division algorithms, which are faster than quotient-remainder division algorithms. In this case, division-free algorithms take advantage of using the number type `mpz_t` while the others 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 four input scenarios in Figs. 2–5. We tested a fifth coefficient scenario (rationals of bit-size 32), but do not show results here because timings are quite proportional to those show in Fig. 2. We stop testing an implementation when it is slow and far from being the fastest (denoted by absence of dots in the Figures). On one hand, without considering the dynamic algorithms, the experiments show the most efficient determinant algorithm implementation in the different scenarios described. This is a result of independent interest, and shows the efficiency of division-free algorithms in some settings.

First, we consider LU decomposition, the current standard in determinant implementations. We test `Eigen` [23] which shown to be the fastest in scenarios (a) and (b), starting from dimension 5 and 6 respectively, as well as in scenario (d) in dimensions between 9 to 12.

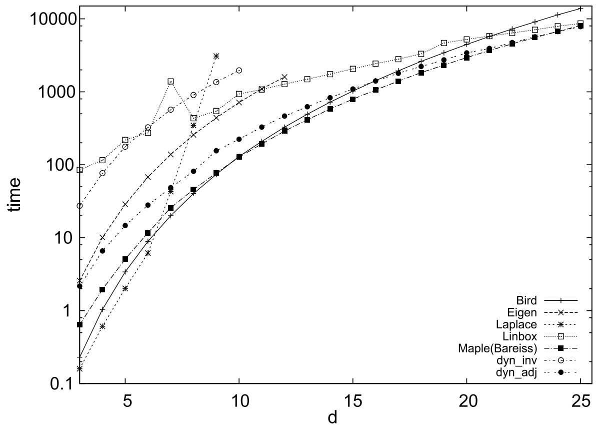

Fig. 4. Determinant experiments, inputs of scenario (c). Times in milliseconds, averaged over 1000 tests for $d < 9$ and 100 tests for $d \geq 9$.

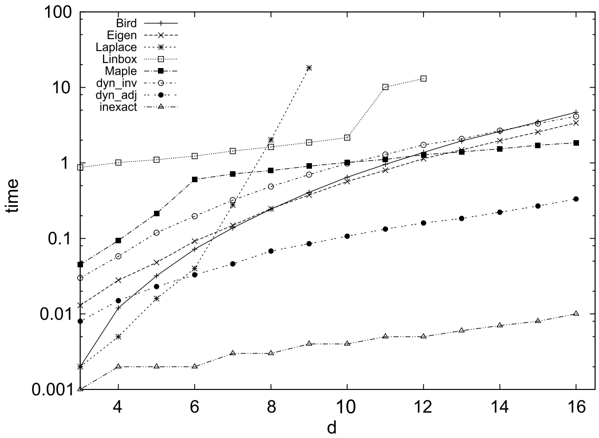

Fig. 5. Determinant experiments, inputs of scenario (d). Times in milliseconds, averaged over 10 000 tests.

Second, we consider determinant algorithms implemented in `LinBox` [22]. `LinBox` implements state-of-the-art algorithms with the best known asymptotic complexity bounds. However, their implementation usually has a big computational overhead and `LinBox` shows the best results only when working in high dimensions (the results of the tests of this section corroborate this claim). `LinBox` provides a myriad of algorithms for computing determinants: many known dense and sparse elimination methods, the block Wiedemann algorithm [44] and an algorithm using a hybrid method mixing Chinese remaindering and last invariant factor [45]. We tested them and used for our tests the faster algorithm for our scenarios (c) and (d),[1] the hybrid elimination algorithm (which is also the default in `LinBox`). `LinBox` is never the best, due to the fact that it focuses on high dimensions. For instance, observe Figs. 4 and 5. In the former, LinBox is competitive only in high dimensions (*i.e.* > 15), but tends to be the most efficient in dimensions larger than 25, for which we didn't perform experiments. In the latter, `LinBox` is at least two times slower than `Maple` until dimension 10. In this case, for larger dimensions, `LinBox` switches the internal algorithm it uses and, while the former relation still holds, timings get much slower than `Maple`.

We consider `Maple` 14 `LinearAlgebra[Determinant]`. `Maple` implementation chooses between Bareiss algorithm [9], Gaussian elimination [46, §2.2] and Berkowitz algorithm [47], based on the properties of the underlying algebraic structure. Note that, for scenario (c), we experimentally check that it is more efficient to force `Maple` to use Bareiss algorithm. Experimental results of that case are presented in Fig. 4. `Maple` is the fastest only in scenario (d), starting from dimension 13.

To test the behavior of the class of division-free combinatorial algorithms, we choose to implement Bird's algorithm [13] despite of the existence of combinatorial algorithms with better asymptotic complexity. Those algorithms are using fast matrix multiplication, which carries big constants in the complexity [48,49]. As reported in [50] implementations of fast matrix multiplications are more efficient for matrices with dimensions bigger than 100. On the other hand, Bird's algorithm does not rely on a particular matrix multiplication algorithm; its complexity is expressed as a function of the complexity of the

[1] For technical reasons, we only tested `LinBox` with integer matrices; however, our results can be readily generalized to the rational case.

matrix multiplication algorithm used. We choose to implement Bird's algorithm using schoolbook matrix multiplication [46, §3.1]: since Bird's algorithm only operates with some rows of upper-triangular matrices, few scalar operations are actually done (only $\frac{1}{4}d^4 + O(d^3)$ scalar multiplications, and the same number of additions, are needed, see Appendix A). Interestingly, naive matrix multiplication makes Bird's algorithm very competitive in small to medium dimensions. It is faster, in cases, than algorithms using fast matrix multiplication and faster than common decomposition methods when working with big integers. In particular, it is the fastest in scenario (c), starting in dimensions 7 to 9, and in scenario (d), in dimensions 7 and 8.

The classic Laplace expansion [46, §4.2] which falls in the category of division-free algorithms is implemented and proved to be the most efficient until dimension 4, 5, 6, 5 for scenario (a)–(d) respectively. It has exponential complexity, but it behaves very well in low dimensions because of the small constant of its complexity and the fact that it performs no divisions.

We consider our implementations of *dyn_inv* and *dyn_adj*. In the initialization step of these algorithms, we compute the inverse, the adjoint and the determinant of the initial matrices using Gaussian elimination. This step affects only infinitesimally the total running time, because it is performed only once, and thus we did not search for optimal implementations of these algorithms. Experiments show that *dyn_adj* defeats the other algorithms in the most common scenarios (b), (d) starting in dimension 6. This happens mainly because of its better asymptotic complexity. In scenario (c), *dyn_adj* beats the most efficient non-dynamic methods (which are the division-free methods) only in high dimensions. It outperforms Bird only in dimension 16, while it is faster than Bareiss only in dimension 24. It worth mentioning that *dyn_adj* performs always better than *dyn_inv*, despite its worse arithmetic complexity. This is somehow because we are working with multiple precision arithmetic, on which the cost of arithmetic operations is a function on the size of the operands. Since the sizes of the coefficients of the adjoint matrix are bounded, the sizes of the operands of the arithmetic operations in *dyn_adj* are also bounded, which is not the case for *dyn_inv*.

Finally, we report results of inexact computation for scenarios (b) and (d), that is, `Eigen` using double-precision floating-point arithmetic (denoted by *inexact* in Figs. 3 and 5). Though largely faster than the timings of exact computations, the correct value of the determinant is not computed. These experiments provide an insight of the timings one would obtain using filtered computations, in the ideal case that no exact computation needs to be done. See Section 4.6 for a discussion on filtering.

4.4. Geometric computation experiments

We perform an experimental analysis on the behavior of the application of dynamic determinants in geometric computation. Our main focus is to provide exact determinant constructions to volume computation. Since convex hull computation is closely connected to volume computation (cf. Section 3) we study also convex hull algorithms. We experiment with four state-of-the-art convex hull packages. Two of them implement incremental convex hull algorithms: `Triangulation` [42] implements [51] and `beneath-and-beyond (bb)` implements the Beneath-and-Beyond algorithm in `polymake` [52]. The package `cdd` [53] implements the double description method, and `lrs` implements the gift-wrapping algorithm using reverse search [54]. All packages apart from `cdd` can be used to compute volumes of polytopes. We show that the application of our method accelerates `Triangulation` and outperforms other software.

We design the input of our experiments parametrized on the number type of the coefficients and on the distribution of the points. We test our method with *synthetic data* first. 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 the three uniform point distributions described below. When the performance of the tested algorithms on two different distributions is similar, we present the results that correspond to only one of the 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 the ball of (ii).

First, we test our method against volume computation provided by `lrs`. Our software in dimension 6 can be up to 20 times faster (Fig. 6). This is an experimental evidence that our method could be used to compute volumes of polytopes for which state-of-the-art methods halt. Also note that the algorithms of `vinci` [55] another state-of-the-art software for exact volume computation were not faster than `lrs` in our experiments. In particular, the only available algorithm that `vinci` provides when the input polytope representation is given by points and the inequalities are not known uses `lrs`.

Second, we perform an experimental comparison of the four convex hull 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. 7(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. 7(b)). In the rational coefficients case, `hdch_q` is competitive to the fastest package (Fig. 8). Note that the rest of the packages cannot perform arithmetic computations using `mpz_t` because they are lacking division-free determinant algorithms. 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.

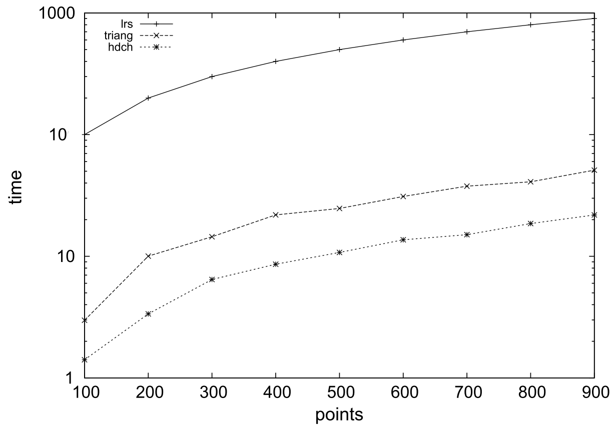

Fig. 6. Volume computation experiments; input is random points in a cube of dimension 6; *i.e.* distribution (i). Times in seconds averaged over 100 tests.

Table 1
Comparison of `hdch_q`, `hdch_z` and `Triangulation`. Points from distribution (iii) with integer coefficients; `swap` means that the machine used `swap` memory. Times averaged over 100 tests.

n	*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

Table 2
Computing resultant polytopes. Times averaged over 100 tests.

n	*d*	Time (sec)			Volume
		`hdch_q`	`hdch_z`	`Triangulation`	
80	6	0.54	0.27	0.66	368 986.7
100	6	0.69	0.33	0.87	108 096.3
110	6	1.20	0.52	1.40	1 456 226 058.5
125	6	1.28	0.61	1.66	66137.3
376	7	17.07	7.80	24.41	1 713 149 926.2
414	7	23.02	10.91	32.54	82 132 445.9
500	7	29.40	13.05	41.22	2 593 047 991.6
528	7	38.22	17.96	54.91	33 727 790.7

Moreover, we quantify the improvements of hashed dynamic determinants scheme on `Triangulation`. 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 (see Table 1). Table 1 also quantifies the memory consumption needed to obtain these speed-ups.

We emphasize the utilization of the hashed dynamic determinants scheme when working with *real data*. We carry out experiments using as input several resultant polytopes. These polytopes are fundamental in algebraic geometry [56] and have been also studied from a computational point of view [30]. The list of their applications contains polynomial system solving and computer aided design [30]. A basic property of these polytopes is that their vertices have integral coefficients. The results in Table 2 show a speed-up of up to 3 times using `hdch_z` with respect to `Triangulation`. The last column shows the exact volume computed for these polytopes.

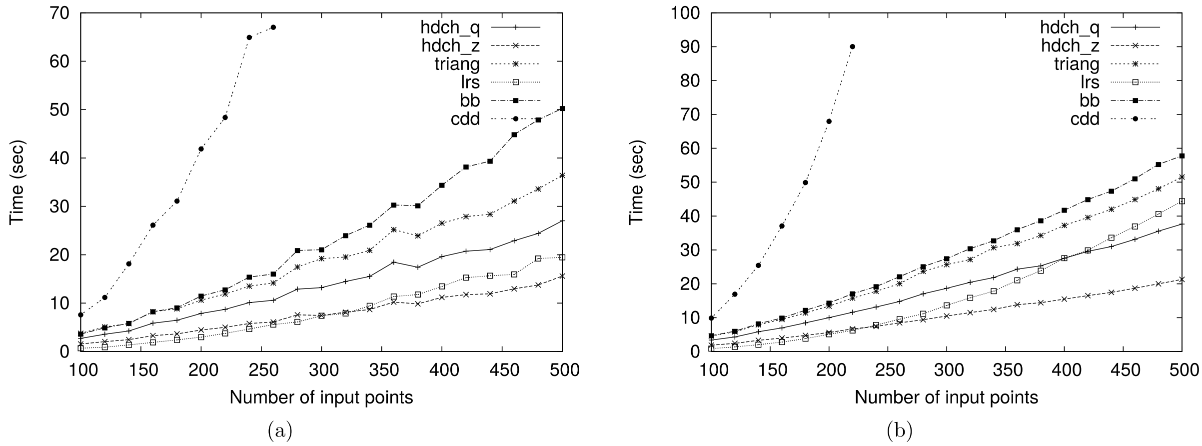

Fig. 7. Comparison of convex hull packages for 6-dimensional inputs with integer coefficients. Points are uniformly distributed (a) inside a 6-ball and (b) on its surface. Times averaged over 100 tests.

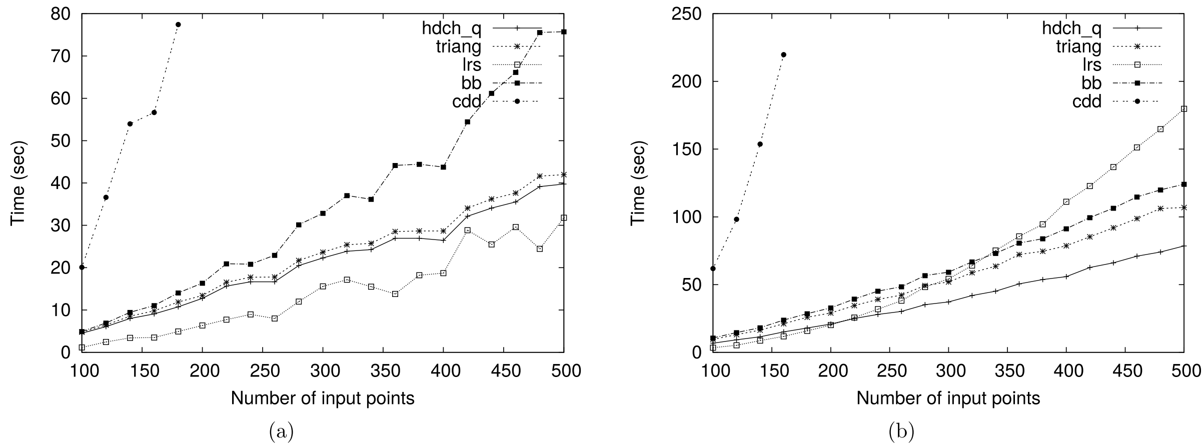

Fig. 8. Comparison of convex hull packages for 6-dimensional inputs with rational coefficients. Points are uniformly distributed (a) inside a 6-ball and (b) on its surface. Times averaged over 100 tests.

We test the efficiency of hashed dynamic determinants scheme on the *point location* problem in a triangulation. Given a pointset, `Triangulation` constructs a triangulation of the convex hull of the pointset and a data structure that can perform point locations of new points. In addition to that, `hdch` constructs the hash table with matrices and determinants used for faster orientation computations. We perform tests with `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 by a factor of 35 to 78 when dimension ranges from 8 to 11 using points with integer coefficients (scenario (d)) (see Table 3).

4.5. Memory consumption

The main disadvantage of `hdch` is the amount of memory consumed, which allows us to compute up to dimension 8 (see Table 1). One can think at this point that an intelligent memory allocation scheme could improve the performance of our algorithms. However, tests with an implementation of `hdch` using the Boehm–DeMers–Weiser conservative garbage collector [57] did not show improvements in computing time. This can be due to the fact that the complexity of the operations performed on the allocated numbers surpasses the complexity of the allocated space. Thus, changing the allocation scheme would not reduce significantly the computation time. This drawback can be seen as the price to pay for the obtained speed-up.

The large memory consumption of our method can be overhauled by exploiting 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 (Section 4). Alternative options are to clean periodically the hash table or to use a Least Recently Used (LRU)

Table 3
Point location experiments. Time of 1 K and 1000 K (1 K = 1000) query points for `hdch_z` and `Triangulation` (`triang`), using distribution (iii) for preprocessing and distribution (i) for queries and integer coefficients. Times averaged over 100 tests for 1 K data.

	d	n	Preproc. time (sec)	Data structs. (MB)	# of cells in triangul.	Query time (sec)	
						1 K	1000 K
`hdch_z`	8	120	45.20	6913	319 438	0.41	392.55
`triang`	8	120	156.55	134	319 438	14.42	14 012.60
`hdch_z`	9	70	45.69	6826	265 874	0.28	276.90
`triang`	9	70	176.62	143	265 874	13.80	13 520.43
`hdch_z`	10	50	43.45	6355	207 190	0.27	217.45
`triang`	10	50	188.68	127	207 190	14.40	14 453.46
`hdch_z`	11	39	38.82	5964	148 846	0.18	189.56
`triang`	11	39	181.35	122	148 846	14.41	14 828.67

cache to avoid storing for long time unused determinants and matrices. For the latter, techniques for efficiently computing determinants of matrices with more than one update, as described by Sankowski [33], could be utilized.

4.6. Filtering

As shown by experiments, one main advantage of the dynamic determinant method shows up when applied to exact geometric constructions. One question that arises, and could be a subject of future work, is whether we can use this method to geometric predicates that benefit from filtering techniques. While in low dimensions filtering provides a very efficient framework for computing signs of determinants, in medium and high dimensions filtering with double-precision floating-point numbers is not efficient, since it reverts too often to exact computations [43]. Recent work in `CGAL`, namely the `Epick_d` kernel, tries to overcome this limitation using hardware and software advances, pushing forward the dimensions on which filtering can be used. Preliminary tests indicate that our implementation, without filtering, is faster than the filtering implemented by Boissonnat et al. [43], but slower than the new implementation in `Epick_d`.

Brönnimann et al. [28] propose another filtering scheme for determinant computations in medium dimensions, using a decomposition method which is numerically more stable than the usual LU decomposition. However, the authors are not aware of any work that evaluates the efficiency of this technique in practice.

5. Concluding remarks

Our paper introduces a method of optimizing the computation of sequences of determinants, using dynamic determinant updates and the well-known Sherman–Morrison formulas. Despite of being well-known this is the first time these formulas are use to geometric algorithms, which make heavy use of similar determinant computations. We demonstrate how this can be done and also present experimental evidences about the supremacy of these methods over state-of-the-art methods in determinant and geometric computations.

A future improvement in the memory consumption of our method could be the exploitation of hybrid memory management techniques as discussed in Section 4. One extension of the proposed method of this paper would be the application of dynamic determinants to the *gift wrapping* (GfR) convex hull algorithms [58,54]. Such an extension would certainly improve the memory consumption of our method.

Overall we hope that the research results presented in this paper will promote the use of these update formulas in other geometric algorithms implementations, and will trigger some further applied-research regarding searching and storing the determinant-adjoint pairs as well as the use of dynamic determinants methods together with filtered computations.

Acknowledgements

We would like to thank Ioannis Z. Emiris for his advice and encouragement, Elias Tsigaridas for bibliographic suggestions, Menelaos Karavelas for discussions on efficient dynamic determinant updates, Olivier Devillers for discussions on `Triangulation` and Monique Teillaud for her careful comments on a previous version of the manuscript. We also thank the anonymous referees for their comments which helped us improve the presentation.

Appendix A. Complexity of Bird's determinant algorithm

So far we choose to implement Bird's algorithm [13] to represent the class of combinatorial determinant algorithms. In the original paper, it is stated that the complexity is bounded by $O(dM(d))$, where $M(d)$ is the cost of multiplicating two matrices.

We choose in this paper to implement the above mentioned algorithm using schoolbook matrix multiplication [46, §3.1]. The given complexity bound still holds, but we compute in this section a tighter bound for our specific case.

The tool we use in the analysis is Faulhaber's formula [59], which gives a form to compute sums of powers. For particular values of the exponent of the summed numbers, this formula turns into

$$P(d)=\sum_{k=1}^{d}k^2=\frac{1}{6}\left(2d^3+3d^2+d\right),\text{ and} \tag{A.1}$$

$$T(d)=\sum_{k=1}^{d}k^3=\frac{1}{4}\left(d^4+2d^3+d^2\right). \tag{A.2}$$

With Equations (A.1) and (A.2), we are ready to develop the formulas to compute the complexity bound. Let us assume that the matrix has size d. We will compute the number of scalar multiplications done by the algorithm, and show then that the number of additions is bounded by the same function.

The algorithm performs, on its first step, a partial multiplication of one upper-triangular matrix and a full matrix. Moreover, only the upper triangular part of this matrix will be used in the next step; thus we consider in the sequel only the computation of the entries which will be used.

Let us consider the rows of the resulting matrix. The first row contains d elements and, to compute each one of them, we need d multiplications. The second row contains $d-1$ non-zero elements and, to compute each one of them, we need to perform only $d-1$ multiplications (since we do not perform one multiplication, because we know the first element of the second row of an upper-triangular matrix is zero). With analogous reasoning for each row, we can conclude that, to compute the $(d-k)$-th row of the product, we need k^2 multiplications. To compute the first matrix, we need thus $P(d)=\sum_{k=1}^{d}k^2$.

Let us consider the second step. Of the matrix computed on the first step, we need all but the last row. In fact, in step s of the algorithm, we need only the first $d-s$ rows. In fact, we will compute only the rows which are needed. This means that, in step s of the algorithm, we will perform $P(d)-P(s)$ scalar multiplications. It follows that the amount of scalar multiplications needed by the algorithm is

$$A(d)=dP(d)-\sum_{j=0}^{d-1}P(j) \tag{A.3}$$

Since we know how to compute the minuend, let us concentrate on the sum of the $P(j)$'s.

$$\begin{aligned}\sum_{j=0}^{d-1}P(j)&=\frac{1}{3}\left(\sum_{j=0}^{d-1}j^3\right)+\frac{1}{2}\left(\sum_{j=0}^{d-1}j^2\right)+\frac{1}{6}\left(\sum_{j=0}^{d-1}j\right)\\&=\frac{1}{3}T(d-1)+\frac{1}{2}P(d-1)+\frac{1}{12}(d-1)d\\&=\frac{1}{12}\Big((d-1)^4+2(d-1)^3+(d-1)^2+2(d-1)^3+3(d-1)^2+(d-1)+(d-1)d\Big)\\&=\frac{1}{12}\Big((d-1)^4+4(d-1)^3+4(d-1)^2+(d-1)(d+1)\Big)\end{aligned} \tag{A.4}$$

Substituting Equation (A.4) in Equation (A.3) we have the following.

$$\begin{aligned}A(d)&=dP(d)-\sum_{j=0}^{d-1}P(j)\\&=\frac{1}{12}\Big(4d^4+O(d^3)\Big)-\frac{1}{12}\Big((d-1)^4+O(d^3)\Big)\\&=\frac{1}{4}d^4+O(d^3)\end{aligned} \tag{A.5}$$

Equation (A.5) bounds the number of scalar multiplications done by Bird's algorithm when using schoolbook matrix multiplication. Let us now bound the number of scalar additions done. Observe that, for multiplicating two matrices, the number of scalar additions is always smaller than the number of scalar multiplications. Beyond those, the algorithm needs to perform at most d scalar additions on each step. This means that the number of scalar additions performed by the algorithm is also bounded by $A(d)$.

References

[1] C.K. Yap, T. Dubé, The exact computation paradigm, in: D.-Z. Du, F. Hwang (Eds.), Computing in Euclidean Geometry, World Scientific, Singapore, 1995, pp. 452–492, Ch. 11.
[2] CGAL, Computational Geometry Algorithms Library, http://www.cgal.org, 2015.
[3] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C.-K. Yap, Classroom examples of robustness problems in geometric computations, Comput. Geom. 40 (1) (2008) 61–78, http://dx.doi.org/10.1016/j.comgeo.2007.06.003.
[4] J.-D. Boissonnat, M. Yvinec, Algorithmic Geometry, Cambridge University Press, New York, NY, USA, 1998.
[5] R. Seidel, A convex hull algorithm optimal for point sets in even dimensions, Master's thesis, Dept. Comp. Sci., Univ. British Columbia, Vancouver, 1981, http://circle.ubc.ca/bitstream/handle/2429/22652/UBC_1981_A6_7S45.pdf.
[6] B. Büeler, A. Enge, K. Fukuda, Exact volume computation for polytopes: a practical study, in: Polytopes: Combinatorics and Computation, in: Oberwolfach Seminars, vol. 29, Birkhäuser, 2000, pp. 131–154.
[7] J. Bunch, J. Hopcroft, Triangular factorization and inversion by fast matrix multiplication, Math. Comput. 28 (125) (1974) 231–236, http://dx.doi.org/10.2307/2005828.
[8] F. Le Gall, Powers of tensors and fast matrix multiplication, in: Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, ISSAC '14, ACM, New York, NY, USA, 2014, pp. 296–303.
[9] E.H. Bareiss, Sylvester's identity and multistep integer-preserving Gaussian elimination, Math. Comput. 22 (1968) 565–578, http://dx.doi.org/10.2307/2004533.
[10] E. Kaltofen, G. Villard, On the complexity of computing determinants, Comput. Complex. 13 (2005) 91–130, http://dx.doi.org/10.1007/s00037-004-0185-3.
[11] M. Mahajan, V. Vinay, A combinatorial algorithm for the determinant, in: Proceedings of the Eighth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '97, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997, pp. 730–738, http://dl.acm.org/citation.cfm?id=314161.314429.
[12] G. Rote, Division-free algorithms for the determinant and the Pfaffian: algebraic and combinatorial approaches, in: H. Alt (Ed.), Computational Discrete Mathematics, in: Lecture Notes in Computer Science, vol. 2122, Springer, Berlin, Heidelberg, 2001, pp. 119–135.
[13] R.S. Bird, A simple division-free algorithm for computing determinants, Inf. Process. Lett. 111 (2011) 1072–1074, http://dx.doi.org/10.1016/j.ipl.2011.08.006.
[14] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, in: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, STOC '87, ACM, New York, NY, USA, 1987, pp. 1–6.
[15] A. Urbańska, Faster combinatorial algorithms for determinant and Pfaffian, Algorithmica 56 (2010) 35–50, http://dx.doi.org/10.1007/s00453-008-9240-9.
[16] C. Krattenthaler, Advanced determinant calculus: a complement, Linear Algebra Appl. 411 (2005) 68, http://dx.doi.org/10.1016/j.laa.2005.06.042.
[17] A. Barvinok, J.E. Pommersheim, An algorithmic theory of lattice points in polyhedra, in: New Perspectives in Algebraic Combinatorics, vol. 38, 1999, pp. 91–147, http://library.msri.org/books/Book38/files/barvinok.pdf.
[18] J. Rambau, TOPCOM: triangulations of point configurations and oriented matroids, in: A. Cohen, X.-S. Gao, N. Takayama (Eds.), Math. Software: ICMS, World Scientific, 2002, pp. 330–340.
[19] D.A. Cox, J.B. Little, D. O'Shea, Using Algebraic Geometry, Graduate Texts in Mathematics, Springer-Verlag, Berlin–Heidelberg–New York, 2005.
[20] S. Basu, R. Pollack, M.-F. Roy, Algorithms in Real Algebraic Geometry, Springer-Verlag, Berlin, 2003.
[21] I.Z. Emiris, T. Kalinka, C. Konaxis, T.L. Ba, Implicitization of curves and (hyper)surfaces using predicted support, in: Symbolic-Numerical Algorithms, Theor. Comput. Sci. 479 (2013) 81–98, http://dx.doi.org/10.1016/j.tcs.2012.10.018.
[22] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B.D. Saunders, W.J. Turner, G. Villard, LinBox: a generic library for exact linear algebra, in: A.M. Cohen, X.-S. Gao, N. Takayama (Eds.), First International Congress of Mathematical Software, ICMS'2002, August 2002, World Scientific, Beijing, China, 2002, pp. 40–50, http://lara.inist.fr/bitstream/handle/2332/793/LIP-RR2002-15.pdf.
[23] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org, 2010.
[24] K.L. Clarkson, Safe and effective determinant evaluation, in: Proc. 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh PA, 1992, pp. 387–395.
[25] H. Brönnimann, I.Z. Emiris, V. Pan, S. Pion, Sign determination in residue number systems, Theor. Comput. Sci. 210 (1) (1999) 173–197, http://dx.doi.org/10.1016/S0304-3975(98)00101-7.
[26] J. Abbott, M. Bronstein, T. Mulders, Fast deterministic computation of determinants of dense matrices, in: Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, ISSAC '99, ACM, New York, NY, USA, 1999, pp. 197–204.
[27] H. Brönnimann, M. Yvinec, Efficient exact evaluation of signs of determinants, Algorithmica 27 (1) (2000) 21–56, http://dx.doi.org/10.1007/s004530010003.
[28] H. Brönnimann, C. Burnikel, S. Pion, Interval arithmetic yields efficient dynamic filters for computational geometry, in: 14th European Workshop on Computational Geometry, Discrete Appl. Math. 109 (1–2) (2001) 25–47, http://dx.doi.org/10.1016/S0166-218X(00)00231-6.
[29] E. Kaltofen, G. Villard, Computing the sign or the value of the determinant of an integer matrix, a complexity survey, J. Comput. Appl. Math. 162 (1) (2004) 133–146, http://dx.doi.org/10.1016/j.cam.2003.08.019.
[30] I. Emiris, V. Fisikopoulos, C. Konaxis, L. Peñaranda, An oracle-based, output-sensitive algorithm for projections of resultant polytopes, Int. J. Comput. Geom. Appl. 23 (4–5) (2013) 397–423, http://dx.doi.org/10.1142/S0218195913600108 (special issue of invited papers from SoCG'12).
[31] J. Sherman, W.J. Morrison, Adjustment of an inverse matrix corresponding to a change in one element of a given matrix, Ann. Math. Stat. 21 (1) (1950) 124–127, http://dx.doi.org/10.1214/aoms/1177729893.
[32] M.S. Bartlett, An inverse matrix adjustment arising in discriminant analysis, Ann. Math. Stat. 22 (1) (1951) 107–111, http://dx.doi.org/10.1214/aoms/1177729698.
[33] P. Sankowski, Dynamic transitive closure via dynamic matrix inverse: extended abstract, in: 45th Annual IEEE Symposium on Foundations of Computer Science, 2004. Proceedings, IEEE, 2004, pp. 509–517.
[34] K. Avrachenkov, N. Litvak, The effect of new links on Google PageRank, Stoch. Models 22 (2) (2006) 319–331, http://dx.doi.org/10.1080/15326340600649052.
[35] V. Fisikopoulos, L. Peñaranda, Faster geometric algorithms via dynamic determinant computation, in: Proceedings of the 20th European Symposium on Algorithms, ESA 2012, in: Lecture Notes in Computer Science, vol. 7501, Springer, Ljubljana, Slovenia, 2012, pp. 443–454.
[36] D.A. Harville, Matrix Algebra from a Statistician's Perspective, Springer-Verlag, New York, 1997.
[37] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer-Verlag, New York, Inc., New York, NY, USA, 1987.
[38] K.L. Clarkson, P.W. Shor, Applications of random sampling in computational geometry, II, Discrete Comput. Geom. 4 (1) (1989) 387–421, http://dx.doi.org/10.1007/BF02187740.
[39] D.J.H. Garling, Inequalities: A Journey into Linear Analysis, Cambridge University Press, 2007, Cambridge Books Online.
[40] G.M. Ziegler, Lectures on Polytopes, Springer, 1995.
[41] Boost: peer reviewed C++ libraries, http://www.boost.org, 2015.

[42] S. Hornus, O. Devillers, C. Jamin, dD triangulations, in: CGAL User and Reference Manual, 4.6.1 edition, 2015, CGAL Editorial Board http://doc.cgal.org/4.6.1/Manual/packages.html#PkgTriangulationsSummary.

[43] J.-D. Boissonnat, O. Devillers, S. Hornus, Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension, in: SoCG, ACM, 009, pp. 208–216, http://dx.doi.org/10.1145/1542362.1542403.

[44] G. Villard, A study of Coppersmith's block Wiedemann algorithm using matrix polynomials, IMAG Research Report 975-I-M. Apr. 1997, http://perso.ens-lyon.fr/gilles.villard/BIBLIOGRAPHIE/PDF/rr0497.pdf.

[45] J.-G. Dumas, A. Urbańska, An introspective algorithm for the integer determinant, in: J.-G. Dumas (Ed.), Transgressive Computing 2006, Copias CoCa, Madrid, Granada, Spain, 2006, pp. 185–202, http://arxiv.org/abs/cs/0511066.

[46] D. Poole, Linear Algebra: A Modern Introduction, Cengage Learning, 2006.

[47] S.J. Berkowitz, On computing the determinant in small parallel time using a small number of processors, Inf. Process. Lett. 18 (3) (1984) 147–150, http://dx.doi.org/10.1016/0020-0190(84)90018-8.

[48] C.S. Iliopoulos, Worst-case complexity bounds on algorithms for computing the canonical structure of finite Abelian groups and the Hermite and Smith normal forms of an integer matrix, SIAM J. Comput. 18 (4) (1989) 658–659, http://dx.doi.org/10.1137/0218045.

[49] S. Robinson, Toward an optimal algorithm for matrix multiplication, SIAM News 38 (9) (2005), http://www.siam.org/news/news.php?id=174.

[50] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd edition, Cambridge University Press, 2007.

[51] K.L. Clarkson, K. Mehlhorn, R. Seidel, Four results on randomized incremental constructions, Comput. Geom. 3 (4) (1993) 185–212, http://dx.doi.org/10.1016/0925-7721(93)90009-U.

[52] E. Gawrilow, M. Joswig, polymake: a framework for analyzing convex polytopes, in: G. Kalai, G. Ziegler (Eds.), Polytopes – Combinatorics and Computation, Birkhäuser, 2000, pp. 43–74.

[53] K. Fukuda, cddlib, version 0.94f, http://www.ifor.math.ethz.ch/~fukuda/cdd_home, 2008.

[54] D. Avis, Lrs: a revised implementation of the reverse search vertex enumeration algorithm, in: Polytopes – Combinatorics and Computation, in: Oberwolfach Seminars, vol. 29, Birkhäuser-Verlag, 2000, pp. 177–198.

[55] B. Büeler, A. Enge, VINCI, http://www.math.u-bordeaux1.fr/~aenge/index.php?category=software&page=vinci.

[56] I. Gelfand, M. Kapranov, A. Zelevinsky, Discriminants, Resultants and Multidimensional Determinants, Birkhäuser, Boston, 1994.

[57] H.-J. Boehm, Space efficient conservative garbage collection, in: Programming Language Design and Implementation, ACM, 1993, pp. 197–206.

[58] D.R. Chand, S.S. Kapur, An algorithm for convex polytopes, J. ACM 17 (1) (1970) 78–86, http://dx.doi.org/10.1145/321556.321564.

[59] J. Conway, The Book of Numbers, Copernicus, New York, NY, 1996.