Efficient Exact Geometric Predicates for Delaunay Triangulations *

Olivier Devillers† Sylvain Pion‡

Abstract

A time efficient implementation of the exact geometric computation paradigm relies on arithmetic filters which are used to speed up the exact computation of easy instances of the geometric predicates. Depending of what is called "easy instances", we usually classify filters as static or dynamic and also some in between categories often called semi-static.

In this paper, we propose, in the context of three dimensional Delaunay triangulations:
— automatic tools for the writing of static and semi-static filters,
— a new semi-static level of filtering called *translation filter,*
— detailed benchmarks of the success rates of these filters and comparison with rounded arithmetic, long integer arithmetic and filters provided in Shewchuk's predicates [25].

Our method is general and can be applied to all geometric predicates on points that can be expressed as signs of polynomial expressions. This work is applied in the CGAL library [10].

1 Introduction

A geometric algorithm usually takes decisions based on some basic geometric questions called *predicates*. Numerical inaccuracy in the evaluation of geometric predicates is one of the main obstacles in implementing geometric algorithms robustly. Among the solutions proposed to solve this problem, the exact geometric computation paradigm is now recognized as an effective solution [27]. Computing the predicates exactly makes an algorithm robust, but also very slow if this is done by using some expensive exact arithmetic. The current way to speed up the algorithms is to use some rounded evaluation with certified error to answer safely and quickly the easy cases, and to use some expensive exact arithmetic only in nearly degenerate situations. This approach, called *arithmetic filtering*, gives very good results in practice [17, 7, 8].

Although what we propose is quite general, we focus now on the particular case which has been used to validate our ideas: the predicates for the three dimensional Delaunay triangulations. This work is implemented in the CGAL library [10]. Many surface reconstruction algorithms [5, 1, 2] are based on Delaunay triangulations and we will take our point sets for benchmarking from that context. Predicates evaluation can take from 40% to almost 100% of the running time depending of the kind of filters used, thus it is critical to optimize them.

The predicates used in Delaunay algorithms are the `orientation` predicate which decides the orientation of four points and the `in_sphere` predicate which decides among five points if the fifth is inside the sphere passing through the four others. Like many other geometric predicates, those reduce to the evaluation of the sign of some polynomial $P(x)$. A filter computes a rounded value for $P(x)$, and a certified bound for the rounding error. The filter is called *static* if the error is computed off-line based on hypotheses on the data, *dynamic* if the error is computed at run time step by step in the evaluation of $P(x)$ and *semi-static* if the error is computed at run-time by a simpler computation.

In this paper, we propose,
— to compute the off-line error in static and semi-static filter by an automatic analysis of the generic code of the predicate,
— to use an *almost static filter* where the error bound is updated when the new data does not fit any longer the hypotheses,
— a new semi-static level of filtering: the *translation filter* which starts by translating the data before the semi-static error computation and
— detailed benchmarks on synthetic and real data providing evidence of the efficiency of our approach.

The efficiency of static filters was proved by Devillers and Preparata [14] but their analysis was based on probabilistic hypotheses which does not usually apply to real data. Automatic code generation for exact predicates has been proposed [8, 23], but it was limited to dynamic filters [7] or static filters with strong hypotheses on the data [17]. Some existing techniques

*Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces).

†`Olivier.Devillers@sophia.inria.fr` INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis cedex, France.

‡`Sylvain.Pion@mpi-sb.mpg.de` Max Planck Institut für Informatik, Saarbrücken, Germany.

using static filtering need hypotheses on the input coordinates, like a limited bit length [4], or requiring to have fixed point values which may require truncation as a preprocessing step [10, 17].

Finally, we also compare running times with the simple floating point code (which is not robust), with a naive implementation of multi-precision arithmetic, and with the well known robust implementation of these predicates by Jonathan Shewchuk [25].

2 Our case study

2.1 Algorithm Our purpose is to study the behavior of the predicates in the practical context of a real application, even if our results can be used for other algorithms, we briefly present the one used in this paper.

The algorithm used for the experiments is the Delaunay hierarchy [11], which uses few levels of Delaunay triangulations of random samples. The triangulation is updated incrementally inserting the points in a random order, when a new point is added, it is located using walking strategies [13] across the different levels of the hierarchy, and then the triangulation is updated. The location step uses the `orientation` predicate while the update step relies on the `in_sphere` predicate.

The randomized complexity of this algorithm is related to the expected size of the triangulation of a sample, that is quadratic in the worst case, but subquadratic with some realistic hypotheses [3, 16, 15], practical inputs often give a linear output [19] and an $O(n \log n)$ algorithmic complexity.

The implementation is the one provided in CGAL [6, 26]. The design of CGAL allows to switch the predicates used by the algorithm and thus makes the experiments easy [21].

2.2 Predicates The two well-known predicates needed for Delaunay triangulation are:

— the `orientation` test, which tests the position of a point relative to the oriented plane defined by three other points, when they are not coplanar.

— the `in_sphere` test, which, given four positively oriented points, decides whether a fifth point lies inside the circumscribing sphere of the four points, or not.

In this paper, we do not focus on degenerate cases. Dealing with 4 coplanar or 5 cospherical points can be handled in the algorithm or by standard perturbation schemes. This can possibly involve other predicates (e.g. in-circle test for coplanar points) but these degenerate evaluations are rare enough to be neglected in the whole computation time.

The `orientation` predicate of the four points p, q, r, s boils down, when using Cartesian coordinates, to the sign of the following 4x4 determinant, which can be simplified to a 3x3 determinant and an initial set of subtractions:

$$\begin{vmatrix} p_x & p_y & p_z & 1 \\ q_x & q_y & q_z & 1 \\ r_x & r_y & r_z & 1 \\ s_x & s_y & s_z & 1 \end{vmatrix} = \begin{vmatrix} p'_x & p'_y & p'_z \\ q'_x & q'_y & q'_z \\ r'_x & r'_y & r'_z \end{vmatrix}$$

with $p'_x = p_x - s_x$ and so on for the y and z coordinates and the points q and r.

We use the C++ template mechanism in order to implement the `orientation` predicate generically, using only the algebraic formula above. This template can be used with any type `T` which provides functions for the subtraction, addition, multiplication and comparison. We will see how to use this code in different ways later.

```
template <class T>
int
orientation(T px, T py, T pz, T qx, T qy, T qz,
            T rx, T ry, T rz, T sx, T sy, T sz)
{
  T psx=px-sx, psy=py-sy, psz=pz-sz;
  T qsx=qx-sx, qsy=qy-sy, qsz=qz-sz;
  T rsx=rx-sx, rsy=ry-sy, rsz=rz-sz;

  T m1 = psx*qsy - psy*qsx;
  T m2 = psx*rsy - psy*rsx;
  T m3 = qsx*rsy - qsy*rsx;
  T det = m1*rsz - m2*qsz + m3*psz;

  if (det>0) return 1;
  if (det<0) return -1;
  return 0;
}
```

Similarly, the `in_sphere` predicate of 5 points t, p, q, r, s is the sign of a 5x5 determinant, which can be simplified to the sign of the following 4x4 determinant:

$$\begin{vmatrix} t'_x & t'_y & t'_z & {t'_x}^2 + {t'_y}^2 + {t'_z}^2 \\ p'_x & p'_y & p'_z & {p'_x}^2 + {p'_y}^2 + {p'_z}^2 \\ q'_x & q'_y & q'_z & {q'_x}^2 + {q'_y}^2 + {q'_z}^2 \\ r'_x & r'_y & r'_z & {r'_x}^2 + {r'_y}^2 + {r'_z}^2 \end{vmatrix}$$

We implement it similarly using a template function, the 4x4 determinant being computed using the dynamic programming method (i.e. first compute the minors of rank 2, then use them to compute the minors of rank 3, then use them in turn to compute the determinant).

2.3 Data sets For the experiments, we have used the following data sets (see Figures 1, 2, 3):

— (R5) — 500,000 random points uniformly distributed in a cube (the coordinates have been generated by the

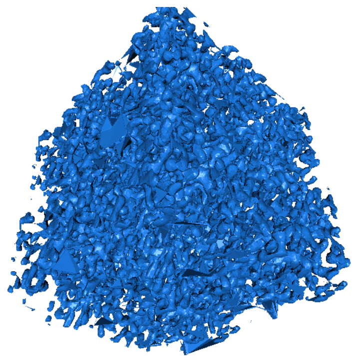

Figure 1: Points on the surface of a molecule (M).

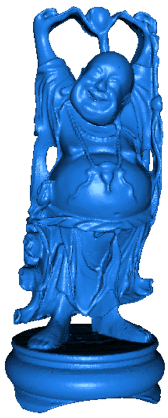

Figure 2: Points on the surface of a Buddha statue (B).

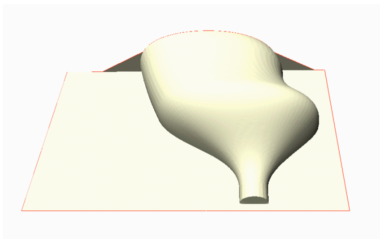

Figure 3: Points on the surface of a dryer handle (D).

drand48 C function).
— (R20) — 2,000,000 random points uniformly distributed in a cube.
— (E) — 500,000 random points almost uniformly distributed on the surface of an ellipsoid.
— (M) — 525,296 points on the surface of a molecule.
— (B) — 542,548 points on the surface of a Buddha statue (data from Stanford scanning repository).
— (D) — 49,787 points on the surface of a dryer handle (data provided by Dassault Systèmes). The way the scanning was done has produced a lot of coplanar points which exercises the robustness a lot.

Experiments have all been performed on a Pentium III PC at 1 GHz, with 1 GB of memory, the compiler used is GCC 2.95.3. We have gathered some general data on the computation of the 3D Delaunay triangulations of these sets of points in Table 1.

3 Simple floating point computation

The naive method consists of using floating point arithmetic in order to evaluate the predicates. Practically, this means using the C++ built-in type `double` as `T` in the generic predicates described above. This does not give a guaranteed result due to roundoff errors, but is the most efficient method when it works.

The triangulation algorithm happened to crash only on data set (D). Also note that even if it doesn't crash, the result may not be exactly the Delaunay triangulation of the points, so some mathematical properties may not be fulfilled, and this may have bad consequences on the later use of the triangulation.

We first measured the number of times the predicates gave a wrong result when computed with floating point, by comparing its result with some exact computation (see Table 2), with the algorithm using the exact result to continue its path.

The `orientation` predicate is used for walking in the triangulation during the point location. Depending on the walking strategy [13] and the particular situation, `orientation` failures often have no consequences but it may cause a loop or return a wrong tetrahedron as a result, which could result in an incorrect Delaunay triangulation.

The reason why the `in_sphere` predicate fails more often is due to its larger algebraic complexity, which induces larger roundoff errors. The failure of the `in_sphere` predicate will definitely create a non-Delaunay triangulation after the insertion of the point.

We also note that the random distribution does not incur any failure, it is due to a better conditioning of the computed determinants. This fact questions the relevance of theoretical studies based on random distributions.

	R5	R20	E	M	B	D
# points	500,000	2,000,000	500,000	525,296	542,548	49,787
# tetrahedra	3,371,760	13,504,330	3,241,972	3,588,527	3,864,194	321,148
# `orientation` calls	39,701,472	166,623,287	56,340,962	44,280,741	55,201,542	3,810,257
# `in_sphere` calls	22,181,509	89,033,404	13,945,582	23,697,290	25,073,049	1,924,558
MB of memory	153	574	148	161	172	25

Table 1: Informations on the computation of the Delaunay triangulations of the data sets.

	R5	R20	E	M	B	D
% of wrong results for `orientation`	0%	0%	0%	0.0001%	0.002%	0.17%
% of wrong results for `in_sphere`	0%	0%	0%	0.005%	0.0005%	0.8%

Table 2: Wrong results given by the floating point evaluation of the predicates.

Running times for the computations can be found in Table 6, they provide a lower limit, and the goal is to get as close as possible from this limit with exact methods. The percentage of time spent in the predicates can be roughly evaluated to 40%.

4 Naive exact multi-precision arithmetic

The easiest solution to evaluate exactly the predicates is to use an exact number type. Given that, in order to evaluate exactly the sign of a polynomial, it is enough to use multi-precision floating point arithmetic, which guarantees exact additions, subtractions and multiplications. These number types are provided by several libraries such as GMP [20]. CGAL also provides such a data type on its own (via the `MP_Float` class), which is efficient enough for numbers of reasonable bit length which is the case in our predicates. Again, it is enough to pass the `MP_Float` type as the type `T` of the template functions described above.

Here we notice that the naive exact method is approximately 70 times slower (see Table 6) compared to floating point, which makes it hardly usable for most applications, at least if we use it naively. However, exact evaluation is acceptable as the last stage of a filtered evaluation of the predicate, where performance doesn't matter so much. GMP does not give better result in that context, the Delaunay triangulation of 500,000 random points with integer coordinates on 31 bits needs about 2000 seconds, which is of the same order as the 3000 seconds we get with `MP_Float` and very far from the running time using floating point arithmetic.

5 General dynamic filter based on interval arithmetic

As we will show, interval arithmetic [7, 24] allows us to achieve an already quite important improvement over the previous naive exact method. We have used the implementation of this method provided by CGAL through the `Filtered_exact` functionality. It is still a very general answer to the filtering approach. When the interval arithmetic is not precise enough, we rely on `MP_Float` in order to decide the exact result. We can reuse the template version of the predicate over both the interval arithmetic number type, as well as `MP_Float`, and we use the C++ exception mechanism to notify filter failures : the comparison operators of the interval arithmetic class raise an exception in case of overlapping intervals. So we basically use the following code, which is independent of the particular content of the algebraic formula of the predicate:

```
int dynamic_filter_orientation(
              double px, double py, double pz,
              double qx, double qy, double qz,
              double rx, double ry, double rz,
              double sx, double sy, double sz)
{
  try
  {
    return orientation<Interval_nt>
                  (px, py, pz, qx, qy, qz,
                   rx, ry, rz, sx, sy, sz);
  }
  catch (...)
  {
    return orientation<MP_Float>
                  (px, py, pz, qx, qy, qz,
                   rx, ry, rz, sx, sy, sz);
  }
}
```

Table 3 shows the number of times the interval arithmetic is not able to decide the correct result for a predicate, and thus needs to call a more precise version. For the randomly generated data sets, there is not a single filter failure. For the other data sets, if we compare these numbers to the number of wrong results given by floating point (Table 2), we can see that this filter doesn't require an expensive evaluation too often, about three times the real failures of the floating point evaluation.

Moreover, if we compare these numbers to the total number of calls, we obtain that the filter fails with a probability of 1/200,000 for the `orientation` predicate, and 1/13,000 for `in_sphere`, for the M data set, which is a high success rate. For the D data set, we obtain 1/254 and 1/77. Given that the exact computation with `MP_Float` is 70 times slower than floating point, its global cost at run time is negligible for the M data set, since its called rarely, and what dominates is therefore the evaluation using interval arithmetic.

Table 6 shows that the running time overhead compared to floating point is now approximately on the order of 3.4. This is far better than the previous naive exact multi-precision method, but still quite some overhead that we might want to remove.

6 Static filter variants

The previous method gives a very low failure rate, which is good, but for the common case it is still more than 3 times slower compared to the pure floating point evaluation. The situation can be improved by using static filter techniques [17, 8]. This kind of filter may be used before the interval computation, but it usually needs hypotheses on the data such as a global upper bound on the coordinates.

Given that we need to evaluate the sign of a known polynomial, if we know some bound b on the input coordinates, then we can derive a bound $\epsilon(b)$ on the total round-off error that the floating point evaluation of this polynomial value will introduce at worst. At the end, if the value computed using floating point has a greater absolute value than $\epsilon(b)$, then one can decide exactly the sign of the result. We explain in the next section how to compute $\epsilon(b)$, but let us just give here the result for the `orientation` predicate: $\epsilon(b) = 3.908 \times 10^{-14} \times b^3$.

We can apply this remark in two different ways.

First, if we know a unique global bound b on *all* the input point coordinates, then we need to compute $\epsilon(b)$ only once for the whole algorithm, and $\epsilon(b)$ can be considered a constant. This is traditionally called a static filter.

Sometimes it is not possible to know a global bound, at compile time or even at run time, or it is simply inconvenient to find one for some dynamic algorithms. In that situation, we introduce the *almost static filter* in which the global bound b on the data is updated for each point added to the triangulation, and $\epsilon(b)$ is updated when b changes. This is relatively cheap, and completely amortized since inserting a point in a triangulation costs hundreds of calls to predicates (see Table 1). Notice also, that this approach needs to get out from the classical filtering scheme where only the code of the predicates is modified ; we need here to change also the point constructor to be able to maintain b and $\epsilon(b)$.

Since b is growing along with the inserted points, b may be largely over evaluated for some specific instance of the predicate. Thus if the almost static filter fails, we use a second stage which computes a bound b' from the actual arguments of the predicate, at each call, and computes $\epsilon(b')$ from it. This one is usually called a semi-static filter.

Table 6 shows that these methods behave far better than the interval arithmetic filter alone from the running time point of view on all data sets. We now get within 10% to 70% more time compared to the floating point version.

On the other hand, Table 4 shows that these filters fail much more often than interval arithmetic: the static filter fails between 6% and 75% of the time for `in_sphere`, so we'd better keep all stages to achieve best overall running time. Here we also notice an important difference between `orientation` and `in_sphere`, the later failing much more often, even on the random data set. We explain this behavior by the fact that the points over which the predicates are called are closer to each other on average in the `in_sphere` case, due to the way the triangulation algorithm works.

Automatic error bound computation Evaluating an error bound $\epsilon(b)$ is tedious to do by hand for each different predicate, but it is easy to compute automatically, given the polynomial expression by its template code, by following the IEEE 754 standard rules on floating point computations.

First, we remark that the polynomials we manipulate are homogeneous of some degree d, it means we can compute an error bound of the form: $\epsilon(b) = \epsilon(1) * b^d$ where $\epsilon(1)$ is a constant, as it depends only on the way the polynomial is computed. Computing $\epsilon(b)$ from $\epsilon(1)$ is therefore quite cheap once b is known, and b is not very expensive to compute either since it is the maximum of the absolute values of the input coordinates.

In order to compute $\epsilon(1)$, we wrote a class that allows to compute it easily for any polynomial expression, and since it has the interface of a number type, we can directly pass it through the code of the template predicates, via the C++ template mechanism, again. It automatically propagates the error bounds through each addition, subtraction and multiplication, and when a

	R5	R20	E	M	B	D
% of failures for `orientation`	0%	0%	0%	0.0005%	0.005%	0.4%
% of failures for `in_sphere`	0%	0%	0%	0.007%	0.001%	1.35%

Table 3: Number of filter failures for interval arithmetic.

	R5	R20	E	M	B	D
`orientation`						
almost static filter failures	0%	0.000003%	0.001%	0.015%	0.1%	7.3%
semi static filter failures	0%	0%	0.0002%	0.01%	0.1%	7.3%
`in_sphere`						
almost static filter failures	7%	37%	39%	16%	74%	17%
semi static filter failures	0.6%	4.6%	20%	4%	37%	7.5%

Table 4: Static and semi static filter failures.

comparison is attempted, it just stops and prints the error bound $\epsilon(1)$. More precisely, we instantiate the predicate with a special number type used only for error bounds computation, this number type has two fields: the upper bound field `x.m` and the error field `x.e`. For this number type, the default constructor creates values with bound 1 and error 0, the addition and multiplication are overloaded such that

```
(x+y).m = x.m + y.m
(x+y).e = x.e + y.e + ulp((x+y).m)/2
(x*y).m = x.m * y.m
(x*y).e = x.e*y.m + y.e*x.m + ulp((x*y).m)/2
```

and the comparison operator is overloaded to print the current bound on the error (`ulp` is the "unit in the last place" function which gives the value of the smallest bit of the mantissa of a floating point number).

7 Translation filter

We now make the following observation: the predicate code begins by translating the input points so that point s goes to the origin. In the new frame, the algebraic expression is simplified, which makes the static error bound slightly better. But the most important is that the algorithm often calls the predicates with points which are close to each other and thus have small coordinates in the new frame. This makes a semi-static filter, for this new simplified predicate, very efficient since the bound b' is often small. Translating the data before applying the simplified predicate, without getting out the exact geometric computation paradigm, needs that the translation is done in an exact way. Fortunately, this is often the case for the same reason as previously: if the points are closer to the new origin than to the previous one the translation is done exactly by the floating point arithmetic.

This comes from the following property of floating point arithmetic : given two floating point numbers `a` and `b`, if they differ by at most a factor of two, i.e. $\texttt{a}/2 \leq \texttt{b} \leq 2\texttt{a}$ when they are positive and similarly when they are negative, then their subtraction is computed exactly.

Testing that all the initial subtractions have not created any roundoff errors must be done in a first step. We can quickly determine whether the subtraction `c = a-b` was exact by testing that the two following equalities are true: `a == b+c` and `b == a-c`. This can be shown with the IEEE 754 properties.

Table 5 shows that, among the semi static filter failures, only a low percentage had inexact subtractions, for the `in_sphere` predicate. The new smaller error bound allowed to conclude in all cases for (R5), (R20) and (E), almost all cases for (M) and (B) and 90% of the cases for the difficult point set (D). What this shows is that this filtering stage is also very effective.

8 Conclusion

Table 6 gives the running time of all the different combinations of filters we have tried and shows their efficiency by a comparison with Shewchuk's predicates [25]. We have also considered general approaches which do not involve changing the code of the predicates at all : they still use some kind of dynamic filtering, but they pay a price for dynamic memory allocation for almost all arithmetic operation (since they store expression DAGs). These approaches are the `Expr` class of CORE [22] (a beta version of 1.6), the `real` class [9] of LEDA (version 4.2), and the `Lazy_exact_nt<MP_Float>` class of CGAL.

In this paper we gave a detailed analysis of the efficiency of various filter techniques to compute geometric

	R5	R20	E	M	B	D
semi static filter failures	0.6%	4.6%	20%	4%	37%	7.5%
inexact diffs among semi static failures	0.06%	0.04%	21%	0.3%	2%	7%
translation filter failures	0%	0%	0%	0.007%	0.003%	0.8%

Table 5: Statistics about the efficiency of translation filter stage for the `in_sphere` predicate.

	R5	R20	E	M	B	D
`double`	40.6	176.5	41.0	43.7	50.3	`loops`
`MP_Float`	3,063	12,524	2,777	3,195	3,472	214
Interval + `MP_Float`	137.2	574.1	133.6	144.6	165.1	15.8
semi static + Interval + `MP_Float`	51.8	233.9	61.0	59.1	93.1	8.9
almost static + semi static + Interval + `MP_Float`	44.4	210.1	55.0	52.0	87.2	8.0
almost static + translation + Interval + `MP_Float`	43.6	198.8	52.2	48.8	64.3	7.5
almost static + semi static + translation + Interval + `MP_Float`	43.8	195.8	48.6	48.7	63.6	7.7
Shewchuk's predicates	57.9	249.2	57.5	62.8	71.7	7.2
CORE Expr	570	2431	3520	1355	9600	173
LEDA real	682	2784	640	742	850	125
`Lazy_exact_nt<MP_Float>`	705	2750	631	726	820	67

Table 6: Timings in seconds for the different computation methods of the triangulations.

predicates on points. We have introduced an almost static filter which reaches the efficiency of a static filter without the drawback of imposing hypotheses on the data. We also identified a new filtering step taking into account the initial translation which is usually performed in the predicates in order to simplify their algebraic expression. Our benchmarks show that our scheme is effective and compares favorably to other previous methods for the case of 3D Delaunay triangulations.

We actually simplified the text of the paper by mentioning only a unique bound b of the static filter variants, but we indeed experimented with per-coordinate bounds: b_x, b_y, b_z, and this gave slightly nicer results, especially in the case of data sets with points which can sometime be in a plane parallel to a coordinate plane, as this gives the semi static filter a smaller error bound.

In the future, we plan to make these predicates directly available in CGAL, as well as applying these methods to more predicates such as those needed to compute 2D and 3D regular triangulations. Some work has already been done for predicates on circle arcs [12].

We also propose automatic tools based on C++ template technique to compute the error bounds involved in all the proposed filters directly from the generic code of the predicate; this avoids painful code duplication and error bound computation. We have preferred this use of computer aided predicate design to a complete code generation tool [18, 8] for its simplicity.

Acknowledgments The authors would like to thank the developpers of CGAL who made the 3D triangulation code available, as well as Monique Teillaud for helpful discussions.

References

[1] N. Amenta, M. Bern, and M. Kamvysselis. A new Voronoi-based surface reconstruction algorithm. In *Proc. SIGGRAPH '98*, Computer Graphics Proceedings, Annual Conference Series, pages 415–412, July 1998.

[2] N. Amenta, S. Choi, T. K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 213–222, 2000.

[3] D. Attali and J.-D. Boissonnat. Complexity of the Delaunay triangulation of points on polyhedral surfaces. In *Proc. 7th ACM Symposium on Solid Modeling and Applications*, 2002.

[4] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C16–C17, 1995.

[5] J.-D. Boissonnat and F. Cazals. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 223–232, 2000.

[6] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[7] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[8] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *Internat. J. Comput. Geom. Appl.*, 11:245–266, 2001.

[9] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[10] *The CGAL Manual*, 2002. Release 2.4.

[11] O. Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[12] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 139–147, 2000.

[13] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 106–114, 2001.

[14] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.

[15] R. Dwyer. On the convex hull of random points in a polytope. *J. Appl. Probab.*, 25(4):688–699, 1988.

[16] J. Erickson. Nice point sets can have nasty Delaunay triangulations. In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 96–105, 2001.

[17] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[18] S. Fortune and C. V. Wyk. *LN User Manual.* AT&T Bell Laboratories, 1993.

[19] M. J. Golin and H.-S. Na. On the average complexity of 3D-Voronoi diagrams of random points on convex polytopes. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 127–135, 2000.

[20] T. Granlund. GMP, the GNU multiple precision arithmetic library. `http://www.swox.com/gmp/`.

[21] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes Comput. Sci.*, pages 79–90. Springer-Verlag, 2001.

[22] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *15th ACM Symp. on Computational Geometry*, 1999.

[23] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. In *International Conference on Functional Programming*, Florence, Italy, 2001. Also Carnegie Mellon CS Tech Report CMU-CS-01-141.

[24] S. Pion. *De la géométrie algorithmique au calcul géométrique.* Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 1999. TU-0619.

[25] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[26] M. Teillaud. Three dimensional triangulations in CGAL. In *Abstracts 15th European Workshop Comput. Geom.*, pages 175–178. INRIA Sophia-Antipolis, 1999.

[27] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.