Efficient Exact Geometric Predicates for Delaunay Triangulations

Olivier Devillers* Sylvain Pion*

Abstract

We propose filtering schemes to make efficient geometric predicates. We analyze the run-time behaviour of various approaches to the robustness problems in the frame work of the 3D Delaunay triangulation of CGAL.

The reader interested in more details is encouraged to read the full version of the paper [7].

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 computation paradigm is now recognized as an effective solution. 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 [9, 4, 5].

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. Many surface reconstruction algorithms [3, 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. As many other geometric predicates, those reduce to the evaluation of the sign of some polynomial $P(x)$. A filter computes a rounded value and a certified 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.

We analyze the relevance of various types of filter in practice. 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 [12].

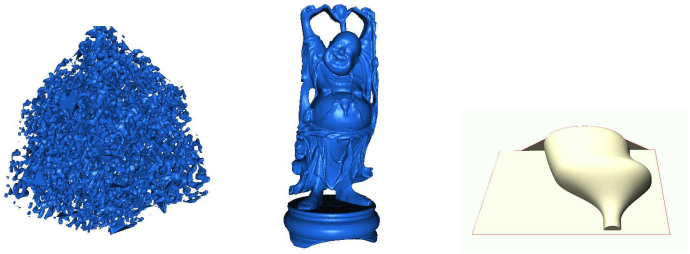
2 Our case study

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 [6], 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 [8] 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 linear in practice, and an $O(n \log n)$ algorithmic complexity. The implementation is the one provided in CGAL [13]. The design of CGAL allows to switch the predicates used by the algorithm and thus makes the experiments easy [10].

*{Olivier.Devillers,Sylvain.Pion}@sophia.inria.fr INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis cedex, France.



(M) (B) (D)

Figure 1: Realistic data sets.

2.2 Predicates

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. The Cartesian kernel of CGAL uses the C++ template mechanism in order to implement the `orientation` predicate generically, using only the algebraic formula. This allows to run this polynomial formula over 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;
  return sign(det);
}
```

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 a 4x4 determinant, which CGAL computes using the dynamic programming method.

2.3 Data sets

For the experiments, we have used the following data sets (see Figure 1):
— (R5,R20) — 500,000 (resp. 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 (from Stanford repository).
— (D) — 49,787 points on the surface of a dryer handle (data provided by Dassault Systèmes). This data set contains a lot of coplanar points which exercises the robustness a lot.

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

	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.

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.

Depending on the walking strategy [8] and the particular situation, `orientation` failures often have no consequences but it may cause a loop or return a wrong tetrahedron as a result. 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 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). Running times for the computations can be found in Table 3, 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. 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.

Here we notice that the naive exact method gives disastrous running times (see Table 3), approximately a factor of 70 compared to floating point, which makes them useless for most applications, at least if we use them naively, but we will now see that they are useful as the last exact stage of a filtered evaluation of the predicate, where performance doesn't matter so much.

5 General dynamic filter based on interval arithmetic

As we will show, interval arithmetic [4, 11] 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. 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, ... sz)
{
  try { return orientation<Interval_nt> (px, py, ..., sz); }
  catch (...) { return orientation<MP_Float> (px, py, ..., sz); }
}
```

Table 2 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.

Table 3 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 we can do better.

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 technics [9, 5]. 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 full version of the paper 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 the code is modified only within the predicates; we need here to overload 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 (semi-static filter). Table 3 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%-70% slower compared to floating point.

On the other hand, Table 2 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.

`orientation`	R5	R20	E	M	B	D
almost static filter failures	0	5	759	6,565	54,187	278,039
semi static filter failures	0	0	97	6,548	53,064	278,039
interval arith. filter failures	0	0	0	207	3,012	15,307
wrong results with f.p.	0	0	0	50	1,114	6,777
`in_sphere`						
almost static filter failures	1,555,568	33,012,786	5,467,029	3,806,699	18,501,189	332,859
semi static filter failures	136,911	4,113,613	2,778,438	955,889	9,192,703	144,192
interval arith. filter failures	0	0	0	1,759	294	25,889
wrong results with f.p.	0	0	0	1,100	128	16,169

Table 2: Various filter failures.

	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
Shewchuk's predicates	57.9	249.2	57.5	62.8	71.7	7.2

Table 3: Timings in seconds for the different computation methods of the triangulations.

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. We have implemented a mecanism to help computing these bounds from the template code of a predicate. The reader can refer to the full version of the paper for the details.

7 Conclusion

In this paper we gave a detailed analysis of the efficiency of various filter technics to compute geometric 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 propose automatic tools based on C++ template technic 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.

Acknowledgments

The authors would like to thank the developpers of CGAL who made the kernel and 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] Jean-Daniel Boissonnat and Frédéric Cazals. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 223–232, 2000.

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

[5] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 175–183, 1998.

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

[7] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. Rapport de recherche 4351, INRIA, 2002.

[8] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 106–114, 2001.

[9] 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.

[10] Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Michael 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.

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

[12] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

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