A Performance Comparison of Interval Arithmetic and Error Analysis in Geometric Predicates

S. A. Seshia G. E. Blelloch R. W. Harper

December 2000
CMU-CS-00-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The first author is supported in part by a National Defense Science and Engineering Graduate Fellowship. This research was supported in part by NSF Grant CCR-9706572.

Abstract

Exact arithmetic is used to build robust implementations of geometric algorithms. However, it is slow, and computing to arbitrary precision is unnecessary most of the time. Floating-point filters, which are commonly used instead, are fast self-checking computations that fall back on exact arithmetic when the check indicates that the fast calculation is incorrect. The use of *interval arithmetic* in floating-point filters is attractive because they can be used to build geometric software that does not assume error-free inputs. However, the use of interval arithmetic might impose a penalty on performance.

In this report, we study the performance impact of using interval arithmetic based filters in the line-side and in-circle geometric predicates. We report results obtained with implementations of two commonly used geometric algorithms: Delaunay triangulation and convex hull computation, and for a range of point distributions. Our results indicate that interval arithmetic imposes a performance penalty of at most 2 in the worst case, and even improves performance in some cases.

Keywords: Computational geometry, geometric predicates, geometric robustness, interval arithmetic, error analysis.

1 Introduction

Many geometric algorithms rely on boolean-valued tests (predicates) on geometric objects. Examples of such predicates include the *lineside* test, which tests which side of a line a point lies on, and the *incircle* test, which determines whether a point lies within a circle. These predicates generally assume that the objects they operate upon are represented in terms of real numbers. However, the implementations of these algorithms on finite-precision machines are susceptible to failure arising from roundoff errors. Arbitrary precision floating-point arithmetic, also known as *exact arithmetic* [14], is therefore used to make geometric algorithms robust to such errors. However, exact arithmetic is slow, and it is preferable to use it only when necessary.

A *floating-point filter* is a fast self-checking calculation that computes an approximate value of an arithmetic expression, resorting to exact arithmetic only when its check indicates that the answer cannot be trusted (i.e., when it "fails"). For geometric predicates such as the lineside and incircle tests, the arithmetic expression to be evaluated is usually the sign of a determinant. A floating-point filter for geometric predicates, hereafter referred to simply as *filter*, can check its accuracy (i.e., whether the sign of its result can be trusted) by testing whether the magnitude of its result is greater than the error it might incur.

Filters can be classified into three types: *static*, *semi-static* and *dynamic*. Static filters are those in which an error bound B is computed prior to compile-time. Let V be the value of the arithmetic expression of interest obtained by performing the fast calculation. If $|V| < B$, then the filter fails, i.e., one must resort to exact arithmetic. Otherwise, the value V is correct and can be used. An example of the use of static filters is in the LN system [5]. In semi-static filters, the computation of the error bound is delayed until run-time. The user supplies a formula at compile-time that evaluates to the error bound at run-time. This error bound is used exactly as in the static case. Semi-static filters can thus be designed to produce error bounds that are less conservative, but at the added cost of run-time computation. Examples of semi-static filters have been proposed by Avnaim et al. [1] and Burnikel et al. [3]. Shewchuk [13] has also proposed an adaptive combination of semi-static filters. In dynamic filters, the error bound is calculated in a syntax-directed fashion, by considering how each primitive operation (such as `add` or `multiply`) propagates errors from input to output. There are two kinds of dynamic filters. In the first kind, the expression for the error bound is derived automatically at compile-time [4]. The other kind of dynamic filter incorporates the error into the number representation [2]. Dynamic filters generally produce the least conservative error bounds, but at a larger performance penalty.

The focus of our work is to better understand the tradeoff in filters between efficiency and robustness. We measure efficiency in terms of running time. Robustness is measured in terms of how often the filter fails. Our study is restricted to semi-static and dynamic filters, as static filters are mostly too conservative to be useful. We compare two classes of filters with respect to the efficiency-robustness tradeoff. Filters in the first class are based on a technique called *error analysis*, that computes an error bound after making the fast filter calculation, and uses this error bound to decide whether the result of the filter calculation can be trusted. Techniques that fall into this category include Shewchuk's work [13] and the filter in LEDA [4]. The error analysis technique is only as good as the formula available for the computing the error bound for the predicate in question. Formulae that are derived manually might generate tighter bounds, but are harder to come up with. On the other hand, automatic generation of error-bound formulae leads to looser bounds. Also, error analysis techniques are designed to work under the assumption that the inputs are error-free. This assumption breaks down when, for example, different geometric algorithms are composed together. The second kind of filter is based on *interval arithmetic.* The core idea in interval arithmetic is to make the roundoff error part of the real number representation. Each real number is represented by an interval representing the corresponding floating-point number and its roundoff error. Arithmetic operations are redefined to operate on intervals. This method has the advantage of eliminating the need to derive an error bound formula at compile-time. Moreover, since the error is part of the representation, interval arithmetic can be used in geometric algorithms that are to be composed with each other. However, representing each real by two floats (the two ends of the interval) increases the cost of arithmetic operations.

Filters based on interval arithmetic(IA) were first introduced by Bronnimann et al. [2]. The authors report empirical results indicating that predicates with IA based filters run with only a minor overhead as compared to predicates using error analysis based filters. They also report that IA based filters fail less often than static filters on degenerate instances. They however do not indicate how the predicates perform inside a larger application for different point distributions, and what overhead exact arithmetic may impose when the filter fails. The interval arithmetic based filters of Bronnimann et al. are used by Funke and Mehlhorn in their Lazy Object-oriented Kernel (LOOK) [6]. They use interval arithmetic as a second level filter after inexact floating-point arithmetic. Since LOOK includes many other techniques as well, their experimental results do not isolate the effect of the interval arithmetic filter.

In this paper, we report the results of a performance evaluation of the use of interval arithmetic filters in two- and three-dimensional geometric predicates. Specifically, our study attempts to answer the following questions:

1. What is the overhead of using interval arithmetic filters in conjunction with exact arithmetic in commonly used geometric algorithms?

2. For what applications and input distributions is interval arithmetic a preferred option over error analysis?

3. What is the effect, if any, of different hardware platforms on the performance of interval arithmetic?

The rest of the paper is organized as follows.In Section 2, we discuss our implementation of an interval arithmetic based filter, and briefly present other filter implementations used in this study. We describe our results in Section 3 and conclude in Section 4.

2 Filter Implementations

2.1 Interval Arithmetic

The use of interval arithmetic in numerical computation was originally proposed by Moore [11]. A real number is replaced by an interval that represents it along with the uncertainty in its representation. Interval arithmetic operations make extensive use of rounding. Let $\underline{a}$ represent the value of a after rounding down and $\overline{a}$ represent that after rounding up. Using this notation, the add, subtract and multiply operations on intervals are defined below:

$$[a_1, a_2] + [b_1, b_2] = [\underline{a_1 + b_1}, \overline{a_2 + b_2}] \quad (1)$$

$$[a_1, a_2] - [b_1, b_2] = [\underline{a_1 - b_2}, \overline{a_2 - b_1}] \quad (2)$$

$$[a_1, a_2] * [b_1, b_2] = [min(\underline{a_1 b_1}, \underline{a_1 b_2}, \underline{a_2 b_1}, \underline{a_2 b_2}), max(\overline{a_1 b_1}, \overline{a_1 b_2}, \overline{a_2 b_1}, \overline{a_2 b_2})] \quad (3)$$

Rounding is necessary because the exact result may not be representable in the finite precision representation.

2.2 Interval Arithmetic Filter Implementation

2.2.1 Architectural Dependence

As can be seen from the preceding section, every interval arithmetic operation potentially requires us to change the rounding mode, to round up or to round down. Rounding mode changes are typically done by setting flags in the *floating point status register(fsr)* of the processor. The IEEE floating-point specification provides for four rounding modes: *Round to nearest*, *Round Down (to $-\infty$)*, *Round Up (to $+\infty$)* and *Round to zero.*

We selected two representative platforms for our experiments that differed on the mechanisms they provided for setting the rounding mode: the DEC Alpha and the Sun UltraSparc. The DEC Alpha allows the rounding mode flag to be specified as part of the floating point arithmetic instruction. This eliminates the overhead of having to set the fsr explicitly. The Sun Sparc, on the other hand, needs an explicit setting of the fsr each time the rounding mode needs to be changed. This rounding mode setting has a negative effect on performance since it breaks the pipeline.

However, this apparent advantage of the Alpha is negated by the following design peculiarity. An arithmetic instruction in the Alpha contains only two bits to specify the rounding mode. One of the four combinations is reserved to mean that the fsr must be looked up for the rounding mode. Thus, only three out of four rounding modes can be specified in the instruction. In particular, we discovered that rounding up on the Alpha requires an explicit setting of the fsr, while rounding down can be specified "on-the-fly". This means that rounding up on the Alpha cannot be specified as part of the arithmetic instruction.

2.2.2 Implementation Details

We implemented interval arithmetic based filters for the lineside and incircle geometric predicates. The target machines included a Sun UltraSparc Enterprise system with 6 Ultrasparc-2/250 MHz processors, and an Alpha 21164 running at 500 MHz.

Our implementation of interval arithmetic is based on the Basic Interval Arithmetic Subroutines(BIAS) library [8], which we optimized for our application. Since we only needed BIAS routines that performed add, subtract and multiply, we converted exactly these routines to macros with gcc ASM directives to perform arithmetic instructions. This is especially required for the Alpha as we need to use instruction mnemonics that specified the rounding mode (the generic instruction generated by the compiler does not suffice).

Optimizations are also made in reducing the number of primitive operations in the predicate implementations. Interval arithmetic multiplication is more expensive than other arithmetic operations because it includes at least two branches and at least two multiply instructions. Therefore, we use as few interval multiply operations as possible in computing the geometric predicates. Secondly, setting the rounding mode is also an expensive operation, and therefore we wish to minimize the number of rounding mode settings. If we represent intervals as they are written, the Alpha needs only one rounding mode setting per predicate call, while the Sparc needs two rounding mode changes per arithmetic operation. One would therefore expect interval arithmetic to be far more expensive on the Sparc as compared to the Alpha.

However, by using an alternative interval representation, the per-operation rounding mode overhead can be eliminated altogether! This representation, also suggested by Bronnimann et al., represents the interval $[a, b]$ as $[-a, b]$. For this representation, rounding up suffices because $\overline{-a} = -\underline{a}$. Thus, the new IA add and subtract operations can be rewritten as:

$$[-a_1, a_2] + [-b_1, b_2] = [\overline{-a_1 - b_1}, \overline{a_2 + b_2}] \tag{4}$$

$$[-a_1, a_2] - [-b_1, b_2] = [\overline{-a_1 + b_2}, \overline{a_2 - b_1}] \tag{5}$$

This implementation eliminates the round down overhead on the Sparc. Analogously, we can also represent the interval $[a, b]$ as $[a, -b]$. This representation only requires rounding down, and completely eliminates the rounding mode overhead on the Alpha. However, rounding in one direction has only a small impact on running time since the rounding mode is only being set twice per predicate call anyway. In experiments, we found that less than 7% of the total predicate time involves changing the rounding mode, if we restrict to rounding in one direction only. We can therefore conclude that architectural differences with respect to rounding mode play little role in the efficiency of interval arithmetic; the only platform specific effect arises from performance differences in floating point instruction execution.

In experiments on the UltraSparc, we found that when compared to BIAS, our implementation of interval arithmetic improves the performance of lineside by a factor of 4 and the performance of incircle by a factor of 3.

2.3 Error Analysis

Several filters based on error analysis(EA) have been proposed. The two EA methods we evaluated interval arithmetic against are the adaptive method of Jonathan Shewchuk [13], and the method of LEDA based on *expression compilation* [9]. We made this choice for a few different reasons. First, they represent two very different approaches to error analysis. Shewchuk's EA method is adaptive, computing answers and the corresponding error bounds at 3 successively higher levels of precision (starting with level "A", which is floating-point arithmetic, and two intermediate levels "B" and "C") before finally resorting to exact arithmetic (level "D"). The error bound formulae are hand derived, and computations performed at a lower level are reused at the next higher level for efficiency. On the other hand, the EA method used by LEDA is completely automated. An expression compiler is used to generate an error bound formula from the code for the geometric predicate. The second reason for our choices is that these implementations are fairly efficient, and have been tested with macrobenchmarks, making it suitable for a "macrobenchmark-based" performance study of the kind presented in this paper. Finally, the availability of source code for these filter implementations as well as the associated macrobenchmarks simplified our evaluation task a great deal.

A detailed description of Shewchuk's filter and of the methods used in LEDA is outside the scope of this paper; we refer the reader to the relevant papers [13, 9] for details.

3 Results

We tested the use of interval arithmetic (IA) against the two error analysis (EA) implementations described in the previous section. We used our IA implementation of geometric predicates within two macrobenchmarks: two dimensional Delaunay triangulation and three dimensional convex hull computation. In addition, we tested the macrobenchmark implementations on three different input point distributions: (1) points generated according to a uniform random distribution in the unit square, (2) points that are approximately co-circular on the unit circle and and (3) a grid of lattice points, tilted so as not to be aligned with the coordinate axes. These are the same point distributions that Shewchuk used in his experiments, with the last two distributions chosen for their degeneracy. All experiments were run on a sets of 50,000 points generated according to the selected point distributions. We generated uniform random, co-circular and co-spherical point sets using the `rbox` program distributed with the Minnesota `qhull` code [10], and the tilted grid point set with our own random point generator. The input points had double precision floating point coordinates.

To test our IA implementation against Shewchuk's EA method, we modified *Triangle*, which is Shewchuk's 2D Delaunay triangulator [12]. For the LEDA EA method, we made modifications to LEDA predicate computation code[1]. In both cases, the modified code using our interval arithmetic filter worked as follows. We first evaluate the determinant using interval arithmetic, which generates the result $[lo, hi]$. If $0 \in (lo, hi)$, we use exact arithmetic else we return the median value of the interval. We experimented with using different implementations of exact arithmetic, including Shewchuk's level D, exact rational arithmetic in the GNU multiprecision (MP) library [7], and LEDA's exact rational arithmetic. Our results are summarized in Tables 1, 2 and 3.

Impl.	Time (sec)	Lineside Exact/Total #(calls)	Incircle Exact/Total #(calls)	Speedup IA/EA
(A) Uniform Random in the unit square (50,000 points)				
EA	1.259	0/425022	0/380652	
IA+Level D	2.125	0/425022	0/380652	0.592
IA+GNU MP	2.131	0/425022	0/380652	0.591
(B) Almost Co-circular on unit circle (50,000 points)				
EA	1.289	0/271728	5319/199461	
IA+Level D	1.663	0/271728	271/199461	0.775
IA+GNU MP	2.45	0/271728	271/199461	0.526
(C) Tilted Grid (50,000 points)				
EA	1.473	12137/377280	52900/255419	
IA+Level D	1.413	0/377280	0/255419	1.042
IA+GNU MP	1.418	0/377280	0/255419	1.039

Table 1: Performance Comparison with Shewchuk's Error Analysis for 2D divide-and-conquer Delaunay triangulation.

Table 1 shows a comparison of IA with Shewchuk's EA implementation, for 2D Delaunay triangulation. Shewchuk's implementation is denoted by "EA". We test this against two

[1]We changed methods `orientation` and `side_of_circle` in class `rat_point` in LEDA 4.1

IA implementations. Both IA implementations are modified versions of *Triangle* where all predicates use interval arithmetic based filters that use exact arithmetic on failure. They differ in the kind of exact arithmetic used; "IA+Level D" uses Shewchuk's level D exact computation, while "IA+GNU MP" uses exact rational arithmetic provided by the GNU MP library. The number of exact calls in the case of Shewchuk's filter is the number of calls made to level B[2], while in our case, it is the number of calls made if the IA filter fails. We can see that for uniform random, there are hardly any predicate calls where exact arithmetic is needed, and so Shewchuk's implementation outperforms both IA implementations due to the overhead of interval arithmetic over floating-point arithmetic. However, for degenerate distributions such as the tilted grid, the IA implementations perform better, as the EA based filter is too conservative. Among the IA implementations, the "IA+GNU MP" implementation is less efficient as it incurs the overhead of several procedure calls. Finally, we note that even though Shewchuk's implementation will outperform IA on non-degenerate distributions, using IA does not lose more than a factor of 2 in performance.

Impl.	Time (sec)	Lineside Exact/Total #(calls)	Incircle Exact/Total #(calls)	Speedup IA/EA
(A) Uniform Random in the unit square (50,000 points)				
EA	4.43	54/789647	875/404175	
IA	4.70	0/789647	0/404175	0.94
(B) Almost Co-circular on unit circle (50,000 points)				
EA	33.2	15239/479223	200476/200476	
IA	5.06	0/479223	10833/200476	6.56
(C) Tilted Grid (50,000 points)				
EA	7.01	105524/786941	2440/358805	
IA	5.06	364/786941	22/358805	1.39

Table 2: Performance Comparison with LEDA's Error Analysis (EA) for 2D Delaunay triangulation using the Dwyer algorithm.

Impl.	Time (sec)	Lineside Exact/Total #(calls)	Speedup IA/EA
(A) Uniform Random in the unit cube (50,000 points)			
EA	17.52	30769/958973	
IA	6.31	10/958973	2.78
(B) Almost Co-spherical on unit sphere (50,000 points)			
EA	7.63	1086/1262285	
IA	8.07	0/1262285	0.95

Table 3: Performance Comparison with LEDA's Error Analysis (EA) for 3D Convex Hull.

Tables 2 and 3 show results of comparing LEDA's EA implementations with our IA implementation for 2D Delaunay triangulation and 3D convex hull respectively. Here "IA" stands for our IA implementation, while "EA" denotes LEDA's EA implementation. Our

[2]levels C and D are never called on our test data

IA implementation is a modified version of LEDA where all predicates first use interval arithmetic to compute the predicate, and if this fails, use LEDA's exact rational arithmetic.

Consider table 2. We can see that for the co-circular and tilted grid distributions IA outperforms EA by a significant factor, which is larger when EA makes a higher proportion of incircle calls. Incircle is inherently more expensive than lineside as it contains almost 8 times as many multiply operations. Moreover, the EA filter fails 100% of the time on incircle calls for the co-circular distribution. The reason for this can be explained as follows. The incircle predicate tests the sign of a polynomial of degree four in the Cartesian coordinates of the input points. If the input points have d-bit precision, the error in a floating evaluation of the polynomial (computed by EA) is about $2^{d^4}2^{-52}$ [9]. For co-circular points the actual value of the polynomial is close to zero and hence the filter will be effective as long as $d \leq 12$. However, our input points are of double precision, hence, LEDA's EA filter always fails. For the uniform random distribution, however, EA makes very few calls to exact arithmetic and hence performs slightly better than IA.

Table 3 shows similar results for convex hull. However, the cases are inverted here; for uniform random points, convex hull does very many lineside tests and a larger proportion of them are between almost collinear points. Hence, EA needs to make more calls to exact arithmetic, causing it to slowdown by 2.78 in comparison to IA.

4 Conclusions

We can conclude from the results in this paper that

1. Interval Arithmetic does not impose an excessive cost as compared to Error Analysis. In addition, it is always a better filter, failing less often.

2. Rounding mode changes are a smaller factor in performance than we originally thought them to be; in particular, architectures such as that of the Alpha offer no major advantage over the Sparc.

3. On degenerate input distributions, Interval Arithmetic can be faster than Error Analysis. The distribution on which IA outperforms EA might depend on the macrobenchmark.

4. Interval Arithmetic can be used in conjunction with exact arithmetic to compose together geometric software without having to worry about generating error-free inputs for each stage. In particular, any error may be propagated in a "lazy" manner between transformations so long as it does not affect robustness; when it does, exact arithmetic can be used to do a "cleanup" operation.

There are many interesting avenues for future work. From a programming language standpoint, it is interesting to see how a code consumer may be guaranteed, via safe programming language constructs, that the library call she makes does not make arbitrary changes in the processor state, such as changing the rounding mode. It might also be useful to extend the performance study to other macrobenchmarks, and study the performance tradeoffs with respect to varying point distributions.

Acknowledgments

This work was initiated during a course organized by Prof. M. Satyanarayanan. The first author would like to thank Satya for his guidance and feedback on the project.

References

[1] Avnaim, F., Boissonnat, J.-D., Devillers, O., Preparata, F., and Yvinec, M. Evaluating Signs of Determinants using Single-Precision Arithmetic. *Algorithmica 17* (1997), 111–132.

[2] Bronnimann, H., Burnikel, C., and Pion, S. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proceedings of the 14th. Annual ACM Symposium on Computational Geometry* (1998), pp. 165–174.

[3] Burnikel, C., Funke, S., and Seel, M. Exact Arithmetic using Cascaded Computations. In *Proceedings of the 14th. Annual ACM Symposium on Computational Geometry* (1998), pp. 175–183.

[4] Burnikel, C., Konemann, J., Mehlhorn, K., Naher, S., Schirra, S., and Uhrig, C. Exact Geometric Computation in LEDA. In *Proceedings of the 11th. Annual ACM Symposium on Computational Geometry* (1995), pp. C18–C19.

[5] Fortune, S., and Wyk, C. V. Static Analysis yields efficient Exact Integer Arithmetic for Computational Geometry. *ACM Transactions on Graphics 15*, 3 (July 1996), 223–248.

[6] Funke, S., and Mehlhorn, K. LOOK: A Lazy Object-oriented Kernel for Geometric Computation. In *Proceedings of the 16th. Annual ACM Symposium on Computational Geometry* (2000), pp. 156–165.

[7] GNU MP package. `http://www.swox.com/gmp`.

[8] Knueppel, O. PROFIL/BIAS - A Fast Interval Library. *Computing 53*, 3-4 (1994), 277–287.

[9] Mehlhorn, K., and Naher, S. *LEDA: A platform for combinatorial and geometric computing.* Cambridge University Press, 1999.

[10] Minnesota Qhull package. `http://www.geom.umn.edu/software/qhull/`.

[11] Moore, R. *Interval Analysis.* Prentice-Hall, 1966.

[12] Shewchuk, J. `http://www.cs.cmu.edu/~quake/triangle.html`.

[13] Shewchuk, J. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Tech. Rep. CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, 1996.

[14] Yap, C., and Dube, T. The Exact Computation Paradigm. In *Computing in Euclidean Geometry, 2nd Edition.* World Scientific, 1995.