The Implementation of Geometric Algorithms

Kurt Mehlhorn   Stefan Näher
Max-Planck-Institut für Informatik, D-66123 Saarbrücken

April 18, 1994

Abstract

Most geometric algorithms are formulated under the exact arithmetic and the non-degeneracy assumptions. However, both assumptions do usually not hold: floating point arithmetic is not exact and inputs are frequently degenerate. We discuss the different approaches to the exact arithmetic problem with a particular emphasis on how it is attacked in the LEDA platform of combinatorial and geometric computing [20, 25]. We refer the reader to the companion paper [5] for a discussion of the degeneracy problem.

1 Introduction

Implementing geometric algorithms is a notoriously difficult task. Most of the difficulties can be traced to the fact that most algorithms in computational geometry are formulated under the *exact arithmetic assumption* and the *non-degeneracy assumption.*

By exact arithmetic assumption we mean that all arithmetic operations are assumed to be performed without error. The standard implementation of real numbers, namely floating point arithmetic, does not satisfy this assumption and so we face the exact arithmetic problem.

In order to explain the second assumption we need to make a definition. Following Emiris and Canny [9] we view a geometric problem P as a function from $\mathbb{R}^{nd}$ to $S \times \mathbb{R}^m$, where n, m, and d are integers and S is some discrete space, modelling the symbolic part of the output (e.g., a planar graph or a face incidence lattice). A problem instance $x \in \mathbb{R}^{nd}$, which for concreteness we view as n points in d-dimensional space, is called *degenerate* if P is discontinuous at x. For example, if $d = 2$ and $P(x)$ is the Voronoi diagram of x, i.e., a straight-line planar graph together with coordinates for its vertices, then x is degenerate iff x contains four cocircular points. By non-degeneracy asumption we mean that the input is assumed to be non-degenerate, or in other words that the algorithm needs to work only for non-degenerate inputs. Actual inputs are not always non-degenerate (e.g., for some geometric problems collinearity of three points is a degeneracy and 100 random points with integer coordinates in $[0, 1000]$ are very likely to contain three collinear points) and hence we face the degeneracy problem.

We survey the different approaches to the exact arithmetic problem with a particular emphasis on how it is attacked in the LEDA platform of combinatorial and geometric computing [21]. For a discussion of the degeneracy problem we refer the reader to [5].

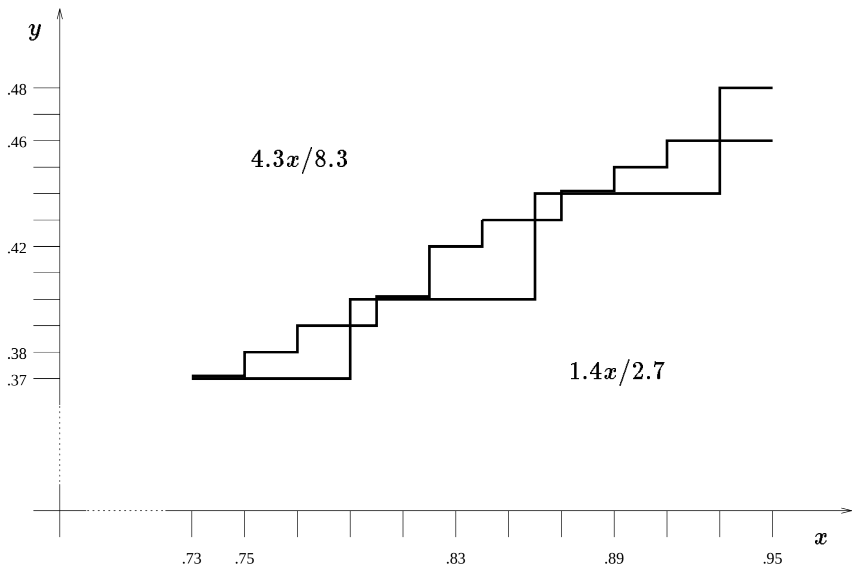

Figure 1: Braided Lines (Ramshaw [27]): The line equations are evaluated with a mantissa of two decimal places.

2 The Exact Arithmetic Problem

Let's first see what strange things floating point arithmetic can do to geometric objects. An illustrative example was given by Ramshaw [27]: the braided lines. Consider two lines with equations $y = 4.3x/8.3$ and $y = 1.4x/2.7$ respectively and assume that we evaluate the line equations with floating point arithmetic with a mantissa of two decimal places. Figure 1 shows what happens: The lines turn into step functions which intersect more than once contradicting a basic fact of planar geometry. In the light of this example it is not surprising that a naive use of floating point arithmetic may have disastrous effects: programs may reach states where the numerical and the combinatorial information is inconsistent and will then abort.

There are basically two scientific approaches to the exact arithmetic problem. One can either design new algorithms that can cope with inexact arithmetic or one can use exact arithmetic.

2.1 Coping with Inexact Arithmetic

This appoach is followed in [8, 10, 11, 12, 14, 15, 16, 17, 21, 23, 24, 28, 29, 30]. We discuss one result in more detail. Li and Milenkovic [19] showed how to compute an approximate convex hull with floating point arithmetic. Given a set S of points in the plane and a positive real ε their algorithm computes a convex polygon P with the following properties: P is contained in the convex hull of S. No point of S has distance from P larger than $12\varepsilon + 54 eps \cdot K$ where eps is the precision of the floating point system (2^{-53} for double precision) and K is the maximal absolute value of any coordinate of the points in S. The vertices of P may be moved by as much as ε without destroying the convexity of P. The algorithm is a highly non-trivial modification of Graham's scan and runs in time $O(n \log n)$.

The other papers cited above describe algorithms for $2D$- and $3D$-Voronoi diagrams and Delauney triangulations, arrangements of lines and algebraic curves, and solid modelling. In many cases the algorithms differ significantly from their exact arithmetic counterparts and that is probably the biggest disadvantage of the inexact arithmetic approach: it does not allow us to directly use the vast body of available geometric algorithms.

2.2 Exact Arithmetic

The alternative approach is to avoid the use of inexact arithmetic and to use exact real arithmetic instead. This road is followed in [1, 4, 13, 18, 20, 26, 31]. In principle, this approach is trivial. After all, it is well known how to compute exactly with integers, rationals, and even algebraic numbers [22]. So the question to be addressed is really: Can exact arithmetic be provided in an efficient and convenient way? Before we discuss this question let us consider two examples to get a feeling what kind of numbers one has to handle in geometric computations.

Consider first the well-known plane sweep algorithm by Bentley and Ottmann [2] for intersecting a set of line segments. One of the geometric tests needed in this algorithm is to compare intersection points by x-coordinate. The x-coordinate of an intersection is of the form x/w where x is a third and w is a second degree homogeneous polynomial in the coordinates of the segment endpoints. Comparing two x-coordinates x_1/w_1 and x_2/w_2 amounts to determining the sign of $x_1w_2 - x_2w_1$, a fifth degree homogeneous polynomial. We conclude that if the coordinates of the segment endpoints are k-bit integers then integer arithmetic with precision $5k + 0(1)$ suffices to perform the test (the exact value of $0(1)$ is 8).

In Voronoi diagram computations (we consider here the Voronoi diagram of line segments) an important test is the incircle test. A vertex of a Voronoi diagram is the center of a circle touching three sites. The incircle test asks for the position (inside, on, or outside) of a fourth site with respect to a Voronoi circle. Assume for concreteness that the three defining sites are the origin p and lines l_1 and l_2 and that the fourth site is a line l_3. Assume further that the equation of line l_i is $a_ix + b_iy + c = 0$ with a_i, b_i, and c_i being k-bit integers. The Voronoi vertex v has coordinates (x_v, y_v) where

$$x_v = \frac{a_1c_2 + a_2c_1 + \sqrt{2c_1c_2(a_1a_2 - b_1b_2 + \sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)})}}{\sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)} - a_1a_2 - b_1b_2}$$

and

$$y_v = \frac{b_1c_2 + b_2c_1 + \sqrt{2c_1c_2(b_1b_2 - a_1a_2 + \sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)})}}{\sqrt{(a_1^2+b_1^2)(a_2^2+b_2^2)} - a_1a_2 - b_1b_2}.$$

The incircle test requires to compare the distance between v and p and between v and l_3 and is therefore tantamount to comparing $x_v^2 + y_v^2$ and $(b_3x_v + b_3y_v + c_3)/(a_3^2 + b_3^2)$. In [4] it was shown that this test can be decided by repeated squaring using arithmetic on integers of length at most 48 k. This is a two order of magnitude improvement over the bound implied by classical root separation bounds [22]. It was also shown in [4] that precision 10 k suffices for the test whether the two quantities are equal or not. More generally, Blömer [3] has shown that the equality test for sums of radicals can be done more efficiently than sign evaluation. We come back to this point below. On the experimental side Burnikel et al report that the full 48 k precision was rarely needed in examples. On average, precision 6 k seems to suffice.

We can now turn to the question how to make exact arithmetic fast and convenient to use. There are two approaches: floating point filters and lazy evaluation.

2.2.1 Floating Point Filters

The idea is simple. When you need to determine the sign of an expression E evaluate it first using floating point arithmetic to obtain an approximation $\hat{E}$ of E. Also perform an error analysis to obtain a bound δ on the absolute value of the difference of E and $\hat{E}$. If $\delta = 0$ or $|\hat{E}| > \delta$ the signs of E and $\hat{E}$ agree. If $|\hat{E}| \leq \delta$ and $\delta > 0$ the floating point filter failed and you need to compute E exactly. For a given test let C_{Exact} and C_{Filter} be the cost of computing E and $\hat{E}$ and δ respectively and let q_{Filter} be the probability that the filter fails. Then $C_{Filter} + q_{Filter} \cdot C_{Exact}$ is the expected cost of the test when using the filter.

The cost C_{exact} is determined by the exact arithmetic package that is used. Fortune and van Wyk [13] have developed a special exact arithmetic package particularly suited for computational geometry. It exploits the fact that in geometry one frequently has good a-priori bounds on the required precision (see above) and that the numbers to be dealt with are "small big numbers" (a few hundred bits). Minimizing C_{Filter} and q_{Filter} are conflicting goals: extreme values are $C_{Filter} = 0$ and $q_{Filter} = 1$ (use no filter at all) and $C_{Filter} = C_{Exact}$ and $q_{Filter} = 0$ (use the exact computation as the filter). What other choices do we have and how do they compare?

We describe three methods for computing error bounds. For concreteness assume that we have to compute $c = a \cdot b$, and have already approximations $\hat{a}$ and $\hat{b}$ and error bounds δ_a and δ_b such that $|a - \hat{a}| \leq \delta_a$ and $|b - \hat{b}| \leq \delta_b$. Let $\hat{c} = a \odot b$ where $\odot$ denotes floating point multiplication. Then $|\hat{c}-c| \leq |\hat{c}-\hat{a}\cdot\hat{b}|+|\hat{a}\cdot\hat{b}-a\cdot b| \leq |\hat{a}\odot\hat{b}-\hat{a}\cdot\hat{b}|+|\hat{a}|\cdot\delta_b+|b|\cdot\delta_a \leq \ eps\cdot|\hat{c}|+|\hat{a}|\cdot\delta_b+ \ max(|\hat{b}+\delta_b|, |\hat{b}-\delta_b|)\cdot\delta_a$ and hence δ_c can be computed by three multiplications and some additions. Actually, the situation is slightly more complicated since we need to compute δ_c using floating point arithmetic. We will not pursue that point further. We call this method *dynamic error analysis*.

Here is another method. Assume inductively that we have computed max_a, ind_a, max_b, ind_b such that max_a and max_b are powers of two and $|\hat{a} - a| \leq ind_a \cdot max_a$ and $|\hat{b} - b| \leq ind_b \cdot max_b$. Then $|\hat{c}| \leq max_a \cdot max_b$ and $|\hat{c} - c| \leq (eps/2 + ind_a + ind_b) \cdot max_c$; i.e., $max_c := max_a \cdot max_b$ and $ind_c := ind_a + ind_b + \ eps/2$. How does one initialize the process? For a leaf x of an expression tree one can either take an a-priori bound for max_x (say the smallest power of two bounding all inputs) or compute max_x as $2^{\lceil \log |x| \rceil}$. We refer to the first variant as *static error analysis* and to the second variant as *semi-dynamic error analysis*. Static error analysis has the advantage that all error bounds can be precomputed and so c_{Filter} is just the cost of evaluating the expression in floating point arithmetic. Semi-dynamic analysis also needs to compute the *max*- and the *ind*-values and has about twice the cost of the static analysis. Dynamic analysis is slightly more expensive than semi-dynamic analysis. Static analysis is used in the LN-package of Fortune and van Wyk [13], semi-dynamic analysis is used in LEDA's Floats [20].

What can we say about q_{Filter}? We report on two experiments with the Bentley-Ottmann line segment intersection algorithm and refer the reader to [1, 13, 26] for further experiments. We intersected n segments. Let size be a random k-bit integer and let $y = 2size/n$. The i-th segment had endpoints $(size + rx1,\ 2 \cdot size - iy + ry1)$ and $(3size + rx2,\ 2size + iy + ry2)$ where $rx1$, $rx2$, $ry1$, $ry2$ are random integers in $[-s, s]$ for some small integer s. Table 2.2.1 gives the result for $s = 3$, $k = 10, 11, \cdots, 100$, and $n = 50$. For each filter we list the percentage of the tests ($\hat{=}$ lexicographic ordering of points) that were left undecided by the floating point filter and the running time. The last column shows the number of tests where the floating point computation would have decided incorrectly. Let us try to explain the experiments.

We showed above that to compare the x-coordinates of two intersection points is tantamount to testing the sign of a fifth degree homogeneous polynomial in the coordinates of the segment endpoints. These coordinates are of the form $size+rx$, $3size+rx$, $2size+iy+ry$, $2size-iy+ry$.

k	dynamic filter		semi-dynamic filter		static filter		errors
	failure rate	running time	failure rate	running time	failure rate	running time	
10	0.15	0.65	0.15	0.67	4.13	0.83	0
20	0.05	0.80	0.05	0.63	26.07	0.78	0
30	1.42	1.03	1.42	0.97	99.99	1.28	123
40	2.96	1.03	3.94	1.10	100	1.50	107
43	15.97	1.23	19.43	1.32	100	1.60	266
46	46.38	1.22	50.87	1.22	100	1.52	1125
50	67.42	2.08	69.36	2.13	100	2.30	3476
53	80.32	2.42	81.00	1.65	100	2.07	5938
60	83.45	1.87	83.45	1.93	100	1.78	8818
100	78.10	2.73	78.10	3.58	100	2.93	8551
total		168sec		162sec		194sec	

Table 1: The difficult example: the total time is for the runs with $h = 10, 20, 30, \cdots, 100$.

k	dynamic filter		semi-dynamic filter		static filter	
	failure rate	running time	failure rate	running time	failure rate	running time
10	0	0.53	0	0.58	6.38	0.75
15	0	0.58	0	0.65	18.33	0.82
20	0	0.73	0	0.80	63.41	1.32
30	0	0.68	0	0.78	61.97	1.33
40	0	0.75	0	0.90	100	1.50
60	0	0.72	0.01	0.75	100	1.62
70	7.37	1.08	8.66	1.08	100	1.92
73	30.10	1.03	31.39	1.27	100	1.68
76	65.93	1.87	64.64	1.38	100	1.87
80	98.08	1.93	96.74	2.10	100	1.93

Table 2: 100 random segments, coordinates are random k-bit integers.

Thus if we write the polynomial in terms of the perturbations rx and ry we essentially obtain a term of order $size^5$ independent of the perturbations (this term is actually zero but has maximal size in intermediate results) plus a term or order $size^4$ times a linear function in the perturbations plus a term of order $size^3$ times a quadratic function in the perturbations $\cdots$. Since the input coordinates are not equal to size but vary between $size$ and $3size$ the various terms are actually spread out over some orders of magnitude. In the semidynamic analysis the error bound δ is about $3size^5 \cdot 2^{-53}$. We conclude that the floating point filter becomes worthless for k about 50 and that we are starting to loose the linear terms for k about 40. For smaller k the filter fails only in those rare cases where the second order term must decide. This is well reflected in Table 2.2.1. The dynamic filter has a slightly smaller error rate but because of its higher cost leads to large total running time. The static filter has a significantly higher error rate (the error bound δ in the static filter is about 1000 times the bound in the semi-dynamic filter) and despite its smaller cost a larger total running time. Table 2.2.1 gives the results for 100 segments whose endpoints have random k bit coordinates. The findings are about the same.

Static error analysis is packaged in the LN system [13]. LN is a precompiler that operates on the arithmetic expressions of a given program. It performs static error analysis, computes the maximal precision required for subexpressions, and generates C++-code for the filter and the optimized exact integer arithmetic. The paper [7] reports on experiences with the LN system.

Semi-dynamic error analysis is packaged in LEDA's Float class. A Float is a floating point approximation to an integer. Besides the arithmetic operations $+, -, *$ Floats also offer the functions Non_zero and Sign. Non_zero (x) either returns ZERO, NON_ZERO or NO_IDEA. The first two outcomes are correct with respect to the underlying exact value. If Non_zero $(x) =$ NON_ZERO then $sign(x)$ returns the correct sign. Internally, Floats manipulate the value of the approximation and also the quantities max and ind mentioned above. The Float filter is very easy to use. Assume that points are given through their homogeneous coordinates and that we want to compare the x-coordinates of two points. We first define a function template cmp_pt that actually makes the comparison between points of arbitrary numerical representation template class

```
numtype int cmp_pt (numtype x1 numtype & w1, numtype & x2, numtype and w2)
{ numtype d = x1 * w2 - x2 * w1;
  int i = NON_ZERO(d);
  if (i == ZERO) return ZERO;
  if (i == NON_ZERO) return sign(x);
  return NO_IDEA
}
```

and then call this function first for the floating print approximations and then (if necessary) for the exact values. Here we assume that for a print p the exact homogeneous coordinates (type Int [1]) are stored in $p.X$, $p.Y$ and $p.W$ and the floating point approximations (type Float) are stored in $p.x$, $p.y$ and $p.w$.

```
int compare (point & p, point & q)
{ int i = cmp_pt(p.x, p.w, q.x, q.w);
  if (i ≠ NO_IDEA) return i;
  return cmp_pt(p.X, p.W, q.X, q.W);
}
```

[1] Int is the type of arbitrary precision integers

Note that a compare-function that uses only exact arithmetic would be written as:

int exact-compare ($point\ p,\ point\ q$)
{ $Int\ d = p.x * q.w = q.x * p.w$;
  $int\ i =$ NON_ZERO(d);
  if ($i \equiv$ ZERO) return ZERO;
  return $sign(d)$;
}

So the changes required for using the filter are minor and purely syntactical.

2.2.2 Lazy Evaluation

For our second example, the Voronoi diagram of line segments, a simple floating point filter does not suffice since the range of possible precisions is too large: 48k is needed in the worst case and 6 to 8k suffices on average. Lazy evaluation is an appropriate approach in this situation: the idea is to represent each number twice, once by its definition (which is either an integer or an operator and pointers to the arguments) and once by an approximation (which is an arbitrary precision floating point number together with an error bound). Whenever the sign of a number needs to be determined the quality of the approximation is improved (say repeatedly doubled) until a decision can be made. Improving the quality is simple when the definiton is an integer. When the definition is an expression, one first has to improve the quality of the operands appropriately.

Assume for concreteness that $x = \sqrt{y}$, that we know already an approximation $\bar{x}$ and a bound δ with $|\bar{x} - x| \le \delta$, and that we want to compute $\hat{x}$ with $|x - \hat{x}| \le \gamma$ for some given γ. We first compute $\hat{y}$ with $|\hat{y} - y| \le max(\gamma^2/4, \gamma/2 \cdot (\bar{x} - \delta))$ and then $\hat{x}$ with $|\hat{x} - \sqrt{\hat{y}}| \le \gamma/2$. We have

$$|\hat{x} - x| \le |\hat{x} - \sqrt{\hat{y}}| + |\sqrt{\hat{y}} - \sqrt{y}| \le \frac{\gamma}{2} + \frac{|\hat{y} - y|}{(\sqrt{\hat{y}} + \sqrt{y})} = \frac{\frac{\gamma}{2}(1 + max(\frac{\gamma}{2}, \bar{x} - \delta))}{(\sqrt{\hat{y}} + \sqrt{y})} \le \gamma$$

if either $\bar{x} - \delta \ge \gamma/2$ (note that $\sqrt{y} \ge \bar{x} - \delta$) or if $\bar{x} - \delta < \gamma/2$ and $\sqrt{\hat{y}} \ge \gamma/2$ or $\sqrt{y} \ge \gamma/2$. If neither is the case then $\sqrt{y} < \gamma/2$ and $\sqrt{\hat{y}} \le \gamma/2$ and hence $\hat{x} \le \gamma$. Thus $|\hat{x} - \sqrt{y}| \le \gamma$ also in the last case.

Lazy evaluation is very effective in determining the sign of a non-zero number. It does so using close to the minimal possible precision. It fails however miserably when it has to determine the sign of the number zero. Here, the user either has to provide a separation bound (if x is non-zero then $|x|$ is at least...) or a special prupose procedure that decides zeroness. For the Voronoi diagram of line segments $2^{-(48k+0(1))}$ is an appropriate separation guarantee; we also mentioned above there is an effective special purpose zero test.

Lazy evaluation is packaged in the LEA-system [1] and also in LEDA's class Algebraic. The class Algebraic offers the arithmetic operations $+, -, /, *, \sqrt{}$ the tests Non_zero and $Sign$, and a hidden function $Improve\_quality$. Moreover, every Int is an algebraic number. Internally, an algebraic number is represented by its definition def (either an Int or an operator plus pointers to the arguments), an approximation val and a bound $delta$ such that $|x - val| \le delta$. Both val and $delta$ are of type Bigfloat, the type of arbitrary precision floating point numbers. A call $x.Improve\_quality(q)$ recomputes val and $delta$ such that $|x - val| \le delta \le q$. A call

$x.Sign()$ repeatedly halves $delta$ and calls $x.Improve\_quality(delta)$ until either $delta = 0$ or $|x.val| > delta$. It then returns the sign of $x.val$. Note that $x.Sign()$ is not guaranteed to terminate if x is zero. There are two versions of the function Non_zero. For the first version the user has to provide a separation guarantee g, i.e., in $x.Non\_zero(g)$ the implemention is allowed to use the fact that $|x| \leq g$ implies $x = 0$, and for the second version, the user has to provide a special purpose zero test, i.e., in $x.Non\_zero$ (my_test) the responsibility is immediately shifted to my_test. Thus a user of the class Algebraic either has to prove a separation guarantee or design a special purpose zero test. Either task is non-trivial but manageable as the experience with the Voronoi diagram computation shows.

References

[1] M.O. Benouamer, P. Jaillon, D. Michelucci, and J-M. Moreau. A "lazy" solution to imprecision in computational geometry. In *Proc. of the 5th Canadian Conference on Computational Geometry*, pages 73–78, 1993.

[2] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[3] J. Blömer. Computing Sums of Radicals in Polynomial Time. Technical Report No. B93-13, Freie Universität Berlin, 1993.

[4] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. manuscript.

[5] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 16–23, 1994.

[6] J. Canny, B.R. Donald, and E.K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. of the 8th ACM Symp. on Computational Geometry*, pages 251–260, 1992.

[7] J. Chang and V. Milenkovic. The practicability of using LN for efficient exact geometric computations. In *Proc. 5th Canad. Conf. Comput. Geometry*, pages 67-72, 1993.

[8] T.K. Dey, K. Sugihara, and C.L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Computer Aided Geometric Design*, 9:457–470, 1992.

[9] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. of the 8th ACM Symp. on Computational Geometry*, pages 74–82, 1992.

[10] S. Fortune. Stable maintenance of point-set triangulations in two dimensions. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Sience*, pages 494–499, 1989.

[11] S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations and Voronoi diagrams. In *Proc. of the 8th ACM Symp. on Computational Geometry*, pages 83–92, 1992.

[12] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. of the 7th ACM Symp. on Computational Geometry*, pages 334–341, 1991.

[13] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th ACM Symp. on Computational Geometry*, pages 163–172, 1993.

[14] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proc. of the 5th ACM Symp. on Computational Geometry*, pages 208–217, 1989.

[15] C.M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufman, San Francisco, 1990. Chapter 4.

[16] J.E. Hopcroft and P.J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7:339–380, 1992.

[17] H. Ingaki, K. Sugihara, and N. Sugie. Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 334–339, 1992.

[18] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[19] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.

[20] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the segment intersection problem. manuscript.

[21] K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. In *LNCS*, 379:88–106, 1989.

[22] M. Mignotte. Identification of Algebraic Numbers. *Journal of Algorithms*, 3:197–204, 1982.

[23] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Sience*, pages 500–505, 1989.

[24] V. Milenkovic. Robust polygon modelling. *CAD*, 25(9):546–566, 1993. Special Issue: *Uncertainties in geometric design.*

[25] S. Näher. LEDA Manual. Technical Report No. MPI–I–93–109. Max-Planck-Institut für Informatik, 1993.

[26] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. of the 3rd ACM Symp. on Computational Geometry*, pages 119–125, 1987.

[27] Ramshaw. Braided lines. Unpublished report. 1992.

[28] P. Schorn. *Robust Algorithms in a Program Library for Geometric Algorithms.* PhD thesis, Informatik-Dissertationen ETH Zürich, 1991.

[29] K. Sugihara. A simple method for avoiding numerical errors and degeneracies in Voronoi diagram construction. *IEICE Trans. Fundamentals*, E75-A(4):468–477, 1992.

[30] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *Journal of Information Processing*, 12(4):380–393, 1989.

[31] C.K. Yap. Towards exact geometric computation. In *Proc. of the 5th Canadian Conference on Computational Geometry*, pages 405–419, 1993.