Invited Lecture: Real Numbers and Robustness in Computational Geometry

Stefan Schirra [a]

[a] *Department of Computer Science, Otto von Guericke University Magdeburg, 39106 Magdeburg, Germany*

Abstract

Robustness issues due to imprecise arithmetic used in place of exact real number computation are a notorious problem in the implementation of geometric algorithms. We briefly address some robustness issues and discuss approaches to resolve them.

Key words: computational geometry, robustness, exact geometric computation

1 Introduction

Computational Geometry as a discipline has its roots in theoretical computer science. The central issues are designing efficient algorithms and data structures for and investigating the intrinsic complexity of geometric problems [3,13,42]. The computational model most frequently used in Computational Geometry is the so-called real RAM, a powerful random access machine that can store and perform exact arithmetic operations with arbitrary real numbers. Thus, by definition of the computational model, real number computation is not an issue in Computational Geometry.

Practice, however, is different. Precision caused robustness problems are often a nightmare for the programmer of a geometric algorithm if standard floating-point arithmetic is used as a substitute for real numbers [27,22,19,51]. Program failures caused by imprecise computation are very hard to find and seem to be very hard to resolve. Failures caused by imprecision include program crashes, infinite loops, and inconsistent or totally wrong output. We present some examples in the next section and discuss approaches to deal with such problems subsequently.

Email address: stschirr@isg.cs.uni-magdeburg.de (Stefan Schirra).

2 Robustness Issues in Computational Geometry

Often, imprecise computation causes program crashes. This is in contrast to purely numerical computations. There, no combinatorial structures are involved and, with the exception of division by zero, some output is always computed. In geometric computing, program crashes are caused since imprecise computation produces inconsistent decisions that lead algorithms into states that contradict basic axioms of geometry and hence cannot be handled by the algorithm. A well-known example of geometrically impossible situations caused by imprecise computation are Ramshaw's braided lines [43,36]: Because of the inaccurate comparison of y-coordinates of two different straight lines at different x-coordinates, with limited precision floating-point arithmetic, a program might conclude that the lines are braided and intersect more than once!

A similarly simple violation of geometric axioms can be observed when one computes an intersection point p of two lines ℓ_1 and ℓ_2 and tests whether p lies on ℓ_i, $i = 1, 2$, both with limited precision floating-point arithmetic. The problem arises with both almost parallel lines as well as with lines where intersection point computation is not ill-conditioned. Most of the time, at least one of the tests fails, see Fig. 1 and the robustness demo in the 2.x releases of the computational geometry algorithms library CGAL [10].

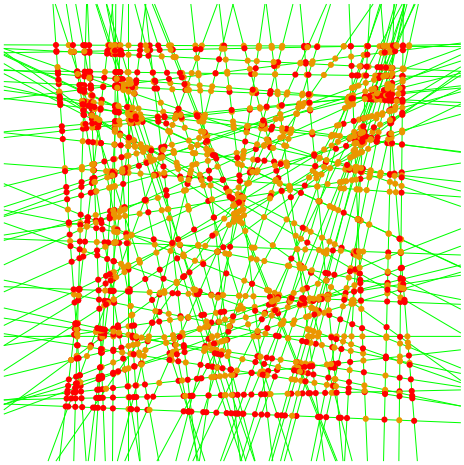

Fig. 1. Testing whether the intersection point of a horizontal and a vertical line lies on both lines with floating-point arithmetic. Whenever one of the tests fails, a dot is drawn. The dot is dark if both test fail. According to floating-point arithmetic, only 20% of the intersection points lie on both lines in this typical example.

Such violations of the axioms and theorems of Euclidean geometry can lead to catastrophic errors, even with the simplest algorithms in Computational Geometry like computation of the convex hull of a set of points in the plane.

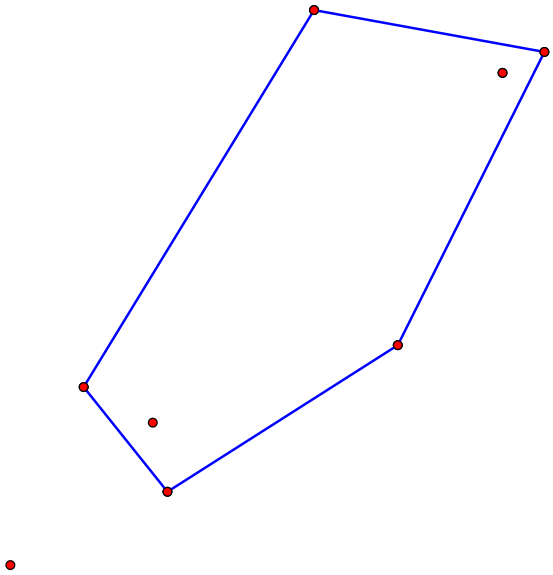

Fig. 2. Incorrect convex hull of 9 points in the plane computed with an incremental convex hull algorithm based on floating-point arithmetic. Note that two of the points are almost identical and thus only 8 points are visible.

Fig. 2 shows a set of points where the convex hull computed with an incremental convex hull algorithm implemented with double precision floating-point arithmetic does not contain a point which clearly lies outside of the computed hull polygon. There are even examples, where a point outside is orders of magnitude away from the hull computed with floating-point arithmetic, see [30] for concrete coordinates.

The basic theorems on which the correctness of incremental convex hull computation is based, and possible violations of these theorems with floating-point computation are discussed in [30]. For example, with incremental convex hull computation, an edge e of the current hull is removed when point q is added if the supporting line of e separates q from the current hull. In this case we say that the point q *sees* the edge e. There are examples where a point outside the current convex hull sees all edges of the convex hull according to floating-point arithmetic. Again, such a situation is geometrically impossible. Depending on the data structure used to represent a convex hull polygon and on the update strategy (immediate or postponed updates), an implementation will crash or loop forever in such a case; see [30] for further details.

Program crashes are probably the most frequent form of catastrophic errors due to precision problems. However, it is debatable whether crashing or garbage in the output is more annoying. With a crash it is obvious that there is something wrong while the garbage might be undetected as such, since usually there is no post-processing check [2,34] for correctness.

3 Approaches to Resolve Robustness Problems

Robustness problems arise because the substitution used for real number computation in practice does not behave like the exact arithmetic assumed in theory, when geometric algorithms or data structures are designed and proved to be correct. There are two obvious approaches to resolve this contradistinction: (1) Take the imprecision of the arithmetic into account when an algorithm is designed in theory, or (2) compute exactly in practice. The phrasing in (2) needs some explanation. What we need is a correct combinatorial part of the output. In order to achieve this, we request that all decision made by a program are made as if they had been made with exact computation. However, we do not ask for numerical exactness. It suffices to have some symbolic representation that allows for computing approximations to whatever precision we want. The subtleness concerns precision and decision. There is no need for precise numerical values, but for correct decisions that guarantee that the control flow in the program is the same as in its theoretical counterpart on a real RAM. Fortunately, the second approach is feasible for most problems studied in Computational Geometry, at least, theoretically. The second approach is known as the *exact geometric computation paradigm* [49,50].

The first approach is not attractive because it requires redesigning of the rich collection of algorithms devised in Computational Geometry based on the real RAM. Still, there are some approaches in this direction, e.g. [23,38,26], see [51,44,20] for more complete listings. The most promising among the approaches in the first category is *topology-oriented implementation* [46]. It has been applied to a number of geometric problems, especially various kinds of Voronoi-diagram computations [47]. The idea is to rely on logical and combinatorial computations but not on the numerical part. With topology-oriented implementation, a program computes some output even if the results of all numerical computations are replaced by random numbers. In spite of this, the combinatorial part of the output is guaranteed to have certain certified properties and is the correct combinatorial output for some perturbation of the input. Logical and combinatorial computations are used to avoid inconsistencies, such that a program never crashes because of contradicting decisions based on imprecise numerical computation. Of course, the computed output might not be correct. Actually, computed output and correct output might be very different. For example, the convex hull from Fig. 2 is a perfect output in the sense of topology-oriented implementation because it is convex and the correct output for some perturbation of the input points, but is it useful?

Robustness issues are closely related to degeneracies. Degeneracies are special configurations of geometric objects like three or more collinear points or four or more cocircular points in the plane. Degeneracies can be a curse in the implementation of a geometric algorithm published in a conference or journal

paper, because handling degeneracies is often intentionally not discussed in publications, but left to the reader. Detecting a degeneracy involves some equality testing.

4 Exact Geometric Computation

Exact geometric computation assures that all decisions made by a program are correct. Without loss of generality, we may assume that branching in decision steps always depends on the sign of some real number computed during program execution only. The goal with exact geometric computation is to compute exact signs, but not necessarily exact numerical values. If you consider different levels of geometric computing, with arithmetic as the lowest level and geometric predicates as the next higher level, exact geometric computation assures exactness on the level of geometric predicates, not necessarily on the arithmetic level.

Exact computation is a prerequisite for symbolic perturbation, [17,45,48] which has been proposed as a mean to deal with degeneracies, i.e. rather to avoid dealing with them. More recently, controlled (explicit) perturbation [24,25] has become another alternative to avoid handling of degeneracies. If there is a need for exact solutions, degeneracies must be handled. However, most of the time, the primary motivation for using exact geometric computation is rather reliability of the software than exactness.

Exact geometric computation is known to be feasible for almost all problems in Computational Geometry. Here we assume that all numerical values in the input data are rational numbers. This is not a severe restriction as all real numbers representable by floating-point and integer number types provided by standard programming languages are indeed rational numbers. If all geometric objects involved are linear objects (e.g. points, straight line segments, ray, ...), computations stay within the field of rational numbers and thus can be done exactly, for example using an arbitrary precision integer number type to represent numerators and denominators. Geometric computations with non-linear geometric objects usually involve irrational numbers as well. Fortunately, for most geometric problems, they involve real algebraic numbers only. There are some exceptions, for example, computing the shortest path amidst a set of discs in the plane, where the length of a path usually is a transcendental real number [11].

It is well known that exact computation with real algebraic numbers is feasible [33]. In the classical approach, a real algebraic number α is represented by a (square-free) polynomial P and an isolating interval I, such that $\alpha \in I$ and α is the only root of P in I. There are algorithms to perform basic arithmetic oper-

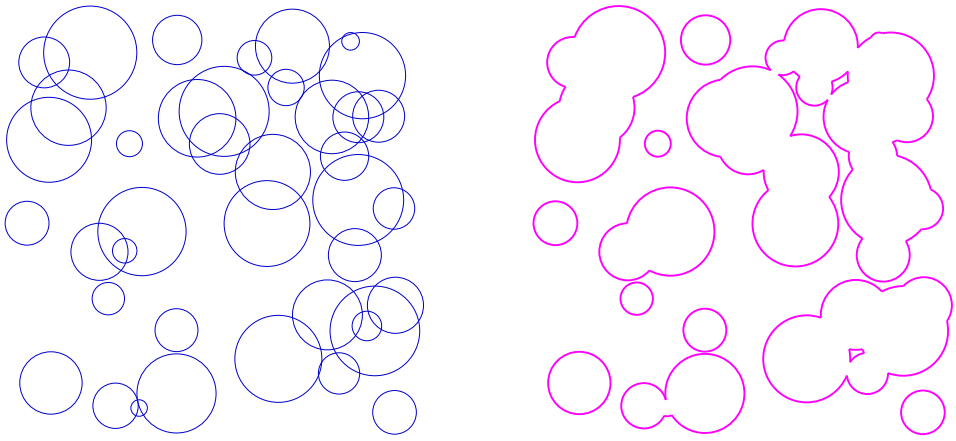

Fig. 3. Union of discs.

ations with such representation. Furthermore there are algorithms to compare two real algebraic numbers given in this representation, see also [39]. Thus, like in the rational case, exact computation with basic arithmetic operations is possible. Computing isolating interval representation for the roots of a univariate polynomial with rational or real algebraic coefficients is possible as well. Therefore, for example, exact computation involving square root operations is feasible, too.

Surprisingly, even in the presence of non-linear geometric objects, rational numbers are sometimes sufficient. Let us assume we are given a number of discs in the plane and we want to compute (the boundary of) their union, see Fig. 3. Each circle is given by three defining points with rational Cartesian coordinates. The union can be computed using inversion and duality, see [13,35]. Only the very last step of this algorithm, computing the circular arcs of the boundary, involves potentially irrational numbers. In all intermediate steps all occurring numbers are rational. The irrational numbers in the last step can be avoided, too, if a symbolic representation for the circular arc endpoints as a specified intersection point of two circles is used in the output.

The challenge with exact geometric computing, both with rational and real algebraic numbers, is efficiency, since a simple substitution of the arithmetic by arbitrary precision rational arithmetic or real algebraic arithmetic based on isolating interval representation can slow down computation by several orders of magnitude compared to pure hardware-supported floating-point arithmetic [1] [29]. Exact geometric computation became an adequate alternative by

[1] Of course, here we compare apples and oranges. On one hand, we have a program that is fully reliable, on the other hand we have a program which sometimes crashes or computes garbage. Furthermore, we can compare running times only for those cases where both programs compute some more or less meaningful result. So this comparison is unfair.

the use of so-called floating-point filters [4,7,14,21,29,31]. The idea is to use floating-point computation whenever it is known to be reliable. In a floating-point filter, a floating-point computation is combined with an error computation. Whenever the error is zero or smaller than the computed absolute floating-point approximation, the sign of the floating-point approximation is the sign of the exact value. Only if the correctness of the floating-point computation cannot be detected this way, the computation is redone using an alternative exact method. Since degeneracies or near degeneracies are rare in many applications, floating-point filters often lead to a significant speed-up compared to sole use of much slower exact rational or algebraic arithmetic.

The smaller the error bounds computed by a floating-point filter, the more often the filter will assert correctness of a floating-point computation. Depending on the computation of error bounds and approximations, one distinguishes static filters, semi-static filters, and dynamic filters. With static filters, a-priori-knowledge about the size of the operands and the operations to be performed is used to compute an error bound beforehand. Thus, if all floating-point computations are detected as reliable by such simple filters, all you have to pay for in terms of running time in addition to a program purely based on floating-point computation are comparisons with precomputed error bounds. Semi-static filters precompute an error bound partially. At run time, maximum size of the operands is computed and combined with the precomputed part. Dynamic floating-point filters do not make any a priori assumptions about operands and operations. The most prominent representative of this species is interval arithmetic (based on floating-point numbers). An interval gives us both an approximation and a corresponding error bound and tells us the sign if it does not contain zero.

The trade-off between speed and the quality of approximation and error bound can be used to built cascaded filters. After a filter failed, one can use a more selective, but also more time-consuming filter, before one switches to exact arithmetic.

In any case, these techniques require the ability to recompute a numerical value and therefore require access to an exact representation of the operands of a computation. In order to enable recomputation, expression trees are used in the LEA system for lazy rational arithmetic [1], the number type `Expr` in the CORE library [28], and the number type `real` [9,5] in LEDA [36] to record the computation history of a numerical value. Pointers to the trees representing the operands of an arithmetic operation are maintained when an operation is performed. Actually, we do not have trees but directed acyclic graphs (dags) since there might be more than one pointer to the same operand, see Fig. 4 for an example.

Expression dags enable lazy adaptive evaluation of the sign of a number given

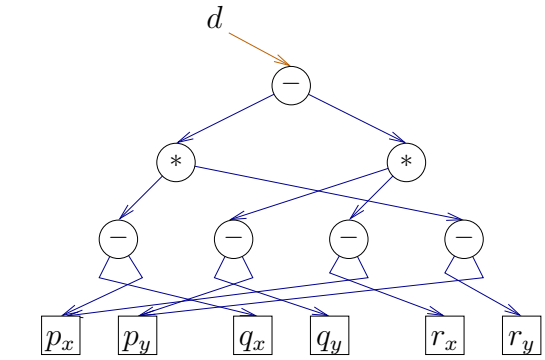

Fig. 4. Expression dag for orientation computation according to the expression shown on top of the figure. p_x, p_y, q_x, q_y, and r_x, r_y are Cartesian coordinates of three points. The sign of d gives us the orientation of the points, i.e., it tells us whether the sequence of points forms a left turn, a right turn, or whether the points are collinear.

by some expression. An approximation with high precision is computed only if this is necessary. If the result of a computation is not close to zero, a rough approximation is sufficient to determine the sign. Only if a current approximation of a numerical value is not sufficient to determine the sign, the computation history stored in the expression dag is used to recompute a better approximation with higher precision. This is iterated with higher and higher precision if necessary. Software floating-point number types with variable mantissa length (bigfloats) are used to compute these approximations.

If the true sign of a numerical value is not zero, this strategy terminates since the precision of the computed approximation is increasing and the error bound is decreasing, such that finally the error bound will become smaller than the absolute value of the approximation of the non-zero numerical value. In order to detect that a value is zero, however, additional information is needed. A separation bound (sometimes also called root bound) for an arithmetic expression is a lower bound on the absolute value of the expression, if the value is non-zero. Since in concrete program code, the expression used in a computation is fixed and since all possible operands belong to a finite set of values, e.g., the set of all integers representable by 32 bits or the set of double precision floating-point numbers, there are only finitely many possible value and hence the value of the expression cannot be arbitrarily close to zero. Thus there is a gap between zero and next smallest absolute value and hence the existence of a separation bound is obvious; the crux is to find constructive separation bounds that can be computed efficiently. Concerning constructive separation bounds, there has been much progress concerning the strictness of the bounds as well as the supported operations [6,32,8,37,41]. The most recent versions allow for computation of separation bounds for all kinds of real alge-

braic numbers whereas earlier constructive separation bounds, especially those used in `Expr` and `leda::real` support only a subset of real algebraic numbers, namely the closure of the integers under the basic arithmetic operations (including division) and square root resp. kth-root operations.

For the sake of concreteness, we state a simplified constructive separation bound for the restricted division-free case. Let E be an arithmetic expression involving addition, subtraction, multiplication and kth root operations only. For expression E, a real number $u(E)$ is computed according to the following table:

E	$u(E)$
0	1
integer N	$\lceil \log \lvert N \rvert \rceil$
$E_1 \pm E_2$	$\max(u(E_1), u(E_2)) + 1$
$E_1 \cdot E_2$	$u(E_1) + u(E_2)$
$\sqrt[k]{E_1}$	$\lceil u(E_1)/k \rceil$

Then the absolute value of expression E is either zero or at least

$$\left((2^{u(E)})^{\deg(E)-1} \right)^{-1}$$

where $\deg(E)$ is a bound on the algebraic degree of the value of E.

There are further techniques to compute with real algebraic numbers exactly, e.g. coding by polynomials and sign sequences [12]. Such techniques have not been exploited in Computational Geometry yet.

The number types `Expr` and `leda::real` use expression dags, iterated floating-point evaluation with higher precision, and separation bounds to guarantee correctness of all comparison operations between such numbers. The sign evaluation is called adaptive because the amount of work for the sign computation depends on the size of the absolute value. Easy cases can be decided quickly, only difficult cases take longer. Let us call sign computations, where the sign is zero, degenerate, analogously to geometric degeneracies. If there are hardly any degenerate cases, the lazy evaluation strategies are faster than the exact arithmetic alternatives; if there are many degeneracies, it might be better to use exact arithmetic right from the beginning.

CORE's `Expr` and `leda::real` are general purpose tools that put exact geometric computation into effect on the arithmetic level in an extremely user-friendly way. Usually, for a special geometric problem there are adjusted more efficient ways that implement exact geometric computation on the level of

geometric predicates. For linear geometry, theory and practice of designing efficient exact predicates are already quite advanced. First advanced results for non-linear geometry have been achieved in the Effective Computational Geometry project [16] recently.

Exact geometric computation is used CGAL [10,18,40] and in the geometric part of the LEDA library [36]. Both are software libraries implemented in C++. LEDA provides two exact geometry kernels, one for purely rational computations and another one based on `leda::reals`. These kernels use floating-point filters to speed up exact geometric computation. The CGAL library is largely based on the generic programming paradigm known from the standard template library of C++. Once familiar with this paradigm and the use of traits classes, the user can easily choose among geometry kernels with various kinds of filters and support of exact geometric computation as well as exchange the underlying number type(s).

Beyond the use in Computational Geometry, especially the above mentioned libraries, exact geometric computation has not yet found widespread application because it slows down computation even in those cases where floating-point arithmetic works well. Apparently, users are rather willing to accept catastrophic failures in rare and not so rare cases. Moreover, in many applications, exactness in its own right is not an important objective because the geometric data is known to be inaccurate anyway, for example, due to measurement errors. If real algebraic numbers of high algebraic degree are involved, exact geometric computation is not yet a valid alternative because performance slows down tremendously.

5 Equality Testing

Equality testing is unavoidable if degeneracies must be treated correctly. If correct treatment of degeneracies is not necessary, topology-oriented implementation is an alternative where equality testing of numerical values is not needed. Anyway, equality testing should be avoided whenever possible. With adaptive exact arithmetic it is most expensive and with imprecise arithmetic equality will not be detected correctly. Often equality or zero testing can be easily avoided by geometric considerations. For example, assume that one wants to compute the intersection points of two circles. Using elimination, one can compute the x- and y-coordinates of the intersection points, both of the form $\alpha \pm \sqrt{\beta}$. In order to find the correct combinations among the four possible pairs, one is tempted to plug the pairs of coordinates into the circle equations and to check whether this yields zero. However, a comparison of the coordinates of the centers of the circles suffices to find the valid combinations, see Fig. 5.

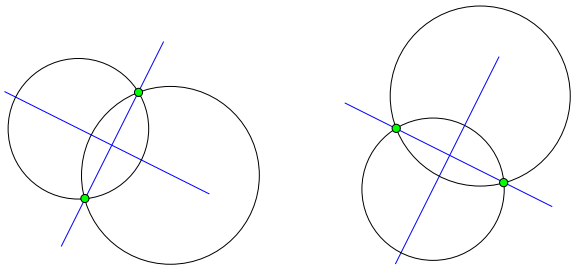

Fig. 5. There is no need for equality testing when intersection points of circles are computed.

For some geometric problems, degeneracies arise self-acting, however. Consider computation of the Delaunay triangulation of points which are given as intersection points of circles, see Fig. 6. Here it is very likely that there are four or more cocircular points.

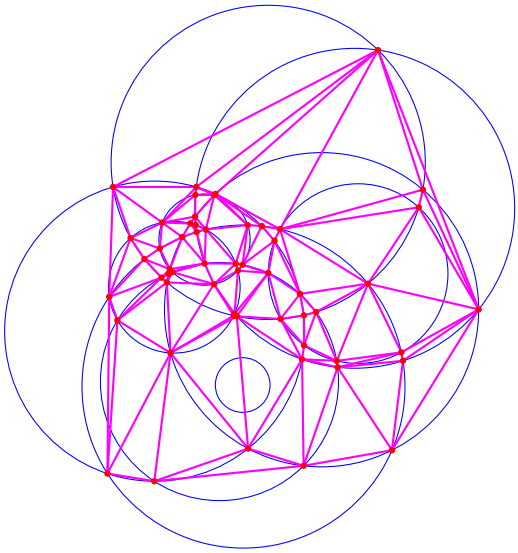

Fig. 6. Delaunay triangulation of intersection points of circles.

6 Conclusions

The exact geometric computation paradigm has been successfully applied to a large number of problems in Computational Geometry. This was possible since almost all geometric problems are algebraic in nature if we assume that all input coordinates are rational numbers (or algebraic numbers). The assumption that all input numbers are rational is a realistic one for practical applications and not a restriction. Nevertheless, there are first approaches to extend exact geometric computation to non-algebraic computations [11,15].

The selection of examples and techniques presented here is certainly biased and definitely incomplete. For further discussion, examples and references we refer the interested reader to survey papers on precision and robustness issues in Computational Geometry like [51] or [44]. A current challenge in this area is to avoid the growth of the complexity of the real numbers involved in cascaded geometric computations by rounding geometric objects to simpler ones while preserving as much of the topology as possible.

References

[1] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.

[2] Manuel Blum and Sampath Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, January 1995.

[3] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.

[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, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 341–350, 1999.

[6] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27(1):87–99, 2000.

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

[8] Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. A separation bound for real algebraic expressions. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 254–265. Springer-Verlag, 2001.

[9] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[10] CGAL. `http://www.cgal.org`.

[11] Chien Chang, Sung Woo Choi, DoYong Kwon, Hyungju Park, and Chee Yap. Shortest path amidst disc obstacles is computable.

[12] M. Coste and M. F. Roy. Thom's lemma, the coding of real algebraic numbers and the computation of the topology of semi-algebraic sets. *Journal of Symbolic Computation*, 5:121–130, 1988.

[13] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[14] Olivier Devillers and Franco P. Preparata. Further results on arithmetic filters for geometric predicates. *Comput. Geom. Theory Appl.*, 13:141–148, 1999.

[15] Z. Du, M. Eleftheriou, J. Moreira, and C. Yap. Hypergeometric functions in exact geometric computation. *Electronic Notes in Theoretical Computer Science*, 66(1), 2002. Proceedings, Fifth Workshop on Computability and Complexity in Analysis, Malaga, Spain. July 12-13, 2002.

[16] ECG. Effective computational geometry for curves and surfaces. `http://www-sop.inria.fr/prisme/ECG/Results`.

[17] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[18] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[19] A. R. Forrest. Computational geometry in practice. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, volume F17 of *NATO ASI*, pages 707–724. Springer-Verlag, 1985.

[20] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Computational Geometry*, pages 81–128. Information Geometers, 1993.

[21] 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.

[22] W. R. Franklin. Cartographic errors symptomatic of underlying algebra problems. In *Proc. Internat. Sympos. Spatial Data Handling*, volume 1, pages 190–208, 20–24 August 1984.

[23] Leonidas J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[24] Dan Halperin and Eran Leiserowitz. Controlled perturbation for arrangements of circles. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 264–273. ACM Press, 2003.

[25] Dan Halperin and Christian R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comput. Geom. Theory Appl.*, 10:273–287, 1998.

[26] M. Held. Vroni: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Comput. Geom. Theory Appl.*, 18:95–123, 2001.

[27] C. M. Hoffmann. The problems of accuracy and robustness in geometric computatio n. *IEEE Computer*, 22(3):31–41, March 1989.

[28] 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*, pages 351–359, 1999.

[29] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, January 1991.

[30] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classrom examples of robustness problems in geometric computations. In *Proc. ESA 2004*, 2004.

[31] Chen Li, Sylvain Pion, and Chee Yap. Recent progress in exact geometric computation. *Journal of Logic and Algebraic Programming*, 2004. To appear, special issue on "Practical Development of Exact Real Number Computation".

[32] Chen Li and Chee Yap. A new constructive root bound for algebraic expressions. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 496–505. Society for Industrial and Applied Mathematics, 2001.

[33] R. Loos. Computing in algebraic extensions. In B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 173–187. Springer-Verlag, 1983.

[34] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Comput. Geom. Theory Appl.*, 12(1–2):85–103, 1999.

[35] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[36] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[37] Kurt Mehlhorn and Stefan Schirra. Exact computation with leda_real - theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic-algebraic Methods and Verification Methods*. Springer Mathematics, Wien, 2001.

[38] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artif. Intell.*, 37:377–401, 1988.

[39] Bhubaneswar Mishra. *Algorithmic algebra*. Springer-Verlag New York, Inc., 1993.

[40] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, May 1996.

[41] Sylvain Pion and Chee K. Yap. Constructive root bound for k-ary rational input numbers. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 256–263. ACM Press, 2003.

[42] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, October 1990.

[43] L. Ramshaw. CSL notebook entry: The braiding of floating point lines. Unpublished note, 1982.

[44] Stefan Schirra. Robustness and precision issues in geometric computation. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[45] R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete Comput. Geom.*, 19:1–17, 1998.

[46] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation - an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.

[47] K. Sugihara, Y. Ooishi, and T. Imai. Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 36–39, 1990.

[48] C. K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10:349–370, 1990.

[49] C. K. Yap. Towards exact geometric computation. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 405–419, 1993.

[50] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[51] Chee K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook in Computational Geometry*. CRC Press, 2004. Completely revised and expanded Chapter.