On the Use of Adaptive, Exact Decisions Number Types Based on Expression-Dags in Geometric Computing

Stefan Schirra*

Abstract

We discuss how (not) to use number types based on expression dags and adaptive precision in geometric computing. Such number types provide exact decisions for (a subset of the) real algebraic numbers.

1 Introduction

Wrapped in a C++ class, expression-dag-based adaptive-precision number types support exact geometric computing in an utmost user-friendly way. Such number types like `CORE::Expr` [12, 33] and `leda::real` [4, 6, 17, 18] guarantee exact decisions in geometric computing for (a subset of the) real algebraic numbers. Since all decisions are exact, inconsistencies caused by numerical imprecision are abandoned and the correctness of the combinatorial part of a geometric computation is ensured. Thanks to the wrapping in a number type a user need not know how it works. The available number types of this kind support a subset of the real algebraic numbers that includes the rational numbers and is closed under the basic arithmetic operations $+$, $-$, $*$, $/$ and $\sqrt{}$.

You can use these number types like other number types provided in the programming language. There are a few caveats, however, that we are going to address in this paper. Furthermore, some sales messages regarding the number types `CORE::Expr` and `leda::real` are debatable. We will have a closer look at them and revise some conclusions.

Geometric algorithms branch on geometric predicates, e.g., on the orientation of three points in the plane. The evaluation of a geometric predicate usually amounts to the evaluation of the sign of an arithmetic expression. If such a sign computation returns a wrong sign due to numerical impression, the computation may go astray, see [13] for illustrating examples. The exact geometric computation approach to reliable geometric computing [15, 16, 32, 30] proposes a simple remedy: Evaluate all geometric predicates exactly. Plugging exact decisions number types into a parameterized CGAL kernel [3, 7] lets you apply the exact geometric computation paradigm very easily. You get a foolproof geometry kernel that you can use in implementation projects accompanying classes teaching computational geometry without ever worrying about numerical imprecision and its consequences. It lets you focus on algorithmic issues and have a course on geometry instead of a course on numerical mathematics. Using the exact geometric computation paradigm saves the correctness proof of the paper and pencil algorithm to the actual code. Furthermore, these number types provide a valuable tool for rapid prototyping, enable the use of symbolic perturbation schemes [9, 10, 26, 28, 29], allow one to reliably detect geometric degeneracies, and can be used to debug incorrect decisions in geometric floating-point computations. However, you have to pay for the exact decisions in terms of time and space efficiency. Thus, unfortunately, for exact decisions with utmost efficiency you have to learn more about numerical precision and robustness problems in geometric computing [23, 31] and supporting techniques, e.g. [27], that are much less user-friendly.

2 How It Works

Before we discuss the use of exact decision number types based on expression dags and lazy evaluation we briefly recap how they work. These number types record the computation history in expression trees, more precisely in expression dags (i.e., directed acyclic graphs), since different operations can share operands. For example, Fig. 1 shows an expression dag that is built when evaluating the orientation predicate for three points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$ in the plane by computing the sign of the determinant

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} p_x - r_x & p_y - r_y \\ q_x - r_x & q_y - r_y \end{vmatrix}$$

The expression dags allow one to (re)compute an approximation of the value of the expression at any time at any precision. The expression dag they maintain is an exact symbolic representation. However, in contrast to other exact representations you can not easily read off the sign of the represented real number. Therefore, whenever the sign is needed but not known yet, we iteratively compute better and better approximations using

*Otto von Guericke University, Magdeburg, Germany, stschirr@ovgu.de

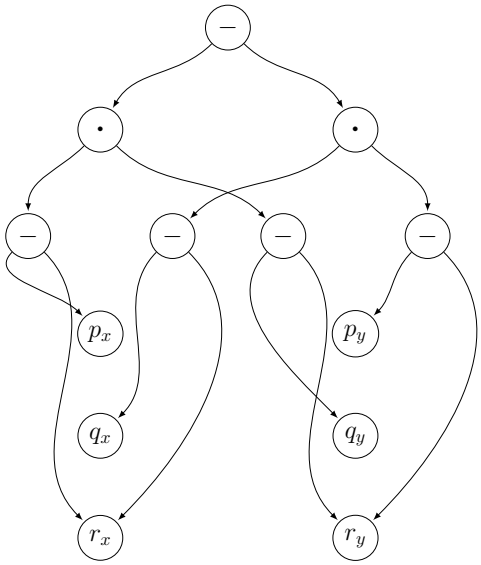

Figure 1: Expression dag for orientation predicate of 2D points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$. Note that the nodes representing p_x, ..., r_y might be (root nodes of) expression dags as well.

the expression dag. If the actual value is zero this is verified with the help of zero separation bounds [24]. If the value is non-zero, increasing the target precision will sooner or later reveal the correct sign. The approximation process stops as soon as we can verify the sign with the help of error bounds and zero separation bounds. Further techniques can be used to enhance efficiency, for example, interval arithmetic can be used as an initial floating-point filter.

Thanks to this lazy approximation strategy, the sign is more quickly computed if the absolute value of the expression is large. Therefore we call this an adaptive precision computation. `CORE::Expr` and `leda::real` were the first to implement such a strategy. A similar strategy without adaptive evaluation was previously used by Benouamer et al. [1] in the context of robust geometric computing. Their number type records the computation history in an expression dag as well. However, whenever an evaluation with double precision is not sufficient to verify the computed sign, the computation is redone with a number type for the rationals based on integers of arbitrary length. $\sqrt{}$-operations are not supported.

Since the first releases of `CORE::Expr` and `leda::real` these number types have been extended to support a larger subset of real algebraic numbers. In a fruitful competition between the groups at NYU and Max Planck Institute improved constructive zero separation bounds have been developed [5, 22] and integrated into the number types. More recently, a configurable expression-dag-based number type called `RealAlgebraic` [19, 20] has been designed and implemented. Like the most recent version of `CORE::Expr` [33] this number type is a C++ class template. For instance, `RealAlgebraic` allows one to exchange the underlying bigfloat arithmetic, to select a floating-point filter, to use different strategies for deferring dag-construction, e.g., by using error-free floating-point transformations or adding tests that check whether the result of a floating-point computation is exact.

3 Not a Floating-Point Number Type

In the following sections, we take a closer look at some sales slogans. The LEDA guide [14] says `leda::real`s

> *can be used like double and together with built-in number types.*

Indeed, thanks to features provided by the C++ programming language, these exact decisions number types can be used like any other number type, at least syntactically. However, they are not always a reasonable substitute for `float`s or `double`s or bigfloats.

For example, floating-point numbers are used in iterative numerical processes. While increasing the precision of the floating-point type usually leads to better results, replacing the floating-point type used in iterative processes by an expression-dag-based exact decisions number types does not make much sense. Let's take a closer look at an example: approximating square roots by Newton's method disguised as the Babylonian method, also known as Heron's method. In order to compute an approximation for $\sqrt{a}$ from a starting value x_0 we use the formula

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}$$

and stop, if the relative difference

$$\frac{|x_{n+1} - x_n|}{|x_{n+1}|}$$

is smaller than some given bound ϵ. The adaptive, exact decisions number types record the overall computation in an expression dag, see Fig. 2.

While the intermediate arithmetic operations basically just extend the dag, the comparison in the termination test triggers a lazy evaluation of x_{n+1} and x_n. If zero separation bounds are recomputed at the beginning of each sign evaluation, this causes a walk over an expression dag, whose size is proportional to the number of previous iteration steps. If zero separation bounds are buffered, we might nevertheless walk over the whole expression dag in order to compute a new improved approximation. Only if the current approximation from a previous lazy evaluation is already sufficiently precise at

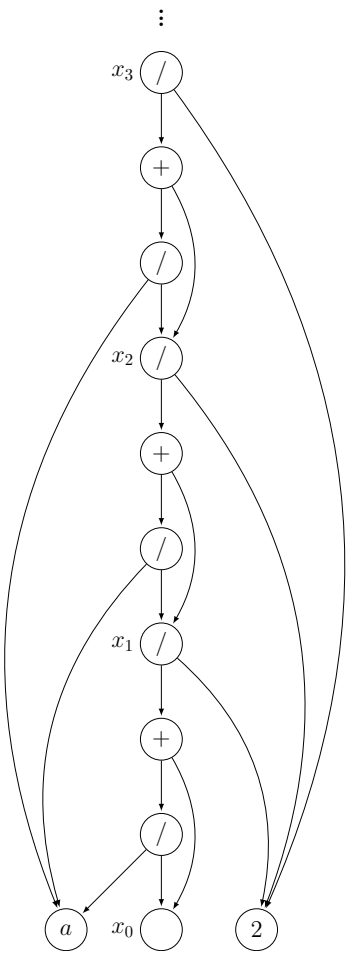

Figure 2: (Lower part of) the expression dag constructed when computing $\sqrt{a}$ by the Babylonian method.

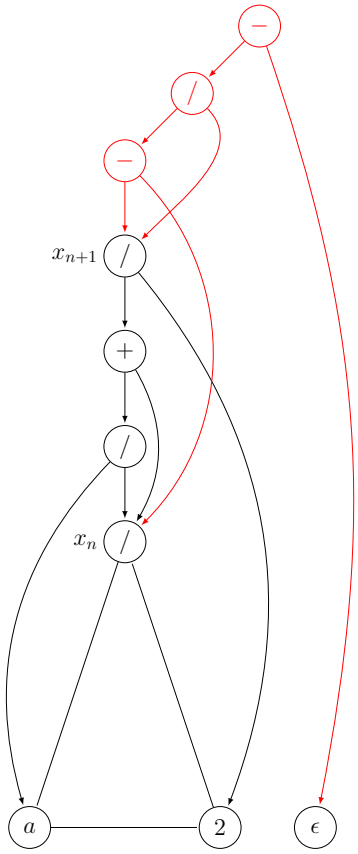

Figure 3: An expression dag corresponding to the termination test of the Babylonian method. Such a test whether the desired tolerance has been reached might trigger a pass over the expression dag representing the whole computation history. The red part of the dag is deallocated after the test.

a node, we do not inspect its sub"trees". Fig. 3 shows an expression dag (sketch) corresponding to a termination test.

If you use such iterative processes in order to compute floating-point approximations in your code, simply substituting the floating-point number type by an adaptive, expression-dag-based exact decisions number type is usually not a good idea. Surprisingly, using `leda::real` in the Babylonian method is as fast as using `leda::rational`, i.e., the adaptive lazy evaluation pays off. However, even for small integral values of a, both are about three orders of magnitude slower than using `double`.

4 Exact vs. Exact Decisions Number Types

The LEDA guide [14] says that `leda::real`s are

> *in general less efficient than integers of arbitrary length and rational numbers.*

On the contrary, in general they are more efficient. Assume we have a computation within the rational numbers. If the sign of the actual value is zero, the adaptive precision computation internal to expression-dag-based exact decisions number types must detect this with the help of a zero separation bound. The number types adaptively increase the precision of the computed approximation until the computed zero separation bound allows for the conclusion that the sign is zero. Although we know that the algebraic degree is at most one, a number of iterations are required, where, in the case of integers, the last one essentially computes exact values. Number types for integers of arbitrary length compute the exact value immediately, saving all the precision increasing iterations of the lazy evaluation except for the last one. Thus they are faster if the actual sign is zero.

If the sign is positive or negative, the situation is different. Sooner or later, the verified approximation error will be smaller than the absolute non-zero approximation value. Sooner or later, this depends on the absolute

value. The larger the absolute value, the sooner the correct sign is detected. Thus, if there are not too many degeneracies and near-degeneracies, adaptive, expression-dag-based exact decisions number types will be faster. For example, for input data generated uniformly at random exact decisions number types with adaptive precision will in general be more efficient, since the lazy evaluation strategy will frequently lead to quick decisions. You can observe this behavior when comparing Cartesian CGAL kernels with adaptive, expression-dag-based exact decisions number types and CGAL kernels with arbitrary precision rational types, where both are applicable.

5 Programming Style

CORE [8] says,

> *it is intuitive, as users can achieve numerically robust algorithms without any change in programming style and prior knowledge of non-robustness issues.*

Most programmers already have some prior knowledge of precision and robustness problems. When they implement a paper and pencil algorithm, they do not simply replace the exact real arithmetic of the Real-RAM from theory by a floating-point number type. They also introduce epsilon-tolerances in their code. Then, however, simply replacing the floating-point number type by an exact decisions number type does not rescue the theoretical correctness proof anymore. First, you have to get rid of the tolerances again.

In the context of geometric computing with real algebraic numbers arising from polynomial system solving, users often use zero testing with potential solution vectors to identify the actual solutions. They plug arbitrary combinations of solution coordinates into multivariate polynomials and check which of them evaluate to zero. Since the actual solution vectors are among the test candidates, we have some self-made degeneracies here. You better use more geometry! For example, assume you want to compute the intersection points of two circles. Using elimination, one can compute the x- and y-coordinates as expressions involving radicals. In order to identify the correct combinations among the four possible pairs, one is tempted to plug pairs of coordinates into the circle equations and to check which pairs yield zero. However, a comparison of the coordinates of the centers of the circles suffices to detect the valid combinations, see Fig. 4.

There is another caveat regarding programming style. With software number types, programmers can not rely anymore on compiler optimization. For example, users should take care of common subexpression elimination themselves. There have been efforts to let `leda::reals` search for common subexpressions automatically at run time [25]. However, since the rare savings did not compensate the always present additional cost, this strategy never made it into a release version.

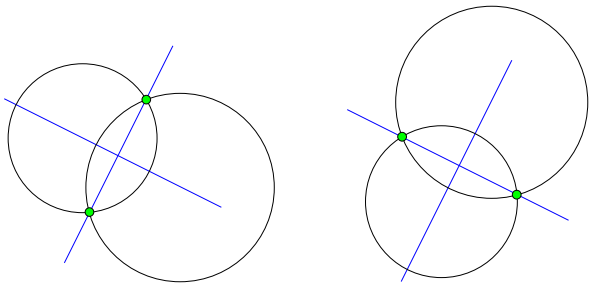

Figure 4: Avoid zero testing using geometric reasoning: By comparing the center coordinates you can identify the correct combination of solution coordinates.

6 Algorithm Design Style

While you might have to change your programming style regarding numerical issues, you do not have to change your paper and pencil algorithm design style. You may assume that all decisions involving numerical computations are correct. However, you might have to think about handling degeneracies, since they are correctly detected if you use exact decisions number types. If you are an epsilon-tweaker you have to do that anyway, but with exact decisions you have to handle the real ones only. CORE [8] also correctly says,

> *it is intuitive, as users can achieve numerically robust algorithms without any change to the algorithms themselves.*

7 Not A Number Type for Rational Numbers?

The LEDA guide [14] recommends to use `leda::reals`

> *to do exact computations with square roots and k-th roots and to consider using bigfloats otherwise.*

As discussed above, in general the adaptive, exact decisions number types based on expression dags are not less efficient than arbitrary precision integers or rationals. The same holds for bigfloats: Use of bigfloats for exact geometric computing makes sense only if the computation is division-free, e.g. if we use homogeneous coordinates. Furthermore, for exact geometric computing, bigfloats must adjust the precision (mantissa length) as necessary in order to get exact results and hence exact decisions. Then, however, bigfloats basically behave like scaled arbitrary precision integers. Thus, as discussed above, the level of degeneracies is crucial for choosing

the number type, not the presence of square root operations. If we use `leda::bigfloat` in the exact mode in our code for the Babylonian method, the resulting code is not faster than the code based on `leda::real`.

The misleading recommendation might have its roots in a corresponding comparison of LEDA's geometry kernels. The well-engineered rat-kernel of LEDA, which uses Cartesian coordinates for floating-point filtering and homogeneous integer computations otherwise, is in general significantly faster than the so-called real-kernel of LEDA, which has been added later on and uses Cartesian coordinates of type `leda::real` only.

8 Efficiency

The LEDA guide [14] says `leda::real`s

> *are about 10-80 times slower than double.*

Frankly speaking, this is sometimes quite euphemistic. For degeneracies where a zero separation bound must be used to terminate adaptive approximation, the factor can be much worse and for cascaded computations, it can be arbitrarily bad. Even for simple geometric calculations with low arithmetic demand recording the computation history in expression dags causes a big slow-down. Even if rough approximations suffice to compute sign correctly, there is a significant slow-down because of the cost of dynamic memory allocation.

9 User-friendly Alternatives

For rational geometric computations, both CGAL [7] and LEDA [17] provide exact decisions geometry kernels that are much more efficient than a CGAL kernel parameterized with an adaptive exact decisions number type based on expression dags. Both CGAL's exact predicates exact construction kernel and LEDA's rational kernel support cascaded rational computations. For geometric computations involving $\sqrt{}$-operations, CGAL provides yet another exact decisions kernel, whereas LEDA offers its real-kernel for this context. The latter is usually faster than CGAL's kernel parameterized with `leda::real`.

These kernels are user-friendly as well, since they wrap all the numerical issues regarding exact decisions in geometric predicates and algorithms, whereas exact decisions number types do this on the level of arithmetic. However, for these exact decisions geometry kernels you have to stick with the operations they offer. It is not that easy to add similarly efficient auxiliary functionality. Here, adaptive, exact decisions number type based on expression dags can be very useful. More recently, more evolved geometry kernels have been added to CGAL in order to better support non-linear computational geometry [2]. These kernels handle real algebraic numbers that are given as roots of univariate polynomials and bivariate polynomial systems.

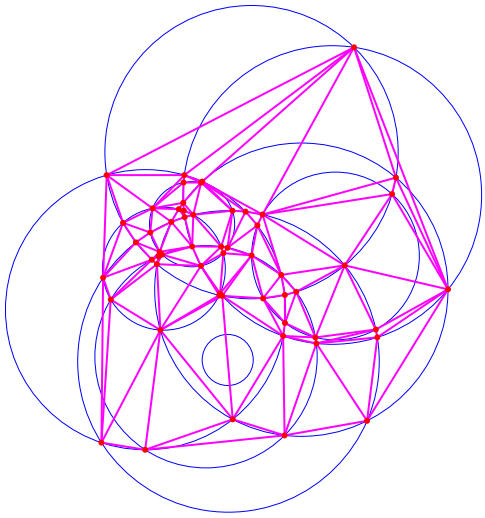

Figure 5: Delaunay triangulation of the intersection points of a set of circles.

While the adaptive, exact decisions number types considered here record the computation history on the level of arithmetic operations, it is also possible to record the computation history on the level of geometric constructions. This has been done in LOOK [11] and similar approaches [21] are used in CGAL. This saves space and can be a source of efficiency. Once again, however, this approach is limited to geometric operations predefined in these kernels. Recording the computation history on the level of arithmetic operations is the more general approach.

10 Cascading

Recording the computation history allows for adaptive approximation and makes the adaptive, expression-dag-based exact decisions number types applicable to cascaded geometric computations where the output of some previous calculations is used as input for later ones. For example, using adaptive, exact decisions number type based on expression dags you can easily compute the Voronoi Diagram of the intersection points of the circumcircles of the Delaunay triangles of a set of points in the plane. Fig. 5 shows a similar example.

Although cascaded computations is one of the strengths of these number types, it also shows its performance limitations. You have to perform some geometric rounding after some stages.

11 Conclusions

Adaptive expression-dag-based exact decisions number types are a very useful tool for rapid-prototyping when implementing the exact (decisions) geometric computation paradigm, for adding additional primitives to existing exact decisions geometry kernels, for debugging geometric programs suffering from numerical precision issues, and for student implementation projects when teaching computational geometry. Using such a number type you get the Real-RAM behavior restricted to (a subset of) algebraic numbers. They are inferior to exact decisions kernels implementing advanced techniques or recording computation history on a higher level, in terms of efficiency, but in general superior to kernels parameterized with rational number types. However, due to different design philosophies, it is not straightforward to turn well-engineered robust geometric programs based on tolerances into exact decisions programs simply by replacing floating-point computations by computations with adaptive, expression-dag-based exact decisions number types.

The author would like to thank Martin Held, Stefan Huber, Kurt Mehlhorn, Marc Mörig, and Chee Yap for interesting discussions on the topic.

References

[1] M. O. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation using lazy rational arithmetic: a detailed implementation. In *Solid Modeling and Applications*, pages 115–126, 1993.

[2] E. Berberich, M. Hemmer, M. Kerber, S. Lazard, L. Peñaranda, and M. Teillaud. Algebraic kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2014.

[3] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Pion, and S. Schirra. 2D and 3D geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 edition, 2014.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Symposium on Computational Geometry*, pages 341–350, 1999.

[5] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. *Algorithmica*, 55(1):14–28, 2009.

[6] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in leda. In *Symposium on Computational Geometry*, pages C18–C19, 1995.

[7] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[8] Core pages. http://cs.nyu.edu/exact/core_pages/features.html.

[9] 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.

[10] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1/2):219–242, 1997.

[11] S. Funke and K. Mehlhorn. LOOK: A lazy object-oriented kernel design for geometric computation. *Comput. Geom.*, 22(1-3):99–118, 2002.

[12] V. Karamcheti, C. Li, I. Pechtchanski, and C.-K. Yap. A core library for robust numeric and geometric computation. In *Symposium on Computational Geometry*, pages 351–359, 1999.

[13] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C.-K. Yap. Classroom examples of robustness problems in geometric computations. *Comput. Geom.*, 40(1):61–78, 2008.

[14] LEDA guide. http://www.algorithmic-solutions.info/-leda_guide/number_types/real.html, 2006

[15] C. Li, S. Pion, and C.-K. Yap. Recent progress in exact geometric computation. *J. Log. Algebr. Program.*, 64(1):85–111, 2005.

[16] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *IFIP Congress (1)*, pages 223–231, 1994.

[17] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[18] K. Mehlhorn and S. Schirra. Exact computation with leda_real - theory and geometric applications. In *Symbolic Algebraic Methods and Verification Methods*, pages 163–172. 2001.

[19] M. Mörig. *Algorithm Engineering for Expression Dag Based Number Types*. PhD thesis, Otto von Guericke University Magdeburg, Magdeburg, Germany, in preparation.

[20] M. Mörig, I. Rössling, and S. Schirra. On design and implementation of a generic number type for real algebraic number computations based on expression dags. *Mathematics in Computer Science*, 4(4):539–556, 2010.

[21] S. Pion and A. Fabri. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.*, 76(4):307–323, 2011.

[22] S. Pion and C.-K. Yap. Constructive root bound for k-ary rational input numbers. In *Symposium on Computational Geometry*, pages 256–263, 2003.

[23] S. Schirra. Robustness and precision issues in geometric computation. In J. R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*, pages 597–632. Elsevier, New York, 2000.

[24] S. Schirra. Much ado about zero. In *Efficient Algorithms*, pages 408–421, 2009.

[25] S. Schmitt. Common subexpression search in leda_reals. Report ECG-TR-243105-01, Effective Computational Geometry for Curves and Surfaces, Sophia Antipolis, France, 2003.

[26] R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete & Computational Geometry*, 19(1):1–17, 1998.

[27] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–368, 1997.

[28] C.-K. Yap. Geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40(1):2–18, 1990.

[29] C.-K. Yap. Symbolic treatment of geometric degeneration. *J. Symb. Comput.*, 10(3/4):349–370, 1990.

[30] C.-K. Yap. Towards exact geometric computation. *Comput. Geom.*, 7:3–23, 1997.

[31] C.-K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. Revised and expanded from 1997 version.

[32] C.-K. Yap. Theory of real computation according to EGC. In *Reliable Implementation of Real Number Algorithms*, pages 193–237, 2008.

[33] J. Yu, C. Yap, Z. Du, S. Pion, and H. Brönnimann. The design of Core 2: A library for exact numeric computation in geometry and algebra. In *ICMS*, pages 121–141, 2010.