Exact Computation with leda_real – Theory and Geometric Applications

Kurt Mehlhorn and Stefan Schirra

Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

1 Introduction

The number type `leda_real` provides exact computation for a subset of real algebraic numbers: Every integer is a `leda_real`, and `leda_real`s are closed under the basic arithmetic operations $+, -, *, /$ and k-th root operations. `leda_real`s guarantee correct results in all comparison operations. The number type is available as part of the LEDA C++ software library of efficient data types and algorithms [12, 15]. `leda_real`s provide user-friendly exact computation. All the internals are hidden to the user. A user can use `leda_real`s just like any built-in number type.

The number type is successfully used to solve precision and robustness problems in geometric computing [2, 23]. It is particularly advantageous when used in combination with the computational geometry algorithms library CGAL.

2 Theory and practice in geometric computing

For the sake of better understanding underlying mathematical principles, theory of algorithm design often makes simplifying assumptions. In practice of geometric computing such simplifying assumptions cause notorious problems, if they don't hold, see for example [7, 9, 13, 21, 26]. Besides the assumption of so-called *general position* which excludes special cases in the input for an algorithm like three points lying on a line, the most puzzling assumption the implementor of a geometric algorithm is left to deal with is the assumption of *exact arithmetic* over the reals. The ubiquitous model of computation in computational geometry is the so-called *real RAM*. A real RAM can hold a single real number in each of its storage locations and perform basic arithmetic operations $+, -, *, /$, comparison between two real numbers, and $\sqrt[k]{\ }$ and exp, log, if needed [18]. All these operations are assumed to give the correct result at unit cost.

With standard floating-point arithmetic, the default substitution for the real numbers in scientific computing, this assumption does not hold. Due to rounding and cancellation errors, the operations above might give incorrect results. Implementations simply using floating-point arithmetic as a substitute for exact real arithmetic typically "work" most of the time, but sometimes produce catastrophic errors, i.e., they crash, compute useless output, or even loop forever, although the implemented algorithms are theoretically correct. Such behavior can be observed also with currently available CAD-systems.

The root cause of such errors is that rounding errors in the computation lead to incorrect values resulting in incorrect and contradictory decisions. For example, a program using imprecise floating-point arithmetic might detect two different points of intersection between two non-identical straight lines. Of course, such a situation can not arise in real geometry, so the algorithm is not designed to handle such situations and it is not surprising that the program crashes.

Geometric computing goes beyond numerical computing, since it also has a discrete combinatorial component. Whereas in numerical computing one can often argue that the numerical results computed with floats are correct for some (small) perturbation of the numerical values in the input, such reasoning is intrinsically much harder in geometric computing. Computing the orientation of three points p, q, and r in the plane corresponds to computing the sign of a 3×3 determinant. Even if the orientation computed using floating-point arithmetic is not correct the three points can always be perturbed a little bit to points p', q', and r' such that the computed orientation is the orientation of p', q', and r'. However, in a geometric program points p, q, and r are involved in many orientation computations, and although for each single orientation computation there is always a correcting perturbation of the points, it is not guaranteed at all that there is a valid perturbation for all points making all computed orientation computations correct. Now, deciding whether there is a set of points realizing given orientation information is a hard problem. In fact, it is at least as hard as deciding the existential theory over the reals [16].

There are two obvious approaches to close the gap between theory and practice of geometric computing with respect to precision problems. Either change theory or change practice, i.e., take imprecision into account when designing a geometric algorithm or compute "exactly". The latter approach is known as the "exact geometric computation paradigm" [27]. It assures that all decisions, i.e., all comparison operations, made in a program are correct and thereby assures that a program behaves as its theoretical counterpart. `leda_real`s are an extremely useful tool to implement an algorithm according to the exact geometric computation paradigm. Redesigning algorithms such that imprecision is taken into account, is considered not very attractive as the many algorithms developed in theory so far under the real RAM model would get lost.

3 Verified computation with `leda_reals`

In this section we describe how the `leda_real`s assure correct comparisons. The `leda_real`s record computation history in an expression dag (i.e., a directed acyclic graph) in order to allow for re-evaluation. The leaves of the dag are integers, and each internal node of the dag is labeled with a unary or binary operation and points to the operands from which the subexpression was computed. Fig. 1 shows an example. Furthermore, during construction of the expression dag, the `leda_real`s already compute rough approximations. Here, one can maintain a value and an error bound or, alternatively, use interval arithmetic.

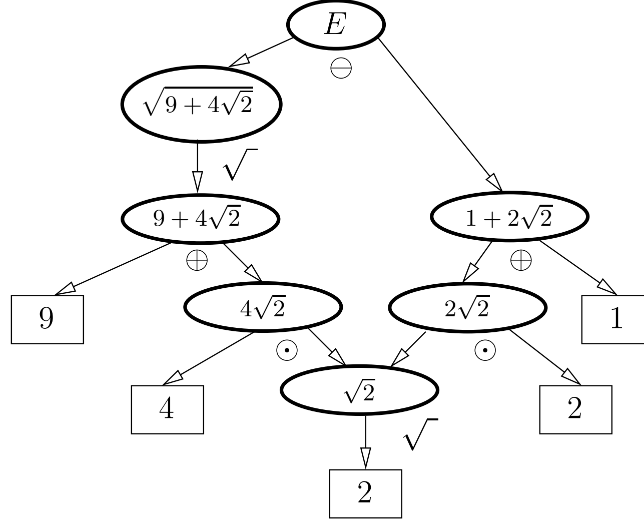

Fig. 1. Recorded computation history for the code fragment

```
leda_real root_two = sqrt(leda_real (2));
leda_real E = sqrt(9 + 4 * root_two) - (1 + 2 * root_two);
```

The actual work, however, is done in the comparison operations which are reduced to sign computations. Let E be the expression whose sign we are interested in, and let ξ denote the value of the expression E. If, in a sign computation for E, the current approximation $\widetilde{\xi}$ is not sufficient to verify the sign, the expression dag is used to re-compute better approximations. Approximations are computed as `leda_bigfloat`s, a software floating-point number type that allows you to choose the mantissa length. We compute approximations $\widetilde{\xi}$ of ξ with increasing quality until either the sign of ξ is equal to the sign of $\widetilde{\xi}$ or until we can conclude that ξ is zero. We use separation-bounds to check for zero. A separation bound for an expression E is a number sep_E such that

$$|\xi| < sep_E \Rightarrow \xi = 0.$$

In the next section we prove such a separation bound for expressions. With each approximation $\widetilde{\xi}$ we compute an error bound $\Delta_{\text{error}} \geq |\xi - \widetilde{\xi}|$. If

$$|\xi - \widetilde{\xi}| \leq \Delta_{\text{error}} < |\widetilde{\xi}|$$

we have $\text{sign}(\tilde{\xi}) = \text{sign}(\xi)$. If, on the other hand,

$$|\widetilde{\xi}| + \Delta_{\text{error}} < sep_E$$

we know that ξ must be zero. Since the quality of the approximations increases and hence Δ_{error} decreases, we eventually get the correct sign. The sign computation is adaptive. The running time depends on how small the value $|\xi|$ of E is.

leda_reals have been implemented as a C++-class by Christoph Burnikel [4]. Thanks to operator overloading, they can be used with the natural syntax, just like any other (built-in) number type in C++.

4 Theoretical foundations

In this section we prove a separation bound that generalizes the bound proved in [3] and is more general than what is needed for the current implementation of leda_reals. An algebraic number is called *algebraic integer*, if it is a root of a monic[1] polynomial with integral coefficients. It is well known that algebraic integers form a ring containing the (rational) integers.

The following lemma is a straightforward generalization of Theorem 1 in [3]. It allows for algebraic integer instead of (rational) integer as basic operands.

Lemma 1. *Let E be an expression with operations $+, -, *, \sqrt[k]{\ }$ for integral k where the operands are algebraic integers. Let ξ be the value of E and $\deg(\xi)$ denote the algebraic degree of ξ. Let $u(E)$ be defined inductively by the structure of E by the rules shown in the table below.*

E	$u(E)$
algebraic integer γ	U_γ
$E_1 \pm E_2$	$u(E_1) + u(E_2)$
$E_1 \cdot E_2$	$u(E_1) \cdot u(E_2)$
$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$

Here U_γ is chosen such that $|\varrho| \le U_\gamma$ for all roots ϱ of monic $P \in \mathbb{Z}[X]$ with $P(\gamma) = 0$. We have

$$\left(u(E)^{\deg(\xi)-1}\right)^{-1} \le |\xi| \le u(E).$$

Algebraic integers are closed under the operations $+, -, *$ and $\sqrt[k]{\ }$. More generally, if we have a monic polynomial whose coefficients are algebraic integers, then the roots of this polynomial are algebraic integers again. We briefly prove that there is $P_E(X) = \prod(X - \gamma_\ell) \in \mathbb{Z}[X]$ with $P_E(\xi) = 0$, such that for all roots γ_ℓ of $P_E(X)$:

$$|\gamma_\ell| \le u(E)$$

By the theorem of elementary symmetric functions, any polynomial which is symmetric in $z_1, \ldots, z_n$, can be written as a polynomial in the z_is' elementary symmetric functions

$$\begin{aligned}
\sigma_1 &= z_1 + z_2 + \cdots + z_n \\
\sigma_2 &= z_1 z_2 + z_1 z_3 + \cdots + z_{n-1} z_n \\
&\vdots \\
\sigma_n &= z_1 z_2 \cdots z_n
\end{aligned}$$

[1] a polynomial is called monic, if its leading coefficient is 1.

We prove the claim by structural induction. In the base case, the claim holds by definition of an algebraic integer and $u(E)$. Note the special case, that E is an integer N. In this case, $P(X) = X - N \in \mathbb{Z}[X]$ and $u(E) = |N|$.

For the induction step we distinguish unary and binary operations.

$E_1 \overset{\pm}{*} E_2$:
By induction hypothesis we have

$$P_{E_1}(X) = \prod_{i=1}^{n} (X - \alpha_i) = \sum_{s=0}^{n} a_s X^s \in \mathbb{Z}[X],$$

$$P_{E_2}(X) = \prod_{j=1}^{m} (X - \beta_j) = \sum_{t=0}^{m} b_t X^t \in \mathbb{Z}[X].$$

By the theorem of elementary symmetric functions, the polynomial

$$P_E(X) := \prod_{i=1}^{n} \prod_{j=1}^{m} (X - (\alpha_i \overset{\pm}{*} \beta_j))$$

has integral coefficients. Furthermore, we have $|\alpha_i \overset{\pm}{*} \beta_j| \leq u(E_1) \overset{+}{*} u(E_2)$ by induction hypothesis and definition of u.

$E = \sqrt[k]{E_1}$:
By induction hypothesis, we have

$$P_{E_1}(X) = \prod_{i=1}^{n} (X - \alpha_i) = \sum_{s=0}^{n} a_s X^s \in \mathbb{Z}[X].$$

Then

$$P_E(X) = P_{E_1}(X^k) = \prod_{j=1}^{k} \prod_{i=1}^{n} (X - \zeta_{(k)}^j \sqrt[k]{\alpha_i}) \in \mathbb{Z}[X]$$

where $\zeta_{(k)}$ is a primitive k-th root of unity. We have $|\zeta_{(k)}^j \sqrt[k]{\alpha_i}| = |\sqrt[k]{\alpha_i}| \leq |\sqrt[k]{u(E_1)}|$ by induction hypothesis. Thus, $|\zeta_{(k)}^j \sqrt[k]{\alpha_i}| \leq u(E)$ by definition of u.

To complete the proof of Lemma 1, let $\xi \neq 0$ and $M_E(X)$ be the minimal polynomial of ξ.
Since $M_E(X)$ divides $P_E(X) = \prod (X - \gamma_\ell)$, we have

$$M_E(X) = \prod_{t=1}^{\deg(\xi)} (X - \gamma_{\ell_t}) \in \mathbb{Z}[X].$$

W.l.o.g. $\xi = \gamma_{\ell_1}$. Since all coefficients are integral, we have $\left|\prod_{t=1}^{\deg(\xi)} \gamma_{\ell_t}\right| \geq 1$. Hence $\left(\prod_{t=2}^{\deg(\xi)} |\gamma_{\ell_t}|\right)^{-1} \leq |\xi|$ and $\left(u(E)^{\deg(\xi)-1}\right)^{-1} \leq |\xi|$. $\square$

Lemma 2. *Let E be an expression with operations $+, -, *, /, \sqrt[k]{\ }$ for integral k where the operands are roots of univariate polynomials with integral coefficients. Let ξ be the value of E and $\deg(\xi)$ denote the algebraic degree of ξ. Let $u(E)$ and $l(E)$ be defined inductively by the structure of E by the rules shown in the table below. Let $K(E)$ be the product of the indices of the radical operations in E. Furthermore, let $D(E)$ be the product of the degree of the polynomials defining the operands.*

	$u(E)$	$l(E)$
algebraic number α	U_α	L_α
$E_1 \pm E_2$	$u(E_1)\cdot l(E_2) + l(E_1)\cdot u(E_2)$	$l(E_1)\cdot l(E_2)$
$E_1 \cdot E_2$	$u(E_1)\cdot u(E_2)$	$l(E_1)\cdot l(E_2)$
E_1 / E_2	$u(E_1)\cdot l(E_2)$	$l(E_1)\cdot u(E_2)$
$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$	$\sqrt[k]{l(E_1)}$

Here $P = \sum_{i=0}^{d} a_i X^i \in \mathbb{Z}[X]$ with $P(\alpha) = 0$, not necessarily monic, $L_\alpha = a_d$ and U_α such that $|\varrho| \le U_\alpha / a_d$ for all roots ϱ of P. Then we have

$$\left(l(E)u(E)^{K(E)^2 D(E)^2 - 1}\right)^{-1} \le |\xi| \le u(E)l(E)^{K(E)^2 D(E)^2 - 1}$$

Lemma 2 follows by Lemma 1, if we postpone division operations, i.e. replace an expression by a quotient of two division-free expressions whose values are algebraic integers. The size of the numerator is bounded by u, while l bounds the size of the denominator of this quotient. Note that if α is a root of $P = \sum_{i=0}^{d} a_i X^i \in \mathbb{Z}[X]$, then $\gamma = a_d\alpha$ is an algebraic integer. γ is a root of $a_d^{d-1}P(X/a_d)$. So we can replace α by $\frac{\gamma}{a_d}$.

There is no need to compute the l- and u-values exactly. It suffices to compute upper bounds on these values in order to derive separation bounds. The current implementation of `leda_real`s computes such upper bounds logarithmically. We plan to extend the implementation of `leda_real`s to include algebraic integer operands as discussed above.

5 Geometric applications

Exact computation with radicals is frequently required in geometric computations involving distances. Fig. 2 shows a Voronoi diagram of line segments, computed with an algorithm that uses `leda_real`s [23]. In contrast to a previous implementation, the new program never crashes due to rounding errors.

Parametric search [14] is a very nice technique to derive algorithms for solving certain optimization problems from algorithms for related decision problems [20]. Parametric search can be applied if the problem can be phrased as an optimization problem parameterized by a real-valued parameter r where the goal is to compute a unique optimal value r^* for a given input parameterized by r. The related decision problem is to decide whether a given concrete r_0 is at least as large as r^*. Parametric search simulates the decision problem for the unknown

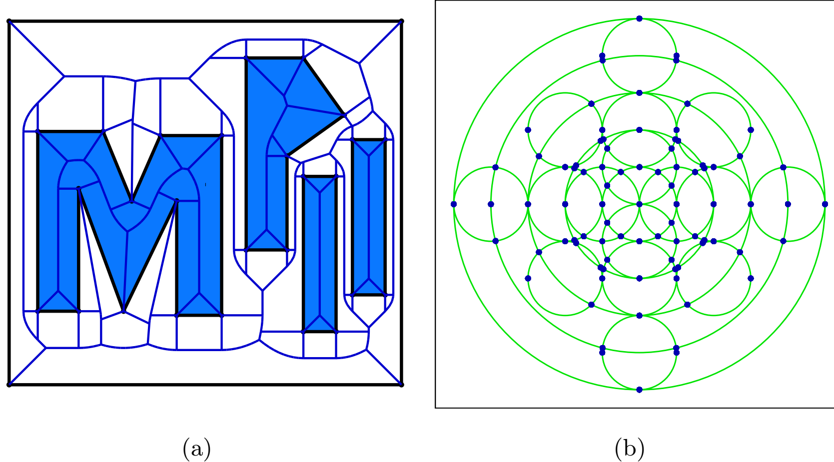
(a) (b)

Fig. 2. Two geometric problems involving square roots: (a) Voronoi diagram of line segments, (b) arrangement of circles [8].The points shown are intersection points and points of vertical tangency.

r^* and computes r^* during the simulation. Applying the parametric search technique involves solving the decision problem for zeros of functions in r that arise during the parameterized simulation. In almost all geometric applications, these functions are polynomials in r. Schwerdt et al. [24] provide, to the best of our knowledge, the first actual implementation of parametric search for the problem of computing the point in time where a set of points moving with constant velocity has minimum diameter. It uses `leda_real`s to exactly solve decision problems for roots of polynomials of degree 2.

In combination with the CGAL framework [5, 6, 17] for geometric computation, the number type `leda_real` is particularly fruitful. The geometry kernels of CGAL, a computational geometry algorithms library, are ready to use `leda_real`s. CGAL provides kernels with Cartesian coordinate representation as well as a kernel based on homogeneous coordinates. Both are parameterized with the number type used for coordinates and arithmetic. The use of an exact number type yields an exact kernel. Using `leda_real`s with the CGAL kernels yields easy-to-write, correct and still reasonably efficient geometric programs. Fig. 2 (b) shows an arrangement of circles computed with the CGAL arrangement algorithm using `leda_real`s as number type.

Fig. 3 (a) shows a basic geometric problem that is challenging in terms of precision because of degenerate configurations. The task is to compute the extreme points among the intersection points of a set of line segments. A point is called *extreme* with respect to a set of points P, if it is a corner point of the convex hull of P. Extreme point computation involves orientation computation for point triples. In our example, the difficulty in terms of precision is caused by the fact that by construction there are many collinear points. Using CGAL

with pure floating point computation often leads to reporting incorrect extreme points. Note that in these cases there is no perturbation of the input such that the reported set of extreme points is correct for the perturbed input, unless you give up straightness of the input segments. If you want to compute a Delaunay triangulation of the intersection points, see Fig. 3 (b), the situation is even worse: Using `double` as number type, the CGAL algorithm often complains about violation of invariants and stops. Of course, using `leda_real`s, we get correct results for both problems.

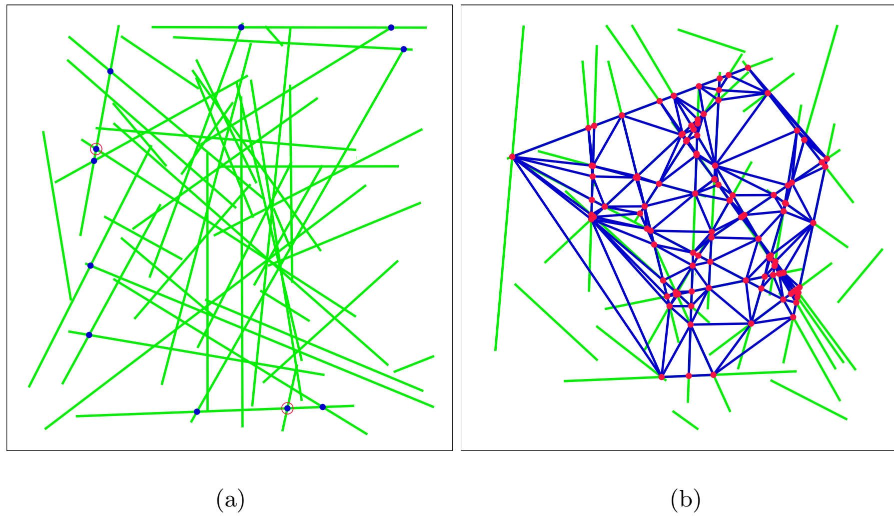

Fig. 3. Extreme point computation (a) and Delaunay triangulation (b) for intersection points of random line segments with `double` coordinates. Points incorrectly classified as extreme by floating-point computation with `double` precision are encircled.

`leda_real`s are reasonably efficient in geometric computing whenever the expressions defining the algebraic numbers do have bounded size. In these cases the use of `leda_real`s slows down the computation by a constant factor. However, exact computation has its costs. In our experience, the slow down factor with respect to pure floating-point computation is between 5 and 20. For challenging problem instances it might be even more. However, here we compare apples and oranges. While the float based algorithm produces catastrophic errors sometimes, the `leda_real` based algorithm always gives a correct result. The comparison of running times is with respect to those cases only, where we get a result with both number types. If we would compare the time it takes a program to compute the correct result, a `leda_real`-based program would be "much faster", because it computes the correct result in finite time whereas the float-based program never reaches this goal. For further discussion on running times with `leda_real`s we refer the interested reader to [2] and [22].

leda_reals are not a panacea. If there are algebraic numbers involved whose expressions do not have bounded size, the use of leda_reals might not lead to satisfactory solutions anymore.

6 Conclusions

leda_reals offer a convenient and (reasonably) efficient way of computing with expressions involving roots. For geometric computing, leda_reals are especially useful in combination with the CGAL kernels.

A number type with similar functionality is available as part of the CORE-package developed at NYU [10]. The work presented in this paper is furthermore related to algebra systems and algebra toolkits like those presented in [11] and [19].

References

1. H. Brönniman, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. Research Report MPI-I-98-1-030, Max-Planck-Insitut für Informatik, 1998.
2. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Companion page to 'Efficient exact geometric computation made easy'. http:/www.mpi-sb.mpg.de/~stschirr/exact/made_easy.
3. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica* 27:87-99, 2000.
4. C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Research Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996. A more recent documentation of the implementation is available at http://www.mpi-sb.mpg.de/~burnikel/reports/real.ps.gz.
5. CGAL. http://www.cs.uu.nl/CGAL/.
6. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Research Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, 1998.
7. S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81 – 128. Information Geometers Ltd., 1993.
8. I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. Manuscript. Tel-Aviv University, Israel, 2000.
9. C. Hoffmann. The problem of accuracy and robustness in geometric computation. *IEEE Computer*, pages 31–41, March 1989.
10. V. Karamcheti, C. Li, I. Pechtanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 351–359, 1999.
11. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360–369, 1999. See also http://www.cs.unc.edu/~geom/MAPC.
12. LEDA. http://www.mpi-sb.mpg.de/LEDA/leda.html.

13. K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.
14. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM* 30:852–865. 1983.
15. K. Mehlhorn and S. Näher. *LEDA – A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, England, 1999.
16. M. E. Mnev. The Universality theorems on the classification problem of configuration varieties and convex polytopes varieties. In *Topology and Geometry - Rohlin Seminar*, Lecture Notes Math. 1346, Springer-Verlag 1989, pages 527–544.
17. M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 53–58. Springer LNCS 1148, 1996.
18. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction* Springer-Verlag New York, 1985.
19. A. Rege. *APU User Manual – Version 2.0*, 1996. `http://www.cs.berkeley.edu/~rege/apu/apu.html`.
20. J. Salowe. Parametric search. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 683–698, CRC Press, 1997.
21. S. Schirra. Robustness and Precision Issues in Geometric Computation. Handbook of Computational Geometry, Elsevier, 2000, pages 597-632.
22. S. Schirra. A case study on the cost of geometric computing. In *Algorithm Engineering and Experimentation* (ALENEX'99), LNCS 1619, pages 156–176, Springer Verlag, 1999.
23. M. Seel. LEP: Abstract Voronoi Diagrams. `http://www.mpi-sb.mpg.de/LEDA/friends/avd.html`.
24. J. Schwerdt, M. Smid, S. Schirra. Computing the Minimum Diameter for Moving Points: An Exact Implementation using Parametric Search. In *Proc. of the 13th ACM Symp. on Computational Geometry* 1997, pages 466-468.
25. C. K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7:3–23, 1997.
26. C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 653–668, CRC Press, 1997.
27. C. K. Yap and T. Dubé. The exact computation paradigm. In D. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, 1995. 2nd edition.