Recent Progress in Exact Geometric Computation *

Chen Li   Chee Yap
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012, USA.
Email: {chenli, yap}@cs.nyu.edu

June 14, 2001

Abstract

Computational geometry has produced an impressive wealth of efficient algorithms. The robust implementation of these algorithms remains a major issue. Among the many proposed approaches for solving numerical non-robustness, Exact Geometric Computation (EGC) is one of the most promising. This survey describes recent progress in EGC research. We specifically focus on the problems of constructive root bounds, precision-driven computation and numerical filters.

1 Introduction

Numerical non-robustness represents a major challenge in the implementation of geometric algorithms. By its nature, geometric computation has two components, a numerical part and a combinatorial part. Numerical computation is involved in both the construction of new geometric objects and in the evaluation of geometric predicates. An example of the former is computing intersection points and an example of the latter is deciding whether a point is on a hyperplane. Geometric predicates are especially critical, as they determine the combinatorial relations among objects. Incorrect evaluation of such predicates can lead to inconsistencies. In general, computational geometry algorithms are designed under a Real RAM model of computation where all numerical computations are exact. As machine arithmetic is widely used as substitute for this assumed exact arithmetic, numerical errors are inevitable in implementation. Today, machine arithmetic has converged to the IEEE standard [82, 28]. But more generally, machine arithmetic is an example of fixed-precision arithmetic. Although numerical errors can sometimes be tolerated and interpreted as small perturbations in inputs, serious problems arise when such errors lead to invalid combinatorial structures or inconsistent state during a program execution.

*This paper is based on a talk presented at the DIMACS Workshop on Algorithmic and Quantitative Aspects of Real Algebraic Geometry in Mathematics and Computer Science, March 12 – 16, 2001. The work is supported by NSF/ITR Grant #CCR-0082056.

The numerical non-robustness problem has received much attention in the computational geometry community in the last 15 years ([22, 38, 81, 49, 16, 23]). The next section will briefly review some approaches to non-robustness, we refer the reader to the current surveys ([86, 68, 64]) for more details. In [86], robustness literature was classified along two lines: the papers that aim to make fixed-precision algorithms computation robust, and those that aim to make the exact computation approach efficient. Call these the **inexact** and **exact** approaches, respectively, and the corresponding algorithms **Type I** and **Type II** algorithms. Our review below will classify the literature into those that try to make the arithmetic more robust, and those that achieve robustness by ensuring certain geometric (which subsumes topological) properties of problem at hand. Call these the **arithmetic** and the **geometric approaches**, respectively, and the corresponding algorithms **Type A** and **Type B** algorithms. The I-versus-II taxonomy is orthogonal to the A-versus-B taxonomy. For instance, both Type I and Type II algorithms can benefit from Type A development: an example of this is the use of machine arithmetic with scalar product primitive, as proposed by Ottmann et al [61]. Similarly, the introduction of low-degree predicates [50] in the development of Type II algorithms can be useful for Type I as well as Type II algorithms. On the other hand, there is a strong correlation between Type I algorithms and Type B algorithms: if an algorithm computes in the fixed-precision model, then it is very likely that it has to investigate the geometric properties of the computational problem.

Paper Outline. In Section 2, we review the arithmetic and geometric approaches to non-robustness. Then we focus on some issues in Exact Geometric Computation (EGC). In Section 3, we discuss in detail the constructive root bound problem which is a key problem in EGC. We treat the techniques of precision-driven computation and numerical filtering in Sections 4 and 5, respectively. We conclude in Section 6

2 Literature Review

We review some robustness techniques developed under the Arithmetic and Geometric Approaches. We also single out the Exact Geometric Computation approach which is the focus of this paper.

2.1 Arithmetic Approaches

This is a natural first place to look for a solution since we know that the root cause of numerical non-robustness is errors from approximate arithmetic. The "naive" arithmetic solution says that we just have to compute exactly, without any errors. This requires the use of multi-precision (i.e., unbounded precision) arithmetic. Such arithmetic is implemented in software libraries called "big number packages" (big integers, big rationals, big floats, big complex, etc). For a survey of multi-precision number packages, see [89]. All big number packages support the four basic arithmetic operations $(+, -, \times, \div)$. Within the domain of rational numbers, it is clear that these operations can be performed without errors.

However, if we need irrational numbers (for example when we take square roots), then simple big number packages will still be insufficient. The usual understanding of "number representation" is that it is some form of positional number system (basically, strings of digits to some base). In this sense, the algebraic number $\sqrt{2}$ cannot be represented exactly. However, it is well-known (e.g., in the computer algebra community) that we can represent and perform all the usual arithmetic operations on algebraic quantities as well. Furthermore, by appeal to a general result that goes back to Tarski, any problem that is algebraic [87] can be computed without errors. This general result forms the backdrop for the development of type I.

The problem with the above naive view of exact arithmetic is the inefficiency of algebraic computations. The complexity of each operation depends on the bit length of operands. In cascaded computations, the bit length of numbers increases quickly. Even for rational operations, the worst-case complexity is exponential in the number of operations. For an instance, Yu [90] concluded that exact rational arithmetic for 3-D polyhedral modeling is impracticable. Karasick et al [45] reported that the naive use of rational arithmetic in the divide-and-conquer algorithm for 2-D Delaunay triangulation costs a performance penalty of 10^4 over the corresponding floating-point implementation.

Approximate arithmetic. Approximate arithmetic is used by all Type I approaches, mostly in the form of machine arithmetic. But it is increasingly used in Type II approaches as well. An early example of Type II algorithm based on approximate arithmetic is [20] where it is applied to Fortune's sweepline algorithm using big floats, combined with root bounds. Approximate arithmetic for Type II algorithms requires multi-precision packages, typically some form of big float. This also opens up the possibility of adaptive precision arithmetic. Here, it is helpful to have arithmetic systems that is equipped with the ability to track errors, and to increase precision on demand [89]. Error tracking amounts to forward error analysis; it is also a form of significance arithmetic [2, 54]. More generally, significance arithmetic can be viewed as a form of interval arithmetic [58, 1].

It is important to see how approximate arithmetic solves the kind of efficiency bottleneck noted under the naive use of exact arithmetic. The latter ultimately reduces to big rational computation. Rational computations tend to be very slow for various reasons, including the phenomenon of exponential growth of bit sizes and the presence of non-trivial content in fractions (which requires the GCD computation to remove). On the other hand, big floats arithmetic has complexity that is intermediate between big rationals and big integers (we may take big integer arithmetic as the base reference for multi-precision arithmetic). In fact, big float arithmetic is basically big integer arithmetic, plus overhead. In addition to these, there is the possibility of adaptive precision computation.

Other ideas to improve robustness (without necessarily eliminating non-robustness) at the arithmetic level is to improve the accuracy or range of fixed precision arithmetic [17, 52]. More low level techniques here include providing accurate scalar products [61] and machine architectures that provide a "fused multiplication and add" (FMA) primitive. Programming language support for robustness can be valuable for programmers. For instance, special facilities for robust arithmetic has been incorporated into programming languages (e.g., Numerical Turing [43], Pascal-SC [7]).

2.2 Geometric Approaches

The geometric approach lends itself to considerably more diverse forms than the arithmetic approach. Hence our review here only touches on some major representatives. The general idea is to ensure that certain geometric properties are preserved by our algorithms. This is often enough to ensure no inconsistent states in the algorithm (hence robust). For instance, if we are computing the Voronoi diagram of a planar point set, we want to ensure that the output is a planar graph [77]. We stress that in our classification, purely topological approaches is subsumed under geometric approaches (see below for a more formal expression of this). The first decision facing the robust algorithm designer is the choice of which properties to preserve, and this is dictated by efficiency considerations and the needs of the application; as a corollary, the algorithm gives up some other properties.

Finite Resolution Geometry. If we compute in fixed precision arithmetic, then one approach is to invent novel "finite resolution geometries" [32] as a substitute for the standard Euclidean geometry. This ersatz geometry can only preserve a few of the properties found in Euclidean geometry. A very natural and popular finite precision geometry is the grid (usually regular, but this is not essential). Greene and Yao [32] investigated line arrangement computation in this geometry. See also [35, 33]. Here, line segment may become polygonal lines so that their intersections preserve properties such as non-braiding and connected intersection. But we give up the properties such as the intersection of two lines is a single point.

Approximate Predicates and Fat Geometry. Another approach to robustness is to focus on the imprecise nature of predicate evaluations. In Epsilon Geometry [34], Guibas et al introduced "epsilon-predicates" which return a real number instead of the usual $-1, 0, +1$. A return value of $\epsilon > 0$ means that there is a perturbation of the input by at most ϵ such that the exact predicate becomes true. If $\epsilon < 0$, this means that any perturbation of the input up to ϵ will still satisfy the exact predicate. Designing robust algorithms in this setting is difficult.

In terms of geometry, we can think of the imprecise predicates as inducing "fat objects" (so a point becomes a ball, and a line becomes a cylinder, etc). Indeed, this is basically what happens in the widespread programming trick called epsilon tweaking [87]. See also [71] for another example. Milenkovic [57] proposed to perturb objects so as to guarantee a minimum separation distance ("well separated"). This ensures that approximate predicates actually yield correct decisions.

Consistency and Topological Approach. The question "what is geometry?" has been asked many times throughout the history of mathematics. In geometric computing, the formula "Geometry = Combinatorics + Numerics" is particularly suggestive (combinatorics here can be identified with topology). For example, in representing the convex hull of a set of points, the combinatorics comprises the set of faces (of each dimension) and their incidence relations, and the numerics comprises the point sets determined by the faces, given in some suitable form (say the equations of affine subspace spanned by the faces). There is an implicit consistency relation that must hold between the combinatorics and the numerics. The **consistency approach** to robustness says that our main goal is to ensure that the combinatorics we compute must be part of some consistent geometric object. This amounts to saying that the decisions we make in evaluating predicates during a computation must never be inconsistent. Below we give a slightly more formal version of the consistency approach.

Fortune [24] points out that in principle it is possible to make many algorithms "parsimonious" (meaning that we only perform conditional tests which are independent of the results of previous tests). The basis for this observation is that, assuming all predicates are polynomial sign evaluations, the question of whether the result of current predicate evaluation is a logical consequence of previous evaluations can be phrased as a statement of the existential theory of the reals. This question is at least NP-hard but decidable in polynomial space [15]. Using such a decision procedure, we simply avoid all redundant predicate evaluations.

In practice, of course, we want more than just consistency. Otherwise, we can give the non-redundant predicates any answer we like! In any case, strictly parsimonious algorithms are infeasible. But we can often minimize the dependency between the combinatorial part and numerical part of an algorithm. This is the gist of the Topology-Oriented Approach advocated in a series of papers by Sugihara and Iri [79, 78, 75, 76, 74, 80]. In making decisions, the combinatorial part is given primacy over the numerical part. As a result, the combinatorial structures is guaranteed to be valid (in the sense of satisfying certain selected properties such as planarity). Somewhat similar ideas are advocated by Schorn [69], but phrased in terms of ensuring properties of primitive operations. For instance, these properties ensure topological properties.

For any particular problem, the topology-oriented approach leaves open as to which topological properties the algorithm designer should pick. A general and systematic theory is possible using the idea of "realizable" combinatorial structure. This is first advocated by Hoffmann, Hopcroft and Karasick [37]. Assume that the algorithm has to compute a geometric object D, which we view as combinatorial structure G together with associated numerical data λ: $D = (G, \lambda)$. Without much loss of generality, we can assume $G = (V, E)$ is a directed graph and $\lambda : V \cup E \to \mathbb{R}^n$ is a labeling function. Note that this is just a formalized of the "geometry=combinatorics+numerics" formula above; see [86] for further details. While λ may be regarded as arbitrarily specified, certain choices of λ are defined to be "consistent". We say G is **valid** if there is a λ which is consistent. In this setting, we can classify an algorithm A to be **geometrically exact** if for any input I, the output $A(I) = (G_I, \lambda_I)$ is such that G_I is the correct structure. For our current purposes, we will not require λ_I to be consistent. We say that A is **consistent** if G_I is valid. Realistically, we would like consistent algorithms to satisfy additional properties: for instance, G_I ought to be the correct structure for some small perturbation I' of I. Hoffmann et all [37] pointed out that designing consistent algorithms may lead to answering hard questions equivalent to theorem proving (akin to Fortune's parsimonious algorithm).

2.3 Exact Geometric Computation

We now describe an approach that, strictly speaking, ought to be classified under arithmetic approaches. In [87], we call this the **Exact Geometric Computation** (EGC for short) to emphasize that the "exactness" is in the geometry, not in the arithmetic. This is precisely what we defined as geometrically exact in the previous paragraph.

But if EGC is basically an arithmetic approach, how does it ensure anything about geometry? This is a simple but critical point, so we will elaborate on it. As above, assume that our algorithm A on input I computes a geometric object $D_I = (G_I, \lambda_I)$. We had already stated above that *combinatorics (i.e., G_I) is completely determined by the predicate evaluations in the algorithm $A(I)$*. To see this, view each computational step of A as either a **construction step** or a **conditional step**. The former computes new values, and the latter causes the program to branch. Different computational paths lead to a particular combinatorially outputs. Next, assume that each conditional step evaluates a real predicate and branches depending on the sign of the predicate evaluation $(-1, 0, +1)$. Thus to ensure exact combinatorics, it suffices to ensure that all predicates are evaluated correctly.

In short EGC simply amounts to ensuring that we never err in predicate evaluations. Two important computational consequences flow out of this simple assertion:

(1) *EGC is computationally feasible.* EGC represents a significant relaxation from the naive concept of numerical exactness. We only need to compute to sufficient precision to make the correct predicate evaluation. This has led to development of techniques such as precision-driven computation [89], lazy evaluation [4, 3], adaptive computation [73] and floating-point filters [26]. Thus the pessimistic bounds [90] of exact rational arithmetic are unnecessary. Let us expand on this remark, using the example of floating-point filters. Two preliminary remarks are in order: first, the gold standard of efficiency in scientific computation is machine floating-point arithmetic. Second, this standard works reasonably well for many applications, with nonrobustness arising relatively infrequently. Floating-point filters exploit these two remarks, and seeks to achieve robustness at the cost of having only a small constant factor overhead over the gold standard. Basically, one maintains some upper bound on the error in the machine floating-point arithmetic. We evaluate predicates using machine arithmetic, using the error bound to determine whether we can trust the result (this is the "filter test"). Only if the filter test fails, we fall back to some other expensive but foolproof evaluation of the same predicate. The cost of this filtered arithmetic has a fixed overhead and a variable overhead: the former is a small constant factor and the latter can be a large overhead. Under the assumption that the filter fails infrequently, the average cost of the variable overhead is also a small constant. In section 4, we treat filters in detail.

(2) *It is possible to create a software library whereby programmers can write robust programs just by calling the library to perform their arithmetic.* In other words, EGC can provide a general and purely arithmetic solution for a large class of problems. By "general solution"

we mean that the EGC solution need not[1] be problem-specific or algorithm-specific. This contrasts with the geometric approaches where solutions invented for one problem often do not extend to closely related problems. By "purely arithmetic solution" we mean that the user only has to use the right "number type" in their programs (in place of the standard number types such as `double` found in programming languages such as `C` or `Java`). The class of problems which is amenable to such a treatment are the algebraic problems [87]. In the following, we use the term **EGC library** to refer to any software library that supports for exact comparisons.

The algebraic problems constitute the overwhelming majority[2] of problems treated in contemporary computational geometry. Early studies in EGC assume that the number type is some form of big integer or big rational (e.g., [25]). This suffices for the class of bounded-depth rational problems [87], which already cover all the problems that other approaches have successfully treated. But for a EGC library to reach the algebraic class, we need an additional sophistication not found in big number packages, or computer algebra systems for that matter, that is the ability to guarantee exact comparisons. Such a capability[3] was first demonstrated in the `Real/Expr` Package [89]. What `Real/Expr` offers is the ability to guarantee any number of correct bits when computing a number. These bits can be either relative or absolute bits; in floating point number representation, relative (resp., absolute) bits means we count the bits from the most significant bit (resp., the radix point). This is generalization of exact comparisons, because the exact comparison of α and β amounts to computing $\alpha - \beta$ to guarantee 1 relative bit. The Core Library is a second generation version of `Real/Expr` with two visible changes: it adopts a standard programming semantics for assignments, and provides a natural and simple "numerical accuracy" model (API) [84].

Geometric exactness is a solution to the non-robustness problem because the root cause is numerical errors that lead to wrong predicate evaluations and ultimately inconsistent geometry. By definition, exact geometry is consistent. For many applications, consistent geometry is sufficient. This is especially true in situations where we know the input is approximate anyway. Why bother with *exact* geometry? The reason can be found in a previous remark that ensuring consistency may be as hard as geometry theorem proving [37]. In short, geometric exactness is often the best way to ensure consistency.

EGC is now in the midst of a development that seems quite remote 10 years ago: it is now possible for any programmer to routinely write completely robust and reasonably efficient geometric code for a large class of important problems. There are basically two[4] EGC libraries currently: LEDA [10, 40] and the Core Library [44, 39]. The CGAL library

[1] But see [20] for algorithm-specific or [9] for problem-specific EGC analysis.

[2] The problems found in standard references [65, 21, 60, 8, 59, 18, 30] are all algebraic. In fact it is not easy to find non-algebraic examples, but they arise in special kinds of Voronoi diagrams, in the form of the logarithmic spiral, and in non-holonomic motion planning.

[3] Although this was only implemented for $+, -, \times, \div$ and $\sqrt{\cdot}$, the basic technique extends to all algebraic computation. Moreover, the addition of $\sqrt{\cdot}$ captures the most important subclass of the algebraic problems beyond the rational problems.

[4] LEDA is a very large library that offers many services, including a large suite of data structures and algorithms. We are only referring to a specific facility here, namely the number type called `LEDA_real`. Core Library is based on the `Real/Expr` Package [89] which has the basic EGC capability but without the efficient techniques developed in recent years.

also provides robust algorithms based on EGC principles. The major challenge of EGC is efficiency, both in practical terms as well as theoretically.

At the practical level, users of EGC libraries often find surprising efficiency penalties for innocuous decisions. These typically involve the use of divisions and square-roots in expressions. For instance, in going around the vertices of a polygon $(P_0, P_1, \ldots, P_n)$ where $P_0 = P_n$, one might have a while loop conditioned on the test $P_i \neq P_0$ (for $i = 1, 2, \ldots n$) where P_0 is the first point of the polygon. In an actual example, where the P_i's are computed points on the unit 2-sphere (with coordinates that is a fraction with a square-root in the denominator), this slowed the computation to a halt. In this case, one should simply perform the check for $i = n$. Another example, if there are common subexpressions involving square-roots, the difference between writing code that shares these subexpressions and code that does not share can be huge. But this phenomenon should not surprise us – it happens in any high level programming language that is not implemented correctly with the right compiler technology. Thus, all modern compilers are knowledgeable about common subexpressions, loop constants, and a host of other tricks. There is no reason why similar technology should not be developed for EGC to take these concerns out of the minds of programmers. For that matter, using machine floating point arithmetic is not without its pitfalls for the unwary user. EGC arithmetic is no different.

An area of surprise for users of EGC is numerical input and output. Recall that in EGC, inputs must be assumed to be exact. But if we input a value like 1.23 in conventional languages, this will be converted to the closest machine representation. This can be surprising. In EGC, we represent numbers like $\sqrt{2}$ exactly, but during the computation, some approximation to $\sqrt{2}$ is computed. If we want to output this value to, say, 100 digits, we may not see[5] this many digits because the currently computed approximation has fewer than 100 correct digits. We refer to [85] for a discussion of many of these issues in the context of the Core Library.

The theoretical challenges of efficiency for EGC libraries relates to high degree algebraic computation. While EGC enables provable robust implementations of a wide range of geometric problems (basically subsuming those problems that other approaches can solve), high degree algebraic computations such as found in CAD applications remain a severe challenge. In this paper, we will address three important areas in which there is significant recent development: constructive root bounds, adaptive precision computation and floating point filters.

We note two important problem areas, one practical and the other theoretical, that are central to EGC. At present, we have little experience with EGC solutions as embedded in large software systems, such as in a CAD or mesh generation system. Here, we need to cascade the output of one algorithm into the input of another algorithm. The geometric output of an EGC algorithm may be in high precision, and we would like to reduce this precision in the cascade. The **geometric rounding problem** is this: given a consistent geometric object T in high precision, to round it to a consistent object T' at a lower precision. For instance, suppose T is a triangulation (in the plane or higher dimensions).

[5]Worst, if the EGC system is incorrectly implemented, we may see 100 digits though many of these may be wrong without the user realizing this.

Note that we do not assume that T' is combinatorially equivalent to T – only consistency is required. Otherwise, it is easy to run into NP-hardness, as seen in [56]. In particular, we expect topology to change in T' (e.g., if two close points in T may collapse to a single point in T'). In the robustness literature, "rounding problems" are often a composition of two problems: a construction problem followed by a bona-fide rounding problem. For instance, the problem of snap rounding [36] for intersecting line segments is such a composite problem. In EGC, we prefer to reduce such composite problems to two distinct steps: first compute some geometric object T, and then rounding T to a lower precision version T'. The first step is considered solved using EGC. The second step is what we call the rounding problem. See also [31].

The central theoretical problems relate to the **constant zero problem** for a set Ω of algebraic operators over an appropriate algebraic domain: given a (constant) expression E over Ω, is $E = 0$? The qualification "constant" means that Ω (or E) does not contain any variables. The open questions here relates to non-algebraic expressions. If Ω includes the rational operations and one transcendental function (such as $\sin(x)$ or $\log(x)$), is this problem decidable? The reason for our interest in non-algebraic functions is because, without such a decidability result, we do not even know whether there exists EGC solutions for non-algebraic problems.

3 Constructive Root Bounds

The fundamental decision problem in EGC computation is determining the sign of a constant algebraic expression. Comparison between two algebraic expressions is easily reduced to this problem. Although it is possible to solve this problem by purely algebraic and symbolic means, the current EGC libraries follow a numerical approach first used in `Real/Expr` based on root bounds. The numerical approach seems to be inherently more efficient and depends on the notion of a root bound. Although there are many forms of "root bounds", our definition below makes it plain that we are interested in bounding roots away from 0.

In this paper, an **expression** E refers to a syntactic object constructed from a given set Ω of operators over the reals $\mathbb{R}$. Each operator in Ω either has a fixed arity (which a natural number) or is "anadic" (taking any number of arguments). Let $\mathcal{E}(\Omega)$ denote the set of expressions over Ω. For instance, if $\Omega = \{0, 1, x, +, -, \times, \sqrt{\cdot}\}$, then $\mathcal{E}(\Omega)$ is the set of division-free radical expressions over the single variable x. For our purposes, Ω is not allowed to have variables; so each operator of zero arity denotes a real constant. Thus $E \in \mathcal{E}(\Omega)$ denotes a real value $\mathrm{val}(E)$, defined inductively in the natural way. Since some algebraic operators are partial, this value may be undefined (written $\mathrm{val}(E) =\uparrow$). Our expressions are basically straightline programs (cf. [83]), which are basically as rooted, labeled directed acyclic graph (DAG). Thus, sharing of subexpressions is allowed. We will need the sign function, $\mathrm{sign}(E) \in \{-1, 0, +1\}$. But since $\mathrm{val}(E)$ is partial, the sign function is also partial: $\mathrm{sign}(E) =\uparrow$ iff $\mathrm{val}(E) =\uparrow$. Below, we follow the usual abuse of notation by writing "E" instead of $\mathrm{val}(E)$.

Definition 1 *We call $b > 0$ a* **root bound** *for an expression E if the following holds: if*

$E \neq \uparrow$ and $E \neq 0$ then $|E| \geq b$. We also say $(-\log_2 b)$ is a **root bit-bound** *for E.*

To determine the sign of E from a root bound, we compute a numerical approximation $\widetilde{E}$ such that if $E = \uparrow$ then $E' = \uparrow$; otherwise, $|E - \widetilde{E}| < \frac{b}{2}$. Then

$$\text{sign}(E) = \begin{cases} \text{sign}(\widetilde{E}) & \text{if } |\widetilde{E}| \geq \frac{b}{2} \text{ or } \widetilde{E} = \uparrow \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Following [48], we classify the algebraic expressions into five categories as follows:

- $\Omega_0 = \{\pm, \times\} \cup \mathbb{Z}$ (where $\mathbb{Z}$ are the integers). Thus Ω_0-expressions are polynomials.
- $\Omega_1 = \Omega_0 \cup \{\div\}$. Thus Ω_1-expressions are rational expressions.
- $\Omega_2 = \Omega_1 \cup \{\sqrt[n]{\cdot} : n \geq 2\}$. Thus Ω_2-expressions are radical expressions.
- $\Omega_3 = \Omega_2 \cup \{\text{Root}(P, i) : P \in \mathbb{Z}[x]\}$. Here $\text{Root}(P, i)$ means the i-th real root of the polynomial P, which is assumed to be presented by its sequence of $n+1$ integer coefficients if $\deg(P) = n$.
- $\Omega_4 = \Omega_2 \cup \{\text{Root}(P, i) : P \in \Omega_4[x]\}$. Thus Ω_4 expressions include general algebraic expressions.

We can see from (1) that root bounds determine the worst-case complexity in exact sign determination. Thus, a main problem here is to find tight root bounds, and the efficient ways to compute them.

3.1 Review of Constructive Root Bounds

The problem of root bounds has been extensively studied (e.g., [51] or [55, chap. 2]). However, from an algorithmic point of view, many classical results are not constructive. Here the notion of "constructive" depends on the representation of expressions. Directed acyclic graphs (DAGs) are common forms of representation. In such cases, constructive bounds could mean those that can be computed inductively on the DAG structure. Thus, the constructive root bound problem is stated as: given a set $\mathcal{E}$ of expressions (e.g., the radical expressions), give a set of inductive rules for computing a root bound for each expression in $\mathcal{E}$. Typically, the inductive rules are based on some properties about the generation polynomials (of which the expression concerned is a root). A number of constructive root bounds have been proposed.

Table 1: Rules for degree-length and degree-height bounds

E	d	ℓ	h
rational $\frac{a}{b}$	1	$\sqrt{a^2+b^2}$	$\max\{\lvert a\rvert, \lvert b\rvert\}$
$E_1 \pm E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}2^{d_1 d_2+\min\{d_1,d_2\}}$	$(h_1 2^{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$E_1 \times E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}$	$(h_1\sqrt{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$E_1 \div E_2$	$d_1 d_2$	$\ell_1^{d_2}\ell_2^{d_1}$	$(h_1\sqrt{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}$
$\sqrt[k]{E_1}$	kd_1	ℓ_1	h_1

Canny's bound. Canny [14] shows that given a zero-dimensional system Σ of n polynomial equations with n unknowns, if $(\alpha_1, \ldots, \alpha_n)$ is a solution, then $|\alpha_i| \geq (3dc)^{-nd^n}$ for all non-zero component α_i. Here c (resp., d) is an upper bound on the absolute value of coefficients (resp., the degree) of any polynomial in the system. An important proviso in Canny's bound is that the homogenized system $\widehat{\Sigma}$ has a non-vanishing U-resultant. Equivalently, $\widehat{\Sigma}$ has finitely many roots at infinity. Yap [88, p. 350] gives the treatment for the general case, based on the notion of "generalized U-resultant". Such multivariate root bounds are easily translated into a bound on expressions, as discussed in [11].

Degree-length and degree-height bounds. The degree-length bound [88] is a bound for general algebraic expressions, based on Landau's root bound. For an expression E, the algorithm computes the upper bounds on the degree d and on length ($\|\cdot\|_2$) ℓ of the minimal polynomial of E. If $E \neq 0$, then from Landau's bound we know $|E| \geq \frac{1}{\ell}$. The extended Hadamard bound on polynomial matrix [29] is used to compute an upper bound of ℓ. A similar degree-height bound based on Cauchy's root bound is found in [89]. Here "length" and "height" refer to the 2-norm and ∞-norm of a polynomial, respectively. Both results are based on the resultant calculus. The bounds are maintained inductively on the structure of the expression DAG using the recursive rules found in Table 1, where the parameters d, ℓ, h denote the upper bounds for the degree, length and height, respectively.

Degree-measure bound. It is known that if $\alpha \neq 0$, we have

$$\frac{1}{m(\alpha)} \leq |\alpha| \leq m(\alpha). \tag{2}$$

Here, the measure $m(\alpha)$ is defined as $|a_m| \cdot \prod_{i=1}^{m} \max\{1, |\alpha_i|\}$, where a_m is the leading coefficient of α's minimal polynomial, and α_i's are the conjugates of α.

Let α and β be two nonzero algebraic numbers of degrees m and n respectively. The following relations on measures are given in [55],

$$\begin{aligned} m(\alpha \pm \beta) &\leq 2^{mn} m(\alpha)^n m(\beta)^m \\ m(\alpha \times \beta) &\leq m(\alpha)^n m(\beta)^m \\ m(\alpha \div \beta) &\leq m(\alpha)^n m(\beta)^m \\ m(\alpha^{1/k}) &\leq m(\alpha) \end{aligned}$$

Table 2: BFMS Rules

	E	$u(E)$	$l(E)$
1.	integer a	$\lvert a \rvert$	1
2.	$E_1 \pm E_2$	$u(E_1)l(E_2) + l(E_1)u(E_2)$	$l(E_1)l(E_2)$
3.	$E_1 \times E_2$	$u(E_1)u(E_2)$	$l(E_1)l(E_2)$
4.	$E_1 \div E_2$	$u(E_1)l(E_2)$	$l(E_1)u(E_2)$
5.	$\sqrt[k]{E_1}$	$\sqrt[k]{u(E_1)}$	$\sqrt[k]{l(E_1)}$

$$m(\alpha^k) \quad \leq \quad m(\alpha)^k$$

Based on Mignotte's work, Burnikel et al [11] develop recursive rules to maintain the upper bounds for degrees and measures of radical expressions and call it the *degree-measure bound*. The degree-measure bound turns out to be always better than the degree-length bound [11]. Improvements over the original degree-measure bound are reported in [48, 72].

BFMS bound. One of the best constructive root bounds for the class of radical expressions is from Burnikel et al [11] (hereafter called the "BFMS bound"). For division-free expressions, it is an improvement over previously known bounds and is essentially tight. But in presence of divisions, the BFMS bound is not necessarily an improvement of the degree-measure bound. Conceptually the BFMS approach first transforms a radical expression E to a quotient of two division-free expressions $U(E)$ and $L(E)$. Two parameters $u(E)$ and $l(E)$, the upper bounds on the conjugates of $U(E)$ and $L(E)$, respectively, are maintained by the recursively rules in Table 2. Clearly, if E is division-free, then $L(E) = 1$ and $\mathrm{val}(E)$ is an algebraic integer (i.e., a root of some monic integer polynomial).

For an expression E having r radical nodes with indices $k_1, k_2, \ldots, k_r$, the BFMS bound is given by

$$\text{(3)} \qquad \mathrm{val}(E) \neq 0 \Rightarrow (u(E)^{D(E)^2-1} l(E))^{-1} \leq |\mathrm{val}(E)| \leq u(E) l(E)^{D(E)^2-1},$$

where $D(E) = \prod_{i=1}^{r} k_i$, and $u(E)$ and $l(E)$ are (respectively) upper bounds on the absolute values of algebraic conjugates of $\mathrm{val}(U(E))$ and $\mathrm{val}(L(E))$.

For division-free expressions, the BFMS bound improves to

$$\text{(4)} \qquad \mathrm{val}(E) \neq 0 \Rightarrow |\mathrm{val}(E)| \geq (u(E)^{D(E)-1})^{-1}.$$

Improved and generalized BFMS bound. Note that the root bit-bound in (3) is quadratic in $D(E)$, while in (4), it is linear in $D(E)$. This quadratic factor can be a serious efficiency issue. Consider a simple example: $E = (\sqrt{x} + \sqrt{y}) - \sqrt{x + y + 2\sqrt{xy}}$ where x, y are L-bit integers. Of course, this expression is identically 0 for any x, y. The BFMS bound yields a root bit-bound of $7.5L + \mathcal{O}(1)$ bits. But in case, x and y are viewed as rational numbers (with denominator 1), the bit-bound becomes $127.5L + \mathcal{O}(1)$. This example shows

Table 3: Scheinerman's Rules

	E	$n(E)$	$b(E)$
1.	integer a	1	$\|a\|$
2.	$\sqrt{cd}$	2	$\max\{\|c\|, \|d\|\}$
3.	$E_1 \pm E_2$	$n_1 n_2$	$b_1 + b_2$
4.	$E_1 \times E_2$	$n_1 n_2$	$b_1 b_2$
5.	$\sqrt[k]{E_1}$	kn_1	b_1
6.	$P(E_1)$	n_1	$\overline{P}(n_1 b_1)$

that introducing rational numbers at the leaves of expressions has a major impact on the original BFMS bound.

Most recently, Mehlhorn et al [53] extend the BFMS to support general algebraic expressions (the Ω_4 expressions). Their new root bit-bound depends on $D(E)$ linearly for most algebraic expressions. The significant improvement is obtained from a trick to avoid the doubling of the degree in transforming E into a division of two algebraic integer expressions. For radical expressions, it can be shown that this new bound is always an improvement over the original one, and depends on $D(E)$ linearly.

Scheinerman bound. This bound adopts an interesting approach based on matrix eigenvalues [67]. Let $\Lambda(n, b)$ denote the set of eigenvalues of $n \times n$ matrices with integer entries with absolute value at most b. It is easy to see that $\Lambda(n, b)$ is a finite set of algebraic integers. Moreover, if $\alpha \in \Lambda(n, b)$ is non-zero then $|\alpha| \geq (nb)^{1-n}$. Scheinerman gives a constructive root bound for division-free radical expressions E by maintaining two parameters, $n(E)$ and $b(E)$, satisfying the property that the value of E is in $\Lambda(n(E), b(E))$. These recursive rules are given by Table 3.

Note that the rule for $\sqrt{cd}$ is rather special, but it can be extremely useful. In Rule 6, the polynomial $\overline{P}(x)$ is given by $\sum_{i=0}^{d} |a_i| x^i$ when $P(x) = \sum_{i=0}^{d} a_i x^i$. This rule is not explicitly stated in [67], but can be deduced from an example he gave. An example given in [67] is to test whether $\alpha = \sqrt{2} + \sqrt{5 - 2\sqrt{6}} - \sqrt{3}$ is zero. Scheinerman's bound requires calculating α to 39 digits while the BFMS bound says 12 digits are enough.

Li-Yap bound. In [48], we give a new constructive root bound that is applicable to a general class of algebraic expressions (Ω_3). Our basic idea is to bound the leading and tail coefficients, and the conjugates of the algebraic expression with the help of resultant calculus. The new bound gives significantly better performance in many important computations involving divisions and root extractions. For any algebraic number α, we will exploit the following relation:

$$\alpha \neq 0 \Rightarrow |\alpha| \geq (\mu(\alpha)^{\deg(\alpha)-1} \mathrm{lead}(\alpha))^{-1}, \tag{5}$$

where $\mu(\alpha) = \max\{|\xi| \ : \ \xi \text{ is a conjugate of } \alpha\}$, $\deg(\alpha)$ is the degree of the minimal polynomial $\mathrm{Irr}(\alpha)$ of α and $\mathrm{lead}(\alpha)$ is the leading coefficient of $\mathrm{Irr}(\alpha)$. A similar relation was also used in [11] for bounds on algebraic integers. Our bound requires the computation of three upper bounds

$$D(E), \quad \mathrm{lc}E, \quad \overline{\mu}(E)$$

on the corresponding parameters $\deg(E), \mathrm{lead}(E)$ and $\mu(E)$. Suppose that E has k radical nodes or root-of-polynomial nodes $\{r_1, r_2, \ldots, r_k\}$. We choose $D(E) = \prod_{i=1}^k k_i$ where k_i is either the index of r_i if r_i is a radical node, or the degree of the polynomial if r_i is a polynomial-root node. Due to the admission of division, we also need to maintain upper bounds $\mathrm{tc}(E)$, $M(E)$ on $\mathrm{tail}(E)$ and $m(E)$ in bounding $\mathrm{lead}(E)$. Here the tail coefficient $\mathrm{tail}(()E)$ is defined as the constant term of $\mathrm{Irr}(E)$.

Table 4 gives the recursive rules to maintain $\mathrm{lc}(E), \mathrm{tc}(E)$ and $M(E)$.

Table 4: Recursive rules for $\mathrm{lc}(E)$ (and associated $\mathrm{tc}(E)$ and $M(E)$)

	E	$\mathrm{lc}(E)$	$\mathrm{tc}(E)$	$M(E)$
1.	rational $\frac{a}{b}$	$\lvert b\rvert$	$\lvert a\rvert$	$\max\{\lvert a\rvert, \lvert b\rvert\}$
2.	$\mathrm{Root}(P)$	$\lvert\mathrm{lead}(P)\rvert$	$\lvert\mathrm{tail}(P)\rvert$	$\lVert P\rVert_2$
3.	$E_1 \pm E_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}2^{D(E)}$	$M_1^{D_2}M_2^{D_1}2^{D(E)}$
4.	$E_1 \times E_2$	$\mathrm{lc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}$
5.	$E_1 \div E_2$	$\mathrm{lc}_1^{D_2}\mathrm{tc}_2^{D_1}$	$\mathrm{tc}_1^{D_2}\mathrm{lc}_2^{D_1}$	$M_1^{D_2}M_2^{D_1}$
6.	$\sqrt[k]{E_1}$	lc_1	tc_1	M_1
7.	E_1^k	lc_1^k	tc_1^k	M_1^k

The upper bounds on conjugates, $\overline{\mu}(E)$, are obtained through resultant calculus and standard interval arithmetic techniques. It turns out that it is necessary to maintain a lower bound $\underline{\nu}(E)$ on the conjugates at the same time. The recursive rules to maintain these two bounds are given in Table 5.

Table 5: Recursive rules for bounds on conjugates

	E	$\overline{\mu}(E)$	$\underline{\nu}(E)$
1.	rational $\frac{a}{b}$	$\lvert\frac{a}{b}\rvert$	$\lvert\frac{a}{b}\rvert$
2.	$\mathrm{Root}(P)$	$1 + \lVert P\rVert_\infty$	$(1 + \lVert P\rVert_\infty)^{-1}$
3.	$E_1 \pm E_2$	$\overline{\mu}(E_1) + \overline{\mu}(E_2)$	$\max\{M(E)^{-1}, (\overline{\mu}(E)^{D(E)-1}\mathrm{lc}(E))^{-1}\}$
4.	$E_1 \times E_2$	$\overline{\mu}(E_1)\overline{\mu}(E_2)$	$\underline{\nu}(E_1)\underline{\nu}(E_2)$
5.	$E_1 \div E_2$	$\overline{\mu}(E_1)/\underline{\nu}(E_2)$	$\underline{\nu}(E_1)/\overline{\mu}(E_2)$
6.	$\sqrt[k]{E_1}$	$\sqrt[k]{\overline{\mu}(E_1)}$	$\sqrt[k]{\underline{\nu}(E_1)}$
7.	E_1^k	$\overline{\mu}(E_1)^k$	$\underline{\nu}(E_1)^k$

Finally, we obtain the new root bound as follows: Given an Ω_3-expression E, if $E \neq 0$, then

$$|E| \geq (\overline{\mu}(E)^{(D(E)-1)}\mathrm{lc}(E))^{-1}. \tag{6}$$

We implemented the new bound in our Core Library and experiments show that it can achieve remarkable speedup over previous bounds in the presence of division [48]. Although we have described our bounds for the class of Ω_3-expressions, it should be clear that our method extends to more general expressions.

3.2 Comparisons of Constructive Root Bounds

Comparisons between various constructive root bound can be found in [11, 48]. In general, a direct comparison of the above root bounds is a difficult task because of the different choice of parameters and bounding functions used. Therefore, following the tact in [48], we compare their performance on various special subclasses of algebraic expressions.

1. For division-free radical expressions, the BFMS bound is never worse than all the other bounds. Moreover, for this special class of expressions, Li-Yap bound is identical to the BFMS bound.

2. For general algebraic expressions, in terms of root bit-bound, Li-Yap bound is at most $D \cdot M$ where D is the degree bound, and M is the root bit-bound from the degree-measure bound.

3. Considering the sum of square roots of rational numbers (a common problem in solving the shortest Euclidean path problem), it can be shown that each of Li-Yap bound and the degree-measure bound can be better than the other depending on different parameters about the expressions. But both of them are always better than the BFMS bound.

4. Given a radical expression E with rational values at the leaves, if E has no divisions and shared radical nodes, Li-Yap bound for E is never worse than the BFMS bound, and can be better in many cases.

5. A critical test in Fortune's sweepline algorithm is to determine the sign of the expression $E = \frac{a+\sqrt{b}}{d} - \frac{a'+\sqrt{b'}}{d'}$ where a's, b's and d's are $3L$-, $6L$- and $2L$-bit integers, respectively. The BFMS bound requires $(79L + 30)$ bits and the degree-measure (D-M) bound needs $(64L + 12)$ bits. Li-Yap root bit-bound improves them to $(19L + 9)$ bits. We generate some random inputs with different L values which always make $E = 0$, and put the timings (in seconds) of the tests in Table 6. The experiments are performed on a Sun UltraSPARC with a 440 MHz CPU and 512MB main memory.

Table 6: Timings for Fortune's expression

L	10	20	50	100	200
NEW	0.01	0.03	0.12	0.69	3.90
BFMS	0.03	0.24	1.63	11.69	79.43
D-M	0.03	0.22	1.62	10.99	84.54

3.3 Treatment of Special Cases

Root separation bounds. In both the Core Library and LEDA, the comparison of two expressions α and β is obtained by computing the root bound of $\alpha - \beta$. However, we can use root separation bounds [88] for any polynomial P that has α and β as roots. If $P(X) \in \mathbb{C}[X]$ is a non-zero polynomial polynomial, $\mathrm{sep}(P)$ denotes the minimum $|\alpha_i - \alpha_j|$ where $\alpha_i \neq \alpha_j$

range over all pairs of complex roots of P. When P has less than two distinct roots, define $\mathrm{sep}(P) = \infty$. The following general bound of Rump [66] (as rectified by Schwartz [70]) is

$$\mathrm{sep}(P) > \left[2 \cdot m^{m/2+2}(\|P\|_\infty + 1)^m\right]^{-1}$$

where m is the degree of P. Suppose $A(X)$ and $B(X)$ are the minimal polynomials for α and β, then $|\alpha - \beta| \geq \mathrm{sep}(AB)$. Therefore, if we maintain upper bounds d, d' on the degrees of A and B, and upper bounds h, h' on the heights of A and B, we obtain

$$|\alpha - \beta| \geq \left[hh'(1+n)\right]^{-2n+1}(2n)^{-n-1} \tag{7}$$

where $n = \max\{d, d'\}$ (see [88, p.173]). The advantage of using equation (7) is that the root bit bound does not have a dd' term, as would be the case if we use resultant calculus. Note that using this bound does not nicely fit into our recursive root bound framework (in particular, it does not generate bounds for a new minimal polynomial). On the other hand, it is unnecessary to maintain parameters for the minimal polynomial of $\alpha - \beta$ since this is not an actual expression that the user constructed.

Zero test. Zero testing is the special case of sign determination in which we want to know whether an expression is zero or not. Many predicates in computational geometry programs are really zero tests (e.g. detection of degeneracy, checking if a point lies on a hyperplane). In other applications, even though we need general sign determination, the zero outcome is very common. For instance, in the application of EGC to theorem proving [83], true conjectures are equivalent to the zero outcome. In our numerical approach based on root bounds, the complexity of sign determination is determined by the root bound when the outcome is zero. Since root bounds can be overly pessimistic, such tests can be extremely slow. Hence it is desirable to have an independent method of testing if an expression is zero. Such a zero test can be used as a filter for the sign determination algorithm. Only when the filter detects a non-zero do we call the root-bound based numerical method.

Yap and Blömer [6] noted that for the case of sum of square roots of integers, zero testing is deterministic polynomial time while the sign determination problem is not known to be polynomial time. Blömer [5] gave a probabilistic algorithm for un-nested real radical expressions. When the radicals are nested, we can apply denesting algorithms [42, 46]. Note that these methods are non-numerical methods.

4 Precision-Driven Computation

The idea of adaptive precision computation has many manifestations in EGC. Here, we want to describe the approach called **precision-driven computation** [89]. Here, we wish to approximate the value of an expression E to some user-specified precision p. We will propagate the precision p to all the nodes in E using suitable inductive rules. In the simplest case, this propagation may proceed only in one direction from the root down to the leaves. Then, we evaluate the approximate values at the leaves and recursively apply the operations at each node from the bottom up. In our implementations, the approximate values at each node is a big float number. Complications arise the propagation of precision requires bounds on the magnitude of the values at some nodes. This is where the root bounds from the previous section become essential.

In the literature, a technique called "lazy evaluation" (e.g, [4]) has superficial similarities to our method. However, lazy evaluation typically "pumps" increasingly precise values from the leaves to the root, and simply tracks the forward error. If this error is larger than the desired precision at the root, the process is iterated. Note that this is insufficient to guarantee the sign of an expression (in particular, it cannot detect a zero value in a finite number of iterations).

In the following, we will describe some improved methods for precision-driven evaluation of radical expressions. It is assumed that the leaves of these expressions are rational constants.

4.1 Bounds on the Magnitude of Expressions

In `Real/Expr` and Core Library, the evaluation begins by computing bounds on the absolute value of each node (see [62]). We now review this computation. For any expression E, we define $\mathrm{MSB}(E)$ to be $\lfloor \lg(|E|) \rfloor$. Intuitively, MSB of E tells us about the location of the most significant bit of E. By definition, the $\mathrm{MSB}(0) = -\infty$. In practice, we only want to compute a bounding interval $\left[\mu_E^-, \mu_E^+\right]$ that contains $\mathrm{MSB}(E)$. The rules in Table 7 is used to maintain this interval.

Table 7: Rules for upper and lower bounds on $\mathrm{MSB}(E)$

E	μ_E^+	μ_E^-
rational $\frac{a}{b}$	$\lceil \lg(\frac{a}{b}) \rceil$	$\lfloor \lg(\frac{a}{b}) \rfloor$
$E_1 \pm E_2$	$\max\{\mu_{E_1}^+, \mu_{E_2}^+\} + 1$	$\lfloor \lg(\lvert E \rvert) \rfloor$
$E_1 \times E_2$	$\mu_{E_1}^+ + \mu_{E_2}^+$	$\mu_{E_1}^- + \mu_{E_2}^-$
$E_1 \div E_2$	$\mu_{E_1}^+ - \mu_{E_2}^-$	$\mu_{E_1}^- - \mu_{E_2}^+$
$\sqrt[k]{E_1}$	$\left\lceil \mu_{E_1}^+ / k \right\rceil$	$\left\lfloor \mu_{E_2}^- / k \right\rfloor$

The main subtlety in this table is the entry for μ_E^- when $E = E_1 \pm E_2$. We call this the **special entry** of this table because, due to potential cancellation, we cannot derive a lower bound on $\mathrm{MSB}(E)$ in terms of the bounds on E_1 and E_2 only. There are three possible ways to derive this entry. First, we could use the negative of the root bit bound (see Definition 1). However, root bounds are generally not tight. Hence, the second method is to approximate E numerically to obtain its most significant bit, or to reach the root bound in case $E = 0$. Thus, this method really determines the true value of $\mathrm{MSB}(E)$, and provides the entry shown in the above table. A third method is more efficient, but is applicable only under certain conditions: either when E_1 and E_2 have the same sign, or their magnitudes are quite different (say, by looking at their MSB's). In this case, we can deduce the μ_E^- from the bounds on E's children. For instance, if $\mu_{E_1}^- > \mu_{E_2}^+ + 1$ then $\mu_{E_1}^- - 1$ is a lower bound for μ_E^-. Note that $|\mu_E^+ - \mu_E^-| \leq 2^{m+1}$ where the number of operations in E.

4.2 Guaranteed Precision Approximation

We use a notion of numerical precision from [89] which combines both absolute and relative precision in one framework: given a real number X, and extended reals $a, r \in \mathbb{R} \cup \pm\infty$, we say that a real number $\widetilde{X}$ is an *approximation* of another real X to *(composite) precision* $[r, a]$, denoted $\widetilde{X} \simeq X\,[r, a]$, provided either

$$\left|\widetilde{X} - X\right| \leq 2^{-r}\,|X| \qquad or \qquad \left|\widetilde{X} - X\right| \leq 2^{-a}.$$

When $a = +\infty$ (resp., $r = +\infty$), then $[r, a]$ is simply a relative (resp., absolute) precision bound. In practice, r and a will be integers.

An algorithm to propagate composite precision bounds is described in [63, chapter 7]. Here, we describe a simpler method. First we translate the composite bound $[r, a]$ to an **absolute error bound** α_E such that if $|X - \widetilde{X}| \leq \alpha_E$ then $\widetilde{X} \simeq X\,[r, a]$. It is not hard to see that it is sufficient to set

$$\alpha_E = \max\{2^{-r+\mu_E^-}, 2^{-a}\}.$$

The rules to propagate α_E to the children of E are presented in the second column of Table 8. If E_1, E_2 are the children of E, we want to define α_{E_i} $(i = 1, 2)$ in such a way that if $\widetilde{E_i}$ satisfies $|E_i - \widetilde{E_i}| < \alpha_{E_i}$ then $\widetilde{E}$ satisfies $|E - \widetilde{E}| < \alpha_E$. For simplicity, we write α_{E_i} for α_i.

Here, $\widetilde{E}$ is an approximation obtained by applying the operator at E to $\widetilde{E_1}, \widetilde{E_2}$. The rule for computing this approximation is given in column 3 of Table 8. A notation used in column 3 is that, for any real X and $\alpha > 0$, $(X)_\alpha$ refers to any approximation $\widetilde{X}$ for X that satisfies the bound $|X - \widetilde{X}| < \alpha$.

In summary, column 2 tells us how to propagate absolute precision bounds downward towards the leaves, and column 3 tells us how to compute approximations from the leaves upward to the root. At the leaves, we assume an ability to generate numerical approximations to the within the desired error bounds. At each node F, our rules guarantee that the approximate value at F satisfies the absolute precision bound α_F.

Note that for the addition, subtraction and multiplication operations, the computation of $\widetilde{E}$ can be performed exactly (as in [47]). But the present rules no longer require exact computation. The new rule is clearly never worse than the old rule (at the cost of at most one extra bit), but is sensitive to the actual precision needed. In fact, for all operations, we now allow an absolute error of $\frac{\alpha_E}{2}$. We omit the justification of Table 8, since it is rather similar to the proofs in [47].

Actually, column 2 is not exactly the rule one uses in implementation. We give formulas for α_i $(i = 1, 2)$ in terms of $|E_1|$ and $|E_2|$ to make the formulas easier to understand. But it is generally not possible nor desirable to compute $|E_i|$ exactly. Instead, we compute lower bounds on the logarithms of the expressions given for α_1 and α_2, respectively. Whenever $|E_i|$ appears in an expression, we replace it by $2^{\mu_{E_i}^-}$ (if $|E_i|$ is in the numerator) or by $2^{\mu_{E_i}^+}$ otherwise. After taking logs, we are left with $\mu_{E_i}^-$ or $\mu_{E_i}^+$.

Table 8: Rules for (1) Propagating absolute precision α_E and (2) Approximation $\widetilde{E}$

E	Downward Rules	Upward Rules
$E_1 \pm E_2$	$\alpha_1 = \alpha_2 = \frac{1}{4}\alpha_E$	$\widetilde{E} = (\widetilde{E_1} \pm \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \times E_2$	Let $c = \left\lfloor \sqrt{\alpha_E/(2\lvert E_1\rvert \cdot \lvert E_2\rvert) + 1} \right\rfloor - 1$, and $\alpha_1 = c \cdot \lvert E_2\rvert$, $\alpha_2 = c \cdot \lvert E_1\rvert$	$\widetilde{E} = (\widetilde{E_1} \times \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$E_1 \div E_2$	$\alpha_1 = \alpha_E\lvert E_2\rvert/4$, $\alpha_2 = \frac{\alpha_E\lvert E_2\rvert^2}{4(\lvert E_1\rvert+\alpha_1)+\alpha_E\lvert E_2\rvert}$	$\widetilde{E} = (\widetilde{E_1} \oslash \widetilde{E_2})_{\frac{\alpha_E}{2}}$
$\sqrt[k]{E_1}$	$\alpha_1 = \alpha_E \sqrt[k]{E_1}^{k-1}/2$	$\widetilde{E} = (\widetilde{\sqrt[k]{\widetilde{E_1}}})_{\frac{\alpha_E}{2}}$

4.3 The MSB and Approximation Algorithms

There are two important algorithms which we derive from the above tables: one is APPROX(E, α_E) which computes an approximation of E to within α_E. The other is ComputeMSB(E) which computes upper and lower bounds for MSB(F) for every node F in the expression E. This basically follows the Table 7. It should be noted that APPROX and ComputeMSB are mutually recursive algorithms: this is because ComputeMSB will need to call APPROX to compute the special entry (for $\mu^-_{E_1 \pm E_2}$) in Table 7. Clearly, APPROX needs ComputeMSB at various places as shown in Table 8. It is not hard to verify that this mutual recursion will not lead to infinite loops since the underlying graph of E is a DAG.

Two additional remarks should be made: (i) This ComputeMSB Algorithm computes enough information to deduce the exact sign at every sub node of E, assuming the signs are available at the leaves. (ii) Because of (i), root bounds of addition/subtraction nodes may be needed. These root bounds are separately computed (say, when E is constructed).

Our original Core Library basically uses the above approach. By looking at Table 8, however, we see that some of the MSB bounds computed by ComputeMSB are unnecessary:

- Addition and subtraction $E = E_1 \pm E_2$. No MSB bounds on E_1, E_2 are needed to propagate precision bounds.
- Multiplication $E = E_1 \times E_2$. All the MSB bounds on E_1, E_2 are needed.
- Division $E = E_1 \div E_2$. Only the bounds $\mu^+_{E_1}$, $\mu^+_{E_2}$ and $\mu^-_{E_2}$ are needed.
- Root extraction $E = \sqrt[k]{E_1}$. Only $\mu^-_{E_1}$ is needed.

We now develop a more sophisticated ComputeMSB algorithm to compute only the required upper and lower MSB bounds. Let the refined algorithm take two additional arguments: ComputeMSB$(E, \mathrm{uf}, \mathrm{lf})$ where uf, lf are Boolean flags. The refined algorithm does not automatically compute signs of every sub node; however, when the sign is available from the known data, the signs will be set. The uf (resp., lf) flags says that an upper (resp., lower) bound on MSB(E) is needed. We simply present a self-explanatory algorithm here in a C++-like syntax:

```
Algorithm
  ComputeMSB(E, uf, lf) {

                // E: an expression node;
                // uf: a flag to force the computation of E.umsb;
                // lf: a flag to force the computation of E.lmsb.

            switch (E.operation_type) {
                case 'constant':
                    if (uf) E.umsb = ceilLog(E.value));
                                   // ceilLog is ceiling of log_2
                    if (lf) E.lmsb = floorLog(E.value));
                                   // floorLog is floor of log_2
                    break;
                case '+' or '-':
                    if (uf and (not lf))
                        ComputeMSB(E.first, true, false);
                        ComputeMSB(E.first, true, false);
                        E.umsb = max{E.first.umsb, E.second.umsb} + 1;
                    if (lf) {
                       ComputeMSB(E.first, true, true);
                       ComputeMSB(E.second, true, true);
                       if (E.first and E.second has same sign) {
                          E.umsb = max{E.first.umsb, E.second.umsb} + 1;
                          E.lmsb = min{E.first.lmsb, E.second.lmsb};
                       } else {
                          APPROX(E, E.root_bound);
                          E.lmsb = floor(lg(E.value));
                          E.umsb =  ceil(lg(E.value));
                       }
                    }
                    break;
                case '*':
                    MSB(E.first, true, true);
                    MSB(E.second, true, true);
                    E.umsb = E.first.umsb + E.second.umsb;
                    E.lmsb = E.first.lmsb + E.second.lmsb;
                    break;
                case '/':
                    MSB(E.first, true, lf);
                    MSB(E.second, true, true);
                    if (uf) E.umsb = E.first.umsb - E.second.lmsb;
                    if (lf) E.lmsb = E.first.lmsb - E.second.umsb;
```

```
                break;
            case 'root extraction':
                ComputeMSB(E.first, uf, lf)
                if (uf) E.umsb = E.first.umsb / k;
                if (lf) E.lmsb = E.first.lmsb / k;
                break;
        } //switch
    } //ComputeMSB
```

5 Numerical Filters and Certification

In the EGC techniques of the previous sections, the use of multi-precision arithmetic is essential. Another avenue to gain efficiency is to exploit machine floating-point arithmetic which is fast and highly optimized on current hardware. The basic idea is simple: we must "check" or "certify" the output of machine evaluation of predicates.

Model for checking and filtering. We give a simple formal model for checking and filtering (certifying). Assume I and O are some sets called the **input and output spaces**. In numerical problems, it is often possible to identify I and O with $\mathbb{R}^n$ for some n. A **computational problem** is simply a subset $\pi \subseteq I \times O$. A **program** is simply a partial function $P : I \to O$. So $P(x)$ may be undefined for some x, denoted $P(x) \uparrow$; if $P(x)$ is defined, then we write $P(x) \downarrow$. We say P is a **partial algorithm** for π if for all $x \in I$, if $P(x) \downarrow$ then $(x, P(x)) \in \pi$. An **algorithm** A for π is a partial algorithm that happens to be a total function. A **filter** for π is a total program $F : I \times O \to \{0, 1\}$ such that $F(x, y) = 1$ implies $(x, y) \in \pi$. A **checker** C for π is a filter for π such that if $C(x, y) = 0$ then $(x, y) \notin \pi$. Thus, a checker is a filter, but not necessarily vice-versa. Finally, a **filtered program** for π is a pair (P, F) such that P is a total program and F is a filter for π. We view (P, F) to be a new program P_F, such that on input x, if $F(x, P(x)) = 1$ then $P_F(x) = P(x)$; otherwise $P_F(x) \uparrow$. Thus P_F is a partial algorithm for π. We want to "anchor" a filtered program (P, F) for π with an algorithm A for π. The triple (P, F, A) would constitute a program $P_{F,A}$ for π: on any input x, we define $P_{F,A}(x) = P_F(x)$ if $P_F(x) \downarrow$, otherwise $P_{F,A}(x) = A(x)$.

Logically, the algorithm (P, F, A) is no different from A alone. It is the complexity considerations that motivates (P, F, A). For, if $C_A(x)$ denotes the complexity of the algorithm A on input x, we have

$$C_{(P,F,A)}(x) = C_P(x) + C_F(x, P(x)) + \delta$$

where

$$\delta = \begin{cases} 0 & \text{if } \ F(x, P(x)) = 1, \\ C_A(x) & \text{else.} \end{cases}$$

The algorithm (P, F, A) may be more efficient than A especially if $C_A(x)$ is expensive, and the filter is so efficacious that most of the time, $\delta = 0$.

We consider the filtered program (P, F) for π because P is presumably some useful program for π. We can consider filter cascades: let F be a sequence

$$F = (F_1, \ldots, F_n) \tag{8}$$

where each F_i is a filter for π. We call F a (n-level) **filter cascade** for π. We view F as a filter for π by defining $F(x, y) = \max_{i=1}^{n} F_i(x, y)$. Thus $F(x, y) = 1$ iff some $F_i(x, y) = 1$. In practice, we evaluate $F(x, y)$ by searching for the first i such that $F_i(x, y) = 1$; and otherwise output 0. Funke, et al [27, 13] are among the first to exploit multi-level filter cascades. A filter cascade (8) in practice would have the property that each F_i is more "effective" but less efficient than F_{i-1}.

In EGC, the filters are usually **numerical filters**. These filters certify some property of a computed numerical value, typically its sign. This often amounts to computing some error bound, and comparing the computed value with this bound. When such filters aim to certifying machine floating-point arithmetic, we call them **floating point filters**. Ultimately, filters are used for efficiency purposes in EGC, not for correctness purposes.

There are two main classifications of numerical filters: static or dynamic. Static filters are those that can be computed at compile time, and they incur a low overhead at runtime. However, static bounds may be overly pessimistic and thus less effective. Dynamic filters exhibit opposite characteristics: they have higher runtime cost but are much more effective (i.e., fewer false rejections). We can have semi-static filters which combine both features.

Filters can be used at different levels of granularity: from individual machine operations (e.g., arithmetic operations), to subroutines (e.g., geometric primitives) to algorithms (e.g. convex hull or meshing algorithm) to entire systems.

Computing upper bounds in machine arithmetic. In the implementation of numerical filters, we need to compute sharp upper bounds on numerical expressions. To be specific, suppose you have IEEE double values x and y and you want to compute an upper bound on $|z|$ where $z = xy$. How can you do this? We can compute

$$\widetilde{z} \leftarrow |x \odot y|. \tag{9}$$

Here, $|\cdot|$ is done exactly by the IEEE arithmetic, but the multiplication $\odot$ is not exact. One aspect of IEEE arithmetic is that you can change the rounding modes [82]. So if you have changed the rounding mode to round away from 0 or round towards $+\infty$, we will have $\widetilde{z} \geq xy$. Otherwise, we only know that $\widetilde{z} = z(1 + \delta)$ where $|\delta| \leq \mathbf{u}$. Here $\mathbf{u} = 2^{-53}$ is the "unit of rounding" for the arithmetic. Unfortunately, turning the rounding mode on or off seems like an expensive proposition, not meant to be casually used in the middle of a program. So instead of relying on rounding modes, we further compute $\widetilde{w}$ as follows:

$$\widetilde{w} \leftarrow \widetilde{z} \odot (1 + 4\mathbf{u}). \tag{10}$$

It is assumed that overflow and underflow do not occur during the computation of $\widetilde{w}$.

Note that $1 + 4\mathbf{u} = 1 + 2^{-51}$ is exactly representable. Therefore, we know that $\widetilde{w} = \widetilde{z}(1 + 4\mathbf{u})(1 + \delta')$ for some δ satisfying $|\delta'| \leq \mathbf{u}$. Hence,

$$\begin{aligned}
\widetilde{w} &= z(1+\delta)(1+\delta')(1+4\mathbf{u}) \\
&\geq z(1-2\mathbf{u}+\mathbf{u}^2)(1+4\mathbf{u}) \\
&= z(1+2\mathbf{u}-7\mathbf{u}^2+4\mathbf{u}^3) \\
&> z
\end{aligned}$$

Note that if any of the operations $\oplus$, $\ominus$ or $\oslash$ is used in place of $\odot$ in (9), the same argument still shows that $\widetilde{w}$ is an upper bound on the actual value.

We summarize this result:

Lemma 1 *Let E be any rational numerical expression and let $\widetilde{E}$ be IEEE double value approximation to E. Assume the input numbers in E are IEEE doubles and E has $k \geq 1$ operations.*
(i) We can compute a IEEE double `MaxAbs`(E) *which satisfies $|E| \leq maxabs(E)$ in $3k$ machine operations.*
(ii) If all the input values are positive, $2k$ machine operations suffices.
(iii) The value $\widetilde{E}$ is available as a side effect of computing `MaxAbs`(E)*, at the cost of storing the result.*

Proof. In proof, we simply replace each rational operation in E by at most 3 machine operations: we count 2 flops to compute $\widetilde{z}$ in equations (9), and 1 flop to compute $\widetilde{w}$ in (10). In case the input numbers are non-negative, $\widetilde{z}$ needs only 1 machine operation. **Q.E.D.**

5.1 Static Filters

Fortune and Van Wyk [26] were the first to implement and quantify the efficacy of filters for exact geometric computation. Their filter was implemented via the `LN` preprocessor system. We now look at the simple filter they implemented (which we dub the "FvW Filter"), and some of their experimental results.

The FvW filter. Static error bounds are easily maintained for a polynomial expression E with integer values at the leaves. Let $\widetilde{E}$ denote the IEEE double value obtained by direct evaluation of E using IEEE double operations. Fortune and Van Wyk compute a bound `MaxErr`(E) on the absolute error,

$$|E - \widetilde{E}| \leq \texttt{MaxErr}(E). \tag{11}$$

It is easy to use this bound as a filter to certify the sign of $\widetilde{E}$: if $|\widetilde{E}| > \texttt{MaxErr}(E)$ then $\text{sign}(\widetilde{E}) = \text{sign}(E)$. Otherwise, we must resort to some fall back action. Unless we specify

Table 9: Parameters for the FwW filter

Expr E	$\texttt{MaxLen}(E)$	$\texttt{MaxErr}(E)$
Var x	$\texttt{MaxLen}(x)$ given	$\max\{0, 2^{\texttt{MaxLen}(E)-53}\}$
$F \pm G$	$1 + \max\{\texttt{MaxLen}(F), \texttt{MaxLen}(G)\}$	$\texttt{MaxErr}(F) + \texttt{MaxErr}(G) + 2^{\texttt{MaxLen}(F \pm G)-53}$
FG	$\texttt{MaxLen}(F) + \texttt{MaxLen}(G)$	$\texttt{MaxErr}(F)^{\texttt{MaxLen}(G)} + \texttt{MaxErr}(G)^{\texttt{MaxLen}(F)}$ $+2^{\texttt{MaxLen}(FG)-53}$

otherwise, the default action is to immediately use an infallible method, namely computing exactly using a Big Number package.

Let us now see how to compute $\texttt{MaxErr}(E)$. It turns out that we also need the magnitude of E. The base-2 logarithm of the magnitude is bounded by $\texttt{MaxLen}(E)$. Thus, we say that the FvW filter has two **filter parameters**,

$$\texttt{MaxErr}(E), \quad \texttt{MaxLen}(E). \tag{12}$$

We assume that each input variable x is assigned an upper bound $\texttt{MaxLen}(x)$ on its bit length. Inductively, if F and G are polynomial expressions, then $\texttt{MaxLen}(E)$ and $\texttt{MaxErr}(E)$ are defined using the rules in Table 9.

Observe that the formulas in Table 9 assumes exact arithmetic. In implementations, we have to be careful to compute upper bounds on these formulas. We assume that the filter has failed in case of an overflow; it is easy to see that no underflow occurs when evaluating these formulas. Checking for exceptions has an extra overhead. Since $\texttt{MaxLen}(E)$ is an integer, we can evaluate the corresponding formulas using IEEE arithmetic exactly. But the formulas for $\texttt{MaxErr}(E)$ will incur error, and we need to use some form of lemma 1.

Framework for measuring filter efficacy. We want to quantify the efficacy of the FvW Filter. Consider the primitive of determining the sign of a 4×4 integer determinant. First look at the unfiltered performance of this primitive. We use the IEEE machine double arithmetic evaluation of this determinant (with possibly incorrect sign) as the **base line** for speed; this is standard procedure. This base performance is then compared to the performance of some standard (off-the-shelf) Big Integer packages. This serves as the **top line** for speed. The numbers cited in the paper are for the Big Integer package in `LEDA` (circa 1995), but the general conclusion for other packages are apparently not much different. For random 31-bit integers, the top line time yields 60 time increase over the base line. We will say

$$\sigma = 60 \tag{13}$$

in this case; the symbol σ (or $\sigma(31)$) reminds us that this is the "slowdown" factor. Clearly, σ is a function of the bit length L as well. For instance, with random 53-bit signed integers, the factor σ becomes 100. Next, still with $L = 31$, but using static filters implemented in `LN`, the factor σ ranges from 13.7 to 21.8, for various platforms and CPU speeds [26,

Figure 14]. For simplicity, we simply say $\sigma = 20$, for some mythical combination of these platforms and CPUs. Thus the static filters improve the performance of exact arithmetic by the factor

$$\phi = 60/20 = 3. \tag{14}$$

In general, using unfiltered exact integer arithmetic as base line, the symbol ϕ denotes the "filtered improvement". We use it as a measure of the efficacy of filtering.

In summary, the above experimental framework is clearly quite general, and aims to reduce the efficacy of a particular filter by estimating a number ϕ. In general, the framework requires the following choices: (1) a "test algorithm" (we picked one for 4×4 determinants), (2) the "base line" (the standard is IEEE double arithmetic), (3) the "top line" (we picked `LEDA`'s Big Integer), (4) the input data (we consider random 31-bit integers). Its simplicity means that we can extract this ϕ factor from almost any[6] published experimental papers. Another measure of efficacy is the fraction ρ of correct approximate values $\widetilde{E}$ which fails to pass the filter. In [19], a general technique for assessing the efficacy of an arithmetic filter is proposed based on an analysis which consists of evaluating both the threshold value and the probability of failure of the filter.

For a true complexity model, we need to introduce size parameters. In EGC, two size parameters are of interest: the combinatorial size n and the bit size L. Hence all these parameters ought to be be written as $\sigma(n, L)$, $\phi(n, L)$, etc.

Realistic versus synthetic problems. Static filters has an efficacy factor $\phi = 3$ (see (14)) in evaluating the sign of randomly generated 4-dimensional matrices ($L = 31$). The paper [13] calls such problems are "synthetic benchmarks". So it would be interesting to see the performance of filters using "realistic benchmarks", meaning actual algorithms for natural problems that we want to solve. But even here, there are degrees of realism. Let us[7] equate realistic benchmarks with algorithms for problems such as convex hulls, triangulations, etc. We look at one of these benchmarks from Sugihara below.

The point of realistic benchmarks is that they will generally involve a significant amount of non-numeric computation. Hence the ϕ-factor in such settings may not look as good as our synthetic ones. To quantify this, suppose that a fraction

$$\beta \quad (0 \le \beta \le 1) \tag{15}$$

of the running time of the algorithm is attributable to numerical computation. After replacing the machine arithmetic with exact integer arithmetic, the overall time becomes $(1-\beta) + \beta\sigma = 1 + (\sigma - 1)\beta$. With filtered arithmetic, the time becomes $1 + (\sigma - 1)\beta\phi^{-1}$. Thus efficacy factor ϕ' for the algorithm is really

$$\phi' := \frac{(1 + (\sigma - 1)\beta}{1 + (\sigma - 1)\beta/\phi}.$$

It is easy to verify that $\phi' < \phi$ (assuming $\phi > 1$).

[6]Alternatively, we could require these numbers as the minimum standard for any experimental paper on filters.

[7]While holding on tightly to our theoretician's hat!

Note that our derivation assumes the original time is unit! This normalization is valid in our derivation because all the factors σ, ϕ that we use are ratios and are not affected by the normalization.

The factor β is empirical, of course. But even so, how can we estimate this? For instance, for 2- and 3-dimensional Delaunay triangulations, Fortune and van Wyk [26] noted that $\beta \in [0.2, 0.5]$. Burnikel, Funke and Schirra [13] suggests a very simple idea for obtaining β: simply execute the test program in which each arithmetic operation is repeated $c > 1$ times. This gives us a new timing for the test program,

$$T(c) = (1 - \beta) + c\beta.$$

Now, by plotting the running time $T(c)$ against c, we obtain β as the slope.

5.2 Dynamic Filters

To improve the quality of the static filters, we can use runtime information about the actual values of the variables, and dynamically compute the error bounds. We can again use $\mathtt{MaxErr}(E)$ and $\mathtt{MaxLen}(E)$ as found in Table 9 for static error. The only difference lies in the base case: for each variable x, the $\mathtt{MaxErr}(x)$ and $\mathtt{MaxLen}(x)$ can be directly computed from the value of x.

Dynamic version of the FvW filter. It is easy enough to convert the FwW filter into a dynamic one: looking at Table 9, we see that the only modification is that in the base case, we can directly compute $\mathtt{MaxLen}(x)$ and $\mathtt{MaxErr}(x)$ for a variable x. Let us estimate the cost of this dynamic filter. We already note that $\mathtt{MaxLen}(E)$ can be computed directly using the formula in Table 9 since they involve integer arithmetic. This takes 1 or 2 operations. But for $\mathtt{MaxErr}(E)$, we need to appeal to lemma 1. It is easy to see that since the values are all non-negative, we at most double the operation counts in the formulas of Table 9. The worst case is the formula for $E = FG$:

$$\mathtt{MaxErr}(E) = \mathtt{MaxErr}(F)^{\mathtt{MaxLen}(G)} + \mathtt{MaxErr}(G)^{\mathtt{MaxLen}(F)} + 2^{\mathtt{MaxLen}(FG)-53}.$$

The value $2^{\mathtt{MaxLen}(FG)-53}$ can be computed exactly in 2 flops. There remain 4 other exact operations, which requires $2 \times 4 = 8$ flops. Hence the bound here is 10 flops. Added to the single operation to compute $\mathtt{MaxLen}(F) + \mathtt{MaxLen}(G)$, we obtain 11 flops. A similar analysis for $E = F + G$ yields 8 flops.

After computing these filter parameters, we need check if the filter predicate is satisfied:

$$|\widetilde{E}| > \mathtt{MaxErr}(E).$$

Assuming $\widetilde{E}$ is available (this may incur storage costs not counted above), this check requires up to 2 operations: to compute the absolute value of $\widetilde{E}$ and to perform the comparison. Alternatively, if $\widetilde{E}$ is not available, we can replace $\widetilde{E}$ by $2^{\mathtt{MaxLen}(E)}$. In general, we also need to check for floating point exceptions at the end of computing the filter parameters (the filter is assumed to have failed when an exception occurred). We may be able to avoid the exception handling, e.g., static analysis may tell us that no exceptions can occur.

Fortune and Van Wyk gave somewhat similar numbers, which we quote: let f_e count the extra runtime operations, including comparisons; let f_r count the runtime operations for storing intermediate results. In the `LN` implementation, $12 \le f_e \le 14$ and $24 \le f_r \le 33$. The $36 \le f_e + f_r \le 47$ overhead is needed even when the filter successfully certifies the approximate result $\widetilde{E}$; otherwise, we may have to add the cost of exact computation, etc. Hence $f_e + f_r$ is a lower bound on the σ-factor when using filtered arithmetic in `LN`. Roughly the same bound of $\sigma = 48$ was measured for `LEDA`'s system.

Other ideas can be brought into play: we need not immediately invoke the dynamic filter. We still maintain a static filter, and do the more expensive runtime filter only when the static filter fails. And when the dynamic filter fails, we may still resort to other less expensive computation instead of jumping immediately to some failsafe but expensive big Integer/Rational computation. The layering of these stages of computations is called **cascaded filtering** by Burnikel, Funke and Schirra (BFS) [12]. This technique seems to pay off especially in degenerate situations. We next describe the BFS filter.

The BFS filter. The is a dynamic filter, but it can also be described as "semi-static" (or "semi-dynamic") because one of its two computed parameters is statically determined. Let E be a radical expression, *i.e.*, involving $+, -, \times, \div, \sqrt{\cdot}$. Again, let $\widetilde{E}$ be the machine IEEE double value computed from E in the straightforward manner (this time, with division and square-roots). In contrast to the FvW Filter, the filter parameters are now

$$\texttt{MaxAbs}(E), \quad \texttt{Ind}(E).$$

The first is easy to understand: $\texttt{MaxAbs}(E)$ is simply an upper bound on $|E|$. The second, called the **index** of E, is a natural number whose interpretation is harder to characterize. Together, they satisfy the following invariant:

$$|E - \widetilde{E}| \le \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{16}$$

Of course, the value 2^{-53} may be replaced by the unit roundoff error $\mathbf{u}$ in general for other floating point systems.

Table 10 gives the recursive rules for maintaining $\texttt{MaxAbs}(E)$ and $\texttt{Ind}(E)$. The base case (E is a variable) is covered by the first two rows: notice that they distinguish between exact and rounded input variables. A variable x is **exact** if its value is representable without error by an IEEE double. In any case, x is assumed not to lie in the overflow range, so that the following holds

$$|\texttt{round}(x) - x| \le |x| 2^{-53}.$$

Also note that the bounds are computed using IEEE machine arithmetic, denoted

$$\oplus, \quad \ominus, \quad \odot, \quad \oslash, \quad \widetilde{\sqrt{\cdot}}.$$

The question arises: what happens when the operations lead to over- or underflow in computing the bound parameters? It can be shown that underflows for $\oplus$, $\ominus$ and $\widetilde{\sqrt{\cdot}}$ can be

Table 10: Parameters of the BFS filter

Expression E	$\texttt{MaxAbs}(E)$	$\texttt{Ind}(E)$
Exact variable x	x	0
Approx. variable x	$\texttt{round}(x)$	1
$E = F \pm G$	$\texttt{MaxAbs}(F) \oplus \texttt{MaxAbs}(G)$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G)\}$
$E = FG$	$\texttt{MaxAbs}(F) \odot \texttt{MaxAbs}(G)$	$1 + \texttt{Ind}(F) + \texttt{Ind}(G)$
$E = F/G$	$\frac{\lvert\widetilde{E}\rvert \oplus (\texttt{MaxAbs}(F) \oslash \texttt{MaxAbs}(G))}{(\lvert\widetilde{G}\rvert \oslash \texttt{MaxAbs}(G)) \ominus (\texttt{Ind}(G)+1)2^{-53}}$	$1 + \max\{\texttt{Ind}(F), \texttt{Ind}(G) + 1\}$
$E = \sqrt{F}$	$\begin{cases} (\texttt{MaxAbs}(F) \oslash \widetilde{F}) \odot \widetilde{E} & \text{if } \widetilde{F} > 0 \\ \sqrt{\texttt{MaxAbs}(F)} \odot 2^{26} & \text{if } \widetilde{F} = 0 \end{cases} =$	$1 + \texttt{Ind}(F)$

ignored, and in the case of $\odot$ and $\oslash$, we just have to add a small constant $MinDbl = 10^{-1022}$ to $\texttt{MaxAbs}(E)$.

Under the assumption (16), we can use the following criteria for certifying the sign of $\widetilde{E}$:

$$|\widetilde{E}| > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-53} \tag{17}$$

Of course, this criteria should be implemented using machine arithmetic (see (10) and notes there). One can even certify the exactness of $\widetilde{E}$ under certain conditions. If E is a polynomial expression (*i.e.*, involving $+, -, \times$ only), then $E = \widetilde{E}$ provided

$$1 > \texttt{MaxAbs}(E) \cdot \texttt{Ind}(E) \cdot 2^{-52}. \tag{18}$$

Finally, we look at some experimental results. Table 11 shows the σ-factor (recall that this is a slowdown factor compared to IEEE machine arithmetic) for the unfiltered and filtered cases. In both cases, the underlying Big Integer package is from LEDA's. The last column adds compilation to the filtered case. It is based on an expression compiler, EXPCOMP, somewhat in the spirit of LN. At $L = 32$, the ϕ-factor (recall this is the speedup due to filtering) is $65.7/2.9 = 22.7$. When compilation is used, it improves to $\phi = 65.7/1.9 = 34.6$. [Note: the reader might be tempted to deduce from these numbers that the the BFS filter is efficacious than the FwW Filter. But the use of different Big Integer packages, platforms and compilers, etc, does not justify this conclusion.]

Table 11: Random 3×3 determinant

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	42.8	2.9	1.9
16	46.2	2.9	1.9
32	65.7	2.9	1.9
40	123.3	2.9	1.8
48	125.1	2.9	1.8

While the above results look good, it is possible to create situations that filters are ineffective. Instead of using matrices with randomly generated integer entries, we can use degenerate determinants as input. The results recorded in Table 12 indicate that filters have very little effect. Indeed, we might have expected it to slow down the computation, since the filtering effort are strictly extra overhead. In contrast, for random data, the filter is almost always effective in avoiding exact computation.

Table 12: Degenerate 3×3 determinants

BitLength L	Unfiltered σ	BFS Filter σ	BFS Compiled σ
8	37.9	2.4	1.4
16	45.3	2.4	1.4
32	56.3	56.5	58.4
40	117.4	119.4	117.5
48	135.2	136.5	135.1

The original paper [12] describes more experimental results, including the performance of the BFS filter in the context of algorithms for computing Voronoi diagrams and triangulation of simple polygons.

6 Conclusions

Among existing approaches to nonrobustness, Exact Geometric Computation shows its distinctive advantages in its generality and ease-of-use. The presence of the major software libraries such as LEDA and CGAL based on EGC, testifies to its utility. We discussed some key efficiency issues. The sign determination problem and the role of constructive root bounds is basic. When carefully combined with techniques such as precision-driven computation and filtering, the efficiency of EGC can be greatly improved.

Through EGC, numerical non-robustness for a large class of geometric computation (low dimensions and low degree problems) has been brought out of the realm of "unsolved" to the realm of "solved but practically challenging". For these problems, in addition to the above topics, we believe that compiler-like techniques will help to us to solve these problems routinely in the future. Filter technology itself can be developed into a profoundly interesting subject on its own right, including making connections to program checking.

For high degree problems, efficiency will be a serious challenge. But this seems inherent rather than a failing of the EGC approach. Here we must rely on progress in root bounds methods, problem specific techniques, and perhaps completely new ideas. For non-algebraic problems, the fundamental question whether they even admit EGC solutions is completely open and relates to some deep questions in mathematics.

Reliable geometric computation can find direct applications in many areas, including geometric modeling, CAD and robotics. EGC has so far been applied at the level of stand-alone algorithms. Applying EGC in a real and complex application like CAD system would be a challenge. Then the problem of geometric rounding should be addressed in greater earnest.

Another real world issue which we have ignored is how to treat inexact inputs. For simple applications, we can treat them as "nominally exact". But if this can lead to inconsistent inputs, we basically need to treat more complex classes of geometric objects (e.g., we now think of points as balls) and new semantics. But research on this topic has not yet begun. We have begun to explore the notion of robustness as a computational resource [41]. That is, we no longer view robustness as a 0-1 proposition. Instead, we want to compute at various points of some implicit speed-robustness tradeoff curve. In some sense, this is simply acknowledging the lay man's view of nonrobustness. The difficulty lies in building a model for this.

In conclusion, we have reasons to be optimistic that EGC concepts will become more and more a part of the computing landscape. The inexorable logic of Moore's law implies that more and more applications will no longer be speed-critical, and thus able to use the robust, if slower, EGC solutions.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computation.* Academic Press, New York, 1983.

[2] R. L. Ashenhurst and N. Metropolis. Error estimates in computer calculation. *Amer. Math. Monthly*, 72(2):47–58, 1965.

[3] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy arithmetic library. In *Proceedings of the IEEE 11th Symposium on Computer Arithmetic*, pages 242–269, Windsor, Ontario, June 30-July 2, 1993.

[4] M. Benouamer, D. Michelucci, and B. Péroche. Boundary evaluation using a lazy rational arithmetic. In *Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 115–126, Montréal, Canada, 1993. ACM Press.

[5] J. Blömer. A probabilistic zero-test for expressions involving roots of rational numbers. *Proc. of the Sixth Annual European Symposium on Algorithms*, pages 151–162, 1998. LNCS 1461.

[6] J. Blömer. *Simplifying Expressions Involving Radicals.* PhD thesis, Free University Berlin, Department of Mathematics, October, 1992.

[7] G. Bohlender, C. Ullrich, J. W. von Gudenberg, and L. B. Rall. *Pascal-SC*, volume 17 of *Perspectives in Computing.* Academic Press, Boston-San Diego-New York, 1990.

[8] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, 1997. Translated by Hervé Brönnimann.

[9] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* Ph.D thesis, Universität des Saarlandes, March 1996.

[10] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, 1999.

[11] 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.

[12] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. *Proceedings of the 14th Annual Symposium on Computational Geometry*, pages 175–183, 1998.

[13] S. Burnikel, Funke. Exact geometric computation using cascading. *to appear in special issue of IJCGA*, 2000.

[14] J. F. Canny. *The complexity of robot motion planning.* ACM Doctoral Dissertation Award Series. The MIT Press, 1988. PhD thesis, M.I.T.

[15] J. F. Canny. Some algebraic and geometric configurations in PSPACE. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 460–467, 1988.

[16] B. Chazelle et al. Application challenges to computational geometry. In *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 407–463. AMS, 1999. The Computational Geometry Impact Task Force Report (1996).

[17] C. Clenshaw, F. Olver, and P. Turner. Level-index arithmetic: an introductory survey. In P. Turner, editor, *Numerical Analysis and Parallel Processing*, pages 95–168. Springer-Verlag, 1987. Lecture Notes in Mathematics, No.1397.

[18] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[19] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.

[20] T. Dubé and C. K. Yap. A basis for implementing exact geometric algorithms (extended abstract), September, 1993. Paper from URL `http://cs.nyu.edu/cs/faculty/yap`.

[21] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[22] A. R. Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In D. F. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, 1987.

[23] S. Fortune, editor. *Special Issue on Implementation of Geometric Algorithms*, volume 27:1 of *Algorithmica*. Springer-Verlag, 2000.

[24] S. J. Fortune. Stable maintenance of point-set triangulations in two dimensions. *IEEE Foundations of Computer Science*, 30:494–499, 1989.

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

[26] S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[27] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Max Planck Institute for Computer Science, Saarbrücken, Germany, 1997.

[28] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[29] A. J. Goldstein and R. L. Graham. A Hadamard-type bound on the coefficients of a determinant of polynomials. *SIAM Review*, 16:394–395, 1974.

[30] J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, 1997.

[31] M. Goodrich, L. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th ACM Symp. on Computational Geom.*, pages 284–293, 1997.

[32] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. *IEEE Foundations of Computer Science*, 27:143–152, 1986.

[33] L. Guibas and D. Marimont. Rounding arrangements dynamically. In *Proc. 11th ACM Symp. Computational Geom.*, pages 190–199, 1995.

[34] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. *ACM Symp. on Computational Geometry*, 5:208–217, 1989.

[35] J. Hobby. Practical segment intersection with finite precision output. Technical report, Bell Labs, 1993. Tech. Report.

[36] J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13(4):199–214, Oct. 1999.

[37] C. Hoffmann, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. *ACM Symp. on Computational Geometry*, 4:106–117, 1988.

[38] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3), March 1989.

[39] The CORE Project Homepage. URL `http://www.cs.nyu.edu/exact/`.

[40] The LEDA Homepage. URL `http://www.mpi-sb.mpg.de/LEDA/`.

[41] Exact Geometric Computation homepage, 1996. FAQs, projects, documentation, and related links available from the URL `http://cs.nyu.edu/exact/`.

[42] G. Horng and M. D. Huang. Simplifying nested radicals and solving polynomials by radicals in minimum depth. *Proc. 31st Symp. on Foundations of Computer Science*, pages 847–854, 1990.

[43] T. Hull and M. Cohen. Toward an ideal computer arithmetic. In *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 5–48. IEEE, 1987.

[44] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core Library for robust numeric and geometric computation. In *Proceedings of the Fifteenth ACM Symposium on Computational Geometry (SoCG 1999)*, pages 351–359, June 1999.

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

[46] S. Landau. Simplification of nested radicals. *SIAM Journal of Computing*, 21(1):85–110, 1992.

[47] C. Li. *Exact Geometric Computation: Theory and Applications*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, Jan. 2001. URL: http://www.cs.nyu.edu/csweb/Research/theses.html.

[48] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *Proceedings of the Twelfth ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 496–505, Jan. 2001.

[49] M. C. Lin and D. Manocha, editors. *Proceedings of the First ACM Workshop on Applied Computational Geometry*, 1996.

[50] G. Liotta, F. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. *ACM Symp. on Computational Geometry*, 13:156–165, 1997.

[51] M. Marden. *The geometry of the zeros of a polynomial in a complex variable.* American Mathematical Society, 1949.

[52] S. Matsui and M. Iri. An overflow/underflow-free floating-point representation of numbers. *J. Inform. Process*, 4(3):123–133, 1981.

[53] K. Mehlhorn and S. Schirra. A generalized and improved constructive separation bound for real algebraic expressions. Technical Report MPI-I-2000-004, Max-Planck-Institut für Informatik, Nov. 2000.

[54] N. Metropolis. Methods of significance arithmetic. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 179–192. Academic Press, London, 1977.

[55] M. Mignotte and D. Ştefănescu. *Polynomials: An Algorithmic Approach.* Springer, 1999.

[56] V. Milenkovic and L. Nackman. Finding compact coordinate representations for polygons and polyhedra. *ACM Symp. on Computational Geometry*, 6:244–252, 1990.

[57] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[58] R. E. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966.

[59] K. Mulmuley. *Computational Geometry: an Introduction through Randomized Algorithms.* Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1994.

[60] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[61] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. 3rd ACM Sympos. Comput. Geom.*, pages 119–125, 1987.

[62] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[63] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, New York, January 1997.

[64] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modellers, 1994 progress report. In *Proc. 1995 NSF Design and Manufacturing Grantees Conference*, pages 139–140, 1995.

[65] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[66] S. M. Rump. Polynomial minimum root separation. *Math. Comp.*, 33:327–336, 1979.

[67] E. R. Scheinerman. When close enough is close enough. *Amer. Math. Monthly*, 107:489–499, 2000.

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

[69] P. Schorn. An axiomatic approach to robust geometric programs. *J. of Symbolic Computation*, 16:155–165, 1993.

[70] J. T. Schwartz. Polynomial minimum root separation (Note to a paper of S. M. Rump). Technical Report 39, Courant Institute of Mathematical Sciences, Robotics Laboratory, New York University, Feb. 1985.

[71] M. G. Segal and C. H. Sequin. Consistent calculations for solids modelling. In *Proc. 1st ACM Sympos. Comput. Geom.*, pages 29–38, 1985.

[72] H. Sekigawa. Using interval computation with the Mahler measure for zero determination of algebraic numbers. *Josai Information Sciences Researches*, 9(1):83–99, 1998.

[73] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th ACM Symp. on Computational Geom.*, pages 141–150, May 1996.

[74] K. Sugihara. An intersection algorithm based on Delaunay triangulation. *IEEE Computer Graphics Appl.*, 12(2):59–67, 1992.

[75] K. Sugihara and M. Iri. A solid modeling system free from topological inconsistency. *J.Information Processing, Information Processing Society of Japan*, 12(4):380–393, 1989.

[76] K. Sugihara and M. Iri. Two design principles of geometric algorithms in finite precision arithmetic. *Applied Mathematics Letters*, 2:203–206, 1989.

[77] K. Sugihara and M. Iri. Construction of the Voronoi diagram for 'one million' generat ors in single-precision arithmetic. *Proc. IEEE*, 80(9):1471–1484, Sept. 1992.

[78] K. Sugihara and M. Iri. An approach to the problem of numerical errors in geometric algorithms. *Proc., 37th Convention of the Information Processing Society of Japan, Kyoto*, pages 1665–1666, September 12–14, 1988.

[79] K. Sugihara and M. Iri. Geometric algorithms in finite-precision arithmetic. Research Memorandum RMI 88-10, Dept. of Math. Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, September, 1988. 13th International Symposium on Mathematical Programming, Tokyo, Aug 29–Sep 2, 1988.

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

[81] R. Tamassia et al. Strategic directions in computational geometry. *ACM Computing Surveys*, 28(4), Dec. 1996.

[82] The Institute of Electrical and Electronic Engineers, Inc. IEEE Standard 754-1985 for binary floating-point arithmetic, 1985. ANSI/IEEE Std 754-1985. Reprinted in SIGPLAN 22(2) pp. 9-25.

[83] D. Tulone, C. Yap, and C. Li. Randomized zero testing of radical expressions and elementary geometry theorem proving. In *Proceedings of the Third International Workshop on Automated Deduction in Geometry (ADG 2000)*, pages 121–136, Sept. 2000. Final version to appear in LNAI series.

[84] C. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998. Invited Talk. Brown University, Oct 11–12, 1998. Abstracts, `http://www.cs.brown.edu/cgc/cgc98/home.html`.

[85] C. Yap and C. Li. Core Library Tutorial: a library for robust geometric computation.

[86] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, 1997.

[87] C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997. Invited talk, Proceed. 5th Canadian Conference on Comp. Geometry, Waterloo, Aug 5–9, 1993.

[88] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford Univ. Press, Dec. 1999.

[89] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, 1995. 2nd edition.

[90] J. Yu. *Exact arithmetic solid modeling.* Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992. Technical Report No. CSD-TR-92-037.